Université de Paris 8 - Vincennes Saint-Denis
UFR 6 - Langage, Informatique, Technologie (LIT) -
Ecole doctorale « Enjeux Sociaux et Technologies de la Communication »

Thèse de doctorat

De Sciences de l'Information et de la Communication

Présentée par :
Frédéric Drouillon

Sujet :

Chimères et gargouilles informatiques

Sous la direction de
Jean-Marc LEPERS

Thèse soutenue au mois d'octobre 2003

J'exprime mes remerciements au Professeur Jean-Marc Lepers qui a accueilli avec intérêt et générosité la démarche et a permis l'aboutissement de ce travail,

à Stéphane Rideau, sans lequel j'aurais eu de gros de soucis pour apprendre et comprendre le langage C,

à Pierre Audibert, dont le contenu des cours est d'une richesse sans fin,

à Patricia Philippot qui a rendu possible l'intégration de concepts issus de la psychanalyse,

à Francis Rousseaux, toujours fidèle aux rendez-vous importants et dont les conseils sont précieux,

à Sylvie Gueugnon pour sa confiance immense et également pour sa perspicacité en matière de langue française.

Je dédie cette thèse à mes enfants,

Mais aussi, bien entendu, à toutes les personnes auxquelles ce travail pourra profiter de quelque manière que ce soit.

Résumé

Ecrivain programmeur et conteur… Réunir art et science dans un langage commun, la programmation, crée une situation propice à de nombreuses expérimentations. Chimères et gargouilles informatiques sont autant de créations et d'élucubrations étranges. Elles prennent corps dans des réalisations de logiciels comme autant de prétextes aux partages et rencontres avec le public.
L'objet de cette recherche est de fondre art et informatique dans une pratique commune à travers la programmation, et d'expérimenter, selon une sensibilité de culture artistique contemporaine, des possibilités expressives spécifiques à l'écriture informatique.

Art, science, jeu ou bidouille ? La chimère informatique se conçoit à travers un corpus personnalisé qui dérive de ces différents domaines d'investigations. Elle touche un public par ce qui dans le programme est non calculable et existe en tant que phénomène. Des programmes réalisés pendant la recherche sont présentés selon quatre approches : systèmes scénaristiques, systèmes génératifs, mise en scène de graphisme et référence à la culture du jeu.

Dans notre démarche de création, un graphique, si simple soit-il, est d'abord une petite machine. Tracer une ligne ce n'est pas tracer une ligne, c'est construire un robot, le programme, qui trace la ligne. Ce que nous appelons un « robographe ». Ni robot ni graphe, le robographe dépasse les perspectives de l'image. C'est une troisième voie et la perspective est celle de la découverte d'un espace, l'espace robographique. Cet espace se constitue de figures robotiques telles qu'elles peuvent coïncider avec l'imaginaire du concepteur. Et l'imaginaire trouve dans cette coïncidence les moyens de s'exprimer, à savoir de bâtir une écriture. Cette écriture qui relie imagination et technique de programmation ouvre le champ perceptif spécifique de cette pratique. Et sur la base des nouvelles impressions possibles, le regard est pour ainsi dire sensibilisé à un nouveau monde. Il s'agit d'un impressionnisme technologique.

L'espace robographique est alors visité de l'intérieur par la pratique. Outre ce qui a trait à la matière informationnelle elle-même en dur dans la machine et à sa manipulation par la programmation, nous abordons les concepts de « croquis » et de « germes de création ». Ces concepts utilisent la propriété de la matière informationnelle d'être sujette à évolution dans ses formes et sa restitution ; à savoir qu'un objet informationnel peut être soumis à des métamorphoses et prendre des formes différentes selon la démarche d'un auteur tout en restant un objet informationnel. De là une élaboration méthodologique pour une approche de la conception informatique orientée vers des créations de sensibilité artistique.

Nous observons également que ce qui est de l'ordre du non-calculable et relève des investissements psychiques se manifeste néanmoins à travers la forme d'un calcul. Ce qu'exprime alors le calcul n'est pas de l'ordre de la résolution de problème, mais de l'ordre du plaisir et en particulier du plaisir de calculer l'incalculable : le calcul de la beauté par exemple. Mais surtout, ce qui peut apparaître comme non-calculable dans la réalisation de programmes informatiques est la démarche elle-même qui toutefois peut faire l'objet d'une

rationalisation a posteriori. Nous suggérons le « principe de l'oignon » qui fournit un modèle de développement à la démarche afin d'illustrer un processus de création et de conceptualisation évolutif.

Dans la mesure où l'espace robographique apparaît en relation avec de nombreuses disciplines et non limité aux seules problématiques de l'image ou du son, il fournit un modèle d'ouverture généralisée : c'est la nature polymorphe de l'espace robographique. La programmation informatique apparaît dans l'espace robographique comme un moyen d'expression et une écriture : en théorie, n'importe quel sujet peut y être abordé. Cependant une transposition du sujet abordé vers l'espace robographique est opérée. L'écriture construit et réalise un programme et cet objet devient manifeste concrètement dans la machine ; en tant que tel, il acquiert des propriétés d'autonomie, ce que nous désignons par le « principe de Pinocchio ». En sorte qu'il ne s'agit pas uniquement du sujet traité mais également de l'espace robographique lui-même et de l'idée que l'auteur du programme s'en fait. En ce sens de réduction à l'espace robographique cette ouverture généralisée fournit aussi sa propre limite.

Table des matières

Introduction 1

Partie 1 : Pratiquer un art du programme 7

Chapitre 1 « Art du programme » et non « programmes de l'art »…. 7
Chapitre 2 Poser des projets 23
Chapitre 3 Définir des modèles 37
Chapitre 4 Mécaniser des abstractions 61
Chapitre 5 Art, science jeu ou bidouille ? 77

Partie 2 : Être créatif 93

Chapitre 6 Phénoménologie des programmes 95
Chapitre 7 Exemples de réalisations 105
Chapitre 8 Développer un imaginaire 143

Partie 3 : Espace robographique, pixel et ligne sur écran 155

Introduction Figures robotiques de l'imaginaire 155
Chapitre 9 Couleur numérique du pixel 161
Chapitre 10 Palettes de nombres 175
Chapitre 11 Surface de mémoire 195
Chapitre 12 Formes 217
Chapitre 13 Quelques techniques du mouvement 239
Chapitre 14 Croquis de mouvements 269
Chapitre 15 Histoires : le non calculable 299
Chapitre 16 Polymorphisme robographique 319

Conclusion : Jouer d'un langage de programmation 349

Références bibliographiques 357

Introduction ……. 1

Partie 1 : Pratiquer un art du programme ……. 7

Chapitre 1 « Art du programme » et non « programmes de l'art »..... 9

1. Artiste programmeur ……. 11
   1.1 Utilisation de logiciels ……. 11
   1.2 Passage à la programmation ……. 13

2. Entrer dans la conception d'un programme ……. 16
   2.1 Réalisation d'un logiciel musical ……. 16
   2.2 Conséquences sur l'approche musicale initiale ……. 18

3. Concevoir un espace d'expression ……. 19
   Audiovisuel de synthèse ……. 20

Chapitre 2 Poser des projets ……. 23

1. Connaissance par projet ……. 24
   1.1 Projet ou résolution de problème ? ……. 24
   1.2 Connaissance contrainte à l'information ……. 26
   1.3 Connaissance prise au jeu de l'information et du sens ……. 27
   1.4 Double jeu de la connaissance au sein du projet ……. 28

2. Fonctionnement d'un projet ……. 29
   2.1 Trois aspects principaux ……. 29
   2.2 Relier des idées ……. 30
   2.3 Constitution d'un domaine de connaissance ……. 32

3. Exemple de projet : « La Vénus du mythe des races métalliques » ……... 33
   3.1 Début ……. 33
   3.2 Développement ……. 34
   3.3 Fin ……. 35

Chapitre 3 Définir des modèles ……. 37

1. Modèle, construction subjective ……. 38
   1.1 Systèmes, modèles ……. 38
   1.2 Cécité créatrice ……. 39
   1.3 Programmer librement ……. 39

2. Modélisation dans un projet ……. 40
   2.1 Lieu d'action ……. 40
   2.2 Liaison double face ……. 41
   2.3 Contact expérimental ……. 42

3. Formes de la modélisation ……. 43
3.1 Modèles croisés ……. 43
3.2 Roues cognitives ……. 45
3.3 Spirale finalisée ……. 46
3.4 Modèles extrapolés ……. 47

4. Eléments de formalisation ……. 47
4.1 Langage formel, représentation ……. 48
4.2 Formalisme ……. 49
4.3 Concepts ……. 50
4.4 Exemple : programmer des Nénuphs ……. 51

Chapitre 4 Mécaniser des abstractions ……. 61

1. Abstraction ……. 61
1.1 Pragmatique de l'usage ……. 61
1.2 Exemple d'une figure : le graphe ……. 64
1.3 Voisinage simplification – complexification ……. 64
1.4 Trois outils fondamentaux ……. 65

2. Modèles de données ……. 66
2.1 Définition ……. 66
2.2 Modèle de données d'un langage de programmation …….. 66
2.3 Terminologie des types abstraits, méthodologie ……. 67

3. Algorithmes ……. 68
3.1 Exemple et précision de la définition ……. 68
3.2 Algorithme et notation ……. 68
3.3 Temps d'exécution, analyse de la complexité d'un algorithme ……. 69
3.4 Choix des algorithmes ……. 71

4. Structures de données ……. 72
4.1 Définition ……. 72
4.2 Limite d'une structure de données ……. 73
4.3 Présentation du concept de liste chaînée ……. 73
4.4 Choix d'une structure de données ……. 74

5. Interdépendance, orientations ……. 75

Chapitre 5 Art, science jeu ou bidouille ? ……. 77

1. Corps chimérique ……. 78
1.1 Art-science-jeu ……. 78
1.2 Bidouille ……. 78
1.3 Chimère ……. 79

2. Sujet ……. 80
2.1 Expérience du mythe ……. 80
2.2 Sphinx-machine ou simulation de l'esprit ……. 81

3. Langage 82
3.1 Problématique de diversification 83
3.2 Spatialisation de la raison 83

4. Expression 85
4.1 Essai de liaison littérature-IA 85
4.2 Argument mathématique 87

5. Orientations 89
Questions de l'erreur 89

Partie 2 : Être créatif 93

Chapitre 6 Phénoménologie des programmes 95

1. Propriété de réfraction 96
1.1 Une propriété émergente du programme 96
1.2 Comme une mélodie, un objet de culture 97

2. Entrer en résonance avec le réel 98
2.1 Compléter une organisation 98
2.2 Création de situations nouvelles 98
2.3 Statuer sur une situation 99

3. Langage sculptural 99
3.1 Deux langages irréductibles 100
3.2 Deux niveaux de pression pour la propriété de réfraction 100
3.3 Concepteur en entrée sur réseau de préoccupations 101

4. « Intentionalité » 101
4.1 Intentionalité selon John Searle 101
4.2 « Formes fonctionnantes » 102

Chapitre 7 Exemples de réalisations 105

1. Systèmes scénaristiques 107
1.1 NOTES VOLANTES 107
1.2 DICTY L'AGREGUE 110

2. Systèmes génératifs 114
2.1 GRAPHITOS 114
2.2 GRAPHITOR 123
2.3 DISCOURS COTON 127

3. Mises en scène de graphisme 132
3.1 MIKADO 133
3.2 BUISSON PLATONIQUE 137
3.3 BUISSON VANITE 138

3.4 BUISSON EROTIQUE 139

4. Référence à la culture du jeu 140
KANDINSKY SHOOT AGAIN 140

Chapitre 8 Développer un imaginaire 143

1. Le robographe 145

2. Contexte d'une dichotomie technique-création 146
2.1 Disjonction plus conjonction 146
2.2 Artiste consommateur et « fonctionneur » d'outils 147
2.3 Malaise face à la technique 148

3. Ouvrir l'espace du robographe 150
3.1 Troisième voie 150
3.2 Impressionnisme technologique 151

4. Machine ou image ? 152
4.1 Ecriture de « machines-images » 152
4.2 Elargissement scénaristique 153

Partie 3 : Espace robographique, pixel et ligne sur écran 155

Introduction : Figures robotiques de l'imaginaire 156

1. Espace technique-création, espace perceptif 156
1.1 Mouvements d'intériorisation et d'extériorisation 156
1.2 Etablir une écriture artistique et de création 158

2. Aspect pratique de l'environnement de développement 159
Librairie Allegro 159

Chapitre 9 Couleur numérique du pixel 161

1. Définition du pixel 162

2. Questions liées à l'implémentation machine de nombres entiers 163
2.1 Espace binaire de la mémoire numérique physique 164
2.2 Faire un nombre avec des octets 165
2.3 Faire un nombre positif ou négatif 166
2.4 Différents types d'entiers 169
2.5 Se représenter les variables dans la mémoire 170

3. Deux modes des couleurs 172
3.1 Mode « vraies couleurs » 172

3.2 Mode 256 couleurs 172

Chapitre 10 Palettes de nombres 175

1. Définir une palette 176
1.1 Codage de la palette 176
1.2 Initialiser une palette 178
1.3 Exemple d'une palette en nuances de gris 178

2. Quatre dégradés de couleurs 181
2.1 Dégradé constant de couleur constante 182
2.2 Dégradé constant de couleur variable 184
2.3 Dégradé avec variation de constance sur couleurs variables 186
2.4 Dégradé ponctuellement inconstant à partir d'une origine fixe 189

3. Réflexions liées aux styles de programmation 191
3.1 Introduction aux styles 191
3.2 Exemple de recommandations 193

Chapitre 11 Surface de mémoire 195

1. Plan de l'image numérique 196
Codage de l'image 197

2. Saisir la mémoire dans le langage C 198
2.1 Manier des adresses 199
2.2 Précision sur les tableaux et les pointeurs 200
2.3 Calculer une taille en octets 201
2.4 Fonctions d'allocation de mémoire 203

3. Création d'une image en mémoire 205
3.1 Reprise des éléments du codage 206
3.2 Détail d'une fonction d'initialisation 207

4. Deux robographes primitifs 211
4.1 Colorer un pixel ou connaître sa couleur 211
4.2 Conversion de type pour le mode « vraie couleur » 214

5. Accéder à l'écran de l'ordinateur 215
Mémoire vidéo, image écran 216

Chapitre 12 Formes 217

1. Formes simples 219
1.1 Tracer une droite, tracés dérivés 220
1.2 Tracer un cercle, extrapolations ludiques 225

2. Compositions de pixels 229
2.1 Contours d'une forme 229

2.2 Courbes dérivées de contours ……. 234

Chapitre 13 Quelques techniques du mouvement ……. 239

1. Animation de couleurs ……. 241
1.1 Rotation de palette ……. 241
1.2 Interpolation de palettes ……. 243

2. Déplacement de formes géométriques ……. 245
2.1 Principe général ……. 245
2.2 Double buffer, fonctions d'affichage ……. 247
2.3 Déplacement sur des images, modes d'affichages……. 254

3. Bouger à partir d'images ……. 256
3.1 Sur place ……. 256
3.2 Autour de l'écran ……. 259
3.3 Dans une boite en mouvement ……. 264
3.4 Ressources image, fichiers « *.dat » ……. 268

Chapitre 14 Croquis de mouvements ……. 269

1. Effet de feu ……. 271
1.1 Principe ……. 271
1.2 Mise en forme et initialisations ……. 272
1.3 Processus générateur ……. 275

2. Croquis de sable ……. 278
2.1 Principe ……. 278
2.2 Mise en forme et initialisations ……. 280
2.3 Processus générateur ……. 282

3. Vers une histoire de danse à partir de flammes ……. 285
3.1 Principe ……. 285
3.2 Mise en forme et initialisations ……. 286
4.3 Processus générateur ……. 288

4. Entre explosions et fantômes ……. 292
4.1 Principe ……. 292
4.2 Mise en forme et initialisations ……. 292
4.3 Processus générateur ……. 295

Chapitre 15 Histoires : le non calculable ……. 299

1. Principe de l'oignon ……. 301
1.1 La démarche est non calculable ……. 301
1.2 Métamorphoses de l'objet avec la démarche ……. 303
1.3 Résistance à propos des matériaux algorithmiques …….. 309
1.4 Compositions « fonctionnantes » ……. 310

2. Germes de création ……. 311
2.1 Point de départ exploratoire ……. 311
2.2 Plusieurs niveaux d'approche au regard de la programmation…….. 312

3. Exemple de racine automate ……. 313
3.1 Principe ……. 314
3.2 Mise en forme et initialisation ……. 314
3.3 Processus générateur ……. 315

Chapitre 16 Polymorphisme robographique ……. 319

1. Ouverture généralisée ……. 321
1.1 Distance de l'écriture aux sujets traités ……. 321
1.2 La forme issue de la démarche d'un auteur ……. 321
1.3 Diversifications des approches pragmatiques ……. 322
1.4 Une ouverture qui est sa propre limite ……. 322

2. Exemple d'ouverture sur le son, robographe sonore ……. 322
2.1 Du son dans une surface de mémoire ……. 323
2.2 Organiser un flux sonore continu ……. 325
2.3 Sonorités, perspectives de morphing ……. 328

3. Implémentation d'un robographe textuel ……. 331
3.1 Principe ……. 331
3.2 Mise en forme et initialisation ……. 331
3.3 Processus générateur ……. 337

4. Vers des curiosités robographiques : un robographe organisationnel ……... 341
4.1 Principe ……. 342
4.2 Mise en forme et initialisation ……. 342
4.3 Processus générateur ……. 343

Conclusion : Jouer d'un langage de programmation …….. 349

1. Articuler idée et action ……. 350
1.1 Cheminement individuel, modèle ouvert ……. 350
1.2 Acte de conscience ……. 351
1.3 Rencontrer le public ……. 351

2. Toucher le public ……. 351
2.1 Transmettre la programmation ……. 352
2.2 Réalisations artistiques, « Démo », installation ……. 352
2.3 Contact avec l'entreprise ……. 353
2.4 Contact avec le milieu associatif ……. 354

Références bibliographiques ……. 357

INTRODUCTION

Terreau informationnel

La chimère est l'être hybride, en attente d'unité, dans le mouvement de la composition. La gargouille est une figure fantastique, le rattachement à un grand corps comme une cathédrale, une fonction d'évacuation de l'eau, le rappel du bruit de l'eau qui circule dans des canalisations et avec lui celui des entrailles et des angoisses qui s'y manifestent.

Chimères et gargouilles informatiques représentent ici, entre art et science, le nulle part qui sépare deux territoires, un entre-deux mondes qui passe par la créativité. Toutes deux désignent des objets informationnels errants, non répertoriés, empruntés à droite ou à gauche, vaguement inutiles, hors cadre des connaissances institutionnelles, banals ou incompréhensibles, souvent abandonnés, comme des objets jetés à la rue, des objets nomades, des objets dévalorisés, sans grand intérêt apparent. Ce sont des espèces de programme qui ne font pas par nature l'objet d'un plan d'ensemble et au sujet desquels il n'y a pas de cadre théorique. Ces objets programmés se trouvent regroupés éventuellement sous l'action de communautés Internet, autour de différents types de projets d'informaticiens chevronnés, amateurs ou hobbyistes. Ils apparaissent comme des sortes de micro-capteurs d'attention pseudo-organiques, ou de réceptacles d'idées, des lieux d'agrégats d'informations en forme de petits concepts autonomes.

Aux yeux de la recherche technologique dure ils sont insuffisants, futiles, inexistants, surannés, des « tags » ou « pochoirs » le long de grosses autoroutes informationnelles, aux yeux des artistes ils seront trop techniques et de ce fait invisibles souvent. Ainsi chimères et gargouilles informatiques coïncident-elles avec une situation de porte-à-faux, une situation de passerelle, une ligne de crête, une indication de rupture entre deux horizons, ni ordinaire, ni extraordinaire, comme un tableau renversé par une femme de ménage chez Wassily Kandinsky, elles représentent le territoire des glissements et des entre deux.

Toutefois il ne s'agit pas ici d'écrire le roman de la vie de ces « bestioles » informatiques au sein de notre culture contemporaine avec les modalités d'apparition et de développement de ces populations d'objets informationnels plus ou moins domestiques, dont certains peuvent être agressifs comme les virus. Mais il y a là un lieu de la conception informatique et ce lieu d'apparition d'une multiplicité de ces sortes de programmes originaux retient surtout notre attention. Il y a là une probabilité de vie et de développement à partir d'une démarche de création d'une part et de la réalisation de logiciels d'autre part. L'enjeu pour nous est la démarche de création de logiciels originaux.

Objet de la recherche

La question que nous nous sommes posée initialement est la suivante : se peut-il qu'en tant qu'artiste on ne veuille plus dessiner, peindre, composer ou jouer de la musique mais programmer ?

Ainsi notre recherche porte t'elle sur la programmation informatique dans des langages fondamentaux comme le C ou le C++, en tant qu'elle permet à un artiste ou un créateur de définir sa propre écriture, son propre langage. Selon ce positionnement, la programmation intervient dans les processus cognitifs plutôt

qu'elle ne permet de les simuler. Cette approche favorise le développement d'une créativité technique qui est associée à l'écoute des investissements psychiques et de la production de sens.
La première moitié du travail consiste à se repérer entre informatique science dure et informatique science humaine. Elle est constituée de deux parties. Il s'agit de montrer comment « implémenter » au sens informatique « science dure » une idée ou une représentation au sens informatique « science humaine » à travers la réalisation d'un programme. L'objectif est de poser la base d'une pratique de l'informatique.
La première partie est une exposition de la problématique ainsi que la définition d'un espace de connaissance qui reste ouvert et propice à cette démarche de recherche. Ensuite la seconde partie montre des résultats obtenus et une mise en pratique : des créations de logiciels sont présentées, chacun ayant fait l'objet de diverses présentations publiques.

La seconde moitié de l'ouvrage, constituée par la troisième partie, correspond à l'élaboration théorique de la pratique. En partant du domaine de l'image numérique, nous insistons sur l'activité d'un programme dans la construction d'une image : le lieu de la composition de l'image c'est la conception de ce programme. Le robot devient le lieu de l'imagination et il est solidaire d'une écriture. La conception de robots qui est le « lieu de l'imagination » nous détache de la réalisation d'images, et notre objectif ensuite est de faire percevoir la notion de *robographe* telle qu'elle s'étire entre le robot manifeste, celui que l'on voit, et le robot latent, celui que l'on se représente mentalement. A ce point nous observons que ce qui est de l'ordre du non-calculable peut néanmoins apparaître sous la forme d'un calcul et que la nature polymorphe de l'espace robographique apparaît clairement non limitée à des images ou du son.

Nous concluons finalement sur des actions entreprises auprès de différents publics, dans les cadres universitaire, de l'entreprise, ou de l'action socioculturelle, du grand public et des associations.

Le choix du langage C

Nous avons cherché un « point de départ » pour l'informatique, une origine s'il en est une aujourd'hui… Finalement le langage C a répondu à cette attente. Initialement conçu pour l'écriture du système Unix, on peut voir dans ce langage une origine. Le C est à l'origine technique de nombreux projets.
C'est-à-dire qu'avec lui, même lorsque des aspects techniques sont masqués par une librairie qui fournit des réponses préfabriquées, rien de l'environnement n'est inaccessible à l'écriture ou à la lecture. C'est toutefois théorique compte-tenu des apports de connaissances autres que la seule connaissance du langage qui sont nécessaires souvent à la compréhension ou à la réalisation. Cependant toutes les questions semblent pouvoir être abordées avec le langage C et le coût éventuellement élevé en connaissances annexes n'est pas une propriété du langage. Le C est un langage conceptuellement très libre.

Dans ce travail nous avons essentiellement voulu engager le langage, et la place possible des mathématiques par exemple est extrêmement réduite.

Matière informatique

Glaner des exemples de code source fait partie de la conception informatique. Il n'est pas possible en effet de réinventer la roue pour chaque programme et de ce point de vue il y a souvent un côté chercheur de champignons à l'activité de conception. Nous avons en effet

importé beaucoup de matière informatique dans ce travail. C'était inévitable puisque cette matière et ce que l'on en fait sont le sujet. Nous avons dû nous employer à donner les clefs nécessaires à sa compréhension. Le faisant nous n'ignorons pas la nécessité de la pratique et des exercices répétés pour la maîtrise d'un langage de programmation. L'important reste toutefois la construction de la démarche qui tout à la fois donne lieu à la réflexion et à l'activité et il y a dans ce travail une réelle interrogation scientifique sur comment construire une connaissance. Ainsi n'y a-t-il pas d'a priori sur la formation initiale du lecteur qui peut être de culture scientifique, littéraire ou artistique. Plusieurs lectures sont possibles qui incluent différents degrés de familiarité avec les notions de programmation proposées.

Développement évolutif

En informatique le terme consacré pour les activités est le développement, le développement informatique. Nous avons opté pour une logique de recherche plutôt que de développement dans la mesure où l'efficacité recherchée ne réside pas dans de nouveaux algorithmes opérationnels. Nous avons voulu rapprocher la notion de développement de l'esthétique et de l'imaginaire, développer un imaginaire. En particulier cela soulève la difficile question d'un développement évolutif qui est comme une métamorphose continue. En général on entend par développement le développement de quelque chose qui est déjà là, et sur la base d'une étape donnée l'approfondissement et l'élargissement de cette étape, mais tout en restant au même niveau d'étape. L'avancée métamorphique d'une idée est certes un développement de l'idée mais un développement à la transversalité imprédictible liée au développement de l'être, à son investissement et son ouverture psychique, son devenir.

Positionnement artistique

Si l'artiste souvent ne se définit pas comme un expert mais comme quelqu'un qui fait une proposition alors nous proposons ceci :
L'artiste dialogue avec la matière mais l'aspect scientifique qui l'intéresse paraît être la connaissance qui est développée à partir d'un lien affectif à la matière, c'est la connaissance de l'engagement de l'être à travers les modalités de son art. Ainsi la programmation peut-elle apparaître non seulement comme une problématique scientifique dure mais également comme une problématique d'expression que nous rattachons à l'art et à la connaissance de l'être avec l'élaboration d'une démarche et d'une écriture.

Nous n'avons cependant pas souhaité faire un travail théorique d'histoire de l'art. Du point de vue théorique en art, nous nous sommes appuyés essentiellement, et du fait d'une affinité ressentie, sur la démarche du peintre Wassily Kandinsky, décrite dans l'ouvrage « Point et ligne sur plan ». Les aspects que nous avons retenus sont le principe de la « résonance intérieure » et du rapport à l'inconscient, le fait de dégager des primitives de conception, mais également de concevoir un espace comme un plan de référence à partir duquel donner sens et forme.
L'art, a priori, se caractérise aujourd'hui par une grande liberté de démarche. La démarche repose ici sur l'appropriation d'un langage de programmation. Elle est artistique dans notre optique dans la mesure où elle ouvre des perspectives d'exploration de formes rendues possibles par l'informatique sans pour autant être mue par des préoccupations ou des objectifs purement scientifiques. En fait, ce positionnement se trouve à l'interface de différents milieux entre informatique ET autre chose. C'est le fait de la frontière poreuse

de l'informatique qui se confond avec une écriture et se fond dans les sujets que l'on aborde avec.

Tension entre deux langages, questions typographiques

Notre travail, qui traite de l'appropriation d'un langage de programmation dans une dynamique de créativité, a soulevé de façon inattendue des difficultés typographiques du fait de l'imbrication des deux écritures, le langage C et le français. Les deux langages entrent dans une relation particulière. Nous avons abordé ce point sous l'aspect d'une phénoménologie des programmes au chapitre Six. Cependant il paraît important de préciser ici quelles sont les conventions typographiques qu'il a fallu envisager et en partie créer. En effet, étant sur une passerelle entre informatique et autre chose, il n'y a pas d'appartenance implicite au domaine de connaissance de l'informatique et la plupart des termes employés apparaissent comme spécialisés et venant de l'extérieur. Mais d'autre part, un des objectifs est précisément de réduire la distance d'avec le langage de programmation. En soi cet aspect de la typographie est assez captivant et peut certainement faire l'objet d'une recherche spécifique[1].

Notre point de vue est à la fois de créer une familiarité avec le langage mais également de conserver la distance avec les engagements auxquels il donne lieu et l'objectif est bien sûr de rendre la lecture la plus simple et compréhensible possible.

Pour ce faire nous adoptons la convention suivante :

- Les mots-clés du langage sont entre guillemets lors de leur première apparition et sans ensuite.
- De même pour les noms de variables. Elles prennent des guillemets lorsqu'elles apparaissent la première fois dans le texte et n'en ont plus tant qu'elles demeurent à proximité de leur contexte d'apparition. Elles reprennent des guillemets en cas de réapparition plus lointaine, un peu comme s'il s'agissait d'une citation.
- Les noms de fonction sont toujours entre guillemets même lorsqu'ils sont connus.

A la différence des mathématiques, semble-t-il, nous avons constaté l'importance de baptiser les variables dans le cadre de la programmation. Les variables sont baptisées en fonction du sens qu'elles ont dans le programme mais également selon l'esprit, la méthode, la mémoire de celui qui écrit le programme. Il y a une construction du nom des variables, puis une tradition souvent personnalisée de ces noms qui se rattache à l'auteur du programme. La variable « i » par exemple n'est pas explicite comme les variables x et y pour noter une position, mais d'une façon générale les variables en programmation n'ont pas le caractère d'anonymat que l'on rencontre dans les mathématiques.

Nous avons aussi été confronté à des conflits syntaxiques d'apparence anodine comme celui-ci : sachant que « char » est un mot clé du langage de programmation, dans l'expression « un tableau de char(s) » , difficile de savoir à quel langage doit appartenir le mot char. S'il reste attaché à la programmation il ne prend pas de pluriel et oui s'il est incorporé en tant qu'élément du texte en français.

Notre point de vue est que la prééminence de la syntaxe du langage de programmation a

[1] Par exemple à propos uniquement de la ponctuation française, nous avons trouvé et consulté un traité étonnant : [DRILLON, 1991]

plus de sens qu'un accord de cette syntaxe sur la langue (c'est peut-être une raison à cause de laquelle il est fréquent d'avoir une mauvaise orthographe lors de périodes assidues de programmation)

D'une façon générale, nous avons laissé les formules entre guillemets ainsi que les noms de fonctions mais également quelques noms de variable qui pouvaient prêter à confusion. Les guillemets servent alors à séparer les deux contextes. La plupart du temps les guillemets sont utilisés soit pour introduire un terme, soit pour éviter une confusion possible entre une expression de programmation et un terme du langage naturel.

PARTIE 1

Pratiquer un art du programme

Chapitre 1

« Art du programme » et non « programmes de l'art »

Expérience de musicien

Le passage à la micro-informatique pour un certain nombre de musiciens dans les années 1980 s'est fait assez « naturellement ». Les premiers séquenceurs numériques ont répondu très adroitement à des méthodes de travail qui reposaient sur des manipulations de bandes, collages, boucles et tout ce que l'usage du magnétophone pouvait permettre. Il fallait accepter l'usage de synthétiseurs MIDI et, la puissance des ordinateurs étant insuffisante, renoncer pour un temps à des enregistrements de sons, de bruitages et d'instruments acoustiques.

Le « home studio » numérique s'est imposé à son prédécesseur analogique[1]. L'approche numérique de la musique s'est généralisée et avec de nombreux avantages, parmi lesquels un nombre de pistes quasiment illimité (ce qui pour des utilisateurs de magnétophones multipistes très onéreux avait beaucoup de sens).

En particulier, la pratique initiale a pu être élargie par le regroupement de plusieurs aspects possibles pour un même matériau musical. Par exemple, une mélodie peut apparaître sous la forme graphique d'un ruban, ou sous celle d'une notation chiffrée ou encore sous forme de partition. De plus les possibilités de copier coller, déplacer, transposer, modifier sont devenues énormes. Il faut ajouter l'emploi de synthétiseurs répondant à la norme MIDI. Cette norme est un protocole de communication qui permet l'échange des informations entre synthétiseurs et séquenceur. Le séquenceur envoie des instructions codées en MIDI au synthétiseur et peut lui faire exécuter tout ce que la norme admet.

L'expérience de la pratique de l'informatique musicale fait apparaître progressivement de nouveaux faits avec, de la part des musiciens concernés[2], une conscience accrue et une meilleure perception de l'activité. Par exemple, l'observation que l'utilisation de séquenceurs numériques qui proviennent d'éditeurs différents peut avoir des répercussions qualitatives sur la composition musicale. En effet, des différences dans l'agencement d'un même ensemble de concepts pour l'approche de la musique peuvent influencer la composition musicale. Peut-être s'agit-il de la répartition différente des tâches qui oriente, par le jeu des facilités d'accès, l'action plutôt dans un sens que dans un autre. Ou, dans le même ordre d'idée, il y a également le rapport esthétique à l'outil logiciel, son ergonomie.

[1] Même principe mais avec du matériel analogique. Pour l'essentiel : deux ou trois magnétophones (au moins un multipiste quatre canaux et un stéréo) quelques micros, une table de mixage, des synthétiseurs, deux ou trois racks d'effets (reverb, delay, chorus, flanger etc.). Le terme de « home studio » n'est arrivé que plus tard avec la généralisation de l'emploi d'ordinateurs et de matériel numérique.

[2] Nous pensons à des musiciens qui envisagent le logiciel pour ainsi dire comme un instrument de musique et non comme un vulgaire « calepin ». C'est-à-dire des personnes qui se sont intéressées à cette nouveauté en tant que telle et qui ont pu, du fait de leur pratique initiale, non seulement effectuer un raccordement, mais également réenvisager des habitudes de travail et modifier leur approche.

En tout cas, l'idée que se fait de la musique un éditeur-concepteur de logiciels n'est pas neutre sur le plan musical. Elle intervient au niveau de l'interprétation des concepts de neutre sur le plan musical. Elle intervient au niveau de l'interprétation des concepts de séquenceur numérique et de leur mise en oeuvre dans le logiciel. Du coup naît une curiosité pour l'objet logiciel derrière lequel se révèlent des auteurs.

La curiosité se porte sur cette possibilité d'interprétation du concept de séquenceur numérique. Est-il possible d'avoir une autre idée de séquenceur numérique ? Une rencontre avec Manfred Hanke programmeur et Chris Hanke son frère musicien[1] apporte une réponse. Ils réalisent sur ordinateur Atari deux logiciels de recherche Hypnosis et Quaderno. Ils proposent notamment la possibilité d'obtenir des mélodies à partir de gestes à la souris dans des rectangles à l'écran, ou encore d'utiliser des processus de génération automatique pour la réalisation des séquences musicales et des accompagnements.

Ainsi, le logiciel résulte d'un travail d'auteur et peut faire l'objet de recherches. Son écriture renvoie à la définition d'une stratégie par rapport au domaine de connaissance initial. Ecrire un logiciel de musique revient à mettre en question sa pratique musicale tout en réalisant concrètement par un objet logiciel la question elle-même.

Cette ouverture vers la recherche informatique dans les domaines du son et de la musique conduit éventuellement vers des centres de recherche afin d'expérimenter d'autres propositions de logiciels musicaux et de recherche : par exemple, Audiosculpt, Diphone, et Max qui sont des productions de l'IRCAM. En particulier, Max est un logiciel de programmation graphique dédié initialement à la musique et largement utilisé pour le multimédia maintenant.

Insensiblement, l'intérêt est passé de la composition musicale vers l'écriture de logiciels.

1. Artiste programmeur

1.1 Utilisation de logiciels

Cas de l'artiste qui utilise un ou plusieurs logiciels dans le cadre de ses activités : une pratique artistique sert de référent à la création du programme informatique ainsi qu'à son utilisation future. Cette pratique a été modélisée. C'est-à-dire que l'activité a été à la fois décrite et réduite en une structure formalisée _ le programme _ et orientée de façon subjective selon un sens, une volonté, des objectifs en particulier.

Ces programmes sont souvent considérés comme une assistance, une aide à la conception ou à la production dans les domaines de l'image, de la musique et du son « Musique Assistée par Ordinateur ». Ils font l'objet d'une ingénierie et nécessitent des équipes de concepteurs-ingénieurs qui ne sont pas des artistes sur des périodes pouvant être longues, surtout si l'on tient compte du phénomène des « mises à jour ».

[1] En 1994 il s'agissait de KO STUDIO, Label indépendant, 107 rue de Reuilly 75012, Paris.

Outils et normes intrinsèques

Ces logiciels, dits souvent « logiciel auteur[1] », sont considérés comme des outils par leurs utilisateurs. Et, de fait, il n'est pas très judicieux de parler d'aide ou d'assistance. Pour l'utilisateur, il y a un phénomène de rétroaction entre sa pratique effective de référence et le modèle programmé de l'activité. L'utilisation du nouvel outil interroge la pratique initiale par des frustrations parfois, mais aussi et surtout, par des possibilités spécifiques. La curiosité peut être renouvelée sur certains aspects de la pratique artistique. Ces outils n'empêchent pas d'interroger la tradition artistique et de conduire à des frontières de cette tradition.

Cependant, les logiciels auteur contribuent à normaliser la nouvelle pratique qui s'appuie sur le modèle véhiculé par le programme. Par exemple, un séquenceur numérique (séquenceur MIDI) correspond à un modèle d'activité de composition musicale. Il peut y avoir différentes présentations de séquenceurs mais finalement un seul domaine d'activité, celui défini par l'idée de séquenceur (idem pour la retouche d'image, le montage multimédia). L'élasticité de la pratique artistique qui est étendue par le programme est en même temps contrôlée par lui.

Ce phénomène est en partie lié aux principes formels sur lesquels repose la modélisation des connaissances sous-jacentes. Il y a pour ainsi dire une syntaxe et un langage du modèle qui rassemblent « des primitives conceptuelles auxquelles sont associées des significations et des formes » [CAPLAT, 2002]. Pour Guy Caplat : « Exprimer un modèle dans un formalisme donné induit une adhésion aux principes sur lesquels il repose et en particulier le respect des règles de structuration et de composition de ces primitives. *En retour, il offre à ses utilisateurs un langage commun : un modèle peut alors être interprété d'une façon homogène par tous*. »

Ces outils-programmes qui réticulent et normalisent des savoir-faire artistiques techniques facilitent l'intégration de ces savoirs-faire et de ces techniques dans des productions industrialisées. En effet, parler d'outils plutôt que d'instruments pour un musicien, c'est déjà un peu délocaliser la pratique artistique. Des savoirs-faire ou des objectifs industriels viennent compléter ces outils qui finalement concernent au premier chef les « ouvriers créatifs » des industries audiovisuelle et multimédia. Par exemple, un savoir-faire inspiré de l'industrie audiovisuelle est lisible dans un logiciel comme Director. De même que celui du studio d'enregistrement est compris dans Logic Audio ou Cubase.

Matériaux son et image ?

Par ailleurs, qu'il s'agisse du traitement d'une image ou du traitement d'un son, un clic souris reste un clic souris ! L'informatique introduit un tronc commun de savoirs-faire valable pour la plupart des logiciels auteurs et il y a un effet d'alignement de préoccupations initialement différentes comme le traitement du son ou de l'image.

Son et image se dématérialisent en partie au profit du « traitement de l'information ». Il n'est pas sûr, au plan sémantique, que l'information puisse garder la même « teneur de son » ou « teneur d'image ». Il n'y a pas de différence physique dans l'expérience de

[1] Parmi les plus connus : Photoshop pour la retouche d'image, Director pour le montage multimédia, Flash pour l'animation vectorielle, Première pour la vidéo, Logic Audio et Cubase pour la musique etc.

traitement, et via les motifs standards des interfaces, moins de différences intellectuelles également, alors que les positions de peintre ou de musicien sont tout de même deux réalités bien différenciées.

Il ne s'agit plus seulement de jouer avec du son ou des images, mais ce sont les différents traitements de l'information qui sont en jeu et au travers d'un modèle formalisé. En dépit de la prégnance de la culture de référence du domaine artistique concerné, il s'agit de « jouer avec un modèle » et son substrat formel. C'est donc la réalité de l'information et des traitements opérés qui intéressent au premier chef, alors que ni le peintre ni le musicien n'ont à voir avec les notions d'information ou de traitement dans une pratique classique indépendante de l'informatique.

1.2 Passer à la programmation

Le passage à la programmation est un basculement. Il n'y a pas de lien nécessaire, logique ou rationnel qui conduise un artiste dans son domaine à s'approprier les termes et les techniques de la conception informatique elle-même et pour elle-même, d'autant que cela suppose beaucoup de travail. C'est le fruit d'un cheminement personnel. En particulier, le nouvel espace de conception artistique ET informatique qui en résulte n'est pas une extension de l'espace précédent de conception. Le musicien, le peintre ou le plasticien ne feront pas « plus de quelque chose », ils découvrent autre chose et c'est en cela qu'il y a basculement.

Passer à la programmation, c'est changer de regard sur ce qu'est un programme. Si nous revenons à cette idée qu'un logiciel offre à ses utilisateurs un langage commun issu de l'adhésion d'un modèle à un formalisme donné, savoir dans quelle mesure l'utilisation de ce langage n'est pas déjà en soi un peu de la programmation est une question qui finalement peut se poser. En effet, utiliser le logiciel revient à spécialiser le logiciel dans le cadre de la syntaxe qu'il met en œuvre et qu'il propose, des calculs qu'il sait faire, et selon un sens personnalisé. De ce point de vue, utiliser un logiciel serait comme une entrée dans la programmation : l'utilisateur met son activité sous la forme d'un programme par l'utilisation d'un programme, et par extrapolation nous pouvons presque dire que l'instanciation du programme crée en quelque sorte un sous programme à l'intérieur du programme.

Par exemple, le jeu « Sim » pourrait être qualifié de système de programmation graphique pour un univers quotidien virtuel. Il rend possible la création de personnages avec des attributs sociaux et psychologiques, et aussi le suivi de leurs manifestations comportementales, en fonction de l'évolution des contextes et des conditions de vie. Il y a également des possibilités d'agir sur l'environnement, de créer des habitations avec leurs aménagements intérieurs et tout un attirail d'objets qui s'articulent autour. Des interactions naissent du fait des différents personnages et il faut conduire tout ce petit monde en sachant qu'il est contrôlé par des règles un peu caricaturales en vue de distraire.

Dériver un langage formel vers l'imaginaire

Le joueur a le sentiment de manipuler du sens, il manipule en fait des expressions qui ont été codées dans un langage formel, en l'occurrence un langage de programmation. Or, ce

qui caractérise des expressions codées dans un langage formel, c'est d'être manipulables en fonction de leur forme syntaxique et non en fonction de leur contenu sémantique [CAPLAT, 2002, p 28]. Dans un logiciel quelconque, un utilisateur peut souhaiter ou imaginer quelque chose qui est cohérent du point de vue du sens qu'il attribue aux expressions auxquelles il est confronté mais que le programme ne peut pas faire.

Ainsi dans notre exemple du jeu Sim, le système d'une part doit pouvoir être interprété du point de vue de la représentation qu'il met en scène et de l'imaginaire qui lui est associé (vie quotidienne, personnages etc.), mais aussi, dans le même temps, cette interprétation doit pouvoir coïncider avec une mécanique formelle, et le moins possible être démentie ou paraître incohérente. Le programme apparaît dans la double dimension d'un revêtement sémantique et d'une syntaxe formelle, la syntaxe étant enveloppée par le sens comme une main de fer dans un gant de velours. La définition de cette syntaxe est faite de façon à ce que dans un éventail donné de possibilités, elle puisse coïncider au mieux avec une représentation des mouvements et des actions de la vie quotidienne et supporter par là des projections de l'imaginaire du joueur.

En d'autres termes, du point de vue du sens, il y a une dérive possible d'un langage formel vers l'imaginaire, même processus transversal, par exemple, que le glissement du jeu d'un instrument de musique, considéré dans son aspect formel, vers l'imagination ou la sensualité de la musique.

Ainsi, l'ensemble des manipulations syntaxiques offertes par le logiciel devient l'éventail des possibilités scénaristiques et crée l'espace sémantique du logiciel. Eventuellement, l'utilisateur va pouvoir détourner le sens originel de propositions de fonctions du logiciel. L'utilisateur entre dans la définition de l'espace sémantique. Il y a bien deux espaces superposés et mis en parallèle.
Cependant, ce sur quoi le joueur ou l'utilisateur agit effectivement dans le programme ne peut être que l'ensemble des manipulations syntaxiques proposées. Par là, l'utilisateur manipule un langage formel comme Mr. Jourdain fait de la prose.

Dériver un langage formel informatique vers un programme

Conjointement, il y a une autre idée de transversalité qu'il est intéressant d'appliquer à ce langage formel informatique. Il va être mené par phases successives, tant sur le plan méthodologique que technique, à une spécialisation dans le programme. Nous pensons ici aux différentes étapes du développement d'un logiciel[1] ainsi qu'aux différentes phases d'un compilateur[2] : en quelque sorte, une autre forme de dérive de l'écriture informatique qui,

[1] En ce qui concerne la méthodologie de programmation, il existe de nombreuses propositions. Alfred Aho souligne par exemple les grandes étapes suivantes : définition du problème, spécification, réalisation, tests, maintenance [AHO, 1993, p.22]. Voir également les modèles de démarche de Simon [SIMON , 1991, p 131] et de Lemoigne [LEMOIGNE, 1990, p 58].

[2] En ce qui concerne la compilation, selon une définition simplifiée donnée par Alfred Aho, le compilateur est un programme capable de traduire un programme écrit dans un premier langage (langage source) dans un programme équivalent écrit dans un second langage (langage cible). Nous retenons que « Conceptuellement, un compilateur opère en phases, chacune d'elles transformant le programme source d'une représentation en une autre. » [AHO, 2000, p.25]. L'objectif sera, à partir du programme source écrit dans le langage source et éventuellement divisé en module réparti dans des fichiers séparés, de créer un programme cible exécutable (langage machine).

au long de différentes étapes techniques d'abstractions et de traduction, se trouve finalisée dans l'accomplissement de certaines tâches.

Un programme peut apparaître comme un point de convergence du langage informatique qui a été en quelque sorte spécialisé dans l'expression du programme et sous l'action arbitraire d'un auteur (figure 1.1).

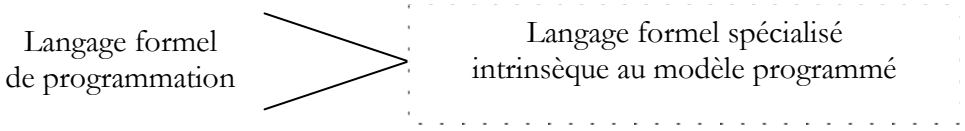


Fig. 1.1 : Spécialisation des possibilités sémantiques d'un langage de programmation

Ainsi vu, le programme conserve une nature linguistique issue du langage de programmation, même s'il réduit ce langage parce qu'il le spécialise dans les tâches qui sont les siennes. Nous pourrions rapprocher cette translation du langage de programmation vers le programme de l'image d'un emboîtement de machines de Turing, chacune simulant une suivante de moindre capacité[1]. Selon cette perspective, un traitement de texte serait un dérivé de langage de programmation spécialisé pour le traitement de documents textes selon un modèle socialement admis de contraintes et de mise en forme.

Incrustation explicite de langages de programmation dans des logiciels auteurs

Si l'on admet cette idée du programme en tant que point de convergence du langage, l'ouverture expressive laissée à ce langage spécialisé est plus ou moins grande selon les objectifs du modèle et les possibilités souhaitées du programme. Des logiciels auteurs comme Director ou Flash définissent explicitement dans leur modèle des langages de programmation en tant que tel (Lingo et ActionScript). Ces langages sont dédiés aux domaines d'activités des logiciels auteurs et ils demeurent intrinsèques aux programmes qui en interfacent la mise en oeuvre. L'implémentation de ces langages permet la réalisation et la création de petits programmes dérivés. Selon les termes que nous avons employés, ces logiciels auteurs laissent à l'utilisateur la possibilité de finaliser lui-même une part de langage laissé à disposition en tant que tel, c.à.d. de programmer. Dans ce cas, c'est explicite.

Ainsi est-ce finalement dans ce travail de spécialisation et de finalisation du langage informatique que nous mettons du sens. En ce qui concerne la création, notamment la création artistique, nous substituons la programmation informatique à l'utilisation de logiciels. L'objectif n'est plus de chercher à finaliser un travail musical avec l'utilisation de

[1] Les machines de Turing correspondent à un modèle mathématique de machines à états discrets, c.à.d. qui passent par bonds instantanés d'un état à un autre. Selon Turing, l'équivalence entre les machines à états discrets se traduit par le fait qu'une machine peut en simuler une autre à la condition d'avoir plus de mémoire. Jean Gabriel Ganascia le résume ainsi : « Turing a démontré l'équivalence de toutes les machines à états discrets qui disposent d'une mémoire infinie. Toute machine finie pouvant, bien évidemment, être simulée par une machine infinie, il est donc théoriquement possible de construire une machine numérique capable de simuler toutes les machines numériques existantes » [GANASCIA, 1990, p. 30].

logiciels de musique, mais de travailler sur le langage même qui est à l'œuvre dans la définition du logiciel et de l'idée musicale qu'il véhicule.

Cette évolution, qui part d'une pratique musicale et aboutit à une pratique de la programmation informatique, pose la question essentielle d'un projet d'écriture de programmes.

2. Entrer dans la conception d'un programme

La première expérience de conception d'un logiciel, nous l'avons menée à partir du domaine musical et dans le cadre d'une recherche en Multimédia [DROUILLON, 1996, 1997]. L'objectif était un travail sur la notion de forme et la création d'un système musical original. Le rôle de la forme a été tenu par un Automate Cellulaire[1].

L'hypothèse est que la dynamique d'une forme créée peut être mise en correspondance avec des données visuelles et sonores et que ce rapprochement vaut la peine d'être étudié. De façon inattendue, les aspects musicaux ont renvoyé à une approche psychologique de la musique. A savoir les relations que peut entretenir un compositeur avec la langue qu'il s'est créée et le degré d'attachement à ce qu'elle lui donne la possibilité d'énoncer.

2.1 Réalisation d'un logiciel musical

Automate Cellulaire

Le moteur du programme, la forme, repose sur un Automate Cellulaire. Brièvement : un Automate Cellulaire traduit le comportement d'un ensemble d'individus, de cellules, de points, etc. Il repose sur des règles simples exercées localement pour chaque individu en fonction de ses voisins immédiats. C'est le mouvement d'ensemble produit par toute la population soumise aux mêmes lois qui intéresse. Pierre Audibert le décrit ainsi :

> « A chaque étape de temps, tous ces individus évoluent en même temps, en fonction de leur situation locale. Le processus est typiquement parallèle (⋯) soumis à des règles simples dictées par leur voisinage immédiat, ils donnent lieu à des phénomènes complexes et contrastés. Apparitions localisées de formes organisées à partir d'un contexte aléatoire, ou inversement, développement de mouvements désordonnés et complexes qui cependant obéissent à des lois d'ensemble, partout règne l'harmonie des contraires. Cela concerne aussi bien des populations que l'évolution de formes dans le monde naturel, et même les incendies de forêt» [AUDIBERT, 2000, p 27].

Fonctionnement du logiciel « Dialectic »

Dans un plan (l'écran), on répartit des sonorités. Ces sonorités sont des échantillons de

[1] Un *Automate Cellulaire* désigne en réalité une population *d'automates cellulaires*. Nous avons mis des majuscules pour distinguer le plan d'ensemble des individus.

son. Des coordonnées horizontales(x) et verticales (y) leur sont attribuées. En parallèle, chaque position de ce plan est une cellule de l'Automate Cellulaire. Afin de lancer son fonctionnement, quelques cellules sont initialisées (une valeur leur est attribuée pour créer des différences de voisinage et permettre l'activation des règles de transition).

Lorsque, sur ce même plan, le fonctionnement de l'Automate Cellulaire est lancé, le principe est le suivant :

1. L'état de chaque cellule est comparé à celui de ces voisines, et selon les règles en vigueur, pour chaque cellule un nouvel état est calculé et ainsi de suite pour l'ensemble des cellules. Ensuite, le nouvel état d'activation de toutes les cellules est affiché avec des petits cercles de couleur, ce qui permet d'avoir une représentation graphique à l'écran, par pages successives, des mouvements d'ensemble des cellules de l'Automate. Cette succession des pages introduit une troisième dimension qui se traduit par une animation à l'écran. C'est l'historique de la forme.

2. En fonction de la coïncidence entre l'état d'activation de chaque cellule du plan et de l'éventuelle présence d'une sonorité aux mêmes coordonnées, la sonorité est déclenchée ou pas. La lecture de deux échantillons ne peut pas se superposer et la structure sonore est monophonique.

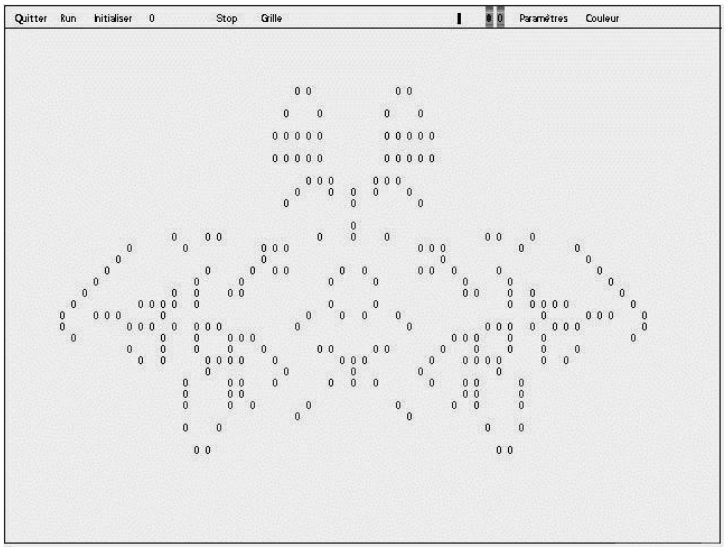


Figure 1.1 : Visualisation de la forme en développement

Dans son principe, l'Automate est considéré comme un « générateur de cohérence ». Sa forme, son fonctionnement sont une organisation indépendante des données. Ni les sons ni les éléments graphiques ne déterminent un ordre quelconque. La cohérence du moteur qui fonctionne ordonne les données et produit une structure dans le temps et dans l'espace restituée de façon visuelle et sonore. Le choix des sons et le choix de la représentation graphique jouent cependant un rôle important dans le rendu final.

La mise en oeuvre du programme, l'observation et l'expérimentation de son fonctionnement ont provoqué une réflexion musicale spécifique, une nouvelle idée de ce que peut signifier faire de la musique.

2.2 Conséquences sur l'approche musicale initiale

L'idée que nous avions pour le logiciel Dialectic a renvoyé dans un premier temps à l'image que nous pouvions avoir de la musique. Le concept global que nous avons retenu pour l'aspect musical est une adaptation technique originale d'une conception antérieure, de nature visuelle, de la musique. Les sons sont disposés dans un espace comme sur la toile d'un peintre et le temps est fruit du développement de la forme qui les active.

« N'importe quoi »

Mais lors d'une expérience faite avec des échantillons de mots, un décalage du sens s'est produit, et le modèle s'est en quelque sorte échappé. Il n'a plus répondu à ce qu'il était sensé produire. Il y avait sur le plan sonore quelque chose qui n'était plus en rapport avec l'idée que nous avions de la musique. Nous n'avions jamais entendu de telles séquences de sons et d'ailleurs, il eut été impossible de les produire autrement.
Du coup, c'était un peu comme si le logiciel ne répondait pas. Cet effet n'étant pas prévu, il n'y avait évidemment pas de moyen non plus de le controler. Les résultats obtenus échappaient à l'image originelle qui devait les enserrer. Certes, nous aurions pu discréditer le logiciel lui-même « qui fait n'importe quoi ». Mais c'était amusant et intéressant. Il a fallu créer du sens et pour ce faire, modifier notre habitude de la musique et avancer une idée musicale.

Modifier une habitude de cet ordre pour un musicien ne va pas de soi. Nous avons essayé de décrire ce processus dans sa dimension psychologique de cognition en nous appuyant sur les travaux d'Erickson et de Paul Watzlawick, inspirés par la théorie des hémisphères cérébraux [DROUILLON, 1997].

Intérioriser de nouvelles données musicales

L'hypothèse est que la musique est perçue et rendue intelligible par un individu parce qu'il possède en lui-même une idée de la musique. Et cette idée est comme une image qui procède de l' « image du monde » associée à l'hémisphère droit et dont Paul Watzlawick décrit des propriétés dans une perspective thérapeutique [WATZLAWICK, 1980].

A cette image de la musique correspondent des compétences de l'hémisphère gauche, à savoir pour le musicien, la « cérébralité » de *sa* musique, la manière dont il intellectualise *son image* de musique. Une sorte d'instrumentation abstraite, à la fois théorique et pratique, qui fonde tout ce qui entre en compte dans la définition de sa musique, et qui n'a pas d'autre but que de produire et reproduire cette image. Ainsi, quelque soit la complexité des moyens mis en oeuvre, ils iront probablement toujours vers le même but, celui de confirmer l'image initiale. D'où l'intérêt de travailler sur l'image que l'on s'est faite de la musique, ou, ce qui revient au même, sur la manière dont la musique s'intègre, plus généralement, dans notre image du monde.

Pour ce faire, il a été intéressant d'emprunter à l'hypnose, sur la base de travaux d'Erickson cité par Paul Watzlawick, quelques éléments de connaissances. Notamment le blocage de l'hémisphère gauche par saturation avec des informations incohérentes[1]. Le

[1] Erickson noie le patient sous un flot d'explications pseudo-logiques compliquées, obscures et déroutantes : « On pense et tout est relatif mes pensées pour les vôtres et les vôtres pour les miennes et ma chaise est ici et

fonctionnement du logiciel dans la situation particulière créée avec des échantillons de voix produit ce phénomène de blocage, d'où l'impression de « n'importe quoi » et une certaine forme de désarroi qui interdit de percevoir ce qui est en train de se passer effectivement. Ce qui se passe n'est pas un fruit attendu comme pourrait l'être celui d'un développement logique. C'est autre chose qui n'a rien de comparable. D'où l'impossibilité dans un premier temps de l'identifier et de l'intégrer dans une conception élargie de la musique.

Toutefois, après acceptation et compréhension du phénomène, il devient possible progressivement de faire apparaître et de construire une nouvelle image de la musique. Cela revient à faire émerger, pour soi personnellement, un nouveau paradigme musical, une nouvelle idée de ce que peut être la musique, le plaisir d'en faire et d'en écouter. Cela revient à faire évoluer ses connaissances et non s'en tenir à leur développement selon un même paradigme.

La création du logiciel Dialectic a provoqué une telle démarche de connaissance. Programmer un programme de musique nous a renseigné sur la façon dont nous étions nous même « programmé » quant à l'approche et la conception musicale !!

3. Concevoir un espace d'expression

Après cette première approche musicale de la programmation, une autre création d'instrument de musique est tentée, basée cette fois sur un système de vie artificielle. Ce sont « Les notes volantes ». Inspirées du vol des chauves-souris, les notes volantes sont des entités, mâles et femelles, qui entrent en relation les unes avec les autres en fonction de différentes modalités. Elles sont dans un environnement où croissent des moucherons harmoniques et dont elles se nourrissent, ce qui leur donne à la fois une valeur rythmique et harmonique.

L'écriture de ce logiciel a mis en évidence tout un potentiel narratif et le centre d'intérêt s'est ouvert plus largement à la création informatique de fictions. Réaliser un système de vie artificielle revient en effet à faire état d'une vision de la vie. Les possibilités narratives propres à une écriture informatique sont devenues notre principal sujet d'étude et de recherche. En particulier, laissant en arrière plan notre spécialisation musicale initiale, nous avons imaginé le concept de *synthèse audiovisuelle*. C'est une tentative de généralisation pour

---

pour moi la votre est là parce que mon ici est ici et mon là est là et pour vous mon là est votre ici et votre là est mon ici, juste comme pour le temps parce que le même temps est le présent··· ». Dans la perspective thérapeutique il procède ensuite à ce qu'il appelle le saupoudrage, c'est-à-dire des suggestions positives lâchées par bribes pendant le discours abscons. Ces suggestions positives se réfèrent à des structures linguistiques de l'hémisphère droit avec notamment une impossibilité de la négation. Le patient qui commence par essayer de suivre les élucubrations finit par échouer et par abandonner, l'hémisphère gauche est surchargé. Du coup, les suggestions positives sont capturées directement par la moitié droite du cerveau : « Le patient s'efforce de suivre la pseudo-logique du thérapeute et de résoudre le paradoxe qui lui est soumis, mais il échoue, et dès lors l'hémisphère droit saisit l'occasion de traiter ceux des éléments de la communication thérapeutique qui sont traduisibles dans son "langage". »
[WATZLAWICK, 1980, p. 55, 63, 72, 101].

l'élaboration d'un espace d'expression relatif à la programmation orientée création.

Audiovisuel de synthèse

Une synthèse audiovisuelle est-elle concevable en amont des synthèses de l'image et du son et quelles pourraient en être les spécificités ? Dans une perspective informatique classique, envisager l'audiovisuel revient à modéliser l'activité de production audiovisuelle et à proposer des outils pour cette production. Par exemple, créer des touts audiovisuels par éditeurs informatiques interposés. Il est loisible d'imaginer, même si c'est un peu de la science fiction, le fonctionnement d'un tel logiciel de synthèse audiovisuelle. Vous écrivez une histoire dotée de différents paramètres (temps, durée, lieux, couleurs, ambiance, personnages …) et l'application en construit le film tout en sons et images de synthèse articulés. L'augmentation des espaces de stockages, « DVD » et disques durs, et l'accroissement de la puissance des ordinateurs rendent concevable un tel projet de recherche. Ce n'est cependant pas exactement dans cette direction que nous envisageons l'audiovisuel de synthèse.

Qu'est-ce que c'est dans la réalité qu'un phénomène audiovisuel ? Il semble difficile de le saisir en laboratoire et de l'analyser en tant que tel sans faire appel à une histoire qui couple et le son et l'image. C'est la dynamique d'une situation donnée qui coordonne différents types de paramètres dont certains sont sonores et d'autres visuels. Le phénomène audiovisuel apparaît comme un phénomène lié à un sens, à un ensemble. C'est un phénomène construit. Son essence est culturelle, sociale, dépendante de la perception, de la pensée, du regard porté, etc. Un phénomène audiovisuel finalement repose sur une assez grande complexité, et il est directement relié à la conception de modèles qui lui confère de l'intelligibilité.

C'est dans cette direction, autour de la notion de modèle, que nous pressentons de nombreuses préoccupations pour une informatique de création, souhaitée visuelle et sonore. D'où cette première proposition pour une définition de départ à l'audiovisuel de synthèse. *L'audiovisuel de synthèse est une unité transversale de préoccupations pluridisciplinaires et un champ de la modélisation pour la conception de systèmes informatiques dont les résultats sont audiovisuels, éventuellement interactifs.*

Cette définition ne rejette pas la perspective peu réaliste évoquée plus haut mais elle est à la fois plus large et ouverte, c'est-à-dire non limitée à de l'audiovisuel tel que nous le connaissons avec la vidéo. L'objectif est de concentrer une recherche sur ce qui est propre à l'informatique : la création d'organisations, la réalisation de modèles. Ces modèles peuvent être empruntés à différentes disciplines (biologies, mathématiques etc.) ou inspirés par elles et dotés de possibilités interactives.

La dimension narrative est affirmée comme une composante de la modélisation et entre dans la définition de modèle. Pour le mathématicien Nicolas Bouleau par exemple, « le modèle est un récit symbolique ». Dans le cadre de l'audiovisuel de synthèse, cette dimension narrative du modèle peut être extrapolée : faire des modèles non pas de quelque chose mais des modèles qui racontent, en quelque sorte des modèles de rien qui ont du sens, de la signification. Il s'agit de raconter sous la forme d'un programme. L'audiovisuel de synthèse devient la création de formes liées au support informatique et restituées de

façon audiovisuelle. On pourrait parler d'organisations audiovisuelles, « d'organis-audiovisuel ».

Le son et l'image sont générés par un modèle d'organisation. Ils en dépendent et s'y rejoignent. La relation du son et de l'image se fait tout à la fois dans une autonomie et une interdépendance du son et de l'image. Ils peuvent être traités séparément puis réunis dans un traitement de niveau supérieur. La synthèse audiovisuelle pourrait être une conjonction de synthèse du son et de synthèse d'image, mais il n'y a pas d'obligation. Quelque soit le niveau de traitement pour le son et l'image, le plus important est le modèle sous-jacent qui détermine leur imbrication. Le scénario est confondu avec des problématiques d'organisations systémiques sur lesquelles repose éventuellement un principe d'interactivité.

La plupart des jeux peuvent être considérés comme des œuvres de synthèse audiovisuelle sous réserve que le son soit impliqué par l'organisation sous–jacente et non seulement le fait d'une simple juxtaposition illustrative et, dans un sens, passive. Précisons cependant qu'un système audiovisuel classique peut être considéré comme un cas particulier de synthèse audiovisuelle dans la mesure où il y a système avec interdépendance du son et de l'image. L'interactivité n'est pas nécessaire et la linéarité peut être assimilée à un cas particulier de non-linéarité. C'est pourquoi la mise en œuvre purement illustrative du son dans un jeu peut être considérée comme un cas particulier d'audiovisuel de synthèse. Simplement, ce sont des cas pauvres en organisation du point de vue de l'audiovisuel de synthèse.

Chapitre 2

Poser des projets

La création d'une chose, et la création, doublée d'une parfaite compréhension, de l'idée correcte de la chose, font très souvent partie d'un processus unique et indivisible et ne peuvent être séparées sans en provoquer le blocage. Le processus lui-même n'est pas guidé par un programme bien défini, et ne peut pas l'être, car il contient les conditions de réalisation de tous les programmes possibles. Il est plutôt guidé par une impulsion vague, par une « passion ».

Paul Feyerabend.

Dans notre perspective de créativité informatique, le principe du projet est de relier des idées avec la réalisation d'un ou plusieurs programmes informatiques sans qu'il s'agisse à proprement parler de résoudre un problème.

De ce fait le projet est une figure à la fois simple et complexe. Simple parce qu'intuitivement, on voit assez bien de quoi il est question : le projet est un sujet pour la réalisation d'un programme informatique. Et complexe parce que, dans notre optique, le projet est une idée de programme qui peut évoluer, et que le sujet, rarement préconçu, fait souvent l'objet d'une élaboration. Ce que cette pratique de créativité informatique sous-entend d'engagement et d'aspects opérationnels ne repose pas sur un a priori de résolution de problème comme dans l'ingénierie des connaissances. L'idée initiale d'un projet est mise en chantier jusqu'à la réalisation d'un programme qui n'est peut-être pas celui promis par l'idée initiale et la naissance du chantier. Les motivations de l'auteur, la finalité de son comportement sont interrogées. Son travail n'est pas réservé à des buts relevant de croyances du dehors ou d'un principe unique de rationalité. Le projet incorpore quelque chose d'arbitraire et d'imprédictible qui relève de l'individu qui le propose et le conduit. Il fait l'objet d'un cheminement de connaissance. Est en jeu la cohésion et la consistance du cheminement. La démarche de connaissance qui correspond à celle du projet est ainsi par définition très ouverte. Sans minimiser la rigueur nécessaire à cet engagement scientifique, nous pouvons dire « tout est bon » selon la formule de Paul Feyerabend [FEYERABEND, 1979].

1. Connaissance par projet

Le terme de « connaissance » associé à l'informatique nous emmène vers l'ingénierie des connaissances, elle-même reliée au domaine de l'Intelligence Artificielle (IA). Il nous rapproche de ce domaine. Mais également, nous allons nous en distinguer. Certes, nous partageons avec l'intelligence artificielle des interrogations sur la notion de « connaissance » et le moyen de la relier à l'information dans son acception rigoureusement technique. La raison est l'objectif commun d'implémenter en machine des idées à partir de connaissances soumises à modélisation. En revanche, dans le principe du projet que nous proposons, cette notion de connaissance n'est pas animée par l'objectif de « résolution de problème ». Le projet fait sens, mais il n'a aucune vocation utilitaire à résoudre quoique ce soit.

1.1 Projet ou résolution de problèmes ?

Un projet n'est pas un problème à résoudre. La nature d'un projet n'est pas de conduire à la résolution de quelque chose, la nature d'un projet est de conduire à une réalisation. Bien entendu, des problèmes vont se poser avec la réalisation : comment faire pour obtenir ceci ou cela ? L'intérêt se porte alors sur la possibilité de soulever et d'inventer un ou plusieurs

problèmes, probablement assez localisés lors de la réalisation du projet. Mais le projet lui-même n'est pas réduit à cette expression de « résolution de problème » dans sa connotation de culture scientifique et industrielle.

Par exemple, « Le Kandoscope » est un programme qui est à l'origine un travail sur la mobilité. A la base, on a des objets de formes simples, ronds, carrés, triangles, et de tailles variables. Ils sont assujettis chacun à une trajectoire. En jouant sur les paramètres de chacun des objets et sur des groupes d'objets simultanément, on obtient des images et mouvements d'ensemble intéressants. L'idée est venue alors de faire un tableau « métamorphique » à partir de ce générateur. Le souhait est d'obtenir un tableau avec des possibilités de voyage dans le tableau par métamorphoses de figures d'ensemble, c'est-à-dire de saisir dans un premier temps par tâtonnements des figures d'ensemble satisfaisantes, puis de trouver le principe d'une métamorphose des ensembles afin de passer d'une figure à une autre et d'explorer au passage d'autres agencements.

Nous avons réussi à réaliser des agencements et faire des ensembles très intéressants pour nous sur le plan esthétique, mais selon ce même point de vue, un principe de métamorphose qui permettrait de passer de l'un à l'autre n'est pas encore au point. Soulignons que cette idée de métamorphose correspond également à l'évolution temporelle du programme. A la différence du cinéma et de la vidéo, " l'image d'après" n'existe pas dans un programme informatique. Il n'y a pas de trame temporelle fixe comme une bande magnétique sur laquelle s'appuyer. L'image informatique est un robot en exercice, un automate en action. Elle fait l'objet d'un calcul réitéré et sur lequel il est possible d'interagir. L'image est générée de façon endogène par le système créé et pour lequel il a fallu concevoir la dimension temporelle. Le principe de tableau métamorphique aborde la difficile question de la modélisation du temps dans un programme informatique.

Cette situation de questionnement n'est pas à proprement parler une situation de résolution de problème car il n'y a pas de solution. C'est l'émergence d'une problématique et d'un espace de recherche. Mais l'efficacité recherchée n'obéit pas à des objectifs précis, clairement déterminés. Il y a quelque chose d'énigmatique mais aussi de ludique dans la situation créée. Ainsi, *se poser problème,* dans ce sens d'énigme, est-il plus important pour la démarche de création que la recherche d'une solution qui résolve la question. Se poser problème ouvre un espace de propositions dans lequel un sentiment esthétique et de curiosité pousse à développer et choisir des énigmes. L'esthétique répond ici à un développement et à un choix d'énigmes.

Un projet ne résout rien certes, mais il a un sens. Le sens du projet est au dessus des objectifs. Les objectifs sont à découvrir, à explorer. Ainsi le concept de projet peut intégrer une dimension de recherche et d'évolution imprédictible pour la définition des objectifs. Les objectifs ne sont pas automatiquement orientés et contraints dans une optique de rationalisation et d'automatisation des décisions, de résolution de problèmes généraux de l'entreprise et des organisations par exemple. Le sens dans le projet correspond à un « état d'esprit » dans des termes proches de ceux de Pierre Schaeffer à propos des objets sonores et musicaux et l'on y retrouve un phénomène de « résonance intérieure » décrit par Wassily Kandinsky à propos de la peinture.

Or, sur le plan technique et scientifique, la distance n'est pas impossible avec la notion de connaissance telle qu'elle est proposée en ingénierie des connaissances. Guy Caplat définit

la connaissance comme une attribution de sens à de l'information. Notamment, « La connaissance est présente quand résonne l'information, i.e. quand elle trouve sa place dans le tissu cognitif de l'agent mais également lorsque par sa capacité de 'raisonnance', elle est mobilisée dans une dynamique··· » [CAPLAT, 2002, p.44].

1.2 Connaissance contrainte à l'information

Dans le vocabulaire philosophique, « connaissance » désigne à la fois l'acte de connaître et la chose connue, le contenu de la connaissance. Pour « connaître », Andrée Lalande donne cette définition : « avoir dans l'esprit un certain objet de pensée non seulement en tant que donné, mais en tant que bien saisi dans sa nature et ses propriétés » [LALANDE, 1993, p.171, 172]. Ainsi, le terme « connaissance » va de la faculté de se représenter un objet, réalité perçue, au fait de le comprendre, ce qui touche alors à son explication et à son interprétation. Constater, comprendre, concevoir, percevoir, etc. sont des actions que l'on peut affilier à la connaissance. Lorsqu'il s'agit du contenu de la connaissance, selon cette définition, on trouvera souvent « connaissances » au pluriel : les connaissances humaines etc.

Le domaine de l'intelligence artificielle en ingénierie des connaissances ajoute à cette optique de la connaissance un objectif. C'est *la modélisation des connaissances mobilisées par l'être humain dans le cadre de processus de résolution de problème. Il s'agit de parvenir ensuite à automatiser la résolution de problème, avec la traduction des modèles obtenus en programmes informatiques.*

Il y a là une contrainte majeure. L'informatique est définie par le traitement automatique de *l'information*. L'information est inscrite dans le prolongement du « traitement de l'information » et des techniques associées pour ainsi dire comme un synonyme. Elle correspond à des techniques qui reposent sur l'utilisation d'ordinateurs. Aussi contribue-t-elle à définir la connaissance de façon à ce qu'elle entre dans la technique. En ce sens restrictif, modéliser des connaissances comprises au sens large afin de réaliser un programme exécutable par une machine, signifie opérer un passage entre ces connaissances et l'information, un passage entre la fluidité, la continuité, la subjectivité de la connaissance humaine et la rigidité formelle, l'objectivité de la technique et de la machine. Cependant, l'enjeu est d'adjoindre à la connaissance la technique et non de réduire la connaissance à l'information. Très largement, c'est l'étude des transactions possibles entre connaissances et formalismes.

En 1950 apparaît l'intelligence artificielle, nouvelle discipline dont la volonté est de « faire faire aux machines ce qui, autrement, requerrait une intelligence humaine » [GANASCIA, 1990, p.33]. Un principe est de concrétiser la connaissance, d'en faire une donnée objective dont les éléments constitutifs peuvent être connus et représentés, et les mécanismes relationnels intrinsèques qui les unissent, clairement identifiés et automatisés. La connaissance toutefois apparaît comme un phénomène complexe qui ne se laisse ni réduire ni fixer dans une forme. Plus précisément, quelque soit la forme que l'on puisse donner à une connaissance, cette forme fera toujours l'objet de l'interprétation de ceux qui lui seront confrontés. La forme donnée à une connaissance ne contient pas en elle-même les principes de son interprétation et de sa signification, c'est-à-dire qu'elle ne contient pas ce qui fait d'elle une connaissance [CAPLAT, 2002, p.23].

C'est pourquoi, plutôt que de se demander ce qu'est la connaissance afin de la représenter et qu'elle devienne information, il est intéressant à l'inverse de s'interroger sur ce qu'il faut ajouter à l'information pour qu'elle devienne connaissance. Pour ce faire Guy Caplat définit la notion de connaissance par un aspect sémantique rendu solidaire d'un aspect informationnel qui provoque perception et interprétation.

1.3 Connaissance prise au jeu de l'information et du sens

La connaissance résulte de la perception d'une réalité et de l'interprétation qui en est faite, à savoir une attribution de sens : « Lorsque l'humain acquiert la connaissance d'une chose, il construit en lui une image de cette chose, il sémantise en interprétant l'objet perçu ». L'information, dans sa composante de message et de support de message, peut être considérée comme le vecteur de simples données soumises à interprétation : « La connaissance est une manière de s'approprier un objet, de transformer l'information perçue à travers sa manifestation en quelque chose de porteur de sens. » [CAPLAT, 2002, p.25].

Ainsi, une information est perceptible du fait de sa manifestation. Elle déclenche une interprétation qui fait accéder au niveau de la connaissance. Et en conséquence, la connaissance peut être définie comme une attribution de sens à des informations. L'attribution de sens est liée à un processus d'interprétation et les informations qui en font l'objet doivent être rendues perceptibles du fait de leur manifestation. Définition que Guy Caplat résume par la formule :
« Connaissance = Information / manifestation + Sémantique / interprétation »

Cependant, d'où vient le sens attribué à l'information perçue ?
De la connaissance elle-même, répond Guy Caplat :« La connaissance est sens et le sens ne peut opérer sans connaissance » [CAPLAT, 2002, p.27]. En effet, il y a une dynamique de la connaissance. La connaissance est à la fois dans l'information « sémantisée » qui a un sens une fois interprétée, et dans le processus d'interprétation qui donne un sens. *Un effet de résonance est déclenché par l'information sur les ressources en connaissances de celui qui reçoit l'information. Cet effet de résonance est à la base de l'interprétation qui transforme la simple information en connaissance.* Plus largement, cette capacité de résonance de l'ensemble référentiel des connaissances de celui qui reçoit est aussi une capacité d'extrapoler l'information et de produire de nouvelles informations par le jeu des propositions qui peuvent être déduites (figure 2.1) :

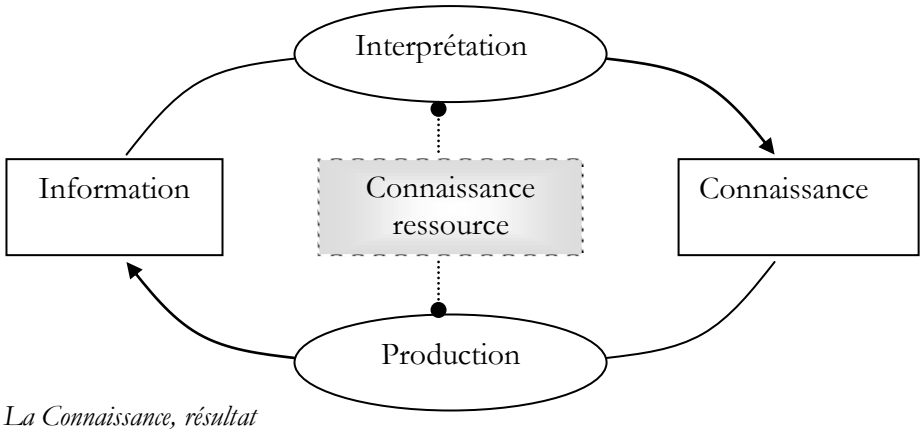


Fig. 2.1 : La Connaissance, résultat d'une interprétation de l'Information et également productrice d'Information

Connaissance et Information sont liées tant pour l'interprétation que pour la production. Mais l'existence de cette relation n'est pas une raison de limiter la définition de la Connaissance à cette relation, comme l'indique la nécessité de Connaissance ressource. D'où un double jeu de la connaissance au sein du projet.

1.4 Double jeu de la connaissance au sein du projet

Au sein d'un projet, il y a un objectif limitrophe de celui cité plus haut pour l'Intelligence Artificielle et l'ingénierie des connaissances. Cet objectif est *la réalisation de modèles à partir de connaissances mobilisées par un être humain dans le cadre de processus de création de fictions, et avec la traduction des modèles obtenus en programmes exécutables par une machine.*

Ainsi, toute la décomposition et l'analyse de la connaissance qu'opère Guy Caplat nous intéresse beaucoup. Elle vise à faire entrer de la connaissance dans un cadre formel adéquat à la réalisation d'un exécutable sur machine. Le projet nécessite un passage entre des connaissances et des techniques de traitement de l'information. Mais nous divergeons quant à la manière de se positionner pour l'établissement d'un domaine de connaissances et des actions à opérer dessus.

Dans le cas de la résolution de problèmes, des connaissances de nature variées sont regroupées à partir du problème posé et elles sont orientées en fonction des solutions recherchées. Dans le cas du projet que nous proposons, le regroupement des connaissances progresse en fonction d'une sorte d'inspiration qui est liée à un sujet initial. Par exemple, dans le cas des tableaux métamorphiques cités plus haut, la question des métamorphoses de figures d'ensemble n'existe pas au début du projet. Au début de ce projet, il y a juste l'étude de trajectoires affectées à des formes simples en nombre et de tailles variables. En quelque sorte, le domaine de connaissance propre au projet ne préexiste en aucune manière au projet. Son élaboration nécessite de vivre le projet. Simultanément à la constitution d'un domaine de connaissance propre au projet, il y a constitution, création, élaboration d'un sens. Le premier point pour nous est la création d'un sens qui est aussi une motivation qui fonde le projet psychologiquement.

Ainsi avons-nous introduit une définition philosophique au début de ce propos sur la connaissance. En effet, nous proposons de conserver davantage son indépendance à la connaissance par rapport à une trop haute spécialisation au domaine informatique investi de la mission de résoudre des problèmes.

Dans le cadre du projet, une exigence est ajoutée pour la connaissance. Non seulement elle doit exister dans la relation qui unit connaissance et information pour la réalisation d'un programme informatique, mais également elle est dans le principe de création d'une action et d'un projet qui relient au réel. Ainsi, pour revenir à la formule de Guy Caplat, « Connaissance = Information + Sémantique », nous adjoignons le principe créatif à cette définition et remplaçons l'égalité par une implication. Ce qui donne pour le projet :
Connaissance => (Action+Sémantique) ET (Information+Sémantique)

En conséquence, si le projet est légitimé par la création de programmes informatiques, il n'est toutefois pas limité à cette vocation du fait de cette composante d'élaboration des

modalités d'action et d'engagement. Ce double jeu de la connaissance au sein du projet peut susciter d'autres productions autour du dynamisme inféré par l'idée et la réalisation d'un programme. Par exemple, le projet peut engendrer des situations dans lesquelles le programme trouve une place. Expositions, rencontres, spectacles, débats etc. peuvent faire partie de la dynamique du projet et de la réalisation de programmes. Certaines situations orienteront le projet vers de nouvelles acquisitions de connaissances.

2. Fonctionnement d'un projet

Comment s'organise la créativité ? Y a t-il une méthode pour être créatif ou bien la créativité s'exerce-t-elle mystérieusement par le fait d'un don magiquement reçu ? Il est certain que manière d'être et créativité réagissent l'une à l'autre. Citons par exemple l'opposition entre habitude et originalité présentée (avec humour) par Arthur Koestler :

> « Les habitudes forment l'indispensable noyau de la stabilité et du comportement ordonné ; elles ont tendance aussi à se mécaniser et à réduire l'homme à l'état d'automate. L'acte créateur, en reliant des dimensions d'expérience jusque là étrangères l'une à l'autre, lui permet de s'élever à un niveau supérieur d'évolution mentale. C'est un acte de libération : l'habitude succombe à l'originalité » [KOESTLER, 1964, p.82].

Pour Paul Feyerabend, à l'origine de la création, il y a un élan, une passion. Et cette passion « donne naissance à un comportement spécifique, qui à son tour crée les circonstances et les idées nécessaires à l'analyse du processus, à son explication, et à sa « rationalisation » [FEYERABEND, 1979, p.24]. Dans cette optique, sans entrer dans les mécanismes complexes qui fondent la créativité, nous allons essayer d'examiner sous un angle pragmatique comment s'articule le dynamisme interne d'un projet et dégager quelques grands axes de son fonctionnement.

2.1 Trois aspects principaux entrent dans la définition du projet

Nous dégageons trois aspects principaux qui composent le projet : une idée initiale, un espace de travail, une tension vers la réalisation. L'idée initiale a une valeur psychologique qui fonde intérêt et motivation pour le projet. Elle déclenche des associations d'idées, motive une certaine forme d'attention ainsi que la réalisation de quelques programmes de tests. Autour d'elle s'agrègent des informations et des connaissances. Le projet s'élabore progressivement comme un « filtre perceptif ». Avec lui sont organisées la captation et la réception d'informations et de connaissances dans un processus d'élargissement et de développement des principes initiaux. Il devient un espace de préoccupations et toutes ne sont pas visibles de prime abord. La réalisation effective du projet peut même être éloignée de sa formulation initiale. Toutefois, la notion de projet est orientée par une tension vers la réalisation de quelque chose. Cette tension est un ensemble psychologique de motivations, d'intérêts et de désirs qui sont personnels, éventuellement objets de négociation dans une équipe. Cette tension suit et anime l'évolution du projet et conserve une identité, même lorsqu'il est amené à se transformer.

Nous proposons la définition très large suivante pour le projet :
Le projet est le chantier de recherche et de réalisation associé à une idée initiale en vue de la réalisation d'un programme informatique. Par « chantier », il faut entendre à la fois le regroupement de données réelles, informations, connaissances, références, exemples etc. qui existent en dur et sont stockées quelque part et le dynamisme psychologique qui anime ce chantier. Eventuellement, un projet peut donner naissance à d'autres projets. Ce peut être par exemple la perspective de suivre un autre cheminement pour le projet.

2.2 Relier des idées à la conception d'un programme

La liaison entre une idée ou un ensemble d'idées et un programme avec son code source appelle plusieurs types de transitions. La démarche peut être globalement descendante des idées vers le programme, ou globalement montante du programme aux idées, et une boucle de rétroaction peut se constituer qui fasse évoluer l'idée initiale et avec elle la définition et la réalisation du programme cible (figure 2.2)

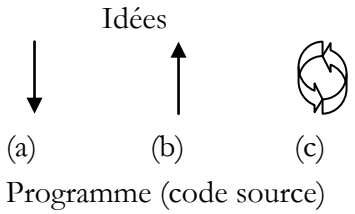


Fig. 2.2 : Modalités de relations entre idées et programme qui induisent différentes approches de réalisation. (a) descendante, (b) montante, (c) rétroaction

La démarche est descendante lorsque, à partir d'une idée initiale, ne disposant d'aucun exemple de programme ni d'aucun extrait de code source, il s'agit de descendre par étapes successives jusqu'à la réalisation du codage. Il s'agit d'un glissando, de l'idée au programme, proche de la démarche classique de l'analyste-programmeur[11].

La démarche est montante lorsque des bouts de code source existants, des exemples de programmes, donnent des idées de programmes ou de fonctions dans le programme. Ces exemples de code sont intégrés au développement ensuite, moyennant quelques modifications. Cette démarche repose sur la possibilité de redonner sens à un bout de programme existant ; c'est en effet une propriété du code source. Un extrait de code source dans sa forme syntaxique peut être intégré dans des programmes différents pour ce qu'il peut faire dans les mêmes termes de son programme d'origine. On peut citer par exemple un certain nombre de séquences typiques de code, comme ouvrir ou créer un fichier et stocker dedans des résultats, désignées par le terme générique de paradigmes

[11] Généralement cinq grandes étapes comme suivent :
- Le scénario : on se fait le « film » de la situation où devra s'insérer le programme (les acteurs, leurs interactions, la connaissance du milieu et les motivations etc.)
- Le cahier des charges où sont consignés, avec précision cette fois, les objectifs et buts à atteindre. On fait tourner le futur programme en langage naturel.
- L'analyse fonctionnelle, soit toutes les fonctions à mettre en œuvre, globalement.
- L'analyse détaillée, le détail des fonctions.
- Le codage, la programmation.

[CLANCY, 1993, p.17]. Mais il est possible également de lui donner un sens différent, de l'interpréter autrement. La démarche montante joue beaucoup sur la variabilité du contenu sémantique d'extraits de code source ainsi que l'atteste l'exemple du « Brodock à plumeau » ci-dessous avec le réinvestissement sémantique d'un extrait de code source détourné de son sens initial.

La rétroaction alterne les deux démarches précédentes. Une idée permet de réaliser du code source et un programme (aspect descendant). Par ailleurs, un exemple de programme avec son code source est intéressant (aspect montant). Il est trouvé sur Internet, auprès d'amis etc. Ce que sait faire le code source de cet exemple, ou une partie seulement, peut enrichir ou infléchir dans un nouveau sens l'idée du programme, ou de certains aspects du programme. Ce code source va alors être intégré, après quelques modifications adéquates, à la réalisation.

Par exemple, le « Brodock à plumeau » est un projet de programme informatique. Il est né de la rencontre avec un cirque, le « Cirque 360 », qui développe une ménagerie de robots. Dans ce cadre, l'idée initiale très générale a été de proposer une galerie de « logiciels monstrueux » afin de les présenter comme on présentait autrefois des monstres dans des foires. Le principe du programme informatique pour le Brodock est né du *réinvestissement sémantique* d'un travail sur la réalisation d'un tore extrait d'un cours de programmation de Pierre Audibert [AUDIBERT, 2000, p.196]. Le Brodock à plumeau est devenu une sorte de beignet plus ou moins boursouflant et à poils rétractiles. Dans le cadre d'une interaction, les poils ont la propriété de se hérisser subitement ou au contraire de se résorber comme les filaments d'une anémone de mer. De plus, la pointe d'un poil hérissé peut exploser. Le sens de cette explosion dans l'identité du Brodock n'est pas encore clairement établi, c'est une ouverture possible pour un développement de l'histoire et de la description du Brodock. Cette dernière particularité est venue avec un extrait de code source qui permet de simuler un effet de feu d'artifice. Elle s'est jointe à l'identité du projet ainsi qu'à la réalisation du programme par pure fantaisie, selon l'avantage esthétique qu'elle peut apporter et l'attrait présenté par le codage.

L'intérêt de repérer ces trois types d'opérations qui relient idées et programmes est que le processus de réalisation du projet, du point de vue de la création que nous proposons, n'est pas systématique. Il est possible d'adopter une démarche classique descendante lorsqu'il y a possibilité d'identifier clairement les objectifs dès le départ. Mais le projet peut également faire l'objet d'un cheminement et d'un parcours ponctués de nouvelles préoccupations. Dans ce cas, la réalisation du projet passe par différentes étapes et ces opérations, qui vont des idées au codage et du codage aux idées en passant par des acquisitions de connaissances et d'informations, vont s'enchaîner dans le temps selon un processus qui n'est pas aisément repérable. Cette décomposition en mouvements simples, descendre, monter, monter et descendre, accepte une progression en étapes déterminées par des apports de connaissances ou d'informations. Ce principe est solidaire de la souplesse requise à la conception pour une exploration créative. Ainsi le programme Brodock est-il un peu travaillé comme le personnage d'un roman. On se demande qui il est et comment il va réagir dans telle ou telle situation. Progressivement sa personnalité s'élabore et avec elle évolue la définition du projet et la réalisation du programme.

Cette forme de modularité de la conception qui tolère une évolution du projet se retrouve dans la méthodologie de programmation. Elle se retrouve aussi du point de vue des

informations et connaissances collectées et mises en relation. Deux aspects apparaissent : le premier est la localisation du projet du point de vue des connaissances en général, c'est-à-dire l'apparition au début d'un sujet qui ancre le projet et intervient dans sa définition. Le second correspond au développement d'un domaine de connaissance propre au projet après initialisation avec son point d'ancrage.

2.3 Constitution d'un domaine de connaissance propre au projet

Début et sujet d'un projet sont liés à l'idée initiale. Mais l'idée initiale est presque toujours précédée d'un « potentiel d'idée initiale ». Souvent, ce sera une formule qui agit comme un titre et qui fait que l'on se pose la question « qu'est-ce que ça pourrait être ? ».

Par exemple, qu'est ce qu'on peut faire comme programmes à partir des titres suivants ?
Les administrés de la catastrophe, La mouche de Grattécollé, Les nénuphs d'Algor, Un rêve de trente ans, Pour une vie de champignon (roman programmé d'une amybe), Bouddha neuromimétique (mandalas neuronaux), Troupeau de Pirlufs et social-mécanique, L'antiquaire du veau d'or, Le programme des programmes, Bination complexe-bination réplexe (de reflex), les runes électriques, Les agrumes quotidiens, petites pièces informatiques…

En fait, le potentiel d'idée initiale devient idée initiale en même temps qu'est opéré un rapprochement avec « quelque chose » qui concrétise, enracine dans un domaine ou agit comme un attracteur de connaissances. Par exemple Le Brodock, est premièrement relié à l'idée d'un logiciel monstrueux. Cette première liaison est attractive. Elle attire le principe d'un être et cet être va prendre la forme d'un tore. Avec cette concrétisation, reliée à un cours d'informatique et donc un domaine de connaissance, le Brodock devient une idée initiale. Ensuite, le développement du chantier associé au projet se met en place et s'ajoute. Apparaît alors le principe des poils rétractiles à explosion et le Brodock devient le Brodock à plumeau. Notons au passage que le principe de logiciels monstrueux peut faire l'objet de différentes interprétations et attirer d'autres liaisons. Dans un état d'esprit totalement différent, Patrick Greussay a orienté l'attention sur les Quines[12] et le langage de programmation BrainF[13]. Mais quelque soit l'interprétation qui est faite, elle connote le potentiel d'idée initiale et en réduit l'ouverture. Elle l'attire dans un sens en particulier ou dans une exploration spécifique.
Ainsi, par définition, une propriété importante de l'idée initiale est de relier le projet à un

[12] Un « quine » est un programme informatique qui s'imprime lui-même, c'est-à-dire qui imprime le listing de son propre code source, y compris les instructions d'impression, alors même qu'il fonctionne et selon l'opinion de David Madore : « This may sound either impossible, or trivial, or completly uninteresting, depending on your temper and your knowledge of computer »
[www.eleves.ens.fr:8080/home/madore]A l'origine le concept est introduit par le mathématicien et logicien Willard van Orman Quine.

[13] Créé par Urban Müller qui l'a écrit à l'origine pour Amiga, le BrainF est un langage de programmation qui dispose uniquement de huit commandes. Chaque instruction est représentée par un simple caractère : > < + − . , [ ]
Voici l'exemple traditionnel du premier programme qui imprime « Hello World! » écrit en BrainF :

```
>++++++++++[<+++++++++>-]<.>++++++++[<++++>-
]<+.+++++++..+++.>>>++++++++[<++++>-]
<.>>>++++++++++[<++++++++++>-]<---.<<<<.+++.------.--------.>>+.
```

bluesorcerer.net/esoteric/bf.html et esoteric.sange.fi/brainfuck/

contexte général de connaissance. Et le projet, qui est le contexte d'élaboration du programme constitué à partir de l'idée initiale, délimite ensuite, selon un dynamisme interne, l'action du point de vue de la connaissance en général. Au commencement du projet, les connaissances qui serviront son développement ne sont pas nécessairement clairement identifiées, d'autant que les besoins en connaissance du projet vont suivre son évolution. Des éléments appartenant à des horizons différents et sans relation évidente se trouvent regroupés lors de la réalisation de certains projets. C'est le cas du Brodock lors de la liaison avec les plumeaux. C'est également le cas dans le projet de « La Vénus du mythe des races métalliques » qui réunit vie artificielle et mythologie grecque.

Un contexte de connaissance locale au projet va se constituer du fait des orientations sous-tendues par l'idée initiale qui sont ensuite reprises et développées dans des modalités de fonctionnement et de dynamisme intrinsèque au projet (figure 2.3). Un domaine de connaissance propre au projet est créé. Il s'élargit et s'oriente progressivement en fonction de la vie du projet.

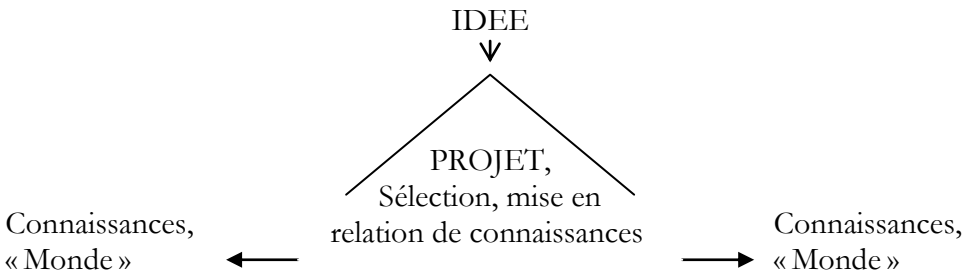


Fig. 2.3 : L'idée initiale positionne le projet qui engage des connaissances et l'élaboration d'un espace de travail en vue de la création d'un programme.

Pour illustrer notre propos voici l'exemple du programme « La Vénus du mythe des races métalliques ». Cet exemple illustre bien le développement d'un projet particulièrement riche, et dès son origine qui est le fait d'un projet antérieur, « Chauves-souris ».

3. Exemple de projet : « La Vénus… »

3.1 Début

Les chauves-souris

Dans le chapitre précédent, nous avons évoqué le programme « Les notes volantes ». A la racine de ce programme, il y a un projet issu de lectures sur la vie artificielle. L'idée initiale est de représenter des chauves-souris qui puissent se mouvoir et entrer en communication les unes avec les autres avec éventuellement l'idée d'en faire un instrument de musique. Ce projet initial est baptisé « Chauves-souris ». Il provoque une expérience intime assez forte de la programmation. Le sentiment extrêmement clair de raconter une histoire à travers la définition de ces entités et l'imagination de leurs relations. Dés lors, le projet se développe

dans deux directions. L'une reprendra un an plus tard le projet initial d'instrument de musique pour faire « Les notes volantes », l'autre s'intéresse au principe de la narration. C'est un nouveau projet, « La Vénus du mythe des races métalliques ». Il est tourné au départ vers des éléments de culture théâtrale.

3.2 Développement

La vie artificielle et le théâtre

Jean-Claude Heudin donne de la vie artificielle la définition suivante : « La vie artificielle repose sur l'abstraction des propriétés caractéristiques des êtres vivants, puis sur leur implémentation sur des supports arbitraires (...) Dans ce cadre, l'ordinateur permet des réalisations abstraites sous forme d'algorithmes. » [HEUDIN, 1994, p.24]. Il nous est apparu que finalement, au point de vue de l'informatique, un projet de vie artificielle revenait à raconter une vision de la vie et, précisément, qu'il serait intéressant de raconter une histoire à travers un système de vie artificielle.
Dans un premier temps, nous nous sommes tournés vers la culture théâtrale « sans but précis », de façon totalement intuitive et par pure curiosité. L'idée était d'aller vers ceux dont le métier est de raconter et de mettre en scène des histoires afin de mieux les connaître et de s'en inspirer.

La poétique structurale

Cette réceptivité au théâtre, avec en arrière-plan une préoccupation à propos de la narration, a débouché sur le mouvement structuraliste de la « poétique ». Son objet est la recherche d'une grammaire narrative universelle : « Par opposition à l'interprétation d'oeuvres particulières, la poétique ne cherche pas à nommer le sens mais vise la connaissance des lois générales qui président à la naissance de chaque œuvre. » [TODOROV, 1968, p.19].
Outre des auteurs comme Algirdas Julien Greimas, Gérard Genette, Claude Lévi Strauss, Tzvetan Todorov, nous nous sommes plus particulièrement intéressés à Vladimir Propp et Claude Brémond. Mais notre objectif n'est pas d'entrer dans la problématique littéraire structuraliste ; le souhait est de s'en servir, d'extraire des éléments de modélisation pour réaliser un programme.
L'idée est alors d'obtenir des objets dotés d'une autonomie, d'un comportement cohérent non aléatoire, soumis à une structure narrative déterminée mais avec des développements historiques imprédictibles dans le temps du fait des interactions de ces objets entre eux et éventuellement avec un utilisateur.

Système multi-agents

Voici une description d'un agent :

> « Un agent est une entité capable d'agir sur son environnement et qui, dans un univers multi-agents, peut communiquer avec les autres par envoi de messages. (⋯) Le comportement d'un acteur est la conséquence de ses observations mais aussi de ses capacités sociales telles que par exemple, raisonner sur autrui (exploiter les connaissances et les capacités des autres, anticiper et influencer leur comportement), coopérer, partager des ressources et des objectifs⋯ » [CAPLAT, 2002, p.20].

Transposé dans un univers de fiction, l'objectif de « La Vénus ··· » est de mettre en scène un système multi-agents audiovisuel de marionnettes booléennes pour l'animation simultanée et corrélée de structures sonores et d'images [DROUILLON, 1998, p.F2-1].
A ce stade de développement du projet, curieusement, le souhait du point de vue de la dramaturgie est une expression qui rejoigne la mythologie. C'est là qu'intervient la découverte du Mythe des races métalliques de Hésiode.

Mythe des races métalliques

La charge du mythe des races métalliques est de connoter les relations entre « marionnettes », agents ou entités. Mais cet aspect nécessite une interprétation du mythe lui-même. Dans le récit d'Hésiode, les différentes races d'or, d'argent, de cuivre, des héros et de fer correspondent à des générations successives de l'humanité et se succèdent dans le temps, chacune imposant au monde ses propres qualités. Tristement d'ailleurs, il s'agit d'une dégradation avec une aggravation progressive des maux qui étreignent l'humanité. Dans le programme Vénus, l'intérêt dramatique est de les confronter comme si toutes se retrouvaient dans un même temps. Il faut entreprendre une réflexion qui articule justice (diké), démesure (hybris) et discorde (éris). Quoique non programmée, une recherche à ce sujet est alors amorcée, notamment à partir des travaux de Jean Pierre Vernant [VERNANT, 1996].

3.3 Fin

Ce qui a été effectivement réalisé se traduit par des sortes de marionnettes dotées d'une autonomie cohérente, avec des déplacements curieux, très différents qualitativement de ceux qu'occasionnerait une simple fonction aléatoire.

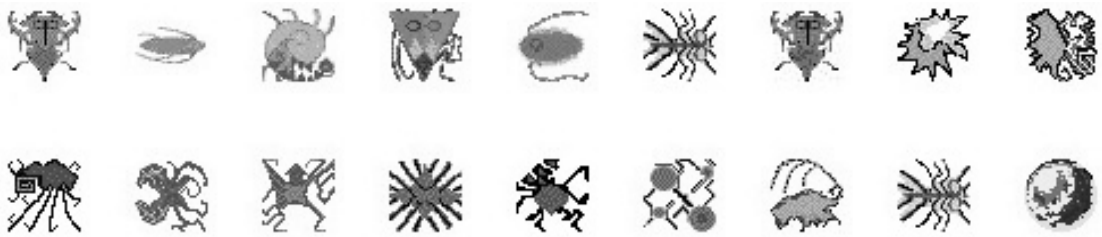


Fig. 2.4 : Les marionnettes booléennes

Elles se meuvent et interagissent avec leur environnement, déclenchent des sonorités et agissent sur le signal sonore (panoramique, fréquences, amplitude) en fonction de leur état intérieur. Cet état est soumis aux interactions avec les autres et chacune conserve la mémoire de ce qu'elle a déjà fait dans telle ou telle situation et qui puisse intervenir dans ses choix successifs. D'une certaine façon, l'identité de chaque marionnette se construit au fur et à mesure des rencontres lors desquelles elles acquièrent des réflexes comportementaux.
Par ailleurs, à la demande des organisateurs du festival de cyber culture Artoïd 98 à l'occasion duquel le programme a été présenté, un second moteur a été greffé. Il s'agit d'un petit générateur automatique de discours coton. Des textes en langue coton se succèdent et sont lus par un synthétiseur vocal. Ils contribuent à la mise en scène générale en tant que décor. Il reste de nombreuses pistes à explorer !

La découverte et l'acquisition de connaissances nouvelles ont été particulièrement importantes pour la réalisation de ce programme. Il y a eu une sorte d'enquête, un « filet de pêche » pour l'acquisition de connaissances ; en fait, un travail de recherche. L'enquête fait évoluer la définition du projet et l'idée du programme. Mais à un moment donné, une interprétation s'impose et un certain nombre d'objectifs à atteindre avec elle. Une mise en forme telle qu'elle puisse faire l'objet d'une programmation commence à se mettre en place. C'est le phénomène de modélisation qui va permettre d'articuler le sens du point de vue de l'action au sein du projet avec le sens tel qu'il peut être supporté par des informations au sein du programme. Le modèle apparaît comme un moyen d'articuler du sens à une forme.

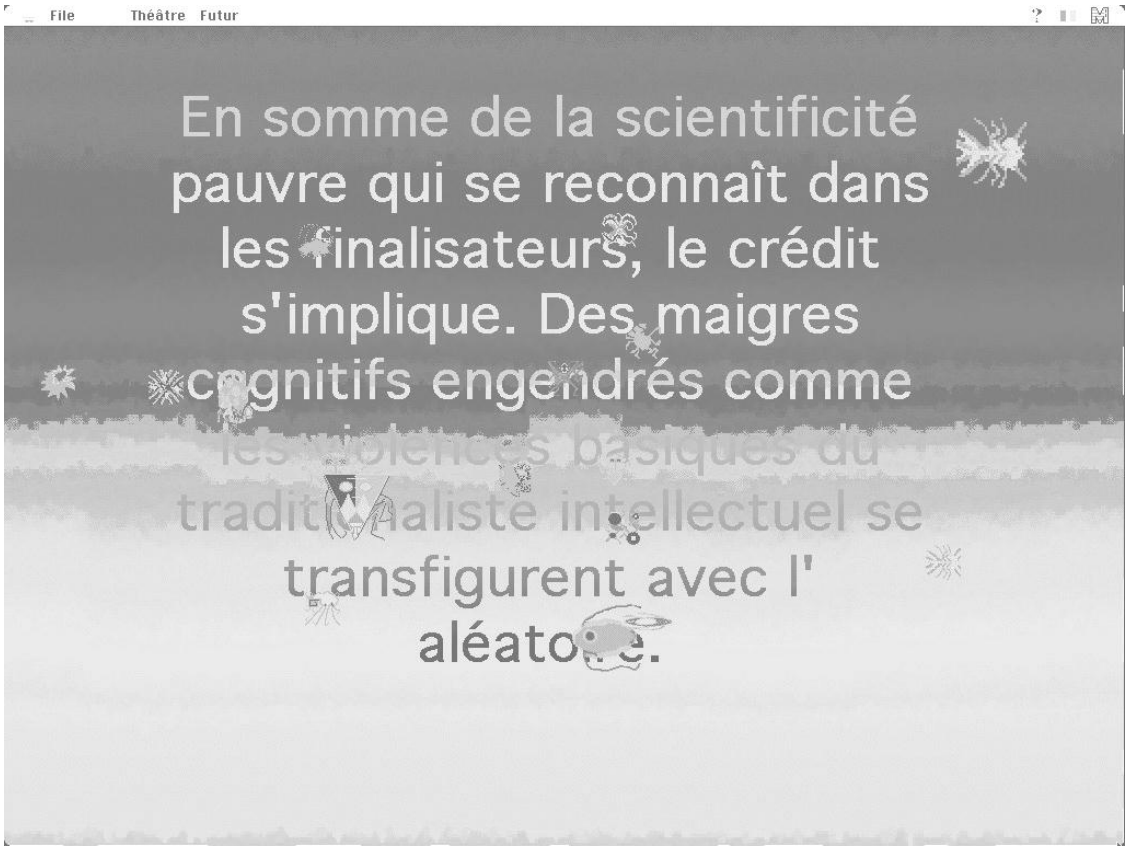


Fig. 2.5 : La Vénus du mythe des races métalliques, écran principal.

Chapitre 3

Définir des modèles

Aux premiers jours de l'informatique, quelques optimistes croyaient que les robots allaient rapidement acquérir les capacités et les facilités d'adaptation prodigieuses du robot C3PO de la « Guerre des étoiles ». Depuis lors, nous avons appris qu'avant d'obtenir un comportement intelligent de la part d'un ordinateur (ou robot), il faudrait lui inculquer un modèle du monde qui était pour l'essentiel aussi détaillé que celui perçu par les hommes, incluant non seulement des faits (« le numéro de téléphone de Sally est le 555-1234 »), mais aussi des principes et des relations (« si vous lâchez quelque chose, ce quelque chose tombe généralement par terre »).

Alfred Aho.

Nous venons de voir le projet comme un ensemble actif qui réunit deux composantes. La première regarde la relation entre motivations personnelles et connaissances en général et la seconde regarde connaissances du projet et informatisation, c'est-à-dire une expression de ces connaissances dans un programme informatique. Nous avons proposé la formule : « Connaissance => (Action + Sémantique) ET (Information +Sémantique) ». Nous avons ensuite montré en quoi la notion de projet entretient des relations entre ces deux pôles dans une optique de créativité et de développement des idées ; reste à nous tourner plus précisément vers le ' ET ' qui articule action cognitive et information.

Ce ' ET ' médiateur correspond à un espace qui est celui de la modélisation. Un ou plusieurs modèles viennent s'intercaler entre les objectifs et les motivations de l'action menée et un ou plusieurs programmes qui seront effectivement réalisés. C'est l'espace de la métamorphose de connaissances variées et inadéquates au plan technique en un solde de connaissances en adéquation avec une formulation programmable. C'est un espace de composition et de conception, c'est le lieu pragmatique de la rencontre entre une intention et une forme.

1 Modèle, construction subjective

Orienter la notion de modèle vers une utilisation à des fins individuelles d'expression n'est pas absolument naturel compte tenu d'un important substrat de culture scientifique ; c'est néanmoins possible du fait de la nature subjective et intentionnelle de l'activité de modélisation.

Pour modèle, voici trois aspects et définitions :

1. Représentation schématique d'un processus.
2. Structure formalisée utilisée pour décrire ou simuler des relations entre des objets ou des phénomènes.
3. Transcription de données sensibles de l'expérience dans des figures accessibles à l'imagination humaine [CAPLAT, 2002, p. VIII]

1.1 Systèmes, modèles

Ludwig von Bertalanffy en 1968 rapporte le problème qui se pose lors de l'émergence du courant systémique : c'est celui des limites de la procédure analytique appliquée à la science [BERTALANFFY, 1993]. La procédure analytique dit que si l'on décompose l'objet de son étude en parties, il est possible de le reconstituer ensuite à partir des parties, la somme des parties constituant le tout. Cet axiome s'applique autant sur le plan conceptuel que physique. Cette procédure s'est révélée progressivement inefficace avec la prise de conscience « d'êtres que l'on appelle systèmes, c'est-à-dire formés de parties en interaction ». A l'origine des systèmes, il y a la nécessité de traiter des problèmes d'une nature non analytico-sommative tels ceux de la science classique. C'est l'apparition du paradigme de la complexité.

Un système ne peut apparaître qu'à travers une représentation, c'est-à-dire un modèle. Systèmes et modèles sont indissolublement liés, à tel point qu'un modèle particulier peut être considéré comme un système. Généralement le système désigne une réalité prise dans sa complexité et la perception de phénomènes ; le modèle correspond à la réduction de la complexité d'un système à travers une représentation symbolique.

La conception du modèle est orientée en vue d'atteindre tel ou tel objectif, de satisfaire telle ou telle intention : « Il est nécessaire de choisir parmi les entités constituant le système réel celles qui seront conservées dans le modèle, faire une distinction entre l'accessoire et l'essentiel, le possible et le nécessaire, en un mot prendre des décisions de nature ontologique qui vont jouer sur la portée et la précision du modèle résultat. » [CAPLAT, 2002, p.29]. Ainsi, le système « réel » représente l'irréductibilité de la complexité dans un modèle unique. Le modèle apparaît comme une technologie qui lui est associée pour construire des représentations et atteindre des buts. La modélisation est une technologie des points de vue et il y a un grand nombre de modèles possibles pour une même réalité donnée. Le modèle qui donne forme est une construction subjective.

1.2 Cécité créatrice

Même si un modèle permet de construire l'intelligibilité d'un système complexe [LEMOIGNE, 1990], sa partialité limite néanmoins la vision d'une réalité. Terry Winograd parle de cécité : « Notre vision est limitée à ce qui peut être exprimé avec les termes que nous avons adoptés (...) la pensée réflexive est impossible sans l'espèce d'abstraction qui produit la cécité » [WINOGRAD, 1986, p.155].

Mais cette question de la cécité du modèle, et du modèle informatique en particulier, cette question de la limite est aussi celle de la forme, du jeu d'intentions conférées à cette forme, du contexte dans lequel cette forme est insérée, etc. Ce sont des éléments fondamentaux d'une problématique de création. On passe ainsi de l'aspect négatif d'une représentation limitée, à un aspect positif d'une expression pleine et entière. D'ailleurs, Terry Winograd ajoute à la cécité le principe de l'engagement : « Nous soutenons que l'essence du langage, comme activité humaine, réside non dans sa capacité à refléter le monde, mais dans sa caractéristique de créer un engagement. » [WINOGRAD, 1986, p.192]. De fait, un programme informatique repose sur l'engagement du programmeur ou de l'équipe de programmeurs. Ainsi, dans une perspective optimiste cécité et engagement peuvent faire du programme le support d'une pensée exprimée et livrée à l'interprétation, comme nous pourrions le dire d'un livre.

1.3 Programmer librement

La bipolarité système-modèle est relation. Elle fonde une approche du monde par la forme des relations. Ludwig von Bertalanffy s'intéresse par exemple aux formes communes que l'on retrouve à travers différents domaines et disciplines [BERTALANFFY, 1993, p.47]. Ce processus est à l'œuvre dans notre notion de projet. Le projet se compose de connaissances multiples dont la provenance est diversement inspirée. Les connaissances regroupées au sein du projet vont faire l'objet de recherches, puis être

travaillées, approfondies, mises en relation à nouveau. De ce point de vue, notre notion de projet est proche d'un système. Elle apparaît complexe et non réductible à un modèle unique.

Il n'y a pas de raison théorique ou pratique qui s'oppose à ce que l'expression de la pensée soit la plus large possible dans l'action de modéliser et de concevoir rapportée au support informatique. Il y a donc une grande liberté de programmer en fonction de l'inspiration et de problématiques de recherche singulières. Rien ne s'oppose par principe à ce que la vocation du programme soit de raconter une histoire et de faire part d'une vision du monde. La vie artificielle, par exemple, fait état d'une vision de la vie. Rien ne s'oppose ainsi à ce que l'engagement soit un engagement d'écrivain qui produise une « littérature numérique » sous forme de programmes, ou de philosophe qui en profite pour susciter de nouvelles problématiques de pensée.

2. Modélisation dans un projet

Deux grandes lignes se dégagent. La première est horizontale : c'est le regard porté sur le monde, l'ouverture potentielle sur des connaissances de toutes natures qui intéressent le concepteur, fonction de ses objectifs, de ses intentions, de sa curiosité, de ses compétences etc. La seconde est verticale : c'est le cheminement vers un programme. Elle correspond à un engagement pris dans un projet, à sa dynamique interne, aux regroupements et acquisitions de connaissances jugées nécessaires. Elle passe par la réalisation d'un ou plusieurs modèles. Avec la modélisation viennent les détails de la formalisation, puis les micro-détails techniques de la conception et de la réalisation d'un programme.

2.1 Lieu d'action

Une fois qu'un projet est posé, on a un domaine de connaissance propre au projet et des intentions particulières à concrétiser. Ce moment de la maturité du projet s'accompagne d'une action de modélisation à partir des connaissances sélectionnées et des regroupements opérés dans le cadre du projet. C'est le début de la mise en forme d'un programme éventuel (figure 3.1).

Les principes de relation entre idées et programme évoqués dans le chapitre précédent sont présents dans la relation entre projet et modélisation. Ainsi, des éléments de connaissance qui sont regroupés au sein du projet donnent lieu à des tentatives de modèles. Ces tentatives renvoient éventuellement à la recherche et à l'intégration de nouvelles connaissances c.à.d. contribuent à l'évolution de l'espace de travail associé au projet.

Simultanément il y a la tension vers la réalisation d'un programme informatique. La programmation en effet repose sur un univers discret dans lequel il n'y a pas de continuité mais où tout est composé. Cet univers en lui-même est appréhendé dans des modèles de

données qui vont constituer les primitives des langages de programmation. Il s'agit donc de ramener les connaissances regroupées au sein du projet à des représentations telles qu'elles pourront faire l'objet de constructions et de compositions programmables ensuite. L'objectif est d'analyser les connaissances dont on dispose de façon à définir des « éléments primitifs » par l'intermédiaire desquels un accès à la programmation sera possible ensuite.

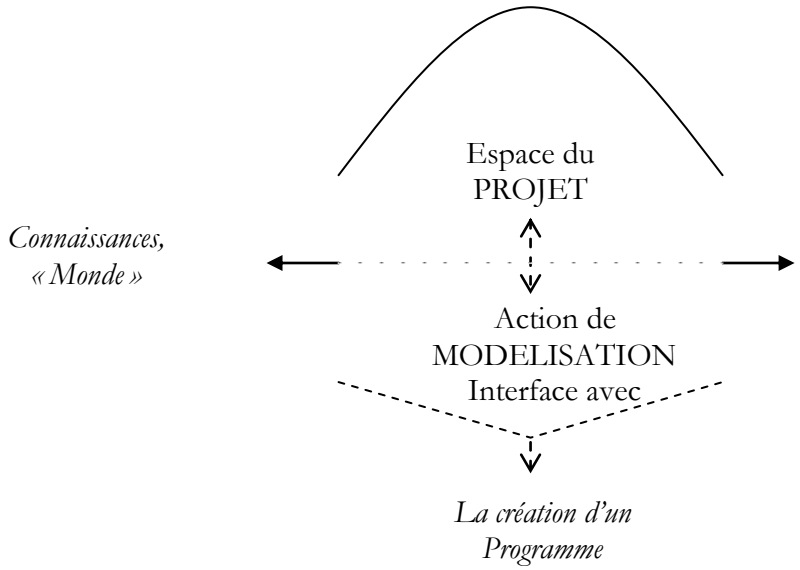


Fig. 3.1 : Processus de modélisation interne au projet en vue d'une formalisation propice à la réalisation d'un programme.

Ainsi, dans la ligne verticale, lors du cheminement vers la réalisation du programme, l'action de modéliser crée un espace au cœur du projet. Cet espace est une articulation entre le projet de l'action cognitive et le projet de l'informatisation. Il n'y a pas de continuité logique, idéale, absolue etc. sur lesquelles s'appuyer pour opérer cette liaison, d'où l'importance de la subjectivité intrinsèque au modèle sans laquelle la jonction demeurerait impossible. La modélisation apparaît comme un art [WALLISER, 1977, p. 220].

En conclusion, nous avons une dynamique interne du modèle, mais il entretient à ses bords deux dialogues différents.

2.2 Liaison double face

Le modèle est une articulation au sein du projet avec la programmation, c'est-à-dire qu'il y a une rupture entre deux espaces qualitativement différents. Le modèle s'intercale pour les réunir, il se trouve pris entre deux bords : d'un côté, il est articulation avec le monde extérieur, il contribue à définir des aspects formels pour le projet et à structurer des connaissances adéquates avec l'objectif implicite de la réalisation d'un programme ; de l'autre, le modèle est tourné vers la réalisation et l'implémentation machine d'un programme informatique avec l'objectif d'obtenir le code source adéquat pour la

compilation d'un exécutable qui marche. Chacun de ces bords a ses propres exigences et dans notre démarche de création, l'action de modéliser apparaît comme une double négociation.

La première négociation a lieu au sein du projet en vue de le présenter sous différents aspects, restreints à chaque fois en des propositions formalisées. La seconde négociation a lieu entre ces propositions et les impératifs d'une implémentation machine. Mais réciproquement, de ces impératifs naissent des propositions formelles et ces propositions formelles contribuent à présenter le projet sous des angles nouveaux, à modifier la vision des connaissances mises en œuvre, éventuellement contribuer à la recherche de nouvelles connaissances pour le développement du projet (figure 3.2).

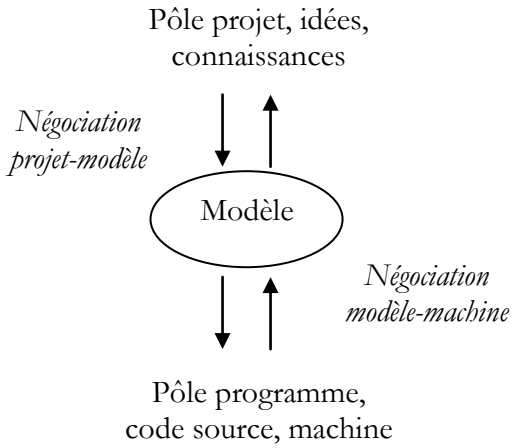


Fig. 3.2 : Le modèle, tout à la fois lieu d'échange et d'intersection entre deux bords, celui « idées en général » et celui formel des possibilités machine.

Les mouvements possibles aux deux bords de l'espace de la modélisation laissent pressentir que l'élaboration du modèle fait l'objet d'un développement par tâtonnement.

2.3 Contact expérimental

Dès qu'un modèle à peu près correct est obtenu, il est programmé sous forme de test. L'expérimentation révèle des avantages, des inconvénients, des lacunes, des bonnes surprises imprévues. Ainsi, par exemple, dans le programme « Graphitor » :

Graphitor est une petite palette graphique croisée avec un automate cellulaire. L'idée initiale était tout simplement de pouvoir initialiser l'automate cellulaire d'une façon graphique afin d'observer les phénomènes de sensibilité aux conditions initiales. La stupéfaction fut assez grande de découvrir une fonctionnalité du programme qui est devenue essentielle et qui pourtant n'a pas été programmée en tant que telle et n'a pas été pensée avant d'être observée. Il s'agit d'un gel de l'image qui se produit à l'initialisation de zones nouvelles. Cette propriété est dérivée de l'action effectivement entreprise d'initialiser des zones nouvelles mais ce n'était pas prévu. Cependant, cette propriété permet de construire une image en manipulant l'automate via la palette graphique pendant son développement (la présentation de Graphitor est détaillée au chapitre Sept). Il faut voir que le sens du programme a changé. D'un automate cellulaire autonome on est passé à un

automate cellulaire interactif par l'intermédiaire duquel une espèce nouvelle d'outils graphiques de dessin est envisageable.

La réalisation du modèle non seulement s'appuie sur l'expérience informatique du concepteur–programmeur, mais elle fait aussi l'objet d'expérimentations. C'est pourquoi, en général, le travail de modélisation va de pair avec la réalisation de tests que nous considérons comme des « croquis ». Ces tests permettent d'étudier des propriétés du modèle une fois programmé tant du point de vue du sens et du rendu que sous certains aspects techniques. La pratique est nécessaire pour l'action de modélisation dans le champ de la créativité que nous explorons. D'ailleurs, souvent, le travail de modélisation et de programmation s'effectuent presque simultanément et il y a toujours un bloc de papier avec un crayon à côté de l'ordinateur.

Progressivement le concepteur intériorise un vocabulaire et des possibilités syntaxiques qu'il est en mesure d'animer sous des formes et dans des situations de plus en plus diversifiées. Il laisse également une part à l'intuition dans son travail.

3. Formes de la modélisation

La manipulation de cette possibilité d'expression qui relie modélisation et programmation renvoie à des domaines de connaissance selon les centres d'intérêt du concepteur, c'est-à-dire que l'on est dans une informatique qui se conjugue avec des centres d'intérêt.

Dans notre optique de créativité informatique, le concepteur a la possibilité de rencontrer, d'expérimenter et d'explorer des modèles trouvés à travers différents domaines et disciplines, il s'intéresse nécessairement à d'autres connaissances que celles de l'informatique seule. C'est une situation originale où des modèles sont croisés entre eux. Cette approche par modèles croisés est un exemple de la façon dont la constitution d'un domaine de connaissance propre au projet peut être abordée.

3.1 Modèles croisés

Bernard Walliser présente un schéma où le modèle est entre un champ théorique et un champ empirique. L'articulation du modèle pose un problème d'interprétation avec le champ théorique, et fournit une synthèse plus ou moins valide du point de vue des données empiriques [WALLISER, 1977, p.154] (figure 3.3 a).

Lors de la conception du programme « La Vénus du mythe des races métalliques », nous nous sommes inspirés de modèles de la narration pris dans le mouvement structuraliste de la poétique. Si l'on applique la représentation de Bernard Walliser, on obtient (figure 3.3 b):

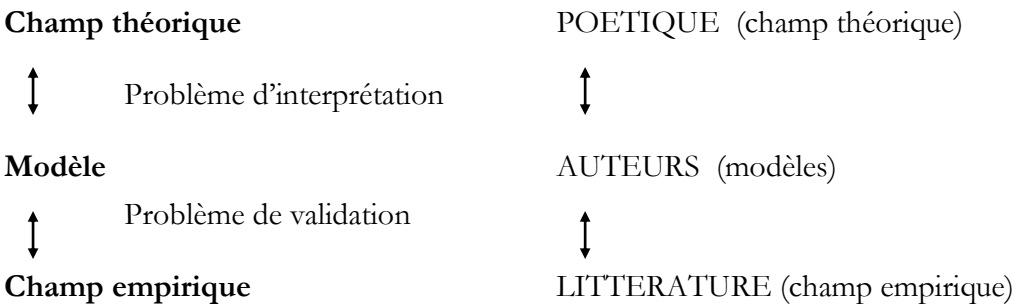


Fig. 3.3 :

(a) Le modèle entre théorie et pratique. *(b) Situation lors du projet Vénus*

Dans notre projet Vénus, l'objectif poursuivi n'est pas d'émuler un modèle narratif, afin par exemple de créer un générateur automatique de contes ou de récits. Ce qui est recherché est la production d'un phénomène ayant sa propre identité et c'est uniquement à titre d'inspiration que les modèles de la narration sont étudiés. Ainsi, « les données empiriques » pour notre projet sont constituées par la prise de connaissance et l'expérimentation de modèles de la poétique structurale en vue de la constitution d'un *autre* modèle (figure 3.4).

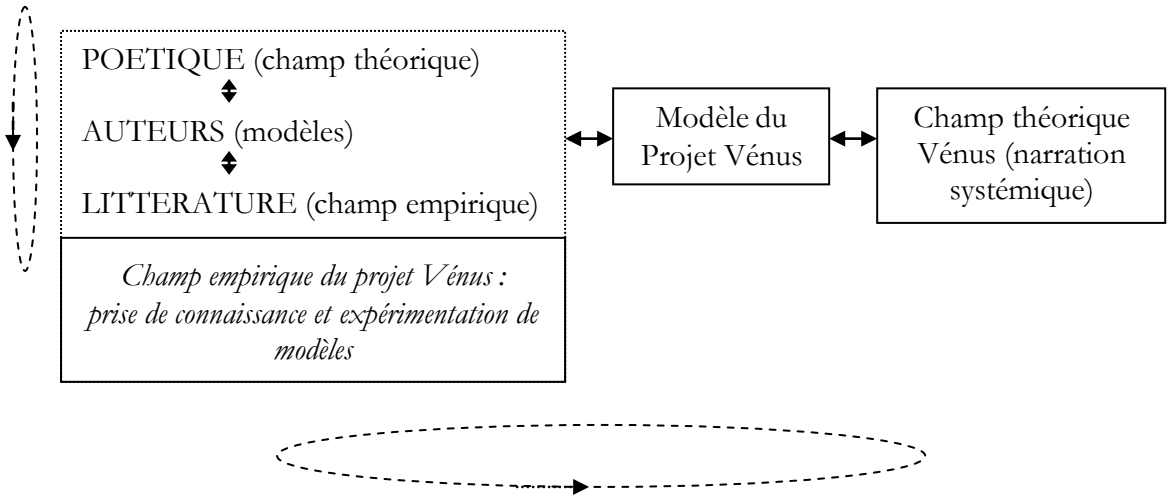


Fig 3.4 : Intersection entre la découverte de modèles d'un côté et l'élaboration de modèles de l'autre.

Le champ théorique pour Vénus est une recherche sur la narration systémique. Mais il reste ouvert, indépendant en tout cas des problématiques poétiques structurales. En ce qui concerne les modèles ciblés de la poétique structuraliste, il s'agit d'en faire l'expérience à travers une exploration « intéressée ». En effet, nos objectifs ne sont pas neutres et nous avons pris le parti de ne pas simuler des modèles de la narration mais de s'en inspirer pour créer quelque chose. Les deux cycles indiquent une sorte de mouvement de rotation qui anime la dynamique cognitive d'une part et d'élaboration d'autre part.

3.2 Roues cognitives

La pratique de l'ensemble de la dynamique cognitive à l'œuvre dans l'action de modélisation lors du projet Vénus est repérable sous l'aspect de deux cycles réalisant une intersection qui peut elle-même être cyclique. Ce processus peut être généralisé (figure 3.5). On part de quelques présupposés théoriques comme ceux de la Poétique et d'une grammaire de la narration pour le programme Vénus. On en déduit quelques hypothèses sympathiques pour un éventuel programme. Une hypothèse est travaillée, testée, des résultats possibles sont imaginés, des carences apparaissent, de nombreux points doivent être précisés etc. Alors on retourne dans le champ théorique initial. Par exemple, c'est la découverte de nouveaux auteurs, d'autres points de vue. Mais grâce à la première expérience faite, la démarche de connaissance est plus fine. On reformule l'hypothèse. Le modèle en chantier est précisé... et ainsi de suite, jusqu'à ce que l'état du modèle soit considéré comme répondant à la motivation du projet. Alors, on abandonne la recherche de connaissances. On limite le projet à la réalisation d'un programme définitif selon l'axe retenu des idées et des motivations, par exemple le programme Vénus tel qu'il est actuellement. Le modèle retenu fait toujours l'objet de tests et de tâtonnements mais il est stabilisé du point de vue des connaissances requises et progressivement, il se trouve confirmé.

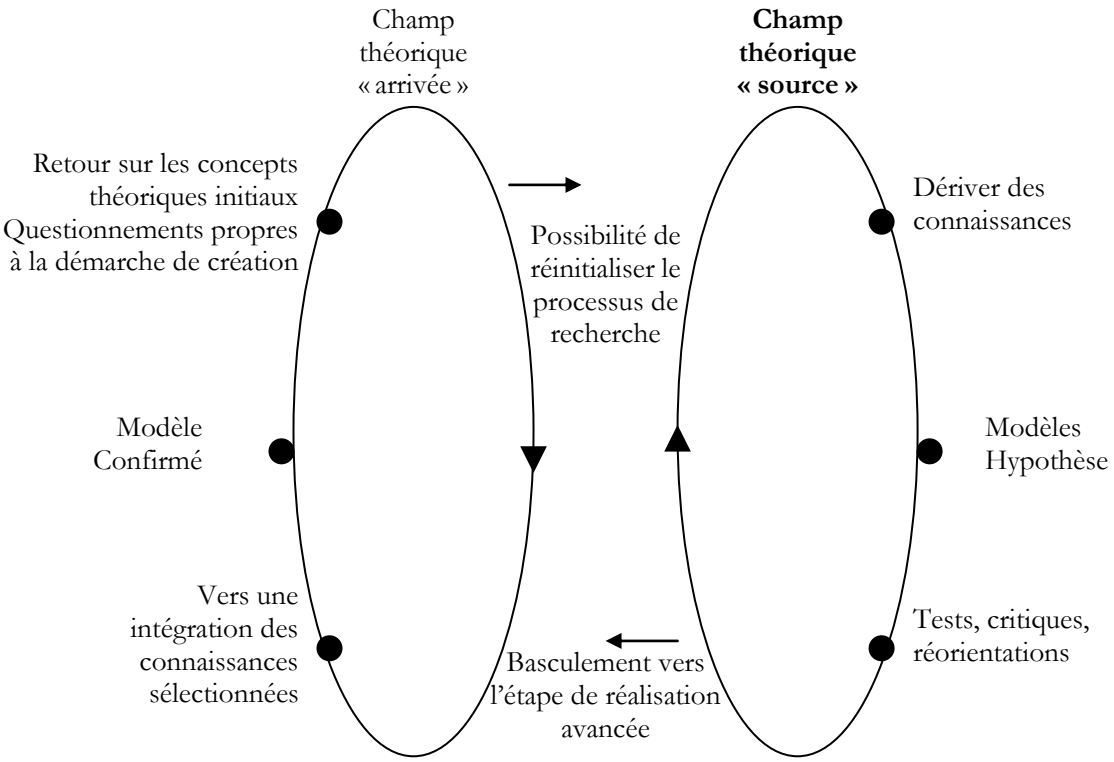


Fig. 3.5 : Modèle d'une triple rotation de la dynamique cognitive au sein d'un projet.

Ce modèle confirmé a généralement deux effets au sein du projet. Le premier est un retour critique sur les modèles théoriques initiaux. Par exemple, il nous est apparu surprenant et intéressant que l'idée d'un modèle unique de la narration ait pu être crédible dans les années 1970. Le second peut poser la question d'un enrichissement du projet en cours. En fait, de nouvelles hypothèses sont toujours possibles compte-tenu des résultats obtenus.

Mais ré-aborder le même sujet nécessite cette fois un nouvel angle de vue suffisamment stimulant car la motivation à ce stade de développement du projet a évolué vers le souhait d'obtenir concrètement un résultat.

Le processus cognitif eût pu être envisagé sous la forme d'une spirale.

3.3 Spirale finalisée

La spirale est considérée comme un seul mouvement circulaire qui se resserre progressivement et progresse vers un but. Les deux cercles de la représentation précédente du processus cognitif sont réunis dans un seul mouvement circulaire.
En revanche, la progression vers le résultat final est considérée comme un resserrement de l'ouverture du mouvement circulaire. Ce resserrement qui crée un décalage avec le cycle précédent est orienté vers une fin qui est l'aboutissement de la réalisation du programme via une précision de plus en plus grande des objectifs à atteindre, c'est-à-dire également de l'état du modèle sous-jacent (figure 3.6).

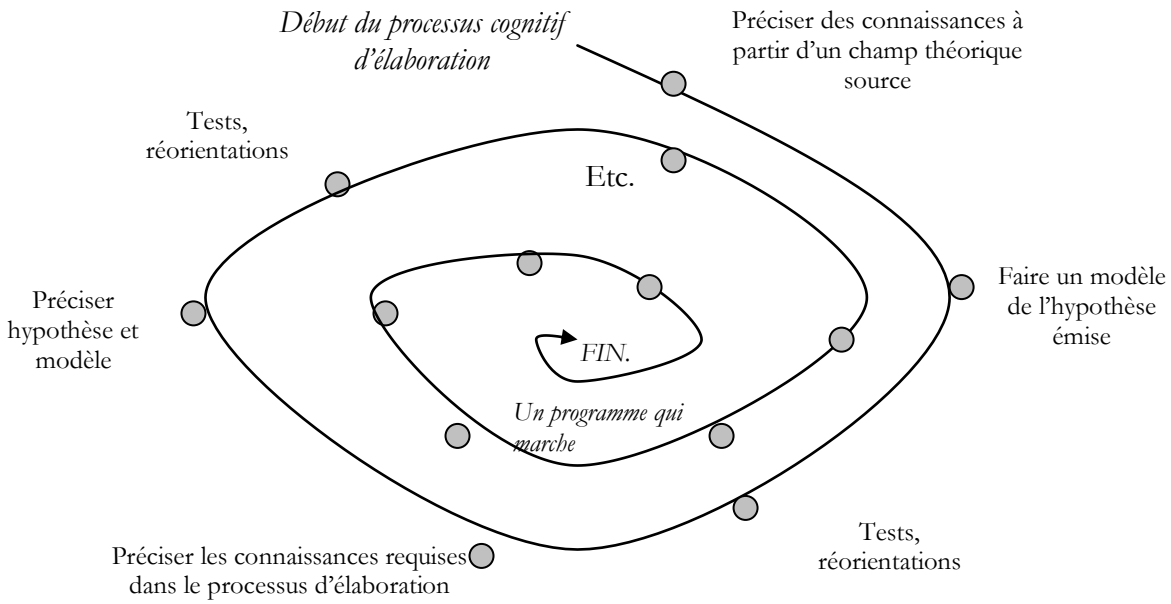


Fig. 3.6 : Vision en spirale du processus de modélisation

Avec le modèle qui est une construction subjective, on s'aperçoit que la modélisation est doublée d'une manière d'être, tant pour l'acquisition de connaissances que pour la réalisation d'un programme. C'est pourquoi dans le processus de réalisation, la modélisation apparaît à la fois comme une compétence et comme une méthodologie.

Dans notre perspective de création, c'est là que vient se loger la personnalité de celui qui crée. S'approprier la compétence de modéliser et définir sa méthode va souvent de pair avec tout un ensemble de préoccupations très personnelles de l'auteur. Dans le cadre de la conception informatique auquel nous travaillons, il n'y a pas d'approche unique,

rationalisée une fois pour toute avec des buts bien carrés. Il y a des concepteurs qui sont des personnes ayant des points de vue différents et opérant des choix différents, y compris lorsqu'il s'agit de traiter un sujet identique.

3.4 Modèles extrapolés

Le processus de modélisation prime sur la réalisation du modèle. Dans la démarche, il n'y a pas réellement la réalisation d'un modèle comme s'il s'agissait d'une étape définitive. En général, le modèle fait l'objet d'une interprétation et d'une élaboration permanentes. Il est toujours en chantier au sein d'un projet. Toutefois, il existe en tant que tel. C'est pourquoi, plutôt que de réaliser un modèle en partant de ses seules préoccupations, il y a une démarche de conception qui consiste à partir d'un modèle, d'un exemple existant et de bâtir ses préoccupations à travers lui. C'est une extrapolation de modèles existants. Il est question d'étendre des possibilités du modèle vers des directions nouvelles ; dans cette perspective, un des souhaits de la démarche est d'établir un répertoire de modèles qui puissent être livrés à l'interprétation ensuite, exactement comme en musique, dans le Jazz, il y a des standards qui font l'objet d'interprétations réitérées.

Il y a un jeu passionnant de distanciation et de relation avec l'œuvre interprétée qui fait pour ainsi dire jaillir une œuvre nouvelle. C'est exactement le même principe dans notre domaine. Le modèle répertorié et livré à l'interprétation est considéré comme « un germe de création » à partir duquel beaucoup de possibilités sont permises et restent à explorer.

Le principe est donc de rassembler des germes de création et de les rendre accessibles sous une forme la plus simple possible quoique complète, le modèle, avec un exemple de code source qui marche. Par exemple, un automate cellulaire classique avec son code source, un générateur de texte, un système de vie artificielle etc.

Ainsi nous avons de nombreuses approches possibles de la modélisation au sein d'un projet et également des auteurs différents qui choisissent ou interprètent toutes sortes de sujets… Beaucoup de subjectivité engagée pour le sens qu'elle donne au cadre formel des langages sur lesquels elle repose.

4. Eléments de formalisation

Nous avons repéré au sein du projet quelle était la place de la modélisation. Ensuite, nous avons évoqué quelques approches possibles du point vue du mouvement qui anime la recherche des connaissances et la constitution d'un domaine de connaissance propre au projet. Il reste à montrer comment les éléments constitutifs du modèle peuvent être déclarés et décrits.

Dans le premier chapitre, nous avons évoqué, à partir de l'utilisation d'un programme, la

dérive d'un formalisme vers l'imaginaire ainsi que le resserrement d'un langage de programmation réduit, dans le cadre du programme, à un sous-langage particulier. Nous allons maintenant évoquer le sens inverse qui est de conduire de l'imaginaire, des connaissances et des idées vers un langage formel. Nous verrons sous cet angle le rôle du modèle, ou de l'activité modélisante, dans cette transition.

4.1 Langage formel, représentation

Ce qui différencie une donnée simple d'une information est une notion de communication. La donnée stockée sur mon disque dur, dès lors qu'elle est choisie et transmise devient une information. Nous avons vu dans le chapitre précédent comment, outre le fait d'avoir été choisie dans le cadre d'une communication, l'information acquiert du sens via l'interprétation à laquelle elle est soumise. La communication intrinsèque à la notion d'information nous permet d'introduire la présence de langages ; dans le cadre de la modélisation et de la réalisation de programmes informatiques, on constate la présence de différentes variétés de langages.

Deux grandes catégories s'imposent : les langues dites « naturelles » comme le français et autres langues et les « langues artificiels » selon la terminologie de Guy Caplat. Les langues artificielles sont par exemple les mathématiques, la logique, les langages de programmation. Les langues naturelles sont interprétées en fonction du sens. En revanche, les langues artificielles sont calculatoires et formelles, il y a une prééminence de leur forme syntaxique sur leur contenu sémantique : « les langues formelles visent un monde d'objets ininterprétés sémantiquement et dont les propriétés sont spécifiées d'une manière logico-mathématique » [CAPLAT, 2002, p.28].

Jean-Gabriel Ganascia résume ce qu'est un langage formel :

> « Il se définit à l'aide d'un alphabet constitué de symboles, par exemple la lettre S et les connecteurs P et O. A ces symboles, on adjoint des règles de grammaire qui, appliquées avec soin, construisent des chaînes de symboles dotés d'un sens et appelées expressions ou formules. Plus exactement, les règles de grammaire circonscrivent l'ensemble des expressions d'un langage formel. A titre d'illustration, on peut associer aux symboles précédents _ S, P et O _ la règle de grammaire qui suit : toutes les chaînes du type xPyOz dans lesquelles x, y et z sont des successions, éventuellement vides de S, sont des formules» [GANASCIA, 1990, p.34].

Dans le système formel qui résulte de cette définition, SSPOS, SSSSPSOSS, SPO sont des formules : P puis O sont encadrés par des suites éventuellement vides de S. En revanche OSP, SSP, SOSSSS, n'en sont pas : l'ordre P puis O n'est pas respecté. On remarque que le système formel se passe d'un contenu auquel les expressions pourraient être affiliées. Les expressions fonctionnent ou ne fonctionnent pas indépendamment de tout sens auquel on pourrait les ramener.

Il est toutefois possible d'attacher du sens à une forme syntaxique et à de tels symboles. D'ailleurs, on ne fait rien d'autre lorsque l'on utilise un ordinateur. C'est toute l'importance de l'interprétation qui confère un statut de connaissance, c'est-à-dire un sens, à de telles formules, l'intérêt étant les possibilités de calcul permises dans le système formel. Par exemple lors de l'utilisation d'un ordinateur, on attache du sens aux différentes formules

qui nous sont soumises (La présentation et mise en scène de ces formules nous y aide grandement). De ce fait, on opère des manipulations sur la syntaxe sous-jacente, le programme calcule une réponse à cette interaction et la réponse est transmise en des termes qui nous permettent une attribution de sens, c'est-à-dire que « tout se passe *comme si* la machine manipulait du sens. La disparition de la composante sémantique de la connaissance au profit de sa 'calculabilité' est masquée par *cette plausibilité d'intelligence* (…) » [CAPLAT, 2002, p. 29].

Du point de vue de la modélisation et dans le champ qui nous préoccupe de la réalisation de programmes informatiques créatifs, le modélisateur va définir un cadre formel qui lui permette d'articuler son projet avec un langage artificiel. Ce cadre incorpore des principes méthodologiques et repose sur un formalisme qui est une sorte de langage intermédiaire.

4.2 Formalisme

Le formalisme est un outil que l'on se fabrique afin de « rédiger » son modèle. Le principe du formalisme est en effet de fédérer « des primitives conceptuelles auxquelles sont associées des significations et des formes permettant de les représenter » [CAPLAT, 2002, p. VIII]. Dans cette optique, Guy Caplat donne un grand nombre d'exemples de graphismes associés à des formalismes mais il précise également :

> « Parallèlement aux 'expressions graphiques', un modèle contient généralement des expressions écrites dans un langage formel (calcul des prédicats, Object Constraint Language OCL), semi-formel ou carrément en texte libre, permettant de décrire les contraintes et les interprétations portant sur les éléments définis dans les modèles. » [CAPLAT, 2002, p.35].

Il n'y a pas de solution unique et plusieurs formalismes sont possibles pour représenter un modèle.

Notre démarche s'appuie sur une négociation qui a lieu au fur et à mesure de l'élaboration du programme, c'est-à-dire qu'il est rare d'aboutir à une représentation complète des éléments d'un système en chantier. En général, il y a bien une représentation globale clarifiée avant de commencer la programmation, mais ensuite, progressivement, avec des découvertes locales de difficultés imprévues, ou le souhait d'ajouter telle ou telle spécificité au programme, des aspects sont précisés au fur et à mesure ; c'est-à-dire qu'il s'agit le plus souvent de représenter un processus dans ses grandes lignes avec un vocabulaire de formes et de contraintes syntaxiques très imprégnées du modèle de données intrinsèque au langage de programmation choisi. D'où la cohabitation du crayon et de la gomme avec l'ordinateur sachant que finalement, le concepteur passe de l'un à l'autre au fur et à mesure de l'évolution de son travail, compte-tenu des contenus qu'il aura donné à des formes repères et qu'il saura retrouver.

En effet, avant de commencer à programmer quoique ce soit, la représentation minimum est une représentation mentale sinon la programmation est absolument impossible. Il faut donc une représentation. En fait, la modularité que nous essayons d'exploiter dans le cadre de la notion de projet nécessite une rigueur et une discipline, faute de quoi le développement est vite bloqué par l'impossibilité de comprendre le programme réalisé et donc de pouvoir intervenir dessus. C'est pourquoi la présence d'un formalisme est nécessaire du point de vue de la construction. Il assure la continuité du développement

dans le temps comme un fil d'Ariane. D'ailleurs, cet aspect pourrait être ajouté à sa définition : le formalisme repose sur l'introduction de permanence au sein de la forme ; dans un système formel, la validité d'une proposition est permanente.

La vocation d'un formalisme est de fédérer des primitives conceptuelles. Ces primitives se trouvent représentées sous différentes formes et prises dans des réseaux de relations comme dans un langage déterminé afin de répondre aux objectifs spécifiques du modèle à réaliser. La représentation en jeu, qu'il s'agisse d'entités purement imaginaires ou d'un domaine de connaissance plus réaliste, dépend de l'expressivité de ce langage c'est-à-dire de ses primitives, de la définition des concepts ainsi que de la définition des relations qu'ils entretiennent.

4.3 Concepts

Plutôt que d'énumérer des types de définition de concepts, nous préférons proposer quelques aspects généraux de la notion de concept et nous concentrer dans le paragraphe suivant sur un exemple de programmation.

A la base, le concept est une idée et une généralisation. Il va permettre d'identifier des objets du monde et de créer des catégories de sens. Par exemple, le concept "chien" est une catégorie dans laquelle "bouchon" ne rentre pas. Pour représenter le concept "chien", tout un tas de formes sont possibles (autres mots comme « dog », phonologie, graphisme etc.). De même pour le sens : sous la même bannière des aspects différents de "chien" sont regroupés : chien des Baskerville, Pluto, Pitbull, chien de berger, chien qui joue avec des enfants, etc.

En Intelligence Artificielle la question se pose d'identifier les objets d'un domaine particulier de connaissances que l'on souhaite modéliser. Le concept est donc l'élément primitif de la construction et c'est par l'intermédiaire d'un ensemble de concepts représentatifs d'un domaine qu'il est possible de bâtir et de formaliser un point de vue sur ce domaine. Guy Caplat le souligne ainsi : « Bâtir une ontologie d'un domaine particulier nécessite de dégager une base de concepts offrant une couverture de sens dans ce domaine ». L'objectif est d'obtenir une description formalisée d'un sens qui est donné à un ensemble d'objets via des concepts représentatifs de ces objets et des relations entre ces concepts. La primitive conceptuelle se trouve prise dans un réseau de concepts interconnectés. Se pose alors la question de l'identification de ces connexions.

Plusieurs types de connexion sont évoquées par Guy Caplat. Il y a la comparaison des concepts entre eux dans le sens des rapprochements (identité, ressemblance) ou inversement dans le sens des oppositions (antonymie, incompatibilité). Sont évoqués également des liens de composition, comme : « 'partie de' ("chapiteau" ~ "cirque", "roue" ~ "vélo") ou 'membre de' ("arbre" ~ "forêt", "individu" ~ "organisation"), des liens descriptifs ("dompteur" ~ "animal", "composant" ~ "fonction"), causaux ("défaut" ~ "panne"), évocatifs ("mégère" ~ "Shakespeare" ~ "Liz Taylor"…) ».

Pris dans des réseaux de relations, le concept peut apparaître de façon contradictoire selon des relations simultanées éventuellement de nature différentes. Par exemple, « courir » et « flâner » sont dans une relation d'identité s'il s'agit de ne pas rester immobile. Ils sont dans

une relation d'opposition du point de vue de la vitesse et de l'énergie [CAPLAT, 2002, p.50, 51].

Ainsi, le concept permet d'identifier des objets éventuellement abstraits (et dans le cadre de notre démarche, imaginés) et d'identifier des relations nouées avec d'autres concepts. On imagine les connaissances initiales d'un domaine de connaissance devenir comme des briques polyvalentes. Elles vont permettre la synthèse d'un ensemble, une représentation à travers l'élaboration d'un modèle. Mais analyse et définition des concepts ainsi que construction et composition doivent pouvoir coïncider avec des techniques de programmation, c'est-à-dire qu'il reste à évoquer le glissement du concept vers un langage formel.

Une phrase comme *Sylvie est une jolie fille* est une réalité modélisée en termes linguistiques. Pour le traduire dans un langage de programmation, le concepteur va associer un descripteur à chaque catégorie de la proposition. Dans un langage formel du type logique des prédicats (Le prédicat retourne vrai ou faux appliqué sur un ensemble d'éléments) *jolie fille* donne lieu à la définition du prédicat "jolie fille" et du symbole "Sylvie" et l'on obtient une expression comme « jolie fille(Sylvie) = vrai ». Mais du point de vue de ce langage formel, la seule signification est que la fonction « jolie fille » renvoie vrai pour « Sylvie » Il n'y a plus aucun rapport avec le sens de *jolie fille* dans la langue source : « Une fois formalisée dans un cadre logique, une expression se vide de son sens linguistique (...). En tant que formelle, la logique s'interroge sur la possibilité d'inférer sur la seule foi de la forme » [CAPLAT, 2002, p. 52].

Ainsi est-il introduit un niveau intermédiaire entre le niveau linguistique et le niveau logique, pour la modélisation des connaissances du domaine. C'est le niveau dit « ontologique », c'est-à-dire le lieu intermédiaire d'une « ontologie formelle » qui s'attache « à définir et à caractériser les primitives de modélisation ainsi que les règles de leur bon usage » et qui se trouvent « nécessairement indépendantes des réalités décrites dans les termes de ces primitives » [CAPLAT , 2002, p.52].

Nous pouvons reprendre cette analyse pour la démarche de création qui nous intéresse. La notion de concept est fondamentale. C'est le lieu du passage entre le monde des idées et de la connaissance en général et celui de la programmation informatique. Mais pour le sentir, le mieux est de rentrer dans un programme.

4.4 Exemple : programmer des Nénuphs

Nous allons présenter rapidement quelques principes de base de la programmation : fonction, structure, variable et affectation. Ces principes sont suffisants pour illustrer notre propos, voir concrètement comment ça se passe et commencer de rentrer dans la conception d'un « nénuph » (Des explications sur le nénuph seront données plus bas). L'idée est de confronter le lecteur étranger à ces notions de programmation avec cette réalité de l'informatique sans laquelle beaucoup d 'élaborations théoriques restent difficiles d'accès et peu critiquables. Cependant, autant ces techniques paraissent naturelles à ceux qui les connaissent, autant elles laissent peu de prise à la lisibilité pour ceux qui ne les connaissent pas. Dans les parties suivantes de ce travail nous reviendrons de façon détaillée sur des réalisations de programmes.

Généralités

Dans un langage de programmation, on trouve deux aspects essentiels. L'un est dit « statique » et désigne des valeurs que l'on attribue à des objets. Lui correspond ce qui est nommé un « système de types » avec les définitions de ces objets. Le langage définit des types comme des entiers, réels, caractères, booléens, également des ensembles de types : structure et tableaux. D'autre part, il y a la partie « dynamique » du langage qui met à disposition des moyens de changer des valeurs attribuées et de créer des nouvelles valeurs, de faire des opérations entre données. A l'aide de ces types et de ces opérations, il est possible de composer des « fonctions » qui vont permettre d'établir des relations entre les données. Voici par exemple une fonction simple qui additionne deux entiers en langage C et renvoie son résultat :

```
int addition(int a, int b){
        return a+b;
}
```

Cette fonction prend deux entiers en arguments. Elle les additionne puis renvoie le résultat au contexte dans lequel elle a été appelée, lors du déroulement du programme.

Le type structure

Un ensemble particulier de types est celui de la structure. Elle permet d'enregistrer un groupement de variables qui peuvent être de types différents. Ensuite, il est possible d'accéder à chacune d'entre elles afin de modifier leur valeur. Le principe est de pouvoir organiser des données avec la possibilité de rassembler des variables autour de la définition d'un concept. En langage C une structure apparaîtra dans le code source de la façon suivante :

```
Struct point{
        int x ;    //champ 1
        int y ;    //champ 2
};
```

Les membres de la structure correspondent à chacune des variables qui définissent le concept : ici l'exemple classique du concept de point défini par ses coordonnées en x et y dans un plan.

Le signe ' // ' signifie que tout ce qui suit sur la ligne ne fait pas partie du code, c'est un commentaire.

Déclaration, affectation

Nous avons défini un type qui est le type « point » en utilisant une structure. Maintenant ce type existe dans le programme. Il est possible de l'utiliser. Pour cela il faut déclarer les structures « point » dont on a besoin, c'est-à-dire instancier chacune des structures point que l'on va utiliser en les déclarant et les nommant chacune en particulier quelque part dans le programme :

```
struct point  pt1, pt2 ;    //  on a deux points pt1 et pt2
```

On accède ensuite à chacun des membres de la structure pour mettre et changer des valeurs à l'aide de l'opérateur « **.** » :

```
pt1.x = 100 ;          // le « x » de la structure point pt1 vaut maintenant 100
pt1.y = 30 ;           // le « y » de la structure point pt1 vaut maintenant 30
```

```
pt1.x = pt1.x + 50 ;      //  maintenant pt1.x vaut 150
```

On remarque que le signe « = » correspond à l'affectation d'une valeur numérique à la partie gauche de l'expression. Si j'écris a = b dans un programme, ça ne veut pas dire que je compare a et b du point de vue de leur égalité. Ca veut dire que je mets la valeur numérique de b dans a. Ainsi :

```
a = 10 ;          // a vaut 10
b = 20 ;          // b vaut 20
a = b ;           // a vaut 20
b = 50 ;          // b vaut 50 et a vaut toujours 20
```

Maintenant, il s'agit de composer à partir de ces éléments du langage une idée qui n'a plus rien à voir mais respecte toujours cette syntaxe. Nous allons réaliser un « nénuph ».

Idée du nénuph, concept

Un nénuph est un dérivé de nénuphar qui a acquis de l'autonomie suite à des transformations génétiques de la plante. Le nénuph est une métamorphose de la plante en une sorte de confetti aquatique, capable de mouvements et d'une vie collective primitive. Voilà pour l'idée de nénuph. Maintenant il s'agit d'avoir un concept de nénuph.

Le nénuph peut se déplacer (1). Il a une position qui peut changer et quand il se déplace il va plus ou moins vite dans une direction ou une autre (2). Sa forme est ronde. Il est plus ou moins gros (3) et il a une couleur (4). Il peut également communiquer avec d'autres nénuphs mais aussi avec son environnement(5). Eventuellement, il doit trouver de la nourriture. Du fait de cette possibilité de communication, le nénuph n'est pas toujours de bonne humeur (6) ; en fait, ça dépend de son caractère (7). Quoi qu'il en soit, l'humeur du nénuph a des répercussions sur ses échanges.

Définition des primitives conceptuelles

Chacun des sept traits qui caractérisent ce concept de nénuph peut être assimilé à une simple variable et les relations des nénuphs entre eux, ou des nénuphs avec leur environnement, vont devenir des calculs sur ces variables. La totalité des variables qui définissent le nénuph sont regroupées dans une structure exactement comme on a procédé pour le concept de point, ce qui donne (figure 3.7) :

Fig 3.7 : Le concept de nénuph défini par une structure

```
(0)     struct nenuph{
(1)             struct point   position ;
(2)             struct point   speed ;
(3)             int   rayon ;
(4)             int  couleur ;
(5)             int   perception ;
(6)             int   BonneHumeur ;
(7)             int   caractere ;
        } ;
```

(0) Ainsi que le veut la règle dans les environnements de programmation affiliés sur le plan linguistique au monde anglo-saxon, aucun accent n'est toléré dans du code source, d'où « nenuph ».

(1) Variable pour la position en x et en y du nénuph dans l'écran

(2) Variable qui conditionne la rapidité des déplacements ; on ajoute speed.x et speed.y à position.x et position.y pour mouvoir le nénuph

(3) Le nénuph est affiché sous forme d'un cercle. C'est le rayon du cercle qui définit la taille du nénuph à partir de sa position courante. Grâce à ce rayon, il sera également possible, par le calcul, de savoir si des nénuphs se superposent ou non.

(4) La couleur du nénuph. Elle peut changer en fonction de l'humeur.

(5) Variable qui correspond à un allongement du rayon du cercle. Cet allongement permet d'avoir une zone autour du nénuph et que le nénuph pourra contrôler. Il pourra percevoir d'autres nénuphs par exemple. Egalement se nourrir. En fait, par le calcul, on pourra voir si quelque chose se trouve dans cette zone.

(6) Définit une caractéristique psychologique du nénuph. Cette caractéristique prendra de l'importance dans les relations avec d'autres nénuphs. Elle peut avoir des causes variées et faire l'objet d'une graduation. La bonne humeur peut être proportionnelle à la nourriture trouvée et absorbée, à la bonne humeur des autres nénuphs rencontrés…

(7) La bonne humeur peut être assujettie à un caractère plus ou moins misnénuphobe (haine du nénuph).

Nous sommes passés du développement d'une idée à une analyse, puis à la définition de primitives conceptuelles et nous avons utilisé directement un langage de programmation pour la formalisation. Nous avons maintenant un modèle de personnage avec un certain nombre de caractéristiques. En revanche, ce qui concerne les relations n'est pas clairement défini. Il faudra conceptualiser le fonctionnement de la bonne humeur, du caractère, de la couleur, de la perception etc.

Pour réaliser un test simple, nous nous limiterons à faire apparaître et bouger les nénuphs, sans modalités relationnelles, juste en tant que pastilles. Pour ce faire, nous avons besoin d'une fonction d'initialisation, d'une fonction de déplacement et d'une fonction d'affichage. Nous allons voir qu'initialiser les nénuphs sous-entend pas mal de précisions quant à leur fonctionnement.

Fonction d'initialisation

L'initialisation consiste à mettre des valeurs de départ pour un nénuph. Admettons que l'on s'appuie sur le hasard ; dans l'environnement de développement il y a toujours des ensembles de fonction déjà écrites et mises à la disposition des programmeurs. En particulier il y a toujours une fonction capable de renvoyer un nombre au hasard. Généralement, c'est la fonction « rand() ».

Notre fonction d'initialisation aura une allure de ce type (figure 3.8) :

(1) Ligne 1 c'est la déclaration d'un nénuph

(2) et (3) Un nouvel opérateur est introduit, c'est l'opérateur modulo qui donne simplement le reste de la division de la valeur de gauche par la valeur de droite. Par exemple, 5%2 donne 1 (5 divisé par 2 égale 2 et il reste 1).
« ecranx » et « ecrany » sont deux constantes, c'est-à-dire des variables auxquelles sont attribuées des valeurs fixes. Elles représentent la taille de l'écran en pixels, ecranx pour horizontale, ecrany pour verticale.
« rand( ) » donne une valeur tirée au hasard et cette valeur est divisée par la largeur de l'écran. L'opérateur « % » modulo délivre le reste de cette division qui est affecté à la valeur de gauche : la position en x du nénuph, puis idem pour la position en y. L'intérêt est la certitude d'une valeur comprise entre les bornes de l'écran.

```
    Struct nenuph init_nenuph( ){
(1) struct nenuph  n;
(2)       n.position.x        = rand()%ecranx ;
(3)       n.position.y        = rand()%ecrany ;
(4)       n.speed.x           = (rand()%10)-5 ;
(5)       n.speed.y           = (rand()%10)-5 ;
(6)       n.rayon             = (rand()%20)+20;
(7)       n.perception        = 50-n.rayon ;
(8)       n.BonneHumeur       = rand()%100 ;
(9)       n.caractere         = rand()%10 ;
(10)      n.couleur           = (n.perception * n.BonneHumeur)%256 ;
(11)      return n ;
    }
```

Fig 3.8 : Initialiser une structure nenuph

(4) et (5) Même principe pour le mécanisme d'attribution de valeur à la variable attribuée aux déplacements. Le « – 5 » ajoute la possibilité d'avoir une valeur négative ou positive comprise entre –5 et 4 . Si rand()%10 renvoie 7, ça donne 7-5 = 2. Si rand()%10 renvoie 1, ça fait 1-5 = -4. L'avantage, c'est d'avoir des déplacements dans toutes les directions. Si par exemple le nénuph est à la position (160, 200) dans l'écran et que ses valeurs de déplacements sont (-3) pour speed.x et 4 pour speed.y, le déplacement sera :

```
position.x = position.x+ speed.x ;          // position.x vaut 160 + (-3), c.a.d. 157
position.y = position.y+ speed.y;           // position.y vaut 200 + 4, cad 204
```

Maintenant, le nénuph est à la position (157, 204), il est allé vers la gauche de –3 pixels et est descendu vers le bas de l'écran de 4 pixels.

(6) Il s'agit de déterminer le rayon du nénuph qui a la forme d'un cercle, c'est-à-dire sa taille. Le petit dispositif de contrôle fait qu'elle sera comprise entre 20 et 39 : si rand()%20 renvoie 0, 0+20 = 20 ; si rand()%20 renvoie 19 qui est le maximum, 19+20=39.

(7) Il s'agit de déterminer la capacité perceptive de notre nénuph. L'idée ici est de poser qu'elle est inversement proportionnelle à la taille du nénuph. Plus le nénuph est petit, plus sa perception est grande. Et inversement plus il est grand, plus sa perception est petite. Nous avons choisi arbitrairement la constante 50 comme repère dans cet effet de proportion.

(8) Idem pour la bonne humeur, nous avons pris une valeur de 100 pour établir plus tard dans le programme une échelle de la progression de l'humeur. Cette variable permettra d'établir des comparaisons entre les entités, et sur la base de ces comparaisons influencer telle ou telle décision dans un sens ou dans un autre. Par exemple, admettons que l'humeur joue un rôle sur l'aptitude à se déplacer du nénuph. Plus son humeur est sombre (c'est-à-dire proche de 0), moins il se déplace. D'autre part l'humeur pourrait être communicative et permettre des influences réciproques entre nénuphs pondérées d'ailleurs par le trait de caractère.

(9) Si le caractère est bon, il y a une bonne influence ; en revanche, elle peut s'avérer catastrophique …Il serait intéressant de définir des caractères qui serviront à colorer des processus relationnels entre les entités. Admettons qu'il y ait en tout et pour tout 10 possibilités de caractères. Evidement, on peut envisager de faire évoluer les caractères des entités. Le caractère d'une entité peut se dégrader ou s'améliorer par exemple en fonction de compositions de couleurs réalisées par l'ensemble de la communauté des nénuphs.

(10) La couleur d'un nénuph résulte de la bonne humeur mise en relation avec la perception. Comme l'humeur sera amenée à varier, la couleur aussi et donc les caractères également, etc.

(11) La fonction retourne une structure de nénuph avec des valeurs initialisées.

On remarque que l'on arrive assez vite à des interactions assez complexes. C'est là qu'intervient la recherche et la création de modèles plus construits des relations entre nos entités. Sinon, on finit par ne plus rien comprendre à son ensemble de variables interconnectées. Il faut alors beaucoup de chance pour arriver à un système qui fasse état d'une cohérence intéressante, et surtout on ne peut plus agir sur un tel système car on ne sait plus comment il marche. Il n'est plus possible d'établir des comportements et des caractéristiques aux nénuphs.

Mais nous nous limitons ici à la fonction de déplacement et la fonction d'affichage. Nous allons faire extrêmement simple. D'ailleurs, on va regrouper les deux. Ce n'est probablement pas une bonne idée pour la suite du programme, mais ça ne sera pas trop difficile à corriger. Pour l'instant, nous sommes dans un contexte de test.

Pour ce faire il faut revenir au langage et introduire un autre ensemble de types, *le tableau* ainsi que la notion de boucle avec l'instruction *for*.

Le type tableau

Tous les éléments d'un tableau sont obligatoirement du même type. Ensuite, le tableau doit avoir une taille définie au moment de sa déclaration. Par exemple :

`int tab[10] ;`	Déclare un tableau de 10 entiers nommé « tab » et chaque entier est accessible via un indice du tableau. Toutes les positions du tableau sont numérotées de 0 à 9. La position « 0 » compte pour une position.

Avec un tableau, je vais pouvoir stocker un ensemble de nénuphs. Par exemple 143 nénuphs et avoir une petite communauté :

```
struct nenuph NENUPH[143] ;
```

Déclare un tableau de 143 nénuphs. Toutes les positions du tableau sont numérotées de 0 à 142. Chacune est accessible via un indice du tableau.

« NENUPH » est l'identificateur du tableau et « [ ] » l'opérateur de déclaration utilisé également pour l'indiçage du tableau, afin de sélectionner telle ou telle position dans le tableau. On a ainsi le nénuph 0, le 1, le 2 etc. jusqu'au nénuph 142 et on peut s'y référer dans le programme de la façon suivante : NENUPH[0], NENUPH[1], NENUPH[2] etc., notamment lorsqu'il est question de les initialiser :

```
NENUPH[0] = init_nenuph() ;
NENUPH[1] = init_nenuph() ;
NENUPH[2] = init_nenuph() ;
etc.
```

Ici la fonction d'initialisation est appelée et elle renvoie des valeurs telles que nous l'avons conçu, à partir de nombres tirés au hasard, pour chaque nénuph dans le tableau.

Boucle for

Le problème c'est qu'il y a 143 nénuphs et que ça va être long et fastidieux de tout écrire. Il existe des fonctionnalités essentielles du langage qui réalisent des boucles. En l'occurrence, la boucle « for » est indiquée. Son fonctionnement est le suivant :

```
for (i = 0 ; i < 143 ; i++) {

        (…instructions...)

}
```

« i » est une variable quelconque initialisée à 0. Ensuite, un test est effectué sur la variable i. Le test regarde si i est inférieur à 143. Si oui, alors le bloc d'instructions contenues entre les deux accolades est exécuté. Ensuite, i est incrémenté de 1 (i = i+1), le test est refait, si la nouvelle valeur de i est toujours inférieure à 143, alors les instructions sont à nouveau exécutées avec cette nouvelle valeur de « i » , etc.

Pour l'initialisation des nénuphs nous écrirons donc :

```
for (i = 0 ; i < 143 ; i++) {
  NENUPH[ i ] = init_nenuph() ;
}
```

i est à 0 au commencement. Il va prendre successivement les valeurs 1, 2, 3, 4 … jusqu'à 143 et là, le test « i < 143 » ? c'est-à-dire 143 est inférieur à 143 ? est faux et donc la boucle s'arrête.

Fonction de déplacement avec affichage

Voyons la fonction de déplacement. Le principe est très simple. Chaque nénuph est représenté par un cercle. Nous puiserons donc dans notre environnement de programmation une fonction déjà écrite pour le tracé de cercle. Ensuite, le processus est le suivant :

1. Le nénuph est effacé à sa position courante,
2. Ses coordonnées sont changées et
3. Il est affiché aux nouvelles coordonnées. D'autre part, nous ferons bouger tous les nénuphs au même moment.

Nous verrons des méthodes plus raffinées un peu plus tard. Notre fonction aura une allure du genre de celle-ci (figure 3.9) :

(1) Déclaration de la variable « i » utilisée dans la boucle.

(2) La boucle avec le test et l'incrémentation. « i » prend tour à tour les valeurs de 0, 1, 2, 3 …,142. A 143, le test est faux ; la boucle s'arrête.

```
    void deplace_affiche(){
(1) int i ;
(2)     for( i = 0 ; i < 143 ; i++){
(3)         cercle(  NENUPH[ i ].position.x,
                     NENUPH[ i ].position.y,
                     NENUPH[ i ].rayon,
                     COULEUR_FOND                    ) ;

(4)         NENUPH[ i ].position.x += NENUPH[ I ].speed.x ;
(5)         NENUPH[ i ].position.y += NENUPH[ I ].speed.y ;

(6)         cercle(  NENUPH[ i ].position.x,
                     NENUPH[ i ].position.y,
                     NENUPH[ i ].rayon,
                     NENUPH[ i ].couleur             ) ;
}
```

Fig. 3.9 : Déplacement d'un troupeau de 143 nénuphs

(3) Nous supposons que la fonction « cercle » existe. Elle a besoin de quatre arguments à partir desquels faire son calcul. Pour faire un cercle sur l'écran, il lui faut une position (x, y), un rayon r et aussi la couleur du cercle.
D'autre part, « COULEUR_FOND » représente la couleur du fond, c'est le moyen d'effacer. Si j'ai un cercle rouge sur un fond noir, je peux l'effacer en traçant le même cercle à la même position avec le même rayon mais avec la couleur noire.
Ici, le nénuph est effacé.

(4) et (5) La position du nénuph est modifiée. Remarque : pour ajouter une valeur lors d'une affectation, on peut utiliser l'opérateur « += » ; a=a+b et a+=b sont deux formulations identiques.

(6) Affichage du nénuph à ses nouvelles coordonnées.

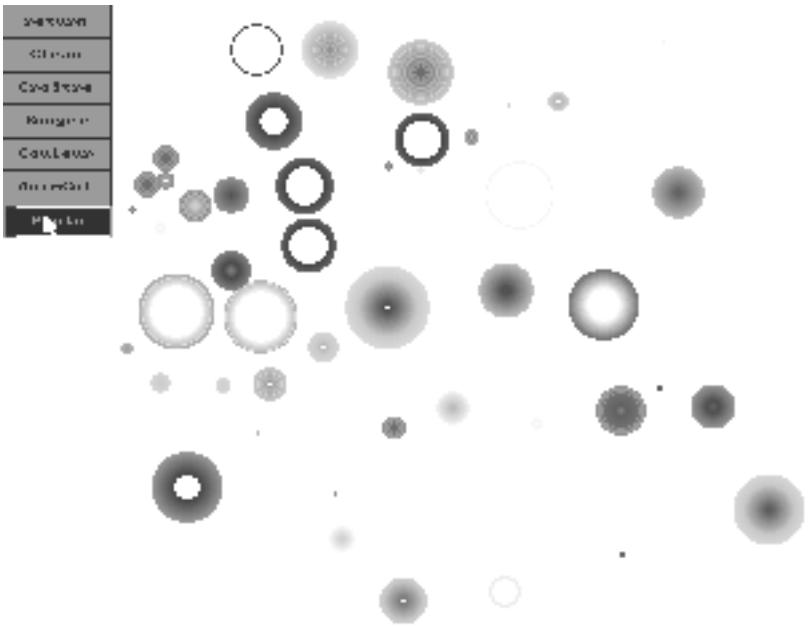


Fig 3.9 : Une version du programme des nénuphs en fonctionnement

Chapitre 4

Mécaniser des abstractions

Tout ce que les machinistes du théâtre classique ont su mettre en œuvre pour manipuler les cintres, descendre ou remonter les rideaux, changer les décors, animer les machines aux noms enflés, « gloire » ou « triomphe », destinées à l'envol des nymphes et des déesses dans le ciel du théâtre, tout cet illusionnisme circonscrit à l 'espace physique de la scène fait pendant à toute une panoplie de techniques utilisées en IA. Abondamment employés, les termes de représentation, d'acteurs, de scénarios ou de cadre en sont les témoins. Ce ne sont pas les seuls ; au jeu de cordes et de poulies, si semblable à l'accastillage des voiliers de haute mer, à tout l'attirail fait pour déployer et tendre les toiles peintes, ramener ou déplacer les cintres, correspond toute une mécanique subtile de pointeurs grâce à laquelle l'environnement d'un programme peut être modifié.

Jean-Gabriel Ganascia

Dans le chapitre précédent nous avons précisé comment nous nous approprions une démarche de modélisation au sein du projet, ainsi que l'articulation de cette activité avec le dynamisme global du projet. Le fonctionnement de la modélisation au sein du projet peut prendre différentes formes. Il n'y a pas d'approche ou de méthodologie unique et nous avons adopté une optique d'auteur, ce qui est pleinement conforme avec la subjectivité impliquée dans la construction de modèles. Ensuite nous nous sommes rapprochés d'une zone très sensible : la bascule entre le monde des idées et des connaissances en général et celui de la machine et de la programmation.

Compte-tenu de cette trajectoire qui aligne idées, projet, modélisation, programmation, nous sommes maintenant passés au dessus du clivage d'avec la technique, du côté de la machine. Nous entrons, comme dans les coulisses d'un théâtre, pour découvrir différents aspects théoriques et techniques, autre angle de vue sur la modélisation et la formalisation qui sont maintenant soumises à une gamme très restreinte de figures comme les listes et les arbres. En réalité c'est pourtant un champ d'investigation très large. Après quelques précisions sur le sens de la notion d'abstraction, nous optons ici pour une présentation sommaire des outils fondamentaux : modèle de données, algorithme et structure de données.

1. Abstraction

Pour obtenir une représentation manipulable dans un ordinateur il faut simplifier la complexité du réel. Il faut réaliser une abstraction c'est-à-dire abstraire une situation et des données de cette complexité. L'abstraction est un synonyme de modèle. Mais elle exprime le modèle surtout du point de vue de ses troncatures par rapport au réel. Par exemple la question du sens est réduite à l'étude des mécanismes du langage formel et de programmation mis en œuvre. Sur le versant de l'abstraction, le modèle devient « un mécanisme sémantique d'évaluation de formule ». Et par sémantique il faut comprendre comme Jean-Marc Alliot et Thomas Schiex le précisent « que nous allons donner un sens à nos atomes, puis donner un sens à une formule à partir de la valeur de vérité des atomes qui la composent et des connecteurs qui relient ces atomes » [ALLIOT, 1993, p. 46]. Le sens tel qu'il apparaît dans un langage naturel est largement évacué.

1.1 Pragmatique de l'usage

Nous sommes là dans un univers abstrait par rapport à la totalité de l'univers du projet. Il ne reste plus que la composante des mécanismes à l'œuvre dans les langages utilisés. A cet égard nous avons affaire à une pragmatique de l'usage et de l'intériorité technique. Autre motif d'abstraction d'ailleurs, cette pragmatique révèle des contraintes restrictives d'une part sur les tâches que peuvent accomplir les ordinateurs et d'autre part sur la vitesse à laquelle ils peuvent les accomplir.

Formes

D'abstraction en abstraction on arrive à un vocabulaire relativement limité de formes exploitables pour la construction de programmes et auxquelles correspondent à des techniques de programmation. Chacune de ces formes va établir une certaine approche du raisonnement. Du point de vue de la créativité, est en jeu, pour un modèle donné, la capacité de reconnaître la forme la plus adaptée qui va représenter ce modèle en machine. Dès que l'on sait reconnaître une de ces formes on a la base sur laquelle appuyer la programmation. Or nous n'avons dénombré que quatre types de formes qui sont sujets d'étude dans tous les ouvrages d'algorithmique informatique : Arbre, liste, ensemble et graphe. Et encore, on peut alléguer que l'implémentation machine des notions d'ensemble et de graphe utilisent des listes et des arbres. Listes et arbres sont la base algorithmique fondamentale présente dans la littérature technique informatique [SEDGEWICK, 1991]. Alfred Aho le souligne : « Le progrès dans la modélisation, ou l'abstraction, du monde réel ou de ses principaux composants reste un défi majeur de l'informatique qui a peu de chance d'être totalement relevé dans un futur proche » [AHO, 1993, p.4].

Algèbres

A ces principes de formes s'ajoute la notion d'algèbre. Une algèbre « désigne toute sorte de système dans lequel il y a des opérandes et des opérateurs à partir desquels on construit des expressions » [AHO, 1993, p. 374]. Une algèbre est un langage formel développé dans le cadre d'un modèle connu, comme une notation pour faciliter l'expression et l'évaluation des alternatives de conception.

Alfred Aho et Jeffrey Ullman décrivent plusieurs algèbres de conception notamment l'algèbre booléenne qui entre dans la conception de circuits et de composants d'ordinateurs. Plus connue sous les termes de logique formelle ou logique des propositions. La logique formelle repose sur l'application de règles d'inférence à des faits telle que « si X est un chat et Y est la mère de X alors Y est un chat » [AHO, 1993, p.5]. La logique des prédicats est une généralisation de la logique propositionnelle. Nous nous sommes référés à la logique des prédicats dans le chapitre précédent à propos de la formalisation et de la définition de concepts afin d'illustrer la perte du sens linguistique d'un langage formel par rapport à un langage naturel. A l'origine, la logique des prédicats est utilisée dans la modélisation du raisonnement, c'est une algèbre de conception intégrée comme modèle de calcul dans des langages de programmation fondés sur la logique, comme le langage « Prolog ».

L'intérêt d'une algèbre va être surtout d'aider à bâtir une théorie de la conception dans un domaine particulier, et par là d'ouvrir le champ à des possibilités nouvelles. [AHO, 1993, p. 6, 374]

Sans rentrer dans les détails d'une implémentation machine, mentionnons brièvement l'aspect que peut prendre une des formes canoniques sous-jacentes à la conception informatique : la figure du graphe.

1.2 Exemple d'une figure : le graphe

Un graphe est un ensemble d'objets appelés sommets (ou nœuds) et de relations entre ces sommets désignées par des arêtes (ou arcs). Il permet de représenter différentes sortes de situations, en termes d'objets et de liens entre ces objets. On dispose ensuite des techniques qui permettent de programmer les graphes. L'exemple le plus évident est le réseau de communication. Mentionnons également l'exemple de « Fify » donné par Alfred AHO (figure 4.1)

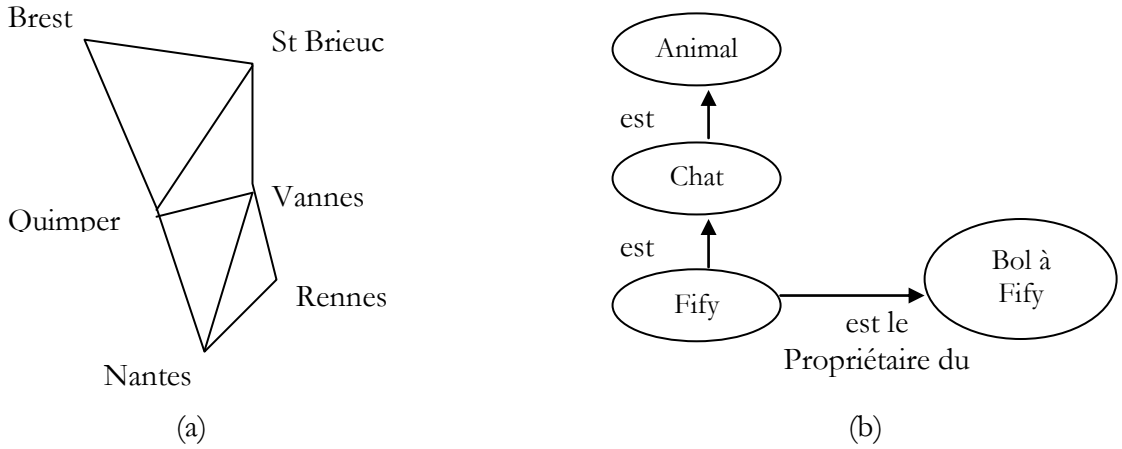


Fig. 4.1 : (a) réseau de communication et (b) des informations sur Fify

On pressent, à travers ces exemples de l'utilisation de cette figure, tout l'art qu'il faut mettre pour réussir à représenter une situation de façon telle qu'elle puisse entrer dans le cadre d'une des figures de référence. L'objectif est pratiquement de piéger des modèles dans ces sortes de trappes ou de modéliser en les utilisant comme charpente initiale.

1.3 Voisinage simplification - complexification

Ce qui est véritablement phénoménal sur le versant de l'abstraction, c'est que plus on s'enfonce au creux de la machine et que l'on se rapproche du processeur, moins il y a de principes. Une terrible simplification semble s'opérer. Le processeur en tout et pour tout ne fait plus que deux choses : consulter le contenu d'un emplacement en mémoire et laisser sa valeur inchangée, ou modifier le contenu d'un emplacement en écrasant la valeur contenue par une nouvelle valeur14. C'est uniquement à partir de là que tout le reste est conçu. En revanche dans le trajet inverse, en remontant vers « la surface », la complexité des opérations rendues possibles est considérablement accrue. L'élaboration de cet accroissement est évidemment un enjeu de recherche pour la conception et l'appréhension du réel avec des programmes.

[14] Philippe Mercier dans un livre sur l'assembleur précise : « Le microprocesseur est le cœur de l'ordinateur, il peut exécuter un nombre limité d'instructions simples : déplacement de données en mémoire, inversion de bits (NOT), opérations logiques (AND, OR, XOR), addition de bits ; déplacement et rotation de bits » [MERCIER, 1989, p. 28].

Le principe de la recherche s'est historiquement appuyé sur l'idée de reproduire ou reconstruire l'activité rationnelle humaine. Il y a différentes approches auxquelles ont souvent correspondu des paradigmes de pensée, des visions du monde et des credo associés. Essentiellement deux grandes familles.

Celle qui repose sur une physiologie de l'activité rationnelle. C'est-à-dire qui s'intéresse surtout aux processus physiques de la pensée en exercice. Ce sont les réseaux neuromimétiques pour lesquels « la pensée se *fonde sur* ou *émerge* d'éléments de niveaux inférieurs ». L'autre famille est tournée vers le calcul sur des symboles : « Penser c'est calculer sur des symboles ». C'est l'approche logique, les Systèmes à base de connaissance avec des problématiques de représentabilité de la connaissance et de formalisation du raisonnement [CAPLAT, 2002, p. 22].

De plus, chacune de ces deux familles peut être envisagée dans une dimension individuelle ou collective. L'approche physique dans une dynamique collective conduit vers les algorithmes génétiques. C'est-à-dire l'introduction des notions d'ordre, de chaos, et surtout d'une interrogation nouvelle sur l'émergence à partir de phénomènes d'évolution. L'approche symbolique dans une dynamique collective donne elle des systèmes multi-agents avec des problématiques de communication et d'organisation.

Récemment une nouvelle voie est travaillée avec les systèmes adaptatifs. Ils sont essentiellement un croisement des systèmes neurominétiques, des algorithmes génétiques et de quelques processus empruntés aux systèmes multi-agents [CARDON, 2000].

Il convient de constater que l'informatique fondamentale est un univers très créatif et très difficile. Il faut vraiment qu'il y ait une sorte de génie humain pour sortir quelque chose d'une machine aussi rigide et bornée qu'un ordinateur ! Dépasser réellement ce que l'on connaît des langages formels, n'irait probablement pas sans une révolution profonde de nos modes de pensée actuels. Mais c'est peut-être exposer une bonne part du problème de l'humanité avec elle-même.

1.4 Trois outils fondamentaux

Les sujets et thèmes principaux de l'informatique sont généralement regroupés en trois volets fondamentaux : modèles de données (selon la terminologie de [AHO, 1993]) ou types abstraits (selon la terminologie de [FROIDEVAUX, 1993]), structures de données et algorithmes. Quatre vingt dix pour cent de la littérature technique informatique pour la programmation, mis à part les ouvrages de formation à des langages, sont consacrés à ces sujets. Et chose curieuse, ils sont très semblables les uns aux autres. En fait ce ne sont pas des ouvrages à thèse comme ceux que l'on peut rencontrer en Intelligence Artificielle ou en sciences cognitives. Ce ne sont pas des ouvrages qui interrogent sur les moyens de reproduire la rationalité humaine. C'est beaucoup plus technique et de ce point de vue beaucoup plus modeste, mais en même temps non dénué de charmes. Ce sont des ouvrages descriptifs d'un savoir qui s'est élaboré doucement et collectivement depuis les premiers ordinateurs[15].

[15] Ce savoir résulte d'un travail énorme. Les algorithmes font l'objet de véritables découvertes. Souvent ils portent le nom d'un auteur même s'ils procèdent toujours à un niveau ou à un autre d'un effort collectif ou

2. Modèles de données

Listes, arbres, ensembles et graphes cités plus haut sont des modèles de données. Les modèles de données sont des abstractions utilisées pour formuler et décrire des situations problématiques dans le cadre de réalisations informatiques.

2.1 Définition

Le modèle de données est une forme, un aspect conceptuel éventuellement mathématique, juste en amont de l'implémentation machine. C'est par exemple *l'idée ou la forme de liste*, *l'idée ou la forme d'ensemble*. Derrière ces formes et ces idées il y a des algorithmes et des structures de données aujourd'hui parfaitement maîtrisés. Mais dans le modèle de données, elles sont présentées au travers de la description des opérations effectuées, détachées de toutes contingences d'implantation particulière. Elles sont comme désincarnées. Ainsi pour un même modèle de données vont correspondre plusieurs possibilités d'algorithmes, de structures de données et de choix de langages. Si vous dites « liste » à un informaticien il y a immédiatement différentes options de programmation. De même pour « graphe » ou « arbre ». Ce sont pour ainsi dire les primitives du vocabulaire de la conception. C'est pourquoi les notions de graphe, de liste, d'ensemble, et d'arbre sont omniprésentes dans tous les ouvrages.

Toutefois il existe des modèles de données qui sont des extensions des fondamentaux ; comme les graphes et les ensembles sont reliés aux arbres et aux listes par exemple. Mais il ne faut pas oublier que graphes et ensembles ont une identité et représentent un espace spécialisé même s'il est fait appel à des listes ou des arbres pour les représenter. Ainsi Alfred Aho ajoute à ces primitives de formes d'autres modèles de données. Le modèle de données relationnel (initialement développé pour les bases de données, en ce sens stockage et gestion de l'information) les automates à états finis (qui utilisent des graphes), les grammaires (arbres, graphes, proximité des automates, définition des langages) et la logique (arbres, algèbre Booléenne) [AHO, 1993]. Chaque modèle de données est en lui-même un objet d'étude assez difficile.

---

de la collectivité. Alfred Aho, Ravi Sethi et Jeffrey Ullman le soulignent ainsi à propos de la programmation des compilateurs : « Pendant toutes les années 50, les compilateurs furent tenus pour des programmes difficiles à écrire. Par exemple, la réalisation du premier compilateur Fortan nécessita 18 hommes-années de travail. On a découvert depuis des techniques systématiques pour traiter la plupart des tâches importantes qui sont effectuées pendant la compilation. En outre, on a développé des langages d'implantation, des environnements de programmation et des outils logiciels bien adaptés au développement des compilateurs. Avec tous ces progrès, on peut même développer un compilateur substantiel au cours d'un projet d'étude associé à un cours de compilation d'un seul semestre. » [AHO, 2000, p.16].

2.2 Modèle de données d'un langage de programmation

La notion de modèle de données est également utilisée par Alfred Aho et Jeffrey Ullman pour définir les caractéristiques d'un langage de programmation, c'est-à-dire l'ensemble des concepts sur lesquels le langage de programmation repose. Les concepts de fonctions, tableaux, structures, entiers que nous avons rencontrés dans le chapitre précédent sont rattachés au modèle de données du langage C, également à celui du langage Pascal, quoiqu'ils présentent tous deux quelques légères différences d'écriture et de définition. Par rapport à un concept mathématique, les caractéristiques d'un langage de programmation illustrent bien l'abstraction éventuellement opérée dans le modèle de données. L'exemple classique est celui de l'implémentation machine des entiers et des réels.

Entiers et réels sont initialement des concepts mathématiques. En machine ce ne sont plus que des abstractions de ces concepts. En effet du fait de la finitude de la mémoire d'un ordinateur mais aussi de sa construction, la précision des entiers et des réels sera nécessairement limitée et tronquée. Par exemple un nombre codé sur trente deux bits pourra prendre une valeur positive maximum de 4 294 967 295 (2^{32}-1). Ce qui veut dire que l'entier ou le réel informatique est limité par rapport à celui du principe mathématique, il n'aura donc pas un comportement identique. D'où l'identité de la notion de « modèle de données » qu'il faut prendre en considération même lorsqu'elle apparaît comme une *abstraction* d'un homologue mathématique.

2.3 Terminologie des types abstraits, méthodologie

Pour Christine Froidevaux, Marie-Claude Gaudel et Michèle Soria la terminologie retenue est celle de type abstrait. Leur définition est orientée davantage vers un principe de conception, une méthodologie. Un algorithme difficile fait l'objet de « raffinements successifs ». En général la première version sera même indépendante de toute implémentation particulière. La représentation des données elle-même ne sera pas déterminée. « A ce premier niveau, les données sont considérées de manière abstraite : on se donne une notation pour les décrire ainsi que l'ensemble des opérations qu'on peut leur appliquer et les propriétés de ces opérations. On parle alors de type abstrait de données. La conception de l'algorithme se fait en utilisant les opérations du type abstrait » [FROIDEVAUX, 1993, p.53]. Ensuite le type abstrait fait l'objet d'une ou plusieurs représentations en fonction des différents langages de programmation que l'on souhaite utiliser.

Robert Sedgewick mentionne un autre aspect toujours dans le cadre d'une méthodologie. Selon lui les « types de données abstraites » ont été rendus nécessaires du fait de programmes trop volumineux pour être maîtrisés tels quels. Il s'agit alors d'un principe méthodologique d'organisation de programmes : « Les types de données abstraites facilitent la compréhension des programmes de grande taille et rendent plus aisée toute modification, voire amélioration, des algorithmes fondamentaux » [SEDGEWICK, 1991, p.33]

Modèles de données, types abstraits, ou encore types de données abstraites sont à peu près synonymes. Mêmes principes dans différentes terminologies avec juste quelques petites variations dans l'approche. Le « modèle de données », selon les travaux de Alfred Aho et

Jeffrey Ullman, inclut la définition des concepts d'un langage et est de ce point de vue un peu plus général.

Ce principe qui définit des abstractions est indissociable des notions d'algorithmes et de structures de données.

3. Algorithmes

L'algorithme est un déroulement précis d'opérations précises en vue d'accomplir une tâche particulière. Dans les termes de Seymour Lipszchutz « L'algorithme est une liste d'instructions qui permet de résoudre un problème particulier en opérant étape par étape » [LIPSCHUTZ, 1982, p. 95].

3.1 Exemple et précision de la définition

Par exemple imaginons un personnage ivre mais lucide assis dans un fauteuil à quatre mètres de la porte de sortie de la pièce où il se trouve. Le feu déclenché à l'étage l'oblige à s'en aller rapidement. Il doit réfléchir à chacun des gestes à faire pour réussir à sortir de la pièce. 1) S'extirper du fauteuil en poussant avec les bras. 2) Basculer légèrement vers l'avant et trouver appui sur ses deux pieds 3) Profiter de cette impulsion pour orienter la chute fatale en direction de la porte de sortie 4) S'agripper à la poignée et reprendre souffle 5) Etudier le mécanisme d'ouverture et tourner la poignée dans le bon sens après rétablissement de l'équilibre 6) Se présenter face à l'ouverture··· Cet exemple est intéressant et il renvoie à la définition de Alfred Aho pour lequel « un algorithme est une spécification précise et non ambiguë d'une séquence d'étapes pouvant être exécutées de façon automatique » [AHO, 1993, p. 5]. La spécification des étapes doit être non ambiguë et notre exemple montre que discrétiser une continuité du réel n'est pas forcément facile. Si l'on voulait s'amuser à décrire exactement toutes les étapes requises pour que notre personnage atteigne effectivement la sortie il en faudrait probablement une infinité. C'est pourquoi Christine Froidevaux précise dans sa définition que « un algorithme est la composition d'un ensemble fini d'étapes, chaque étape étant formée d'un nombre fini d'opérations » [FROIDEVAUX, 1993, p 5]. Les questions que se pose notre personnage seraient probablement importantes dans le domaine de la robotique.

3.2 Algorithme et notation

Un algorithme doit être exprimé dans un langage de programmation pour pouvoir être exécuté par un ordinateur mais il demeure cependant indépendant de toute implantation particulière. L'algorithme d'Euclide pour trouver le plus grand commun diviseur (pgcd) peut être programmé dans des langages différents. C'est toujours le même algorithme. Pour la notation d'un algorithme elle s'effectue généralement à partir d'un langage de programmation assez répandu, comme le C ou le Pascal. Afin d'optimiser la lisibilité de

l'algorithme, le langage utilisé est parfois enrichi de quelques constructions destinées à éviter des détails de programmation. Souvent on rencontre un mélange de langage naturel et de programmation.

Lisibilité d'un algorithme

Plus l'algorithme est simple et facile à comprendre mieux ça vaut. Malgré tous les efforts il arrive de se retrouver devant un programme dont on ne comprend plus le déroulement. C'est parfois plus difficile de reprendre le programme d'un copain que de tout refaire soi-même. Un des objectifs est de rendre autant que possible l'algorithme accessible à différents lecteurs futurs, parmi lesquels l'auteur lui-même figurera éventuellement. Il y a d'ailleurs des règles méthodologiques qui permettent d'augmenter la lisibilité d'un algorithme. Outre la nécessité d'une bonne documentation, la présentation du code, dans le cadre d'une communauté de programmeurs, est généralement soumise à un style de programmation qui est aussi une discipline de programmation. Par exemple la modularité est fortement recommandée. Afin d'éviter un algorithme trop long, il vaut mieux le découper en parties isolées les unes des autres à chaque fois que c'est possible. Et les liaisons ensuite entre les différents modules ont intérêt à être les plus claires possibles. En général au-dessus de quatre à cinq mille lignes de code, si la rigueur n'a pas été suffisante dans l'organisation du travail comme dans l'écriture, le programme ne peut plus être travaillé. Il se trouve paralysé.

Ce point est important pour l'optique de créativité et la notion de projet que nous avons présentées. Compte-tenu du souhait de faire évoluer les idées au sein du chantier de programmation il est indispensable d'avoir une grande rigueur dans la méthodologie de programmation. Sinon très vite le bruit du code paralyse le développement.

3.3 Temps d'exécution, analyse de la complexité d'un algorithme

Un autre aspect important pour les algorithmes concerne le temps d'exécution des programmes. Si le traitement est bon mais qu'il faut deux ans pour le faire, mieux vaut essayer autre chose. La rapidité d'exécution est relative au nombre des données prises en entrée. Par exemple dans le programme des nénuphs que nous avons commencé ce sera 143 nénuphs. Chaque nénuph subit un traitement à peu près identique au sein d'une boucle comme la boucle for.
Pour évaluer la vitesse du traitement le principe est d'établir le rapport, à travers une fonction mathématique, entre le nombres « n » des données entrées et le temps d'exécution. Evidement le temps de calcul augmente avec le nombre des données traitées. Mais cette augmentation dépend du fonctionnement de l'algorithme lui-même.
Par exemple si un algorithme pour chaque donnée en entrée doit consulter toutes les autres on aura une fonction du type $T(n) = n^2$. C'est ce que l'on appelle une complexité quadratique. Dans ce cadre, s'il y a deux fois plus de données en entrée, n double de valeur mais le temps de calcul est lui multiplié par quatre (Si $T(n) = n^2$ on a $T(nx2) = (nx2)^2 = n^2 \times 2^2 = n^2 \times 4$).
Ce cas de figure pourrait être trouvé dans le programme des nénuph à propos de la détection de collisions ou de rencontres entre nénuphs. Chaque nénuph, en fonction de sa position et de sa taille devra évaluer sa distance avec chacun des autres nénuphs, en fonction de leurs tailles et positions, pour voir s'il y a ou non une collision ou une

proximité et avec lesquels.

Voici un exemple donné par Alfred Aho et Jeffrey Ullman [AHO, 1993, p. 101]. Deux fonctions, T1(n) et T2(n) correspondent chacune au temps pris par un algorithme pour n données traitées. Elles sont définies ainsi : T1(n) = 100n et T2(n) = $2n^2$ (en millisecondes).
Pour n = 1, c'est-à-dire pour une donnée en entrée, T1(1) = 100 et T2(1) = 2. Si on trace les deux courbes on s'aperçoit en les comparant que T2 est plus rapide pour un nombre de données inférieure à 50. En revanche l'évolution de T1 est plus rapide que celle de T2 pour des nombres supérieurs à 50.

T1(50) = 5 000, T2(50) = 5 000
T1(100) = 10 000, T2(100) = 20 000
T1(1000) = 100 000, T2(1000) = 2 000 000
Pour 100 en entrée T1 est deux fois plus rapide que T2.
Pour 1000 en entrée T1 est 20 fois plus rapide que T2.
T1 croît de façon linéaire et T2 s'envole progressivement du fait de la quadratique n^2.

Cette comparaison des deux approches T1 et T2 permet au passage de savoir non seulement quelle est la plus avantageuse mais aussi d'avoir une meilleure conscience de la taille du problème en fonction du temps.

Alfred Aho donne les résultats suivant pour la comparaison de T1 et T2 (figure 4.2) :

TEMPS en secondes	Nombre des données traitées par T1	Nombre des données traitées par T2
1	10	22
10	100	70
100	1000	223
1000	10000	707

Fig. 4.2 : Comparaison des fonctions T1 et T2 pour le coût en temps de deux algorithmes

Remarquons que l'analyse de complexité fait valoir le coût en temps de l'algorithme indépendamment de la puissance de l'ordinateur utilisé. Cette donnée qualitative d'un algorithme relativise d'ailleurs la puissance du processeur. L'exemples de T1 et T2 le montre bien.

Avec un processeur dix fois plus puissant je ne vais pas dix fois plus vite mais je fais par exemple en une seconde ce qui était fait en dix, ou en dix secondes ce qui était prévu en cent. Et là tout dépend de l'algorithme. Pour T1 si le processeur fait en une seconde ce qui était prévu en dix, il obtient cent données traitées, j'obtiens effectivement dix fois plus de données traitées. En revanche ce n'est plus vrai pour la fonction T2. Si le processeur fait en une seconde ce qui était prévu en dix, il obtient soixante dix données traitées, c'est-à-dire uniquement trois fois plus. Multiplier la puissance du processeur ne signifie pas obligatoirement multiplier dans la même proportion le nombre des données traitées dans un même temps. Les propriétés des algorithmes contribuent à la rapidité d'exécution.

Classification des algorithmes

L'étude des performances des algorithmes a permis de connaître un ensemble de formules mathématiques à partir desquelles une classification des algorithmes est opérée. Il y a sept fonctions essentielles auxquelles le programmeur peut se référer pour évaluer le coût en temps de son idée d'algorithme. En effet 90% du temps d'exécution d'un programme est généralement passé dans 10% du code (c'est la règle des 90-10) il est donc possible de faire rapidement des approximations assez justes. Il suffit d'isoler la portion de code qui prend l'essentiel du temps de calcul et de faire l'évaluation en se rapportant à la classification selon les formules mathématiques connues.

Ces formules sont les suivantes : soit N qui représente le paramètre de base, en général les données en entrée. On a sept termes principaux qui sont : « 1, logN, N, NlogN, N^2, N^3, 2^N (…) la complexité d'un programme particulier a toute chance d'être égale à une constante multipliée par l'un de ces termes plus quelques termes inférieurs » [SEDGEWICK, 1991, p 75].

Ces termes principaux sont classés par ordre de d'augmentation du processus de croissance en temps. « 1 » signifie que quelque soit le nombre de données en entrée c'est toujours la même vitesse. « logN » est une croissance logarithmique ; l'algorithme ralentit légèrement avec l'affluence des données. Pour « N » la vitesse de l'algorithme est directement proportionnelle au nombre des données en entrée ; si « N » est doublé le temps aussi. Avec un algorithme en « NlogN » la croissance est légèrement plus rapide que l'affluence des données ; si « N » est doublé le temps d'exécution est un peu plus que doublé. Ensuit vient la quadratique « N^2 » que nous avons déjà rencontrée. Puis « N^3 » avec une complexité cubique : chaque fois que « N » double de valeur, le temps de calcul est multiplié par huit. Enfin « 2^N » ou le temps croît de façon exponentielle ; un tel algorithme sera difficilement utilisable.

Dans notre démarche de création il peut être important de savoir reconnaître selon quelle formule tourne le programme afin de faire une estimation à la louche lorsque la vitesse et l'amélioration des performances sont un facteur important dans le choix de l'algorithme ou en vue d'une optimisation.

3.4 Choix des algorithmes

En général l'évaluation est bornée par une spécification des données en entrée. Il s'agit de repérer « le pire des cas » qui correspond à une configuration de données en entrée la plus défavorable pour l'algorithme. Celle qui n'est pratiquement pas acceptable. Mais aussi « le cas moyen » qui est la configuration type pour l'algorithme. Par exemple si on a un maximum de cinquante données en entrée avec une moyenne de 25, l'algorithme qui correspond à la fonction d 'évaluation T2 est plus avantageux du point de vue de la rapidité que l'algorithme qui correspond à T1 dans l'exemple donné plus haut. En revanche c'est le contraire à partir de cinquante et la différence est de plus en plus accablante pour l'algorithme de T2.

D'un autre côté le critère de la vitesse n'est pas obligatoirement pertinent. En tout cas gérer la vitesse n'est pas toujours choisir le plus rapide. Parfois il est nécessaire de ralentir

le traitement dans le cas de certains jeux, ou d'introduire des « timers » pour s'assurer que le jeu tournera toujours à la même vitesse quelque soit la machine.

Robert Sedgewick donne pour consigne de toujours commencer par mettre en place l'algorithme le plus simple. Peut-être suffira-t-il à résoudre le problème et sinon il servira de base pour des tests de performance. Juger de la pertinence d'un algorithme renvoie à des situations concrètes très précises où l'expérience du programmeur joue un rôle. Par exemple cette précision de Robert Sedgewick qu'il ne s'agit pas seulement de l'algorithme mais aussi de la situation plus générale dans laquelle se trouve le programme :

> « Si un algorithme doit être utilisé peu de fois sur des cas de taille raisonnable, il est souvent préférable de laisser la machine travailler un peu plus longtemps, parce qu'on aura choisi un algorithme un peu moins efficace, que de prendre le risque de perdre un temps considérable à concevoir une implantation sophistiquée. Bien sûr, on ne peut jamais exclure le risque d'avoir à utiliser le programme plus souvent que prévu au départ (…) » [SEDGEWICK, 1991, p. 88]

Pour la création d'un programme il y a presque toujours plusieurs solutions algorithmiques. Chacune présente des avantages et des inconvénients. Pour choisir il est intéressant de pouvoir évaluer les performances. Ajoutons qu'il n'est pas obligatoirement nécessaire d'avoir recours à une analyse mathématique rigoureuse. Cela peut même se faire finalement de façon très empirique, sur sa machine en mesurant le temps pris par tel algorithme, puis le temps pris par tel autre.

Mais quelque soit le choix de l'algorithme il est toujours lié à celui d'une structure de données.

4. Structures de données

L'algorithme est enchâssé dans une structure de données. Ce qui fait dire à Robert Sedgewick que «pour de nombreuses applications le choix des bonnes structures de données est en fait la seule décision cruciale de l'implantation : une fois ce choix opéré, il ne reste plus qu'à trouver des algorithmes simples » [SEDGEWICK, 1991, p. 15].

4.1 Définition

La structure de données est ce qui définit les éléments primitifs sur lesquels l'algorithme va s'appuyer pour fonctionner. Ces éléments primitifs peuvent être déjà implantés dans le langage ou faire l'objet d'une construction avec le langage.

D'autre part, du point de vue du modèle de données, ces éléments primitifs et les algorithmes associés viennent pour concrétiser un type abstrait dans un langage de programmation. Et il faut entendre que pour un même modèle de données il peut y avoir plusieurs possibilités de structures de données (et d'algorithmes).

4.2 Limite d'une structure de données

L'algorithme est intimement lié à la structure de données sous-jacente. En fait, cette structure va filtrer des opérations possibles. Dans l'exemple des nénuphs notre structure de données est un tableau. Nous avons stocké nos nénuphs dans un tableau. Or la définition du tableau veut qu'un tableau ait une taille fixe. C'est-à-dire un nombre déterminé d'éléments et sur lequel on ne peut pas agir dans le programme. La déclaration « struct nenuph NENUPH[143] ; » vue dans le chapitre précédent signifie que nous pouvons disposer dans le programme d'un ensemble nommé « NENUPH » de 143 structures « nénuph ».

Si on décide que la communauté peut se reproduire et que de nouveaux nénuphs vont apparaître, même si l'on inclut des clauses de mortalité, il est possible que le nombre de nénuphs dépasse la taille du tableau ; on pourrait avoir 178 nénuphs à un moment donné. Dans ce cas la structure de données choisie ne sera pas adaptée. Un tel dépassement peut « planter » la machine. L'incertitude quant au nombre des nénuphs fait qu'il vaut mieux se tourner vers une autre structure de données, par exemple une *liste chaînée*, et abandonner le tableau.

Le tableau est une structure de données qui est une primitive du langage C ; ce qui n'est pas le cas de la liste chaînée. la structure de données liste chaînée peut néanmoins être une primitive d'un langage, en Lisp par exemple. Mais dans le langage C la liste chaînée fait l'objet d'une construction.

4.3 Présentation du concept de liste chaînée

Pour présenter ce concept il faut d'abord préciser que tous les objets informatiques que nous avons présentés, qu'il s'agisse des variables, des fonctions, des tableaux, des structures etc., tous correspondent à une inscription, localisée quelque part, dans la mémoire principale, tous ont une « adresse » dans la mémoire principale[16] (dénommée également mémoire vive ou RAM). Cette adresse est un nombre et il correspond à un emplacement de la mémoire.

Le principe de la liste chaînée est le suivant. On a des maillons reliés entre eux par un jeu d'adresses en mémoire. Chaque maillon est une structure qui contient l'adresse en mémoire de la structure suivante. Pour ce faire, à la structure nenuph que nous connaissons il faut ajouter un type particulier. C'est le type « pointeur ». Par définition il contient une adresse mémoire et permet d'accéder au contenu de la mémoire à cette adresse. Nous allons rajouter dans la structure nenuph un pointeur sur une structure nenuph.

[16] A la base de la construction de la mémoire d'un ordinateur, l'unité principale de stockage est l'octet. Un octet contient une valeur, une donnée d'un programme et chaque octet a sa propre adresse (en relation avec des emplacements physiques) qui le différencie des autres octets. Les adresses d'octets sont des nombres. Pour N octets en mémoire principale, il s'agit d'une numérotation de 0, 1, 2 ⋯ à N-1. La définition technique d'un octet est la « quantité minimum de stockage qui a sa propre adresse » [AHO, 1993, p. 168]. Mais la mémoire est construite afin de pouvoir localiser simultanément plusieurs octets consécutifs dans une seule adresse.

```
Struct nenuph{
   ⋯(caractéristiques vues au chapitre précédent)⋯
   ⋯ struct nenuph *suivant ;
} ;
```

L'opérateur « ***** » permet de définir un pointeur sur le type particulier déclaré à la gauche de cet opérateur.

Ensuite par un mécanisme d'allocation mémoire nous pouvons créer des structures nenuph. Avec le langage C nous disposons d'une fonction importante. Cette fonction réserve un emplacement mémoire selon une taille que l'on précise, et renvoie l'adresse de son emplacement. Créer une structure nénuph revient à disposer de l'adresse de son emplacement en mémoire.

Le mécanisme de la création d'une liste chaînée est le suivant. Une première structure nenuph est créée. Son adresse est gardée précieusement dans un pointeur spécialement nommé (par exemple « PREMIER »). Ensuite, une seconde structure nenuph est créée et l'adresse de cette seconde est affectée au pointeur sur nénuph que la première structure nénuph contient. Une troisième structure nénuph est créée et l'adresse de cette troisième structure nenuph est affectée au pointeur que la seconde contient. Etc. Chaque structure nenuph contient l'adresse d'une suivante, cette suivante contient l'adresse d'une suivante etc. (figure 4.3).

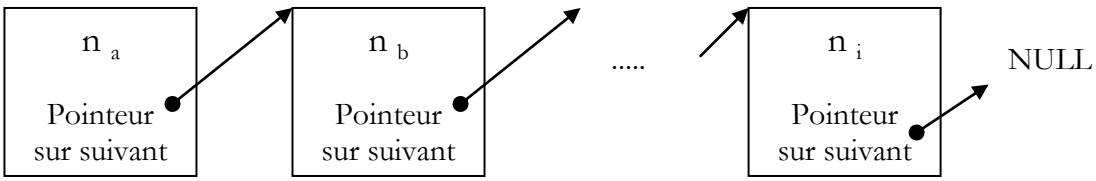


Fig. 4.3 : Chaînage représentant la liste (na , nb , ⋯ni)

L'important sera évidemment de ne jamais perdre la première et d'indiquer la fin avec un pointeur sur rien dit « NULL ». Au résultat on a une chaîne qu'il est possible de parcourir en commençant à la première structure.

Ainsi nous avons montré que pour un même concept de liste il pouvait y avoir plusieurs structures de données, tableau et liste chaînée. Chaque structure de données possède des propriétés spécifiques et il faut choisir la mieux adaptée en fonction de la situation.

4.4 Choix d'une structure de données

Par rapport à un tableau la taille d'une liste chaînée est amovible. A tout moment dans le programme il est possible de lui ajouter des éléments ou d'en soustraire et n'importe où dans la liste. De plus si on souhaite insérer le dernier élément d'un tableau en premier il va falloir déplacer tous les éléments du tableau de une case vers le haut. Dans le cas d'une liste, le dernier élément de la liste devient le premier élément de la liste et l'avant dernier le dernier. Bref, la liste chaînée s'avère plus souple que le tableau. Inconvénient majeur, il est impossible d'accéder directement à un élément de la liste sans passer par les précédents.

Dans le cas du programme des nénuphs, s'il y a des variations du nombre de la population du fait de mortalité ou de naissance, cette structure de données permet d'avoir un nombre amovible de nénuphs. Elle permet également d'enlever des nénuphs ou d'en ajouter n'importe où dans la chaîne. C'est très souvent une structure de ce type qui est utilisée dans les jeux dits « d'action » (En général il faut abattre des adversaires). Et pour les mêmes raisons : facilité d'ajouter de nouveaux adversaires et d'en retirer. C'est la structure de données que nous avons utilisée dans le programme « Kandinsky shoot again », une bataille spatiale dans les tableaux de Kandinsky.

Il y a d'autres variantes de la liste chaînée. Elle peut être doublement chaînée du début à la fin et de la fin au début. Elle peut également être circulaire si la dernière pointe sur la première. D'autre part tableau et liste chaînée peuvent être croisés. Il y a une possibilité d'implanter une liste chaînée par un tableau ce qui permet de faire cohabiter plusieurs listes dans un même tableau [SEDGEWICK, 1991, p. 24]. Mais il est plus fréquent de trouver un tableau de listes chaînées (chaque élément du tableau est une tête de liste) mais aussi de trouver des tableaux dans une liste chaînée. Ou encore on peut avoir un tableau de tableau (une matrice) ou une liste chaînée de tête de liste chaînée etc.

Nous pouvons constater que dans le cadre d'un modèle de données « liste » nous avons essentiellement deux concepts de structures de données : celui du tableau et celui de liste chaînée. Le modèle de données liste présente quant lui plusieurs aspects qui débouchent sur des propositions de listes spécialisées : les piles et les files. Mêmes alternatives, à base de tableaux ou de listes chaînées, pour l'implantation de piles ou de files.

L'algorithme, lui, est conduit par la structure de données. Elle induit des possibilités, un champ d'action que l'algorithme exploite. Modèles de données, algorithmes et structures de données sont interdépendants.

5. Interdépendance, orientations

Modèles de données, algorithmes et structures de données sont les trois pôles de la conception de programmes informatiques. Ce sont trois entrées sur un seul domaine qui est le passage du modèle à la programmation. Ils sont étroitement liés les uns aux autres à tel point qu'il n'est pas évident de les dissocier. Difficile de savoir quel parti est au commencement de la forme finale qui implique les deux autres dans un seul programme (figure 4.4).

Il s'avère souvent adéquat de commencer par une bonne structure de données. De là partir sur des algorithmes simples, et si nécessaire, récapituler le tout dans une figure d'ensemble. Mais parfois il faut au contraire partir d'une figure d'ensemble, d'un modèle de données, puis préciser l'algorithme et prendre la structure de données adaptée. Ou encore on a un algorithme et on cherche la structure de données qui répond le mieux à la situation, éventuellement on peut se référer alors à un modèle de données. Etc.

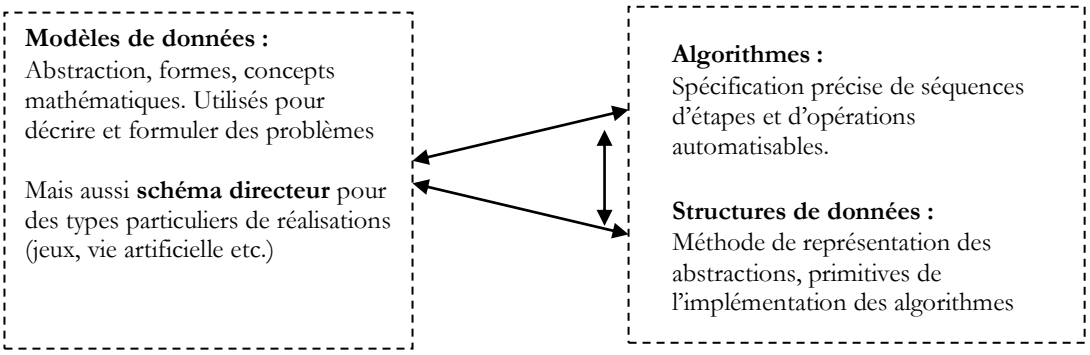


Fig. 4.4 : Interdépendance des modèles de données, algorithmes et structures de données

En fait, c'est tout un art. Toute la culture et l'expérience de programmeur s'ajoutent et se combinent. Différents paramètres personnels prennent un rôle important dans la réalisation. Une certaine forme de talent entre en jeu également. Parfois on est surpris par l'adresse, la subtilité, l'astuce ou la créativité d'un copain. Des fois c'est plus que ça et on est vraiment admiratif. Il y a aussi des cas où on ne comprend rien, ou pas grand chose !

Orientations

Nous utilisons la trilogie et proposons que la part du modèle de données corresponde finalement à *un schéma directeur*. Sur l'autre bord se trouve toujours la dialectique algorithmes et structures de données. Par schéma directeur nous entendons élargir la notion de modèle de données à des domaines spécialisés et personnalisés. Par exemple le modèle de données jeu, et parmi les jeux le modèle de données jeu d'action ou jeu de stratégie etc. On peut avoir également des modèles de données générateur de texte, ou traitement du son…Il s'agit de généraliser une figure en en faisant un modèle de données afin de rendre plus facile son interprétation, son écriture, sa lecture.
L'idée est de rendre plus accessible la programmation dans le champ de la créativité qui nous intéresse. De donner à cette créativité un vocabulaire de formes. En effet la négociation modèle-machine du chapitre précédent a lieu ici. Quelles que soient les idées que l'on puisse avoir eu, et le modèle qu'on a fait, si l'on veut obtenir un programme il va être impératif de pouvoir exprimer son idée selon au moins une de ces entrées, sachant que d'un pôle, il sera probablement possible de déduire ou de passer aux deux autres. D'où l'intérêt de disposer de schémas directeurs qui vont faciliter l'accès à la programmation. Dés que l'on dispose d'un schéma directeur dans le cadre d'un projet il agit pour la conception comme une véritable clef d'entrée sur la programmation.

Chapitre 5

Art, science, jeu ou bidouille ?

(…) il suffit, dit-il, de s'exercer en prenant une raie morte de petite taille, en lui retroussant les nageoires pectorales sur le dos, et en lui enroulant la queue dans la position voulue. En nouant un morceau de ficelle autour de la tête, derrière les mâchoires, on peut lui modeler un cou. Puis on fait sécher la raie au soleil. Peu après, en rétrécissant, les mâchoires deviennent proéminentes et adoptent une forme de museau. Un nœud de cartilages, jusqu'alors cachés, fait saillie à tel point qu'il évoque des bras repliés. Les narines, situées juste au-dessus des mâchoires, se transforment en deux yeux inquiétants, et les laminae olfactives ressemblent à des cils. On obtient ainsi par ce simple trucage, après application d'une couche de vernis et, le cas échéant, de quelques touches de peinture, une jenny haniver (chimère) dont l'aspect savamment étudié pique évidemment la curiosité des amateurs de singularités(…)

Peter Dance

1. Corps chimèrique

Art-science-jeu

Parti de la musique nous sommes devenus informaticien sans pour autant être infidèle à la démarche artistique. La démarche artistique incorpore progressivement une démarche scientifique. Mais une fois cette dynamique lancée, l'inverse est vrai tout autant. Il y a une démarche scientifique qui incorpore une attitude artistique. A ce point il n'y a pas réellement de distinction claire entre art et science. De même la frontière avec le jeu s'estompe tout autant. Par exemple le programme « Kandinsky shoot again » est initialement un jeu d'arcade classique. Mais il se déroule à l'intérieur d'un ensemble de tableaux de Wassily Kandinsky. On tire sur des éléments de tableaux avec des boules de couleur. Et des formes détourées des tableaux apparaissent comme dans une danse. Dans le programme « Kandinsky shoot again » la programmation informatique est un support scientifique, le sujet est artistique et la forme celle d'un jeu.

Bidouille

Une certaine aptitude au bricolage entre dans la réalisation. «La bidouille » qui se glisse par-ci par-là, de la même façon qu'on la trouve dans le spectacle de rue pour la réalisation de décors ou d'effets, ou même , éventuellement, dans des laboratoires scientifiques[17]. Dans notre domaine la bidouille éclot souvent du mélange des attitudes artistique et scientifique. Par exemple pour obtenir tel ou tel effet esthétique on n'est pas trop regardant sur l'élégance des moyens techniques mis en œuvre. Mais l'inverse peut également être vrai. Dans une situation de réalisation d'un programme il peut arriver que la résolution d'une énigme soit ressentie comme un petit chef d'œuvre alors que l'effet produit n'a rien de spectaculaire, voire même peut être totalement invisible. La bidouille est alors manifeste sous forme de tâtonnements ou d'efforts de sophistication éventuellement incongrus sur tel ou tel point précis dans un programme.

Plus largement la bidouille apparaît comme une peau de tambour. C'est-à-dire le lieu des mises en résonance des différentes aptitudes technique, artistique, scientifique, de créativité etc. qui sont engagées dans notre démarche de réalisation de programmes. C'est assez compréhensible dans la mesure où il n'y a pas réellement de formes canoniques d'articulation des disciplines différentes, arts-sciences-jeux, auxquelles se référer systématiquement pour construire une idée. Nous ne sommes pas dans le cadre clairement formalisé d'un appareil de production dans une entreprise par exemple. Nous n'avons pas non plus une discipline scientifique particulière à respecter scrupuleusement _du moment que nous savons respecter une certaine forme de bon sens teinté de la rigueur nécessaire pour l'élaboration et la conception.

[17] A propos du bricolage en sciences, voici le propos d'un physicien, Luc Valentin dans le prologue d'un ouvrage intitulé "L'univers mécanique" aux édition Hermann (Paris 1983) : « La mécanique est un vieux thème inépuisable comme l'amour et la mort. Voilà donc un livre de plus à son sujet. Il s'agit en fait d'une sorte d'introduction à la physique _ des atomes aux étoiles _ et à ses méthodes _ du bricolage aux grands principes. »

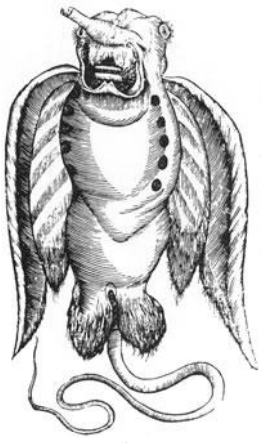
Avec la bidouille il y a quelque chose tout à la fois de flou, de tactile, d'erratique et de fécond. Ce lieu des mises en résonance est aussi un lieu de l'intuition, un lieu de trouvailles et de découvertes. Par là c'est un lieu fort respectable de construction, de conception, et à des niveaux techniques qui peuvent être élevés.

La bidouille peut avoir lieu dans un temps de paresse et de désordre, ou dans un moment d'enthousiasme et d'originalité. Elle dénote des aspects expérimentaux nés d'une organisation informelle et d'une pratique requérant de la souplesse dans les entre deux sciences et arts et jeux et scénarios divers.

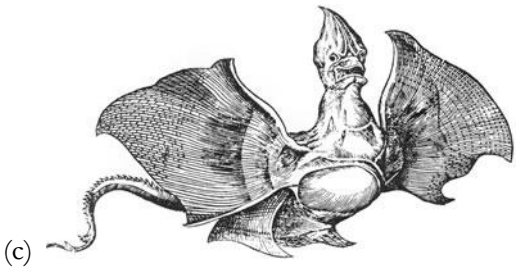
Chimère

En un mot la question « art, science, jeu ou bidouille ? » pose le problème d'identifier un corps chimérique. Selon l'origine mythologique, la chimère est en effet un animal fabuleux qui tient de la chèvre et du lion. Parfois elle possède un arrière train de serpent et une tête de lion sur un corps de chèvre. Parfois elle a plusieurs têtes, une de chèvre, une de lion et elle souffle des flammes. Elle apparaît également comme une « Jenny Haniver », héritière de monstres marins, du basilic et du dragon (figure 5.1).

(a)



(b)



(c)

Fig. 5.1 : Trois représentations de chimères
(a) Jenny haniver ou « dragon volant », Conrad Gessner, 1558.
(b) L'aigle de mer monstrueux d'Ulisse Aldrovandi, , 1613.
(c) Autre chimère d'Ulisse Aldrovandi, 1613.

L'art du programme tel que nous l'approchons est difficile à définir. Il s'agit d'une activité fondamentalement hybride. Il n'y a pas de mélange parfait avec des doses clairement établies de science, d'art etc.. Il n'est pas sûr non plus que la démarche puisse être rapportée à une adition de constituants typiques.

Essentiellement la démarche s'appuie sur une mobilité et une liberté, une curiosité et une expression. *Notre objectif est de créer des points d'équilibre entre des forces de nature et de culture différentes. Non de choisir entre l'une ou l'autre des alternatives. Ce qui est en jeu est la création de sens.* Au moins d'un sens : celui d'une démarche de programmation. Donner une direction à la programmation sans nécessairement revendiquer une appartenance. Art, science, jeu, bidouille… dans le fond peu importe, pourvu que la proposition trouve une motivation et soit reprise, développée, communiquée.

C'est pourquoi nous proposons d'aborder cette question « art, science, jeu ou bidouille ? » de façon transversale et non frontale. La sélection des quatre points suivants nous a paru répondre à cette stratégie d'indépendance : langage, expression, sujet et orientations. Un sujet (par opposition à objet) s'exprime dans un langage et poursuit des orientations. C'est une esquisse d'un portrait d'acteur dont voici quelques traits philosophiques.

2. Sujet

Marie Balmary, psychanaliste, distingue une voie du langage et de l'expérience de la parole constructrice du sujet spécifiquement humain qui dit « je ». Cette voie est à la limite entre les deux credo, religieux et scientifique, antagonistes dans notre culture. Pour le premier « l'homme a été créé par le dieu qui a fait le ciel et la terre ». Et pour le second, tout aussi répandu « l'homme est le produit de la matière et du hasard : il ne doit sa vie à personne et ne la rend à personne lorsqu'elle s'achève » [BALMARY, 1993, p11].

2.1 Expérience du mythe

Pour Marie Balmary ni la proposition religieuse, ni la proposition scientifique ne font état de l'expérience de la parole en tant qu'elle donne accès au *je* spécifiquement humain. Le langage humain relève en effet d'une instance psychique qui n'est pas dans le monde animal même si l'on trouve en celui-ci des échanges fondés sur des messages qui s'apparentent à des sortes de langages.

Cette instance psychique se manifeste en particulier au travers de la *réalité* des mythes. Le récit et le mythe, font état d'êtres invisibles dans le monde et dans la matière, mais bien réels en tant que faits psychiques : « Aucun examen de fossile ni de gène, aucun électroencéphalogramme ne nous permettra d'observer cela qui, pourtant, a lieu dans nos corps biologiques ». La compréhension des processus de la pensée est à rechercher dans l'étude du fonctionnement des mythes et dont l'observation de la matière cérébrale, si fine soit-elle, ne saurait rendre compte. « L'examen de la matière cérébrale ne peut nous expliquer la pensée. Et c'est le mythe qui nous en donne à entendre le processus pour la culture qui est la nôtre » [BALMARY, 1993, p60].

Face à la pression de la science et du savoir objectif, et à une quasi violence des développements récents en matière de biotechnologie, Monette Vacquin, dans le

prolongement de travaux de Marie Balmary, réactualise l'importance fondamentale du mythe dans l'élaboration du sujet et du désir humain :

> « L'origine psychique de l'humain, en effet, ne peut être racontée que mythiquement. La science ne suffit pas à comprendre comment l'humain devient homme, ni à régler les relations qui le génèrent ou le régénèrent comme sujet. L'inscription comme humain ne se fait pas au laboratoire : l'accès à l'humanité n'est pas héréditaire, le sujet est à créer, seul l'objet est créable. Ce que racontent les récits de la genèse, c'est la participation de l'humain à son propre éveil » [VACQUIN, 1999, p 233].

Reporter la question du mythe et du récit sur la conception informatique afin de faire ressortir le *sujet* nous a semblé intéressant dans la perspective chimérique que nous avons adoptée. Pour ce faire nous avons consulté un être monstrueux : le sphinx-machine.

2.2 Sphinx-machine ou simulation de l'esprit

Récemment le philosophe se demande « jusqu'où peut aller la simulation de notre esprit ? » [CHAZAL, 1995, p 104]. Alors le sphinx-machine interroge les passants. Quelle est la machine qui le matin pense sur quatre pattes, le midi sur deux pattes et le soir sur trois pattes ? Curieuses interrogations nées de la machine et ceux qui ne répondent pas sont dévorés.

Sphinx-machine

Nous posons que le programme est bien réel en tant que fait psychique, à l'égal du récit et du mythe, du fait de l'existence d'un sujet qui dit " je " à travers une expérience de langage et un devenir. Si le récit et le mythe, font état d'êtres invisibles dans le monde et dans la matière, le programme quant à lui se donne à voir dans le monde et dans la matière à travers une machine. Il a donc une double réalité : sur le registre mythologique et sur le registre de sa concrétisation matérielle dans le monde. C'est là un fait troublant.

Certaine tendance dure dans le domaine de la vie artificielle (Langton) considère que la machine créée est de fait vivante dans la mesure où l'idée de la vie qui est représentée fonctionne et se manifeste effectivement dans le réel, de façon objective.

En quelque sorte la réalité du mythe en tant que fait psychique se déplace vers la matérialité de l'objet programme et de la machine. Celle-ci en retour est sensée faire la preuve de la véracité du fait psychique parce qu*'elle est ce fait psychique*. En 1950 dans l'article « Computing machinery and intelligence » Alan Turing pose cette question « Les machines peuvent-elles penser ? » [GIRARD, 1995, p 135]. Il conclut : « Je crois qu'à la fin de ce siècle, l'usage des mots et l'opinion publique auront tellement évolué, qu'on pourra parler de machines qui pensent sans crainte d'être contredit ».

Miroir-machine

Ainsi est il une question qui reste perpétuellement en suspens : le sens et le but des programmes, le caractère explicatif, pour la pensée, de leur fonctionnement. Autre être monstrueux, c'est le miroir-machine. Ainsi que le souligne Gérard Chazal, l'étrange jeu de mesure et de perspective entre homme et machine :

> « Utiliser l'informatique comme un miroir dans lequel nous pouvons nous connaître, même si

> maintenant cela se fait en mesurant l'écart, en délimitant des frontières, en prenant conscience des faiblesses et des manques. Peut-être pourra-t-on plus tard décider de la pérennité de ces manques ou de leur caractère provisoire » [CHAZAL, 1995, p 104].

Il y a certes une phénoménologie de l'ordinateur et des programmes. Mais c'est par des questions et non par des solutions, des résolutions, des simulations, que l'ordinateur et le programme s'imposent premièrement. Qu'est-ce que le reflet informatique de notre esprit nous apprend sur nous-mêmes ? Quelles limites séparent l'homme de ses élaborations machiniques ? On songe aux travaux de Gilles Deleuze et Félix Guatari sur l' inconscient machinique[18], soit toute la mécanique inconsciente transposée dans la machine et à l'œuvre dans sa conception. Là interviennent les dimensions mythologique et de récit dans leur réalité première, du point de vue du fait psychique, et non uniquement parce qu'ils sont objectivés par des faits matériels.

Relation humaine

Reprenons alors cette dimension du mythe et du récit transposée à l'ordinateur et ses programmes. Le problème que notre sphinx-machine réactive, c'est d'abord celui d'une relation à l'autre avant d'être celui d'une relation à la machine :

> « Ce que racontent les récits de la genèse, c'est la participation de l'humain à son propre éveil. C'est à la relation humaine qu'est attribué le pouvoir de faire l'homme et la femme, de se mettre debout psychiquement, de naître, de se lever, de s'éveiller. Et le tropisme de l'humain vers l'éveil ne vient ni d'une création ni d'une quelconque pulsion, mais de la rencontre avec un autre humain » [VACQUIN, 1999, p 233].

Le but des programmes est de s'inscrire dans des relations humaines et non de les résoudre. Ainsi réintroduire le sujet revient à réintroduire le concepteur dans la conception, et également l'autre, le public, les autres auxquels la conception s'adresse. Il n'y a donc plus simulation de l'esprit mais exercice d'une pensée réelle qui s'exprime dans un programme, du fait de la machine. Il est question de relations et de communication. De ce fait des problématiques de langage sont inévitables. Dans *L'ordinateur et le cerveau* en 1957, John von Neumann énonce « Nous sommes nécessairement conduits à nous poser des questions sur le langage » [NEUMANN, 1957, 1996, p 79].

3. Langage

Mais ce qui intéresse von Neumann est l'enracinement du langage dans l'activité cérébrale telle qu'elle puisse être décrite à l'aide d'une machine de Turing, avec des algorithmes. Cette hypothèse conduit initialement à l'idée de la généralisation des langages dans un langage source universel. Il serait intrinsèque au fonctionnement cérébral : « La logique et

[18] Gilles Deleuze et Félix Guattari présentent ainsi leurs interrogations : " Qu'est-ce que l'inconscient ? Ce n'est pas un théâtre, mais une usine, un lieu et un agent de production. *Machines désirantes :* l'inconscient n'est ni figuratif ni structural, mais machinique _ Qu'est-ce que le délire ? C'est l'investissement inconscient d'un champ social et historique. On délire les races, les continents, la culture. La *schizo-analyse* est à la fois l'analyse des machines désirantes et des investissements sociaux qu'elles opèrent. " [DELEUZE, 1972, p 7 à 59].

les mathématiques du système nerveux central, considérées comme des langages, doivent être structurellement, fondamentalement différentes des langages de notre expérience courante » [NEUMANN, 1957, 1996, p 80].

3.1 Problématique de diversification

Pour von Neumann la conjugaison ordinateur-cerveau va de soi. Le fonctionnement du cerveau est articulé autour du calcul et l'aspect universel d'un *langage source* est dans cette perspective garanti par le cerveau que tout être humain possède.

Mais se trouve occultée, réduite pour ainsi dire à néant, la diversité des langages, de leurs formes et des réalités qu'ils sous-tendent, ainsi que la nature opératoire de ces formes qui interviennent par rétrocession dans le rapport au monde, en termes de culture, et dans l'élaboration même de l'appareil perceptif. A savoir une compétence en soi de diversification dont il semble contradictoire qu'elle puisse exister en s'inscrivant elle-même comme un langage unique.

Le caractère hautement abstrait d'un langage source unique le rend étrange. Il semble davantage rendre compte de l'idéologie d'un paradigme scientifique réductionniste sous-jacent. C'est donc plutôt du côté de la diversification des langages que nous tournons notre attention. Les possibilités formelles de l'informatique paraissent s'y inscrire comme le ferait n'importe quel langage particulier. *Comment l'informatique intervient dans les processus cognitifs et non en quoi elle peut les simuler est plutôt la problématique dans laquelle nous inscrivons notre travail.*

Au titre de cette diversification née d'une compétence linguistique d'une part, et d'une intervention dans les processus cognitifs d'autre part, il y a possibilité de rapprocher les écritures informatiques avec celles du langage naturel. Or l'écriture née du langage naturel est opératoire et transforme les mentalités. Jack Goody oppose culture orale et culture disposant de l'écriture. « La raison graphique » les distingue, c'est-à-dire l'espace de pensée, de conception et finalement de perception propre aux technologies de l'écrit.

3.2 Spatialisation de la raison

Jack Goody montre que la technologie du support écrit intervient directement dans les processus cognitifs, les modalités de pensée et l'édification d'une vision du monde. Il extrait deux figures propres à l'écriture, la liste et le tableau, plus une troisième qui dans une certaine mesure résulte des deux précédentes : la formule.

Liste, tableau, formule

Dans l'espace planaire de l'écriture la liste est un outil qui organise la linéarité d'une suite en classes d'éléments. Elle permet une numérotation des parties et des éléments dans les parties. Il est possible de la parcourir de différentes manières, de compter les éléments...etc. La linéarité de la liste est affranchie de la condition temporelle de la parole.

Le tableau est en quelque sorte le prolongement naturel de la liste (tableau à une colonne) mais il offre le moyen supplémentaire « de mettre en ordre la connaissance que nous avons

des schèmes classificatoires, des systèmes symboliques et des formes de pensée » [GOODY, 1979, p.110].

La formule est un « schème organisateur », un modèle. Elle est une ligne de pensée réduite dans un formalisme graphique et par un système d'écriture. Son expression triomphe par exemple dans le langage mathématique qui est indépendant des systèmes phonétiques. La formule repose sur l'invention d'un système de notation. Elle est typiquement représentée par des procédures hautement abstraites, décontextualisées et arbitraires du type mathématique [GOODY, 1979, p.212, p.213].

Grammaire

Par ailleurs Sylvain Auroux montre que le phénomène de la grammaire s'est constitué à partir de la technologie intellectuelle développée par les concepts de cette même raison graphique [AUROUX, 1994]. Ainsi avons-nous deux figures simples comme une liste et un tableau et un concept de formule qui peuvent avoir d'immenses conséquences dans la pensée et l'évolution humaine !

Modèles de données

Liste, tableau et formule sont constitutifs de la base de l'informatique. Pouvons-nous considérer que les modèles de données et concepts de l'informatique sont des figures qui s'ajoutent à celles dégagées par Jack Goody ? Pour Sylvain Auroux scripturisation, grammatisation et automatisation peuvent être conçus comme différents seuils que franchit un même processus de formalisation et d'externalisation du langage humain. [AUROUX, 1994, p.165]. Citons également Terry Winograd pour lequel l'informatique se trouve plus proche, sur le plan scientifique, de la linguistique et de la sociologie que des mathématiques ou de la physique [WINOGRAD, 1986].

L'étude d'une « raison informatique » dans le prolongement d'une raison graphique est donc l'étude de la technologie intellectuelle qui découle de la technologie de l'information et du support informatique qui lui sont associés. Non seulement du point de vue technique, de la formalisation logique et du raisonnement, mais du point de vue anthropologique dans les conséquences sur l'acte de penser et de construire ou de modifier une vision du monde. Une étude des fondamentaux morphologiques propres à l'informatique, c'est-à-dire des modèles de données (arbres, listes, graphes, automates etc.), est à envisager dans cette perspective.

Mais également il y a l'étude de la mise en relation de ces figures avec la pensée en général, en tant qu'éléments d'une technologie d'expression, aussi bien littéraire que scientifique.

4. Expression

4.1 Essai de liaison littérature-IA

Langue dans la langue, Gille Deleuze

Pour Gilles Deleuze, la littérature s'impose dans la langue comme une langue étrangère, comme une trajectoire, un chemin qui se trace dans la langue, la remue et concourt à son devenir. La littérature trace dans la langue, précisément, « une sorte de langue étrangère, qui n'est pas une autre langue, ni un patois retrouvé, mais un devenir autre de la langue, une minoration de cette langue majeure, un délire qui l'emporte, une ligne de sorcière qui s'échappe du système dominant » [DELEUZE, 1993, p 15].

Ce devenir de la langue n'est pas l'élargissement du nombre de ses mots, il est véritablement création syntaxique, style : « Il n'y a pas de néologismes qui vaillent en dehors des effets de syntaxe dans lesquels ils se développent » [DELEUZE, 1993, p 16]. La langue source, maternelle, fait ainsi l'objet d'une décomposition et d'une destruction afin que soit possible l'avènement, « l'invention d'une autre langue dans la langue, par création de syntaxe ».

Idées

L'écrivain fait ainsi sa propre langue, « on dirait que la langue est prise d'un délire, qui la fait sortir de ses propres sillons » [DELEUZE, 1993, p 16]. Le retour de cette langue étrangère sur la langue originelle est un bouleversement du langage tout entier qui n'est plus d'aucune langue. L'écrivain le voit et l'écoute du dehors, aux limites. Il découvre des failles, des interstices dans les écarts de langage et ce qu'il voit ainsi sont des idées que l'on ne pourrait faire surgir autrement :

> « Une langue étrangère n'est pas creusée dans la langue même sans que tout le langage à son tour ne bascule, ne soit porté à une limite, à un au-dehors ou un envers constituant en Visions et Auditions qui ne sont plus d'aucune langue. Ces visions ne sont pas des fantasmes, mais de véritables Idées que l'écrivain voit et entend dans les interstices du langage, dans les écarts de langage » [DELEUZE, 1993, p 16].

Le processus est mobile, mouvement, devenir. Il n'y a pas d'interruption du processus par des Idées mais les Idées sont « des haltes qui en font partie [du processus], comme une éternité qui ne peut être révélée que dans le devenir, un paysage qui n'apparaît que dans le mouvement. Elles ne sont pas en dehors du langage, elles en sont le dehors ». Ainsi l'écrivain voit et entend et le but de la littérature « c'est le passage de la vie dans le langage qui constitue les Idées » [DELEUZE, 1993, p 16].

Littérature, intelligence artificielle

Ainsi que la littérature l'informatique fait bouger la langue. Rapprochons par exemple ces propos de Gilles Deleuze avec ceux de Jean-Gabriel Ganascia dans son livre *L'âme-machine* sur les enjeux de l'intelligence artificielle :

> « Nous verrons apparaître des zones de faille, des lignes de fracture au sein de la pensée techniciste, d'où surgissent des réflexions d'un autre ordre, philosophèmes bruts hors du cadre

institutionnel de la philosophie et pourtant proches, à bien des égards, de problématiques de la philosophie classique » [GANASCIA, 1990, p 12].

La comparaison peut même être poussée plus loin encore. Deleuze dans son texte insiste sur le devenir propre de l'écrivain dans sa quête perpétuelle, ouverte : « Il arrive qu'on félicite un écrivain, mais lui sait bien qu'il est loin d'avoir atteint la limite qu'il se propose et qui ne cesse de se dérober, loin d'avoir achevé son devenir. *Ecrire, c'est aussi devenir autre chose qu'écrivain* [nous soulignons] » [DELEUZE, 1993, p 17]. Et voici en écho les termes de Ganascia : « dans ce relief accidenté, *l'intelligence artificielle fait écho à autre chose qu'elle-même* [nous soulignons]. Elle suscite des questions qui lui sont étrangères pour les transplanter et les réactualiser, quitte à les déformer » [GANASCIA, 1990, p 12].

Réinterpréter von Neumann

On le voit, on le pressent, il y a une parenté entre les descriptions. Il y a quelque chose du « dehors » en commun. Essayons de réinterpréter la question du langage minimal de von Neumann à la lumière du texte de Gilles Deleuze comme s'il s'agissait d'un acte de littérature. Von Neumann voit et entend dans les interstices des langues l'Idée sur laquelle il fonde l'informatique. Transposée à la littérature, il ne s'agit plus d'une « réduction » des langages dans un langage source originel selon l'exercice d'un paradigme scientifique, mais de l'Idée, « Vision et Audition qui ne sont plus d'aucune langue », le vu et l'entendu dans les interstices des langues, dans les écarts des langues entre elles. Idée qui fonde l'informatique comme une sorte de sculpture architecturale dérivée du langage et qui participe au devenir du langage, lui ouvre de nouvelles perspectives.

Passage de la vie dans le langage

Certes le moment où le lieu de la concrétisation, de l'objectivation matérielle en machine universelle de programmes peut paraître un clivage, une fracture nette qui délimite deux espaces différents. La description de la littérature de Gilles Deleuze n'est pas celle d'une machine de Turing : il y a le devenir. Mais nous pouvons voir dans l'informatique l'effet d'un « passage de la vie dans le langage ». Apparition d'une nouvelle espèce qui concrétise un passage de vie dans le langage, et constitue un nouveau territoire pour des Idées. Il se crée un bras de rivière qui finalement se détache afin d'irriguer d'autres espaces c'est-à-dire d'engendrer d'autres formes. On ne saurait refuser à l'informatique une fondation sur l'Idée elle-même génératrice de possibilité d'Idées selon de nouvelles contraintes, de nouvelles formes.

Dans cette perspective ce qui probablement ne change pas et permet de relier informatique et littérature, est probablement le lieu étrange de l'extériorité, où tout langage apparaît par son dehors. Il y a là un énorme travail à faire qui permette de relier l'expérience scientifique et l'expérience littéraire de la pensée, de l'Idée. L'informatique telle que nous la connaissons aujourd'hui en est une expression. Mais il est intéressant, au-delà des limites du formalisme logique, de rêver à la possibilité de fonder d'autres branches de langage. Créer un langage sur une base morphogénètique qui intègrerait la dimension temporelle dans des rapports de formes, plutôt que sur un formalisme de type logique, par exemple, est une proposition intéressante [RIDEAU, 1999].

Somme toute la logique n'est ni plus ni moins qu'une sorte de « code de la route » sur la base duquel s'appuie une architecture de composants d'ordinateur, mais peut-on imaginer

d'autres modèles de données d'un ordinateur dont l'assise première ne soit pas un formalisme logique ? Des ordinateurs singuliers dont le fonctionnement serait déroutant. Des cas particuliers d'ordinateurs qu'il faudrait apprendre à connaître et parmi lesquels l'ordinateur que nous connaissons serait finalement lui-même un cas particulier d'ordinateur… Les mathématiques contemporaines rejoignent des problématiques de créativité.

4.2 Arguments mathématiques

Soit une énigme donnée, il existe pour le mathématicien un code à inventer susceptible de l'éclaircir. Pour Nicolas Bouleau, les mathématiques pures reviennent à l'invention de sens et la recherche mathématique est un « déchiffrement inventif ». Les mathématiques progressent non pas par la déclinaison d'une syntaxe mais grâce à des idées qui vont modifier le regard sur une situation et permettre de la simplifier [BOULEAU, 1999, p. 205]. La recherche mathématique permet de construire une interprétation.

Décompilation sémantique

Nicolas Bouleau relie les mathématiques et l'informatique comme suit. Un langage de programmation comme le C ou le Pascal se rapproche de l'expression mathématique d'un problème. Après compilation on obtient le code-machine binaire, qui est incompréhensible. La compilation s'apparente à un codage, un cryptage. Ce qui intéresse Nicolas Bouleau est l'opération de décompilation. Cette opération caractériserait l'activité mathématique :

> « Nous pouvons exprimer l'activité mathématique en termes informatiques grâce à la correspondance de Curry-Howard[19] : l'isomorphisme de Curry-Howard dit essentiellement qu'une preuve mathématique n'est pas autre chose qu'un programme (…) Krivine donne une expression tout à fait pénétrante de l'activité mathématique : *trouver le programme source d'un code* » [BOULEAU, 1999, p 209].

La définition de la tâche du mathématicien à laquelle il aboutit est qu'à partir d'un programme-code, le mathématicien reconstitue un programme-source « autrement dit, en observant ce que fait la machine avec ce code incompréhensible, tenter de reconstituer des procédures et des instructions qui restent en relation de rigueur avec le code et soient intelligibles. Il s'agit bien d'un décryptage mais d'un décryptage interprétatif et créatif » [BOULEAU, 1999, p 210].

[19] Cette conception va dans le sens de récents travaux des logiciens et nous rapproche de l'informatique. Dans une tentative d'interprétation de la logique « qui permettrait de manipuler, non pas les énoncés comme on le fait classiquement, mais les preuves elles-mêmes », les logiciens établissent une correspondance, un isomorphisme entre les preuves et les termes, soit entre des aspects déductifs et des aspects sémantiques : « les logiciens sont parvenus à établir une correspondance, appelée *isomorphisme de Curry-Howard* entre les preuves (calcul des séquents) d'une part et les termes (λ-calcul) d'autre part, donc entre des aspects plutôt *déductif* et des aspects plutôt *sémantiques »* [BOULEAU, 1999, p 208]. Sur la base de ces travaux, Jean-Louis Krivine, mathématicien, propose une thèse selon laquelle utiliser la correspondance de Curry-Howard permet d'établir un dictionnaire entre la théorie de la démonstration et la programmation « il tente de voir à quoi correspondrait, du côté de l'informatique, le travail créatif du mathématicien » [BOULEAU,1999, p 208].

Invention de sens

Cette image du mathématicien décompilateur est pour lui fondamentale dans sa dimension sémantique. Le travail du mathématicien est l'invention du sens : « l'activité du mathématicien est fort différente de celle des machines. *Elle est productrice de sens et productrice de simplicité* » [BOULEAU, 1999, p 211]. C'est-à-dire que l'activité mathématique est indissociable de celle de la signification et celle-ci n'est pas intrinsèque aux mathématiques auxquelles ne reviennent que la syntaxe des symboles.

La thèse de Nicolas Bouleau, à travers l'image du mathématicien décompilateur, est donc principalement que les mathématiques servent à la création sémantique :

> « Il s'agit d'une invention conceptuelle, d'un récit simplificateur, qui anime les signes et leurs relations, là où n'existait qu'une complexité enchevêtrée incontournable tant qu'on voyait les choses avec les sémantiques classiques, légitimes, imposées. Dès lors les mathématiques sont servantes des autres disciplines comme elles le sont d'elles-mêmes, par la fonction simplificatrice attachée aux lectures nouvelles qu'elles recherchent et auxquelles elles portent intérêt » [BOULEAU, 1999, p 211].

Modèles, Récits symboliques

Cette vision des mathématiques a des conséquences très importantes du point de vue de l'ingénieur et de la conception. L'ingénieur est devenu « constructeur de représentations pour agir et décider ». Selon les termes de Nicolas Bouleau ces représentations sont des modèles, et les modèles sont des récits symboliques, des simulacres utiles, des représentations partisanes :

> « Il me semble crucial de mettre en avant que les modèles sont *des représentations partisanes*, au bon sens du terme, c'est-à-dire qu'au sein de multiples possibilités d'expression et de représentation, ils sont le choix d'un parti. Ce parti-pris est l'engagement dans l'action, responsabilité de l'ingénieur » [BOULEAU, 1999, p 301].

Enjeux sociaux

Les enjeux aussi bien sociaux que politiques et économiques sont à considérer. Les mathématiques sont un langage et à partir de lui il est possible à l'ingénieur de dire et de réaliser en fonction de ce qu'il pense ou des contraintes auxquelles il est soumis. C'est-à-dire que les intentions mises en oeuvre comptent tout autant que la réalisation elle-même. Mais elles ne garantissent en rien la validité du projet, même s'il peut être concrétisé :

> « La modélisation intervient dans presque toutes les activités économiques, et du point de vue du modélisateur, le choix des langages, des signes, des méthodes utilise la polysémie mathématique et relève d'un talent expressif mêlant représentation et engagement social, où les mathématiques ne sont plus une symbolisation décrivant un réel abstrait mais un langage entre acteurs sociaux exactement au même titre que le langage ordinaire, aussi tendancieux, aussi trompeur parfois, mais aussi utile, indispensable concrètement » [BOULEAU, 1999, p 268].

5. Orientation

La question de l'erreur et de la remise en cause s'inscrivent dans le prolongement de notre étude d'une raison informatique en tant que technologie d'expression littéraire et scientifique.

Questions de l'erreur

Dans le cadre de la programmation l'erreur peut déterminer l'architecture d'un programme. Une méthodologie de l'écriture qui a pour présupposé que le programme ne va pas fonctionner et qui donne la possibilité à toute opération de faire remonter l'erreur potentielle, probable. Le fonctionnement est alors lié au dépistage des erreurs. Le rôle de l'analyse pour la mise en place de l'algorithme va devoir s'inquiéter de toutes les erreurs possibles. L'erreur met à nu la fragilité du programme et de la conception. Et c'est peut-être une raison de la joie éprouvée lorsque ça marche. Si ça marche, soulagement, c'est bien : un bon programme est un programme qui marche.

Mort par répétition

Le philosophe Michel Serres fait valoir cependant un autre sentiment : « la vie est ce rattrapage manqué, repris, en porte-à-faux, de la continuelle destruction, rattrapage qui relance l'écart, le porte-à-faux, qui s'écroule enfin pour être relevé par une descendance. Ca marche parce que ça ne marche pas » [SERRES, 1980, p 80]. Il ne suffirait donc pas que le programme fonctionne pour que ça marche, mais alors de quoi parle-t-on ? Qu'est-ce qui est sensé fonctionner et qui soit intégré au programme en dépit de son fonctionnement et susceptible d'erreurs ?

Ces questions nous poussent aux limites du programme et de sa technique. La frontière où le programme lui-même pourrait bien être une erreur. Quel est le risque ? On peut suivre la piste du philosophe Michel Serres :

> « le dieu de Laplace n'est savant que dans l'initial, que dans la vision et la prévision. Quand le monde se développe et se déduit des conditions, il les répète en quelque manière. Ce monde est bête, bête sans imprévu, et bête comme redondance, ce monde est mort par répétition. Bête comme une planète, bête comme une méthode et bête comme un mort. Oui le dieu de Laplace est bête comme un automate, bête, oui comme un ordinateur » [SERRES, 1980, p 55].

Passage

Proposons l'erreur comme le défi véritable pour une intelligence artificielle qui s'opposerait à une bêtise réelle. Non pas une erreur technique, ni même une erreur de conception mais le lieu de la fragilité et de la mise en danger, le lieu où scientifiques et littéraires tombent d'accord et se rencontrent enfin, lieu de passage et de recherche : « je cherche le passage entre la science exacte et les sciences humaines ou, à la langue près, ou au contrôle près, entre nous et le monde (…) des sciences humaines aux sciences exactes, ou inversement, le chemin ne traverse pas un espace homogène et vide » [SERRES, 1980, p 15, 18].

Le passage, à la langue près, ou, au contrôle près, entre nous, et vers… on ne sait trop quoi, voilà probablement le lieu qui intéresse l'erreur. C'est-à-dire tout ce que l'informatique est susceptible de s'incorporer et qui demeure insoluble ; ce qui finalement pose question et ne résout rien. Il n'est pas recevable d'automatiser l'insensibilité à la problématique de l'erreur [MORIN, 1990, p 140]. L'activité hérétique, le passage et l'invention qui en découlent peuvent bien caractériser l'existence humaine.

Pouvoir de nier

Gérard Chazal pointe le pouvoir de nier comme un élément important qui différencie la pensée humaine de sa simulation formelle sur ordinateur :

> « Si la négation n'était toujours que la constatation d'une absence, absence d'un objet ou absence d'une propriété, on serait probablement très près de la simuler avec un ordinateur. Mais, et c'est là le plus difficile, nous sommes capables de nier indépendamment de l'existence et de l'expérience d'un objet. Nous voulons parler du doute, du refus, du mensonge comme attitudes négatrices auxquelles, d'une manière générale, la machine n'a pas accès » [CHAZAL, 1995, p 104].

Ainsi le développement informatique qui ne serait pas pris en compte comme une erreur potentielle, pourrait apparaître comme une sorte de refoulement ou de déni de l'erreur : si le programme marche il n'y a pas d'erreur. Le pouvoir de nier s'oppose au déni de l'erreur.

Erreur et vie

L'erreur désigne plus largement tout un lot de préoccupations fondamentales comme la faute, la transgression, la mort, la vie, la vérité etc. Par exemple au plus bas niveau, l'organisme vivant lutte contre la dégradation ou la mort « la vie connaît le risque d'erreur, et c'est pour cela qu'il y a d'innombrables processus et mécanismes, déjà chez les procaryotes, les bactéries, pour réparer l'A.D.N. (qui sans cesse se détériore), c'est-à-dire corriger les risques d'erreur en rétablissant l'information originelle » [MORIN,1990, p 131].

Tracer l'erreur fait apparaître une palette de questions vertigineuses dans des domaines variés, ainsi que la fécondité intrinsèque à l'erreur du point de vue de la création, de la découverte, des stratégies (la mise à profit de l'erreur génétique par le vivant). Mais ce que nous voulons faire ressortir dans notre perspective de créativité informatique, c'est qu'en admettant l'idée même que l'erreur puisse s'exercer sur le programme à ses limites externes, en tant qu'il exprime une idée, c'est reconnaître qu'il exprime une idée.

Erreur et dialogue

Le programme peut ne pas jouer le jeu scientifique de la vérité et de l'erreur. Selon Edgar Morin la vérité *de facto* s'oppose à la vérité référée à « des orthodoxies, des normes, des finalités, des croyances que l'on pense saines, bonnes, justes, nécessaires et vitales pour la société. Il est évident que le problème ne se pose pas au niveau de la vérification. Le problème se pose au niveau des systèmes de valeurs, et le problème se complique, ce type de vérité échappe à la réfutation » [MORIN, 1990, p 135].

Un programme qui marche n'est pas réfutable. Il s'auto-valide. Cependant la vocation d'un programme n'est pas d'être vrai, mais d'exprimer quelque chose qui peut être mis en doute. C'est une proposition. Par son fonctionnement il établit une vérité objective, pour

ainsi dire de nature scientifique ; mais en tant qu'il est, il se prête à une lecture subjective. Et, ainsi que nous l'avons évoqué à propos de la lecture des mythes, pourquoi ne se prêterait-il pas à une « lecture subjectivante qui ne vise pas à l'accroissement d'un savoir mais à l'avènement d'un être parlant en son propre nom dans la rencontre avec d'autres » ? [BALMARY, 1999, p 13].

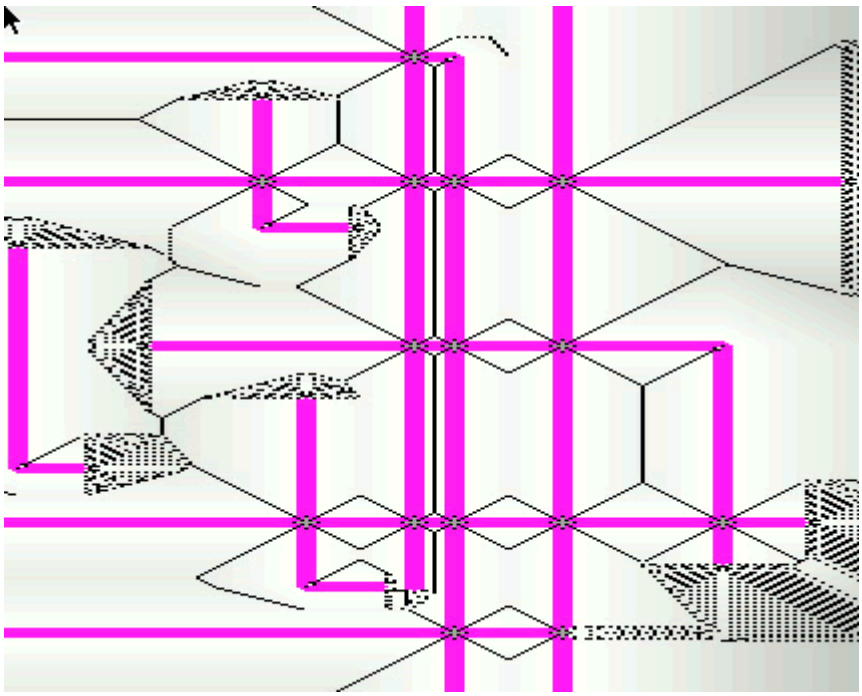
Parfois l'idéologique utilitariste en informatique peut masquer derrière le concept d'utilisateur, la personne qui lit le programme. Dans le monde du travail il peut s'agir pour elle d'être aux ordres du programme et des normes associées. De même peut être masquée la rencontre des concepteurs. Mais cette notion de l'erreur fait le lien avec celle du *sujet* par laquelle nous avons commencé ce chapitre. Ce n'est plus l'utilité du programme qui intéresse, ni même le savoir qu'il peut apporter et sur lequel il repose, mais une rencontre à travers laquelle il se passe quelque chose qui n'a pas besoin d'être clairement élucidé.

P A R T I E 2

Être créatif

Chapitre 6

Phénoménologie des programmes



Dans le chapitre précédent la question « Art, science, jeu ou bidouille ?» nous a permis de faire valoir la cohérence d'un corps chimérique de l'activité de création informatique que nous présentons. Cette cohérence est solidaire de celle d'un sujet acteur et concepteur pour la démarche. Et nous avons envisagé quelques aspects liés au principe d'un sujet concepteur de programmes informatiques, notamment du point de vue du langage, d'une expression et des orientations. La question de l'erreur a permis de mettre en doute la véracité d'un programme, de l'interpeller et de le questionner autrement que comme s'il s'agissait d'un simple outil remplissant plus ou moins correctement sa fonction.

Cette ouverture sur le programme erratique ou hérétique permet d'en développer la curiosité. Il s'agit d'écouter comment « sonne » un programme, c'est-à-dire de sentir les possibilités de l'espace du programme dans une perspective artistique ou de créativité.

Dans le fond le programme peut être rapporté au « plan originel[20] » de Wassily Kandinsky qui est défini « comme un être autonome dans le domaine de son entourage » [KANDINSKY, 1991, p. 143]. Ainsi peut-il en être du programme informatique. C'est ce que nous désignons par la propriété de réfraction du programme.

1 Propriété de réfraction

1.1 Une propriété émergente du programme

La machine informatique est liée à des suites d'opérations discrètes et calculables. Mais la totalité du phénomène de cette machine n'est pas réductible à une telle suite d'opérations. En particulier l'universalité intrinsèque d'une machine de Turing ne rend pas compte de ce qui a trait au sujet et au désir et que le mythe est à même de désigner. En tant qu'objet investi subjectivement, la machine se heurte aux limites du calculable, ou de la mise en opérations calculables.

Une lecture mythologique des programmes élargit la possibilité du sens que peuvent prendre des écritures de programmes ; les bords de l'espace de la conception en sont assouplis. Le programme n'a plus pour vocation de rationaliser quelque chose. Et s'il est mis au service d'une démarche de rationalisation, il reste qu'il ne peut pas gérer ce qui dans le réel échappe au phénomène rationnel.

Intrinsèquement le fonctionnement formel d'un programme ne tolère pas le désordre. Il y a une rationalité nécessaire du point de vue de sa syntaxe. Mais il n'est plus rationnel dans les mêmes termes dès lors qu'il entre en résonance avec la complexité du réel. Ce sont les sujets, utilisateur et concepteur, qui fondent une rationalité externe pour le programme, par un consensus autour d'un objectif. C'est-à-dire par un ensemble de croyances, désirs , espoirs, intérêts etc. placés dans le programme.

[20] Le plan originel de Wassily Kandinsky dans le domaine de la peinture est, « la surface matérielle appelée à porter le contenu de l'œuvre » [KANDINSKY, 1991, p.143]

Ce rapport entre fonctionnement formel et lecture de programme conduit à se représenter le programme informatique comme un phénomène doté de propriétés non programmées que l'on peut considérer distinctement d'un quelconque consensus initial. Il s'agit du rayonnement propre d'un programme indépendamment de ce dont il a pu être investi. C'est ce que nous appelons la *réfraction* du programme. La réfraction du programme est en quelque sorte l'autonomie du programme une fois qu'il est réalisé et rendu public. Cette propriété est fondamentale dans une approche artistique d'écriture de programmes. C'est le lieu d'une autre description du programme.

1.2 Comme une mélodie, un objet de culture

Cette propriété peut être comparée à une mélodie dans le domaine musical. On peut comprendre et discuter comment techniquement une mélodie est faite ainsi que la place qui lui est accordée dans un morceau. En revanche la volonté dont elle témoigne exactement, personne ne peut en rendre compte autrement que par une croyance reliée à l'analyse de résultats qu'elle produirait effectivement sur chaque personne d'un auditoire.
La volonté sous-jacente à l'élaboration même de la mélodie fait elle-même l'objet de croyances et de désir. Désir d'atteindre tel ou tel objectif vis-à-vis d'un public et croyance qu'il faut s'y prendre de telle ou telle manière compte-tenu de ce que l'on sait, imagine et croit à propos de son public.

D'ailleurs ça marche et l'on finit par se mettre d'accord avec son public. Ensuite la rationalisation exagérée d'un tel rapport clôt la possibilité de création qui prend le statut de produit, sujet à des normes, des standards en vue d'une production ou plutôt d'une duplication. Mais cette part, qui ne se prête pas à rationalisation, est probablement ce qui fait, pour le connaisseur, l'attrait de la mélodie : elle entre en résonance avec un public. La mélodie outrepasse largement les seules possibilités de connexions logiques et prédictibles, avec la complexité inouïe qui est à l'œuvre dés le contact avec un public.

Le programme en ce sens est comparable à une mélodie. Cette propriété fait du programme et de la machine non pas un objet qui pense mais tout simplement un objet de pensée livré à la sensibilité humaine. Si l'on se remémore l'étrangeté de la question « Les machines peuvent-elles penser ? », on constate que ce n'est pas la machine qui pense, mais la machine « objet de pensée » qui nous renvoie à nous-mêmes au sein des évolutions de la culture contemporaine.

Objet de culture

Cette capacité de réfraction de l'élaboration informatique traduit à la fois les limites et l'étendue du concept de programme. Les limites parce que des activités humaines ne sauraient se résoudre à des programmes. Elle ne saurait se laisser absorber toute entière pour être régurgitée sous la forme d'un programme. Et l'étendue, parce que la relation de cause à effet entre le programme et l'étendue de la réalité qu'il initie n'est pas totalisée par le programme lui-même. La réalité initiée par le programme ne se réduit pas au programme lui-même.

Un territoire expressif est là. Il fait l'objet de la conception de machines, mais aussi d'une écriture au sens littéraire et artistique. Même si Machine *et* Ecriture sont reliées dans un

tout quasi organique au sens d'un être technique autonome concrétisé tel qu'il est décrit par le philosophe Gilbert Simondon [SIMONDON, 1989].

2 Entrer en résonance avec le réel

La réalité de cet espace de réfraction du programme se retrouve dans la définition du travail que donne Christophe Dejours (dans une perspective de recherche de la médecine du travail).

Le travail selon cet auteur résulte de la nécessité d'adapter des procédures organisationnelles à une réalité qu'elles ne recouvrent pas :

> « Le travail (…) est précisément ce que les travailleurs doivent ajouter aux procédures et à l'organisation du travail prescrite, pour faire face à ce qui n'a pas été prévu et ce qui parfois ne peut pas l'être au niveau de la conception : le travail, c'est l'activité coordonnée des hommes et des femmes pour faire face à ce qui ne pourrait être obtenu par l'organisation stricte des prescriptions » [DEJOURS, 1998, p 74].

2.1 Compléter une organisation

A propos des personnes en situation de travail et du rôle des nouvelles technologies, Christophe Dejours, dans la perspective qui est la sienne, insiste sur le déni du réel au travail et du rôle joué par les nouvelles technologies à ce sujet :

> « La stratégie de distorsion communicationnelle est d'abord fondée sur le déni du réel au travail. Mais ce dernier est indissociable des croyances alimentées par le succès des nouvelles technologies, des sciences cognitives et du développement des travaux sur l'intelligence artificielle » [DEJOURS, 1998, p 75].

Le réel au travail est ce qui vient compléter l'organisation prescrite, or le programme est souvent chargé de récapituler ces prescriptions afin d'en faciliter l'exécution lorsque c'est possible. En quelques clic-souris l'affaire est réglée. Cependant la facilité obtenue multiplie la demande de traitement et d'autre part, même bien conçu un programme n'est jamais parfait. Il nécessite un apprentissage. Et il reste toujours des situations pour lesquelles le programme est mal conçu. Travailler avec un support informatique se traduit souvent par une exploration au fur et à mesure des besoins, des possibilités et des impossibilités, des maladresses du programme etc.

2.2 Création de situations nouvelles

Le programme, qui bien souvent est conçu pour absorber les situations de travail qu'il est sensé contrôler, crée des situations nouvelles qui n'étaient pas prévues par le programme, mais qui sont la conséquence de la présence du programme. Par exemple, dans les domaines administratifs, on trouve des personnes qui n'ont plus la possibilité de prendre des décisions si l'initiative personnelle pour résoudre tel ou tel problème directement par

une personne n'est pas prévue.

Ainsi le programme informatique ne fait pas le travail, il le déplace et en change la nature. Le programme informatique ne fonctionne en terme de travail, dans l'espace de réfraction que parce qu'il est complété par l'activité de l'utilisateur et même plus généralement par l'activité sociale qui l'entoure [WINOGRAD, 1986].

Cet espace de travail qualifié de *réel* par Christophe Dejours, qui par définition n'est pas prévu dans le programme, mais se dégage du fait de la présence du programme, peut être exploité par un concepteur. Si l'on admet d'une part de le reconnaître et d'autre part de lui conférer un statut. L'enjeu est alors de statuer sur la réfraction du programme.

2.3 Statuer sur une situation

Un programme (procédures organisationnelles) n'a pas nécessairement pour vocation de proposer du travail. Ce qui peut être résolu par un programme repose d'abord sur l'interprétation de cet espace et le statut qui lui est accordé. L'espace intermédiaire dit de réfraction du programme peut être réinterprété et étendu à des perspectives ludiques où de plaisir. Des jeux sophistiqués ne sont pas plus simples que des logiciels dédiés à un travail, c'est le statut de jeu ou de travail qui fait la différence.

Selon le statut accordé, tel ou tel type de problématique sont soulevées par le programme et sa réfraction. Si l'on accorde à cet espace un statut artistique, ce sont des problématiques artistiques qui sont posées. Elles font appel à des références du domaine, le statut même de l'œuvre d'art peut être interrogé (faire du public un créateur, invitation à l'interprétation, investissement fantasmatique etc.)

3 Langage sculptural

Pour l'informaticien, le programme se présente un peu comme Janus aux deux visages, en l'occurrence deux langages, machine et naturel.

Certes l'architecture de Von Neumann aboutit à une hiérarchie de langages qui vont du plus proche de la machine, langage binaire, au plus proche de l'humain, langage naturel, en passant par la microprogrammation, l'assembleur et les langages de programmation. Les langages de programmation utilisés pour programmer peuvent eux-mêmes être hiérarchisés (par exemple C, C++, Java).

Mais ils ont en commun d'être en quelque sorte des interfaces de programmation. Le lieu de la dichotomie mais aussi de la rencontre entre extérieur et machine. On pourrait définir le programme informatique par le résultat de cette rencontre qui apparaît comme une intersection extériorisée et objectivée dans le programme et dans les limites de la plasticité de la machine qui agit comme un média.

3.1 Deux langages irréductibles

Le langage de programmation se rapproche, si on veut, du langage naturel. Mais en fait il demeure très proche de la machine, c'est d'ailleurs à sa proximité que se mesure sa puissance. Disons pour simplifier qu'il représente la machine. Nous voulons souligner ici, qu'en dépit de cette interface avec la machine, ce qui est en langage machine n'est pas ce qui est en langage naturel. Les deux langages sont autonomes, non réductibles l'un à l'autre.

Cet aspect s'oppose à l'idée de Von Neumann évoquée au chapitre précédent. Selon cette idée les langages humains seraient intrinsèquement non nécessaires parce que transposables les uns aux autres par le biais d'une souche originelle plus profonde, rivée dans le fonctionnement cérébral [NEUMANN, 1957, 1996, p 79, 80]. Dominique Pignon souligne cet aspect de l'hypothèse de Von Neumann :

> « Si chaque langue possède une sorte de code minimal, au sens de Von Neumann, dont une langue particulière serait une représentation, alors chaque langue peut s'exprimer dans ce code universel (au sens où il n'existe pas de machine de Turing qui ne puisse être décrite à son tour par une machine équivalente). Ce qui entraîne que toutes les langues, suivant cette hypothèse, seraient équivalentes entre elles. Et c'est bien le propre d'une langue d'être traduisible, en un nombre fini de mots, dans une autre langue » [PIGNON, 1996, p 114].

Cette hypothèse élimine en particulier le fait que des langues puissent entrer en relation d'incommensurabilité, comme deux théories, deux paradigmes, font état de mondes différents. Notre propos est que pour un même programme on a deux modèles, un dans le langage naturel et un dans le langage de programmation. Dans l'ingénierie on accède souvent de l'un à l'autre par un glissendo méthodologique ; une succession d 'étapes de l'extérieur vers l'intériorité de la machine.

Mais il ne s'agit pas d'une simple traduction de l'un par l'autre. Les deux langages tendent l'un vers l'autre sans jamais se confondre. Chacun d'entre eux suppose une interprétation, ou plutôt une recréation de l'un par l'autre[21]. Ils trouvent un accord dans le programme, mais celui-ci, en outre, existe en lui-même. Il rend compte de chacun des deux langages mais il n'est pas réductible à chacun des deux langages. Le programme émerge de cette tentative de croisement entre deux langages. La pression exercée par chacun des deux langages converge vers le programme, comme si le programme en était une intersection, un bord créé et qui tend à leur être commun.

3.2 Deux niveaux de pression pour la propriété de réfraction

C'est cette convergence qui crée l'identité du programme. Et si l'on considère que le programme possède une identité propre, indépendante, cette corrélation possède des propriétés spécifiques que l'on retrouve dans la réfraction du programme. La limite vers laquelle tendent chacun des deux pôles linguistiques nécessaires au programme informatique trace l'espace de réfraction du programme. Cet espace, une fois concrétisé par le programme résultant _ fruit de la convergence des deux pôles linguistiques _est pris

[21] La physique contemporaine rencontre aujourd'hui une difficulté sérieuse du fait que son expression mathématique n'est pour ainsi dire plus transposable dans le langage naturel [KLEIN, 2000]

ensuite entre les procédures organisationnelles exercées par le programme et le réel.

Il y a donc deux niveaux de pression pour la propriété de réfraction. Celui des deux pôles linguistiques et celui de l'organisation qui s'exerce avec le réel. Dans cette perspective, le programme émerge comme une forme, et au sens artistique du terme. Travailler sur cette forme rejoint des problématiques par exemple de peintre, et fait intervenir des processus spirituels comme la *résonance intérieure* de Kandinsky.

Apparition de formes

La forme n'est plus ni logique ni rationnelle. elle est à la convergence de pôles linguistiques, comme une abstraction non linguistique et se trouve concrétisée entre le programme et le réel comme une sorte de sculpture abstraite, émanant d'un matériau binaire. L'informatique apparaît comme la production de formes à travers l'articulation et la mise sous tension de langages différents, objectivées dans des programmes et réfractées dans le réel, la communication avec un public.

3.3 Concepteur en entrée sur réseau de préoccupations

La pensée s'investit à chaque niveau dans les termes qui lui sont propres : langage machine, langage naturel, modélisation, objectifs (stratégie), situation publique ; ce ne sont pas les mêmes pensées, pas les mêmes épreuves de pensée.

L'informaticien est celui qui fait et qui pense la jonction entre différents niveaux d'analyse et de préoccupations. Il résout les épreuves et disjonctions à travers un programme qui est l'expression de sa résolution. Ceci plaide pour une informatique en tant que technologie d'expression. En effet, il n'y a pas de langage source à partir duquel s'effectue l'opération de conception prise dans un réseau d'activités et de préoccupations.

Le concepteur qui prend à charge la totalité des opérations se trouve à un méta-niveau qui s'exprime dans l'activité globale. De telle sorte qu'il devient difficile de raconter ce que l'on est en train de faire autrement que par le programme lui-même. Mais la perception que l'on a prend un caractère étrange, indescriptible, intuitif, visionnaire parce que la conception émerge d'un ensemble d'opérations sans se réduire jamais à chacune de ces opérations, et leur somme n'est pas édictable formellement.

4. « Intentionalité »

4.1 Intentionalité selon John Searle

L'Intentionalité avec un " I " majuscule de John Searle n'est pas l'intention au sens courant. L'Intentionalité de John Searle est ce qui se dégage de la vie psychique en général comme « la capacité biologique fondamentale de l'esprit de mettre l'organisme en rapport

avec le monde » [SEARLE, 1985, p 330]. Son modèle décrit la correspondance qui s'établit entre le cerveau et le monde à travers l'action et la perception et par le truchement des états mentaux dits *Intentionnels*.

Ils sont des contenus représentatifs (ou propositionnels) orientés par différents modes psychologiques. Croyances, craintes, espoirs, désirs, amour, haine, aversion, sympathie, antipathie, doute, questionnement, joie, exaltation, dépression, etc. sont des exemples d'états mentaux qui peuvent être Intentionnels. Tous les états mentaux ne sont pas intentionnels. Est intentionnel un état mental *E* (croyance, désir, etc.) tel qu'il permet de répondre à des questions comme *E* à propos de quoi ? *E* de quoi ? *E* que quoi ?
L'Intentionalité se caractérise par une propriété de renvoi à des objets et à des états de chose du monde, et pour John Searle cette propriété est commune aux états mentaux et aux actes de langage.

L'Intentionalité des états mentaux est considérée comme intrinsèque, c'est-à-dire qu'elle relève de la nature même du phénomène biologique cérébral sous-jacent. Mais l'Intentionalité des actes de langage, de l'énonciation est considérée comme dérivée [SEARLE, 1985, p 15, 330]. En fait l'Intentionalité d'une énonciation est dérivée de l'Intentionalité intrinsèque aux états mentaux, ainsi, par rapport au langage, Searle écrit :

> « Une phrase est un objet syntaxique auquel sont imposées des capacités représentatives : croyances, désirs et autres états Intentionnels ne sont pas en tant que tels, des objets syntaxiques (même s'ils peuvent être et sont d'ordinaires exprimés en phrases) ; leurs capacités représentatives ne sont pas imposées mais intrinsèques » [SEARLE, 1985, p10].

Cette différenciation entre des objets syntaxiques d'une part et des états mentaux Intentionnels d'autre part pose la question du sens, c'est-à-dire « comment l'esprit imposent-il de l'Intentionalité à des entités qui ne sont pas intrinsèquement Intentionnelles, soit à des entités telles que les sons et les inscriptions qui, en un sens, ne sont que des phénomènes physiques du monde comme les autres ? » [SEARLE, 1985, p 44, p 203].

Cette question nous intéresse dans notre démarche de création informatique. Non pour y répondre ou nous engager dans une telle recherche mais plutôt pour jouer avec ce phénomène. Le programme rendu perceptible à travers essentiellement des sons et des images apparaît comme un objet syntaxique qui fait sens. Il est une sorte de capteur d'Intentionalité au sens de John Searle. Aussi apparaît-il comme une sorte de sculpture, une sculpture informationnelle.

4.2 « Formes fonctionnantes »

Ainsi que l'indique Herbert Simon, la conception est intimement liée au processus mis en place pour la conception : « la forme de la conception elle-même, et l'organisation du processus de la conception, sont l'une et l'autre deux composants essentiels d'une théorie de la conception » [SIMON, 1991, p.134]. Faire le programme c'est d'abord faire fonctionner le langage de programmation compte tenu des idées initiales, des connaissances acquises, du projet. Par ailleurs, le programme a sa nature propre, il existe en lui-même comme un tableau de Klee ; la vocation de ses tableaux n'est pas de représenter quelque chose, mais que le tableau *fonctionne* et mette en œuvre son autonomie

propre. Le programme, de la même façon, a sa propre identité et il génère une cohérence autonome. Nous sommes sortis du champ de la simulation et avec lui de la représentation. Les données théoriques et empiriques nécessaires à son élaboration ne sont pas « représentées » dans l'écriture du programme. Le rapport à la conception est davantage un rapport d'analogie.

L'œuvre, pour Klee, prend racine dans la perception du phénomène de vie et la cognition. Aux prises avec la complexité, l'artiste comme le profane sont jetés « sans avoir été consultés dans un monde multiforme et, (...) obligés de s'y retrouver tant bien que mal » [KLEE, 1985, p.16]. Pour faire apparaître les moyens spécifiques dont dispose l'artiste pour ne pas se perdre, Paul Klee évoque la parabole d'un arbre :

> « Cette orientation dans les choses de la nature et de la vie, cet ordre avec ses embranchements et ses ramifications, je voudrais les comparer aux racines de l'arbre. De cette région afflue vers l'artiste la sève qui le pénètre et qui pénètre ses yeux. L'artiste se trouve ainsi dans la situation du tronc. Sous l'impulsion de ce courant qui l'assaille, il achemine dans l'oeuvre les données de sa vision. Et comme tout le monde peut voir la ramure d'un arbre s'épanouir simultanément dans toutes les directions, de même en est-il de l'œuvre » [KLEE, 1985, p.16, 17].

Il en est de même pour le programme qui peut, en outre, trouver des racines dans l'exploration et l'inspiration de théories et concepts appartenant à différents domaines.

Le premier rapport recherché est un rapport de « formes fonctionnantes » du point de vue de leur sens. Il n'est pas nécessaire ensuite de reconnaître la filiation entre l'inspiration initiale et le modèle créé. Le modèle ne représente pas les concepts sur lesquels la conception s'est appuyée, s'il représente quelque chose, c'est avant tout la démarche elle-même.

Dans cet esprit nous présentons quelques réalisations, un peu comme une galerie de tableaux, un chantier artistique en cours de développement.

Chapitre 7

Exemples de réalisations

Le point géométrique est un être invisible. Il doit donc être défini comme immatériel. Du point de vue matériel le point égale Zéro. Mais ce Zéro cache différentes propriétés « humaines ». Selon notre conception, ce Zéro _ le point géométrique _ évoque la concision absolue, c'est-à-dire la plus grande retenue, mais qui parle cependant.
Ainsi le point géométrique est, selon notre conception, l'ultime et unique union du silence et de la parole.
C'est pour cela que le point géométrique a trouvé sa forme matérielle en premier lieu dans l'écriture _ il appartient au langage et signifie silence.

Wassily Kandinsky

Un certain nombre de programmes ont été réalisés. Ils ont fait l'objet de diverses expositions :
Dialectic,
Les notes volantes,
Discours coton,
Graphitos,
Graphitor,
La vénus du mythe des races métalliques,
Dicty l'agrègue,
Mikado,
Buisson platonique,
Buisson vanité,
Buisson érotique,
Kandinsky shoot again.

D'autres sont à l'étude comme :
Le routard cosmique,
Système interactif de la vache et du train,
Le Kandoscope,
Algua mécanica,
Le nébulleur carré,
Troupeaux de pirlufs et sociale-mécanique,
Bouddha neuromimétique avec capteurs.

Et aussi en cours d'élaboration une galerie de logiciels monstrueux :
Le Brobock à plumeau,
La mouche de Grattécollé,
Les nénuphs hypnotiques,
Pour une vie de champignon,
Une basse-cour de l'enfer.

Certains ont donné lieu à des approfondissements théoriques pour la formalisation de la démarche comme Dialectic et Vénus que nous avons déjà rencontrés. Dans ce chapitre nous exposons les autres programmes qui ont fait l'objet de présentations publiques dans des situations diverses.

Dans la première partie nous avons vu que les projets sous-jacents aux programmes sont un peu comme des ateliers. Ils ouvrent des domaines de recherches et de développements et ne sont en général pas clos par un programme unique. Ainsi les pistes proposées par ces programmes correspondent à des ouvertures, nous les avons regroupées en quatre orientations principales : les systèmes scénaristiques, les systèmes génératifs, la mise en scène de processus graphiques et la référence à la culture du jeu.

Mais il ne s'agit pas d'un classement au sens formel. Ce regroupement met en lumière des facettes différentes que l'on peut retrouver dans un même programme. Par exemple « Discours coton », en tant que générateur de texte est un système génératif. Du point de vue de sa présentation avec des nuages de lettres colorées il entre dans un processus graphique. Si les textes sont orientés vers un objectif, publicitaire par exemple, il fait l'objet d'un scénario. Il serait possible également d'imaginer un jeu basé sur des mots ou textes.

1. Systèmes scénaristiques

Avec des systèmes scénaristiques nous désignons la possibilité de raconter une histoire via la réalisation d'un programme. « Raconter une histoire » revient à mettre en place un système dont les parties sont en interaction selon des processus imaginés et conçus par un auteur. Il ne s'agit pas du déroulement linéaire d'un conte par exemple. C'est l'attitude mise en oeuvre avec la programmation de nénuphs décrite au chapitre trois et de notes volantes. Les processus réalisés peuvent être affiliés à la vie artificielle soit en esprit soit parce que l'histoire racontée s'inspire d'un modèle biologique ou d'observations de la vie animale et de la nature. C'est le cas pour « Dicty l'agrègue » qui s'appuie sur la description de l'amibe du terreau.

1.1 NOTES VOLANTES

Nous avons déjà rencontré les notes volantes au chapitre un à propos de la conception d'un espace d'expression puis au chapitre deux avec la présentation du programme « La vénus du mythe des races métalliques ». L'idée des notes volantes est de croiser musique et systèmes vivants. Le principe initial est de concevoir un séquenceur musical animé par un système de vie artificielle. L'organisation de cette vie artificielle est apparentée à une dramaturgie.

Comme à l'origine les notes volantes sont des chauves-souris, elles possèdent un radar avec lequel elles entrent en relation et se nourrissent, dans un univers peuplé de moucherons harmoniques. (Figure 7.1)

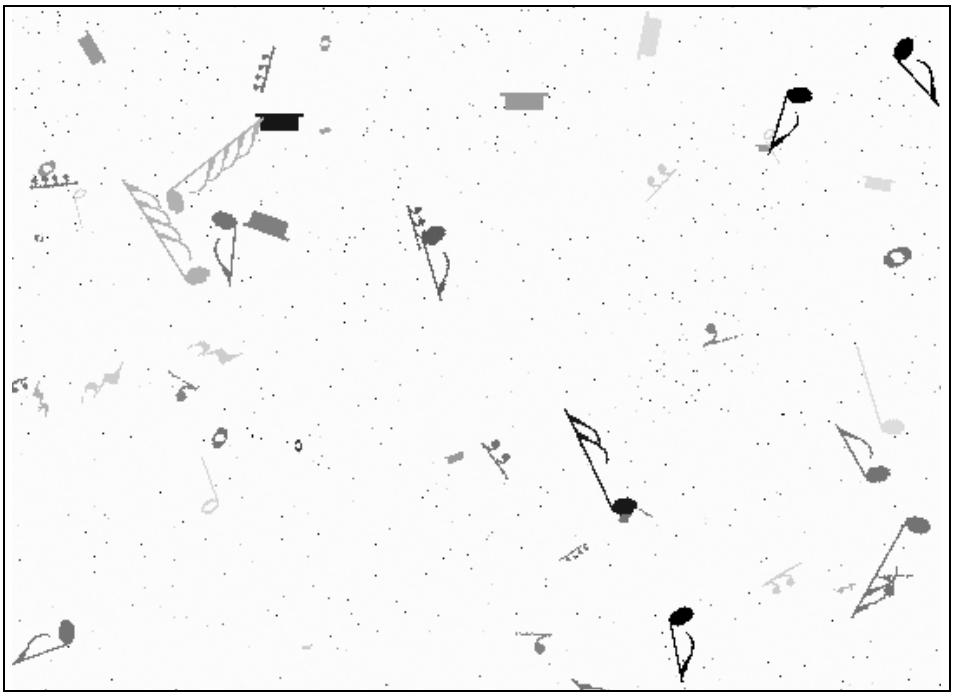


Fig. 7.1 : Notes volantes et moucherons harmoniques

Il y a des mâles représentés en nuances de bleu et des femelles de rouge à orange, ocre. De leurs attractions réciproques se structurent différents types d'états, de comportements et de compétences : désirs, conflits, amours, fuites, réminiscence de situations vécues. Elles sont sujettes à maladies non nécessairement mortelles et une note volante atteinte de maladie devient verte.

Une barre de contrôle des activités permet d'interagir avec les notes volantes (figure 7.2). Les activités sont assez simples. Les notes volantes peuvent manger : « Mang ». « Mouc » désigne le renouvellement des moucherons harmoniques. « Gros » activé, les notes volantes vont se métamorphoser au prorata de la quantité de moucherons harmoniques absorbés et de leurs valeurs nutritivo-harmoniques. S'il y a grossissement le volume sonore augmente et la mobilité diminue. Les mouvements se font plus courts, la valeur rythmique s'accélère. La croche passe à la double croche, la double à la triple. Lorsque le contrôleur de fatigue, « Fati », est activé il pondère cet accroissement en exerçant une force inverse qui peut diminuer le volume sonore de la note et la ramener jusqu'à des valeurs de silence.

Bien entendu les notes peuvent chanter avec le contrôleur « chant ». Le choix de la hauteur de note sonore est fait selon une trame harmonique relativement simple dans un mode précis. Toutes les notes sont soumises à un même mode, une même tonalité. Ce qui permet d'avoir un ensemble effectivement harmonique, non cacophonique. Du point de vue technique le son utilise le synthétiseur midi de la carte son.

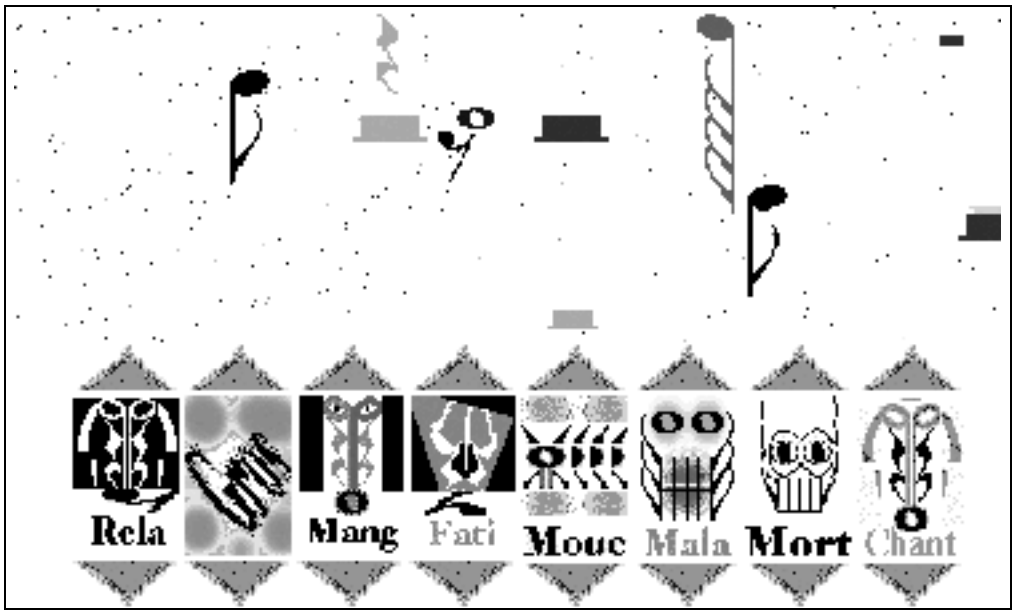


Fig. 7.2 : Contrôles des activités

L'opérateur des maladies, « Mala », apporte une sorte d'usure de la note. Cette usure a des répercussions sur les relations de la note avec les autres notes. Si la mort est active, cette usure peut conduire à la mort. C'est-à-dire la disparition de la note. Mais la santé peut être retrouvée et toute usure être réparée soit avec l'absorption de moucherons harmoniques de qualité, soit par des rencontres sympathiques et des histoires d'amour.

En effet les notes sont sensées adopter des comportements relationnels en fonction de rencontres des unes et des autres. Chaque note développe alors « une vie intérieure »

fondée sur la mémoire de sa propre évolution, de ses activités et de ses « choix » dans les situations où elle se trouve. Chaque entité est « impressionnée » par ses échanges et cette impression en retour, intervient sur les échanges à venir. L'entité conserve l'empreinte de ses rencontres. A terme cette mémoire doit jouer un rôle dans l'orientation des évolutions de l'espèce et des mutations peuvent être envisagées, par exemple l'apparition de prédateurs ou au contraire de sages.

La modélisation des relations s'appuie à l'origine sur des hypothèses psychanalytiques de W.R. Bion le *Combat-fuite* et le *couplage* [1]. Evidemment tout cela ne repose que sur un jeu de variables imbriquées. Elles sont incrémentées ou décrémentées selon la définition de règles et de facteurs en nombre plus ou moins élevé et d'inspiration plus ou moins sérieuse. Dans la première version du logiciel un observatoire était installé pour suivre l'évolution de chacune des notes volantes (figure 7.3).

En fait c'est assez difficile à réaliser parce qu'un équilibre d'ensemble du système des entités est toujours très difficile à obtenir. Obtenir que le système puisse fonctionner sans jamais converger vers une solution d'ensemble uniforme est en soi un défi et approcher un tel résultat est une belle réussite. En général le système va se bloquer sur une solution. Par exemple toutes les entités deviennent grosses, peu mobiles et très sonores, ou au contraire ralentissent et deviennent presque invisibles. Il faut conduire la collectivité par un jeu subtile de déstabilisation ou de stabilisation afin d'obtenir des ensembles les plus riches possibles. Du point de vue musical c'est assez intéressant tout de même d'avoir comme une sorte de banc de poissons à la fois chaotique et convergeant. Par exemple toutes les notes deviennent progressivement des rondes et des accords se constituent ; dans l'autre sens les notes s'égrainent petit à petit de plus en plus réduites à des pizzicatos.

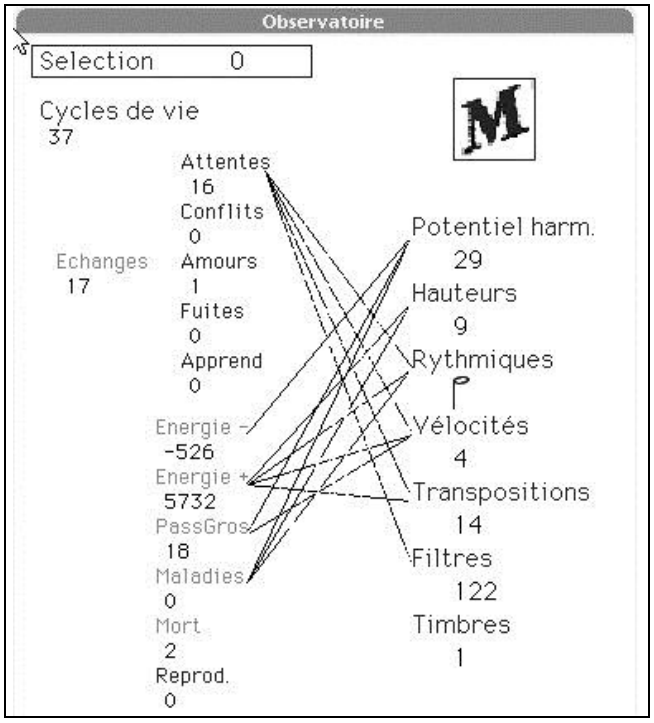


Fig. 7.3 : L'observatoire des notes volantes

[1] Ces concepts sont extraits de l'ouvrage :
BION W. R., *Recherches sur les petits groupes,* PUF, Paris, 1965. Pages 37 à 42.

Les Notes volantes sont écrites en langage C.

Version 1 :		Version 2 :	
Système :	Mac Os	Système :	Windows 98, XP
Compilateur :	Code Warrior.	Compilateur :	Visual C++ 6
Librairie :	Quickdraws	Librairie :	Allegro
Taille :	4700 lignes, 545 Ko	Taille :	2500 lignes, 880 Ko

1.2 DICTY L'AGREGUE

L'idée originale de ce programme revient à Stéphane Rideau auquel nous devons les informations rapportées ici. L'exactitude de ces informations n'est pas ici fondamentale. L'objectif n'est pas de faire une simulation réaliste, mais d'extraire un scénario qui soit à l'origine d'un programme. Le projet est de jouer avec un eucaryote qui est une variété de micro-organisme, un peu comme un musicien de jazz qui fait un chorus à partir d'un thème. Ceci n'invalide pas cependant le sérieux des informations rassemblées initialement. Elles le sont dans l'optique d'un travail d'écrivain qui, par exemple, appuie sa narration sur des événement historiques considérés comme véridiques. En l'occurrence l'eucaryote qui nous intéresse est le *dyctyostelium discoïdeum*, également appelé amibe de terreau.

Les cellules, appelées amibes, vivent sur le sol, indépendantes les unes des autres. Elles se nourrissent de bactéries et de levures. Dans les meilleures conditions, les amibes se divisent environ toutes les trois heures. Quand la nourriture manque, les cellules se rassemblent par chimiotaxie pour former de petits vers d'un ou deux millimètres de long (environ 10.000 cellules). Ces petites limaces se déplacent afin de trouver des conditions extérieures meilleures pour se reproduire. Elles rampent attirées par la lumière. La limace possède des propriétés que n'ont pas les amibes isolées. Pendant cette migration, les cellules de la limace commencent à se différencier pour aboutir lentement à une structure ressemblant à une plante minuscule (environ 30 heures après le début de l'agrégation). Ce corps végétatif contient alors des spores qui peuvent survivre longtemps dans un milieu hostile. Quand les conditions redeviennent favorables, les spores germent pour donner naissance à de nouvelles amibes et le cycle recommence.

L'idée est de faire un programme de formation d'agrégats. Le but est de voir se constituer des agrégats d'amibes en fonction des pénuries alimentaires. En jouant sur les paramètres du programme, on pourrait d'autre part obtenir une grande variété d'animaux pluricellulaires qui seraient viables ou monstrueux. Certains seront des limaces, d'autres auront des formes imprévisibles. Le jeu consistant à chaque cycle, à modifier automatiquement les paramètres afin de créer un bestiaire de pluricellulaires.

Dicty l'agrègue s'en tient au phénomène d'agrégation de cellules. Et les formes obtenues ne font pas l'objet d'une interprétation d'un niveau supérieur. Nous nous sommes d'abord attachés aux mouvements des cellules. L'ordre ainsi créé est déjà intéressant du point de vue visuel et peut être également envisagé du point de vue sonore. Mais le projet n'est pas clos. Un programme intitulé « Pour une vie de champignon » qui sera le premier roman programmé de la vie d'une amibe, est en cours de maturation. Et une version jeu peut suivre. A titre indicatif et pour se mettre dans l'ambiance, voici quelques éléments supplémentaires de l'enquête réalisée sur les amibes.

Enquête chez les amibes : l'agrégation par chimiotaxie

En cas de pénurie alimentaire, les amibes commencent à fabriquer de l'AMP cyclique (Adénosine Monophosphate cyclique). Cette molécule sert de signal chimiotactique. Ce signal attire d'autres amibes qui viennent s'agréger et constituer une limace. L'agrégation semble être initiée au hasard. Quelles que soient les cellules qui commencent les premières à sécréter de l'AMP cyclique, elles attirent d'autres cellules et elles deviennent centres d'agrégation. L'AMP cyclique que synthétisent ces cellules initiatrices est sécrété par pulsations, et il se fixe à la surface de récepteurs spécialisés à la surface des cellules voisines carencées et leur déplacement est orienté vers la source d'AMP (constitution d'un pseudopode en direction de la source d'AMP qui tire le corps cellulaire).

Le signal chimiotactique peut influencer des cellules éloignées des centres d'agrégation car les cellules transmettent le signal de cellule en cellule. Elles fonctionnent comme des relais. Chaque impulsion d'AMP cyclique détermine les cellules environnantes à se déplacer vers la source de l'impulsion et à sécréter leurs propres signaux d'AMP cyclique. De cette façon, des vagues successives régulières d'AMP cyclique se propagent à partir de chaque centre d'agrégation, provoquant la migration convergente des amibes les plus éloignées en vagues déferlantes concentriques ou en spirales. Un tel système de relais a l'avantage de renouveler le signal à maintes reprises, au fur et à mesure qu'il s'éloigne du centre, de sorte qu'il se propage sans affaiblissement sur une grande surface. Le Dyctyostélium minutum, par exemple, ne possède pas de système de relais. Le signal chimiotactique est donc de faible portée, la limace constituée est donc plus petite.

Sur le rôle de l'adhérence intercellulaire dans les amibes

La carence alimentaire ne provoque pas que la production d'AMP cyclique, elle provoque également l'activation de gènes**.** Parmi les gènes qui s'expriment, certains codent des molécules d'adhérence cellulaire. Ces molécules d'adhérence cellulaire sont évidemment impliquées dans le mécanisme d'agrégation. Par exemple, on suppose que la discoïdine-1, qui est une protéine de liaison des glucides, est sécrétée par des cellules carencées pour fournir une forme primitive de guidage de contact. En se liant à la fois à la surface des amibes et au substratum sur lequel celles-ci migrent, elle pourrait aider les amibes à affluer vers les centres d'agrégation, comme une traînée de fibronectine guide la migration cellulaire pendant la gastrulation. Différentes protéines néosynthétisées (glycoprotéines) favorisent le processus d'adhérence intercellulaire qui permet aux amibes migrantes d'adhérer étroitement les unes aux autres et de s'assembler en un organisme pluricellulaire.

Les composantes du programme

Le programme Dicty l'agrègue se décompose en cinq aspects principaux.

1. *Une fonction d'environnement.* Cette fonction a pour but de calculer la densité de nourriture dans l'environnement de façon à aboutir à des pénuries alimentaires. Ce sont ces pénuries qui vont déclencher le rassemblement des amibes (figure 7.4).
   Cette fonction doit pouvoir réapprovisionner l'environnement de façon à provoquer l'éclosion des spores et le commencement d'un nouveau cycle.

2. *Une fonction de déplacement « non agrégative ».* Les amibes se déplacent au sol à la recherche de levures et bactéries. Il faut donc prévoir une fonction de déplacement

individuel. Cette fonction peut être du type aléatoire contraint.

3. *Une fonction de nutrition.* Cette fonction simple doit déterminer pour chaque amibe quelle est la densité des ressources alimentaires dans son environnement immédiat. C'est une fonction à seuil. En-dessous d'un seuil, la production d'AMP est déclenchée, le processus d'agrégation initié. Si la nourriture est répartie aléatoirement dans l'environnement, les centres d'agrégation peuvent apparaître n'importe où.

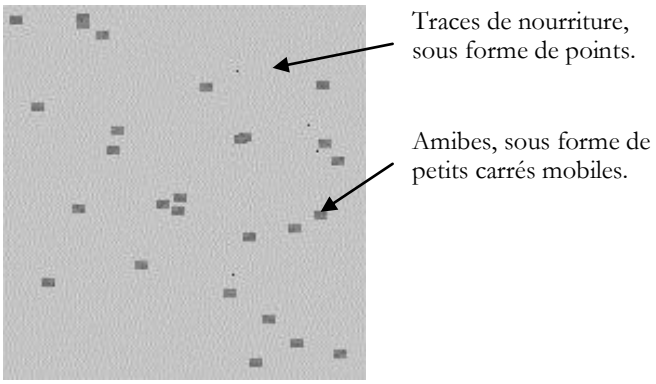


Fig. 7.4 : Les amibes se déplacent dans leur environnement nutritif avant toute pénurie alimentaire

4. *Une fonction de déplacement « agrégatif ».* Quand une amibe détecte une pénurie alimentaire, elle commence à produire un signal selon une fonction de diffusion. Quand une amibe reçoit ce signal, elle se déplace dans sa direction et émet à son tour le signal derrière elle.

   La production du signal peut être imaginée comme une onde chiffrée s'écartant de son point d'émission dans toutes les directions, en cercles concentriques avec des valeurs de plus en plus grandes. Un relief de chiffres est alors créé dans le plan en deux dimensions où circulent les amibes. Ensuite les amibes feront leur chemin en suivant toujours les valeurs numériques les plus petites, qui simulent à l'inverse un taux de concentration de l'AMP élevé (figure 7.5).

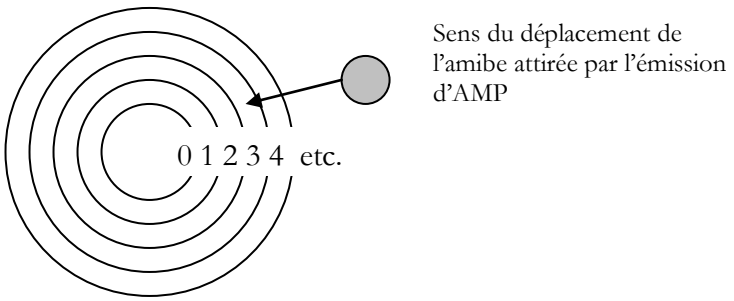


Fig. 7.5 : Bassin attracteur d'une amibe

Si plusieurs centres d'agrégation se forment, des fronts apparaissent entre les différents signaux. (figure 7.6) Eventuellement la trajectoire d'une amibe oscillera

entre deux bassins attracteurs. En principe elle ira vers le centre d'émission le plus proche c'est-à-dire qu'elle suivra les plus petites valeurs. En somme, il y a compétition entre différents bassins d'attraction. Cette situation est intéressante car elle va générer un relief en diagramme de Voronoï.

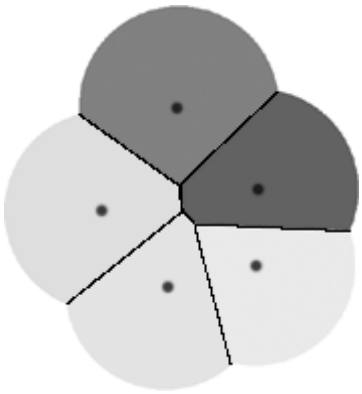
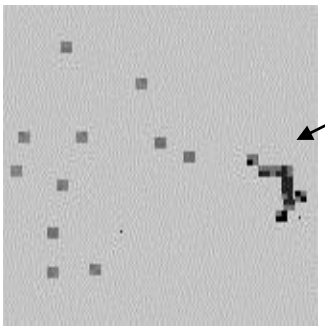


Fig. 7.6 : Le développement des bassins d'attraction crée entre eux des fronts et forment un diagramme de Voronoï

5. *Une fonction d'adhérence cellulaire.* Cette fonction d'adhérence cellulaire peut avoir un comportement multiple. C'est-à-dire que des molécules doivent déterminer des attirances et des répulsions. En cas d'attirance, le contact devient solide, la cellule se fixe. En cas de répulsion, la cellule se déplace à la recherche d'un site attirant. Une loi de guidage de contact peut être ajoutée. C'est-à-dire que des trames, peuvent guider les cellules vers des sites privilégiés (figure 7.7).



Des différences de teintes sur les amibes correspondent à zones plus ou moins attractives

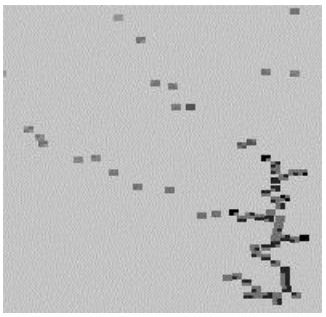
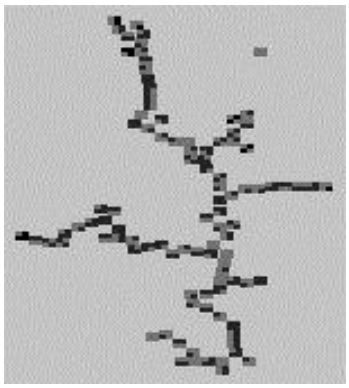


Fig. 7.7: (a) Déplacements agrégatifs après apparition d'un pôle de pénurie alimentaire.

(b) Emission du signal et constitution d'un pluri-cellulaire

« Dicty l'agrègue » est écrit en langage C.

Système :	Mac Os
Compilateur :	Code Warrior
Librairie :	Quickdraws
Taille :	2000 lignes



(c) Achèvement des pluri-cellulaires après absorption de toutes les

2. Systèmes génératifs

Autre domaine à explorer, celui des systèmes génératifs. Il n'y a probablement pas de limite à l'imagination de systèmes génératifs. A partir du moment où le résultat imprédictible d'une opération vient compléter une représentation quelconque on entre dans le génératif. La cohérence de cette imprédictibilité ainsi que la nature de la représentation mise en jeu constituent deux pôles pour la recherche et la création de systèmes génératifs. Le premier pôle s'intéresse surtout à la nature des opérations effectuées alors que le second se tourne davantage vers la structure sur laquelle ces opérations sont exercées. D'un côté des éléments fixes soumis à des opérations font évoluer une structure d'ensemble, de l'autre la structure d'ensemble est fixe et les éléments constitutifs sont variables.

Le pôle axé surtout sur les opérations peut être défini de la façon suivante *: l'itération de lois simples sur les éléments d'un ensemble organise ces éléments en figures ordonnées au sein de cet ensemble. Ces figures sont bien produites par l'application de ces lois, mais elles ne sont connues en tant que telles ni dans la définition ni dans l'application de ces lois.*
Admettons des opérations ou lois reportées sur des éléments. Lorsque dans un programme ces lois sont appliquées sur un très grand nombre d'éléments, elles composent *indirectement* des ensembles ou des figures d'ensemble. Ces compositions sont indirectes parce ces figures d'ensemble ne sont jamais désignées en tant que telles par les opérations effectuées. Elles découlent de l'itération de ces opérations sur les éléments pris séparément. Ces figures sont *générées*. Le résultat n'est accessible que par l'application de l'opération initiale et il ne peut jamais être connu sans l'application réitérée de cette opération.

Au chapitre un nous avons présenté un logiciel, Dialectic, dont le fonctionnement reposait sur un automate cellulaire. L'automate cellulaire est typiquement un système génératif de ce type. « Graphitos » et « Graphitor » sont conçus sur la base d'automates cellulaires. Ils rejoignent une famille d'automates cellulaires impliqués dans le Computer Art [CAILLAUD, 2001, p65 à 87].

Discours coton, que nous présentons ensuite, illustre l'autre versant que nous avons proposé pour la générativité : une structure fixe à contenu variable. Imaginons un coloriage. C'est la structure fixe. Il y a différentes formes déterminées à colorier. Dans ce cadre ce seront les couleurs qui seront générées. Elles constituent le contenu variable. Dans Discours coton ce principe est transposé pour la génération de textes. Le coloriage est un moule à phrase c'est-à-dire un syntagme fixe, et les couleurs la possibilité d'agir sur le paradigme des éléments constitutifs du syntagme, à savoir des ensembles de mots ayant même nature grammaticale (noms, verbes, abjectifs etc.).

2.1 GRAPHITOS

A l'origine de Graphitos, il y a la réalisation du logiciel musical Dialectic présenté brièvement au chapitre un. Dans Graphitos le souhait est toujours d'explorer les possibilités d'un automate cellulaire auquel est ajouté un milieu intracellulaire, mais cette fois pour produire des images.

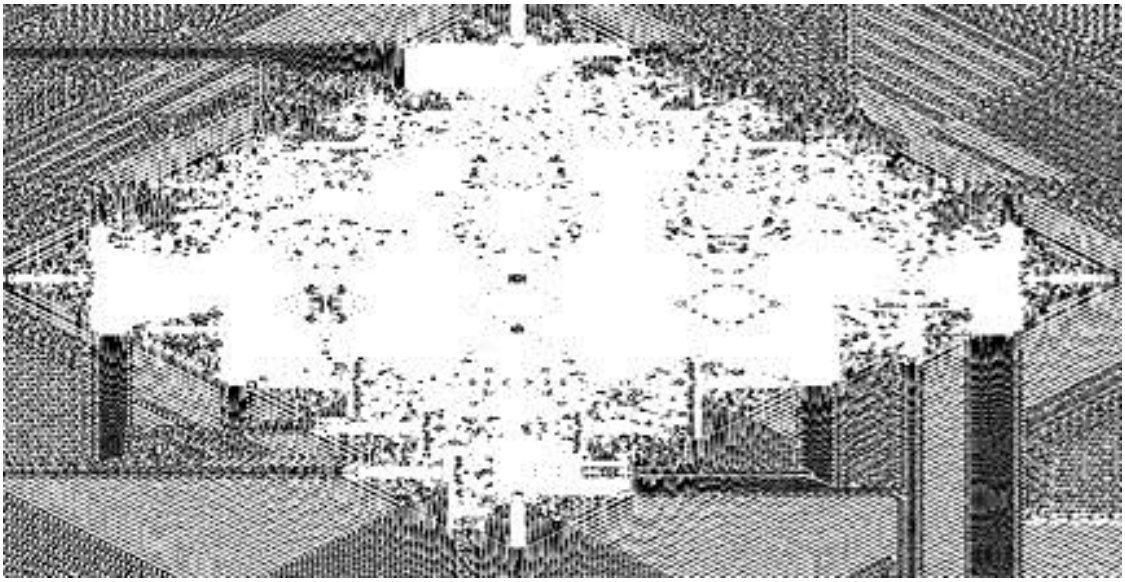


Fig. 7.8 : Graphitos : construire une déchirure.

Graphitos possède deux niveaux d'organisation. Le plan en deux dimensions qui correspond à l'écran, et chaque position ou cellule dans ce plan qui possède une profondeur. C'est l'équivalent de trois dimensions mais il n'y a pas de rencontre du point de vue de la profondeur entre les cellules. Chaque milieu intracellulaire est lui-même un automate cellulaire unidimensionnel ayant ses propres modalités de fonctionnement.

Principe algorithmique de Graphitos

Afin de comprendre les commandes associées à Graphitos il est nécessaire de présenter le fonctionnement du moteur sous-jacent. C'est d'ailleurs le même principe que celui de Graphitor. Nous n'allons présenter ici que le moteur du niveau supérieur, celui qui organise le plan en deux dimensions où se développe l'automate sous la forme d'une image. Ce moteur s'inscrit dans la tradition initiée dans les années soixante par John Horton Conway et nommée « Jeu de la vie[1] ».

Principes initiaux du moteur supérieur

1. Chaque position dans le plan est dotée d'une valeur soit de zéro soit de un. Au départ toutes les positions du plan sont mises à zéro et une fonction d'initialisation permet d'en mettre quelques-unes à un.

2. Ensuite toutes les positions du plan sont passées en revue. Pour chacune d'entre elles les positions voisines sont regardées et les valeurs trouvées comptabilisées (figure 7.9).

3. La valeur de la position courante va évoluer ou rester identique en fonction d'une règle de transition. La règle de transition repose sur les valeurs trouvées dans les positions voisines. Dans l'exemple de la figure 7.9, la position courante

[1] Pour des éléments à ce sujet consulter [HEUDIN, 1994, p.45 à 52] [OSTOLAZA, 1997, p. 37 à 42] et [RENNARD, 2002, p. 65 à 76]

comptabilise trois 1 trouvés dans les cellules adjacentes. En quelque sorte il y a trois voisins. La règle de transition repose sur ce nombre de voisins.

(x-1, y-1) avec une valeur de 0	(x, y-1) avec une valeur de 1	(x+1, y-1) avec une valeur de 1
(x-1, y) avec une valeur de 1	(x , y) avec une valeur de 0 ou 1	(x+1, y) avec une valeur de 0
(x-1, y+1) avec une valeur de 0	(x, y+1) avec une valeur de 0	(x+1, y+1) avec une valeur de 0

Fig. 7.9 : (x, y) est la position courante. « x » est pour la position horizontale et « y » pour la position verticale. Il y a trois positions adjacentes qui valent 1.

Par exemple on aura une règle du type : si le nombre de voisins est inférieur à deux ou si le nombre de voisins est supérieur à trois, c'est-à-dire si le nombre de voisins est différent de 2 ou de 3, alors la position courante prend une valeur de un, sinon elle prend une valeur de zéro. Mais on peut très bien faire varier le nombre de voisins entre zéro à huit positions adjacentes, et on peut changer l'attribution de la valeur.

Ceci nous donne un processus général :
Pour la position courante, si le nombre de voisins est inférieur à « un plancher » ou si le nombre de voisins est supérieur à « un plafond » alors la position courante prend la valeur booléenne « v1 » sinon elle prend la valeur booléenne complémentaire « v2 ».

4. Il convient de préciser que le calcul se fait selon les positions du plan mais que les résultats, pour chaque position, sont stockés au fur et à mesure dans un plan équivalent en miroir du premier. Une fois que les valeurs de chaque position ont été recalculées et stockées dans le plan réservé aux résultats, ce dernier est recopié dans le plan initial et l'opération recommence (figure 7.10).

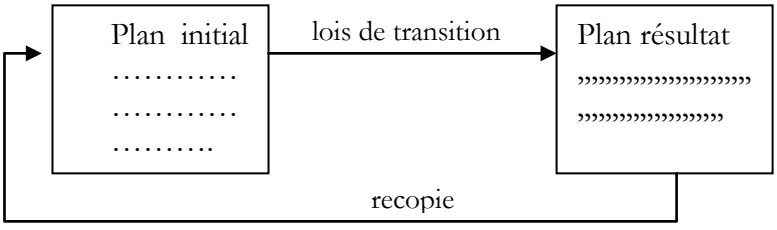


Fig. 7.10 : Boucle pour l'application de la transformation au plan initial

Maintenant que nous avons tous les éléments il ne nous reste plus qu'à monter le moteur. Mais pour ce faire nous avons besoin de six figures supplémentaires du langage C : une matrice, deux opérateurs de comparaison, deux opérateurs d'expressions logiques et l'instruction condition « if » éventuellement suivie de « else ». La matrice est un tableau à deux dimensions qui va nous servir à coder les deux plans « initial » et « résultat ». Les deux opérateurs de comparaison sont tout simplement le « < » *est inférieur à* et le « > » *est supérieur à*. Les deux opérateurs d'expressions logiques sont le *est égal à* représenté par == et le *OU* représenté par ||. Nous allons les présenter dans l'ordre de leurs apparitions dans la réalisation du moteur de Graphitos.

Matrice

Au chapitre Trois nous avons rencontré un tableau à une seule dimension du type :
int tab[nombre_max] ;

Maintenant nous rencontrons un tableau à deux dimensions[1], une matrice, du type :
int mat[nombre_lignes][nombre_colonnes] ;

Par exemple : « int mat[2][3] ; » contient 2x3 = 6 entiers. On pourrait l'initialiser comme suit avec n'importe quelle valeur :

```
mat[2][3] ={1, 2, 3,  4, 5, 6 } ;
```

première et seconde lignes

ou encore de façon plus explicite de la façon suivante :

```
mat[2][3] = { { 1, 2, 3} ,      // première ligne
              { 4, 5, 6}   } ; // deuxième ligne
```

La première ligne correspond aux postions mat[0][0], mat[0][1], mat[0][2], avec des valeurs respectives de 1,2 et 3. La seconde ligne correspond aux positions mat[1][0], mat[1][1], mat[1][2], avec des valeurs respectives de 4, 5 et 6. Les trois colonnes étant constituées des couples (1,4), (2,5), (3,6). On remarque qu'il s'agit somme toute d'un tableau de tableaux. Chaque ligne correspond à un tableau de trois éléments.

Dans le moteur que nous allons présenter nous allons utiliser deux matrices de booléens (un booléen prend uniquement une valeur de zéro ou de un). La première va correspondre au plan initial présenté sur la figure 7.10 et la seconde au plan résultat. La taille de ces matrices sera définie par deux constantes (des valeurs fixes non modifiables dans le programme) . Par exemple ce peut être la taille de l'écran ; chaque position correspond alors à un pixel. Au début du programme nous aurons une déclaration comme suit :

```
bool INITIAL[tailleX][tailleY] ;
bool RESULTAT[tailleX][tailleY] ;
```

Au début du programme également ces matrices, nommées respectivement INITIAL et RESULTAT seront initialisées à zéro et quelques positions seulement seront mises à un.

[1] En fait, dans le langage C, le concept de tableaux est étiré à celui de tableaux multi-dimensionnels . Il peut y avoir des tableaux à « n » dimensions.

Opérateurs d'expressions logiques « est égal à »

Ensuite, pour chaque position nous allons devoir réaliser des tests. Nous allons devoir regarder toutes les cellules voisines, comme sur la figure 7.9, et pour chacune d'elles savoir si elle *est égale à* un. A chaque fois qu'une cellule est égale à un, une variable, qui est mise à zéro au départ, est incrémentée de un, de façon à pouvoir compter toutes les cellules à un trouvées. Pour savoir si une cellule est à 1 nous allons utiliser l'opérateur d'expression logique « == ».

E1==E2 (E1 *est égal à* E2) teste l'égalité entre deux expressions E1 et E2. Si l'expression E1==E2 est vraie elle vaut la valeur booléenne 1. Si elle est fausse elle vaut 0. A ne pas confondre avec l'opérateur d'affectation « = » qui lui affecte une valeur à une expression.

Pour une position (x, y) dans la matrice nous aurons une expression du type :
(INITIAL [x+1][y] ==1) et cette expression sera vraie ou fausse. Si INITIAL[x+1][y] vaut 0, l'expression (INITIAL [x+1][y] ==1) est fausse et retourne 0. Si INITIAL[x+1][y] vaut 1, l'expression (INITIAL [x+1][y] ==1) est vraie et retourne 1.

Instruction condition ***if, else***

Ce propos nous amène à rendre compte de l'écriture du « SI ».
L'expression (INITIAL [x+1][y] ==1) détermine finalement deux cas : si elle est vraie, si elle est fausse. **SI** elle est vraie on fait les instructions prévues pour vrai **SINON** on fait les instructions prévues pour faux.

C'est l'instruction « if » qui permet d'opérer un test du type « Si l'expression E est vraie (E vaut 1) alors faire le bloc qui suit » . A cette instruction peut s'adjoindre le « else », le sinon. C'est-à-dire que si E est fausse (E vaut 0) il faut faire le bloc suivant. Le else est cependant facultatif.

La syntaxe du if est la suivante :

```
if (expression)          // si vraie
   instruction1 ;        // faire ce bloc
else                     // sinon
   instruction2;         // faire celui là
```

Ainsi pour connaître les valeurs des positions adjacentes à une position on aura un test du type :
```
if( INITIAL [x+1][y] ==1 ) //si vraie
   comptevoisin= comptevoisin+1 ;
  /* ajouter un au nombre de voisins trouvés.*/
```
Si faux il n'y a rien à faire dans ce cas et le else peut être ignoré.
Pour chaque position, une fois que nous avons le nombre de voisins il faut appliquer la loi de transition.

Opérateurs de comparaison « est inférieur à » et « est supérieur à »
Opérateur d'expression logique « OU »

Nous avons défini plus haut la loi utilisée dans Graphitos par : p*our la position courante, si le*

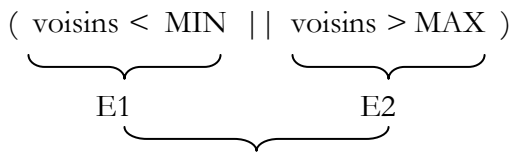
nombre de voisins ***est inférieur à*** *« un plancher »* ***OU*** *si le nombre de voisins* ***est supérieur à*** *« un plafond » alors la position courante prend la valeur booléenne « v1 » sinon elle prend la valeur booléenne complémentaire « v2 ».* Nous allons avoir à évaluer en termes de vrai ou faux (un ou zéro) une expression du type :
« voisins *est inférieur à* plancher *OU* voisins *est supérieur à* plafond »

Les deux opérateurs *est inférieur à* et *est supérieur à* plafond sont identiques à leurs homologues mathématiques. A<B signifie que A est plus petit que B et A>B signifie que A est plus grand que B. Dans un programme une expression du type A<B, si elle est vraie, donne une valeur de 1 et de 0 sinon. Par exemple si A vaut 5 et B vaut 7, l'expression (A<B) est vraie et vaut 1.

En ce qui concerne l'opérateur d'expression logique OU, noté « || ». Le résultat de l'expression E1||E2 (E1 OU E2) est 1 (vrai) si l'une au moins des deux valeurs E1 ou E2 est non nulle, et 0 sinon. De ce fait l'expression E2 n'est évaluée que si l'expression E1 est nulle (fausse).

Dans le moteur de Graphitos, afin de mettre en œuvre la loi de transition nous devons évaluer l'expression « voisin *est inférieur à* plancher *OU* voisin *est supérieur à* plafond », avec :
int voisin ; une variable qui contient le nombre des voisins.
int MIN ; une variable qui correspond au nombre minimum de voisins (plancher). Sa valeur peut osciller entre 0 et 8 voisins.
int MAX ; une variable qui correspond au nombre maximum de voisins (plafond). Sa valeur peut osciller entre 0 et 8 voisins. MIN et MAX sont définies dans l'interface (figure 7.12).

Ce qui donne l'expression :



E1 est évaluée. Si E1 est vraie, (E1|| E2) est vraie et c'est tout.
Si E1 est fausse, E2 est évaluée.
Si E2 est vraie (E1|| E2) est vraie. Sinon (E1|| E2) est fausse.

Cette expression détermine à nouveau deux cas : si elle est vraie, si elle est fausse. Et comme précédemment lors du comptage des voisins, **SI** elle est vraie on fait les instructions prévues pour vrai **SINON** on fait les instruction prévues pour faux. C'est-à-dire que nous allons utiliser à nouveau le *if-else.*

Dans le moteur de Graphitos nous aurons :

```
if( voisins <  MIN || voisins > MAX ){
        //les instructions si vraie
}
else{
        // les instructions si fausse
 }
```

Moteur du niveau supérieur dans Graphitos, les fonctions calcul et compte

Quoique simplifiée afin d'en faciliter la présentation, la figure 7.11 présente le code sur lequel est fondé l'algorithme du moteur de niveau supérieur.

(1) Déclaration de variables internes à la fonction.

(2) Boucle imbriquée qui permet de passer en revue et simultanément toutes les positions des matrices INITIAL et RESULTAT.

(3) La ligne trois fait appel à une autre fonction, la fonction « compte ». Cette fonction compte les positions qui sont à un dans les positions adjacentes comme montré dans la figure 7.9.

(3.1) La fonction « compte » prend deux arguments qui sont les paramètres d'une position dans la matrice INITIAL.

(3.2) Au début de la fonction une variable destinée à recevoir le nombre de voisins est initialisée avec la valeur 0. Avant de commencer à compter il y a zéro voisins.

```
void calcul() {
(1)   int i, j, voisins;
(2)       for(i=1; i<tailleX-1; i++) {
            for (j=1; j <tailleY-1; j++){

(3)           voisin = compte(i , j );

(4)           if(INITIAL [ i ][ j ] == 1){

(5)             if(voisin < MIN || voisin > MAX)
                  RESULTAT [ i ][ j ] = cas1;
(6)             else
                  RESULTAT [ i ][ j ] = cas2;
                }
(7)             else{

(8)             if(voisin < MIN || voisin > MAX)
                  RESULTAT [ i ][ j ] = cas3;
(9)             else
                  RESULTAT [ i ][ j ] = cas4;
                }
              }
            }
      }
```

(a)

```
(3.1) int compte (int i, int j){
(3.2) int comptevoisin = 0;

(3.4) if (INITIAL [ i+1 ][ j ] ==1)
          comptevoisin++;
      if(INITIAL [ i+1 ][ j-1 ] == 1)
          comptevoisin++;
      if(INITIAL [ i ][ j-1 ] == 1)
          comptevoisin++;
      if (INITIAL [ i-1 ][ j-1 ] == 1)
          comptevoisin++;
      if (INITIAL [ i-1 ][ j ] ==1)
          comptevoisin++;
      if (INITIAL [ i-1 ][ j+1 ] ==1)
          comptevoisin++;
      if (INITIAL [ i ][ j+1 ] ==1)
          comptevoisin++;
      if(INITIAL [ i+1 ][ j+1 ] ==1)
          comptevoisin++;

(3.4)     return comptevoisin ;
}
```

(b)

Fig. 7.11 : (a) source de la fonction de transition du moteur de niveau supérieur
(b) détail de la fonction « compte » appelée dans calcul

(3.3) les valeurs contenues dans chaque position adjacente à la position (i , j) passée en argument sont évaluées. Si une position à 1 est trouvée, la variable qui

compte les voisins « comptevoisin » est incrémentée de 1.

(3.4) La fonction « compte » renvoie le nombre trouvé à la fonction appelante, c'est-à-dire la fonction « calcul », et cette valeur est affectée à la variable « voisin » .

(4) et (7) sont symétriques selon la définition du *if-else*. Il s'agit d'un test sur la valeur de la position courante. Si dans INITIAL à la position courante la valeur est 1, (5) et (6) sont exécutés. Sinon on passe directement à (8) et (9).

(5) et (6) Nous sommes sur une position à 1 dans INITIAL et il s'agit du test sur la loi de transition. Si le test est vrai alors nous mettons la valeur de « cas1 » à la même position dans RESULTAT. Sinon nous mettons « cas2 ». cas1 et cas2 sont des valeurs booléennes de zéro ou un définies dans l'interface (figure 7.12).

(8) et (9) Pareil à (5) et (6) mais nous sommes sur une position à 0 dans INITIAL. Si le test est vrai alors nous mettons la valeur de « cas3 » à la même position dans RESULTAT. Sinon nous mettons « cas4 ». cas3 et cas4 sont des valeurs booléennes de zéro ou un définies dans l'interface (figure 7.12).

Commandes de Graphitos

Les clauses d'environnement du niveau supérieur dont nous venons de voir l'algorithme sont paramétrables. Et pour le niveau inférieur, dont nous n'explicitons pas l'algorithme pour ne pas alourdir notre propos, le développement de chaque position est contenue dans un espace de vie au-delà duquel c'est la mort. Cet espace est extensible et comprimable, et il déstabilise le niveau supérieur. De plus, le milieu intracellulaire conduit le développement coloré des images et permet d'obtenir des effets de relief.

Les commandes possibles sont les suivantes, c'est l'action sur les paramètres de fonctionnement de la loi de transition de l'automate :

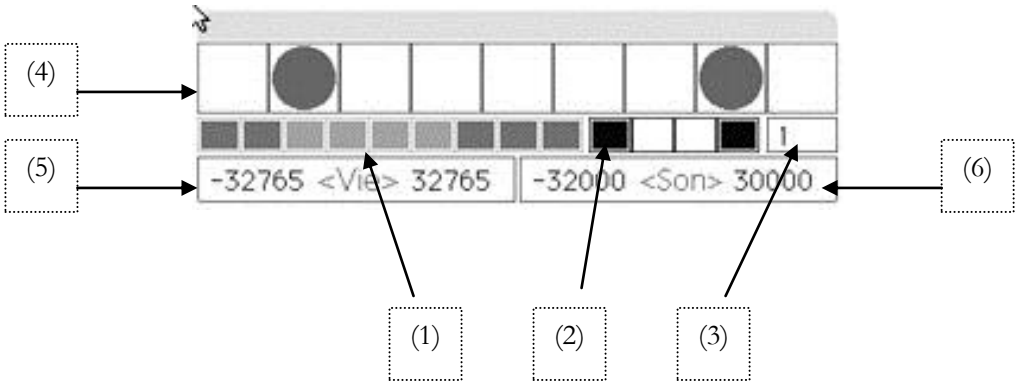


Fig. 7.12 : Commandes dans l'interface du moteur de graphitos.

(1) En couleur c'est une barre avec les voyants rouges. Elle permet de définir un intervalle pour la loi de transition. C'est par là que les variables MIN et MAX sont passées au moteur. A gauche c'est la valeur pour MIN, à droite c'est la valeur pour MAX. Sur l'image MIN égale 2 et MAX égale cinq (la numérotation va de zéro à huit).

(2) En jaune sont paramétrés les différents cas possibles pour la loi d'environnement. Dans l'ordre ce sont les variables cas1, cas2, cas3, cas4 du moteur. Activée, la variable est à un, sinon elle est à zéro. Selon le paramétrage actuel de l'interface, dans le moteur, cas1 égale zéro, cas2 égale 1, cas3 égale 1 et cas4 égale zéro.

(3) Cette case permet de saisir la portée, le rayon autour de la cellule pour le compte des voisines. Dans la fonction « compte » que nous avons présentée, la distance à la position courante est toujours de un en plus ou en moins en x ou/et en y. Mais il peut en être autrement et le calcul eut pu être envisagé avec une portée plus longue. Cet aspect n'est pas visible dans le code source tel que nous l'avons présenté. Le voisinage pourrait correspondre à une distance de deux, trois, quatre etc. Dans ce cas le nombre des cellules voisines possibles croît avec la distance au centre et correspond à cette distance multipliée par huit. Mais dans Graphitos nous avons conservé un nombre fixe de huit voisines qu'il est possible d'aller chercher à des distances variables.

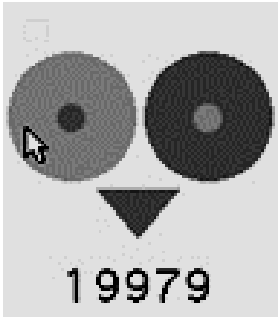
(4) Cette partie concerne l'initialisation de l'automate cellulaire interne à chaque position. Elle permet de configurer sa valeur de départ dans son espace local. Pour chaque cellule du plan, il y a une profondeur de neuf cellules que les cases représentent. Une case avec un cercle foncé est sur un, zéro sinon. Cet automate local est inspiré par celui de [OSTOLAZA, 197, p. 39].

(5) et (6) sont des paramétrages également réservés à l'automate interne. **<Vie>** correspond à une variable numérique qui détermine l'espace de viabilité pour une cellule active, compte-tenu d'un automate intrinsèque à la cellule, surajouté à la position booléenne. Idem pour le son (cependant Graphitos n'a pas été sonorisé).

L'initialisation des cellules au démarrage de l'automate se fait avec la souris. La souris détermine une position à l'écran dans la zone réservée au dessin de l'automate. Si l'automate fait la taille de l'écran, les matrices INITIAL et RESULTAT ont la taille de l'écran et donc la position de la souris correspond directement à une position dans ces matrices. Cliquer positionne des points dans le plan et ces points, au regard de l'automate interne, sont dotés des paramétrages actifs dans la fenêtre de commande.

Graphitos dispose également d'une autre fenêtre de commande pour la mise en œuvre, l'arrêt et la réinitialisation (figure 7.13)

Fig. 7.13 : Le rond clair est la commande « joue », le rond plus foncé est le « stop ». Le chiffre indique le nombre de cycles réalisés, et le triangle permet de réinitialiser tout, d'effacer l'écran.



Graphitos est écrit en langage C.

Système :	Mac Os
Compilateur :	Code Warrior
Librairie :	Quickdraws
Taille :	2800 lignes

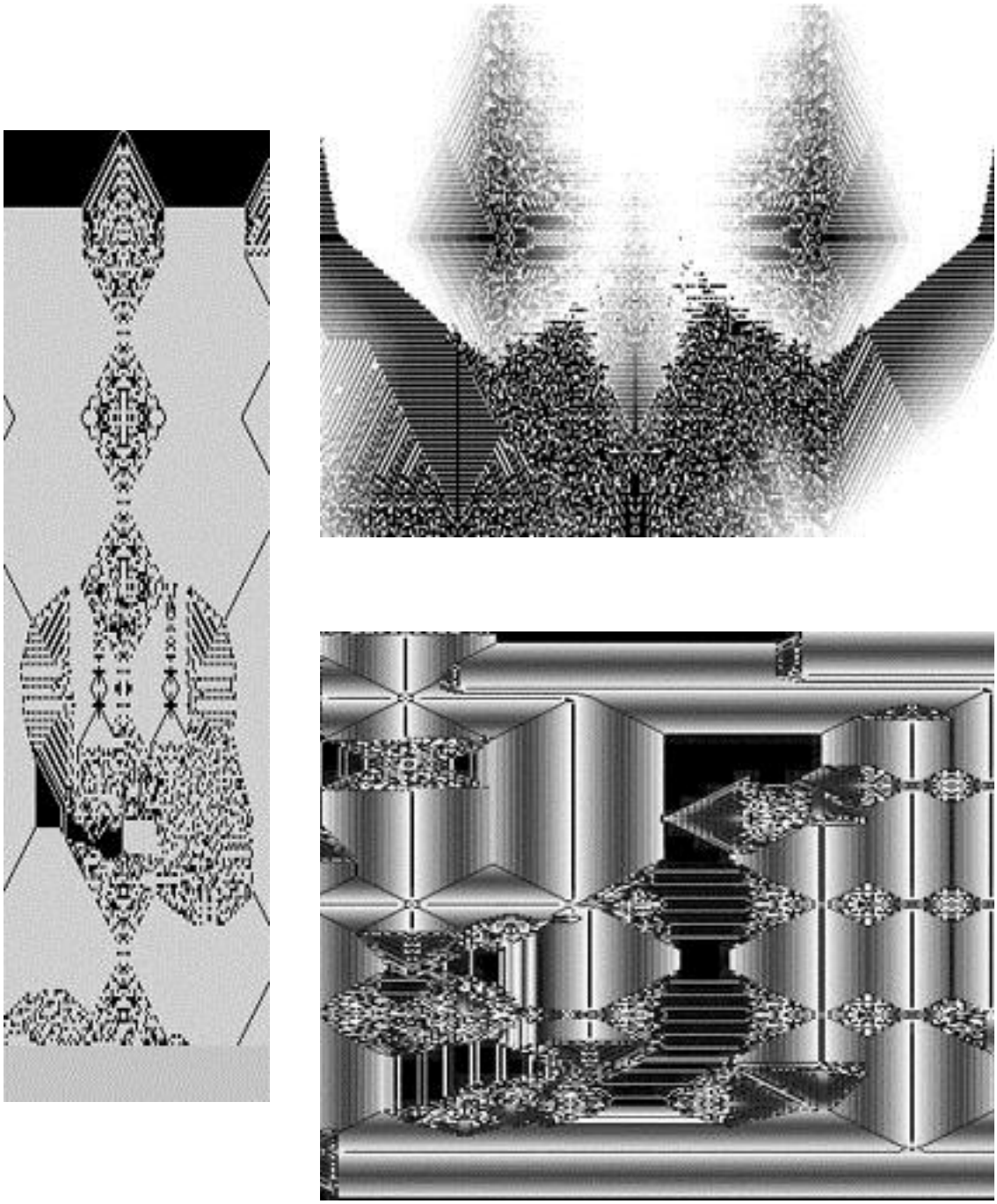


Fig 7.14 : Images de Graphitos

2.2 GRAPHITOR

En principe Graphitor devait être le portage sur PC Windows de Graphitos créé sous Mac Os. La seule différence était d'ajouter à Graphitos une palette graphique pour diversifier et considérablement augmenter les possibilités d'initialisation. Mais deux créations différentes sont ressorties et, ce qui est intéressant, sur la base d'une même idée et d'un même concept. Dans le chapitre Trois, à propos de la modélisation dans un projet, nous avons

illustré avec Graphitor l'importance de la réalisation expérimentale qui permet de découvrir des propriétés « involontaires » d'un programme et éventuellement d'en tirer bénéfice pour l'orientation d'une recherche et d'une réalisation.

Le moteur de Graphitor est réduit à deux dimensions, soit uniquement le moteur de niveau supérieur de Graphitos. Le processus de coloration des points du plan n'est plus le fait d'un automate cellulaire local pour chaque cellule du plan, mais d'un choix entre cinq propositions de styles. Pour la conception du moteur il y a tout de même une petite différence de point de vue. Dans le moteur de Graphitos, pour chaque position on regarde les cellules voisines et les positions non nulles sont comptées. Dans le moteur de Graphitor, chaque changement pour une position donnée informe ses voisines. Ce qui fait que chaque position « connaît » l'état des cellules voisines, le nombre des voisines à un.

Mais la modification la plus conséquente concerne les possibilités d'initialisation de l'automate. D'autant que ces possibilités sont actives pendant le fonctionnement de l'automate. C'est-à-dire qu'il est possible de dessiner à partir d'un matériau en construction, dans une sorte de dialogue avec l'automate, une interaction. Cette idée est vraiment intéressante du point de vue graphique. Et il y a matière à réfléchir à une variété spécifique d'outils pour une telle approche de la réalisation graphique.

Une petite palette graphique (figure 7.15) permet de tracer des figures simples. Ces formes soit colorent, et cela correspond pour l'automate à une mise à un, soit effacent et cela correspond pour l'automate à une mise à zéro. Ce sont des points, des nuages de points, des droites, des rectangles, des triangles, et des cercles. Une gomme est ajoutée qui permet d'effacer des zones en promenant un cercle de rayon plus ou moins grand dans la zone de l'écran.

Fig. 7.15 : Possibilités d'initialiser l'automate en traçant des formes.

Par ailleurs il y a aussi la possibilité d'incorporer une image dans le dessin, de transformer la palette de couleurs et de sélectionner un style de coloration parmi les cinq proposés (figure 7.16).

Fig. 7.16 : Sélecteur d'images, de palette de couleurs et de style de coloration

Comme pour Graphitos, il est possible d'agir sur des paramètres du moteur. Sont accessibles la fourchette du nombre de voisins, la proximité des voisins et les lois de transition qui sont pratiquement les mêmes. Petite variante, un paramètre de résistance qui permet de graduer le passage à un. Au lieu d'avoir comme précédemment :

```
if (INITIAL [ i ][ j ] == 1){
        …..
}
```

On va avoir :

```
if (INITIAL [ i ][ j ] > resist){
        …..
}
```

Et « resist » est défini via l'interface avec les autres paramètres de moteur. Pour Graphitor les lois sont préétablies et accessibles via un sélecteur(figure 7.17). Nous n'avons conservé

que sept lois sur les seize combinaisons possibles :

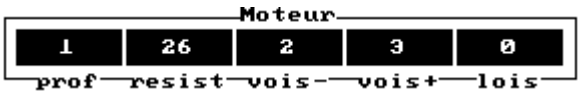


Fig. 7.17 : Paramétrer le moteur de Graphitor.

Enfin reste ce qui concerne la réinitialisation, la mise en œuvre et l'arrêt, ainsi que la possibilité de récupérer des photos de l'automate sous forme d'images bitmaps (figure 7.18).

Fig. 7.18 : Commandes de base de Graphitor

La mystérieuse fonctionnalité du programme qui est devenue essentielle et qui pourtant n'a pas été programmée en tant que telle, a rapport avec l'initialisation par des formes (figure 7.15). En principe toutes les cellules couvertes par la forme tracée sont mises à un de la façon suivante : INITIAL[x][y]=TRUE ;

A priori TRUE est une valeur booléenne qui vaut 1 dans les environnements de développement. Mais il se trouve que dans celui utilisé pour Graphitor, TRUE est défini par la valeur négative -1. Nous ne soupçonnions en rien ce détail lors de la programmation de Graphitor. Les conséquences ont été importantes pour le fonctionnement du moteur et l'idée même du logiciel. En effet, en dépit de la différence de point de vue entre Graphitos et Graphitor, dans le moteur nous avons en commun un test de ce type :

```
if(INITIAL[x][y]>resist){
        tests sur les voisins MIN et MAX
        application loi de transition en fonction des cas 1 et cas 2
}
if(INITIAL[x][y]==0){
        tests sur les voisins MIN et MAX
        application loi de transition en fonction des cas 3 et cas 4
}
```

Notons le fait qu'il n'y a pas de else dans Graphitor, ce qui fait que les deux tests sont effectués. Ainsi, sachant que la variable resist est toujours positive, si INITIAL[x][y] vaut -1 les deux tests sont faux. Donc aucune modification ne peut jamais être effectuée sur ces positions. D'où le gel de l'image, car ces positions échappent ensuite définitivement au développement de l'automate. Pour les réinjecter dans le développement il faut les mettre à zéro.

C'est par hasard que cette propriété potentielle du moteur a été découverte, et ce qui est intéressant, c'est que nous n'avions pas même pensé à une telle propriété. Il était peu probable que nous nous y intéressions. Nous retrouvons la problématique du chapitre un, lorsque nous avons créé un logiciel de musique et agit avec lui sur une habitude de l'approche musicale. La création involontaire d'une situation nouvelle alliée à une certaine adaptabilité de la perception découvre une voie à explorer.

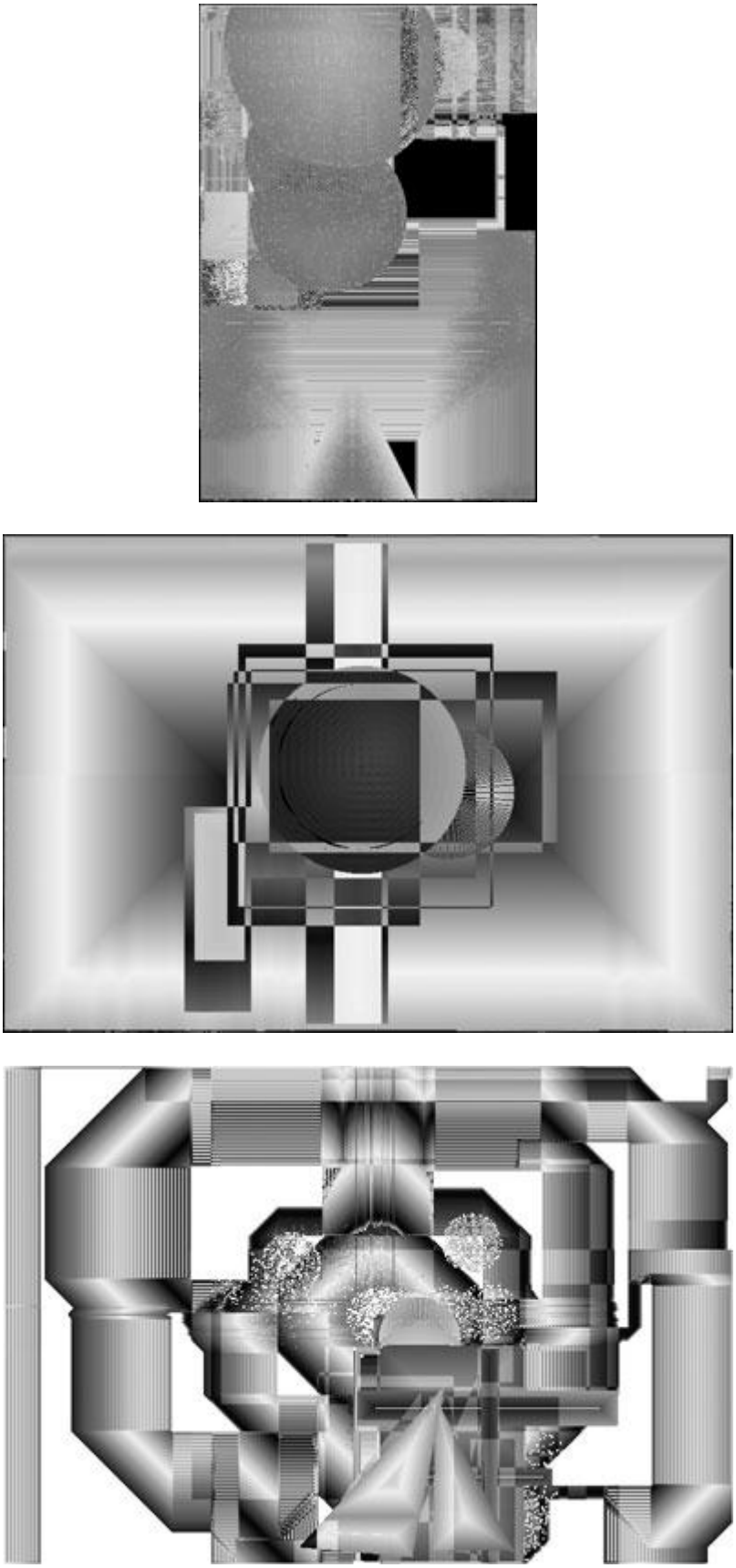


Fig. 7.19 : Trois compositions avec Graphitor.

Graphitor est écrit en langage C.
Système : Windows
Compilateur : Visual C++6
Librairie : Allegro
Taille : 2800 lignes

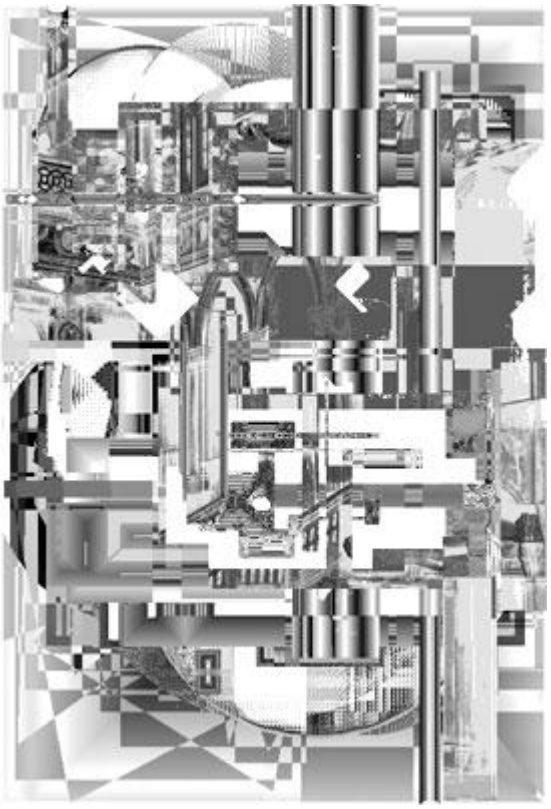


Fig. 7.20 Autre composition avec Graphitor

2.3 DISCOURS COTON

Les traitements en relation avec textes et langues ont fait l'objet de recherches importantes qui ont marqué les débuts de l'intelligence artificielle et probablement le mouvement de la linguistique structurale avec des personnalités comme Bar-Hillel, Bloomfield, Harris et Chomsky [GANASCIA, 1992, p. 71 à 79]. Discours coton est une expérimentation de générateur de texte, thème approfondi notamment dans les travaux de Jean-Pierre Balpe [BALPE, 2000].

Inspiré par *La langue de coton* de François-Bernard Huyghe [HUYGHE, 1991] il s'agit du développement d'un constructeur de discours creux en langue coton. C'est la langue du discours médiatique par excellence éventuellement pratiquée en politique. Ce projet a fait l'objet de plusieurs tâtonnements, cinq expériences différentes. Deux grands aspects ont pu être observés : la génération automatique de type paradigmatique, par catégories de mots et d'expressions, avec des résultats intéressants, et la génération automatique de syntagmes éventuellement étendus à la production de paragraphes entiers, beaucoup plus difficile.

La langue de coton

François-Bernard Huyghe se livre à une critique féroce de ce qu'il considère, dans les médias en général, comme une technique grinçante de l'art de la séduction à des fins de manipulation, de dissimulation ou de pouvoir. Nous rapportons ici quelques éléments de l'auteur pour se faire une idée de la langue de coton :

> « C'est la langue qui pense pour vous. Elle est assez prévisible dans sa forme pour ne transmettre que le sens le plus probable. Ses mots sont séduisants, obscurs ou répétitifs; elle s'orne d'ébouriffantes parures, elle emploie des catégories si vastes, flatte tant le goût du truisme et de la tautologie que nous oublions l'essentiel : elle permet de dire que ce que nous attendions. Elle ne trahit jamais. Qu'elle opte pour l'obscurité ou la platitude, le flou ou la redondance, elle sacrifie chaque fois l'information. Banale ou ésotérique, elle est toujours rétive à la surprise. Elle a réponse à tout parce qu'elle n'énonce presque rien. Ou trop, ce qui revient au même. (...) Elle est automatique, irréfutable et efficace » [HUYGHE, 1991, p 12].

Il résume ainsi les caractéristiques de cette curieuse langue :

> « _ Pour que sa production soit rentable, elle doit avoir un coup minimum : minimum d'efforts, de vocabulaire, d'idées, de recettes...etc.;
> _ La LCD doit être invérifiable : pas de critère qui permette de prouver que ses affirmations contredisent les faits ou se contredisent entre elles ;
> _ Elle doit à la fois concurrencer la langue naturelle, et s'y substituer autant que possible ;
> _ Il faut que le tout passe inaperçu.
> La LDC sert surtout à produire du commentaire à usage médiatique. (...) Les multiples manières de combiner les mêmes stéréotypes autorisent les variations personnelles. » [HUYGHE, 1991, p. 24].

Du point de vue technique, à propos de la rédaction dans une telle langue l'auteur dégage trois éléments fondamentaux :

> « _ Le " mixage " qui permet par exemple de produire un discours qui puisse servir indifféremment pour n'importe quel parti politique. (...) Technique non réservée au domaine politique elle peut être employée en art (...)
> _ Il existe également un exercice dit de la " matrice " : il consiste à combiner totalement au hasard les mots d'une liste préétablie. Le but est de former des phrases applicables à n'importe quel sujet(...)
> _ L'invention de mots nouveaux (...) Rajoutez des préfixes aux mots. Déclinez sur le principe Batman, Batmobile, batcave, etc.; rajoutez : -iste, isme, ition, itude, isation, isationnisme ... » [HUYGHE, 1991, p 48,51,52].

D'autre part il fournit un lexique en annexe, une longue liste de mots adéquats.

Vers une poésie de la langue de coton automatisée

C'est l'exercice de la matrice qui est exécuté dans le programme et il faut bien avouer que la thèse de l'auteur est assez juste. On obtient en effet des textes prêts à révéler le sens que l'on veut bien leur donner. C'est très intéressant du point de vue de la génération de texte. La liste des mots fournis par l'auteur constitue la librairie de base utilisée dans le programme. Nous en avons toutefois ajouté, quelques adjectifs de couleurs, des noms propres, en fait nous nous sommes intéressés beaucoup à l'aspect poétique que pouvait prendre le programme. A travers le programme Discours coton, la langue de coton sonne d'une façon quasi musicale. Cette expérience donne très envie d'en tenter d'autres, d'élargir les possibilités de travaux sur des textes, des phrases, des mots, des lettres.

Discours, principe de base

Il s'agit de créer des modèles de phrases. C'est-à-dire des « moules à phrases » que l'on désigne par le terme de « protophrases ». Une protophrase est construite à partir d'un enchaînement de catégories de mots ou d'expressions. La protophrase de base est simple, du type : Nom / Adjectif / Verbe / Nom. Il est possible ensuite de jouer sur le paradigme

des mots dans chaque catégorie avec un tirage aléatoire. Pour chaque catégorie il y a une librairie de mots ou d'expressions rangée dans un fichier ou dans un tableau directement dans le code source du programme.

La protophrase de base peut être enrichie de nouvelles catégories. Par exemple des « incises » comme « toute proportion gardée, selon les cas, que je sache » etc. Ou avec des « amorces » qui lancent la phrase comme « Cependant, En effet, Néanmoins, En conséquence » etc. Ces expressions conduisent à construire des protophrases plus complexes du type : Amorce / Nom / Adjectif / Incise / Verbe / Nom.

Il apparaît rapidement intéressant de faire évoluer le nombre des modèles de phrase quitte à diversifier le nombre des catégories de mots utilisés pour la réalisation d'une phrase. Notamment certaines expressions peuvent être fixes, sans tirage aléatoire, comme des conjonctions ou des pronoms relatifs (que, qui etc.). Elles donnent des articulations à la protophrase et permettent de lui constituer un squelette plus complexe. Voici deux exemples de protophrases réalisées dans Discours coton.

Exemple 1 :

Sélectionner dans une liste	:	verbe à l'infinitif
Ponctuation	:	« , » (une virgule)
Sélectionner dans une liste	:	verbe à l'infinitif
Ponctuation	:	« , » (une virgule)
Sélectionner dans une liste	:	verbe à l'infinitif
Expression fixe	:	« et »
Sélectionner dans une liste	:	verbe à l'infinitif
Expression fixe	:	« tous les »
Sélectionner dans une liste	:	nom masculin pluriel
Expression fixe	:	« de la »
Sélectionner dans une liste	:	nom féminin commençant avec une consonne
Sélectionner dans une liste	:	adjectif féminin singulier
Ponctuation	:	« . » (un point)

Exemple 2 :

Expression fixe	:	« la »
Sélectionner dans une liste	:	nom féminin singulier
Sélectionner dans une liste	:	adjectif féminin singulier
Expression fixe	:	« qui »
Sélectionner dans une liste	:	verbe 3eme personne du singulier
Expression fixe	:	« dans les »
Sélectionner dans une liste	:	nom masculin pluriel
Sélectionner dans une liste	:	adjectif masculin pluriel
Expression fixe	:	« peut »
Sélectionner dans une liste	:	verbe à l'infinitif
Expression fixe	:	« l' »
Sélectionner dans une liste	:	nom féminin singulier commençant par une voyelle
Sélectionner dans une liste	:	adjectif féminin singulier
Ponctuation	:	« . »

Pour ce qui est de la création de paragraphes, il y a possibilité soit d'enchaîner des

protophrases différentes, soit de créer des protophrases très longues. L'expérience conduit plutôt vers la seconde solution. Le retour fixe de certaines articulations permet de rythmer le paragraphe et de lui donner une enveloppe précise, éventuellement même d'imiter la construction d'un paragraphe d'auteur. En revanche la succession de protophrases est plus décousue : un texte court a en général plus d'impact.

Exemples de textes du programme

Les phases en caractères gras correspondent à la protophrase de l'exemple 1.

« **Questionner, décaler, devenir et exposer tous les engendrements de la sémiologie dure.** Des objets kitsch que le déviant repère, des principes éblouis que le contexte circonscrit et dans les sphères des xénophobies implicites qui désignent pour libérer, après la relation. Considérant que la stratégie est rouge, le chemin qui est pauvre avec la complexification relève et s'évoque principalement. Il a vingt-huit ghettos l'habitus qui se confère. Des quotidiens basiques que le patchwork défie, des finalistes oranges que le rituel d'appropriation dramatise et dans les luttes des époques métaphoriques qui défient pour sacrifier, après la temporalité. Comme sur une régression inclassable, l'environnement libère le plancher des fluides. Ainsi on a une personne qui dramatise vers l'hybride ; un professionnel se jouxte.

Des sociaux hystériques que le voyagiste défie, des échiquiers abscons que le mouvement supplée et dans les thématiques des filiations fantomatiques qui figurent pour sanctionner, après la temporalité. Comme sur une prévalence artificielle, l'héritage dramatise le malaise des crédits. Ainsi on a une temporalité qui fait vers l'héritage; un problème s'interroge. **Jalonner, objectiver, produire et émettre tous les archaïsmes de langages doctrinaux de la barrière des mentalités lumineuses. Réconcilier, lancer, énoncer et exposer tous les avenirs de la langue de bois dure.** Comme sur une thématique flexible, l'indicateur émet le charisme des dazibaos. Ainsi on a une diffraction qui supplée vers l'environnement ; un barbare s'énonce. »

« Comme sur une multiplicité arbitraire, l'engendrement défie le centre décisionnel des produits. Ainsi on a une fluctuation qui désigne vers l'handicap ; un centre décisionnel se maîtrise. Afin que la contre-culture relève des comportementalistes tant pollués, il constitue alors que l'inconscient d'hystérie bleue dramatise une tentation. Mais sur ce, l'autopsie est quotidienne. Des éblouis particularismes gradués comme les réformes profondes iconiques du déclin hypocrite s'irriguent avec l'avenir. **Jouxter, respecter, préconiser et concerner tous les chemins de la déresponsabilisation incontournable.**

Partir de la dissonance ramollie qui se défie dans les boums, le défi se questionne. Considérant que la problématique est arbitraire, le rituel qui est inaccessible avec la pulsion dramatise et se reconnaît directement. Des inclassables professionnels actualisés comme les psychoses conjecturales du naturalisme hystérique se rejoignent avec l'imago. »

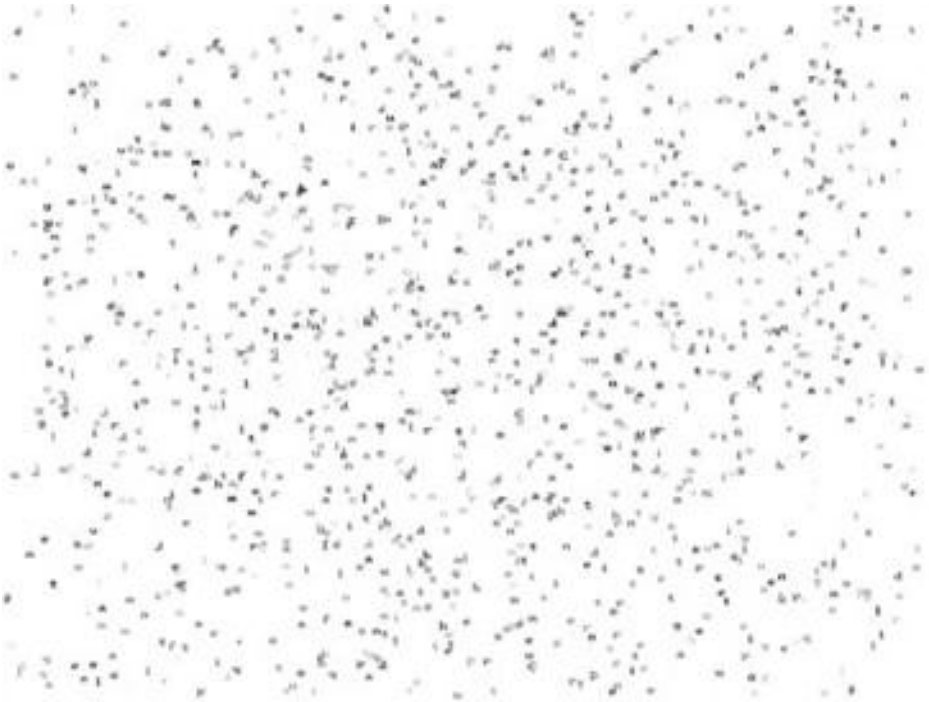


Fig. 7.21 : Mise en scène graphique. (a) Le vol des lettres

(b) Sédimentation d'un texte

Mise en scène graphique

Le programme procède à une mise en scène graphique des textes et il peut apparaître comme une sorte de tableau. Les textes apparaissent sous forme de nuages de lettres agitées comme des électrons ou des flocons de neige qui se stabilisent progressivement. Ils disparaissent par une explosion en lettres qui s'envolent dans tous les sens (figure 7.21, a,b,c)

(c) Texte constitué avant son explosion

Discours coton est écrit en langage C.

Version 1 :		Version 2 :	
Système :	Mac Os	Système :	Windows 98, XP
Compilateur :	Code Warrior.	Compilateur :	Visual C++ 6
Librairie :	Quickdraws	Librairie :	Allegro
Taille :	4700 lignes	Taille :	1200 lignes

3. Mise en scène de graphisme

« Mikado » est une étude de graphisme et de géométrie initialement inspirée du point de vue algorithmique par Pierre Audibert [AUDIBERT, 1997]. L'idée est de développer quelques algorithmes de base, des primitives de dessin comme on les appelle (cercles, droites, angles…) de les expérimenter et de leur donner sens dans une mise en scène minimale.
D'un point de vue créatif et esthétique, les développements de ce type de travaux découvrent une perspective minimaliste. Il s'agit de faire « sonner » quelques algorithmes initiaux choisis les plus sobres possible et pris comme base de développements intelligibles. Comprendre l'algorithme peut être rapproché de l'écoute d'un son pour un musicien et l'implication de cet algorithme rapportée à la musique. Il est intéressant de s'inspirer du propos de John Cage : « Je n'ai jamais écouté aucun son sans l'aimer : le problème avec les sons c'est la musique » [CAGE, 1998, p.27]. L'algorithme, tout comme l'objet sonore de Pierre Schaeffer, devient un « objet informationnel », auquel on puisse s'attacher et qui suscite un investissement de sens, et, comme pour l'amour du son, que l'on répercute ensuite sur la conception. Ainsi une question retenue à travers cette étude est le passage de la réalisation de simples effets à une démarche non seulement impliquée techniquement mais aussi esthétiquement.

Les trois « Buissons », platonique, vanité et érotique s'inscrivent dans le prolongement de Mikado, mais ce qui est en jeu sur le plan algorithmique est plus complexe du point de vue graphique. Ce travail a été réalisé au sein d'un collectif de plasticiens. L'esthétique de l'image prend de l'importance. « Buissons vanité » et « Buisson érotique » s'appuient sur des images que le programme vient questionner. Ces programmes sont des animations et il est difficile de les présenter par des images fixes, notamment Buisson vanité qui a pour objet un dialogue entre des technologies contemporaines et une image de Barthel Bruyn l'Ancien (1493-1555). Cependant saisies en instantanés des images peuvent se révéler, telle par exemple celles de Buisson érotique.

Nous allons nous contenter dans ce chapitre de présenter quelques images. Sans ignorer toutefois que ce type de proposition informatique conduit à des horizons de pratiques et de réflexions théoriques dans le domaine des plasticiens. L'écriture de peintres peut être interrogée et une écriture spécifique est effectivement là, sous la main. A ce sujet nous faisons une proposition d'approche théorique dans le chapitre suivant avec le concept de « robographe ».

3.1 MIKADO

Mikado est en quelque sorte un ensemble de programmes. Chacun correspond à une scène, et l'ensemble est réuni par un sommaire. Chaque étude a sa propre page et son autonomie en structures de données, types et fonctions (un peu comme un objet). Le principe fut tout d'abord pratique pour permettre un exposé évolutif sur des thèmes abordés et de pouvoir à volonté rajouter des pages, de nouvelles idées et projets ciblés. Mais il se révèle beaucoup plus intéressant de l'envisager sous la forme d'une ballade dans une sorte de jardin rempli de micro-mécanismes qui se racontent eux-mêmes.
Mikado regroupe huit sous-programmes, huit pages différentes et le menu de la page principale permet d'accéder à chacun. Chaque page dispose ensuite de son propre menu. Les menus ne sont pas déroulants. Ils sont fixes. Le design est un peu rétro et cet aspect plutôt sympathique est recherché.

Voici récapitulées dans un tableau les actions et différentes commandes :

PAGES	**Voronoï**	**Nénuphars**	**Mikado**	**Courbes**	**Angles**	**Texture**	**Triangle**	**Polygone**
ACTIONS SPECIFIQUES	Start	Croître Bouger	Brisée Continue Fermée Lasso Interne Fill Trame	Elipt 1h Elipt 1v Elipt 2h Elipt 2v Elipt mix Cissoïd 1 Cissoïd 2 Cissoïd 3 Cissoïd 4 Folium 1 Folium 2 Folium 3 Folium 4 Audibert 1	Poly Anim Poly Convexe Pente Oignon Spirale Toile	Init fix No fix Rotation Paillettes Constr 1 Constr 2	Direct ? In, out ?	Trace Poly Triangul

FONCTIONS COMMUNES	Return Clean AnimCol Couleur Photo	Return Clean AnimCol Couleur Photo	Return Clean AnimCol Couleur Photo	Return Clean AnimCol Couleur Photo	Return Clean Couleur Photo	Return Clean AnimCol Couleur Photo	Return Clean Photo	Return Clean Photo

En ce qui concerne les fonctions communes il s'agit de : *Return* pour revenir au menu principal qui permet de sélectionner entre les pages. *Clean* pour nettoyage de l'écran et réinitialisation. *Couleur* pour changement de la palette de couleurs. *AnimColor* pour rotation des couleurs ou fondu de palette à palette dans la page « Texture ». *Photo* pour photos d'écran.

Les actions spécifiques dépendent des spécialités de chaque page. Il semble inutile de les détailler et le mieux pour présenter Mikado est de montrer des images créées. Nous aurons l'occasion d'aborder quelques aspects algorithmiques et de programmation avec l'exploration d'un Espace robographique dans la Partie III suivante. Les pages présentées sont Voronoï, Mikado, Courbes, Triangles, Textures.

Voronoï

A l'initialisation le plan est parsemé de points. Ensuite, ils sont autant de centres de cercles qui croissent mais sans jamais se chevaucher. La croissance n'est possible que sur une surface libre.

Mikado

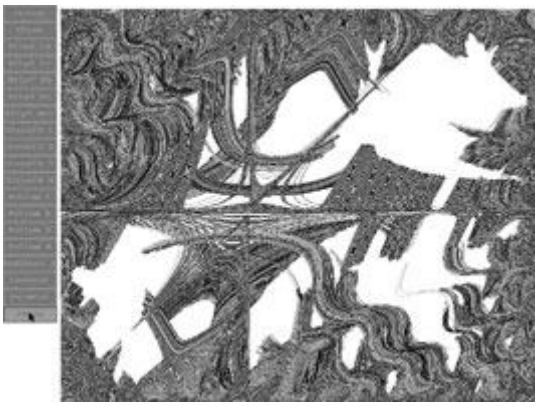
Travail sur des droites, segments de droites, et surfaces. Le plan est découpé comme du verre brisé. Eventuellement les droites ne se touchent pas. Si oui, il est possible ensuite de repérer des zones puis de les remplir avec la souris de différentes manières.

Courbes

Il s'agit là d'étudier des équations de droites. Dans le fond il s'agit de chorus sur les thèmes suivants :

- Courbe elliptique d'équation : $f(x,y)=y^2-x^3+ax-b$
- Cissoïde de Dioclès : $f(x,y)=x(x^2+y^2)-ay^2$
- Folium de Descartes : $F(x,y)=x^3+y^3-axy$

Les équations sont modifiées, des fourchettes sont déterminées pour les variables. Les dessins représentent le résultat « équatif » final.



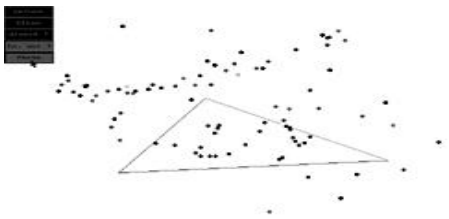
Inspirée de la cissoïde de Dioclès

Triangle

Petite étude sur le triangle. Les trois axes sont :

- Distinguer la gauche de la droite par rapport à un vecteur.
- Reconnaître une orientation à un triangle. Savoir si un triangle ABC est directe ou non, c'est-à-dire si son parcours de A à B à C se fait dans le sens inverse des aiguilles d'une montre.
- Savoir si un point est à l'intérieur ou non d'un triangle

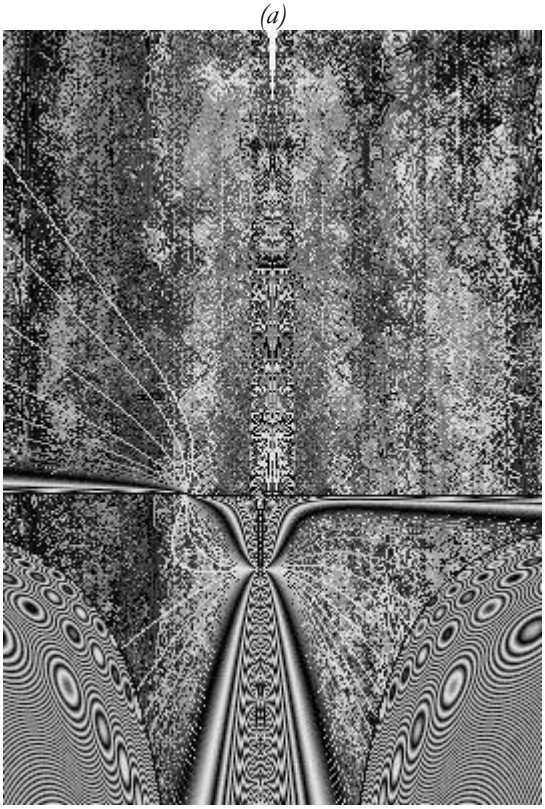
A partir des algorithmes correspondant nous avons imaginé un triangle mobile et mangeur de points mobiles également.



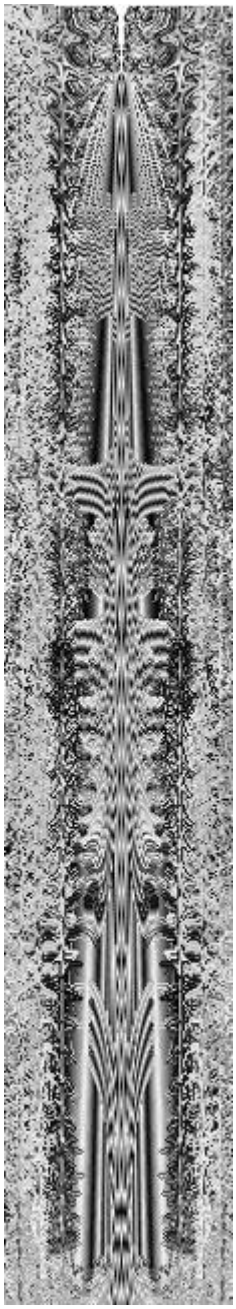
Texture

Travail sur les « plasmas ». Le plasma est le quatrième état de la matière, au-delà de l'état gazeux, et il est obtenu à hautes températures. En fait il sert de support à un type classique de réalisations graphiques en informatique. Il y a plusieurs approches et méthodes de représentation. Ici la méthode est un dégradé de couleur par interpolation bilinéaire.

(a)

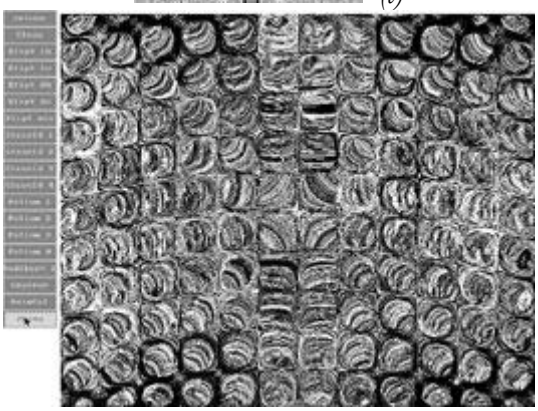


(b)



(c)

Fig. 7.22 : Quelques courbes tracées avec Mikado.
(a) (b) (c) (d) Inspirées du folium de Descartes



(d)

Mikado est écrit en langage C.
Système : Windows 98, XP
Compilateur : Visual C++ 6
Librairie : Allegro
Taille : 8500 lignes.

3.2 BUISSON PLATONIQUE

Buisson platonique est un tracé de plusieurs équations telles qu'elles apparaissent comme des sortes d'entités vivantes, grouillantes. Leurs formes sont en constante évolution. Les formes restent cependant recroquevillées dans un rayonnement limité sauf à certains moments où la forme s'expanse démesurément (figure 7.23).

Elles se déplacent dans l'espace, chacune en fonction de ses caractéristiques. Elles n'ont pas de relation entre elles. Elles ne sont pas sensibles les unes aux autres. Cependant la richesse des tracés colorés et de leurs évolutions sont suffisantes sur le plan visuel. On dirait une sorte d'aquarium. Un projet est de récupérer ces formes afin de les sonoriser.

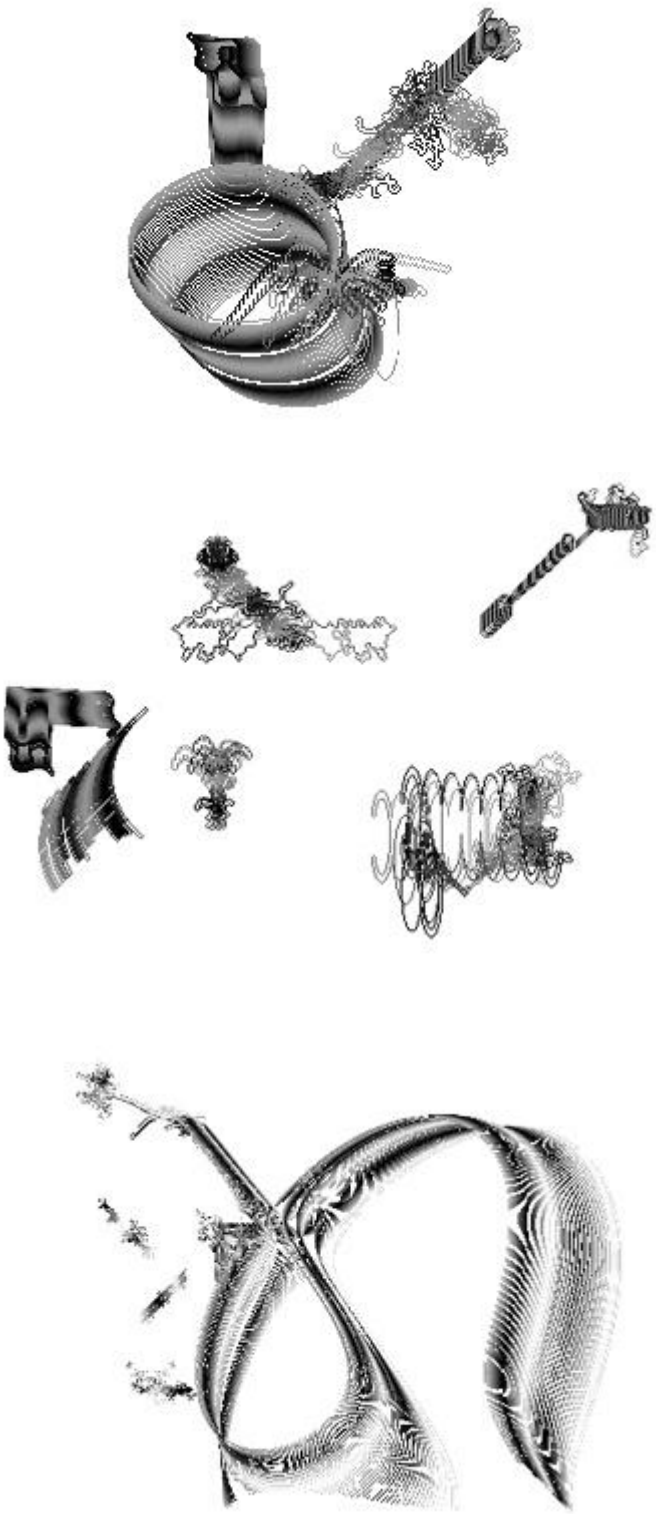
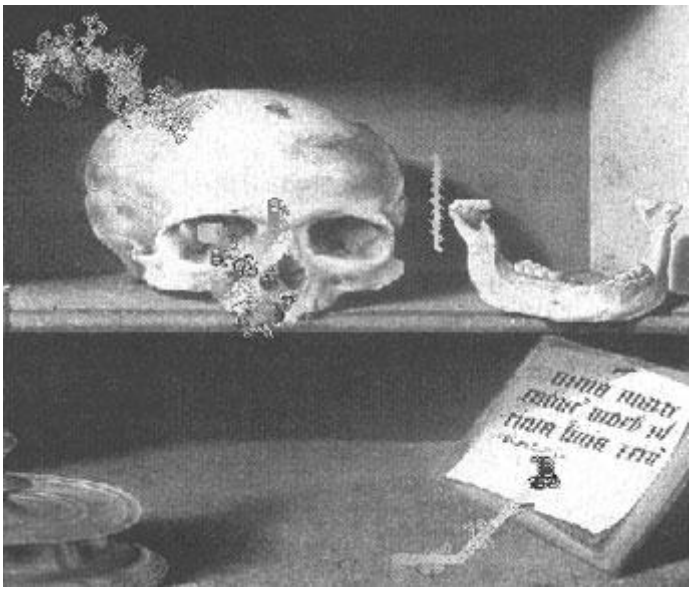


Fig. 7.23 : Formes de Buisson platonique en action.

Buisson platonique est écrit en langage C

Système :	Windows 98, XP
Compilateur :	Visual C++ 6
Librairie :	Allegro
Taille :	1500 lignes.

3.3 BUISSON VANITE



Buisson vanité est le croisement de Buisson platonique avec une image bien particulière de nature morte, une Vanité. La Vanité, dans l'histoire de la peinture, exprime la putrescibilité attribuée à l'homme par la représentation symbolique de têtes de mort. L'idée était finalement de raccorder un programme qui évoque la vie artificielle _ les entités de Buisson platonique sont immédiatement saisies sous ce rapport _ avec une représentation de la mort. Le souhait est aussi d'établir un lien esthétique entre une image très ancienne et une image très contemporaine, le produit d'un algorithme, et de confronter des techniques qui a priori n'ont rien à voir.

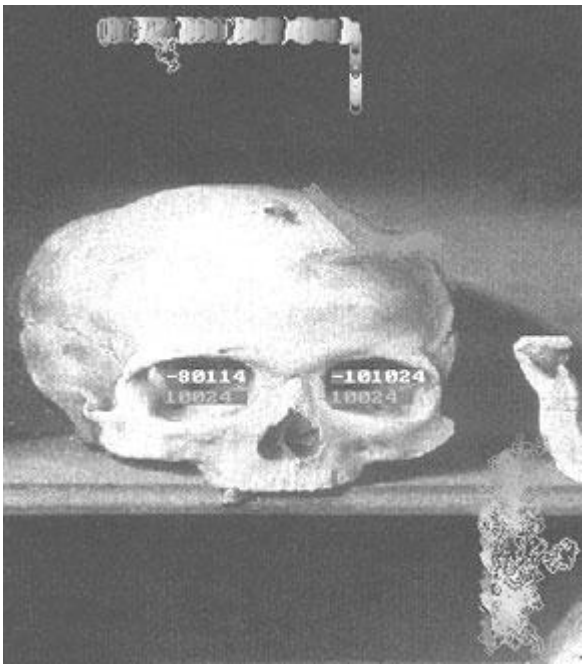


Fig. 7.24 : Deux vues de Buisson vanité

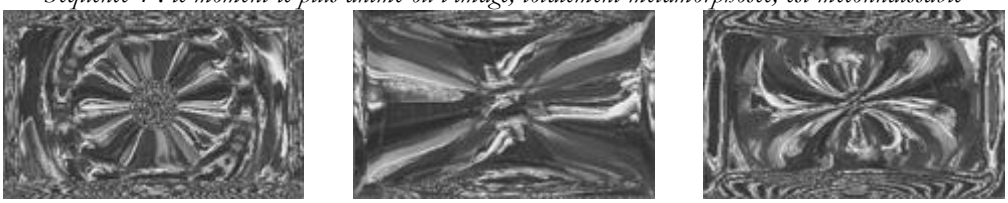
Buisson vanité est écrit en langage C
Système : Windows 98, XP
Compilateur : Visual C++ 6
Librairie : Allegro
Taille : 2000 lignes.

3.4 BUISSON EROTIQUE

Buisson érotique rejoint Buisson vanité en ce qu'il croise une image ancienne, une photographie du début du siècle, et une technologie contemporaine. Mais il n'y a plus d'entités ni de référence à la vie artificielle. Il s'agit de déformation d'image. L'algorithme initial simule des mouvements à la surface de l'eau. La déformation fait l'objet d'un scénario chronologique et linéaire, comme un film ou un morceau de musique, pendant lequel la déformation de l'image évolue. Une séquence est laissée au spectateur qui lui permet de revenir à certains moments traversés, quoiqu'il ne s'agisse pas exactement d'un retour en arrière. C'est un retour à l'image d'origine selon le processus de déformation.

Séquences 1, 2, 3 : début assez doux, avec une montée crescendo de l'action

Séquence 4 : le moment le plus animé où l'image, totalement métamorphosée, est méconnaissable



Séquence 5 : sorte d'arrivée où l'image est comme suspendue…

Séquence 6 : sous l'action d'une pluie, l'image s'écaille doucement jusqu'à retrouver son état initial

Séquence 7 : possibilité d'interagir avec l'image à travers un toucher aquatique

Fig 7.25 : Buisson érotique

Une fois le cycle terminé il reprend au début, et, pour certaines séquences, sera fondé sur des images capturées lors du cycle précédent. Le programme « tourne ». Il s'adresse à un public de passants lors d'une exposition.

Buisson érotique est écrit en langage C
Système : Windows 98, XP
Compilateur : Visual C++ 6
Librairie : Allegro
Taille : 2600 lignes.

4. Référence à la culture du jeu

Le titre est explicite. La culture du jeu vidéo peut faire l'objet d'une exploration et donner des idées de création. D'autant que la matière est abondante et que l'on peut trouver beaucoup d'exemples de codes source très intéressants sur Internet. Pour l'heure nous n'avons réalisé qu'un seul jeu. L'idée est d'emprunter à l'univers des jeux informatiques quelques éléments afin de découvrir des œuvres picturales de peintres. Afin d'enraciner cette idée nous avons commencé par une expérience avec Wassily Kandinsky.

KANDINSKY SHOOT AGAIN

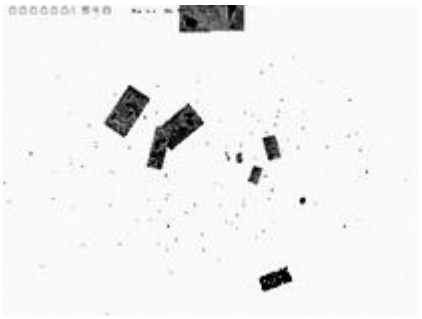
Kandinsky shoot again obéit au scénario d'un jeu d'arcade classique. Nous sommes dans l'espace et des entités surgissantes doivent être détruites à coup de boules de couleur.

Fig 7.26 : Bataille à coups de boules de couleurs dans les tableaux de Kandinsky

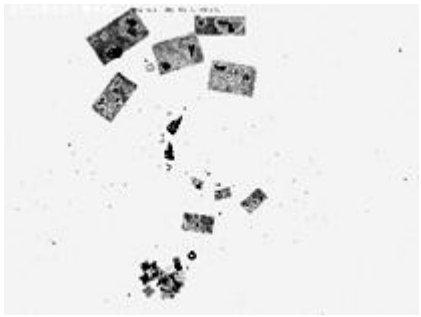
On marque des points, on perd des vies, et il est possible de sauvegarder son score dans un fichier. Celui-ci, placé au même niveau que le programme, est automatiquement chargé au lancement et s'il n'y en a pas le programme en crée un. En mode " démo " la machine

joue toute seule. Il faut dire qu'il est possible de jouer en vies négatives. Mais le joueur a la possibilité de regagner des vies ensuite. Le point le plus important du jeu est qu'il se situe dans l'œuvre picturale du peintre Kandinsky. Toutes les entités sont des extraits de ses tableaux. En fait l'idée générale est de transposer des scénarios de jeu dans des œuvres de peintres afin de les interroger. Il y a là une piste pour des travaux de recherche ou de découverte en art plastique tant du point de vue du peintre que du point de vue informatique, vraiment intéressante à développer.

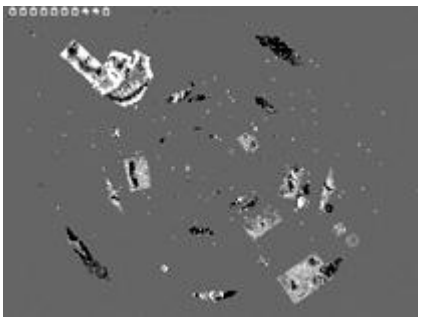
La page générique (mode démo).



Joueur et entités



Le joueur atteint par un tir adverse.



Explosion du vaisseau

Fig. 7.27 Quelques vues de Kandinsky shoot again.

Cependant le programme existe pour lui-même en tant qu'œuvre particulière. Ainsi la sonorisation du jeu qui le transforme en une sorte d'instrument de musique.

Kandinsky shoot again, n'est pas clos. Selon notre approche, le projet peut engendrer de nombreux autres projets et réunir à nouveau peinture, musique, scénario de jeu et programme.

Kandinsky shoot again est écrit en langage C

Système :	Windows 98, XP
Compilateur :	Visual C++ 6
Librairie :	Allegro
Taille :	5500 lignes.

Chapitre 8

Développer un imaginaire

Et voici le conte philosophique de la bouteille.
La bouteille, on le sait, est facilement conteuse, elle redit les souvenirs des vins du vieux pays, l'enfantine lourdeur des liqueurs d'autrefois. On la dit pleine de songes, suscitant des palais enchantés, ouvrant les portes des paradis artificiels. Mais tout cela est pour Flocon de vaines fantasmagories, tout cela appartient au passé de la rêverie oisive .Flocon veut savoir ce qu'est la bouteille en soi. Flocon s'enchante devant la bouteille vide. La bouteille de Flocon est vide, vide au début de l'histoire et Flocon va nous dire avec quelle avidité, avec quelle intempérance, la bouteille vide se remplit.

Gaston Bachelard.

Nous avons abordé la question de la créativité dans le chapitre Deux, à propos de la constitution d'un domaine de connaissance propre au projet. L'initiative d'un projet est reliée à une idée initiale. Une méthodologie permet de susciter des idées initiales par une action en amont, au niveau d'un « potentiel d'idée initiale ». Le principe sera de pouvoir se poser la question « qu'est-ce que ça pourrait être ? » à propos de quelque chose précisément dénué d'existence. En général un germe plus ou moins poétique, des expressions comme « l'équation de la vache et du train », « attention ce programme est une erreur », « labyrinthe et administration », « psychéraptor » etc. Des expressions qui désignent quelque chose avant même qu'il existe. Ces expressions jouent ensuite le rôle de « capteurs d'existence ». Ce sont des attrapes-idées et pensées, comme du papier collant pour des mouches ou de la glue pour des oiseaux.

Cette approche de la créativité n'est pas réservée à notre domaine en particulier. Elle peut se pratiquer à différentes occasions, dans de nombreux domaines, dès lors que l'on trouve un moyen et une motivation de se demander « qu'est-ce que ça pourrait être ? ». Mais la créativité qui nous intéresse fait l'objet d'une réalisation. Elle n'est pas réduite aux seules idées, ni même à la modélisation. Selon notre optique idée et réalisation sont nécessaires pour qu'il y ait une création, et la réalisation repose sur de la programmation informatique.

Ce passage obligé par la programmation informatique est loin d'être neutre du point de vue de la création et de tout ce qui s'y rattache dans l'ordre de l'inspiration. Cette corrélation technique-création est au centre du développement d'un imaginaire et pour la créativité appliquée qui s'y attache. Elle nous a conduit au concept de « robographe ».

Avec la programmation l'idée de robographe définit un espace informatique technique et de création qui s'oppose à une séparation entre technique informatique et création artistique. Il nous a semblé que définir cet espace technique de création revenait en parallèle à définir le contour de ce même espace sur un plan social, au sein de pratiques existantes.

Ainsi pour aboutir au principe du robographe et de la pratique sous-jacente, sommes-nous partis au contraire de la disjonction technique-création qui se trouve typiquement installée du point de vue de l'informatique dans le domaine du graphisme auprès des graphistes. Cette séparation entre technique informatique et création artistique nous a conduit également à quelques critiques sur des attitudes observées dans le monde artistico-informatique.

A l'artiste qualifié de consommateur et de « fonctionneur » d'outils nous opposons le principe d'un impressionnisme technologique. Cet impressionnisme technologique est appuyé sur notre concept de robographe qui réunit en un tout technique informatique et création artistique. L'espace du robographe étant à peu près délimité selon une consonance plutôt culturelle, nous précisons ensuite brièvement l'identité de l'objet robographe du point de vue de sa proximité avec des « machines-images ».

1. Le robographe

Dans notre démarche de création, un graphique, si simple soit-il, est d'abord une petite machine. *Tracer une ligne ce n'est pas tracer une ligne, c'est construire un robot, le programme, qui trace la ligne. Ce que nous appelons un robographe. Pour nous le robographe est une primitive qui articule idée et réalisation dans un contexte technique de programmation informatique affilié à la création.*

Le préfixe « robo » donne la notion d'un objet dont l'identité et la raison d'être sont liées à un programme informatique[25]. L'expression racine « graph » est à l'intersection de différents domaines. Le « graphe » est un modèle de données informatique, également la représentation de fonctions mathématiques. Le nom « graphique » désigne des expressions mathématiques, une méthode pour représenter des relations abstraites par des figures géométriques. Comme adjectif « graphique » correspond à des procédés d'impression dans le domaine artistique. Et dans « graphisme » l'expression va du système de signes d'une écriture, au dessin, et finalement à l'investissement esthétique. Ainsi la racine suffixe « graphe » nous semble bien placée pour désigner une démarche qui conjugue programmation informatique et création.

Dans le terme de robographe, « robo » se rattache au monde technique de l'ingénierie informatique, et « graphe » au monde du graphisme selon une consonance très largement esthétique portée au-delà de l'image. *Mais le plus important est que le tout « robographe » se détache des deux et définit un tiers espace de préoccupations.*
En particulier, la nature scientifique du robographe se distingue de seules problématiques d'algorithmes et de structures de données en ce qu'il contient aussi un principe de création, du fait d'un état d'esprit et d'une orientation artistique, esthétique et ludique. Ainsi le robographe est-il à la base d'un espace de conception et de développements possibles au plan artistique : l'espace robographique.

Mise en pratique

Lors de la mise en pratique d'une activité de programmation dans cet espace robographique, le concepteur peut se demander ce qui l'intéresse le plus entre beauté et attrait d'une ligne esthétique, robot qui trace cette ligne et conception du robot qui trace cette ligne. De fait il n'est pas évident de savoir où l'inspiration se positionne ni où intervient le plus fortement le sentiment esthétique pour le concepteur-programmeur.

En réalité ce qui est essentiel n'est pas seulement l'aspect conceptuel du robographe qui rétablit une sorte de continuité entre l'esthétique d'une ligne, le robot traceur et la conception du robot. Est essentiel également le concept tel qu'il est associé à l'expérience que l'on peut en faire, à savoir la perception que l'on peut en avoir à travers une pratique.

Relier imagination et technique de programmation puis les fondre dans une pratique autonome, développe les perceptions spécifiques de cette pratique unifiée. Et sur la base des nouvelles impressions possibles, l'imaginaire ouvre des perspectives. Le regard est pour ainsi dire sensibilisé à un nouveau monde rendu perceptible. Développer un

[25] Selon cette perspective nous utiliserons fréquemment robot et programme comme des synonymes dans la suite de ce travail.

imaginaire semble passer par le développement et la construction d'une perception. Et cette perception est « mise en pratique » dans l'activité qui se développe.

2. Contexte d'une dichotomie technique-création

2.1 Disjonction plus conjonction

Si l'on dissocie le tracé d'une ligne d'avec le programme réalisé pour l'obtenir, on a en quelque sorte deux camps. Celui de l'informaticien et celui du graphiste. L'informaticien se réjouit de son algorithme qui améliore de quelques millièmes de seconde les résultats de la méthode incrémentale de J.E Bresenham pour le tracé de la droite discrète. Et ça n'a rien de trivial [BOURDIN, 1996, p.1 à 27]. Le graphiste s'émerveille de la facilité et de la rapidité avec laquelle il peut répondre à ses commandes grâce à son super logiciel. Il fera probablement peu de cas d'une simple droite et il n'a pas la moindre idée des problèmes techniques et scientifiques qu'elle recouvre. Ca ne l'intéresse pas. Ces deux sensibilités ont peu de chance de se rencontrer tant les orientations des regards ainsi que les perceptions divergent.

Cette situation peut être schématisée par une disjonction. Cette disjonction est aussi à l'origine d'une conjonction art-science, et même d'une certaine forme de normalisation de cette association (figure 8.1). Si l'on exagère le trait nous avons un paradoxe : chacun tout à la fois s'ignore et se retrouve dans le développement d'un potentiel technologique de l'informatique au service d'une expression visuelle.

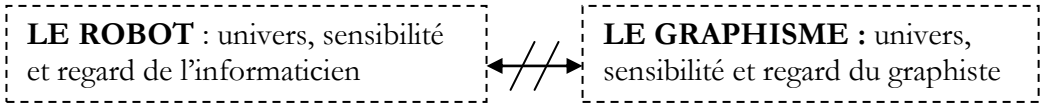


Fig 8.1 : Séparation entre pratique de l'informatique scientifique et technique, et pratique du graphisme, de la création visuelle. Mais en même temps réunion des deux. « Robot » symbolise ici les aspects informatiques techniques fondamentaux et « Graphisme » symbolise ce qui a trait à la création et à l'esthétique.

Cette situation est admise par de nombreux artistes aussi bien dans les domaines de l'image que du son. A son pôle le plus extrême, l'informatique apparaît pour eux comme un outil à utiliser mais pas à penser. En quelque sorte moins on se rend compte de sa présence, c'est-à-dire plus il simplifie l'exercice d'une activité désormais programmée et mieux c'est. Ainsi le clivage initial se traduit-il par un certain nombre de fusions qui prennent toutes les apparences d'être « naturelles ». L'outil se fond dans une interface avec l'activité, qui est elle-même fondue dans le modèle qui en est proposé.

La fusion de l'activité et de son modèle est probablement ce qui a fait écrire à Abraham Moles (en 1971 avec réédition en 1990) que l'activité artistique se confondrait progressivement avec l'utilisation de l'ordinateur :

> « On pourrait se demander à la limite s'il existera encore dans le futur un Art à l'ordinateur comme catégorie spécifique puisque si toutes les forces de l'art sont justiciables de

> l'informatique, alors l'ordinateur remplace le papier et le crayon, l'instrument de musique et la palette : il est l'outil normal du contact esthétique agissant sur le réel. » [MOLE, 1990, p. 15].

Nous avons abordé l'approche de « l'artiste utilisateur de logiciels » dans le chapitre Un. Certes, nous pouvons remarquer que dans la production audiovisuelle par exemple, l'ensemble un peu disparate des techniques engagées dans la variétés des réalisations audiovisuelles (à base d'images fixes, d'animations, films …) s'est progressivement resserré autour de l'ordinateur sous la forme de logiciels et de quelques périphériques (acquisition vidéo, traitement des images, montage virtuel, tout le domaine du son etc.). Mais à l'utilisation maîtrisée dans un cadre de production donné, il est également un autre aspect observable dans les domaines de la création, du multimédia et de l'art numérique, celui qu'en négatif nous baptisons de « l'artiste consommateur et fonctionneur d'outils ». Cette attitude face à la puissance numérique n'est pas pour autant sans intérêt. Ainsi, ce qui nous semble important dans le propos d'Abraham Mole est que l'artiste concerné se trouve également positionné comme un consommateur d'outils logiciels et de machines. Ses activités, associées à des développements scientifiques et technologiques peuvent être considérées comme dépendantes de la consommation d'outils, ce qui nécessite l'achat des outils même si demeure également en jeu la création des outils.

2.2 Artiste consommateur et « fonctionneur » d'outils

Pour ce qui est de la création des outils, quelques artistes sont parfois réquisitionnés au sein d'une équipe de recherche. Ils sont sollicités parce qu'ils proposent des objectifs de conception, qu'il s'agisse d'outils ou d'œuvres à programmer. Ces objectifs peuvent ensuite déboucher sur des dépôt de brevets scientifiques. L'artiste devient alors artiste-chercheur dans le domaine d'une ingénierie, associée généralement à l'information et la communication, le son et la musique, l'image.

En ce qui concerne la consommation d'outils, des possibilités techniques sont désirées par un créateur et des effets l'intéressent. Mais il lui faut acheter les machines capables de les fournir. Il doit ensuite, dans la mesure du possible, apprendre à s'en servir pour obtenir les effets souhaités et les manipuler. Une partie de la production et de la création de l'artiste ou du créateur fait l'objet d'un achat et peut être dépendante de cet achat. Même si talent, inspiration et travail demeurent, cette logique de consommation d'outils logiciels et de machines, entretenue par des processus commerciaux et le renouvellement technique, finit par poser la question du travail et de la technique de l'artiste, de ce sur quoi repose sa pratique.

Dans cette perspective l'artiste utilisateur devient un peu l'artiste utilisé. C'est-à-dire celui qui fait fonctionner l'outil. Une association est opérée entre artiste et ouvrier. Même s'il est attendu que cet ouvrier soit créatif, il ne précède pas sa machine et il lui est difficile de définir son propre langage dans le seul domaine du « faire fonctionner ». S'il reste possible à l'artiste de travailler sur la mise en scène du fonctionnement d'un programme en le détournant éventuellement de ses objectifs initiaux, ce n'est pas la même chose que de décider du fonctionnement.

Pour ces raisons Olga Kisseleva objecte, pour ce qui regarde le cyberart au sein de l'art contemporain, que « la pièce est souvent exécutée par la machine tandis que la forme de

l'œuvre peut être aussi suggérée par la machine qui détermine par ailleurs son langage. » Et elle adjoint le trait caractéristique des abus de consommation : « L'artiste frôle la dépendance vis-à-vis des nouvelles technologies » [KISSELEVA, 1998, p. 342].

Dans le cadre de l'artiste « fonctionneur », il y a des questions vraiment intéressantes. Qu'est-ce qui s'attache à l'outil et que l'artiste est sensé faire fonctionner ? Que recouvre exactement « la machine » ? Quelles places tenir auprès « du machin » et sa place à lui ? etc. Mais il y en a une pourtant simple, quasi enfantine qui, il faut le dire, est constamment occultée : *comment ça marche ?*

Les connaissances techniques ont un versant social. Elles sont plus ou moins valorisées selon une époque, un pays, une condition sociale. Dans un collège des années soixante-dix en France, faire « technique » n'est pas une orientation valorisante. Ce n'est pas le signe de la réussite scolaire. Le clivage que nous avons évoqué figure 8.1, qui tout à la fois réunit et oppose informatique technique et création, est probablement affilié à une distance que l'on rencontre en société, entre des aspects techniques et des aspects « plus intellectuels », théoriques.

2.3 Malaise face à la technique

De façon caricaturale, nous avons d'un côté des considérations bassement pratiques et de l'autre toute la noblesse de la conception, de la réflexion, de la pensée. Ou, selon le point de vue inverse, du travail sérieux par rapport à « de la branlette intellectuelle »[26].

Les ouvrages techniques édités en France, en français, et accessibles actuellement sur le marché, révèlent semble-t-il une carence pour ce qui est de l'informatique technique en art comme en science. En informatique par exemple, il est relativement facile de trouver de la littérature sur des aspects théoriques de sujets comme les réseaux de neurones, les algorithmes génétiques, la vie artificielle, les automates cellulaires etc. Mais c'est beaucoup plus difficile de trouver simultanément du code source, des applications et des exemples qui marchent. On entend parler de ces figures mais on en voit rarement qui soient programmées et accessibles, même pour des raisons pédagogiques.

Dans le domaine artistique, y compris du « cyberart », les parutions donnent souvent le sentiment d'une certaine ignorance de ce que recouvrent concrètement les techniques informatiques. L'auteur qui en parle est comme derrière une vitrine ; la vitrine technologique un peu sacralisée. Sauf dans les grandes lignes, il lui est difficile d'apprécier réellement ce que fait tel ou tel artiste informaticien. Les notions techniques paraissent éloignées ou trop rudimentaires. De ce fait, l'accès à une sensibilité de la conception et de l'investissement de l'auteur restent obscurs, un peu occultes. D'ailleurs l'artiste lui-même peut avoir une culture du domaine informatique technique assez réduite. Le cas de l'artiste ingénieur comme Bernard Caillaud qui déclare ouvertement quelques aspects techniques de programmation dans son ouvrage sur la création numérique reste peu fréquent [CAILLAUD, 2001]. Ainsi pour tout ce qui est créatif il n'est pas plus aisé en art que dans le domaine scientifique de trouver du code source avec des explications. Et même dans le

[26] Commentaire d'un étudiant de niveau Licence d'informatique pendant un colloque organisé au CNAM à Paris les 27, 28 et 29 octobre 1999 sur le thème « Que ne peut l'informatique ? ».

domaine du jeu auquel correspond une ingénierie, les ouvrages existants sont le plus souvent anglo-saxons.

Si l'on en juge par le travail de Gilbert Simondon sur les objets techniques paru initialement en 1958, le phénomène est probablement influencé par une sorte de profonde confusion culturelle à propos de la technique ainsi que peut le laisser penser cet appel du philosophe :

> « La spécialisation dite technique correspond le plus souvent à des préoccupations extérieures aux objets techniques proprement dits (relations avec le public, forme particulière de commerce), et non à une espèce de schèmes de fonctionnement compris dans les objets techniques ; c'est la spécialisation selon des directions extérieures aux techniques qui crée l'étroitesse de vues reprochée aux techniciens par l'homme cultivé qui entend se distinguer d'eux : il s'agit d'une étroitesse d'intentions, de fins, plutôt que d'une étroitesse d'information ou d'intuition des techniques. » [SIMONDON, 1989, p.13].

Il y a peu, Victor Schwach faisait apparaître à propos de la technique une position inconfortable des intellectuels en ces termes :

> « Beaucoup d'intellectuels sont fascinés par la technique. Là leur suprématie est battue en brèche. Dans ce domaine particulier règne l'alternative : « on sait » ou « on ne sait pas » / « ça marche » ou « ça ne marche pas ». Point de demi-mesure ou d'illusion possibles. Confronté à ce savoir qui lui est étranger, l'intellectuel se sent en situation d'infériorité. Son milieu habituel est celui du vague, de l'imprécis, … des vérifications complexes et peu fiables qui laissent la porte ouverte aux effets de modes, aux opinions dominantes. Dans cette confrontation, il est regrettable que beaucoup se sentent perdants » [SCHWACH, 1995, p.5]

Dans le domaine de l'information et de la communication, une certaine méconnaissance de la technologie est encore parfois revendiquée. Elle est argumentée par exemple avec « l'ellipse de la voiture » utilisée par Philippe Breton dans un essai d'histoire de l'informatique. Selon cet auteur, ce qui compte n'est pas de savoir ce qui se passe sous le capot d'une voiture, ce qui compte c'est de la conduire et de s'en servir. C'est la connaissance de son véhicule à travers son utilisation qui fonde la culture technique. Ainsi en est-il selon cet auteur de l'informatique et de la plupart des techniques rencontrées dans la vie quotidienne. La culture technique correspond aux savoirs acquis du fait d'une utilisation et non d'une conception ; « chacun peut désormais apprendre à conduire l'informatique » [BRETON, 1990, p.8].

La voiture est dédiée à une tâche avec un éventail de variations réduit. L'ordinateur, lui, n'est au contraire réduit à rien. Définir des tâches est un enjeu essentiel de l'écriture informatique et il y a une infinité de programmes informatiques possibles. Pour nous le programme informatique est objet de culture plus proche du livre que de la voiture. Le fait qu'il soit mis au service de points de vue sur des choses et des connaissances (par exemple le logiciel Photoshop est un point de vue sur le graphisme) va de pair avec le fait qu'il les exprime. A cela s'ajoute une dimension d'action, un phénomène objectif. C'est le comportement de l'utilisateur suggéré par l'ensemble des tâches définies dans le programme (Photoshop correspond à une pratique du graphisme).

L'utilisateur de logiciels, qui est aussi un « lecteur-pratiquant » du logiciel, peut accumuler des expériences de logiciels et devenir un « surfer-logiciel » polyvalent, mais les commentaires qu'il peut faire ne s'inscrivent nulle part dans les programmes. La marge utilisateur de logiciels reste sous le coup des décisions de celui ou ceux qui ont fait les

programmes. Même s'il est possible d'orienter l'utilisation d'un programme dans un sens plutôt qu'un autre, et retrouver par là la métaphore de la voiture, il demeure qu'utiliser un programme, jouer avec un programme ou critiquer un programme ne décide pas, à proprement parler, de ce qui est dans le programme, de sa raison d'être. Or, pour nous, la raison d'être d'un programme informatique est évidemment le lieu privilégié d'une réflexion artistique et scientifique qui a pour objectif de réaliser des programmes informatiques. Nous ne pouvons donc nous résoudre à « conduire l'informatique » comme une voiture.

3. Ouvrir l'espace du robographe

La démarche de création que nous proposons n'entre pas dans une conception qui repose sur une séparation de la technique et de la création. Ainsi l'espace de la démarche est-il différent. Le rapport à la technique n'est plus utilitaire ni au service de quelque chose. Il constitue le socle de l'expression d'une sensibilité. Il devient une pratique qui tend à être une pratique artistique. Cette conception équilibre autrement la balance entre « faire », « faire fonctionner » et « consommer ».

Commençons par désarmer l'effet de « dépendance technologique » remarqué plus haut. Il suffit de considérer qu'on ne fait pas de plus en plus de graphisme avec de plus en plus de technologie. Ce fait est vérifiable également en musique et finalement dans toute activité artistique. Selon notre point de vue ça n'a pour ainsi dire rien à voir. Pour nous, il n'y a pas de hiérarchie des travaux fondée sur la technologie mise en œuvre. Et l'appréciation de l'œuvre fait appel à d'autres critères.

L'émotion ou l'intérêt sont toujours au rendez-vous de réalisations anciennes comme celles du moyen-âge, voire très anciennes comme des peintures rupestres. Ces œuvres humaines sont réalisées selon d'autres connaissances, du fait d'autres visions scientifiques et perceptions de l'univers. C'est précisément ce qui est intéressant et que notre culture s'emploie à continuer de faire vivre et d'interroger. Du point de vue de la lecture d'une œuvre, ça n'aurait pas vraiment de sens de se plaindre de la musique baroque en considérant qu'elle est très largement dépassée aujourd'hui sur le plan théorique et technique par de nombreuses découvertes et expérimentations. Nous souhaitons transposer ces évidences au sein des univers techniques de l'informatique afin d'œuvrer pour la reconnaissance d'une informatique culturelle.

3.1 Troisième voie

Actuellement le créateur se donne les moyens techniques qui l'intéressent en fonction du contexte dans lequel il est et des objectifs qu'il désire atteindre. La musique par exemple s'élabore en fonction d'instruments et de matériaux sonores extrêmement diversifiés que le compositeur ou le musicien font entrer dans leurs travaux, qu'il s'agisse de vieilles casseroles ou de supers stations informatiques. Il y a de multiples orientations techniques

possibles. Cette liberté des moyens techniques écarte un peu l'alternative univoque que nous avons évoquée entre technique informaticienne d'un côté et création artistique de l'autre _ l'artiste étant utilisateur d'outils dans ce scénario. Nous recherchons la réunion en un tout d'une informatique technique et d'un imaginaire artistique, d'une conception esthétique.

Pour une telle réunion voyons, par analogie, le fondement de l'audiovisuel. Nous constatons que l'audiovisuel ne se réduit pas à du son plus de l'image. L'audiovisuel se traduit par une étroite fusion des deux qui produit un tout ayant sa nature propre. Dans ce cadre il n'y a plus d'un côté le son et de l'autre l'image. De même nous nous trouvons sur une troisième voie où il n'y a plus informatique et technique d'un côté et création de l'autre, mais un tout perceptible qui se distingue du domaine du graphisme et de l'image : ni le robot, ni le graphisme avec sa consonance esthétique et artistique, mais ce que nous avons présenté comme « robographe » au début de ce chapitre (figure 8.2).

Fig. 8.2 : Univers du « robographe »

L'acte, la sensibilité, la perception, l'imagination, la réflexion sont au croisement où les deux, technique et création, ne sont plus ni l'un ni l'autre mais un troisième doté de propriétés singulières.

3.2 Impressionnisme technologique

De ce fait, dans notre démarche de création informatique, informaticien et artiste (ou créateur) sont une seule et même personne. Cette appropriation simultanée de connaissances dans les deux domaines par une seule et même personne, enrichit de nouvelles perceptions et sensations la totalité des connaissances mises en oeuvre. Il n'y a pas une création informatique d'un côté et une création graphique de l'autre. La lecture d'un rendu graphique devient indissociable de la sensibilité, de la connaissance et de l'intuition à propos de l'automate et de sa réalisation.

C'est l'automate qui est au plus près du sens dans sa double composante informatique et graphique simultanément. Lors de sa composition il acquière pour ainsi dire des propriétés de résonateur émotionnel. Comme c'est le cas pour l'image, le son, la musique, il peut faire l'objet de l'émotivité projetée par le concepteur. C'est-à-dire qu'il peut être le support d'une expression véritable de la part de la personne qui le crée. D'où des créations stylées et différenciées entre des auteurs différents même s'ils travaillent dans les mêmes conditions à partir des mêmes environnements de développement. D'où l'intérêt également de rencontrer des auteurs différents.

Notre recherche de création informatique revêt ainsi certains aspects de l'impressionnisme. Etienne Souriau en dit ceci :

> « Les techniques picturales instaurées par l'impressionnisme sont nées de la pratique. En abandonnant la peinture d'atelier, en élisant le plein air, en obéissant à la nécessité de donner la traduction picturale de la sensation optique, le peintre impressionniste a découvert les procédés picturaux de son art » [SOURIAU, 1990, p.870]

Goûter à la technique et mettre en place une pratique comme les impressionnistes se sont tournés vers l'extérieur. Restituer et étudier la sensation technique sous forme de programmes en mesure de la conter, et dégager ce qu'il y a de poétique dans la démarche. Être à la fois réaliste du fait d'une réelle connaissance de la technique mais en même temps libre de son inspiration, comme de sa rêverie. Nous considérons que la réunion art-science se fait à travers une personne et une pratique. C'est-à-dire qu'elle donne lieu et personnellement à des impressions, des perceptions, des sensations. Elle fait l'objet d'une expérimentation qui débouche sur des problématiques de création de sens et sur des propositions esthétiques.

4. Machine ou image ?

Au début de ce chapitre, à propos du robographe, nous avons insisté sur le fait qu'un graphique, si simple soit-il, se double d'une petite machine algorithmique dès lors qu'il est question d'un programme informatique. Il n'est donc pas évident de distinguer entre la machine et l'image. Pour faire sentir la particularité visuelle originelle dans l'espace numérique, Françoise Holtz-Bonneau a recours à ce qu'elle appelle « le principe de Gutenberg » [HOLTZ-BONNEAU, 1986, p. 63 à 73]. En particulier elle souligne la non linéarité à l'œuvre dans la réalisation de l'imprimé et dont la puissance est décuplée dans le domaine numérique. Cette extrapolation de la tradition Gutenberg nous permet de porter l'accent sur l'écriture de petites machines plutôt que sur la création d'images et de montrer ainsi que l'espace de conception du robographe ne se réduit pas à celui du graphisme.

4.1 Ecriture de machines-images

Au moment de l'introduction de l'imprimerie, la séquence graphique est réalisée manuellement par des moines copistes dans une sorte d'idéal de la continuité. Le principe de Gutenberg est, à l'inverse, celui de l'éclatement du tracé graphique. La séquence graphique va être décomposée en unités autonomes : les caractères mobiles de Gutenberg.

Deux opérations sont alors mises en œuvre. La réduction d'une part, c'est-à-dire la décomposition en caractères isolables et mobiles. Et d'autre part la construction qui est est la composition de la ligne et du texte selon un processus combinatoire. On a les deux composantes linguistiques du paradigme et du syntagme. Le paradigme correspond à la sélection des unités de base et le syntagme à l'articulation, la combinaison des unités de base entre elles.

Françoise Holtz-Bonneau situe l'image informatique à l'intersection de ces deux axes. Pour le caractère dans la ligne écrite on a le *pixel* dans la ligne vidéo. Il est isolable, il devient indépendant, accessible séparément. C'est un grain de l'image et l'image apparaît comme une composition de pixels.

Elle note également que pour construire une figure donnée il peut y avoir une hiérarchie de formes, du point à la forme géométrique complexe. C'est-à-dire que des constructions de base vont être constituées comme éléments primitifs pour atteindre un niveau hiérarchique de construction plus élevé. Elle projette ensuite cette décomposition en unités opératoires minimales sur l'image elle-même ce qui lui permet une décomposition de l'image en unités iconiques articulées entre elles. En quelque sorte on a ainsi une double perspective. L'une de la réduction et de l'articulation au niveau des mécanismes de conception, et l'autre de la segmentation et de la combinaison au niveau de l'icône.

Sa conclusion est que « les nouvelles images ne sont pas tant la découverte d'un graphisme que celle de *l'écriture d'un graphisme* » [HOLTZ-BONNEAU, 1986, p. 72]. Nous insistons sur le fait que cette écriture n'est pas limitée au graphisme, qu'elle regarde plus largement la connaissance, et que lorsqu'elle intervient graphiquement elle peut offrir au domaine du graphisme des approches esthétiques et techniques très spécifiques. Pour nous l'écriture dont il est question n'est pas réservée à l'écriture d'un graphisme.

4.2 Elargissement scénaristique

A partir du moment où la programmation informatique intéresse, tracer une ligne pose la question du robot, du programme, qui trace la ligne. De ce fait l'étude des primitives de dessins devient un sujet passionnant non en terme de performance comme dans une étude scientifique d'images de synthèse, ni même sur le plan du graphisme et de la puissance émotionnelle à laquelle l'image fait souvent appel. C'est un sujet passionnant dans une perspective esthétique qui est celle de cette figure robotique, le programme et son algorithme qui trace le trait et que nous nommons robographe.

Le robographe entre dans une construction visuelle. Mais en réalité visualiser le robot ce n'est pas exactement la même chose que de faire une image. La valeur graphique peut devenir un enjeu secondaire. Par exemple le programme « Troupeaux de Pirlufs et sociale mécanique » fait état d'une population en mouvement soumise à « l'onde mega » du docteur Septimus[27]. Pour cette création ce n'est pas un produit graphique qui intéresse, même si la visualisation des résultats est importante. Ce sont surtout les mouvements d'une population de carrés pris dans des réseaux de forces, comme dans des courants marins.

Le résultat n'est ni un dessin animé, ni de la vidéo, ni du graphisme, ce n'est pas à proprement parler de l'image. C'est un fait, un robot en exercice dont la vocation est d'exprimer autre chose qu'une image, même lorsque ce qu'il fait est rendu perceptible par l'image. Ce qui devient attrayant est la conception du robot avec son scénario sous-jacent, non de l'image. Nous sommes dans un domaine différent, tant pour l'inspiration que pour

[27] Le docteur Septimus est un personnage d'une aventure de Blake et Mortimer. Il s'agit de *La marque jaune* , bande dessinée de Edgar Jacobs, édition Blake et Mortimer, Bruxelles 1987 (première édition 1953).

l'esthétique. Le graphisme en tant que tel y devient une voie possible mais il est envisagé sous la forme des mécanismes techniques et mathématiques, c'est-à-dire du point de vue des scénarios mécaniques qui le conditionnent.

Nous entrons dans un domaine où écrire donne le sentiment de fabriquer. Il y a du bricolage dans la programmation. L'imaginaire repose en partie sur des machines que l'on se représente mentalement un peu comme s'il s'agissait de machines réelles. Et ces machines on les habille de fantasmes, de couleurs, de formes mais aussi de recherche et de réflexion.

PARTIE 3

Espace robographique, pixel et ligne sur écran

Si nous observons la rue à travers la fenêtre, ses bruits sont atténués, ses mouvements sont fantomatiques et la rue elle-même, à cause de la vitre transparente, mais dure et rigide, paraît un être isolé palpitant dans un « au-delà ». Mais que l'on ouvre la porte : nous sortons de l'isolement, nous participons de cet être, nous y devenons agissants et nous vivons sa pulsation par tous nos sens. (…)

L'œuvre d'art se reflète à la surface de la conscience. Elle se trouve « au-delà » et, quand l'excitation cesse, elle disparaît de la surface sans trace. Il y a là aussi comme une vitre transparente, mais dure et rigide, qui empêche tout contact direct et intime. Là encore nous avons la possibilité de pénétrer dans l'œuvre, d'y devenir actifs et de vivre sa pulsation par tous nos sens.

Wassily Kandinsky.

La partie Deux, qui expose des créations, se conclut sur le fait que du point de vue de la programmation tout graphisme, toute figure et représentation sont d'abord des petites machines sous forme de structures de données et d'algorithmes. Cette dimension est importante pour la corrélation avec l'imaginaire et nous avons introduit le concept de robographe.

Dans cette partie nous allons préciser l'espace robographique en partant d'éléments primitifs qui le constituent. Notre repère initial sera l'image mais progressivement nous nous en écarteront au fur et à mesure que sera mieux perçue la spécificité fondatrice d'un espace robographique.

1. Espace technique-création, espace perceptif

Ni robot ni graphe, le robographe dépasse les perspectives de l'image. C'est une troisième voie et la perspective est celle de la découverte d'un espace, l'espace robographique. Cet espace se constitue de figures robotiques telles qu'elles peuvent coïncider avec l'imaginaire du concepteur. Et l'imaginaire trouve dans cette coïncidence les moyens de s'exprimer, à savoir de bâtir une écriture. Cette écriture qui relie imagination et technique de programmation ouvre le champ perceptif spécifique de cette pratique. Et sur la base des nouvelles impressions possibles, le regard est pour ainsi dire sensibilisé à un nouveau monde. L'idée d'un impressionnisme technologique est suggérée.

1.1 Mouvements d'intériorisation et d'extériorisation

L'objectif de cette partie est de concrétiser ce point de vue et de montrer des mouvements d'intériorisation et d'extériorisation propre à l'écriture de programmes. Le cheminement adopté part de concepts fondamentaux de l'image numérique. L'image, envisagée du point de vue du programmeur, crée l'ouverture d'une sensibilité à la programmation. C'est évidemment l'appropriation de la programmation qui donne accès à l'espace robographique avec la possibilité de poser une écriture personnelle. Sans donner trop avant dans des questions scientifiques relatives à la synthèse d'image, nous entrons cependant résolument dans des réalisations de programmes. C'est le moyen le plus probant de mettre en évidence l'importance des problématiques de structures de données et d'algorithmes du point de vue de la sensibilité artistique.

Nous avons modélisé la démarche avec le schéma de la figure 1. Tout d'abord, l'image apparaît selon son double qui est un ensemble algorithmique codé de robots calculateurs-traceurs. Pour obtenir une image ce sont ces robots qu'il faut concevoir. Cette conception de robots renvoie à la conceptualisation et à la modélisation de chaque robot et nous amène à la figure abstraite du robot, c'est-à-dire une figure du robot telle qu'elle est

intériorisée par le concepteur. Notre hypothèse est ici que le simple fait de raconter comment faire un robot, alors que nous ne le faisons pas effectivement, nous amène à ce niveau d'abstraction. C'est-à-dire que la description du robot permet de s'en représenter mentalement le processus et le fonctionnement.

Le principe de notre développement peut être schématisé de la façon suivante :

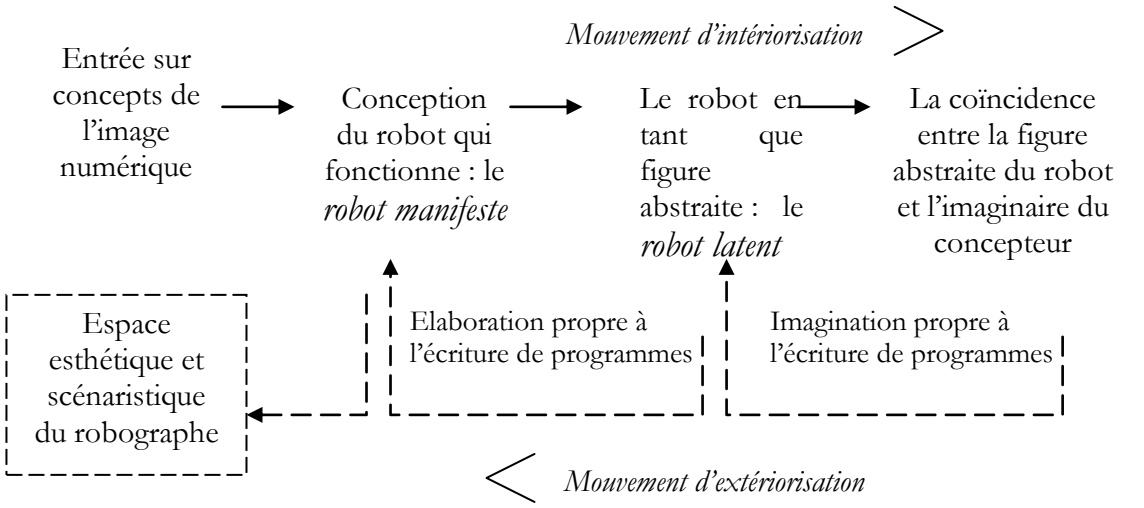


Fig. 1 : Intérioriser la programmation en vue d'imaginer et d'écrire des programmes selon une sensibilité artistique et de création

Image mentale

Cette image mentale détient, à travers la figure abstraite du robot, l'essentiel de la rigueur technique nécessaire à la réalisation d'un fonctionnement du programme. Mais en même temps, en tant qu'image mentale intériorisée, elle peut avoir perdu un peu de la rigidité de la réalité du robot qu'elle représente. C'est grâce à l'acquisition progressive de cette souplesse qu'une coïncidence peut avoir lieu entre intériorisation du fonctionnement possible d'un programme et l'imaginaire, la sensibilité du concepteur. Ainsi dès lors que l'on acquiert la compétence de pouvoir se faire une image mentale d'un programme et de son fonctionnement, cette image peut entrer en résonance de façon très large avec l'imaginaire, et il devient possible d'imaginer de nouveaux programmes. L'image mentale a cette propriété de permettre la transposition des mécanismes, mais aussi l'adjonction de mécanismes nouveaux par des processus intuitifs de métamorphose. En d'autres termes cette image mentale est l'expression d'une connaissance vivante de ce que l'on peut qualifier de « matière à robot », et cette connaissance s'élargit progressivement avec la pratique et l'étude qui intègrent les contraintes spécifiques à cette matière.

Adaptabilité de la compétence

Le degré d'indépendance et de malléabilité qu'une représentation mentale de programme peut prendre dans l'esprit d'un concepteur, correspond à son niveau de connaissance associé à la maturité de sa pratique. Et sur un plan qualitatif, cette compétence dépend aussi beaucoup des grandes orientations personnelles prises par le concepteur. Dans le fond c'est un peu comme le cas du musicien et de son rapport à la maîtrise d'un

instrument de musique. Lui aussi acquiert progressivement, par son travail, un degré d'indépendance et de souplesse par rapport à sa technique. Mais sa technique, au-delà de la maîtrise de l'instrument lui-même, est largement dépendante d'une orientation plus générale. Par exemple un instrumentiste interprète est le plus souvent très en difficulté lorsqu'il s'agit d'improviser. Inversement lorsqu'un maître de l'improvisation interprète une œuvre, la place de l'œuvre est celle d'un repère à partir duquel il compose son propre cheminement. Ajoutons également que la créativité ne nécessite pas un niveau exceptionnel de maîtrise technique, et quelles que soient les orientations choisies. L'exagération d'une spécialisation crée une fermeture qui peut se traduire par une rigidité de la compétence. Ce point est également vrai en musique où il apparaît par exemple que les compositeurs sont rarement des instrumentistes virtuoses.

1.2 Etablir une écriture artistique et de création

Ainsi donc la compétence de pouvoir se représenter mentalement un programme rencontre les tendances propres au concepteur, ses goûts, son imaginaire, son inconscient. Il lui est possible de faire entrer son imaginaire dans des réalisations et probablement, l'inconscient y trouve lui-même sa place. Cette activité conduit alors à formuler de nouvelles figures abstraites de robots en relation avec des problématiques conceptuelles, de structures de données et d'algorithmes. C'est aussi là qu'il devient possible à l'artiste d'établir son écriture.

Bien entendu, ce cheminement n'a plus de rapport nécessaire avec l'image, point de départ que nous avons choisi. L'espace de conception dont il est question est manifestement l'espace esthétique et scénaristique du robographe, l'espace robographique.

Plaisir de la contemplation

Tout l'intérêt que nous portons à l'espace robographique est esthétique non pas exclusivement au regard des résultats que l'on peut obtenir mais dans la démarche même, du point de vue du plaisir qu'il y a à faire des robographes. Peut-on imaginer un peintre qui n'aime pas peindre et ne s'intéresse qu'aux résultats de sa peinture ? Ou un musicien qui n'aime pas jouer de la musique et ne s'intéresse qu'aux morceaux finis qu'il a produit ? Sans doute de telles personnes existent. Mais ce n'est pas ce qui guide notre démarche. Il y a de la joie à *faire* un programme. L'activité de conception se double d'une certaine forme de contemplation. Et il est nécessaire que ce soit porteur. Il arrive que l'effort de résoudre un problème posé par la réalisation d'un robographe ne soit pas agréable et puisse se traduire par un souci éventuellement préoccupant sur plusieurs jours.

Nous allons donc remonter aux sources de la réalisation d'une image. C'est-à-dire que nous allons explorer le « code source » des fondements du pixel, de la toile-écran, de quelques primitives de dessin. Nous passerons de l'espace continu de représentations abstraites à celui discontinu des réalisations automates ; c'est par exemple le cas pour les droites avec leurs algorithmes de tracés. Mais nous verrons également que certaines compositions algorithmiques peuvent être envisagées comme remontant vers un espace abstrait à partir d'une construction séquentielle, par exemple un tracé de contour. Ensuite compte tenu de l'expérience faite, nous pourrons nous échapper du seul cadre de l'image afin de jouir pleinement de la totalité de l'étendue de l'espace robographique.

2. Aspect pratique de l'environnement de développement

Ce qui importe dans cette partie est le dynamisme individuel d'intériorisation, d'imagination et de production dans une démarche de programmation portée sur la création. Cependant aucun informaticien-programmeur ne commence jamais à partir de rien son travail de programmation. Il s'appuie toujours sur un environnement de programmation qui comprend librairies, compilateur, système d'exploitation, choisis en relation avec le projet poursuivi. En ce qui concerne la librairie, nous nous appuyons sur l'environnement de programmation libre « Allegro[28] » qui repose sur un dynamisme collectif très intéressant. Mais notre attention est surtout focalisée sur des questions d'écriture (structures de données et algorithmes), nous n'abordons d'ailleurs pas ce qui a trait au système et au compilateur[29].

Librairie Allegro

Allegro est une librairie de programmation c'est-à-dire un ensemble de fonctions et de structures de données mises à disposition des programmeurs pour faciliter leur tâche (voire la rendre possible !). C'est une librairie de développement pour les jeux et tous les autres types de programmation multimédia. Son développement est le fait d'une communauté de programmeurs réunis sur la toile Internet. Au moins cent quatre vingt quatre personnes participent activement au développement. De très nombreuses autres, utilisateurs attentifs, permettent de remonter des bugs afin qu'ils soient corrigés, ou contribuent à l'apport de nouvelles idées, ou laissent libre et ouvert le code source des applications qu'ils réalisent avec la librairie, ou encore écrivent quelques modules complémentaires à la librairie (des « add-on »). Même si la culture sous-jacente est à l'origine celle du jeu, il y a en fait toutes sortes d'utilisateurs et de concepteurs, toutes sortes de projets de création qui se sont associés à cette librairie. Le tout forme donc un ensemble très efficace et dynamique pour le domaine d'une programmation de création.

Cette précision sur l'environnement à son importance car les programmes que nous allons présenter s'appuient souvent sur des structures de données et des fonctions puisées dedans. Mais ceci ne diminue en rien les explications données ni même le caractère général de ce qui est présenté. En revanche cela légitime l'emploi des termes anglophones qui correspondent en fait, dans ce contexte, à des appellations internationales.

[28] L'origine du nom Allegro est le suivant :
« According to the Oxford Companion to Music, Allegro is the Italian for "quick, lively, bright". It is also a recursive acronym which stands for "Allegro Low Level Game Routines".»

[29] La librairie Allegro est dite « cross-platform » c'est-à-dire qu'elle est conçue de façon à fonctionner sous différents systèmes (DOS, Windows 98, 1000, 2000, XP, Linux, BeOs, Mac OS, QNX) et compilateurs (DJGPP, Visual C++, Borland, Mingw 32, Dev C++, etc.)

Chapitre 9

Couleur numérique du pixel

Etablir un point dans le chaos, c'est le reconnaître nécessairement gris en raison de sa concentration principielle et lui conférer le caractère d'un centre originel d'où l'ordre de l'univers va jaillir et rayonner dans toutes les dimensions. Affecter un point d'une vertu centrale, c'est en faire le lieu de la cosmogénèse. A cet avènement correspond l'idée de tout Commencement (conception, soleils, rayonnement, rotation, explosion, feux d'artifice, gerbes) ou, mieux : le concept d'oeuf.

Paul Klee

Dans le chapitre précédent, à propos du développement d'un imaginaire, nous avons évoqué le travail de Françoise Holtz-Bonneau qui met en relation le monde de l'imprimerie et celui de l'image numérique. Elle rapproche la décomposition de la séquence graphique en unités autonomes, les caractères, avec par analogie, la décomposition de l'image numérique en pixels. Ce faisant nous pouvons faire plusieurs observations. Si l'autonomie du caractère est facile à se représenter tant ces procédés font maintenant partie d'une culture latente, l'autonomie du pixel quant à elle reste encore beaucoup plus problématique. Qu'est-ce que l'autonomie du pixel ? Et même peut-on se poser la question de savoir ce qu'est réellement un pixel, et la place qu'il tient effectivement dans la conception d'une image numérique.

Pour introduire l'espace robographique, et à l'instar de Françoise Holtz-Bonneau, nous sommes partis du pixel. Nous laissons de côté le pixel du point de vue des modèles de fonctionnement physique de l'écran d'ordinateur ou des phénomènes de perception visuelle. Nous n'abordons pas non plus la question de la modélisation de la couleur numérique[1]. Notre attention se tourne vers la nature informatique du pixel telle qu'elle apparaît au programmeur. Il s'agit de glisser de l'identité visuelle du pixel qui est une primitive de l'image numérique vers l'identité du pixel au sein d'un programme informatique, en tant que robographe primitif qui entre dans la construction de l'image.

Cette étude du pixel version programmation informatique est divisée en deux blocs principaux. Le premier bloc a trait à la couleur. Il fait l'objet de ce chapitre ainsi que du chapitre suivant avec la création de palettes de couleurs. Le second bloc est consacré au pixel dans l'espace mémoire de l'image numérique, le chapitre 11. Dans les chapitres qui suivront le pixel sera investi dans des algorithmiques tout d'abord de géométrie, puis de compositions (chapitre 12), et enfin il sera oublié progressivement, comme robographe parmi d'autres, afin que nous parvenions pleinement au-delà de l'image, dans la dimension élargie, polymorphe, de l'espace robographique (chapitre 16).

1. Définition du pixel

Les pixels apparaissent à l'écran sous forme de petits carrés dont la taille dépend de la résolution affectée à l'écran via le système d'exploitation et une carte graphique (les résolutions habituelles sont 640 pixels par 480 pixels, 800 par 600, 1024 par 768 etc.). Indépendamment de l'écran, pour chaque image numérique, les pixels quadrillent un plan rectangulaire de n_y lignes sur n_x colonnes. Ce plan rectangulaire est celui de l'image et chaque pixel y est une position fixe assortie d'une couleur (figure 9.1).

La couleur du pixel est homogène mais elle peut être changée, elle est variable. L'image numérique est restituée comme une mosaïque, par les couleurs associées à chacun des

[1] Pour les questions d'écran et de perception consulter par exemple [BRET, 1988,], pour la couleur [CAILLAUD, 2001]

pixels qui la constitue. Chaque couleur aura un numéro et le nombre total de couleurs accessibles pour l'image va dépendre du nombre de bits utilisés pour le codage de la couleur (8, 15, 16, 24, 32 bits).

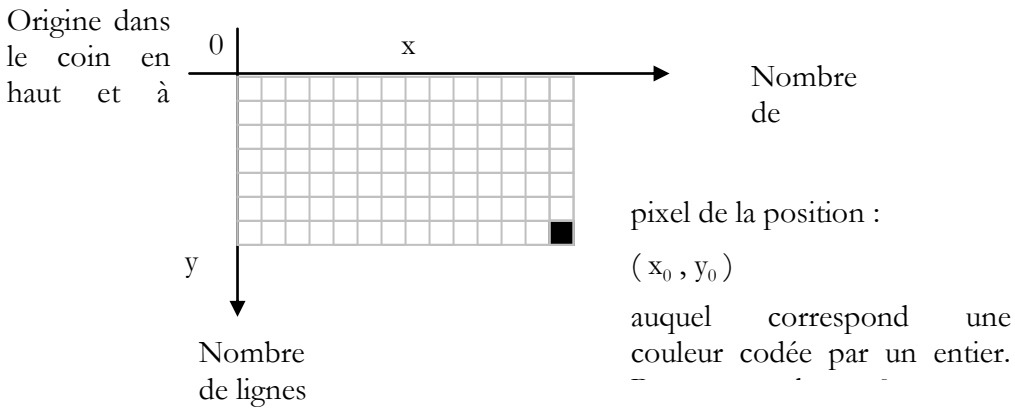


Fig. 9.1 : Trois variables caractérisent le pixel dans un environnement de programmation, deux pour la position (x, y) fixe par rapport à l'image, une autre pour la couleur. Toutes trois prennent des valeurs entières.

2. Questions liées à l'implémentation machine de nombres

Pour celui qui programme, le pixel se traduit par trois nombres entiers, deux pour la position et une pour la couleur. La couleur, dans le modèle « RVB » (ou « RGB » pour red, green, blue) résulte d'un mélange de rouge, de vert et de bleu ; une valeur unique représente ce mélange. Fabriquer une couleur revient à fabriquer un nombre qui soit contient, soit désigne les trois composantes de rouge de vert et de bleu. Pour comprendre la construction d'un tel nombre, le premier point est de se familiariser avec certaines propriétés du concept d'entier et la façon dont ces propriétés sont couplées avec une implémentation machine.

Nombre-octet : une micro-chimère

Comme nous l'avons évoqué dans le chapitre Quatre, un modèle de données langage (de programmation) conjugue une volonté conceptuelle avec la réalité physique de la machine. Par réalité physique de la machine nous retenons essentiellement le fait d'une mémoire constituée de micro-interrupteurs dont les états binaires sont interprétés par des valeurs de 0 ou de 1. L'ensemble de ces micro-interrupteurs est divisé en cellules. Pour être localisée, chaque cellule dispose d'une adresse. Un dispositif complexe de lecture et d'écriture complète la mémoire.
Ces cellules sont là pour l'écriture et la lecture de mots binaires. Cette réalité physique de la mémoire informatique est ainsi appelée à coïncider avec un espace conceptuel numérique binaire à partir duquel il est possible d'étendre plus largement l'univers du calcul. Mais il naît là une espèce à part de nombres qui, pour moitié appartiennent au

monde des idées, et pour moitié à la finitude de leur réalité physique. Nous le baptisons parfois de « nombre-octet ». Le nombre-octet est pour ainsi dire une micro-chimère qui, du fait de cette hybridation, possède des caractéristiques propres. La « matière-octet » du nombre n'est pas sans être organisée, elle n'est pas vide de sens.

Se représenter comment est construite la mémoire de l'ordinateur donne une consistance aux nombres, et véritablement, forme et matière pour toutes les espèces de variables qui sont manipulées lors de la programmation. De plus il est utile de comprendre comment ces variables peuvent être localisées dans la mémoire afin de les retrouver, d'agir sur elles et d'en modifier les valeurs.

2.1 Espace binaire de la mémoire physique

Selon les explications données par Alfred Aho et Jeffrey Ullman, la mémoire repose sur des « puces mémoires » ou « puces RAM ». Une puce est un circuit intégré, d'environ un centimètre carré qui rassemble un grand nombre de conducteurs et composants électriques. C'est là où sont casés les fameux « bits » en grande quantité. Leur nombre est toujours une puissance paire de deux ; c'est-à-dire 2^{2i} avec i un entier. Tous les bits ont une adresse et ils sont en quelque sorte alignés du premier, d'adresse « 0 », au dernier dont l'adresse correspond au nombre de bits sur la puce. Du fait de cette adresse il est possible de lire et d'écrire sur chaque bit de la puce mémoire. Alfred Aho et Jeffrey Ullman décrivent la situation de la façon suivante :

> « Quand on présente une adresse et le signal « lire », la puce trouve le bit à cette adresse et le ramène dans un tampon à partir duquel cette valeur peut être lue ou placée sur le bus de l'ordinateur. Nous pouvons aussi écrire une nouvelle valeur dans un bit désigné, en présentant à la puce l'adresse du bit accompagnée de la nouvelle valeur à écrire et du signal « écrire » » [AHO, 1993, p. 171].

Dans le chapitre Quatre nous avons mentionné que la mémoire principale est construite avec pour unité de base l'octet, quantité minimum de stockage qui a sa propre adresse. Pour avoir des octets, huit puces sont placées en parallèle. Huit puces sont alignées sur la même adresse. Chaque puce fournit alors un bit de l'octet et ils sont lus ou écrits en même temps. Lire un octet prend alors le même temps que de lire un bit.

Vient ensuite la nécessité de lire et d'écrire un « mot », c'est-à-dire de disposer de plusieurs octets consécutifs qui conservent possible l'accès à chacun d'entre eux. Le plus souvent la mémoire est construite sur la base de quatre octets consécutifs ; 32 bits qui ne remettent pas en cause la définition de l'octet comme plus petite unité de mémoire ayant sa propre adresse. Pour ce faire deux niveaux d'adresses sont dégagés. Celui du mot, de quatre octets en quatre octets, et celui, interne au mot, des octets qui le constitue. Si « a » est l'adresse d'un mot, « a+0 », « a+1 », « a+2 », « a+3 » sont les adresses de ses octets ; et « a » progresse de quatre en quatre, a est toujours divisible par quatre. Ce principe permet de conserver l'octet comme plus petite unité de stockage mais également, en 32 bits avec 32 puces parallèles, de bénéficier d'un quasi parallélisme sur quatre octets [AHO, 1993, p. 172].

La mémoire apparaît comme une organisation de cellules-mémoire empilées par rangées de quatre octets, chacun comportant huit bits. Chaque rangée a son adresse, un multiple de

quatre, et les numéros d'adresses progressent de quatre en quatre. Toute variable va correspondre à un bloc de mémoire dont la taille dépend du type de la variable. Ce bloc est repérable du fait de son adresse et dans le programme il est désigné par une étiquette symbolique, c'est-à-dire pour le programmeur le nom qu'il donne à sa variable au moment de la déclaration. A cette organisation et ce découpage de la mémoire en bits, octets et mots-octets correspond un espace conceptuel numérique appuyé sur les traductions binaires possibles des bits, octets et mots-octets.

2.2 Faire un nombre avec des octets

A propos du pixel nous avons remarqué que le nombre de bits utilisés pour le codage des couleurs donne le nombre maximum de couleurs accessibles dans un programme. Pour 8 bits il y a 256 couleurs, pour 16 bits il y a des milliers de couleurs et 24 ou 32 bits sont dits mode « vraie couleur » ou millions de couleurs (« truecolor » selon l'appellation anglophone). Le principe de la notation étendue d'un entier permet de repérer comment un mot binaire peut se décomposer et trouver une équivalence décimale.

Notation étendue

Rappelons qu'un nombre décimal comme « 8542 » peut s'écrire de la façon suivante [LIPSCHUTZ, 1983, p. 2 à 5] :

$8542 = 8*10^3 + 5*10^2 + 4*10^1 + 2*10^0$
$= 8*1000 + 5*100 + 4*10 + 2*1$
$= 8000 + 500 + 40 + 2$.

Et cette décomposition constitue ce qui est nommée la *notation étendue de l'entier*

De même un nombre binaire peut-être représenté en notation étendue. Si l'on prend par exemple le chiffre binaire 110101 on obtient :

$$110101 = 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$$

et il n'y a plus qu'à effectuer le calcul :

$$110101 = 32 + 16 + 0 + 4 + 0 + 1$$
$$= 53$$

De l'octet binaire aux nombres décimaux

Les découpages de mots sont liés à ceux de la mémoire. Chaque découpage correspond à une taille et définit un type. Dans le langage C, le concept mathématique d'entier se traduit par plusieurs types, char, short, long selon que l'entier est respectivement codé sur 8, 16 ou 32 bits. Et, par rapport au concept mathématique, l'entier codé dans la machine est limité par sa taille mémoire.

Prenons l'octet, unité de base de la mémoire centrale ayant sa propre adresse, et occupons nous uniquement des valeurs dites « non-signées » indépendantes de tout système de signe positif ou négatif. L'octet possède huit positions. Huit bits qui sont soit sur un, soit sur zéro. Cela donne un nombre binaire dont on dispose en décimal par une addition de puissances de deux selon le principe de la notation étendue (figure 9.2).

Les bits sont disposés de droite à gauche de la position 0 à la position 7 comprises. La

puissance correspond à la position du bit (« pos » sur la figure 9.2) et la valeur à additionner résulte de la multiplication de 2^{pos} par la valeur du bit.

Le nombre 110101 de l'exemple ci-dessus donne l'octet suivant :

Fig. 9.2 : Polyvalence binaire-décimale d'un octet.

bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1
pos 7	pos 6	pos 5	pos 4	pos 3	pos 2	pos 1	pos 0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	1	0	1	0	1
↓	↓	↓	↓	↓	↓	↓	↓
0	+0	+32	+16	+0	+4	+0	+1

On constate que le nombre décimal le plus grand codé sur un octet est 255. Ce serait :
$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$
Et comme la valeur zéro compte pour un nombre, sur un octet on a 256 nombres possibles, soit 2^8 cas. Le principe de cette addition est le même pour 16 bits avec des positions de 0 à 15 et avec cette fois 2^{16} cas. Idem pour 24 bits avec 2^{24} cas et 32 bits avec 2^{32} cas. En fait pour *n* bits il y a *n* positions de 0 à *n* -1 ce qui donne un total de 2^n cas.

Si le nombre à coder dépasse la possibilité de codage qu'offre l'espace mémoire, il y a un dépassement de capacité. Le langage C précise, en ce qui concerne les entiers non signés, qu'ils se comportent alors selon les lois de l'arithmétique modulo 2^n ou *n* représente le nombre de bits constitutifs de l'entier [KERNIGHAN, 1995, p.195]. Par exemple un seul octet ne peut pas restituer des nombres supérieurs à 255. Pour ces nombres il se comporte comme si un modulo 2^8 était effectué, et la partie qui dépasse est tronquée. Affecter la valeur 257 à un octet donne 257 modulo 256 ce qui est égal à 1 (figure 9.3).

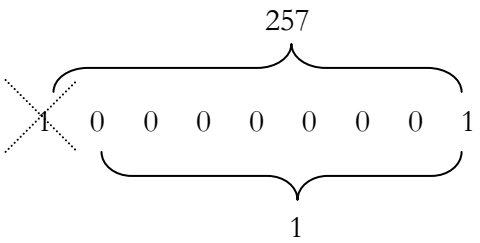


Fig 9.3 : Troncature d'un nombre par l'octet

En C le type qui désigne un octet unique est le type « char ». Mais, à la différence de l'octet que nous avons utilisé, un char peut prendre des valeurs positives ou négatives. *C'est-à-dire que la matière-octet du nombre doit incorporer le principe du signe, qu'il soit positif ou négatif.*

2.3 Faire un nombre positif ou négatif

Entier signé ou non-signé

Dans le langage C, tout nombre est implicitement dit « signé » ce qui veut dire qu'il peut prendre des valeurs négatives *ou* positives. Cette propriété est attribuée par défaut à tous les types d'entiers, mais le cas du type char est particulier. Le char est initialement dédié au

codage des caractères, comme son nom l'indique (character). Il n'est pas réservé à une utilisation en tant que nombre entier. Lorsqu'on lui affecte un caractère il prend la valeur entière, positive ou négative, qui code ce caractère. Utilisé pour stocker des nombres, le fait qu'il soit signé ou non par défaut va dépendre de l'implémentation. Le mot clé du langage « signed » peut alors être utilisé afin de certifier qu'il sera signé. Pour les autres types d'entiers que le char, « signed » est autorisé quoique inutile et redondant [KERNIGHAN, 1995, p.212]. Inversement, il est possible d'avoir des nombres « non signés » qui excluent la propriété d'être positif *ou* négatif et dont la grandeur ne sera *que* positive et calculée sans système de signe. Quel que soit leur type, le mot clé qui devra nécessairement être utilisé pour définir un entier « non signé », est « unsigned ».

Coder le signe, méthode du complément à deux

Dans un environnement de programmation l'éventualité pour un nombre entier d'être négatif a nécessité le codage de cette propriété du nombre avec le nombre lui-même. Prenons par exemple un nombre codé sur un octet, de type char. Pour qu'il soit signé on peut par exemple utiliser la dernière position, le huitième bit, comme bit de signe avec 0 pour positif et 1 pour négatif. Tous les autres bits sont alors réservés à la grandeur que peut prendre ce nombre. Dans ce cas on obtient pour un octet des valeurs de 127 à –127. Mais il y a plusieurs problèmes. D'abord il y aura deux 0, un positif et un négatif. Et aussi les opérations d'addition ou de soustraction entre nombres positifs et négatifs seront compliquées. Si x et y ont des signes différents le signe de x+y est celui du plus grand en valeur absolue. Il faut donc commencer par connaître lequel des deux est le plus grand pour connaître le signe du résultat, puis soustraire du plus grand le plus petit.

Il y a plusieurs méthodes possibles pour le codage des nombres négatifs. Nous allons évoquer la méthode du complément à deux. Cette méthode a l'avantage de permettre la transformation d'une soustraction en addition. Rappelons que parmi le nombre limité d'instructions simples que le microprocesseur effectue[1], il y a l'addition de bits.

Principe du complément

En fait il y a deux modalités de complément. Selon la base dans laquelle on se trouve décimale ou binaire. Il y a le « complément à la base moins un » et le « complément à la base » [LIPSCHUTZ, 1983, p.14 à 16]]

Dans le système décimal le complément à 9 (à la base moins un) d'un nombre A s'obtient en soustrayant 9 de chaque chiffre qui compose A. Le complément à 10 (à la base) de A est le complément à 9 auquel est ajouté 1. Par exemple :

Nombre décimal	4500	92299
Complément à 9	5499	07700
Complément à 10	5500	07701

[1] Le micro processeur exécute les instructions suivantes : déplacement de données en mémoire ; inversion de bits (NOT) ; opérations de type logique (AND, OR, XOR) ; addition de bits ; déplacement et rotation de bit [MERCIER, 1989, p.28]. Le langage C fournit une série d'opérateurs de traitement des bits qui correspondent à ces instructions primitives (« ~ » pour complément à un, « & » pour ET bit à bit, « | » pour le OU inclusif bit à bit, « ^ » pour le OU exculsif bit à bit, « << » pour décalage à gauche, « >> » pour décalage à droite).

Dans le fond le complément à 10 d'un nombre A à *n* chiffres est 10^n – A. Dans notre exemple 5500 est égal à 10000 – 4500. Et 7701 = 100000 – 92299.
Même principe dans le système binaire. Le complément à 1 d'un nombre A s'obtient en soustrayant 1 de chaque chiffre qui compose le nombre A. Le complément à 2 de A est le complément à 1 auquel est ajouté 1. Par exemple :

Nombre binaire	11010011	11001100
Complément à 1	00101100	00110011
Complément à 2	00101101	00110100

Le complément à 1 simplement inverse les chiffres. Ceux qui sont à 1 passent à 0 et ceux qui sont à 0 passent à 1. Le complément à deux inverse les chiffres mais s'arrête au dernier 1 le plus à droite qu'il laisse inchangé ainsi que les 0 restants après, ce qui revient à avoir ajouté 1 au complément à un.

La règle est la même que dans le système décimal. Soit un nombre binaire A à *n* chiffres, le complément à deux CA de A exprime la différence entre 2^n et A soit : CA = 2^n – A. On peut en déduire, si l'on ajoute (– 2^n) à chaque membre, la possibilité d'exprimer A négatif avec CA – 2^n = – A [AHO, 1993, p. 218 à 227] [BRAQUELAIRE, 1998, p.51].

Complément à deux dans un système de signes

La notation en complément à deux départage les nombres positifs des nombres négatifs selon la valeur du bit de tête.
S'il est égal à 0 le nombre est positif. Son interprétation est faite comme s'il s'agissait d'une grandeur non signée. Pour *n* bits, il sera compris dans la fourchette 2^{n-1} – 1 à 0. Sur un octet les valeurs possibles seront comprises entre 127 et 0, c'est-à-dire de 2^7 – 1 à 0.

En revanche si le bit de tête est égal à 1, il est interprété comme un nombre négatif. Puisque le bit de tête est égal à 1, les grandeurs s'échelonnent entre 2^{n-1} et 2^n – 1, c'est-à-dire pour un octet entre 128 et 255. Ces valeurs sont *interprétées* comme étant celles des compléments à deux des valeurs positives et, pour *n* bits, par rapport à 2^n. Pour obtenir les valeurs négatives qu'elles désignent il suffit de soustraire 2^n selon la formule que nous avons évoquée, CA – 2^n = – A..

Par exemple prenons *n* = 4. Nous pouvons visualiser la situation avec un tableau qui récapitule ce principe de codage du signe (figure 9.4) :

Pour *n* = 4, il y a 16 nombres possibles qui vont de 0 à 15. Les nombres positifs vont de 0 à 7. 0 n'a pas de correspondant négatif car le complément à deux de 0 est 2^4 ramené à 0 par le modulo 2^4. Les nombres dont les valeurs littérales vont de 8 à 15 commencent par 1 et sont considérés comme négatifs. Ils sont interprétés comme des compléments à deux auxquels est soustrait chaque fois 2^4. On utilise le principe de la notation étendue et de l'addition des puissances de deux et l'on soustrait 2^4 ce qui produit la valeur négative correspondante. Notons que le complément à deux de cette valeur négative revient à la valeur positive (Le complément à deux du complément à deux donne la valeur de départ). Les deux nombres, positif et négatif, sont reliés par cette relation de complémentarité à deux. Si l'on additionne par exemple 7 et – 7 ça fait 0111 + 1001 = 10000, la retenue provenant du bit de tête, 2^4, est éliminée ce qui donne 0.

NOMBRES POSITIFS		NOMBRES NEGATIFS		
Valeur décimale	Valeur binaire correspondante	Complément à deux « CA »	Valeur décimale littérale	Interprétation en nombre négatif du complément à deux avec la méthode : $CA - 2^n$
0	0000			
1	0001	1111	15	$-1 = 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 - 2^4$
2	0010	1110	14	$-2 = 1*2^3 + 1*2^2 + 1*2^1 - 2^4$
3	0011	1101	13	$-3 = 1*2^3 + 1*2^2 + 1*2^0 - 2^4$
4	0100	1100	12	$-4 = 1*2^3 + 1*2^2 - 2^4$
5	0101	1011	11	$-5 = 1*2^3 + 1*2^1 + 1*2^0 - 2^4$
6	0110	1010	10	$-6 = 1*2^3 + 1*2^1 - 2^4$
7	0111	1001	9	$-7 = 1*2^3 + 1*2^0 - 2^4$
		1000	8	$-8 = 1*2^3 - 2^4$

Fig. 9.4 : Système de signe, méthode du complément à deux.
(Pour mémoire rappelons que $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$)

Le problème du dépassement de capacité peut se poser. L'addition de deux entiers positifs, pour être juste ne doit pas être supérieure à la valeur positive maximum qui peut être codée dans le système signé. Par exemple 7+7 = 14 en grandeur absolue soit 1110 en binaire. Dans notre système signé et sur 4 bits ça donne le nombre – 2.
De même pour l'addition de nombres négatifs. -7 + - 7 = 1001 + 1001 = 10010 soit 0010 après suppression de la retenue c.à.d le nombre 2.
En revanche le résultat est juste quelles que soient les opérations s'il reste dans les limites définies par le nombre *n* de bits du codage (de 0 à 7 et de -1 à -8 dans notre exemple sur quatre bits) : -3+-4 = 1101 + 1100 =11001 soit 1001 après suppression de la retenue c.à.d le nombre -7 qui est le juste résultat.

Nous constatons également que la soustraction est opérable par une addition, ce qui est une propriété des compléments à deux. Est additionné un nombre interprété négatif :
3 – 7 = 0011 + 1001 = 1100 ce qui représente le résultat – 4 .

Le nombre de bits qui servent au codage des nombres définit la fourchette des valeurs possibles que pourront prendre ces nombres. Dans les langages de programmation des types sont prédéfinis par la norme du langage. Ils correspondent essentiellement à différentes tailles en octets.

2.4 Différents types d'entiers par la taille en octets

Dans le langage C il y a trois types pour désigner un entier : « short int », « int » et « long int » et chacun correspond à une taille en octet.

1. « short int » pour entier court codé sur deux octets est généralement abrégé en « short » . S'il est signé sa valeur va de -2^{15} à $2^{15}-1$ c'est-à-dire de –32768 à 32767. S'il n'est pas signé sa valeur est comprise entre 0 et $2^{16}-1$, soit 65535.

2. « long int » pour entier long, codé sur quatre octets est abrégé en « long ». S'il est

signé sa valeur va de -2^{31} à $2^{31}-1$ c'est-à-dire de –2147483648 à 2147483647. S'il n'est pas signé il prend des valeurs entre 0 et 2^{32} –1, soit 4294967295.

3. En fonction de l'environnement de programmation, le « int » tout court est soit analogue au short soit analogue au long : « chaque compilateur est libre de choisir des tailles d'entiers adaptés à la machine sur laquelle il tourne, mais il doit respecter des tailles minimales de 16 bits pour les types short et int, et de 32 bits pour le type long. De plus les shorts ne doivent pas être plus longs que les ints, qui ne doivent pas être plus longs que les longs. » [KERNIGHAN, 1995, p. 36]. Dans l'environnement sur lequel nous travaillons, PC-Windows et compilateur Visual C++ 6, le « int » est sur quatre octets, analogue au long.

Rappelons que par défaut ils sont signés, sinon ils sont préfixés par le mot-clé « unsigned ». [BRAQUELAIRE, 1998, p.52].

Maintenant que nous avons une idée de la constitution des variables fondamentales en relation avec la mémoire centrale, physique, de l'ordinateur nous pouvons aborder le rangement dont elles font l'objet dans la mémoire.

2.5 Se représenter les variables dans la mémoire

Lorsque dans le programme des variables de différents types sont déclarées, les octets qui leur sont nécessaires sont alloués de façon consécutive dans la mémoire principale de l'ordinateur à partir d'une adresse donnée. Toutefois certains types ont besoin de commencer sur une frontière de mot, c'est-à-dire à une adresse divisible par quatre, ce que nous avons vu un peu plus haut.

Soit la déclaration suivante :

```
struct{
        char toto ;
        int titi ;
} ;
```

Cette structure va probablement demander huit octets et non pas cinq en mémoire du fait d'un besoin d'alignement de l'entier « titi » sur une frontière de mot. Cette nécessité d'alignement peut engendrer « des trous » non référencés dans la structure [KERNIGHAN, 1995, p.136].

Alfred Aho et Jeffrey Ullman proposent une illustration comme suit [AHO, 1995, p. 184]. Admettons que nous ayons dans un programme la déclaration suivante :

```
int i, j ;
char tab[6] ;
struct bob{
        int titi ;
        char toto[5] ;
} ;
int calc[5] ;
```

Prenons arbitrairement 100 comme première adresse pour la variable « i » ; « j » est à 104, (le type int représente quatre octets) « tab » commence ainsi à 108 jusque 114 (le char fait un octet). « bob » commence sur une frontière à 116 donc il n'y a rien de 114 à 116. « bob.titi » prend quatre octets jusque 120 et « toto » cinq octets, jusque 125. A nouveau rien de 125 à 128 et « calc » commence sur une frontière à 128 jusque 148. Cette situation est illustrée par la figure 9.5.

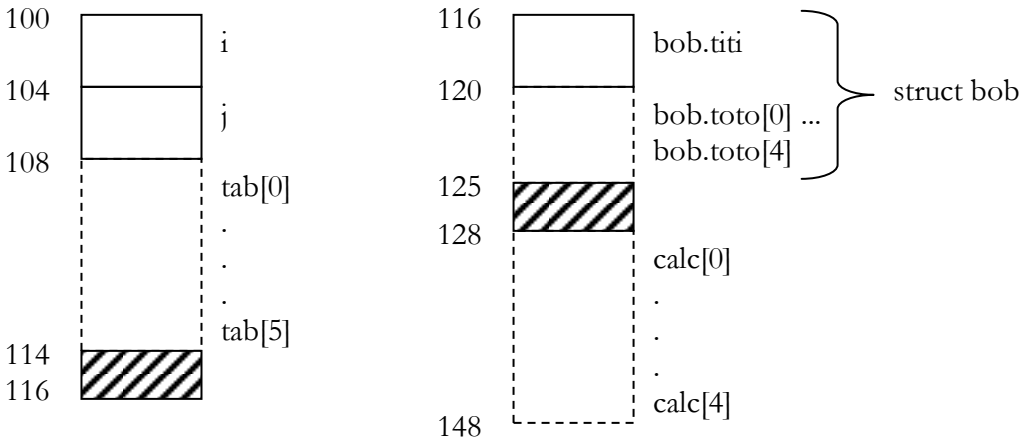


Fig 9.5 : Se représenter des variables en mémoire

Objets et valeur-g

Notons que dans le langage C, un *objet* est simplement défini par une zone mémoire qui porte un nom à la différence du C++ où la notion d'objet est plus élaborée. Dans un programme C, il y a en fait plusieurs façons de désigner un objet, de faire référence à un objet. La plus évidente est par exemple la déclaration d'une variable, avec un type et un identificateur, qui est son nom, à un endroit du programme ; « i », « j », « table », « bob », « bob.titi », « bob.toto », « calc[0] » etc. sont des objets. Ces expressions possèdent une adresse en mémoire et sont qualifiées de « lvalue » avec « l » pour left, où « valeur-g » ou encore « valeur de localisation d'objet ». C'est-à-dire qu'il s'agit d'expressions pouvant être utilisées dans la partie gauche d'une affectation [BRAQUELAIRE, 1998, p. 105] [KERNIGHAN, 1995, p. 196]. Par exemple il est possible d'affecter la valeur 5 à bob.titi ce qui donne l'expression « bob.titi = 5 » en revanche 5 n'est pas une lvalue et il n'est pas possible d'affecter à 5 une valeur. Remarquons qu'un tableau comme « calc » est une lvalue dans la mesure où « calc » peut figurer dans la partie gauche d'une affectation. Mais « calc » n'est pas modifiable sans adjonction d'une position dans le tableau ; « calc = 5 » n'est pas possible en revanche « calc[0] = 5 » est possible.

Avec cette idée des nombres et de leur implémentation dans la machine, nous pouvons maintenant nous rapprocher de la notion de couleur numérique. Ces propriétés des nombres-octets que nous avons vues, confèrent à la couleur une nature robographique. *Dans l'espace robographique, la couleur apparaît d'abord comme un être numérique organisé, invisible dans sa complexion.*

3. Deux modes des couleurs

La couleur RGB résulte d'un mélange de rouge de vert et de bleu et il y a essentiellement deux cas pour opérer ce mélange. Le mode « vraie couleur » c'est-à-dire 15, 16, 24 ou 32 bits. Et le mode 256 couleurs géré avec une « palette ». Dans les deux cas les proportions de rouge, de vert et de bleu sont déterminées par des valeurs numériques et la couleur résultante est codée sur un entier.

3.1 Mode « vraie couleur »

Le codage de la couleur utilise le type unsigned short (short non signé) qui regroupe deux octets, pour le 15 et 16 bits. Il utilise le type unsigned long (long non signé, plutôt que int non signé dans un soucis de portabilité, c'est-à-dire d'indépendance par rapport à l'environnement de programmation) qui regroupe quatre octets pour le 24 et 32 bits. Pour le codage sur 15 bits, il y a cinq bits par couleur. En 16 bits rouge et bleu sont codés sur cinq bits, le vert sur six. Pour 24 et 32 bits, un octet est consacré pour chacune des composantes rouge, verte, bleue de la couleur (figure 9.6). En 32 bits l'octet restant est utilisé pour coder des informations particulières relatives à des effets de transparence ou de mixage d'images codées en différents formats par exemple.

On a quelque chose du genre :

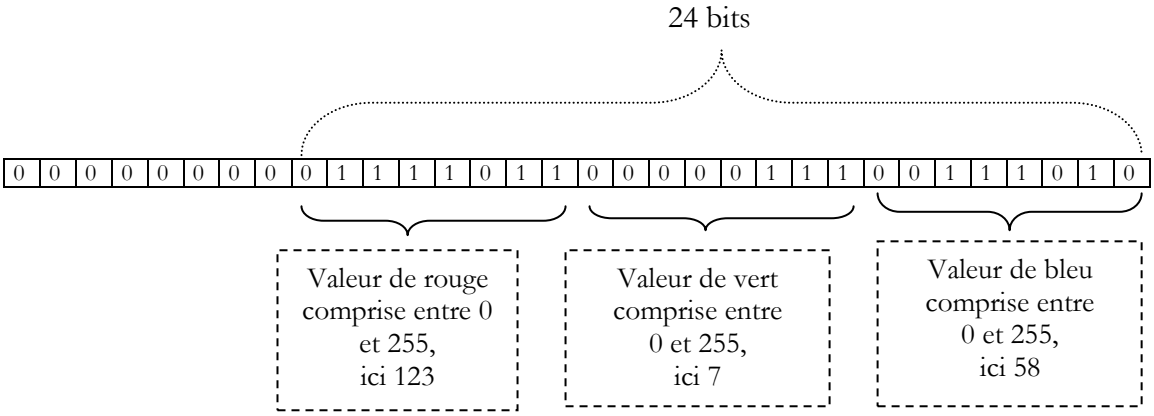


Fig. 9.6 : Décomposition d'un entier en un mélange de couleur

Ce qui donne également un nombre, ce serait la couleur n° 8062778 dans cet exemple.

3.2 Mode 256 couleurs

En revanche pour le mode 256 couleurs, la construction est un peu différente, en relation avec le format VGA pour la partie matérielle, la carte graphique et l'écran. La couleur est construite sur trois octets mais à l'aide d'une structure du type :

```
typedef struct RGB
{
  unsigned char r, g, b;
} RGB;
```

Cette structure comprend trois *char* non signés qui peuvent prendre des valeurs entre 0 et 255 (toutefois le format VGA veut que les graduations pour chaque composante de couleur soient bridées de 0 à 63. C'est-à-dire que chaque composante soit codée sur 6 bits.)

Typedef, type « RGB »

A propos du mot clé « typedef », c'est une fonctionnalité du langage C [KERNIGHAN, 1995, p.143] et [BRAQUELAIRE, 1998, p85]. Le principe de typedef est de transformer un nom d'objet en un nom de type. Pour ce faire « typedef » est placé avant la définition de l'objet, struct RGB dans notre cas, et le nom voulu est donné après l'accolade de fermeture de bloc « } », RGB donc. L'instruction est close ensuite avec le point virgule habituel. Par là « RGB » devient un nouveau nom de type utilisable comme un type ordinaire. De façon générale, l'avantage est de pouvoir baptiser le plus clairement possible des types considérés comme essentiels dans l'écriture du programme. Outre la lisibilité il peut y avoir également un avantage en cas de modification du programme.

Attribuer des valeurs de couleur

La structure RGB comprend trois champs de un octet chacun. Un pour le rouge, un pour le vert et un pour le bleu. Les couleurs sont construites en attribuant des valeurs aux différents champs. Par exemple les déclarations suivantes définissent des structures RGB et elles sont initialisées pour avoir du vert, du noir, du blanc, du gris :

```
RGB vert = { 0,  63, 0  };
RGB noir  = { 0,  0,  0  };
RGB blanc = { 63, 63, 63 };
RGB gris  = { 32, 32, 32 };
```

Le vert sera un vert très vif. C'est le maximum de vert possible dans ce cadre. Mais si la valeur de vert était rapprochée de zéro, la quantité de vert diminuerait d'autant. Si le rouge et le bleu sont à 0, la diminution du vert se traduit par un assombrissement progressif jusqu'à disparition totale de la couleur avec le noir. Inversement si le vert est au maximum, le bleu et le rouge à 0 et que l'on augmente les valeurs du bleu et du rouge, lorsque les trois primitives sont à 63 on obtient du blanc. Avec la même quantité de vert de rouge et de bleu, on obtient du gris échelonné entre le blanc et le noir selon la valeur plus ou moins importante choisie.

Pour une couleur aléatoire il suffit de mettre une valeur au hasard dans chaque champ en utilisant la fonction « rand() » que nous avons vue dans le chapitre Trois à propos de la fonction d'initialisation des nénuphs. De même que pour cette fonction d'initialisation nous utilisons l'opérateur modulo « % » qui donne le reste de la division par le dénominateur juxtaposé. Ce qui donne une écriture du genre :

```
RGB hasard ;
      hasard.r =rand()%64 ;
      hasard.g =rand()%64 ;
      hasard.b =rand()%64 ;
```

Rappelons que l'opérateur « . » (point) permet d'accéder aux champs d'une structure.
Pour chaque champ, une valeur entre 0 et 63 compris est retournée par la fonction « rand ».

Numérotation huit bits des couleurs

Dans le mode 256 couleurs, *une palette* regroupe 256 structures RGB dans un tableau de structures RGB. Dans ce tableau chaque couleur a une position qui correspond à un indice du tableau compris entre 0 et 255. La couleur d'un pixel est définie par cet indice. Comme il n'y a que 256 couleurs cet indice peut être codé sur 8 bits. C'est pourquoi, en 256 couleurs, chaque position de l'image, chaque pixel, a une valeur de couleur en 8 bits avec une définition très fine des couleurs, comme en 16 bits, avec 6 bits par couleur primitive. Mais si la qualité des couleurs est préservée, elles ne peuvent être que 256 simultanément dans le programme, à la différence du mode 16 bit qui en admet des milliers.

Cette nécessité d'un travail sur des ensembles de 256 couleurs simultanées maximum est véritablement une qualité du point de vue de la créativité qui peut s'y attacher. Le mode 256 couleurs est vraiment un mode graphique. Des images en vraies couleurs ramenées à 256 couleurs peuvent prendre des allures de peinture. Pour obtenir des ensembles de 256 couleurs, il est nécessaire d'imaginer des modalités automatiques de mélanges de rouge de vert et de bleu. Ce sont autant de robographes orientés vers la génération de couleurs.

Chapitre 10

Palettes de nombres

L'impressionnisme, qui oppose un rouge à un vert, ne colore pas ; il construit. Cette construction délicate ne peut jouer deux rôles à la fois ; elle donne « gris ». Seul, par exemple, un bleu pur reste bleu s'il voisine avec un gris ou un ton non complémentaire. S'il s'oppose à un orange, le rapport devient constructif mais ne colore pas.

Trop de couleur, pas de couleur.

Fernand Léger

Le chapitre précédent introduit le pixel sous l'angle de la couleur et des nombres qui s'y attachent. Le pixel contient un « nombre à couleurs » et ce nombre à couleurs exprime un mélange de rouge de vert et de bleu. Mais un programme dispose généralement de nombreuses couleurs simultanément et le chapitre précédent nous a laissé en suspens sur la notion de palette numérique que nous allons approfondir.

La première question que nous abordons est de savoir sous quelle forme le concept de palette peut apparaître dans un environnement de programmation. Ensuite, vient la problématique du mélange numérique des couleurs pour obtenir une palette. Nous nous sommes appuyés sur l'idée de dégradé de couleurs et proposons cinq possibilités pour la création de palettes numériques dans un programme.

Dans la perspective robographique, l'objectif est double. Le premier est d'introduire des enjeux qui gravitent autour de la création de couleurs, particulièrement en mode 256 couleurs. Des algorithmes sont présentés, accompagnés des informations nécessaires à leur compréhension. Mais les algorithmes présentés sont à voir également dans la forme graphique qu'ils peuvent prendre. Il y a une identité graphique dans l'écriture d'un robographe. Sa présentation fait aussi état du raisonnement dont il est le support. Le second objectif est ainsi d'attirer l'attention sur l'identité graphique que peut prendre l'écriture d'un robographe c'est-à-dire également sur des différences de style que l'on observe avec des auteurs différents.

1. Définir une palette numérique

Une palette est un ensemble de couleurs prédéfinies. Dans le mode 256 couleurs, 256 structures RGB sont regroupées dans un tableau ce qui fait que chaque position du tableau correspond à une couleur particulière. Les structures RGB éléments du tableau sont indicées de 0 à 255 et chacune est accessible via son indice. Lorsque l'indice prend la forme d'une variable incrémentée régulièrement dans une boucle, toutes les positions du tableau peuvent être visitées successivement.

1.1 Codage de la palette

Dans l'environnement Allegro, un type a été créé spécialement pour représenter une palette. Il est défini de la façon suivante :

```
(1)     #define PAL_SIZE     256
(2)     typedef  RGB  PALETTE[PAL_SIZE];
```

#define, introduction aux directives

(1) Dans le langage C « #define » est ce que l'on appelle une « macro ». Les macros font partie des « directives ». #include, #if, #ifndef, #ifdef, #else, #endif, #undef sont d'autres directives. Elles sont introduites par « # » au début d'une ligne et se terminent en fin de ligne. Au besoin « \ » ajouté comme dernier caractère de la ligne indique que la ligne se poursuit avec la ligne suivante. Du point de vue de la compilation les directives sont liées à un prétraitement par le préprocesseur dont l'objet est de préparer le code source pour le compilateur. C'est une première étape de la compilation. Elle fournit des portions de code au compilateur et également supprime tous les commentaires du « source » du programme [KERNIGHAN, 1995, p.88] [BRAQUELAIRE, 1998, p. 35, 157 à 175] [CHARBONNEL 1992, p. 41].

Le mot clé « define » définit un nom symbolique qui représente soit une valeur constante, soit une fonction, mais en fait n'importe quel texte de remplacement. On a une écriture du genre :
#define **nom** *texte de remplacement*

Et dans le programme à chaque fois que l'on appelle **nom** c'est *texte de remplacement* qui est substitué. La syntaxe de *texte de remplacement* est en fait indépendante du reste du langage ce qui permet des utilisations variées parfois astucieuses et surprenantes. Toutefois après remplacement du symbole **nom** par *texte de remplacement* dans le programme, le code résultant doit être correct syntaxiquement. Si *texte de remplacement* contient du code incomplet, il devra avoir été complété autour de **nom**, au moment de l'appel de **nom**.

Dans notre cas la macro signifie que PAL_SIZE est équivalent à 256. PAL_SIZE est une macro-constante selon la terminologie de Jean-Pierre Braquelaire. Dès lors que l'on met PAL_SIZE quelque part dans le source du programme c'est remplacé par 256. L'objectif est ici de déclarer un tableau de 256 structures RGB.

L'utilisation d'une macro-constante pour la déclaration d'un tableau est caractéristique. Les tests de parcours de tableaux dans des boucles for ou while[1], se font le plus souvent sur le nombre total des éléments du tableau ; du genre « tant que i < nombre d'éléments du tableau » où nombre d'éléments du tableau est représenté par une macro-constante. En cas de modification de la taille du tableau il suffit de modifier la valeur donnée à la macro-constante et il n'y pas besoin de corriger dans le source « nombre d'éléments du tableau » partout où il était nécessaire.

Ainsi, le premier avantage d'une macro est de pouvoir baptiser le plus clairement possible un repère important dans la construction du programme et de son raisonnement algorithmique. PAL_SIZE ou « taille palette », est plus explicite que 256 dans le programme car ce n'est pas n'importe quelle valeur 256, c'est celle de la taille de la palette. Un second avantage est qu'en cas de modification rendue nécessaire, pour une raison actuellement inconnue, il sera possible de modifier en une seule fois cette valeur de PAL_SIZE partout où PAL_SIZE est utilisée dans le programme, juste en lui affectant une nouvelle valeur avant de recompiler le programme.

[1] Le fonctionnement de la boucle « while » est détaillé au chapitre 12.

Création du « type PALETTE »

(2) Grâce à l'utilisation du typedef rencontré à propos de la structure RGB dans le chapitre Neuf, l'objet « PALETTE » devient un nom pour le type de données « tableau de 256 structures RGB ». Suite à cette définition de type, pour avoir une palette dans un programme il suffit de déclarer une palette comme n'importe qu'elle variable, à savoir une formule comme : « **type** *nom pour la variable* ; ». Si la palette est nommée par exemple « pal » ça donne simplement quelque part dans le programme :

```
PALETTE     pal ;
```

où « pal » est un tableau de 256 structures RGB.

1.2 Initialiser une palette

Maintenant que nous avons un type PALETTE qui représente un tableau de 256 structures RGB, il reste que le mélange de chaque couleur, la quantité de rouge, de vert, de bleu, doit être fait dans le programme. Certainement au début, mais éventuellement la palette peut être modifiée pendant le déroulement du programme. Or il est à peu près impossible de créer une palette numérique entièrement « à la main » en réalisant et testant chaque mélange de rouge vert et bleu individuellement. Il y a 64^3, soit 262144 couleurs possibles à répartir et agencer par ensembles de 256, et qui plus est, l'ordre dans lequel se suivent les couleurs peut avoir son importance dans la palette.

Pour avoir une palette de couleur, il faut concevoir le robot, la séquence algorithmique qui fabrique la palette des couleurs. *Créer des couleurs passe par un robographe qui se charge de générer un ou plusieurs ensembles de couleurs avec des caractéristiques telles qu'il soit jugé intéressant.*

Il n'y a probablement pas de limite à l'imagination de robographes qui initialisent chaque champ des 256 structures RGB. L'exemple de la palette de nuances de gris met en évidence des mécanismes de base liés à l'initialisation d'une palette.

1.3 Exemple d'une palette en nuances de gris

On veut fabriquer une palette avec un dégradé de gris du noir au blanc. Pour obtenir un gris il suffit d'avoir autant de rouge que de vert et de bleu. Plus cette même valeur pour les trois couleurs va augmenter, plus le gris sera clair et finalement avec la valeur égale à 63 ce sera du blanc. Voici le détail de la fonction « nuance_de_gris » (figure 10.1) qui fabrique une telle palette :

(1) Déclaration *externe* à la fonction d'une palette nommée « pal », de type PALETTE. C'est-à-dire un tableau de 256 structures RGB sur lequel la fonction nuance_de_gris va travailler.

Visibilité des variables locales ou externes

La déclaration est externe parce qu'une variable peut être déclarée de façon interne ou externe à une fonction. Une variable déclarée à l'intérieur d'une fonction est locale à cette fonction. Elle est créée automatiquement à l'appel de cette fonction et n'est visible que dans le cadre de cette fonction. Elle disparaît, et ses valeurs avec elle, dès que le programme sort de la fonction. Mais il est possible de définir des variables dites externes qui ne sont affiliées à aucune fonction en particulier mais sont accessibles par n'importe quelle fonction du programme. Une telle variable reste permanente dans le programme. Chaque fonction pouvant lui attribuer une valeur ou lire sa valeur c'est un moyen de communiquer des valeurs de fonction à fonction. Les variables externes sont déclarées une seule fois en dehors de toute fonction, généralement au début d'un fichier source[1]. [KERNIGHAN, 1995, p. 30, 31, 40].

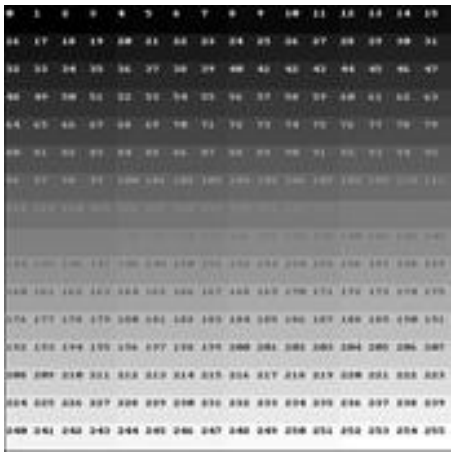
Sachant que nous aurons besoin de la palette pour les autres fonctions d'initialisation de palette que nous proposons, cette palette est déclarée comme « globale » une fois pour toutes, et les autres fonctions pourront s'y référer. Toutes les fonctions que nous proposons utiliseront la même palette « pal » comme si nous étions dans un programme.

```
(1)  PALETTE pal ;

(2)  void nuance_de_gris( ){
(3)  int i, v ;
(4)       for(i=0, v=0 ; i<PAL_SIZE ; i++){
(5)                 pal[ i ]. r = v;
(6)                 pal[ i ]. g = v;
(7)                 pal[ i ]. b = v;

(8)                 if( i%4 == 3)
(9)                       v++;
          }
     }
```

Fig. 10.1 : (a) Une fonction pour créer une palette de nuances de gris



(b) Palette résultat du dégradé de gris, numérotée de 0, noir, à 255, blanc

(2) Identification de la fonction nommée « nuance_de_gris ». *void* à gauche signifie qu'elle ne renvoie rien. La liste entre parenthèse « ( ) » à droite est vide ce qui indique qu'elle ne prend aucun paramètre en argument.

(3) Déclaration de deux variables locales à la fonction et de type int nommées « i » et « v ».

(4) Boucle for que nous avons rencontrée également au chapitre Trois, juste après le concept de tableau, à propos de l'initialisation des nénuphs.

[1] Pour être plus précis au sujet des déclarations de variables, deux aspects sont à dégager : d'une part la profondeur de la déclaration et d'autre part la portée d'une déclaration. La profondeur d'une déclaration est relative à l'emplacement de la déclaration (externe à toute fonction et bloc d'instructions, ou au contraire interne à une fonction ou un bloc d'instruction dans une fonction). La portée des définitions précise la visibilité de l'objet qui peut être relative à tout le programme, seulement au niveau d'un fichier, d'une fonction, d'un bloc d'instruction, d'un sous-bloc d'instruction. A ce sujet consulter [BRAQUELAIRE, 1998, p. 72], [KERNIGHAN, 1995, p. 83].

La première partie, avant le premier point virgule, initialise les variables i et v sur des valeurs de 0. La seconde, avant le second point virgule, correspond au test d'arrêt de la boucle. Tant qu'il est vrai, les instructions du corps de la boucle sont exécutées. En l'occurrence tant que i est inférieur à PAL_SIZE, c'est-à-dire tant que i est inférieur à 256, les instructions sont exécutées. La troisième partie de la boucle for correspond à la phase d'incrémentation. L'incrémentation est de une unité : i = 0, puis 1, puis 2, puis 3 etc. jusqu'à 256 et là, la boucle prend fin lorsque le test est faux, avec i = 256 qui n'est pas inférieur à 256.

(5) (6) et (7) Ce sont les trois premières instructions du corps de la boucle for. Chacune affecte à un champ de la i-éme structure RGB la valeur v. Pour la première structure RGB, celle de l'indice 0, lorsque i est égale à 0, v est aussi égal à 0. Ainsi pal[0].r, pal[0].g et pal[0].b sont tous trois égals à 0 ; en conséquence pal[0] correspond à noir.

(8) et (9) Pour avoir des couleurs différentes il faut que v change de valeur au fur et à mesure que i augmente et que chaque structure RGB du tableau est passée en revue. Pour avoir un gris, rouge, vert et bleu prennent une valeur v identique. Pour changer de gris, en partant du noir et en avançant graduellement vers le blanc, il faut que la valeur de v augmente. Lorsque v sera égal à 63, on aura du blanc.

Sachant qu'il y a 256 couleurs dans la palette une question à voir est comment faire augmenter v. Dans l'algorithme proposé l'idée est d'augmenter v de 1 toutes les quatre positions dans la palette, ce qui permettra d'avoir 64 gris répartis de quatre en quatre sur la palette. Pour ce faire v est incrémenté de 1 si et seulement si i modulo 4 est égal à 3. Ce qui donne, pour :

i = 0, pal[0] prend 0, v reste à 0	i = 4, pal[4] prend 1, v reste à 1	i = 8, pal[8] prend 2, v reste à 2
i = 1, pal[1] prend 0, v reste à 0	i = 5, pal[5] prend 1, v reste à 1	i = 9, pal[9] prend 2, v reste à 2
i = 2, pal[2] prend 0, v reste à 0	i = 6, pal[6] prend 1, v reste à 1	i = 10, pal[10] prend 2, v reste à 2
i = 3, pal[3] prend 0, v passe à 1	i = 7, pal[7] prend 1, v passe à 2	i = 11, v passe à 3 etc.

Notons que lorsque (i %4 == 3) est vrai, avant que v augmente de 1, v est d'abord affectée aux champs de la i-éme structure RGB du tableau. Par exemple au début de la boucle, lorsque i = 3, la valeur de v est toujours égale à 0. C'est-à-dire qu'à la position pal[3], le rouge, le vert et le bleu sont égaux à 0. La modification de v sera effective pour les quatre positions suivantes.

Principes retenus

En conclusion et afin d'aborder la couleur retenons que la fonction « nuances_de_ gris » s'appuie sur trois principes. Le premier principe est celui de l'ordre dans lequel toutes les positions de la palette sont parcourues. « Nuances_de_gris » est parcourue de la position 0 à la position 255. Le second principe est celui du dégradé proprement dit. Il s'agit du pas d'avancement qui augmente régulièrement les valeurs de chaque primitive de couleur à chaque nouvelle position de la palette. Dans la fonction « nuance_de_gris » c'est aux lignes (8) et (9) ce qui concerne l'incrémentation de 1 de la variable v. Le troisième principe est celui de la valeur initiale attribuée aux primitives de couleur de la première couleur. La première couleur est la première position de la palette à partir de laquelle le dégradé va se construire. C'est la position « pal[0] » dans « nuances_de_gris ».

Les quatre fonctions que nous proposons maintenant conservent le premier principe de « nuances_de_gris ». Chacune d'entre elles a une boucle for qui fait défiler les positions de la palette dans l'ordre où elles sont indicées. Toutefois la couleur de la première position d'indice 0 est parfois mise à part et le dégradé est construit à partir de la position suivante, de 1 à 255. Mais surtout, les deux autres principes font l'objet de plusieurs interprétations et expérimentations.

2. Quatre dégradés de nombres à couleurs

Nous avons deux paramétrages à mettre en scène dans nos fonctions. Ce sont ceux des pas d'avancement de chaque primitive de couleur, conjugués avec l'initialisation des primitives de couleur de la première couleur.

Fonctionnement du pas d'avancement

A partir d'une valeur initiale, le résultat de l'accumulation par addition ou soustraction d'un pas d'avancement détermine une suite pour chaque primitive de couleur. Cette suite est l'évolution de la primitive du début à la fin de la palette. Chaque primitive peut avoir un pas identique, comme dans le cas de la fonction « nuances_de_gris », ou différent. Il peut être constant ou périodique. Eventuellement le pas d'une primitive peut être nul avec une progression nulle. Notons également que le comportement du pas d'avancement doit permettre à la variable de primitive de couleur de rester dans la fourchette des valeurs de couleurs de 0 à 63 comprises. Au final les trois suites des primitives sont juxtaposées et donnent tous les mélanges qui parcourent la palette avec chaque structures RGB de la première position à la dernière.

Propriétés de dégradé

Le fonctionnement du principe de pas d'avancement confère des propriétés au dégradé. Nous le qualifions soit de constant, à constance variable ou de ponctuellement inconstant. Nous considérons qu'il est constant lorsque les valeurs absolues de chacun des trois pas d'avancement restent identiques à chaque appel de la fonction et pendant le déroulement de la fonction. Il est dit à constance variable lorsque les valeurs absolues de chacun des trois pas d'avancement sont aléatoirement différentes à chaque appel de la fonction mais restent identiques pendant le déroulement de la fonction. Il est dit ponctuellement inconstant lorsque le pas d'avancement d'une primitive peut changer de valeur entre deux positions de la palette, pendant le déroulement de la fonction.

Couleurs de dégradé

Ensuite il y a la couleur du dégradé. Essentiellement nous pensons à l'origine du dégradé c'est-à-dire les valeurs initiales de la première couleur. Sur ce point nous proposons principalement deux types d'initialisation. L'origine du dégradé peut être fixe ou variable.
Lorsque l'origine est fixe les valeurs initiales de la première couleur sont toujours les

mêmes à chaque appel de la fonction. Si l'origine est variable, l'initialisation donne des valeurs différentes à chaque appel de la fonction.

2.1 Dégradé constant de couleur constante

La palette obtenue avec la fonction « nuances_de_gris » sera toujours la même. Il n'y a pas de variations prévues. L'accroissement de la valeur de couleur pour chaque champ est constant. La variable v est incrémentée de un toutes les quatre positions, commence à 0 et finit à 63. La couleur de départ sera toujours le noir ainsi que celle d'arrivée toujours le blanc. Dans une optique identique une palette de couleurs peut aussi être créée.

Le pas d'avancement est fixé de la façon suivante. Chaque primitive fait le même parcours. Elle est incrémentée de une unité de 0 à 63 puis elle est décrémentée de une unité de 63 à 0. Mais chaque parcourt est décalé par rapport aux deux autres. Avant de débuter et après l'avoir terminé la variable de primitive de couleur est initialisée avec une valeur de 0.

Le parcourt du rouge, premier champ de la structure RGB, commence à la première position de la palette « pal[0] ». Les deux autres primitives sont bloquées avec une valeur unique de 0. Le parcourt du vert commence lorsque le rouge atteint la valeur de 63. Lorsque le vert atteint la valeur 63, le rouge est bloqué à 0 et le bleu commence son parcourt. Lorsque le bleu arrive à 63 le vert est bloqué à 0. Le dégradé obtenu se traduit par une suite constante dont la couleur sera toujours la même. De ce fait, chaque exécution de la fonction reproduira toujours la même palette.

Voici le détail de la fonction « degrade_constant » initialement écrite par Shawn Hargreaves[1] (figure 10.2) :

(1) Identification de la fonction « degrade_constant ». *void* à gauche signifie qu'elle ne renvoie rien. La liste entre parenthèse « ( ) » à droite est vide ce qui indique qu'elle ne prend aucun argument.

(2) Déclaration de la variable nommée « c », locale à la fonction et de type int.

(3) Première boucle for. La variable c permet de parcourir les positions allant de 0 à 63 dans le tableau de structures RGB qu'est la palette « pal ». A chaque incrémentation de c on passe à la position suivante dans le tableau et une valeur est affectée à chacun des trois champs de la palette pal. Les valeurs de vert et de bleu restent toujours à 0. En revanche la valeur de rouge croît avec c de 0 à 63.

(4) Seconde boucle for qui examine les 64 positions suivantes allant de 64 à 127. Cette fois le rouge décroît de 63 à 0. En revanche c'est le vert qui simultanément va croître de 0 à 63. Le bleu reste toujours à 0

[1] Shawn Hargreaves est la personne initialement à l'origine de la communauté de développement Allegro. Cette fonction est tirée d'un exemple d'utilisation de la librairie Allegro : « expal.c » dans le dossier « examples » après installation.

(5) Troisième boucle for qui examine les positions allant de 128 à 191. Le rouge reste sur 0. Le vert décroît de 63 à 0 et le bleu simultanément s'accroît de 0 à 63.

(6) Quatrième boucle for qui examine les positions allant de 192 à 255. Rouge et vert restent sur 0. Le bleu décroît de 63 à 0.

```
(1)  void degrade_constant ( ){
(2)  int c;
(3)       for (c=0; c<64; c++) {
                  pal[c].r = c;       // rouge de 0 à 63
                  pal[c].g = 0;
                  pal[c].b = 0;
          }
(4)       for (c=64; c<128; c++) {
                  pal[c].r = 127-c; // rouge de 63 à 0
                  pal[c].g = c-64;  // vert  de 0 à 63
                  pal[c].b = 0;
          }
(5)       for (c=128; c<192; c++) {
                  pal[c].r = 0;
                  pal[c].g = 191-c; // vert de 63 à 0
                  pal[c].b = c-128; // bleu de  0 à 63
          }
(6)       for (c=192; c<256; c++) {
                  pal[c].r = 0;
                  pal[c].g = 0;
                  pal[c].b = 255-c; // bleu de 63 à 0
          }
     }
```

Fig. 10.2 : Un algorithme de Shawn Hargreaves pour la création d'une palette avec dégradé constant de couleurs constantes

La palette résultante peut être illustrée avec un diagramme (figure 10.3):

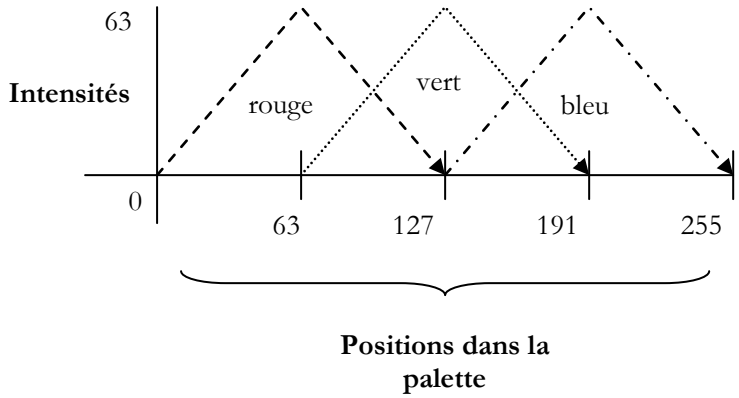


Fig. 10.3 : Mélange des couleurs par la fonction « degrade_constant »

Ni rouge ni bleu ne se mélangent. Mais le vert sert à l'articulation des deux. Au résultat on a trois séquences de dégradé rouge, vert et bleu avec un fondu de rouge à vert et de vert à bleu. Le bleu final revient au noir ce qui permet éventuellement de créer un cycle en revenant au début et en repartant sur le rouge.

2.2 Dégradé constant de couleur variable

Procédons maintenant à la création d'une fonction unique qui produise une variété de palettes de couleurs différentes. La fonction « degrade_variable » génère des palettes de couleurs différentes mais toutes créées selon le même modèle. Si les couleurs sont différentes à chaque palette, la cohérence établie entre toutes les couleurs est, du point de vue algorithmique, toujours la même.

Le dégradé est constant. L'incrémentation des primitives de couleur est toujours identique d'une couleur à la suivante. Ce sont des pas de 1 ou de -1 selon que l'on avance vers 63 où que l'on descend sur 0. Dès qu'est atteinte une de ces deux bornes la valeur du pas est inversée. D'une addition de 1 il passe à la soustraction – 1 et inversement. L'initialisation du pas d'avancement pour chaque primitive de couleur est aléatoirement une valeur de 1 ou de -1 tirée au début de la fonction.

Les valeurs attribuées aux primitives de rouge, de vert et de bleu de la première couleur, à partir de laquelle se développe le dégradé, sont sélectionnées aléatoirement entre les bornes 0 et 63. Ce sont surtout ces valeurs tirées initialement qui font les palettes différentes. Si les valeurs de départ étaient toujours les mêmes il n'y aurait que 2^3 palettes possibles avec les seules variations liées à l'initialisation des pas d'avancement. Là il y a pour chaque alignement des pas d'avancement 64^3 possibilités d'initialisation des couleurs soit au total $2^3 * 64^3 = 2\ 097\ 152$ cas.

En dépit du grand nombre de cas de cette combinatoire, l'unité de méthode pour l'obtention des couleurs dans le programme fait qu'il est difficile de privilégier une palette en particulier. On a envie pendant l'exécution du programme de tester des palettes différentes. La méthode d'obtention de palettes est mise en avant. C'est la permanence de l'agencement de couleurs qui ressort, et les différents ensembles de couleurs obtenus en font la description. C'est-à-dire finalement que la couleur est l'expression dans le programme d'un moyen d'accéder à des couleurs. Et la cohérence des ensembles de couleurs, c'est-à-dire la méthode de génération des ensembles, devient la couleur réelle du programme. En d'autres termes *la couleur d'un programme est celle du robographe qui est à l'origine d'un ensemble de couleurs.*

La fonction « degrade_variable » se présente de la façon suivante (figure 10.4) :

(0) Définition de la macro « irand(n) ». Partout dans le programme où « irand » est appelée, elle est remplacée par le texte « (rand()%(n)) » et « n » est un paramètre passé en argument. Quand « irand » est appelée « n » est remplacé par une variable ou une valeur numérique qui sert à définir la fourchette dans laquelle doit se trouver le nombre à renvoyer. Par exemple irand(62) est remplacé par (rand()%62). La fonction « rand() » qui fait partie de la librairie standard <stdlib.h> renvoie un entier selon une suite aléatoire et « %62 » donne le reste de la division de ce nombre par 62, c.à.d une valeur comprise entre 0 et 61.

(1) Identification de la fonction « degrade_constant ». *void* à gauche signifie qu'elle ne renvoie rien. La liste entre parenthèse « ( ) » à droite est vide ce qui indique qu'elle ne prend aucun argument.

(2) Déclaration des variables nommées respectivement « i », « rouge », « pr », « vert », « pv », « bleu » et « pb ». Ces variables sont locales à la fonction et de type int.

(3) A la différence des deux premières fonctions la première couleur d'indice 0 dans le tableau de la palette « pal » est forcée au noir. Les trois champs de la première structure RGB du tableau sont initialisés à 0.

(4) Les trois variables « rouge », « vert » et « bleu » sont utilisées pour déterminer chacune des valeurs de rouge de vert et de bleu dans le mélange RGB. Chacune est tout d'abord initialisée avec une valeur prise au hasard dans l'intervalle 1 à 62, bornes comprises.

```
(0) #define irand(n) (rand()%(n))

(1) void degrade_variable ( ){
(2) int i,rouge,pr,vert,pv,bleu,pb;

(3)       pal[0].r=pal[0].g=pal[0].b=0;

(4)       rouge = irand(62)+1;
          vert = irand(62)+1;
          bleu = irand(62)+1;

(5)       pr = 2*irand(2)-1;
          pv = 2*irand(2)-1;
          pb = 2*irand(2)-1;

(6)       for(i=1;i<255;i++){

(7)                 rouge += pr;
                    vert += pv;
                    bleu +=pb;

(8)                 pal[i].r = rouge;
                    pal[i].g = vert;
                    pal[i].b = bleu;

(9)                 if( rouge <=0 || rouge >= 63)
                              pr*=-1;
                    if( vert <=0 || vert >= 63)
                              pv*=-1;
                    if( bleu <=0 || bleu >= 63)
                              pb*=-1;
          }
    }
```

Fig. 10.4 : Une fonction pour créer une palette avec dégradé constant de couleurs variables

(5) Initialisation des variables pr, pv, pb. Ces variables vont définir le pas d'avancement des valeurs des couleurs, pr pour le rouge, pv pour le vert, pb pour le bleu. Dans cette fonction elles ne peuvent prendre que deux valeurs : 1 ou -1. En effet, rand()%2 est égal soit à 0 soit à 1. Multiplié par deux ça donne soit 0 soit 2 et en soustrayant 1 on obtient soit -1 soit 1.

(6) Boucle for qui permet de parcourir la palette pal, un tableau de 256 structures RGB, à partir de la position 1 jusqu'à la position 255 par l'intermédiaire de l'incrémentation de un en un de la variable i. La position 0 n'est pas prise en compte dans la boucle et conserve les valeurs de 0 qui lui ont été associées en (3).

(7) Les trois premières instructions dans le corps de la boucle modifient la valeur des variables « rouge », « vert » et « bleu » correspondant aux primitives du mélange RGB. En fonction de la valeur du pas d'avancement qui leur est à chacune associé, respectivement pr pour le rouge, pv pour le vert et pb pour le bleu, elles sont soit augmentées de un soit diminuées de un.

(8) A la i-iéme position dans la palette chaque champ de la structure RGB reçoit la valeur contenue dans la variable qui lui est attribuée.

(9) Il reste que les valeurs de rouge vert et bleu doivent rester dans la fourchette entre 0 et 63 compris. L'augmentation et la diminution des valeurs de ces variables par pas de 1 ou de -1 fait que ces variables vont atteindre et dépasser soit 0 soit 63.
Pour chacune de ces variables un test est fait afin de déterminer s'il y a ou pas dépassement de la fourchette. Le dépassement peut avoir lieu dans un sens OU dans l'autre et il suffit qu'une seule des deux conditions soit vraie pour que le test soit vrai. Si l'on arrive à 0 c'est du fait d'une diminution répétée de la variable par pas de -1. Si l'on arrive à 63 c'est du fait de l'addition répétée de pas de 1. Dans les deux cas, si le test est vrai, il suffit d'inverser la marche et pour ce faire multiplier par -1 la valeur du pas d'avancement (s'il est égal à -1 il passe à 1 et sinon il passe à -1)

A la différence des deux premières fonctions que nous avons présentées, chaque appel de la fonction « degrade_variable » produira une palette différente. Chaque valeur de rouge, de vert ou de bleu débute sur une valeur attribuée au hasard dans la fourchette. Est décidé au hasard également si le pas d'avancement est positif ou négatif. Les rapports entre ces initialisations sont aléatoires cependant la contrainte constante d'un dégradé par pas de 1 ou de –1 assure une certaine forme de continuité entre toutes les palettes que la fonction permet d'obtenir.

2.3 Dégradé avec variation de constance sur couleurs variables

Deux modifications mineures de la fonction précédente permettent la génération d'une nouvelle espèce de palettes. Le pas d'avancement est dans une fourchette plus large, entre – 5 et 5, toujours soumis à un tirage aléatoire. La variabilité des couleurs est fondée sur la combinaison aléatoire des pas d'avancement avec l'attribution aléatoire d'une valeur initiale cette fois comprise entre 5 et 58 pour chaque primitive. Notons la possibilité pour un pas d'avancement d'être égal à 0. Dans ce cas la primitive de couleur qui en dépend n'évoluera pas et restera la même pour tous les mélanges de la palette. Si par hasard les trois sont égales à 0, il n'y aura qu'une seule couleur dans la palette.

Cette fonction est très proche de la précédente et nous en profitons pour modifier la présentation des différentes opérations. Il est intéressant d'observer que les fonctions dans leur écriture prennent des formes au niveau du graphisme qui les exprime. C'est-à-dire que le raisonnement fait l'objet d'une répartition dans l'espace de la page qui est tout de même

très liée à l'esprit de celui ou celle qui programme, à la vision qu'il a de son propre raisonnement, à la façon dont il l'organise dans le cadre de la syntaxe du langage de programmation.

La fonction « degrade_pas_variable » se présente de la façon suivante (figure 10.5) :
(1) Identification de la fonction « degrade_pas_variable ». La fonction ne renvoie rien et ne prend aucun argument.

(2) Déclaration des variables locales à la fonction.

```
(1)  void degrade_pas_variable( ){
(2)  int i,rouge,pr,vert,pv,bleu,pb;

(3)        pal[0].r=pal[0].g=pal[0].b=0;

(4)        pr = irand(11)-5;
(5)        rouge = irand(54)+5;

           pv = irand(11)-5;
           vert = irand(54)+5;

           pb = irand(11)-5;
           bleu = irand(54)+5;

(6)        for( i=1; i<256; i++ ){

(7)                rouge += pr;
(8)                if( rouge <=ABS(pr) || rouge >= 63-ABS(pr))
                           pr*=-1;
(9)                pal[i].r = rouge;

(10)               vert += pv;
(11)               if( vert <=ABS(pv) || vert >= 63-ABS(pv))
                           pv*=-1;
(12)               pal[i].g = vert;

(13)               bleu +=pb;
(14)               if( bleu <=ABS(pb) || bleu >= 63-ABS(pb))
                           pb*=-1;
(15)               pal[i].b = bleu;

           }
}
```

Fig 10.5 : Une fonction pour créer une palette avec dégradé à définition variable de constance et à propos de couleurs variables

(3) La première couleur de la palette pal est forcée au noir avec chaque champ de la structure RGB à 0.

(4) Première modification d'avec la fonction « degrade_palette » précédente. Les variables qui correspondent aux trois pas d'avancement pr, pv, pb pour les primitives de couleur représentées respectivement par les variables « rouge », « vert » et « bleu » sont initialisées avec des valeurs comprises entre 5 et –5. La macro « irand(11) » revient à la fonction (rand()%11) qui renvoie une valeur entre 0 et 10 compris. Soustraire 5 donne une fourchette de – 5 à 5, et la valeur 0 est comprise.

(5) La seconde modification est l'ajustement de la valeur de départ de chacune des variables de couleur « rouge », « vert » et « bleu ». Il y a un léger rétrécissement de la fourchette qui s'étire entre 0 plus le pas d'avancement, et 63 moins le pas d'avancement. La macro « irand» avec 54 en argument renvoie une valeur comprise entre 0 et 53. Si l'on ajoute 5 on a une valeur comprise entre 5 et 58. C'est une précaution afin d'être sûr qu'après ajout du pas d'avancement, la valeur de couleur va rester, du point de vue des bords de la fourchette, entre 0 et 63.

(6) La boucle for qui passe en revue chaque position i dans la palette et permet d'initialiser chaque structure RGB.

(7) (10) (13) La variable de couleur est additionnée du pas d'avancement qui lui correspond.

(8) (11) (14) Test de vérification que la variable de couleur est dans l'intervalle 0 plus la valeur absolue du pas d'avancement et 63 moins la valeur absolue du pas d'avancement. Cette restriction sur les bords s'explique par le fait qu'à la position suivante de la palette la variable sera une fois de plus additionnée du pas d'avancement et qu'elle devra toujours rester dans la fourchette voulue. C'est donc en prévision de cette addition que le test est construit. Si la valeur dépasse d'un côté comme de l'autre, le pas d'avancement est inversé.

(9) (12) (15) Chaque champ de couleur de la structure RGB se voit affecté la valeur de la variable de couleur qui lui correspond.

Organisation graphique du code source

Dans la fonction précédente, l'organisation spatiale de l'algorithme, c'est-à-dire sa disposition graphique, est faite par des regroupements autour des différents traitements. Dans cette version l'organisation graphique privilégie la notion de couleur comme axe de regroupement des instructions. Chaque couleur est abordée séparément avec la totalité des traitements qui lui correspondent. Cette aspect est important dans la mesure où le regroupement d'instructions détermine finalement la constitution des fonctions.

Dans un cadre plus sophistiqué, ces deux approches pourraient déterminer des sous-fonctions différentes. Les unes construites autour de l'identité des traitements, avec pour chaque traitement une fonction. Les autres plutôt constituées autour d'un objet donné et de la diversité des traitements qui lui sont propres. Par exemple comme il faut répéter les mêmes opérations pour le vert le rouge et le bleu une fonction unique pourrait être appelée trois fois et, moyennant un passage d'argument adéquat, permettrait de traiter chaque couleur séparément. Mais dans notre contexte ça ne paraît pas très utile.

Le dernier exemple que nous proposons pour la création de palettes de couleurs autour de l'idée de dégradé nous a été fourni par Robin Fercoq, artiste informaticien qui travaille également en langage C. Nous le restituons ici dans sa forme graphique initiale, sans toucher à la mise en page qui est voulue telle quelle par l'auteur.

2.4 Dégradé ponctuellement inconstant à partir d'une origine fixe

Le dispositif de Robin Fercoq permet d'élargir les possibilité de mélange des couleurs primitives dans la mesure où une série de 0 peut stopper la progression ou la dégression d'une couleur de ce mélange. En ce sens nous avons baptisé le dégradé de ponctuellement inconstant. En revanche les primitives de la première couleur sont toujours initialisées avec les même valeurs.

La fonction « neonpal » se présente de la façon suivante (figure 10.6) :

(1) Identification de la fonction « neonpal ». Cette fonction ne renvoie pas de valeur (void) et ne prend aucun argument comme l'indiquent les parenthèses vides.

(2) Déclaration des variables « d », « dd », « c », « v » locales à la fonction. « c » et « v » sont deux tableaux de trois entiers chacun et de type int. En fait le tableau « c[ ] » regroupe trois variables qui correspondent aux variables « rouge », « vert » et « bleu » vues dans les algorithmes des deux fonctions précédentes. C'est une autre façon de représenter la même idée. De même, « v[ ] » regroupe l'équivalent des variables pr, pv et pb.

Du fait de cette différence de structure de données le fonctionnement de la fonction, du point de vue de son écriture, va différer de celui des deux fonctions précédentes, alors même que le résultat pourrait facilement être identique sur le plan conceptuel.

```
(1)     void neonpal(){
(2)     int d,dd,c[3],v[3];
(3)       for(dd=0;dd<3;dd++){
            v[dd]=irand(3)-1;
            c[dd]=0;}
(4)       for (d=0;d<256;d++){
            pal[d].r=c[0];
            pal[d].g=c[1];
            pal[d].b=c[2];
(5)         for(dd=0;dd<3;dd++){
              c[dd]+=v[dd];
(6)           if (c[dd]<0){c[dd]=0;v[dd]=1;}
(7)           else if (c[dd]>63){c[dd]=63;v[dd]= -1;}
(8)           else if (!irand(32)) v[dd]=irand(3)-1;
      }
    }
  }
```

Fig 10.6: Un algorithme de Robin Fercoq pour la création de palettes colorées

(3) La première boucle for initialise les tableaux c[ ] et v[ ], chaque élément du tableau reçoit une valeur. Dans le tableau v[ ] cette valeur est – 1, 0 ou 1. Ce sont les valeurs destinées à être les pas d'avancement pour chaque primitive de couleur. C'est à peu près l'équivalent de la ligne (5) dans la fonction « degrad_variable » (paragraphe 2.2) sauf qu'ici la valeur 0 est permise.

A la différence des deux fonctions précédentes, toutes les positions du tableau c[ ], à savoir les variables pour rouge, vert et bleu, reçoivent initialement la valeur 0. Il serait facile de le modifier. Mais, comme l'indique la ligne (4), l'idée est ici d'avoir 0 pour valeur de départ à chaque primitive de couleur, ce qui supprime la nécessité d'initialiser la première position de la palette en dehors de la boucle comme nous l'avons fait dans les deux fonctions précédentes aux lignes (3).

(4) Seconde boucle for qui parcourt une à une toutes les positions de 0 à 255 de la palette. Les trois premières instructions affectent respectivement les valeurs de c[0], c[1] et c[2] aux champs du rouge (**.**r) , du vert (**.**v) et bleu (**.**b) de la i-eme structure RGB de la palette. Effectivement lors de la première affectation toutes ces valeurs sont à 0.

(5) Troisième boucle for avec comme première instruction dans le bloc l'affectation des valeurs de pas d'avancement, rangées de 0 à 2 dans le tableau v[ ], à chacune des valeurs de couleur également rangées de 0 à 2 dans le tableau c[ ]. C'est l'équivalent de la ligne (7) dans la fonction « degrad_variable » et des lignes (7) (10) et (13) dans la fonction « degrad_pas_variable »

(6) et (7) Toujours dans le cadre de la troisième boucle for, c'est-à-dire pour chaque position des tableaux c[ ] et v[ ], il s'agit du test aux bornes de la fourchette de 0 à 63. Si une valeur de primitive de couleur du tableau c[ ] dépasse 0 elle est forcée à 0 et le pas d'avancement correspondant stocké dans v[ ] est inversé, il passe à 1. De même si une valeur de primitive de couleur du tableau c[ ] dépasse 63 elle est forcée à 63 et le pas d'avancement correspondant stocké dans v[ ] est inversé, il passe à – 1.

A propos des instructions de condition if, else if, else if …

Remarquons au passage que l'instruction de condition if-else que nous avons rencontrée dans le chapitre Sept à propos du moteur du programme Graphitos, est ici étendue à « if, else if, else if… », on pourrait en ajouter d'autres. Le principe est de pouvoir enchaîner une suite de if mais qui s'arrête dès qu'un test vrai est trouvé. Si par exemple « c[1] < 0 » est vrai alors le bloc d'instruction correspondant à la ligne (6) est effectué et les deux autres tests sont ignorés. Si au contraire « c[1] < 0 » est faux le test suivant est effectué. Est ce que « c[1] >63 » ? Si oui le bloc correspondant d'instruction est effectué, et si non on passe au test suivant et ainsi de suite jusqu'à épuisement des tests ou une réponse vraie. Ce qui fait que sur l'ensemble des trois tests, il ne pourra jamais y avoir qu'une seule réponse à la fois [KERNIGHAN, 1995, p. 56] [BRAQUELAIRE, 1998, p.140].

Si l'on ne met que des if en supprimant les else, tous les tests seront toujours effectués, ce qui représente ici éventuellement une perte de temps : si (c[1] < 0) est vrai il est certain que (c[1] > 0) est faux et ce n'est pas la peine de le vérifier. Mais surtout l'objectif est ici d'isoler le troisième test des deux précédents afin qu'il ne soit effectué que si les deux précédents sont faux.

(8) Cette dernière ligne est l'innovation principale de cet algorithme. L'idée est simple, de temps en temps, selon une valeur tirée au hasard, la valeur v[ ] du pas d'avancement de la position courante est réinitialisée et peut être éventuellement changée. Notamment elle peut prendre une valeur de 0 ce qui immobilise la progression de la variable de primitive de couleur correspondante. Si au contraire elle était déjà sur 0, il se peut qu'elle passe à 1 ou – 1 et permette à la variable correspondante de croître ou de décroître.

3. Réflexions liées aux styles de programmation

Des différences importantes caractérisent des programmes écrits par des personnes différentes. Elles s'observent à plusieurs niveaux. Différences dans les objectifs et la conceptualisation, différences dans les choix de structures de données et l'agencement des algorithmes, mais aussi différences dans l'écriture même du point de vue graphique. Précisons que les aspects graphiques de l'écriture restent indépendants de la syntaxe du langage de programmation et que la tolérance, à part quelques détails (notamment dans le cas des macros) est à cet égard pratiquement totale.

3.1 Introduction aux styles

Entre le premier algorithme de Shawn Hargreaves, celui de Robin Fercoq et les deux autres on sent une différence (figure 10.7). La fonction de Robin est compacte, non seulement dans sa forme visuelle mais également dans le choix des structures de données. Les variables de rouge, de vert, de bleu avec les trois pas d'avancement correspondants sont ramenés à deux variables sous la forme de deux tableaux dont les noms eux-mêmes sont réduits à la plus simple expression, une lettre minuscule. La fonction de Shawn quoi que le problème soit simple n'en demeure pas moins élégante. C'est clair, concis et sans adjonction de variable inutile. Le découpage des blocs d'instructions est explicite. Sans même en détailler le contenu on sait tout de suite de quoi il s'agit, jusqu'au détail du nom de la variable unique « c » pour color.

En ce qui nous concerne, nous n'hésitons pas à être plus explicite quitte à être parfois un peu lourd dans les noms de variables. Egalement nous n'hésitons pas à prendre de la place, à détailler les blocs, à ajouter des commentaires parfois même redondants. Surtout nous avons le sentiment d'une logique visuelle dans l'écriture de la fonction ; il y a une conjugaison entre une manière de voir et une manière de penser. D'ailleurs, au niveau des interfaces, l'informatique repose beaucoup sur ce que l'on pourrait appeler des raisonnements visuels. Nous essayons d'avoir et de mettre en évidence une vision du raisonnement à travers la répartition de l'algorithme dans l'espace graphique du texte source. Il arrive souvent que pour lire du code source le programmeur commence par le remettre en forme du point de vue du style. Ajoutons qu'il y a un certain plaisir à voir le « dessin » d'une fonction et à rentrer dedans pour la lire. Il y a du plaisir à lire du code source.

Du point de vue professionnel d'une ingénierie, l'optique est plutôt de définir des normes pour l'écriture afin qu'un programme commencé par une personne puisse être pris en cours de route par une autre, soit pour le modifier soit pour le continuer. Ce qui va relever d'un projet collectif va devoir faire l'objet d'une certaine normalisation de l'écriture. Dans le domaine de l'ingénierie les normes portent non seulement sur l'écriture mais également sur le raisonnement et l'approche générale des problèmes à traiter ; notamment en raison de questions de sécurité et de fiabilité du code source et des programmes compte-tenu des enjeux qui s'y attachent (bancaire, trains etc.).

```
void degrade_constant ( ){
int c;
    for (c=0; c<64; c++) {
            pal[c].r = c;
            pal[c].g = 0;
            pal[c].b = 0;
    }
    for (c=64; c<128; c++) {
            pal[c].r = 127-c;
            pal[c].g = c-64;
            pal[c].b = 0;
    }
    for (c=128; c<192; c++) {
            pal[c].r = 0;
            pal[c].g = 191-c;
            pal[c].b = c-128;
    }
    for (c=192; c<256; c++) {
            pal[c].r = 0;
            pal[c].g = 0;
            pal[c].b = 255-c;
    }
}
```

```
void neonpal(){
int d,dd,c[3],v[3];
  for(dd=0;dd<3;dd++){
    v[dd]=irand(3)-1;
    c[dd]=0;}
    for(d=0;d<256;d++){
       pal[d].r=c[0];
       pal[d].g=c[1];
       pal[d].b=c[2];
       for(dd=0;dd<3;dd++){
          c[dd]+=v[dd];
          if (c[dd]<0){c[dd]=0;v[dd]=1;}
          else if (c[dd]>63){c[dd]=63;v[dd]= -1;}
          else if (!irand(32)) v[dd]=irand(3)-1;
       }
    }
  }
```

```
void degrade_pas_variable( ){
int i,rouge,pr,vert,pv,bleu,pb;

        pal[0].r=pal[0].g=pal[0].b=0;

        pr = irand(11)-5;
        rouge = irand(54)+5;

        pv = irand(11)-5;
        vert = irand(54)+5;

        pb = irand(11)-5;
        bleu = irand(54)+5;

        for( i=1; i<256; i++ ){

                rouge += pr;
                if( rouge <=ABS(pr)|| rouge >= 63-ABS(pr))
                        pr*=-1;
                pal[i].r = rouge;

                vert += pv;
                if( vert <=ABS(pv) || vert >= 63-ABS(pv))
                        pv*=-1;
                pal[i].g = vert;

                bleu +=pb;
                if( bleu <=ABS(pb) || bleu >= 63-ABS(pb))
                        pb*=-1;
                pal[i].b = bleu;

        }
}
```

Fig 10.7 : Comparaison des fonctions du point de vue du style et de la disposition graphique

Dans des domaines plus informels comme celui représenté par la communauté Allegro à laquelle nous empruntons l'environnement de programmation, la norme portera surtout sur l'écriture elle-même et non sur l'approche des problèmes. Dans cette communauté, il y a non seulement une grande liberté de raisonnement, mais aussi une grande curiosité des raisonnements.

Remarquons toutefois que dans cette communauté de travail la normalisation de l'écriture est effectuée selon plusieurs niveaux. La création individuelle est totalement libre de son écriture. Mais plus elle entre dans une préoccupation collective plus elle nécessite quelques efforts de normalisation. En fin de chaîne, tout ce qui a trait au développement de la librairie Allegro elle-même fait l'objet d'une régulation rigoureuse et n'a rien à envier à une ingénierie ordinaire. Il est en effet essentiel pour tous les membres de la communauté de pouvoir appuyer leurs recherches personnelles sur un tronc commun de fonctions et d'algorithmes de qualité technique irréprochable et fiable. Comme le développement est collectif, des développeurs différents doivent pouvoir se compléter et se comprendre dans un ensemble de code source assez considérable.

3.2 Exemple de recommandations

Un programme sans la moindre rigueur est peu lisible, peu fiable, difficile à corriger etc. Il devient vite impossible à développer et se trouve finalement figé dans son incompréhensibilité. C'est-à-dire que le style est en relation avec l'efficacité de la conception. Souvent le style de programmation est associé à des conseils de programmation. Ces conseils peuvent aller jusqu'à la mise en place d'une véritable méthodologie [BRAQUELAIRE, 1998]. En général le style rejoint une certaine forme de discipline.

Sur les questions de styles dans la pratique de la programmation l'ouvrage de référence reste le plus souvent celui de Ritchie et Kernighan sur le langage C[1] [KERNIGHAN 1995]. Mais voici la description d'une enveloppe stylistique du code source dans la communauté Allegro, avec des propriétés de lisibilité agréées par tous les candidats au développement de la librairie elle-même[2].

Dans la communauté Allegro

Le style s'appuie sur celui de Ritchie et Kernighan avec 3 espaces d'indentation. Les « defines » et les noms de structure sont en MAJUSCULES. Les noms de fonction et de variables sont en minuscules. « LeMélangedesDeux » est systématiquement évité. De même des notations du « m_genre » sont considérées comme « _vraiment_mauvaises » et il vaut mieux ne pas y penser. Toutes les variables globales doivent être déclarées comme « static » (relevant uniquement du fichier source où elles sont déclarées) et dans le cas où ce n'est pas possible elles sont préfixées au moyen d'un tiret ce qui donne « _globale ».
Les fonctions ont cet aspect :

[1] Brian Kernighan et Rob Pike ont publié récemment, traduit en français, un ouvrage directement lié au style dans la programmation [KERNIGHAN, 2001].
[2] Sont rapportées fidèlement les premières instructions données dans le guide des développeurs de la librairie (« the Allegro hacker's guide »).

```
/* taratoto :
 *  Commentaire préliminaire qui décrit ce que fait la fonction taratoto
 */
void taratoto(int tara, int toto)
{
  /* contenu, instructions */
}
```

Trois lignes vides entre chaque fonction

Pour les instruction de condition :

```
  if (tara) {
    /* instructions */
  }
  else {
    /* instructions */
  }
```

La seule fois où il y a du texte après une accolade fermante est celui du « do while » :

```
  do {
    /* actions*/
  } while (tara);
```

Le « switch » (qui est une instruction de prise de décision à choix multiples) :

```
  switch (tara) {
       case toto:
          /* actions */
          break;
       default:
          /* action */
          break;
    }
```

Voici quelques exemples pour les espacements :

```
  char *p;
  if (condition) { }
  for (x=0; x<10; x++) { }
  taratoto(tara, totot);
  (BITMAP *)data[id].dat;
```

Nous ne pousserons pas plus loin cette évocation de la question des styles de programmation, mais avant de continuer notre exploration, juste un commentaire personnel sur l'algorithme de Robin Fercoq et sa présentation (figures 10.6 et 10.7) :
On dirait une arrête de poisson, un bout de Tour Eiffel, du tricot entre les aiguilles, un empilement de lettres comme des grains de sable agglutinés, éventuellement une chaussette ou une chaussure à lacets. En fait c'est un algorithme en C !

Chapitre 11

Surface de mémoire

Le concept de pixel apparaît du point de vue de la programmation sous la forme de trois nombres. Ce sont des nombres-octets qui trouvent une correspondance dans la mémoire physique d'un ordinateur. Sur ces trois nombres un est réservé au codage de la couleur et les deux autres indiquent une position. Nous avons examiné la question de la couleur, puis de l'obtention de couleurs dans un programme avec la réalisation de robographes des couleurs.

L'objet de ce chapitre-ci est l'étude du pixel du point de vue de sa position. C'est à dire aussi la conception de son espace et de son plan de référence, une toile numérique. C'est la structure de données d'une image dans l'environnement Allegro qui est abordée et des aspects relatifs à la mémoire de l'ordinateur sont précisés. Particulièrement comment le programmeur manipule cette mémoire par l'intermédiaire de concepts du langage de programmation. Ensuite nous évoquons deux robographes primitifs. Le premier lit une valeur de couleur à une position précise dans la toile. Le second donne une valeur de couleur à une position de la toile. Pour finir nous mentionnons également comment s'allume cette couleur sur l'écran de l'ordinateur, toujours du point de vue du programmeur, sans étude préalable du fonctionnement de l'écran de l'ordinateur.

1. Plan de l'image numérique

L'image résulte d'un quadrillage de pixels colorés. C'est à dire qu'elle peut être considérée comme une matrice telle que nous l'avons évoquée dans le chapitre Sept à propos du programme Graphitos. C'est une matrice de « y » lignes qui donnent la longueur des colonnes, sur « x » colonnes qui donnent la longueur des lignes. Chaque position contient la couleur du pixel. En 8 bits il s'agit d'une matrice de unsigned char (un octet sans bit de signe). En 15-16 bits une matrice de unsigned short (deux octets) et en 24-32 bits une matrice d'entiers unsigned int ou unsigned long (quatre octets). Nous allons rester dans le mode 256 couleurs, 8 bits. On peut s'imaginer une image dans la mémoire comme un champ d'octets, par un schéma de ce type :

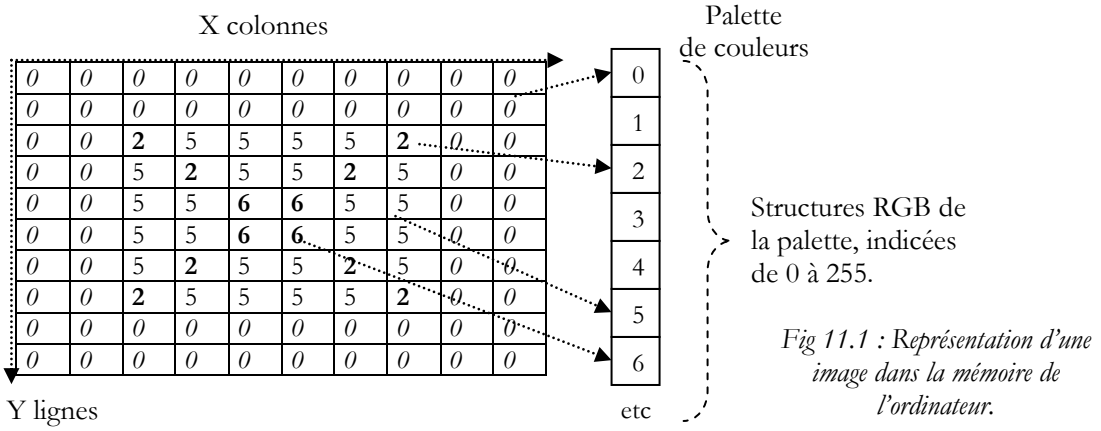


Fig 11.1 : Représentation d'une image dans la mémoire de l'ordinateur.

Il y a là toutes les informations de la trame de l'image : positions en x et y des couleurs associées aux pixels, également les tailles horizontale et verticale de l'image. De même que nous avons vu le codage de la palette des couleurs, voyons le codage de l'image.

Codage de l'image, structure « BITMAP »

Il y a de nombreuses possibilités de format d'image. Celui de la « bitmap », littéralement « plan de bits », est assez largement répandu. Dans l'environnement Allegro l'image bitmap prend la forme d'une structure et nous ne présentons ici que les composantes de cette structure qui peuvent entrer dans des manipulations de la part de celui ou celle qui programme (figure 11.2) ; quelques autres ont trait à la construction interne de l'environnement de programmation et nous ne les abordons pas.

Les informations contenues dans la structure sont dans l'ordre : la taille horizontale de l'image, sa taille verticale, un « clip » qui permet de définir à l'intérieur de l'image une zone rectangulaire accessible à des opérations de dessin. Aucun dessin n'est plus possible en dehors de cette zone si le clip est mis en oeuvre. Le clip sert surtout de sécurité afin de ne pas pouvoir écrire en dehors de l'image, à des adresses mémoire non réservées à l'image. Vient ensuite un pointeur générique pour l'adresse du bloc mémoire qui sera réservé à l'image. Et pour finir ce pointeur est doublé par une autre représentation du même bloc mémoire. Un doublage qui permet d'utiliser la forme de matrice que nous avons évoquée, quoiqu'il ne s'agisse pas exactement d'une matrice mais d'un tableau de pointeurs sur char. Nous verrons cette configuration plus en détail après présentation de la structure bitmap.

```
(1)   typedef struct BITMAP
      {
(2)      int w, h;
(3)      int clip;
(4)      int cl, ct, cr, cb;
(5)      void *dat;
(6)      unsigned char *line[ ];
(7)   } BITMAP;
```

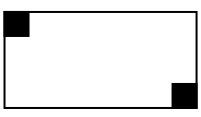
Fig 11.2 : Définition de la structure BITMAP

(1) et (7) Définition de la structure BITMAP et création du type BITMAP.

(2) w pour taille horizontale de l'image et h pour sa taille verticale, en pixels.

(3) Variable pour le fonctionnement du clip. Différent de 0 si le clip est actif et 0 sinon.

(4) La définition du clip. Les coordonnées de deux points pour faire un rectangle : (gauche, haut) et (droite, bas) avec cl pour left, ct pour top, cr pour right, cb pour bottom.



(5) Les champs (5) et (6) permettent d'accéder aux données de l'image, c'est à dire à l'espace mémoire qui les contient.

*void **

« void » signifie vide. Il s'agit d'un type indéfini ne contenant aucune valeur. Son utilisation principale est d'en faire un pointeur générique[1]. Pointeur qui permet de transmettre l'adresse d'un objet dont on ignore le type. Dans la structure BITMAP le pointeur void *dat est destiné à contenir l'adresse du bloc mémoire réservé pour l'image. « void * » est utilisé pour permettre de produire indifféremment des images codée en 8 bits (char), 15-16 bits (short), 24-32 bits (long).

Nous verrons qu'il y a une équivalence entre un pointeur et un tableau. Avec la variable void *dat l'image prend l'allure d'une gigantesque ligne en mémoire. Toutes les lignes qui composent l'image sont les unes à la suite des autres. Cet aspect n'est pas très pratique à manipuler, il est doublé par une autre représentation en (6), et ce sont les mêmes adresses dans les deux cas.

(6) Il s'agit d'un tableau de pointeurs sur unsigned char. C'est une matrice d'octets. sa présence sert à reformuler le bloc mémoire réservé à l'adresse de void *dat. Cette reformulation a lieu au moment de la création de la bitmap, lors de l'appel de la fonction « create_bitmap » qui initialise l'espace mémoire d'une image bitmap.

Objectif : créer une image bitmap

Soit « b_ptr » un pointeur qui contient l'adresse d'une structure BITMAP, c'est à dire une image bitmap du point de vue de la programmation. Cette image à une taille (*x, y*) avec *x* pour l'horizontale et *y* pour la verticale. Soit une couleur quelconque en 8 bits dans une variable « C » ; pour n'importe quelle position (x_0, y_0) de l'image, l'objectif est de pouvoir écrire une expression comme : « b_ptr**->**line[y_0][x_0] = C ; ». Cette instruction, qui affecte la couleur C au pixel de position (x_0, y_0), constitue l'essentiel du robographe primitif de la plupart des tracés. Cette formulation rejoint précisément notre représentation de la figure 11.1. Elle exprime véritablement le principe d'une toile numérique qui coordonne un ensemble de pixels dont le nombre total est *x* multiplié par *y*.

L'image numérique est à sa racine un espace mémoire. Voici les éléments de la manipulation de mémoire dans le langage C requis pour d'atteindre cet objectif et maîtriser la structure de données de la toile numérique avec son écriture.

2. Saisir la mémoire dans le langage C

Plusieurs remarques s'imposent à propos des lignes (5) et (6) de la structure BITMAP. Toutes les données de l'image sont répertoriées à partir d'un seul pointeur générique en (5), comme si le pointeur pouvait contenir l'adresse de toute la suite des informations de

[1] Avant la norme « Ainsi » qui réglemente certains aspects du langage C, le pointeur générique était un « char* ». C'est cette norme qui a introduit le void* [BRAQUELAIRE, 1998, p. 69] [KERNIGHAN, 1995, p . 199].

l'image. Ensuite, à la ligne suivante la taille du tableau n'est pas précisée et l'écriture n'est apparemment pas celle d'une matrice comme nous l'avons affirmé. Ces deux aspects sont à rapprocher d'une équivalence qui existe dans le langage C entre pointeurs et tableaux.

La représentation de variables en mémoire du chapitre Neuf (figure 9.4) illustre comment des variables, entiers, structures ou tableaux correspondent à des adresses mémoire. Eventuellement, il peut être utile d'avoir cette représentation discrètement présente à l'esprit afin de visualiser plus facilement les différentes notions présentées maintenant.

2.1 Manier des adresses

Deux opérateurs sont associés aux pointeurs et sont indispensables à leur utilisation. Il s'agit de l'opérateur de référence **&** (prononcé « adresse de ») et de l'opérateur d'indirection ***** (prononcé « étoile » et à ne pas confondre avec l'opérateur de multiplication qui utilise aussi l'étoile comme signe). Est également associé aux maniements du pointeur l'opérateur –> (prononcé « flèche ») qui intervient dans le cas d'un pointeur de structure[1].

*Les opérateurs & et **

Pour saisir ces deux opérateurs l'important est de différencier l'objet de son adresse et d'identifier la relation qui les unit. Un objet possède une adresse et à une adresse donnée de la mémoire il y a un objet. Il est possible de manipuler l'objet en tant que tel avec ***** ou par le biais de son adresse avec **&**.

Le premier opérateur **&**, accolé à un objet retourne son adresse. C'est à dire qu'il donne un pointeur sur l'objet. On peut dire que si *k* est un objet de type *T* alors « &*k* » est de type pointeur sur *T*. Evidement, cet objet doit posséder une adresse, à savoir être une lvalue. Par exemple soit i un int, « &i » est l'adresse de i qui pointe sur i. Mais, si i est un int, une expression comme « &(i+1) » avec les parenthèses n'est pas correcte syntaxiquement car (i+1) ne correspond pas à un objet en mémoire, ce n'est pas une lvalue.

Le second opérateur *****, accolé à un pointeur, c'est à dire à une adresse, retourne l'objet qui est à cette adresse. Il permet de modifier la ou les valeurs d'un objet à partir d'un pointeur qui contient l'adresse de l'objet. On peut dire que si *k* est de type pointeur sur *T*, **k* est de type *T*. Par exemple si « iptr » est de type pointeur vers entier, *iptr est de type entier et a pour valeur l'entier situé à l'adresse iptr.

Voici des exemples d'utilisation de **&** et de ***** :

```
int i = 23;      // initialisation d'un entier à la déclaration
int  *iptr ;     // déclaration d'un pointeur sur entier de type int
iptr = &i ;      // initialisation du pointeur avec l'adresse de i, c.a.d. que *iptr est égal à 23
*iptr = 55 ;     // 55 affecté à l'adresse de i, i vaut maintenant 55
```

Par ailleurs, l'étoile, placée à la droite d'un type *T* quelconque sert également à la déclaration d'un pointeur : si *T* est un type, *T** est le type pointeur sur *T*.

[1] Sur les opérateurs & et * voir par exemple [CHARBONNEL, 1992, p.49] [BRAQUELAIRE, 1998, p. 121] [KERNIGHAN, 1995, p 91]. Et sur l'opérateur « –> » KERNIGHAN, 1995, p 129].

Pointeur de structure, opérateur –>

Voyons le cas particulier du pointeur de structure qui a son propre opérateur.
Soit une structure BITMAP bmp et un pointeur BITMAP *b_ptr, avec b_ptr = &bmp.
Pour accéder aux champs de la structure il y a plusieurs possibilités.
Soit par l'objet lui-même avec l'opérateur « **.** » (point) avec des expression comme :

bmp**.**w, bmp**.**h, bmp**.**line[y][x] etc.

Soit par l'adresse de l'objet, avec les expressions :

& bmp.w, &bmp**.**h, &bmp**.**line[y][x] etc.

Soit encore via un pointeur sur l'objet avec les formules :

(*b_ptr)**.**w, (*b_ptr)**.**h, (*b_ptr)**.**line[y][x] etc.

Mais pour le pointeur de structure, la sélection d'un champ de la structure s'écrit plutôt :

b_ptr–>w, b_ptr–>h, b_ptr–>line[y][x] etc.

L'opérateur –> est une notation abrégée de « (*b_ptr)**.** » qui regroupe indirection et sélection d'un champ de structure.

Maintenant revenons aux deux opérateurs **&** et ***** qui permettent d'actionner les pointeurs et de gérer la relation étroite qui unit les pointeurs aux tableaux.

2.2 Précision sur les tableaux et les pointeurs

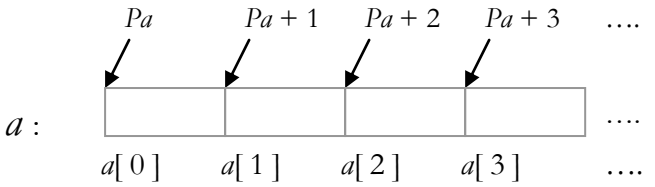
Le tableau est l'adresse en mémoire du premier de ses éléments. En fonction du type et du nombre des éléments du tableau, on accède aux éléments suivants du tableau tout simplement par une incrémentation de la première adresse, origine de l'indiçage du tableau. Soit α une adresse de tableau, en α + 0 on a l'adresse du premier, en α + 1 l'adresse du second, en α + 2 l'adresse du troisième etc. Cette incrémentation correspond à la liste des indices du tableau. Mais aussi à une incrémentation effective de l'adresse réelle selon la taille, le type et le nombre des objets du tableau.

Quant au pointeur *il contient l'adresse mémoire d'un objet.* Si l'on affecte à un pointeur l'adresse du premier élément d'un tableau, le pointeur contient alors l'adresse du premier élément d'un tableau ce qui est la définition même du tableau. C'est à dire qu'une équivalence est établie entre pointeur et tableau. La règle formelle est la suivante : « Si une expression ou une sous-expression est de type « tableau de *T* », ou *T* est un type donné, alors la valeur de cette expression est un pointeur sur le premier élément du tableau, et le type de l'expression se transforme en « pointeur sur *T* » » [KERNIGHAN, 1995 , p. 199].

Soit *Pa* qui est un pointeur d'entiers, et *a*[ ] un tableau d'entier, l'adresse du tableau *a* peut être affectée au pointeur *Pa* avec la formule :

int **Pa* = &*a*[0] ;

On a alors cette situation :



Si, avec l'opérateur ***** on affecte une valeur à l'adresse *Pa*, par exemple **Pa* = 45 ;
alors *a*[0] est égal à 45. Ou encore, une expression comme y = **Pa* signifie que l'on copie le contenu de *a*[0] dans y.
De même, si l'on affecte une valeur x à, par exemple, l'adresse *Pa*+2, ça donne *(*Pa*+2)=x ; et *a*[2] est aussi égal à x.

Les expressions *a*[0] …*a*[n] et **Pa* … *(*Pa*+n), sont des expressions synonymes. Elles peuvent s'employer l'une pour l'autre indifféremment : « Une expression comportant un tableau et un indice est équivalente à une autre écrite avec un pointeur et un déplacement » [KERNIGHAN, 1995, p. 97].

On peut passer de l'une à l'autre de ces formulations avec toutefois cette nuance qu'un pointeur est une variable, alors qu'un nom de tableau désigne le bloc de mémoire des éléments qu'il contient et n'est pas une variable. L'adresse affectée au pointeur peut changer. Le pointeur fait l'objet d'affectations et, sous réserve de rester dans un espace mémoire allouer préalablement, d'additions et de soustractions. Il est possible d'écrire *Pa* = *a* (ce qui est équivalent à *Pa* = &*a*[0]) et ensuite *Pa* = *Pa*+*n* (qui fait pointer *Pa* sur le *n*-eme élément de *a*).
En revanche, pour le tableau, l'adresse du premier élément est invariable et donc il n'est pas possible d'affecter, additionner ou soustraire une valeur au nom du tableau. Pour être plus précis, si *a* est un tableau, une expression comme *(*a*+i) désigne le i-eme élément du tableau car la valeur de *a* n'est pas changée par cette expression. Mais l'affectation *a* = *a* + i qui devrait changer la valeur de *a* n'est pas possible.

Signalons également que l'opérateur d'indiçage **[ ]** qui permet un accès direct à chaque élément d'un tableau, peut être utilisé avec un pointeur. Si *p* est un pointeur et *i* un entier, la valeur de « *p*[*i*] » est identique à celle de « *(*p* + *i*) ». Il s'agit du i-ème élément à partir de l'adresse de départ contenue dans *p*.

Quoique tableau et pointeur restent deux entités distinctes, ils ont un tronc commun d'utilisation du point de vue conceptuel. Tout pointeur peut être assimilé finalement à l'adresse du premier élément d'un tableau d'un type donné, quitte à ce qu'il n'y ait qu'un seul élément dans le tableau ! C'est au moment de l'allocation de mémoire, pendant le déroulement du programme que la zone mémoire allouée au pointeur est définie. Cette zone mémoire correspond à la taille en mémoire d'un élément multipliée par le nombre souhaité d'éléments.

2.3 Calculer une taille en octets

Ce n'est pas l'écriture qui distingue tellement tableaux et pointeurs, c'est surtout la détermination de leur taille. La taille d'un tableau est fixée avant la compilation, et reste

définitive après, pendant le déroulement du programme. En revanche le pointeur permet une allocation dynamique d'une zone de mémoire pendant le déroulement du programme. Cette allocation de mémoire se fait en fonction du type et du nombre d'éléments voulus. La formule est très simple : taille en octet du type ou de l'objet, multipliée par le nombre d'éléments. Et pour obtenir la taille en octet d'un type et d'un objet quelconque, le langage fournit un opérateur qui est l'opérateur « sizeof »[1].

sizeof

L'opérateur unaire[2] sizeof calcule la taille en octet d'un objet (variable, tableau, structure) ou d'un type et donne une valeur entière non signée égale à cette taille (cette valeur est de type « size_t » défini dans le fichier d'en-tête <stddef.h>). Les expressions requises sont sizeof(*objet*) ou sizeof(*nom de type*).

Une expression comme « sizeof(char)*10 » donne l'équivalent de dix chars, c'est à dire dix octets. La structure « nenuph » vue au chapitre Trois contient neuf entiers, neuf ints et il n'y a pas de problème d'alignement. Les entiers sont codés sur quatre octets. Ainsi l'opération « sizeof(struct nenuph) » égale neuf entiers de quatre octets chacun soit 36 octets.

Nous aurions pu utiliser l'opérateur sizeof pour exprimer l'arithmétique des pointeurs qui les rapproche des tableaux. Soit un pointeur ptr sur un tableau de type *T* à n éléments, et i un entier, le i-ème élément de type T à partir de l'adresse contenue dans ptr est :
ptr + (i * sizeof(*T*)), ce qui est correct tant que i reste inférieur à n dans la limite de la taille du tableau.

A ce propos, notons que cet opérateur peut être utilisé pour calculer la taille d'un tableau ou donner son nombre d'éléments.

Soit *a* un tableau, sizeof(*a*) donne la taille totale du tableau. Elle correspond au nombre d'éléments multiplié par la taille-octet d'un élément. La taille totale du tableau divisée par la taille d'un élément donne le nombre d'éléments du tableau. Une macro comme :

```
#define TAILLE_TABLEAU( t ) ( sizeof ( t ) / sizeof ( ( t)[ 0 ] ) )
```

donne la taille d'un tableau quelque soit le type des éléments.

Cependant il faut prendre en considération que l'opérateur sizeof appliqué à un pointeur donne la taille de l'objet pointeur et non celle de l'objet pointé.

Soit ptr un pointeur sur le tableau *a* avec l'initialisation ptr = *a* ; l'expression sizeof(ptr) est égale à 4 octets, la taille du pointeur et non celle du tableau.
La conversion d'un tableau en pointeur sur le premier élément du tableau n'a pas lieu lorsqu'une expression de type tableau est l'opérande d'un opérateur unaire notamment sizeof. C'est pourquoi sizeof(*a*) donne la taille de *a* et non d'un pointeur sur le premier élément de *a*.

[1] Sur l'opérateur « sizeof » voir éventuellement [BRAQUELAIRE, 1998, p. 125, 126] [KERNIGHAN, 1995, p. 133] [CHARBONNEL, 1992, p. 56].
[2] Un opérateur unaire porte sur un opérande unique. Les expressions bâties avec sont évaluées de droite à gauche. Ce sont les opérateurs : ! , ~, ++, --, +, -, *, &, (type), sizeof.

Grâce à l'opérateur sizeof il est possible de définir une taille en octet à partir d'un type ou d'un objet, ce qui entre dans les opérations d'allocation dynamique de mémoire ; en effet, ces opérations permettent de réserver une zone de mémoire pendant le fonctionnement du programme, et d'y accéder ensuite à partir de l'adresse de début stockée dans un pointeur.

2.4 Fonction d'allocation de mémoire

Il y a une fonction standard associée au langage C qui alloue de la mémoire[1]. C'est, dans la librairie <stdlib.h>, la fonction « void* malloc (size_t *taille*) ». Cette fonction alloue un bloc de mémoire de *taille* octets passés en argument. Elle retourne un pointeur générique sur la zone allouée (ou NULL si la mémoire est insuffisante).
La taille d'une zone mémoire de *n* objets de type *T* s'obtient avec la formule :
sizeof (*T*)*n*. Soit ptr un pointeur de type *T**, l'appel de la fonction se réduit à :

ptr = malloc (sizeof(*T*)* *n*) ;

Et on a dans ptr l'adresse d'une zone mémoire réservée suffisante pour *n* objets de type *T*.

Notons qu'à l'inverse, la fonction « void free (void* zone) » libère une zone mémoire précédemment allouée par la fonction « malloc ». Libérer la mémoire retenue à l'adresse de ptr ci-dessus donne simplement :

free(ptr) ;

et l'adresse d'une nouvelle zone peut ensuite être affectée au pointeur ptr.

Reprenons l'exemple des nénuphs donné au chapitre Trois. Toute la communauté des nénuphs est stockée dans un tableau. La taille du tableau est fixe. A chaque nouveau lancement du programme la communauté aura le même nombre de nénuphs. Mais on peut imaginer que les nénuphs aient des comportements différents en fonction du nombre des membres de la communauté. Dans un premier temps il peut être intéressant d'avoir des communautés de tailles différentes mais à chaque fois fixes et sans accroissement ou diminution de population. L'allocation dynamique de mémoire à partir des nombres différents de nénuphs dans la communauté est un moyen de répondre à cet objectif. Admettons par exemple un choix au hasard entre 5 et 200 nénuphs. Dans le programme une fonction d'initialisation peut être écrite de la façon suivante (figure 11.3) :

```
(1)  struct nenuph *N ;

(2)  int initialisation_nenuph()
     {
(3)  int i,j ;

(4)        i = (rand()%196)+5 ;
(5)        N = malloc(sizeof(struct nenuph)*i) ;
(6)        for(j=0; j<i; j++)
                   N[ j ] = init_nenuph();
(7)        return i;
     }
```

Fig 11.3 : Allocation dynamique pour une taille variable de tableau

[1] Pour les fonctions d'allocation mémoire du C voir [BRAQUELAIRE, 1998, p. 198 à 205], [CHARBONNEL, 1992, p. 53, 56, 121], [KERNIGHAN, 1995, p. 164, 182].

(1) Déclaration en global, d'un pointeur sur structure de nénuph. Sa visibilité est étendue au minimum à toutes les fonctions du fichier de code source où il est déclaré.

(2) Identification de la fonction nommée « initialisation_nenuph ». A gauche, int, désigne le type que la fonction retourne ; elle renvoie un entier. La liste entre parenthèse « ( ) » à droite est vide ce qui indique qu'elle ne prend aucun paramètre en argument.

(3) Déclaration de deux variables, i et j, de type int, pour des valeurs entières.

(4) Initialisation de la variable i avec une valeur aléatoire dans la fourchette de 5 à 200.

(5) Initialisation du pointeur N et allocation de mémoire pour i structures nénuph. Le pointeur N reçoit l'adresse du bloc de mémoire réservé.

(5) Il s'agit maintenant d'initialiser les champs de chaque nénuph. Pour ce faire une boucle for passe en revue les i adresses de nénuphs du bloc de mémoire, soit une boucle de 0 à i par la variable j interposée. Une seule instruction qui est un appel à la fonction init_nenuph que nous avons présentée avec les nénuphs au chapitre Trois. Cette fonction retourne une structure nénuph dont les champs sont dûment complétés, et cette structure est affectée à la j-ème structure du tableau-pointeur N.

(6) Comme nous sommes dans le cas d'un tableau de type pointeur et non d'un tableau de type tableau il faut conserver le nombre de nénuphs accessibles à l'adresse N. L'expression sizeof(N) donne la taille du pointeur et non du bloc de mémoire et la macro TAILLE_TABLEAU que nous avons vue ne peut pas fonctionner. D'où l'idée de renvoyer cette valeur et de la récupérer dans la fonction appelante. On aurait également pu déclarer i en variable globale plutôt que locale à la fonction. Ou encore on aurait pu augmenter i de un et ajouter une structure supplémentaire à N, rendre la dernière adresse N+i nulle. Ensuite s'en servir comme borne et utiliser le test N[j] != NULL dans la boucle for. Dans cette perspective voici deux précisions :

Pointeur NULL

Précisons que NULL est défini dans <stddef.h>. C'est une valeur spéciale de pointeur qui vaut en fait 0. Le C garantit que 0 ne sera jamais une adresse utilisable par un objet quelconque. Et 0 est le seul cas où un pointeur peut être affecté d'un entier (il s'agit en fait d'une conversion de type de la forme « ((void*) 0) ») [KERNIGHAN, 1995, p. 100] et [BRAQUELAIRE, 1998, p. 111].

Opérateur != d'inégalité

!= prononcé « est différent de » est l'opérateur inverse de == que nous avons présenté dans le chapitre Sept à propos de Graphitos. Il permet de comparer deux termes. Une expression comme a != b renvoie 0 si c'est faux (a est égal à b) et au contraire 1 si c'est vrai (a n'est pas égal à b). [KERNIGHAN, 1995, p. 41] [BRAQUELAIRE, 1998, p. 116]

Après avoir précisé que N[i] = NULL et joue le rôle de la sentinelle à la fin du bloc mémoire, la boucle for peut ensuite s'écrire :

```
for ( j = 0 ; N[ j ] != NULL ; j++)
```

Ce qui élimine la gestion du nombre d'éléments à l'adresse de N dans le programme. Plus besoin pour la fonction « initialisation_nenuph » de renvoyer la valeur i. Toutefois, si l'on veut accéder à des nénuphs de façon aléatoire ou dans le désordre, le nombre total reste nécessaire pour avoir la fourchette des nombres acceptables et ne pas déborder.

Fonctions calloc et realloc

Il existe deux autres fonctions standard du C toujours dans la librairie <stdlib.h> :
void *calloc(size_t *nombre_elements*, size_t *taille_element*) et
void *realloc(void **adresse*, size_t *nouvelle_taille*)

La fonction « calloc » alloue une zone mémoire qui correspond au nombre d'éléments multiplié par la taille d'un élément. Particularité de la fonction, dans la zone allouée tous les bits sont initialisés à 0.

La fonction « realloc » permet de changer l'allocation mémoire d'une zone désignée par « adresse » qui soit est NULL soit a déjà fait l'objet d'une allocation. Par exemple le pointeur N de la figure 11.3 peut être réalloué pour une taille différente sans qu'il soit nécessaire de passer par la libération avec la fonction « free » de la zone allouée. Si l'on veut 100 nénuphs alors qu'il n'y en a que 50, il suffit d'appeler « realloc(N, 100) ; ». Les éléments de la zone précédemment allouée qui demeurent avec la nouvelle taille restent inchangés. Les 50 premiers nénuphs conservent leurs paramètres, les 50 suivants sont en revanche à initialiser. Si on choisit de n'avoir que 20 nénuphs, seuls les 20 premiers sont conservés. Ce qui est en moins est perdu et ce qui reste conserve ses paramètres.

Adresse, tableaux et pointeurs, taille en octet, allocation de mémoire, tous les éléments nécessaires sont réunis pour aborder la création d'une image en mémoire, à savoir l'usage de la structure BITMAP que nous avons présentée figure 11.2.

3. Création d'une image en mémoire

Revenons aux deux dernières lignes de la structure BITMAP :
Typedef struct BITMAP{

```
        (…)
(5)     void *dat;
(6)     unsigned char *line[ ];

}BITMAP;
```

C'est à ces deux lignes pour ainsi dire que se réduit finalement la notion de toile numérique côté programmeur. Une fonction d'initialisation de la structure BITMAP est présentée qui pose un décor numérique indispensable au moindre tracé du moindre trait.

3.1 Reprise des éléments du codage de l'image

(5) Soit une image bitmap dont la dimension est : *nc*, le nombre de colonnes qui donne la longueur et *nl*, le nombre de lignes qui donne la hauteur. Le nombre total des pixels est le produit *(nc * nl)* qui correspond à toutes les positions de l'image. Comme chaque pixel a une couleur sous la forme d'une valeur entière, pour stocker les pixels le mieux est un tableau redimensionnable, c'est à dire un pointeur qui puisse être adapté quelles que soient *nc* et *nl*. De plus la couleur de l'image peut être codée en 8, 15, 16, 24, 32 bits et le type de notre tableau des pixels doit rester indifférencié et générique. C'est ce que résume la ligne (5), et dans la fonction d'initialisation il y aura une ligne du genre :

bmp_ptr->dat = malloc(sizeof(codage_type) * *nc* * *nl*) ;

Cette expression stocke l'adresse du bloc de mémoire réservé à l'image. C'est un tableau d'octets étiré dans une ligne unique. Chaque position du tableau pouvant accueillir ensuite une valeur de couleur en tenant compte du type de codage qui aura été sélectionné pour la couleur. Cette formulation de l'image peut être utile notamment dans certaines opérations de recopie ou d'initialisation qui utilisent des fonctions des librairies standards du C[1]. Cependant une autre peut lui être préférée pour d'autres raisons pratiques.

(6)[2] Soit un tableau de caractères de taille indéfinie « char tab[ ] ». Du fait de l'équivalence entre tableau et pointeur il est certaines situations où le nombre des éléments du tableau n'est pas nécessaire à la compilation parce que le tableau est considéré comme un pointeur. C'est le cas par exemple lorsqu'un tableau est pris comme argument, et intervient comme paramètre formel dans une définition de fonction. Dans cette situation le tableau est converti en pointeur comme nous l'avons mentionné plus haut, et c'est l'adresse du premier élément de tab, « &tab[0] », qui est transmise. L'expression « char tab[ ] » est alors équivalente à « char* tab » [KERNIGHAN, 1995, p. 97].

Admettons maintenant que tab soit initialement un tableau de pointeurs de caractères, c'est à dire une expression telle que « char *tab[ ] ». Dans ce cas on peut conserver l'interprétation de « tab[ ] » comme celle d'un pointeur. *Il s'agit alors d'un pointeur de pointeurs de caractères* et l'expression pourrait s'écrire « char **tab » [BRAQUELAIRE, 1998, p.132]. Les deux notations traduisent une double indirection. Dans la structure BITMAP, la première indirection concerne un nombre indéfini de lignes avec, sous la forme d'un tableau sans taille, le pointeur « line[ ] », et la seconde la longueur indéfinie de chaque ligne avec pour chaque ligne un « char* ». Dans ce contexte de la toile numérique de l'image, la notation « char* line[ ] » paraît plus claire que « char** line ».

[1] Par exemple citons quelques fonctions de la librairie <string.h>, de celles qui commencent par mem… et servent à manipuler des objets en tant que tableau de caractères c.à.d d'octets :
void* memcpy(void *destination, const void *source, size_t taille) ; qui recopie taille octet de l'adresse source à l'adresse destination sans recouvrement des deux zones initiales.
void* memmove(void *destination, const void *source, size_t taille) ; qui déplace une zone de taille octet vers une autre avec recouvrement initial.
void* memset(void *debut, int valeur, size_t taille) ; qui initialise taille octets à partir de l'adresse début avec une valeur passée à valeur.
A ce sujet consulter [BRAQUELAIRE, 1998, p. 205] [KERNIGHAN, 1995, p. 255].

[2] Nous n'avons pas trouvé de référence à l'expression de la ligne (6) dans une structure. Ni Kernighan ni Braquelaire ne mentionnent ce point directement.

Il faut remarquer que lors de la déclaration d'une structure, si un tableau « sans taille » est membre de cette structure, il ne peut y en avoir qu'un seul et ce doit être le dernier membre de la structure. En effet l'absence de taille du tableau ne permet pas au compilateur d'aligner les membres suivants (l'opérateur sizeof appliqué au champ line donne la valeur 0). C'est dire également que la structure est comme inachevée du point de vue de sa taille et qu'au moment de l'allocation de mémoire d'une BITMAP* il sera nécessaire d'ajouter ce qui manque. Le calcul est fait dans une fonction via un de ses arguments qui lui donne la hauteur, c'est à dire le nombre de lignes souhaité pour l'image.

Avec pour unité le « unsigned char* » le tableau « line » est un tableau de pointeurs sur octets. C'est à dire un tableau de tableaux d'octets. Chaque élément de « line » est l'adresse du premier élément d'un tableau d'octets dont on ignore le nombre. Avec cette formule on a une matrice dont on ne connaît ni le nombre de lignes, ni le nombre de colonnes.

La fonction « create_bitmap_ex » nous renseigne aux lignes 4, 6, 9, 10 et 11 de la figure 11.4 sur la manière dont les lignes (5) et (6) de la structure BITMAP sont initialisées.

3.2 Détail d'une fonction d'initialisation

La fonction « BITMAP *create_bitmap_ex(int type_couleur, int taille_hor, int taille_ver) » réserve un espace mémoire selon les trois paramètres passés en argument. Le premier détermine le nombre de bits pour le codage de la couleur, les deux autres la taille horizontale et verticale voulue pour l'image. La fonction renvoie l'adresse de la zone réservée, c'est à dire un pointeur sur une structure BITMAP. Par exemple pour réserver l'espace mémoire d'une image 8 bits de 100 pixels sur 100, l'appel de la fonction sera, après déclaration d'un pointeur BITMAP* :

```
BITMAP *bmp_ptr ;
        (…)
        bmp_ptr = create_bitmap_ex (8, 100, 100);
```

Dans l'environnement Allegro la fonction originale se présente dans les termes suivants (figure 11.4) :

En italique et non numérotées sont des lignes que nous n'expliqueront pas, relatives aux éléments de la structure BITMAP que nous n'avons pas présentés. Nous nous concentrons sur la notion de toile numérique, et évitons d'axer le propos plutôt vers des questions relatives à la conception de l'environnement de développement. L'objectif est pour nous la perception d'un espace robographique à partir d'un espace graphique modélisé et interprété en types et structures de données.

(1) Identification de la fonction nommée « create_bitmap_ex ». A gauche, BITMAP *, désigne le type que la fonction retourne ; elle renvoie un pointeur de type BITMAP*. La liste entre parenthèses désigne les trois paramètres que nous avons déjà évoqués mais selon les termes anglosaxons. La variable « color_depth » correspond aux nombres de bits du codage de l'image, « width » est la longueur à savoir le nombre de colonnes et « heigth » la hauteur c'est à dire le nombre de lignes.

(2) Ce sont les déclarations de variables locales à la fonction. Il s'agit d'un pointeur BITMAP* et d'un entier i.

```
(1)     BITMAP *create_bitmap_ex(int color_depth, int width, int height)
        {
        GFX_VTABLE *vtable;
(2)     BITMAP *bitmap;
        int i;

          if (system_driver->create_bitmap)
             return system_driver->create_bitmap(color_depth, width, height);

          vtable = _get_vtable(color_depth);
          if (!vtable)
            return NULL;

(3)       bitmap = malloc(sizeof(BITMAP) + (sizeof(char *) * height));
(4)       if (!bitmap)
            return NULL;

(5)       bitmap->dat = malloc(width * height * BYTES_PER_PIXEL(color_depth));
(6)       if (!bitmap->dat) {
            free(bitmap);
            return NULL;
          }

(7)       bitmap->w = bitmap->cr = width;
          bitmap->h = bitmap->cb = height;
          bitmap->clip = TRUE;
          bitmap->cl = bitmap->ct = 0;
          bitmap->vtable = vtable;
          bitmap->write_bank = bitmap->read_bank = _stub_bank_switch;
          bitmap->id = 0;
          bitmap->extra = NULL;
          bitmap->x_ofs = 0;
          bitmap->y_ofs = 0;
          bitmap->seg = _default_ds();

(8)       bitmap->line[0] = bitmap->dat;
          for (i=1; i<height; i++)
            bitmap->line[i] = bitmap->line[i-1] + width *
BYTES_PER_PIXEL(color_depth);

          if (system_driver->created_bitmap)
             system_driver->created_bitmap(bitmap);

(9)       return bitmap;
        }
```

Fig. 11.4 : La fonction create_bitmap_ex d'initialisation d'une bitmap dans l'environnement Allegro.

(3) Le premier point important pour notre démarche concerne l'attribution d'une zone de mémoire susceptible de contenir une image bitmap dans sa totalité et l'affectation de son

adresse dans un pointeur de type BITMAP*. Cette zone est premièrement de la taille d'une structure BITMAP et de ses champs c'est à dire « sizeof (BITMAP) ».

Mais il faut maintenant se soucier du tableau de pointeurs de caractères, « char *line[ ] » dont la taille est nulle. Comme le nombre de lignes est connu lors de l'appel de la fonction, il suffit d'ajouter la mémoire nécessaire à ces lignes. D'où l'ajout à la taille de la structure BITMAP de « sizeof (char *) * heigth », c'est à dire l'espace requis pour l'ensemble des pointeurs char* qui correspondent chacun à l'adresse d'une ligne de l'image, un tableau de char.

(4) Vient ensuite un test sur la valeur de retour de la fonction « malloc ». La fonction retourne NULL en cas d'échec, par exemple si la mémoire disponible est insuffisante.

Opérateur unaire de négation « ! »

Cet opérateur « ! » a été abordé brièvement avec le test de comparaison d'inégalité « != » lors de la présentation de la fonction « initialisation_nenuph » (ligne 6 de la figure 11.3). C'est l'expression logique de la négation. L'opérateur « ! » associé à un opérande renvoie 0 si l'opérande n'est pas nul et 1 si l'opérande est nul.

Si « malloc » renvoie une valeur correcte, elle n'est pas nulle et le test « if(! bitmap) » est faux et vaut 0, le bloc d'instruction n'est pas exécuté. Au contraire si « malloc » renvoie NULL le test est vrai et l'instruction « return NULL » est effectuée (c'est à dire sortie de la fonction avec retour de la valeur NULL)

(5) Compte tenu de ses paramètres la fonction dispose de la taille de l'image en nombre de pixels. C'est width (longueur) multiplié par heigth (hauteur). Pour avoir la taille totale en octets il faut multiplier le nombre de pixels par le nombre d'octets utilisés pour le codage de la couleur à chaque pixel. En 8 bits c'est 1, en 15 et 16 bits c'est 2, en 24 c'est 3 et en 32 c'est 4. L'argument « color_depth » de la fonction se traduit par des valeurs de 8, 15, 16, 24 ou 32. Il reste à les transformer en 1, 2, 3 ou 4 octets.

C'est le rôle de la macro « BYTES_PER_PIXEL » qui se présente de la façon suivante (figure 11.5) :

```
#define BYTES_PER_PIXEL(bpp)  (((bpp) <= 8) ? 1                              \
                                    : (((bpp) <= 16) ? sizeof (unsigned short)        \
                                        : (((bpp) <= 24) ? 3 : sizeof (unsigned long))))
```

Fig 11.5 : Macro qui donne le nombre d'octets de chaque codage de la couleur.

Expression conditionnelle « ? : »

Elle utilise une expression particulière qui se décompose en un test plus deux possibilités en fonction des résultats du test. C'est un opérateur ternaire. L'expression est :

expr1 ? *expr2* : *expr3*

On peut traduire par : « Est-ce que *expr1* est vrai ? Si oui la valeur de l'expression est *expr2* si non la valeur de l'expression est *expr3* ».

Par exemple l'expression « (x >= 0) ? 1 : -1 » renvoie 1 ou -1 selon le signe positif ou négatif d'une variable x.

De même (x >= 0) ? x : -x retourne la valeur absolue d'une variable x. (C'est avec cette formule qu'est écrite la macro ABS(n) utilisée aux lignes 8, 11, 14 figure 10.5 dans la fonction « degrade_pas_variable » du chapitre précédent). Le test porte sur le signe de x. Si x est supérieur ou égal à 0 le résultat est x. Si au contraire x est inférieur à 0 le résultat inverse le signe de x avec (-x).

Il est possible d'imbriquer des expressions conditionnelles. C'est le cas avec l'expression de la figure 11.5. Le nombre de bits qui code la couleur (color_depth) est passé à la macro BYTES_PER_PIXEL via le paramètre bpp. Le premier test détermine si cette valeur est inférieure ou égale à 8. Si oui le nombre d'octets par pixel est 1 et l'expression renvoie 1. Si non, nouveau test pour savoir si cette valeur est inférieure ou égale à 16. Si oui le nombre d'octets par pixel est 2 et l'expression vaut 2. Si non, nouveau test pour savoir si cette valeur est inférieure ou égale à 24. Si oui l'expression vaut 3 et sinon elle vaut la taille d'un long c'est à dire 4.
Ainsi la formule « width * height * BYTES_PER_PIXEL(color_depth) » donne la taille totale de l'image en mémoire compte-tenu de sa longueur (width), de sa hauteur (height) et du nombre d'octets utilisés pour coder l'image (BYTES_PER_PIXEL(color_depth)).

(6) Comme en (4) vient ensuite un test sur la valeur de retour de la fonction « malloc ». En cas d'échec, « malloc » retourne NULL. Dans ce cas la zone mémoire précédemment allouée dont l'adresse est affectée au pointeur « bitmap » est libérée et la fonction create_bitmap_ex retourne NULL.

(7) Les champs de la structure BITMAP sont accédés via le pointeur « bitmap » alloué à la ligne (3) et l'opérateur flèche. La longueur et la hauteur passées en argument à la fonction sont conservées dans leurs champs respectifs (w et h) mais également constituent les bornes droite et bas du clip (respectivement les champs cr et cb). Les variables du clip gauche et haut (cl et ct) sont mises à 0. Le clip lui-même est activé par défaut.

(8) L'objectif est ici de découper la ligne unique dont l'adresse est contenue au champ « dat », en un nombre de lignes équivalent à celui de la hauteur (heigth) de l'image, compte-tenu de sa longueur (width) et également du nombre d'octets du codage de la couleur (BYTES_PER_PIXEL(color_depth)).

Tout d'abord, l'adresse contenue au champ « dat » (bitmap->dat) est affectée à la première ligne du tableau line (bitmap->line[0]).
Ensuite, il y a une boucle for qui part de la seconde ligne (bitmap->line[1]) jusqu'à la dernière (bitmap->line[heigth-1]).

Pour chaque ligne i de 1 à heigth-1, le pixel qui finit la ligne précédente, c'est à dire la ligne i-1, est affecté comme première adresse de la ligne courante i.

Au début lorsque i = 1 la ligne précédente, bitmap->line[i-1] est bitmap->line[0]. Pour avoir le premier pixel de la ligne suivante, il faut ajouter la taille en octets de la ligne à partir de l'adresse de départ en bitmap->line[0].

La taille en octets de la ligne est obtenue en multipliant la longueur de la ligne avec le nombre d'octets du codage de la couleur, c'est l'expression :

```
width * BYTES_PER_PIXEL(color_depth)
```

Cette valeur est ajoutée à l'adresse de départ de la ligne précédente pour obtenir l'adresse du premier pixel de la nouvelle ligne. C'est l'expression :

```
bitmap->line[i-1] + width  * BYTES_PER_PIXEL(color_depth)
```

Ensuite cette adresse est affectée comme adresse du premier pixel de la nouvelle ligne :

```
bitmap->line[i] = bitmap->line[i-1] + width * BYTES_PER_PIXEL(color_depth);
```

Au début, lorsque i = 1, c'est l'adresse de bitmap->line[0] qui est incrémentée avec la valeur de « width * BYTES_PER_PIXEL(color_depth) ». Le résultat est l'adresse du premier pixel de la ligne suivante, la ligne bitmap->line[1], et cette adresse lui est affectée.

Ensuite i est incrémenté de 1 par le i++ de la boucle for et l'opération est recommencée pour chacune des lignes jusqu'à la dernière, lorsque i est égal à heigth-1.

(9) Une fois l'initialisation terminée la fonction retourne le pointeur « bitmap » qui va être récupéré dans la fonction appelante.

L'objectif fixé au début de ce chapitre est maintenant presque entièrement atteint.
A partir d'un pointeur B de type BITMAP* initialisé par la fonction « create_bitmap_ex » il est possible d'accéder à la toile numérique d'une image avec des expressions telles que :

```
B->[ y ][ x ] = couleur ;
```

Avec y entier inférieur à la hauteur de l'image, x entier inférieur à la longueur de l'image et la couleur quel que soit le nombre d'octets du codage.
Il ne reste plus qu'à construire nos deux robographes de lecture-écriture de couleurs.

4. Deux robographes primitifs

Deux fonctions sont pour ainsi dire à la racine d'un grand nombre d'aspects graphiques et peuvent être présentées simplement. Il s'agit des fonctions « putpixel » et « getpixel » présentes dans pratiquement tout environnement de programmation qui offre quelques possibilités graphiques. La première donne une couleur à un pixel, et la seconde permet de connaître la couleur d'un pixel à une position donnée. Dans un premier temps ces fonctions sont abordées selon un codage 8 bits des couleurs, en mode 256 couleurs. Ensuite une modification est apportée afin de passer au mode vraie couleur et les codages 15, 16, 24 et 32 bits de la couleur.

4.1 Colorer un pixel ou connaître sa couleur

Fonction « Putpixel »

Une version minimaliste de la fonction « Putpixel » de coloration d'un pixel peut avoir l'allure suivante (figure 11.6) :

(1) Déclaration de la fonction « Putpixel ». Elle a quatre paramètres. Le premier est un pointeur BITMAP*. Il donne accès à la structure BITMAP sur laquelle l'opération est effectuée. Le pointeur passé en argument devra avoir été précédemment initialisé avec la fonction create_bitmap_ex. Les second et troisième arguments, « x » et « y », sont pour la position du pixel. Le dernier « couleur » est pour la couleur voulue, selon le codage qui aura été passé en argument à la fonction create_bitmap_ex utilisée pour initialiser le pointeur et la structure.

```
(1)   void  Putpixel ( BITMAP* bmp, int x, int y, int couleur){

(2)         bmp->line[y][x]=couleur;
      }
```

Fig. 11.6 : Fonction Putpixel minimaliste, robographe d'écriture de couleur

(2) Une seule instruction qui est simplement d'affecter la valeur contenue dans « couleur » à la position (x, y) dans la matrice de l'image pointée par « bmp ».

Fonction « Getpixel »

La seconde fonction permet au programme de connaître la couleur d'un pixel à une position donnée dans l'image bitmap (figure 11.7) :

```
(1)   int  Getpixel ( BITMAP* bmp, int x, int y ){

(2)       return bmp->line[y][x];
    }
```

Fig. 11.7 : Fonction Getpixel minimaliste, robographe de lecture de couleur

(1) Déclaration de la fonction « Getpixel ». Le type de retour de cette fonction est un int, c'est à dire que cette fonction renvoie une valeur entière. Comme dans la fonction précédente elle prend en argument la BITMAP de référence par l'intermédiaire d'un pointeur « BITMAP * » et deux entiers qui indiquent la position (x, y) du pixel qui intéresse.

(2) Même formule « bmp->line[y][x]; » que dans la fonction précédente avec cette différence qu'il n'y a plus l'affectation d'une valeur mais que la valeur contenue à la position regardée est retournée à la fonction appelante, celle-ci devant prévoir une variable pour la récupérer.

Exemple d'utilisation

On pourra par exemple avoir une fonction de ce type (figure 11.8) :

Cette fonction passe en revue chaque pixel de l'image dont l'adresse est passée en argument via le pointeur bmp. Si, pour un pixel donné de l'image, la valeur retournée par « Getpixel » est supérieure à la valeur passée en argument dans la variable « modif », la couleur de cette position est modifiée de (-modif) pour l'indice dans la palette des couleurs.

(1) Identification de la fonction nommée « modifier_couleurs ». Elle a deux paramètres un pointeur BITMAP* et un entier. Elle ne renvoie aucune variable. L'identification est suivie des déclarations des variables x, y, toto, locales à la fonction.

```
(1)  void modifier_couleurs(BITMAP* bmp, int modif ){
     int x, y ;
     int toto ;

(2)      for( y = 0; y < bmp->h; y++ )
           for( x = 0 ; x  <  bmp->w ;  x++ ){
(3)            toto = Getpixel( bmp, x, y ) ;
(4)            if( toto > modif )
                 Putpixel( bmp, x, y, toto-modif );
           }
}
```

Fig. 11.8 : Fonction de modification de couleurs

(2) Boucle for qui passe en revue les lignes de l'image. Ensuite pour chaque ligne, boucle for qui passe en revue chaque pixel de la ligne. Ainsi sont passées en revue toutes les positions (x, y) de l'image.

(3) « Getpixel » renvoie la valeur de chaque pixel de l'image, et cette valeur est affectée à la variable « toto ».

(4) Test sur la valeur de toto. L'idée est ici que si cette valeur est supérieure à la valeur borne passée en argument et affectée à la variable modif lors de l'appel de la fonction, la valeur de « toto » est diminuée de celle de modif. Ce qui donne l'expression :
« toto– modif », et la fonction « Putpixel » en affecte le résultat à la position (x, y) courante.

Par exemple si modif est égal à 30, chaque pixel ayant une couleur supérieure à 30 est diminué de 30 et les autres pixels sont laissés tel quel. Le résultat dépend ensuite de la façon dont est construite la palette. S'il s'agit du dégradé de gris que nous avons construit plus haut, toutes les valeurs claires au dessus de trente seront ramenée à des valeurs de (–30) plus sombres. Il y aura un certain aplanissement du relief des couleurs et un assombrissement général de l'image.

Notons que l'appel de la fonction « modifier_couleur », après déclaration et initialisation quelque part dans le programme d'un pointeur sur BITMAP, sera par exemple pour une image en 8 bits de 100 sur 100 pixels :

```
BITMAP* im ;

(…)
im = create_bitmap_ex( 8, 100, 100) ;
(…)
modifier_couleurs( im, 30) ;
(…)
```

Telles quelles les fonctions Putpixel et Getpixel sont en 256 couleurs. Tous les nombres supérieurs seront réduits via le modulo que nous avons présenté au chapitre 9 figure 9.3. Une très légère modification permet d'utiliser ces mêmes fonctions pour 15,16 et 32 bits.

4.2 Conversion de type pour le mode « vraie couleur »

Si un autre mode que le mode 8 bits est choisi, par exemple 16 bits, la multiplication par deux du nombre d'octets est faite lors de l'appel de la fonction « create_bitmap_ex ». Il s'agit de la ligne (5) de la figure 11.4 qui concerne l'allocation mémoire pour les informations de couleur de chaque pixel. Nous avons vu que la formule « width * height * BYTES_PER_PIXEL(color_depth) » donne la taille totale de l'image en mémoire.

Les différences de codage de la couleur peuvent s'exprimer ainsi :
Si l'on est en 16 bits, chaque ligne de l'image est une ligne de pixels en 16 bits, chaque couleur étant codée sur deux octets. C'est à dire que ce sont des lignes d'unités de type short et non plus des lignes d'unités de type char. Il faut donc opérer *une conversion de type* de « char » à « short » afin de pouvoir parcourir la ligne de short en short, c'est à dire par unité de deux octets. De même si l'on est en 32. Ce seront des lignes d'entier long, de quatre en quatre octets.

Opérateur cast

Pour opérer cette conversion un « cast » est utilisé. Il s'agit d'un opérateur unaire , évalué de droite à gauche. La construction pour forcer explicitement une conversion d'un type dans un autre est :

(nom-de-type) expression

La définition du cast veut que « tout se passe comme si l'on affectait l'*expression* à une variable du type indiqué, utilisée ensuite à la place de cette construction » [KERNIGHAN, 1995, p. 45, 203, 205].

Ainsi dans le contexte qui nous intéresse pour forcer une ligne de char à être parcourue comme une ligne de short on aura le cast suivant : « (short*)line[y] » qui transforme la ligne y d'un pointeur char* en un pointeur short*. Le char* permet d'accéder à des adresses d'octets. Le short* permet d'accéder aux adresses des short de deux en deux octets consécutifs sur la ligne. Sur la figure 11.9, x devient le décalage d'adresse en fonction du nombre d'octets correspondant au type. Ce décalage est effectué à partir de l'adresse initiale en line[y].

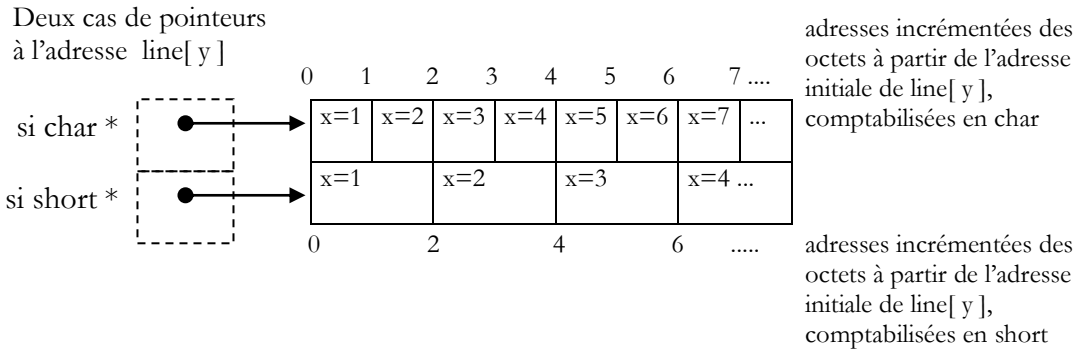


Fig 11.9 : Comparaison du comptage des pixels en fonction du nombre d'octets requis par le type utilisé dans le codage de la couleur d'une image bitmap.

On remarque que le poids de l'image en terme d'octets dépend évidemment du nombre de bits utilisés pour coder la couleur.

Les fonctions « Putpixels » et « Getpixel » deviennent pour 15-16 et 32 bits (figure 11.10) :

```
void Putpixel_15_16 (BITMAP*bmp, int x, int y, int couleur){

        ( (short*)bmp->line[ y ] ) [ x ] = couleur;
}

void Putpixel_32 (BITMAP*bmp, int x, int y, int couleur){

        ( (long*)bmp->line[ y ] ) [ x ] = couleur;
}

int Getpixel_15_16 (BITMAP*bmp, int x, int y ){

        return ( (short*)bmp->line[ y ] ) [ x ] ;
}

int Getpixel_32 (BITMAP*bmp, int x, int y ){

        return ( (long*)bmp->line[ y ] ) [ x ] ;
}
```

Fig. 11.10 : Putpixel et Getpixel en 15-16 et en 32 bits

Ces fonctions sont utilisables telles quelles dans l'environnement Allegro. Cependant elles ne donnent pas accès à l'écran de l'ordinateur et tout se passe uniquement dans la mémoire centrale de l'ordinateur, la « RAM ».

5. Accéder à l'écran de l'ordinateur

L'écran de l'ordinateur fait l'objet d'une bitmap spéciale, créée dans la mémoire vidéo et non plus la mémoire centrale par une fonction d'initialisation fournie par l'environnement : « set_gfx_mode ». Cette fonction est appelée en général au début du programme. Elle prend plusieurs arguments notamment un paramètre de mode graphique, en relation avec le système d'exploitation et la carte graphique (plein écran ou fenêtre et avec plusieurs choix possibles). Egalement elle prend en argument la résolution souhaitée pour l'écran. Eventuellement, si la carte graphique l'accepte, il y a la possibilité d'une zone d'écran dite « virtuelle ». C'est à dire de disposer dans la mémoire vidéo d'une surface d'écran supérieure à la taille réelle de l'écran ceci afin de permettre le « scroll » ; la surface de l'écran réel devenant comme une ouverture sur un espace plus large.

Image écran

La fonction « set_gfx_mode » crée une bitmap dans la mémoire vidéo qui permet un affichage à l'écran. Cette bitmap nommée « screen » correspond, du point de vue de la programmation, à l'écran lui-même. Elle permet de rendre visible à l'écran tout ce qui va réclamer un affichage. C'est une variable globale ; elle n'est réservée à aucune fonction mais est accessible par toutes les fonctions dans le programme.

L'accès à la mémoire vidéo n'est pas le même que celui que nous avons vu pour une bitmap en RAM. Il existe cependant un moyen de modifier les fonctions « Putpixel » et « Getpixel » que nous avons créées pour qu'elles fonctionnent avec cette bitmap screen réservée à l'écran. Mais le plus simple est d'utiliser les deux fonctions « putpixel » et « getpixel » (sans majuscule) qui sont livrées avec l'environnement et qui sont conçues de façon à pouvoir fonctionner indifféremment avec bitmap en RAM ou la bitmap écran screen de mémoire vidéo.

Ces deux fonction prennent exactement les mêmes arguments. Pour attribuer une couleur à un pixel visible à l'écran ou pour connaître, au sein d'un programme, la couleur d'un pixel de l'écran, nous aurons par exemple :

```
putpixel(screen, x, y, couleur) ;
```

et

```
couleur = getpixel(screen , x, y ) ;
```

Ces deux fonctions, du point de vue de la programmation et des concepts associés, résument finalement le concept de pixel dans l'environnement que nous utilisons. Elles sont à la racine de beaucoup d'autres fonctions, en particulier, la plupart de celles qualifiées de primitives de dessin et destinées à différents types de tracés. En quelque sorte ces deux fonctions, associées aux structures de données que nous avons vues sont les deux plus petits robographes que nous connaissons.

Les fonctions acquire_bitmap( ) et release_bitmap( )

A propos de l'affichage sur l'écran et l'utilisation de la mémoire vidéo, notons un détail pratique à défaut d'être en mesure de l'expliquer techniquement.
Sous le compilateur Microsoft Visual C++ et sous un système d'exploitation Windows (98, Millénium, 2000, XP etc.) la librairie Allegro couplée avec l'environnement multimédia « directX » nécessite l'encadrement des fonctions d'affichage par les deux fonctions « acquire_bitmap » et « release_bitmap ». Ce qui donne :

```
acquire_bitmap(screen) ;
putpixel( screen, x, y, couleur) ;
release_bitmap(screen) ;
```

C'est une question de verrouillage de la mémoire vidéo lié au fonctionnement de l'affichage. Sous DOS elles ne sont pas nécessaires.
Dans les exemples qui suivent nous avons omis les appels de ces fonctions afin de ne pas nuire à la lisibilité des algorithmes.

Deux macros sont définies de façon à pouvoir écrire :

```
acquire_screen( ) ;
putpixel( screen, x, y, couleur) ;
release_screen( ) ;
```

Chapitre 12

Formes

Hormis sa valeur scientifique [de l'œuvre d'art], qui dépend d'un examen précis des éléments particuliers de l'art, l'analyse de ces éléments constitue un pont vers la vie intérieure de l'œuvre. L'opinion, répandue aujourd'hui encore, qu'il serait fatal de « disséquer » l'art, et que cette autopsie mènerait inévitablement à la mort de l'art, résulte de l'ignorante dépréciation des éléments mis à nu et de leurs forces primaires.

Wassily Kandinsky

Le chapitre précédent a permis d'envisager l'image numérique sous l'angle d'une structure de données. Le cadre spatial du pixel, une surface mémoire dans laquelle il est incorporé, a été détaillé ainsi que les éléments du langage C nécessaires aux manipulations. Reste à voir comment le pixel est inséré dans des constructions de formes, comment des traits peuvent être tracés par un robot traceur sur un quadrillage de pixels. Il reste à entrer dans la conception algorithmique. C'est le sujet des chapitres qui suivent.

Langue dans la langue

Notre abord de la conception algorithmique repose sur une démarche de créativité et ce chapitre n'est pas dirigé vers la résolution de problèmes. Comme nous l'avons indiqué à propos de la mise en place de projets (chapitre Deux) la connaissance que nous recherchons à travers des réalisations n'est jamais à proprement parler une solution. Le projet technique est un projet de sens à travers la technique, « une langue dans la langue » selon l'expression de Gilles Deleuze que nous avons évoquée ensuite, avec un rapprochement possible de la littérature et de l'intelligence artificielle (chapitre Cinq). Certes, les tracés de formes rejoignent mathématiques, géométrie et la partie scientifique de l'algorithmique. Mais plutôt que de déterminer des propriétés, avantages et inconvénients, rapidité ou lourdeur de tel ou tel algorithme, nous nous en tenons à rendre sensible ce que nous entendons par robographe, et avec le raisonnement pragmatique qui lui convient le mieux. Les emprunts théoriques aux disciplines scientifiques sont minimisés au maximum.

L'objectif est de rapprocher le processus algorithmique et ses structures de données, de l'imaginaire de celui qui le comprend et le manipule. C'est la représentation et l'énoncé en vue d'une appropriation, d'une intériorisation et peut-être d'un détournement qui nous intéresse. D'une certaine façon il est nécessaire de désolidariser ces figures algorithmiques de leur patrimoine de référence scientifique, afin de leur donner une nouvelle référence de sensibilité artistique.

Primitive robographique

Il convient de remarquer que les limites d'un robographe ne sont pas toujours évidentes à isoler avec précision. Plusieurs robographes peuvent entrer dans la conception d'un robographe de niveau supérieur. Ainsi par exemple lorsque nous nous penchons sur l'animation qui intègre typiquement des fonctions fournies par l'environnement de développement. Ainsi le robographe primitif pur n'est pas isolable systématiquement. La primitive robographique dépend du contexte de la conception, du choix des structures de données, du modèle qui est fixé.

Plusieurs niveaux d'élaboration des matériaux d'une création

Il est possible de dégager différents niveaux d'élaboration des matériaux robographiques. Les trois chapitres qui suivent sont composés de cinq étapes, que nous avons repérées, et qui correspondent à plusieurs registres d'élaboration des matériaux d'une création.

La première étape, chapitre Douze sur les formes, regroupe des éléments admis comme primitives dans le domaine de la géométrie et du graphisme. Cette étape s'ouvre également sur un espace que nous considérons comme un espace de composition. Ce sont des chemins de pixels comme des colliers qui élargissent les possibilités de formes simples. Le

principe retenu s'appuie sur la recherche d'un différentiel, d'un front entre deux bords. Ce peut être le détourage d'une forme du fait d'une différence de couleurs sur un fond mais également le partage du plan en deux par une courbe. Ensuite, au chapitre Treize, le mouvement est donné aux formes. En premier il y a un jeu de circulation et d'animation des couleurs, puis c'est l'animation de formes simples et de petites images. La notion de trajectoire est également évoquée. Additionner le mouvement aux formes invite à la mise en scène de propositions, invite à raconter des histoires. C'est l'objet de la quatrième étape. Elle se traduit par une exposition de croquis au chapitre Quatorze. Chaque croquis incorpore une idée et peut être considéré comme un germe de développements possibles. Eventuellement une recherche peut porter sur des enchaînements de plusieurs croquis avec des processus de métamorphoses. La dernière étape, le chapitre Quinze, met en évidence l'apport de ce qui est non calculable dans une composition algorithmique, ce qui en fait un élément qui fonctionne sur des registres de sensibilité culturelle, ce qui fait qu'une mélodie n'est pas seulement un ensemble de notes, mais des notes ensembles qui sonnent.

1. Formes simples

Discontinuité de la surface numérique

Lignes et courbes résultent de chemins établis à partir de pixels connexes mis en relations. Ces relations font l'objet d'une diversité de modèles et d'approches. Lorsqu'elles sont dérivées de la géométrie, ces relations nécessitent de passer de l'espace mathématique continu à celui discret d'une surface de pixels. En effet, le point mathématique infiniment petit a un nombre considérable de points voisins, mais il se retrouve, en tant que pixel dans l'image numérique avec un maximum de 8 pixels voisins (figure 7.9 vue au chapitre Sept à propos du logiciel GRAPHITOS). De plus entre deux points mathématiques on peut toujours placer une infinité d'autres points et cet aspect disparaît complètement sur une surface numérique.

Primitives de dessin

Les chemins qui relient les pixels afin de constituer diverses figures sont figurés par des changements de couleur des pixels concernés et les algorithmes mis en œuvre contrôlent ces changements via des fonctions comme « putpixel » que nous avons évoquée dans le chapitre précédent. En général, un environnement de développement fournit des fonctions dites « primitives de dessin » qui tracent des formes de base telles les cercles, droites, ellipses, arcs, rectangles, triangles, courbes de Bézier etc.

Réalité scientifique

En dépit de la simplicité apparente du propos, des tracés de formes simples comme la droite ou le cercle peuvent faire l'objet de véritables découvertes scientifiques. Des algorithmes consacrés portent le nom du ou des auteurs de la découverte. Ainsi pour la droite les algorithmes de Bresenham, de Duluc, de Castle et Pitteway, de Berstel

[BOURDIN, 1996, p. 8 à 17]. De plus ce problème scientifique du tracé passionne toujours et il trouve encore d'autres solutions. Nous nous appuyons ici sur la proposition de Pierre Audibert [AUDIBERT, 1997, p. 157 à 164].

1.1 Tracer une droite

Tracer une droite discrète entre un pixel A et un autre B revient à chercher le chemin qui serait le plus proche d'une droite réelle les joignant comme deux points (figure 12.1). Une fois trouvées les coordonnées de chacun des pixels constitutifs de ce chemin, la fonction « Putpixel » pourra attribuer à chacun la couleur voulue pour le tracé.

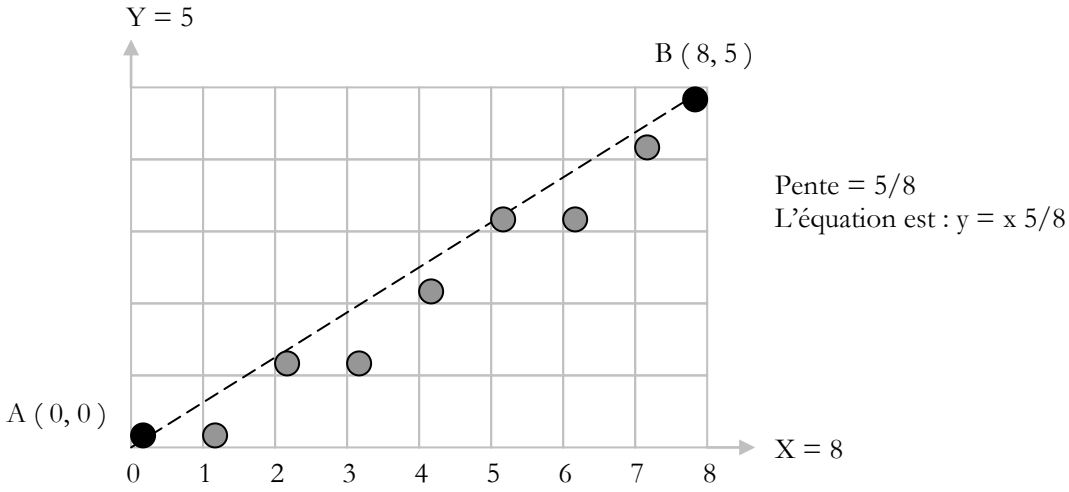


Fig. 12.1 : Tracer une droite appuyée sur un quadrillage de pixels

La pente est l'avancée horizontale sur l'avancée verticale, Y/X, soit 5/8 dans l'exemple de la figure 12.1. Pour aller de A (0, 0) à B (8, 5) il va falloir ajouter de façon régulière un horizontalement et un verticalement. La progression horizontale est constante en revanche la progression verticale est à trouver.

Principe

L'idée est de prendre y *valeur entière* au plus proche de la droite réelle en la suivant par en-dessous. L'équation dit que y est égal à x multiplié par la pente : y = x * (Y / X). On dispose ainsi des informations suivantes :

x = 0,	y = 0 * (5 / 8)	= 0 / 8	ce qui donne la valeur entière 0 et il reste 0
x = 1,	y = 1 * (5 / 8)	= 5 / 8	ce qui donne la valeur entière 0 et il reste 5
x = 2,	y = 2 * (5 / 8)	= 10 / 8	ce qui donne la valeur entière 1 et il reste 2
x = 3,	y = 3 * (5 / 8)	= 15 / 8	ce qui donne la valeur entière 1 et il reste 7
x = 4,	y = 4 * (5 / 8)	= 20 / 8	ce qui donne la valeur entière 2 et il reste 4
x = 5,	y = 5 * (5 / 8)	= 25 / 8	ce qui donne la valeur entière 3 et il reste 1
x = 6,	y = 6 * (5 / 8)	= 30 / 8	ce qui donne la valeur entière 3 et il reste 6
x = 7,	y = 7 * (5 / 8)	= 35 / 8	ce qui donne la valeur entière 4 et il reste 3
x = 8,	y = 8 * (5 / 8)	= 40 / 8	ce qui donne la valeur entière 5 et il reste 0

Pour chaque valeur x+1, y est augmenté de Y, c'est-à-dire de 5 dans l 'exemple. A chaque fois que le total obtenu en y dépasse la valeur de X qui est 8 dans l'exemple, la valeur entière résultat de la division augmente de un, ce qui donne le pixel de coordonnées (x+1, y+1), sinon il n'y a pas de changement en y et on a le pixel de coordonnées (x+1, y).

Il est possible de lire cette situation autrement que comme une division. En effet, pour chaque x on a une augmentation de Y. Stockons au fur et à mesure de l'évolution de x le résultat de cette addition de Y dans une variable nommée « result ». Au passage, pour chaque x, à chaque nouvelle addition de la valeur Y à la variable result, est regardé si result est supérieur ou égal à X. Si oui c'est que le résultat de la division (result / X) augmente de une unité. Dans ce cas c'est y qui est augmenté de une unité et X est ensuite soustrait à result. C'est-à-dire que result est toujours équivalent à « (x * Y) modulo X » ; pour chaque incrémentation x+1 de x, result correspond au reste de la division de l'équation x*(Y/X).

Dans notre exemple, au départ x est à 0, y est à 0 et la variable result est mise à 0. Pour chaque incrémentation x +1, on ajoute 5 à result. Si result demeure inférieur à 8, y conserve sa valeur. Si en revanche result est supérieur à 8, y augmente de un et 8 est soustrait de result ce qui laisse dans result, pour chaque incrémentation x+1 de x, le reste de la division : x * 5 / 8.

Traduction en langage C

Soit deux pixels, A de coordonnées entières (xdepart, yarrive) et B de coordonnées entières (xdepart, ydepart) , pour tracer la droite de A à B on a le bloc d'instructions suivant (figure 12.2) :

```
    {
(1)   int result;
      int X, Y,  i ;

(2)         X = xarrive – xdepart ;
            Y = yarrive – ydepart ;
            result = 0;
(3)         for( i = 0 ; i < X ; i++ ){
(4)               xdepart++;
(5)               result = result + Y;
(6)               if ( result >= X ){
(7)                     ydepart++ ;
(8)                     result = result – X ;
                  }
    }
```

Fig. 12.2 : Traduction en C du principe de base d'un algorithme de tracé de droite

(1) Déclaration des variables locales au bloc. La variable « result » de l'explication que nous avons donnée et les variables destinées à accueillir les valeurs X et Y que nous avons également mentionnées. La variable i est juste utilitaire pour l'itération dans la boucle for.

(2) Initialisation des trois variables X, Y, result. X est égal à la différence entre les coordonnées en x du point d'arrivée et du point de départ ; Y est égal à la différence entre

les coordonnées en y du point d'arrivée et du point de départ ; result est mise à 0

(3)(4)(5) Boucle qui passe en revue le nombre total des positions horizontales de 0 à X. Pour chacune la position en x du départ est incrémentée de un (4) et la variable result est augmentée de Y (5).

(6)(7)(8) Si la valeur de la variable result est égale ou dépasse la valeur de la variable X, c'est la position en y du départ qui est incrémentée de un (7), mais également la valeur de result est diminuée de X (8).

Dans cette explication nous avons supposé X et Y positifs et Y<X soit une pente comprise entre 0 et 1 (entre 0 et 45°) et il reste à généraliser aux autres cas de pente.

Généralisation de l'algorithme

Pour un point de départ A on peut mettre à part les lignes horizontale (Y = 0), les lignes verticales (X = 0) les diagonales (X = Y). Ensuite il reste deux cas : soit la pente est comprise entre 0 et 1 comme dans notre exemple, c'est-à-dire que (X > Y) soit la pente est supérieure à 1 avec (X < Y). Dans ce dernier cas pour conserver le même algorithme le plus simple est d'intervertir abscisse (X) et ordonnée (Y). C'est-à-dire de faire progresser y de façon constante plutôt que x, d'augmenter la variable result de X, de la soumettre à un test avec la valeur de Y, si elle est supérieure d'augmenter x et de diminuer la variable result de Y.

Notons également que le point d'arrivée peut se trouver à gauche du point de départ. Dans ce cas il faut décrémenter x de un plutôt que de l'incrémenter de un. De même le point d'arrivée peut être au-dessus ou au-dessous du point de départ et y doit être modifié en conséquence. Pour cette raison également les valeurs de X et de Y qui résultent de la soustraction entre coordonnées d'arrivée et de départ, peuvent être négatives. Il faudra d'une part déterminer le pas de 1 ou – 1 pour x et y et d'autre part effectuer les tests qui comparent X et Y en utilisant les valeurs absolues |X| et |Y|.

Pour tracer une droite nous obtenons une fonction telle que (figure 12.3) :

(1) Identification de la fonction « droite_audibert ». Elle prend 6 arguments, la bitmap de référence sur laquelle le tracé va être effectué, les coordonnées x et y du pixel de départ, les coordonnées du pixel d'arrivée et une variable pour la couleur.

(2) Déclaration des variables locales à la fonction.
Les variables X et Y pour les avancées horizontale et verticale.
La variable result qui pour chaque nouvelle valeur de x, va correspondre au reste de la division de l'équation x * (Y / X).

Les deux variables « absX » et « absY » sont là pour récupérer les valeurs absolues de X et Y. Les pas d'avancement « pasx » pour x et « pasy » pour y. Le pas de x est négatif si le pixel d'arrivée est à gauche du pixel de départ. Il est positif sinon.

```
(1)  Void droite_audibert ( BITMAP* bmp, int x, int y, int xarrive, int yarrive, int couleur )
     {
(2)    int X, Y, result, pasx, pasy, absX, absY, i ;

(3)          X = xarrive – x ;
             Y = yarrive – y ;
(4)          absX = ABS ( X );
             absY = ABS ( Y );
(5)          pasx = ( X > 0 ) ? 1 : – 1 ;
             pasy = ( Y > 0 ) ? 1 : – 1 ;
(6)          result = 0;
(7)          putpixel ( bmp, x, y, couleur );

(8)          if ( X == 0 ){
                  for ( i = 0 ; i < absY ; i++ ){
                       y += pasy ;
                       putpixel ( bmp, x, y, couleur );
                  }
             }
(9)          else if ( Y == 0 ){
                  for ( i = 0 ; i < absX ; i++ ){
                       x += pasx ;
                       putpixel ( bmp, x, y, couleur );
                  }
             }
(10)         else if ( absX == absY ){
                  for ( i = 0 ; i < absX ; i++ ){
                       x += pasx ;
                       y += pasy ;
                       putpixel ( bmp, x, y, couleur );
                  }
             }
(11)         else if ( absX > absY){
                  for ( i = 0 ; i < absX ; i++ ){
                       x += pasx ;
                       result += absY ;
                       if ( result >= absX ){
                            result –= absX ;
                            y += pasy ;
                       }
                       putpixel ( bmp, x, y, couleur );
                  }
             }
(12)         else{
                  for ( i = 0 ; i < absY ; i++ ){
                       y += pasy ;
                       result += absX ;
                       if ( result >= absY ){
                            result –= absY ;
                            x += pasx ;
                       }
                       putpixel ( bmp, x, y, couleur );
             }
     }
```

Fig. 12.3 : Fonction pour un tracé de droite selon un algorithme de Pierre Audibert

En ce qui concerne le pas de y, il faut tenir compte du fait que la première ligne d'une image bitmap est celle du haut d'indice 0. L'origine est la position « bmp->line[0][0] » selon la structure de données exposée dans le chapitre précédent. Ainsi le pas de y est négatif si le pixel d'arrivée est au-dessus du pixel de départ. Il est positif si le pixel d'arrivée est au-dessous du pixel de départ (figure 12.4)

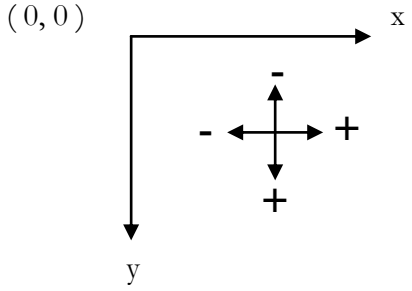


Fig. 12.4 : Illustration repérage dans les coordonnées de l'image bitmap selon l'origine en haut à gauche

La variable « i » est utilisée pour les boucles for.

(3) Première étape, récupération des avancées horizontale et verticale dans les variables respectives X et Y

(4) Récupération des valeurs absolues |X| et |Y| dans les variables absX et absY en utilisant la macro « ABS(n) » détaillée dans le chapitre précédent (à propos des expressions conditionnelles)

(5) Détermination de pasx et pasy en utilisant une expression conditionnelle. Si X est positif, pasx est positif, si X est négatif, pasx est négatif. Idem pour Y, s'il est positif, pasy est positif et s'il est négatif pasy est négatif.

(6)(7) Mise à 0 de la variable result et affichage dans la couleur souhaitée et passée en argument, du premier pixel du tracé, celui de la position de départ.

(8) Cas de la ligne verticale. Il n'y a pas de variation en x et il suffit d'ajouter itérativement pasy à y selon la taille |Y|, et d'afficher à chaque fois le nouveau pixel obtenu pour obtenir la droite.

(9) Cas de la ligne horizontale. Il n'y a pas de variation en y et il suffit d'ajouter itérativement pasx à x selon la taille |X| et d'afficher pour chaque nouveau x le pixel obtenu dans la couleur souhaitée pour tracer la droite.

(10) Cas des diagonales. Les valeurs absolues de X et Y sont égales, x et y sont incrémentés chacun de pasx et pasy selon toute la longueur du côté. Chaque nouveau pixel trouvé est affiché dans la couleur voulue.

(11) La pente est comprise entre 0 et 1 comme dans le cas de la figure 12.1. La boucle for passe en revue toutes les positions horizontales x de 0 à |X|. Pour chacune x est incrémenté de pasx. La variable result est additionnée de |Y|. Si elle devient supérieure ou égale à |X| alors result est décrémentée de |X| et y est incrémenté de pasy. A chaque fois le pixel trouvé est affiché avec la couleur voulue.

(12) Dernier cas où la pente est supérieure à 0. L'astuce consiste à inverser les valeurs de x et |X| avec y et |Y|. Ainsi la boucle passe en revue toutes les positions verticales de 0 à |Y|. Pour chacune y est incrémenté de pasy et result est incrémentée de la valeur absolue |X|. Si result est supérieure ou égale à |Y| alors result est décrémentée de |Y| et x est incrémenté de pasx. Le pixel trouvé est affiché avec la couleur voulue.

Tracés dérivés

A partir du tracé d'une droite, il est possible d'obtenir différentes autres formes géométriques. Pour un triangle on fait une fonction qui prend trois points en argument afin de les relier. Pour un rectangle on passera deux points qui donnent les coordonnées haut gauche et bas droite du rectangle. Pour les autres parallélogrammes il faudra quatre points. Pour les polygones ils varieront selon le nombre des points à relier.

Par le biais de ces formes une relation s'établit avec tout un patrimoine de connaissances mathématiques et de géométrie[44]. Souvent des connaissances de bases en géométrie s'avèrent extrêmement utiles. Mais parfois il ne faut pas hésiter à envisager l'algorithmique comme une sorte de « mécano ». L'astuce et la bidouille deviennent alors des alliés précieux lors de l'élaboration d'un programme, à condition de respecter la rigueur nécessaire du point de vue méthodologique.

Par ailleurs un algorithme de tracé qui à l'origine réalise, selon telle ou telle contrainte, un chemin entre des pixels, peut être utilisé pour d'autres actions. Par exemple un tir de vaisseau spatial pourra emprunter le même algorithme que celui de la droite afin de déterminer la trajectoire d'un rayon désintégrateur.

Dans ce cas pour définir la droite on utilisera un point de départ et plutôt qu'un point d'arrivée, un vecteur v (v_x , v_y) qui donne à la droite sa direction et son sens. Les coordonnées de la droite, à partir du point de départ D (xd, yd) seront tous les pixels dont les coordonnées vérifient $x = xd + Kv_x$ et $y = yd + Kv_y$ avec K nombre réel. La droite se déroulera jusqu'à ce que par exemple elle rencontre un obstacle et là le calcul devra s'arrêter. Notons que sur cette base il est possible de commencer à élaborer un billard.

1.2 Tracer un cercle

Toujours sur un quadrillage de pixels il s'agit de construire un chemin qui ressemble au plus près à la circonférence d'un cercle à partir d'un pixel donné comme centre du cercle et d'un rayon.

L'équation du cercle est $x^2 + y^2 = R^2$. On peut en déduire une fonction $F = x^2 + y^2 – R^2$ telle que le point de coordonnées (x, y) est sur le cercle si et seulement si F = 0. Pour simplifier, la recherche des pixels est effectuée uniquement sur un quart du cercle et les autres positions sont obtenues ensuite par symétrie (figure 12.5).

[44] Les travaux de Pierre Audibert dans ce domaine et élargi à des problématiques de hasard, de chaos, de fractales et de pavages sont extrêmement riches et pratiquement uniques, impossibles à trouver actuellement dans le monde de l'édition. Consulter [AUDIBERT, 1995], [AUDIBERT, 1997], [AUDIBERT, 2000]

On part de (0, R) et F = 0. Soit on avance horizontalement, soit on remonte verticalement. Le point choisi est celui, positif ou négatif, qui est le plus proche de 0. Pour ce faire on compare en valeur absolue les deux valeurs :
$F_1 = (x+1)^2 + y^2 - R^2$, et $F_2 = x^2 + (y+1)^2 - R^2$.
C'est-à-dire les expressions $F_1 = 2x+1$ et $F_2 = -2y+1$.
Si $|F_1|$ est inférieur à $|F_2|$ on avance horizontalement en x sinon on remonte verticalement en y.

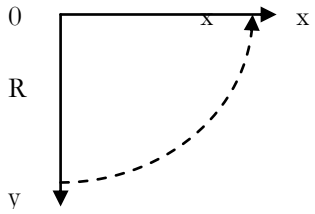


Fig. 12.5 : Construire un cercle

Soient six entiers x, y, R, F, F_1, F_2 la boucle du programme est en langage C (figure 12.6) :

```
(1) x = 0 ; y = R ; F = 0 ;
(2) while ( y >= 0 ){
(3)     F1 = F + 2x + 1 ;
        F2 = F –  2y + 1 ;
(4)     if ( ABS ( F1 ) < ABS ( F2 ) ) {
            x++;
            F = F1 ;
        }
(5)     else{
            y– – ;
            F = F2 ;
        }
  }
```

Fig. 12.6 : Boucle pour la construction du premier quart du cercle

(1) Initialement x est à 0, y est égal au rayon et la valeur de F est 0.

(2) La boucle n'est pas une boucle for. C'est une variété du principe de boucle, la boucle « tant que … ».

Boucle while

Dans l'écriture :

while (*expression*)
instruction

Tant que la valeur de l'*expression* de contrôle est vraie et renvoie la valeur 1, les instructions sont effectuées. Si *expression* renvoie 0, par exemple un test faux qui renvoie la valeur 0, alors la boucle est terminée [KERNIGHAN, 1995, p. 60] [BRAQUELAIRE, 1998, p. 145].

Au départ y est égal à la longueur du rayon. Au fur et à mesure que la courbe remonte il diminue et avec le rayon à l'horizontal il sera égal à 0, ce qui marque la fin de la boucle while avec le tracé du quart de cercle.

(3)(4)(5) Calcul des deux valeurs de F_1, F_2. C'est la plus petite des deux qui est sélectionnée. Si $|F_1|$ est la plus petite alors c'est la valeur horizontale, x, qui est incrémentée de 1. La valeur F_1 est conservée dans F pour le calcul suivant de F_1 et F_2.
Si $|F_2|$ est la plus petite alors c'est la valeur verticale, y, qui remonte et est décrémentée de 1. Et c'est la valeur F_2 est conservée dans F pour le calcul suivant de F_1 et F_2.

Cas général

Tracer le cercle de centre (x_0, y_0) et de rayon R. Le centre n'est plus (0, 0) comme dans précédemment ; au départ on a $x = x_0$ et $y = y_0+R$. Pour avoir les x et y comme précédemment il faut soustraire x_0 de x et y_0 de y. Ce qui donne une légère modification de l'équation du cercle avec $(x – x_0)^2 + (y – y_0)^2 = R^2$; soit $F = (x – x_0)^2 + (y – y_0)^2 - R^2$.
Les valeurs F_1 et F_2 sont alors obtenues par : $F_1 = 2 (x – x_0) + 1$ et $F_2 = - 2 (y – y_0) + 1$.
La symétrie horizontale est obtenue avec $2x_0 – x$ et la symétrie verticale avec $2y_0 – y$ (figure 12.7).

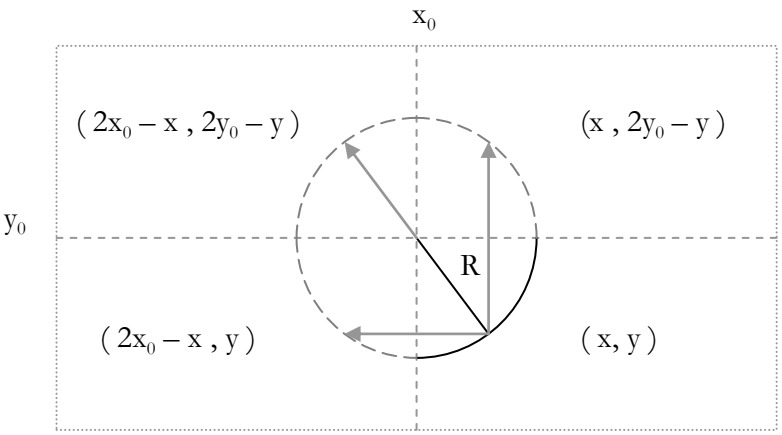


Fig. 12.7 : Symétrie du quart de cercle

Ainsi pour tracer un cercle nous pouvons utiliser la fonction (figure 12.8) :

(1) Identificateur de la fonction nommée « cercle_audibert ». La fonction ne renvoie rien mais prend cinq arguments. La référence d'une bitmap (sur laquelle le tracé va être effectué) la position du centre du cercle, le rayon et la couleur souhaitée.

(2)(3) Déclaration des variables locales à la fonction et initialisation de : x à partir du centre, y à partir du centre et du rayon, F est mise à 0.

(4) Mise en place de la boucle while, tant que y est inférieur à yo les instructions du bloc de la boucle sont effectuées.

(5) Affichage via la fonction « putpixel » standard du point courant ainsi que des trois autres qui lui sont symétriques sur le cercle.

(6)(7)(8) Calcul des valeurs F1 et F2 pour choisir le mouvement horizontal ou vertical afin de déterminer le pixel au plus proche du cercle. Si F1 est la plus petite valeur, c'est l'avancée horizontale et x est incrémenté de 1. La valeur de F pour le point courant devient F1. Si F2 est la plus petite valeur, le mouvement est vertical et y est diminué de 1. La valeur de F pour le point courant est alors F2.

```
(1)  void cercle_audibert( BITMAP* bmp, int xo, int yo, int R, int couleur)
     {
(2)   int x, y, F1, F2, F ;

(3)          x = xo ;
             y = yo + R ;
             F = 0 ;
(4)          While ( y >= yo ){

(5)                  putpixel ( bmp, x, y, couleur ) ;
                     putpixel ( bmp, 2xo – x, y, couleur ) ;
                     putpixel ( bmp, 2xo – x, 2yo – y, couleur ) ;
                     putpixel ( bmp, x, 2yo - y, couleur ) ;

(6)                  F1 = F + 1 + 2 *( x – xo ) ;
                     F2 = F + 1 + 2 *( y – yo ) ;

(7)                  if( ABS ( F1 ) < ABS ( F2 ) ){
                          x+=1;
                          F = F1;
                     }
(8)                  else{
                          y – = 1 ;
                          F = F2;
                     }
     }
```

Fig. 12.8 : Fonction pour un tracé de cercle selon un algorithme de Pierre Audibert

Pour colorer un cercle il suffit de diminuer le rayon et de tracer chaque cercle intermédiaire de la ou des couleurs voulues jusque au rayon nul.

Extrapolations ludiques

Dans le chapitre Sept à propos du logiciel MIKADO nous avons mentionné un processus proposé par Pierre Audibert fait à partir de cet algorithme de tracé de cercle : réaliser graphiquement des diagrammes de Voronoï. Au départ des points, chacun d'une couleur différente, sont jetés à l'écran. Ensuite chacun produit un disque de couleur dont le rayon augmente progressivement. La contrainte est que les disques ne peuvent pas mordre les uns sur les autres. Si donc il y a déjà une couleur à une position de pixel, elle demeure inchangée. Ainsi la croissance des cercles les amène à se rencontrer et tracer les frontières qui les séparent.
Une autre idée suggérée par Pierre Audibert est de simuler une croissance de nénuphars à la surface d'une mare. C'est en nous appuyant sur cette idée que nous avons imaginé un monde de nénuphs.

Tout ce qui a trait au graphisme et qui peut être interprété par l'intermédiaire d'un robot peut bien entendu être qualifié de robographe. *Mais Il important de bien sentir à quel point dans l'espace robographique, tout ce qui initialement appartient au graphisme, soit n'est plus limité au graphisme tel qu'on l'entend généralement à travers la construction de formes, soit est élargi à des dimensions scénaristiques telles que la forme doit être comprise dans le cadre d'un processus et d'un développement*

historico-algorithmique. C'est-à-dire que de la forme à l'entité quasi autonome l'écart est devenu très étroit dans l'espace robographique.

2. Compositions de pixels

Parmi les robographes reliés au graphisme classique il n'y a pas que des traceurs de traits. Par exemple il peut s'agir de trouver le contour d'une forme en fonction de sa couleur. C'est un peu une proposition inverse des deux précédentes. L'objectif n'est pas de calculer un tracé mais de retrouver dans une image un tracé. C'est-à-dire de retrouver les pixels, éléments qui composent une forme et d'être en mesure d'en rebâtir l'ensemble, de s'intéresser à une composition existante. L'idée de composition n'est pas de construire des formes visuellement plus complexes à partir de formes simples, comme peut le laisser entendre le propos de Françoise Holtz Bonneau que nous avons rapporté au chapitre Huit à propos de l'écriture de machine-image. Avec la composition de pixels c'est plutôt la nature même des formes qui peut être abordée.

L'algorithme que nous proposons [AUBIBERT, 1997, p. 162] permet de repérer toutes sortes de différentiels et de frontières qui séparent des zones. Ce peut être le moyen de représenter l'équation d'une courbe. Si f(x, y) = 0 les points sont sur la courbe et ils sont d'un côté ou de l'autre selon que l'on a f(x, y) > 0 ou f(x, y) < 0. Par exemple il est possible de séparer le plan en deux parties à partir de la médiatrice de deux points. Il est possible également d'utiliser cet algorithme pour visualiser un ensemble fractal.

C'est cet algorithme qui nous a donné l'idée du concept de robographe. Il n'a rien de mathématique et fonctionne comme une sorte de petite araignée. Une araignée rotative relativement facile à se représenter et qui est en elle-même sans recours à des connaissances ou modèles extérieurs : une pure production robographique.

2.1 Contours d'une forme

L'objectif, comme illustré par la figure 12.9, est de repérer une forme par un différentiel de couleur entre la forme et son fond.

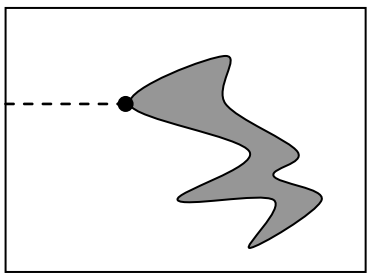


Fig. 12.9 : Repérer une forme

La première chose est d'avoir une fonction qui indique si le pixel courant est couleur fond ou pas. C'est-à-dire s'il est dans la forme ou pas. Pour ce faire nous utilisons la fonction standard « getpixel » (figure 12.10 ci-dessous) :

Fig. 12.10 : Evaluer si un pixel est en dehors ou en dedans de la forme par un différentiel de couleur

```
(1)  #define OUI         1
     #define NON         0

     int CFOND = 0 ;

(2)  int endehors ( BITMAP* bmp, int x, i nt y)
     {
(3)       if ( getpixel ( bmp, x, y) == CFOND )
               return OUI ;
          return NON ;
     }
```

(1) Définition des macros OUI et NON qui correspondent respectivement aux valeurs 1 et 0. C'est accessoire, juste pour rendre le propos de la fonction plus lisible. Ensuite déclaration de « CFOND », variable globale dans le programme qui va prendre la valeur de la couleur de fond. Par exemple, en 8 bits, disons que la première couleur de la palette est noire, utilisée pour le fond d'écran. Eventuellement CFOND pourra changer de valeur, soit que le fond change, soit que l'on veuille détourer une forme à l'intérieur d'une autre forme.

(2) Identification de la fonction « endehors ». Elle prend trois arguments : une bitmap de référence sur laquelle ont lieu les opérations, et deux entiers pour la position d'un pixel. Elle renvoie le résultat, un entier OUI ou NON.

(3) Si le pixel à la position donnée (x, y) est de la couleur du fond alors c'est qu'il est en dehors et la fonction retourne OUI si ce n'est pas le cas, le pixel n'est pas sur le fond et la fonction retourne NON

Mais l'important va être la façon d'interroger les pixels de l'image. En quelque sorte comment utiliser la fonction « endehors » afin qu'elle serve à entourer la forme cherchée.
Le premier point est d'aborder la forme. Pour ce faire on lance un grappin à partir d'un bord de l'écran, par exemple du bord gauche comme sur la figure 12.9 vue plus haut et visible en pointillés sur la figure 12.11 a.

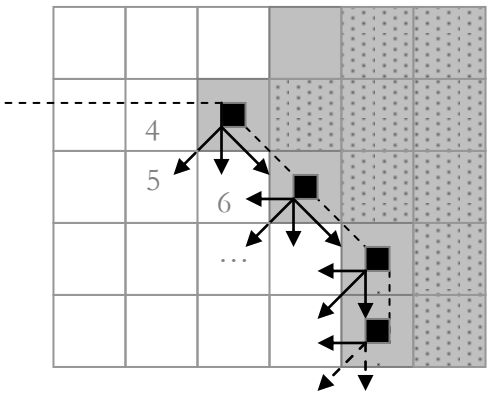
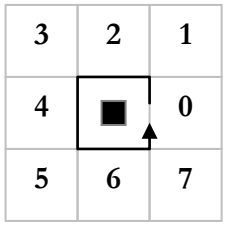


Fig. 12.11 (a) : Mécanisme de progression pour la découverte des bords de la forme



(b) : Les positions voisines d'un pixel, sens du parcours de la recherche

Il s'agit d'une boucle qui, pour une hauteur y, fait avancer la valeur horizontale de x de un en un, du bord jusqu'à rencontrer la forme, c'est-à-dire jusqu'à ce que la fonction « endehors » retourne NON et qu'un premier pixel appartenant à la forme soit trouvé.

A partir de ce premier pixel, on conserve la position voisine du pixel précédent selon le numéro d'ordre indiqué par la figure 12.11 b. La position voisine est une valeur entre 0 et 7. Pour le premier pixel trouvé, le précédent est à la position 4 (figure 12.11 a). La recherche d'un deuxième pixel appartenant à la forme et voisin du premier peut donc commencer à partir de la position voisine qui est la position 5. Sur la figure 12.11 a, le second pixel trouvé est le pixel de la position 7.

Une fois ce pixel trouvé et à partir de lui, la position voisine du pixel chercheur, c.à.d le pixel trouvé précédent, est conservée. Cette position est égale à la position du nouveau pixel plus 4 et modulo 8. Par exemple si le pixel trouvé est à la position 7, la grille des voisins numérotés de 0 à 7 est déplacée et prend comme centre cette position 7, et par rapport à cette nouvelle référence, le pixel précédent est bien à la position (7+4) %8 = 3. Dans ce cas la nouvelle recherche part sur la position suivante 4.

De fil en aiguille tout le tour de la forme va être effectué. En fonction de ce que l'on veut on pourra soit stocker les positions des pixels constitutifs de ce collier, soit colorer le contour. La fin du processus a lieu lorsque après le tour de la forme le pixel du départ est retrouvé.

En langage C l'algorithme peut prendre la forme de la fonction suivante (figure 12.12):

(1) Il s'agit d'un commentaire qui donne une indication de comment obtenir le premier pixel intérieur de la forme. Nous choisissons de détacher cette première opération de recherche du contour afin de pouvoir en avoir plusieurs versions. Admettons en tous cas qu'à l'issue de (1) quelque part dans le programme, on ait les coordonnées du premier pixel d'une forme à détourer sur une bitmap donnée, ainsi que l'indication de position du pixel précédent par lequel on est arrivé sur cette forme. Ce sont les quatre arguments passés ensuite à la fonction « contour ». Quelle que soit la façon dont on s'y prenne ces informations sont nécessaires au fonctionnement de cet algorithme de détourage.

Ici nous donnons l'exemple d'une initialisation à 100 pour y , la valeur de x est déterminée par la boucle while qui incrémente x de un en un tant que la premier pixel dans la forme n'est pas trouvé et que la fonction « endehors » renvoie OUI. La boucle s'arrête si « endehors » renvoie NON avec x à la bonne position. Le pixel trouvé est le voisin de la position 0 et à partir de lui la position précédente est 4. C'est pourquoi la variable « dern » est initialisée à 4 afin de conserver cette information.

(2) Identification de la fonction « contour ». La fonction ne renvoie rien et prend quatre arguments, une image de référence sur laquelle le contour est recherché à partir de la position en x et y du premier pixel déjà trouvé ainsi que l'indication de position du pixel qui a juste précédé la découverte du premier.

(3) Déclaration des variables internes à la fonction. « xdepart » et « ydepart » sont initialisés à la déclaration sur les valeurs passées de la position du premier pixel en x et y. Ses valeurs sont conservées afin de construire le test de fin de la boucle qui consiste à retrouver le

pixel du début. Les deux tableaux « xv[8] » et « yv[8] » sont destinés à stocker, selon la position (x, y) de chaque nouveau pixel trouvé, toutes les positions voisines. La variable « couleur » prend au hasard une valeur entre 1 et 255 (nous supposons être en 8 bits et que le fond correspond à l'indice 0 de la palette). La variable « i » est utilitaire pour une question de lisibilité.

```
(1)/* recherche du premier point intérieur
     BITMAP* bmp ;
     int x, y, dern ;
            y = 100 ;
            x = 0 ;
            while (endehors ( bmp, x, y ) == OUI  )
                  x++ ;
            dern = 4 ;
     */

(2)  void contour ( BITMAP * bmp, int x, int y, int dern  )
     {
(3)  int xdepart = x ;
     int ydepart = y ;
     int xv[ 8 ] ;
     int yv[ 8 ] ;
     int couleur = (rand()%255) + 1 ;
     int i ;

(4)          do {
(5)                  putpixel ( bmp, x, y, couleur ) ;

(6)                  xv[ 0 ] = ( xv[ 1 ] = ( xv[ 7 ] = x + 1 ) ) ;
                     xv[ 2 ] = ( xv[ 6 ] = x ) ;
                     xv[ 3 ] = ( xv[ 4 ] = ( xv[ 5 ] = x – 1 ) ) ;

                     yv[ 1 ] = ( yv [ 2 ] = ( yv [ 3 ] = y – 1 ) ) ;
                     yv[ 0 ] = ( yv [ 4 ] = y ) ;
                     yv[ 5 ] = ( yv [ 6 ] = ( yv [ 7 ] =  y + 1 ) ) ;

(7)                  i = ( dern + 1 )%8 ;

(8)                  while ( endehors ( bmp, vx[ i ], vy[ i ] ) == OUI )
                           i = ( i + 1 )%8 ;

(9)                  x = vx[ i ];
                     y = vy[ i ];
(10)                 dern = ( i + 4 )%8 ;

(11)           }while ( x != xdepart || y != ydepart ) ;
     }
```

Fig. 12.12 : Fonction de détourage d'une forme par différentiel de couleur sur fond continu. Algorithme de Pierre Audibert.

(4)(11) Corps de la boucle du processus de découverte du contour.

Do while

C'est une variante de la boucle while. La syntaxe du « do » est :

```
do
    instructions
while( expression ) ;
```

C'est-à-dire que les instructions sont effectuées avant le test et l'assurance est donnée qu'elles seront exécutées au moins une fois. Ensuite comme d'habitude, si le test est vrai les instructions sont à nouveau exécutées sinon la boucle est finie.

Donc les instructions sont exécutées avant le test, au moins une fois. Et le test final (11) regarde si les nouvelles positions en x et y correspondent aux positions de départ. Si l'une des deux est différente de sa valeur de départ, le test est vrai, la recherche continue, les instructions reprennent.

(5) La première instruction est l'affichage du premier pixel dans la couleur déterminée au hasard avec la variable « couleur ».

(6) Ensuite selon les positions x et y courantes, toutes les positions adjacentes sont stockées dans deux tableaux, un pour les x et un pour les y. En se référant à l'ordre de la figure 12.11.b on passe en revue toutes les positions : à droite de x, positions 0, 1, 7, c'est (x + 1) et à gauche, positions 3, 4, 5, c'est (x – 1). Au-dessus de y, positions 1, 2, 3, c'est (y – 1) et au dessous, positions 5, 6, 7, c'est (y + 1).

(7) C'est l'initialisation du compteur de positions adjacentes. La première position adjacente à regarder suit la position occupée par le pixel trouvé précédemment. La position du pixel précédent par rapport au pixel courant est stockée dans la variable « dern ». Il faut donc lui ajouter 1 et opérer un modulo 8 pour rester dans le comptage des valeurs de positions de 0 à 7. Ce résultat est affecté à la variable i.

(8) Jusqu'à trouver le nouveau pixel, les positions adjacentes du pixel courant sont passées en revue à partir de la position stockée dans la variable i. Pour ce faire la fonction « endehors » est appelée pour chaque position adjacente avec en argument les coordonnées stockée dans les tableaux xv et yv compte tenu de l'indice i de position. Tant que la réponse est OUI, c'est-à-dire l'on est en-dehors, la revue est continuée ce qui se traduit par l'incrémentation de la variable i afin de tester la position suivante. Dès que la position du nouveau pixel du contour est trouvée, la boucle prend fin ;

(9) Les coordonnées de la position trouvée deviennent les coordonnées (x, y) de la position courante ce qui se traduit par l'affectation des valeurs de position à l'indice i des tableaux xv pour x et yv pour y.

(10) Le pixel courant est donc le nouveau pixel trouvé. Par rapport à lui la valeur de position de l'ancien est (i+4)%8 et cette valeur est sauvegardée dans la variable « dern ». Si le test de la boucle while est vrai (la nouvelle valeur de x n'est pas celle du départ ou la nouvelle valeur de y n'est pas celle du départ) tout reprend à (5).

Ce principe algorithmique de la fonction « contour » est employable pour tracer une courbe du type f(x,y).

2.2 Courbes dérivées de contours

Les points sur la courbe vérifient f(x, y) = 0, les autres sont soit d'un côté soit de l'autre selon que f(x, y) < 0 ou que f(x, y) > 0. En d'autres termes le différentiel recherché n'est plus entre la couleur de fond d'écran et celle interne à une forme simple, mais entre appartenir ou pas à une courbe.

Tout d'abord, il faut, pour une position donnée, déterminer son signe par rapport à une courbe, positif d'un côté négatif de l'autre. Une première fonction « signe_position » détermine le signe d'une position, ce qui permet ensuite de le comparer avec celui des positions voisines et de retrouver le mécanisme de recherche de la fonction « contour ».

La comparaison du signe de deux positions est l'objet d'une seconde fonction « meme_signe_que » qui à la différence de la fonction « endehors » devra prendre en argument le signe de référence, positif ou négatif, du côté par lequel on commence la recherche. Ainsi, quelle que soit la position de départ que l'on choisisse, il faut :
1) déterminer le signe du côté où l'on est par rapport à la courbe.
2) Trouver la courbe, c'est-à-dire le différentiel engendré par la fonction. Il est recherché avec le même processus de grappin proposé en (1) figure 12.12 avec la fonction « contour ».
3) Ce différentiel trouvé, l'algorithme de contour permet de le longer.

Notons qu'à la différence d'une forme pleine comme précédemment, la courbe est une ligne probablement non fermée, peut-être avec des boucles internes. Nous devrons tenir compte des limites de l'écran notamment dans la fonction « meme_signe_que ».

Fonction « signe_position »

La fonction « signe_position » nécessite de définir au moins une courbe. Dans le cadre de la réalisation des programmes MIKADO et BUISSON PLATONIQUE présentés au chapitre 7, nous avons découvert par tâtonnement quelques formules. Cependant nous nous appuierons ici sur deux exemples simples : la formule d'une courbe elliptique et la médiatrice d'un segment, l'objectif étant le principe général et non telle ou telle fonction particulière.
Le fonctionnement de base de la fonction est le suivant (figure 12.13) :

```
(1)  int signe_position(  int x, int y  )
     {
(2)       if(    /* formule équation f ( x, y )  */   > 0 )
               return 1;
          else
               return -1;
     }
```

Fig. 12.13 : Début de la fonction signe_equation

(1) Identification de la fonction « signe_position ». Elle prend deux entiers en argument qui correspondent à la position d'un pixel. Elle renvoie le signe de la position par rapport à une formule du type f (x, y). 1 pour positif et – 1 pour négatif.

(2) Test sur le résultat de la formule. Si le test est vrai le résultat est positif et la fonction renvoie 1, sinon le résultat est négatif (ou nul) et la fonction renvoie – 1.
A titre indicatif, Pierre Audibert propose dans son cours pour obtenir une courbe elliptique la formule :

(y – 240)*(y – 240)*15 – (x – 320)*(x – 320)*(x – 320)/7 + A*(x – 320) – 100 > 0

Avec le paramètre « A » à faire varier dans la fourchette de – 1000 et 14000 [AUDIBERT, 1997, p. 164].
Une autre possibilité est de tracer la médiatrice de deux points A et B. La médiatrice du segment [AB] est la droite perpendiculaire à (AB) et qui passe par le milieu de [AB]. Le point M appartient à la médiatrice s'il est à égale distance de A et de B ; si MA = MB, MA – MB = 0. En revanche MA < MB indique que M est plus près de A et à l'inverse, MA > MB indique que M est plus près de B. Cet aspect peut être pratique dans une situation de jeu pour déterminer par exemple quel ennemi est plus près du héros.

Admettons A et B connus et de coordonnées A (xa, ya) et B (xb, yb).
Soit M (x, y), on a $MA^2 = (x – xa)^2 + (y – ya)^2$ c'est-à-dire :
MA = $\sqrt{(x - xa)^2 + (y - ya)^2}$ et la même chose pour B :
MB = $\sqrt{(x - xb)^2 + (y - yb)^2}$. On forme ensuite l'expression f(x, y) = MA – MB ce qui donne : f (x, y) = $\sqrt{(x - xa)^2 + (y - ya)^2}$ - $\sqrt{(x - xb)^2 + (y - yb)^2}$
Si f(x, y) = 0 , le point M (x, y) est sur la médiatrice.
Si f (x, y) > 0, M (x, y) est sur un côté, du côté de B et
si f (x, y) < 0, M (x, y) est sur l'autre côté, du côté de A.

La fonction standard « sqrt » de la librairie <math.h> du C donne la racine carrée d'un nombre. Ainsi le test dans la fonction « signe_position » devient :

```
if ( sqr( (x – xa)*(x – xa)+(y – ya)*(y – ya) ) – sqr( (x – xb)*(x – xb)+(y – yb)*(y – yb) ) >0)
```

Fonction « meme_signe_que »

Quelle que soit la courbe recherchée, ensuite nous avons besoin de la fonction « meme_signe_que » qui s'appuie sur le résultat de « signe_position » (figure 12.14) :

(1) Définition de deux macros qui indiquent quelle est la taille de l'écran et la résolution souhaitée du programme.

(2) Identification de la fonction « meme_signe_que » qui prend trois entiers en arguments ; x et y sont pour une position et « ref_signe » pour le signe de référence, trouvé au préalable à partir d'un pixel de départ et progression avec appel de la fonction « signe_position ». « meme_signe_que » renvoie un entier : OUI si effectivement la position (x, y) est du même signe que ref_signe et NON dans le cas contraire.

(3) Test sur le retour de l'appel à la fonction « signe_position » pour la position (x, y)

passée en argument. S'il est identique à la référence, ref_signe, qui est passée en argument, « meme_signe_que » renvoie OUI.

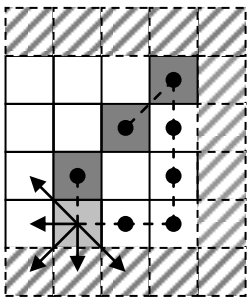
```
(1)  #define TX          800
     #define TY          600

(2)  int meme_signe_que(  int x, int y, int ref_signe  )
     {
(3)       if(   signe_position( x, y ) == ref_signe || x < 0 || x > TX  ||  y < 0 || y > TY )
               return OUI ;
          return  NON ;
     }
```

Fig. 12.14 : Renvoyer le signe d'une position en fonction de l'équation

Mais la fonction renvoie OUI également si x ou y sortent de l'écran. C'est-à-dire que tous les points extérieurs à l'écran sont forcés au même signe que ref_signe et sont considérés comme du même côté de la courbe que celui de ref_signe. En principe la ligne représente le front entre les deux bords tracés par la courbe. De ce fait, l'autre côté de la courbe est l'autre bord qui va être longé par l'extérieur de l'écran, jusqu'à retrouver la courbe à l'endroit où elle coupe l'écran. La situation est la suivante (figure 12.15) :

Fig. 12.15 : Question des bords de l'écran.



Nous arrivons sur un pixel, en gris plus clair, limitrophe de la fin de l'écran qui est hachurée. Pour les positions voisines 3 et 4, du pixel gris clair la fonction « meme_signe_que » renvoie OUI. Lorsque les positions voisines sont extérieures à l'écran, « meme_signe_que » renvoie encore OUI etc. jusqu'à retrouver le différentiel, représenté avec les cases marquées d'un point noir et l'indication en pointillés du parcours du contour.

Fonction « dessine_courbe »

Enfin la fonction « dessine_courbe » (figure 12.16) reprend ensuite l'essentiel de la fonction « contour » :

(1) Identification de la fonction « dessine_courbe ». Elle prend une image de référence, ce peut être l'image écran pour affichage à l'écran ou une image en mémoire juste pour le calcul.

```
(1)  void dessine_courbe(  BITMAP* bmp  )
     {
(2)  int x = 0 ;
     int y = TY / 2 ;
     int ref_signe ;
     int xdepart, ydepart ;
     int couleur = ( rand()%255 ) + 1 ;
     int xv[ 8 ], yv[ 8 ] ;
     int dern , i ;

(3)       ref_signe = ( signe_position( x, y ) == 1 ) ? 1 :  - 1 ;

(4)       while ( meme_signe_que ( x, y, ref_signe ) == OUI )
                  x++ ;

(5)       dern = 4 ;
(6)       xdepart = x ;
          ydepart = y ;

(7)       do {
(8)                 putpixel ( bmp, x, y, couleur ) ;

(9)                 xv[ 0 ] = ( xv[ 1 ] = ( xv[ 7 ] = x + 1 ) ) ;
                    xv[ 2 ] = ( xv[ 6 ] = x ) ;
                    xv[ 3 ] = ( xv[ 4 ] = ( xv[ 5 ] = x – 1 ) ) ;

                    yv[ 1 ] = ( yv [ 2 ] = ( yv [ 3 ] = y – 1 ) ) ;
                    yv[ 0 ] = ( yv [ 4 ] = y ) ;
                    yv[ 5 ] = ( yv [ 6 ] = ( yv [ 7 ] =  y + 1 ) ) ;

(10)                i = ( dern + 1 )%8 ;

(11)                while ( meme_signe_que ( vx[ i ], vy[ i ], ref_signe ) == OUI )
                         i = ( i + 1 )%8 ;

(12)                x = vx[ i ];
                    y = vy[ i ];
(13)                dern = ( i + 4 )%8 ;

(14)          } while ( x != xdepart || y != ydepart ) ;
     }
```

Fig. 12. 16 : Fonction de contour d'une courbe selon un algorithme de Pierre Audibert.

(2) Déclaration des variables locales de la fonction. A la différence de la fonction « contour » vue précédemment (figure 12.12) la découverte du premier pixel est intégrée dans le corps de la fonction. C'est uniquement par souci de lisibilité du processus (On pourrait bien sûr chercher le premier pixel en-dehors de la fonction et lui passer ses coordonnées en argument ensuite). La position en x est initialisée à 0 et la position en y est initialisée à la taille de l'écran divisée par 2.
La variable « ref_signe » va permettre de conserver le signe du côté de la courbe trouvé à la position (x, y) de départ.

Les variables « xdepart » et « ydepart » fonctionnent comme celles de la fonction « contour » : conserver les positions de départ pour le test de fin de la boucle while principale. De même une variable pour la couleur est initialisée au hasard et deux tableaux destinés à conserver les positions adjacentes de x et y afin d'en comparer les valeurs. Les variables « dern » et « i » servent au compteur des positions adjacentes.

(3) La fonction « signe_position » retourne une valeur pour la position (x, y) de départ. Nous avons utilisé une expression conditionnelle. Si le test est vrai le resultat est 1 et – 1 sinon. Ce résultat est affecté à la variable ref_signe.

(4) Lancé de grappin, c'est-à-dire exploration horizontale avec progression de x jusqu'à trouver la courbe c'est-à-dire un signe différent de ref_signe. Dès que c'est fait « meme_signe_que » renvoie NON, la boucle s'arrête, x est à la bonne position.

(5)(6) Sauvegarde de la position de départ et initialisation de dern selon le principe que nous avons rencontré avec la fonction « contour »

Toutes les lignes suivantes sont identiques à la fonction « contour », la seule différence est à la ligne (11) l'appel de la fonction « meme_signe_que » à la place de la fonction « endehors » .

Chapitre 13

Quelques techniques du mouvement

Nous poursuivons l'exploration algorithmique initiée au chapitre précédent. Elle comprenait la réalisation de formes géométriques simples. Egalement, à travers une étude de restitution de contours et de courbes, une évocation de la forme telle qu'elle peut donner lieu à l'étude de théories morphologiques (chaos, fractales, catastrophes, attracteurs étranges). A ces deux premières étapes l'objectif est maintenant d'ajouter le mouvement à des formes simples. Mais qu'est-ce qu'un scénario du mouvement dans l'espace robographique et à quel regard invite-t-il ?

Des mouvements de base et des mouvements génératifs de formes

De façon basique, nous distinguons deux sortes de mouvements. Le mouvement des couleurs ou de la lumière et le mouvement d'une image ou d'une forme qui se déplace. Mais aussi, il y a le mouvement intrinsèque à la forme elle-même et à sa construction, lorsque la notion d'histoire intervient dans l'expression d'une forme en développement. L'objectif de ce chapitre-ci est de poser quelques techniques du mouvement, des primitives robographiques du mouvement. La façon dont ces primitives entrent dans des mises en scène et des représentations fait l'objet du chapitre suivant.

Introduction aux bases du mouvement

La toile du peintre numérique est une surface de mémoire, elle se réduit à des informations de couleur et de positions. Le déplacement d'un point à l'écran est le déplacement d'une couleur de pixel à pixel, le déplacement d'une couleur de points en points parmi d'autres points de couleurs.

Admettons que nous ayons un fond noir et sur ce fond noir un pixel de coordonnées (x, y) avec une couleur c. Pour donner l'illusion d'un déplacement du pixel de couleur, la technique la plus simple se résume à :

1) Mettre le pixel de la position (x, y) en noir.
2) Modifier la position, par exemple (x+1, y).
3) Mettre le pixel de la position (x+1, y) dans la couleur c.
4) Mettre en boucle le processus.

En d'autres termes, soit une couleur donnée à une position donnée, il y a deux actions possibles : changer la couleur, changer la position. En faisant les deux on obtient que la forme suive la couleur, comme dans l'exemple du pixel. Il y a un déplacement de la forme sur la grille des pixels par un jeu d'effacement puis de réaffichage de la forme à de nouvelles coordonnées. Par exemple un carré plein de couleur rouge se déplace sur un fond noir ou coloré de l'écran et il peut également se déplacer sur une image.

Sans déplacement, la forme qui reste immobile peut néanmoins changer de couleur. L'animation des couleurs peut même donner l'illusion d'un changement de forme avec la couleur. C'est aussi un moyen de faire communiquer des images par des jeux d'échanges de couleurs, notamment dans les processus de fondu au noir et d'apparition d'images fixes.

En premier lieu nous entrons dans l'animation de couleurs avec deux exemples : la rotation de palette et l'interpolation entre deux palettes. Ensuite nous abordons le déplacement de formes géométriques. Après exposition de la technique de base, l'utilisation de « buffers » intermédiaires de stockage des images est abordée. Et du même coup la présentation d'un ensemble de fonctions d'affichage propres à l'environnement

Allegro. Le déplacement sur des images est également mentionné. A partir de ces techniques, une méthode empruntée au dessin animé peut être utilisée ; un affichage d'images successives compose une petite séquence d'animation et elle peut être extrapolée.

1. Animation de couleurs

1.1 Rotation de palette

La rotation de palette est très utilisée dans le logiciel Mikado présenté dans le chapitre Sept. Elle apparaît dans beaucoup de situations différentes. L'idée est ici de faire circuler la couleur sur une topographie conique résultant d'un empilement fixe de disques concentriques de rayons décroissants et chacun d'une couleur différente. Pour conférer à l'effet une certaine harmonie nous utilisons une palette cyclique de dégradé, celle que nous avons présentée (figures 10.2 et 10.3, dégradé constant de couleur constante au chapitre Dix). Le principe est simple, pour faire circuler la couleur il suffit de modifier les positions des couleurs dans la palette de référence.

C'est l'objet de la fonction « test_rotation_palette » (figure 13.1) :

```
(1)  PALETTE pal;

(2)  void test_rotation_palette( )
     {
(3)  int k;
     RGB tmp;

(4)       degrade_constant ( ) ;

(5)       for (k = 255; k > 0;  k - -)
               circlefill(screen, SCREEN_W/2, SCREEN_H/2, k , k );

(6)       while ( ! key[KEY_ESC]) {
(7)            tmp = pal[ 255 ];
               for ( k = 255; k >1; k- - )
                    pal[ k ] = pal[ k -1 ];

               pal[ 1 ] = tmp;
(8)            set_palette( pal );
          }
     }
```

Fig. 13.1 : Rotation des couleurs dans une palette.

(1) Déclaration d'une palette en variable globale pour le programme

(2) Identification de la fonction « test_rotation palette ». Elle ne prend aucun argument et ne renvoie rien.

(3) Déclaration des variables locales à la fonction, un entier et une structure RGB.

(4) Appel de la fonction « degrade_constant » définie au chapitre Dix et qui initialise un dégradé de couleur cyclique (La dernière couleur retombe sur la première par dégradé).

(5) Le test de rotation de palette est effectué sur un ensemble de 255 disques de couleur, un par couleur, ayant même centre et encastrés les uns dans les autres du plus grand au plus petit. C'est l'objet de la boucle for qui passe en revue les valeurs de 255 jusqu'à 1. Pour chacune d'entre elles un cercle de couleur est tracé par un appel à la fonction « circlefill » (primitive de dessin de l'environnement Allegro). Cette fonction prend cinq arguments : une image de référence, une position (x, y), un rayon et une couleur. Pour chaque appel les valeurs passées sont : la bitmap écran, screen, vue au chapitre Onze et définie pour le tracé à l'écran, les tailles horizontale et verticale de l'écran, divisée par deux ce qui donne le centre de l'écran, le rayon défini par la variable k qui décroît ainsi que la couleur affecté à chaque cercle. C'est-à-dire qu'il y a un empilement de disques, les plus petits étant au premier plan, et cet empilement coïncide avec l'ordre des couleurs dans la palette.

(6) Une fois l'empilement de cercles établi, il s'agit de faire varier les couleurs et laisser les cercles immobiles. C'est l'objet des instructions qui suivent (7). Mais voyons d'abord le test d'arrêt de la boucle while. C'est un test sur l'utilisation du clavier de l'ordinateur. L'environnement prévoit plusieurs possibilités notamment un tableau « key » dans lequel sont stockés les états de toutes les touches du clavier, avec 0 pour non enfoncée et 1 pour enfoncée. Chaque touche du clavier est définie par une macro qui donne son indice dans le tableau. Ainsi KEY_ESC correspond à la touche « échap » du clavier et l'indice 59 dans le tableau « key ». L'expression « ! key[KEY_ESC] » renvoie vrai et la boucle continue, si, dans le tableau « key » à la position KEY_ESC il y a une valeur de 0, c'est-à-dire que la touche n'est pas enfoncée. Dès que la touche est enfoncée cette valeur passe à 1, le test devient faux et la boucle prend fin[1].

(7) Le mouvement de couleur est opéré sur la palette avec un déplacement des couleurs qu'elle contient par un mouvement tournant (figure 13.2).

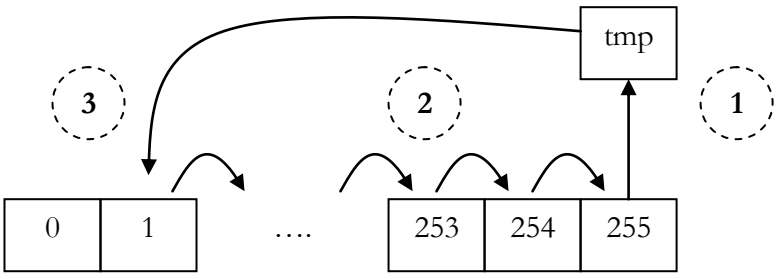


Fig. 13.2 : Faire tourner les couleurs d'une palette

1) Les valeurs de la dernière couleur de la palette, une structure RGB sont sauvées dans une structure RGB temporaire.

[1] Les mécanismes de capture les actions au clavier et d'acheminement ensuite des valeurs appartiennent à l'environnement de programmation et nous ne les abordons pas.

2) Ensuite chaque position de la palette prend pour valeurs les valeurs de la position précédente. C'est l'expression « pal[k] = pal[k -1]; » appliquée à toutes les positions de la palette de 255 à 1. La position 0 est conservée avec la même couleur, du noir pour le fond.
3) Après fin de la boucle for, les valeurs de la dernière couleur conservées dans la variable « tmp » sont copiées à la position 1

(8) A ce moment la palette a changé mais l'image à l'écran est toujours la même. Pour que la modification de la palette soit effective l'environnement fournit la fonction « set_palette » qui est nécessaire à l'activation d'une palette. La palette passée en argument devient la palette visible à l'écran.

Du point de vue de l'image bitmap, en tant que surface mémoire, la forme initiale et sa couleur n'ont pas changé. Toujours à la même place, si mon cercle est de la couleur n° 3 de la palette, il reste toujours de la couleur n°3. Ce sont les valeurs de cette couleur qui sont changées. Elles prennent maintenant celles de la couleur qui était à la position n°2.

1.2 Interpolation de palettes

Les variations de la couleur sont développées dans l'environnement de programmation pour reproduire des effets bien connus de l'audiovisuel : fondu en entrée, fondu en sortie, fondu entre deux images. En fait ce sont toutes les questions de luminosité et d'éclairage du point de vue des couleurs seules et sans rapport à l'espace, qui sont ouvertes, avec des évolutions de couleurs et des effets de transparence dans les modes palette aussi bien que vraies couleurs[1].

La fonction « fade_interpolate » est une fonction de base très utile dans le mode palette pour tout ce qui a rapport au passage d'une palette à une autre ou même d'une partie de palette vers une autre. C'est une fonction de transition des couleurs d'une palette dans les couleurs d'une autre avec le résultat stocké dans une troisième palette, la palette intermédiaire (figure 13.3) :

(1) Identification de la fonction « fade_interpolation ». Elle prend six arguments. D'abord il y a trois palettes. La première « source » est la palette de départ, « const » indique qu'elle peut peut pas être modifiée, que sa valeur est constante. La seconde « dest » est la palette de destination, c'est également une constante qui ne pourra subir aucune modification. La troisième « output » est la palette résultante, intermédiaire entre les valeurs de couleurs de source et dest.
L'entier « pos » donne somme toute la position entre source et dest graduée de 0 à 64. Avec 0 passé en argument la palette résultat output sera identique à la palette source. Avec 64 passé, elle sera identique à la palette dest. Avec 32 elle sera tout juste entre les deux.

Les deux entiers « from » et « to » permettent d'effectuer la transition uniquement sur des segments de palette. Si on passe 0 pour from et 255 pour to c'est toute la palette qui est examinée.

[1] L'environnement Allegro dispose de vingt deux fonctions pour le traitement des palettes et de vingt cinq fonctions pour ce qui concerne les effets de lumière et de transparence.

2) Déclaration d'une variable locale à la fonction utilisée dans la boucle for

(3)(4) Boucle for qui va des valeurs de from à to et passe en revue les positions intermédiaires. Pour chaque position le calcul est une moyenne pondérée par la variable pos entre les valeurs de couleur de la palette source et celle de la palette destination. Plus la valeur de pos est petite, plus la valeur résultante est proche de celle de la palette source. Inversement, plus la valeur de pos est grande, plus la valeur résultante est poche de celle de la palette de destination. Le résultat est stocké dans la palette de sortie output.

```
(1)  void fade_interpolate(  const PALETTE source,
                             const PALETTE dest,
                             PALETTE output,
                             int pos, int from, int to )
     {
(2)  int c;

(3)       for (c=from; c<=to; c++) {
(4)            output[c].r = ((int)source[c].r * (63-pos) + (int)dest[c].r * pos) / 64;
               output[c].g = ((int)source[c].g * (63-pos) + (int)dest[c].g * pos) / 64;
               output[c].b = ((int)source[c].b * (63-pos) + (int)dest[c].b * pos) / 64;
          }
     }
```

Fig. 13.3 : Interpolation de palette, fonction de l'environnement Allegro pour la programmation de jeux.

Avec cet algorithme on obtient à chaque appel de la fonction une palette qui est entre la palette source et la palette destination. Pour obtenir la totalité du fondu, c'est-à-dire chaque palette intermédiaire qui conduit de la palette source jusqu'à la palette destination, il suffit d'ajouter une variable pour le nombre d'étapes souhaitées de 0 à 64 et l'on peut écrire quelque chose comme (figure 13.4) :

```
(1)  void fondu_palette ( PALETTE source, PALETTE dest, int vitesse )
     {
(2)  PALETTE tmp;
     int pos ;

(3)       set_palette(source) ;
(4)       for ( pos  = 0; pos < 64;  pos += vitesse ) {
               fade_interpolate(source, dest, tmp, pos, 0, 255 );
               set_palette ( tmp );
          }
(5)       set_palette(dest);
     }
```

Fig. 13.4 : Obtenir toutes les étapes de palettes intermédiaires entre une palette de départ et une palette d'arrivée

(1) La fonction « fondu_palette » prend trois arguments. La palette de départ « source », la palette d'arrivée « dest » et un entier qui sert à indiquer la vitesse de la transition. La valeur de cet entier doit être comprise entre 1 et 64.

(2) Variables locales à la fonction « fondu_palette ». Une palette temporaire « tmp » qui va servir au calcul des palette intermédiaires et à leur affichage. Et un entier pour le calcul des positions de palettes intermédiaires.

(3) Actualisation de la palette source avec appel de la fonction « set_palette »

(4) Boucle for qui va de 0 à 63 selon un pas donné par la variable « vitesse ». Le bloc d'instruction est un appel à la fonction précédente « fade_interpolate » et elle prend en arguments les palettes « source », « dest » et « tmp ». La position entre les deux palettes est donnée par la variable « pos » et la fourchette est bloquée entre 0 et 255 (c'est-à-dire toute l'étendue de la palette). Ensuite appel de la fonction « set_palette » qui affiche les nouvelles couleurs de la nouvelle palette.

(5) au cas ou « pos + vitesse » ne permette pas d'arriver à 64 mais s'arrête avant, par exemple si vitesse = 5 la dernière palette intermédiaire est à 60, aussi le résultat final d'arrivée est-il forcé aux couleurs de la palette destination à la sortie de la boucle.

Si par exemple la palette source est une palette de noir où toutes les composantes de rouge de vert et de bleu sont à 0, on obtiendra un fondu en entrée du noir vers une plus grande luminosité de l'image jusqu'à ses couleurs réelles dans la palette de destination. Ou inversement pour un fondu au noir.

2. Déplacement de formes géométriques

2.1 Principe général

Imaginons un point de couleur qui se déplace à l'écran sur un fond noir. Il part du centre. Avec une vitesse définie aléatoirement, il va dans une direction. Lorsqu'il rencontre un bord de l'écran il rebondit et repart en sens opposé. Cela peut se coder de la façon suivante (figure 13.5) :

(1) La fonction « deplace_pixel » ne prend aucun argument et ne renvoie rien.

(2) Déclarations des variables locales. Deux entiers pour la position en x et y. Elle est initialisée au centre de l 'écran. Les deux autres entiers « pasx » et « pasy » désignent les pas d'avancement du pixel en x et y. Ils sont initialisés au hasard entre -5 et 5.

(3) Boucle de l'action. Le test d'arrêt est du même type que celui de la figure 13.1 à la ligne (6). Tant que la touche « échap » n'a pas été pressée, les instructions de la boucle sont effectuées.

(4) Effacement du pixel à sa position courante, c'est-à-dire passage en noir comme le fond, de la couleur du pixel (nous avons pris 0 comme position de la palette pour la couleur noire).

(5) Modification de la position en x et y.

```
(1)  void deplace_pixel()
     {
(2)  int x = SCREEN_W/2;
     int y = SCREEN_H /2;
     int pasx  = (rand()%11)-5;
     int pasy  = (rand()%11)-5;

(3)        while( ! key[KEY_ESC] ) {

(4)              putpixel(screen, x, y, 0),
(5)              x+=pasx ;
                 y+=pasy ;

(6)              if( x >= SCREEN_W )
                       pasx = -ABS(pasx);
                 if( x < 0 )
                       pasx = ABS(pasx);
                 if( y >=SCREEN_H )
                       pasy = -ABS(pasy);
                 if( y < 0 )
                       pasy = ABS(pasy);

(7)              putpixel(screen, x,y, 192);

(8)              rest(10);

(9)              if ( key[KEY_ENTER] ){
                       pasx = (rand()%11) - 5 ;
                       pasy = (rand()%11) - 5 ;

                 }
           }
     }
```

Fig. 13.5 : Déplacer un point, base du déplacement de formes géométriques en deux dimensions

(6) Contrôles pour les bords de l'écran. Si la position en x dépasse la taille horizontale de l'écran qui est donnée dans l'environnement par la variable « SCREEN_W », le pas est forcé à devenir négatif. A l'inverse si la position horizontale est inférieure à 0, le bord gauche de l'écran, le pas est forcé au positif. De même pour la position verticale. Si y dépasse la taille verticale de l'écran, « SCREEN_H », le pas est forcé à une valeur négative et s'il est inférieur à 0, le bord haut, il est forcé au positif.

(7) Une fois le changement de position effectué, le point est affiché aux nouvelles coordonnées, dans une couleur autre que celle du fond noir.

(8) Nous avons dù ajouter un temps d'arrêt pour voir le pixel qui sinon ne reste pas suffisamment longtemps affiché pour être visible. La fonction « rest » est fournie par l'environnement pour la famille des fonctions attachées à la gestion du temps. Elle permet

d'introduire un temps de retardement en millième de seconde dans un processus.

(9) C'est une possibilité accessoire de changer les pas d'avancement en x et y en pressant la touche « entrée ». Si cette touche est pressée, deux nouvelles valeurs aléatoires sont affectées aux variables pasx et pasy.

Pour déplacer d'autres formes que le point, il suffit de changer la fonction « putpixel » et de la remplacer par celle de la forme que l'on souhaite, rectangle, cercle, triangle ou droite. Avec cette méthode, l 'animation est saccadée ; plus la forme est grande plus on voit l'effacement et le nouvel affichage. Dans certains cas cela peut s'avérer intéressant sur le plan esthétique, surtout avec de petites formes, par exemple les nénuphs représentés par un tracé de cercle fonctionnent assez bien avec ce procédé. Souvent d'ailleurs des systèmes de vie artificielle, dans la mesure où la maladresse même du mouvement des formes confère à ces formes un petit quelque chose d'amusant presque réaliste : ça fait effectivement de petites bestioles qui gigotent. Il y a cependant une autre méthode qui donne une animation plus fluide, plus rapide également, c'est le principe du double buffer.

2.2 Double buffer, fonctions d'affichage

Le principe du double buffer est simplement d'agencer l'ensemble des formes que l'on veut mouvoir dans une image bitmap intermédiaire puis d'afficher d'un seul bloc cette image à l'écran en écrasant ce qui se trouvait précédemment affiché à l'écran. Ce qui donne les étapes suivantes :
1) Effacer l'image intermédiaire.
2) Pour n objets, modifier les positions des objets en fonction de leurs vitesses et trajectoires respectives etc.
3) Afficher les formes à leurs nouvelles positions dans la bitmap intermédiaire.
4) Plaquer cette image à l'écran.

L'image intermédiaire fonctionne un peu comme une table de montage sur laquelle sont assemblés tous les éléments constitutifs du résultat souhaité. L'avantage est un gain de rapidité important. La majeure partie de l'affichage se fait en mémoire centrale sans passer par l'écran et la mémoire vidéo beaucoup plus lente. De plus il n'y a pas lieu d'effacer chaque forme. Tous les pixels de la bitmap intermédiaire peuvent être mis à zéro en un seul appel d'une fonction d'effacement fournie par l'environnement.
Du point de vue esthétique cette technique donne une animation fluide. Elle fonctionne par plans d'ensemble successifs, sans visualisation de la construction de chaque élément séparé. C'est comme dans un dessin animé traditionnel mais avec la différence importante que le développement de l'animation est calculé en temps réel.

Le déplacement de formes géométriques peut être illustré par un processus d'animation que nous avons utilisé pour la réalisation d'un écran de veille baptisé « vue » et réalisé pour le compte d'une association. Tout d'abord la figure 13.6 définit les structures de données utilisées. Ensuite la figure 13.7 donne la fonction « main » ainsi que la boucle du programme. Pour finir les trois fonctions d'initialisation, de mouvement et d'assemblage, sont détaillées avec les figures 13.8, 13.9 et 13.10.

Structure de données mise en place

La figure 13.6 donne tous les variables essentielles à partir desquelles le programme est construit :

```
(1) #define ECRAN_X     800
    #define ECRAN_Y     600

(2) #define irand(n) (rand()%(n))
(3) enum{rectang, triang,cercle};

(4) typedef struct FORME{
        int type;
        int x,y;
        int tx,ty;
        int dx,dy;
        int couleur;
    }FORME;

(5) #define NB_FORMES       1000
    FORME* TAB[NB_FORMES];

(6) BITMAP*page ;
    PALETTE pal;
```

Fig. 13.6 : Structure de données du programme « vue »

(1) Deux macros pour définir taille et résolution de l'écran, « ECRAN_X » pour l'horizontale et « ECRAN_Y » pour la verticale. Cette définition vient en amont de la définition des globales « SCREEN_W » et « SCREEN_H » données par l'environnement en retour de l'appel à la fonction « set_gfx_mode » évoquée au chapitre Onze à propos de l'accès à l'écran de l'ordinateur. L'avantage sera de pouvoir modifier ces valeurs facilement si besoin est pour une nouvelle compilation. En revanche si la taille et la résolution de l'écran devaient être modifiées pendant l'exécution du programme, il serait nécessaire de s'appuyer sur les globales anglophones définies par « set_gfx_mode ».

(2) Macro fonction pour simplifier l'écriture de rand() modulo un nombre.

(3) Il s'agit d'une liste de constantes numérotées à partir de 0 et incrémentées de un à chaque nouvelle constante.

Les constantes énumérées

C'est une alternative à une liste de #define qui, outre le fait de permettre des regroupements sémantiques, présente l'avantage d'une numérotation automatique. Le premier nom d'une « enum » vaut 0, le second 1 etc. Eventuellement une valeur peut être précisée explicitement. Dans ce cas, celles qui la suivent sont toujours incrémentées de 1 en 1 mais à partir de la valeur spécifiée [KERNIGHAN, 1995, p.39] [BRAQUELAIRE, 1998, p.54].
Ce choix a pour objectif de regrouper ici toutes les alternatives de formes qui seront utilisées (des rectangles, des triangles et des cercles).

(4) Définition d'une structure « FORME » qui comprend les informations nécessaires à une forme.

- L'entier « type » sert à déterminer si la forme est un rectangle, un triangle ou un cercle. Il sera initialisé avec une valeur de 0, 1 ou 2.
- Les entiers « x » et « y » correspondent à la position de la forme dans l'écran.
- « tx » et « ty » donnent la taille de la forme.
- « dx » et « dy » sont là pour le déplacement de la forme.
- « couleur » conserve la couleur donnée à la forme.

(5) Macro pour le nombre de formes souhaité et regroupement de l'ensemble de formes dans un tableau de pointeurs sur structure FORME

(6) Deux variables globales : « page » pour la bitmap intermédiaire (le fameux « buffer ») et « pal » pour une palette de couleurs.

Programme

A partir de cette structure nous pouvons construire une action : un ensemble de formes qui se déplacent sur l'écran. La figure 13.7 donne la fonction « main » complète du programme :

La fonction « main »

Il s'agit d'une fonction particulière qui marque le début d'un programme. Tout programme doit comporter quelque part son « main » et ce nom est invariable du point de vue de la norme ANSI qui régule le langage C (il peut y avoir cependant des différences de main en fonction du système d'exploitation).

(1) et (16) Dans l'environnement Allegro, ces deux lignes constituent l'appel de la fonction main de tout programme. Cet environnement est multi-plateforme. Il a fallu définir une écriture qui reste identique sous plusieurs systèmes d'exploitation, afin que n'importe quel programme puisse être compilé sans changement dans le code sous plusieurs systèmes et selon différents compilateurs. C'est ce qui donne cette écriture originale du main en deux parties ; une macro « END_OF_MAIN » et la fonction main proprement dite qui répond à la définition de la macro.

(2) Variable outil utilisée pour des boucles un peu plus bas.

(3) Une série de quatre fonctions qui ne sont pas liées directement à notre algorithme mais qui sont nécessaires au fonctionnement des programmes et concerne l'environnement : initialisation en relation avec Allegro, fonction pour initialiser l'utilisation du clavier de l'ordinateur, appel de « set_gfx_mode » pour initialisation écran et résolution écran. En cas d'erreur, c'est-à-dire si la fonction retourne une valeur différente de 0, le bloc d'instruction consiste en un message d'erreur et la sortie de programme ensuite avec la valeur de retour 1 qui indique au système une erreur du programme. Enfin une fonction, « srand » de la librairie standard <time.h> . Cette fonction initialise avec la valeur qu'elle prend en argument le générateur de nombres pseudo-aléatoires, la fonction « rand ». Ici elle prend en argument la valeur du temps telle qu'elle est codée par le système d'exploitation et retournée par la fonction « time » .

(4) Il s'agit de fabriquer une palette avec un appel à la fonction « degrade_pas_variable » vue au chapitre Dix. Elle prend la palette « pal » supposée déclarée en globale comme argument.

```
(1)  int main( )
     {
(2)  int i ;

(3)       allegro_init();
          install_keyboard();
          if (set_gfx_mode(GFX_AUTODETECT, ECRAN_X, ECRAN_Y, 0, 0) != 0) {
               allegro_message("Erreur pour le mode graphique\n%s\n", allegro_error);
               return 1;
          }
          srand(time(NULL));

(4)       degrade_pas_variable(pal);

(5)       page = create_bitmap(ECRAN_X, ECRAN_Y);

(6)       for ( i = 0; i <NB_FORMES; i++){
               TAB[ i ] = (FORME*)malloc(sizeof(FORME));
               initialisation_formes(TAB[i]);
          }

(7)       do{
(8)            clear(page);
(9)            for ( i = 0; i <NB_FORMES; i++){
(10)                mouvement_formes(TAB[ i ]);
(11)                assemble_formes(TAB[ i ], page);
               }
(12)           blit(page, screen, 0,0,0,0, page->w, page->h);

(13)      }while( !key[KEY_ESC] );

(14)      destroy_bitmap(page);
          for (i=0; i< NB_FORMES; i++){
               p=TAB[ i ];
               free(p);
          }
(15)      return 0;
     }
(16) END_OF_MAIN( );
```

Fig. 13.7 : Base des mouvements dans le programme « vue »

(5) Création de la bitmap « page » déclarée en globale avec un appel à la fonction « create_bitmap » vue au chapitre 11. C'est une bitmap qui a la taille de l'écran.

(6) Boucle for pour l'initialisation des formes, c'est-à-dire pour les « NB_FORMES » pointeurs de structures « FORME* » stockés dans le tableau « TAB ». La première

instruction consiste à allouer la mémoire nécessaire à chaque structure avec appel à la fonction « malloc » que nous avons rencontrée dans le chapitre Onze.

La seconde instruction est un appel à la fonction d'initialisation de chaque structure FORME dont le détail est donné plus bas (figure 13.8).

(7) et (13) La boucle a la forme d'un do-while avec pour test d'arrêt l'attente que la touche « échap » soit pressée.

(8) La première instruction du corps de la boucle est l'utilisation de la fonction « clear » qui pour initialise à 0 tous les pixels de la bitmap « page » passée en argument.

(9)(10)(11) Boucle for qui passe en revue toutes les structures FORME afin de modifier premièrement en (10) la position de chaque forme avec un appel à la fonction « mouvement_formes » (figure 13.9), suivi d'un appel à la fonction « assemble_formes » (figure 13.10) qui dessine les formes à leurs nouvelles positions. Ces deux fonctions sont détaillées plus bas, après la fonction d'initialisation des formes.

(12) Après la boucle d'appel des fonctions « mouvement_formes » et « assemble_formes » pour chaque pointeur FORME* du tableau TAB, le programme continue et termine le processus de l'animation avec le plaquage à l'écran de l'image intermédiaire qui vient d'être achevée. C'est le rôle de l'appel de la fonction « blit » (BLoc, Image, Transfer) fournie par l'environnement. Cette fonction prend huit arguments :

```
void blit(      BITMAP *source, BITMAP *dest,
                int source_x, int source_y,
                int dest_x, int dest_y,
                int longueur, int hauteur               );
```

La bitmap source est l'image à copier, et la bitmap dest est l'image sur laquelle l'image source est copiée. Les arguments source_x et source_y associés aux arguments longueur et hauteur définissent quelle partie de l'image source est copiée et les arguments dest_x et dest_y à quelle position dans l'image de destination la partie de l'image source est copiée.

Ainsi le premier argument passé à blit est l'image intermédiaire « page », la destination est l'écran et la partie est l'image entière du coin haut gauche (0, 0) au coin bas droite (page->w, page->h) selon un accès aux champs de la structure BITMAP qui contient les informations de longueur et hauteur de l'image. La destination est le calage en (0, 0) à l'écran.

(14)(15) Au sortir de la boucle, après que la touche échap ait été pressée, il y a quelques instructions afin de libérer la mémoire allouée pendant le fonctionnement du programme, et en (15) la fonction « main » retourne 0 afin de signaler la fin normale du programme au système d'exploitation.

Fonction « initialisation_formes »

La fonction « initialisation_formes » (figure 13.8 ci-dessous) affecte une valeur aléatoire, dans une fourchette donnée, à chacun des champs de la structure dont l'adresse est passée en argument.

Le type de forme est limité à trois types possibles (rectangle, triangle et cercle) et la position par la taille de l'écran. La taille de la forme, de façon arbitraire a 60 pixels maximum et 5 minimum. Le déplacement est entre -5 et 5, 0 compris, et la couleur est choisie au hasard dans la palette.

```
void initialisation_formes(FORME*f)
{
        f->type = rand()%3;
        f->x = rand()%ECRAN_X;
        f->y = rand()%ECRAN_Y;
        f->tx = (rand()%56)+5;
        f->ty = (rand()%56)+5;
        f->dx = (rand()%11)-5;
        f->dy = (rand()%11)-5;
        f->couleur = rand()%256;
}
```

Fig. 13.8 : Initialiser les structures FORME

Fonction « mouvement_formes »

Comme le montre la figure 13.9, cette fonction prend en argument l'adresse de chaque structure concernée, c'est-à-dire un pointeur de type FORME*. En (1) elle ajoute à la position courante de la forme les valeurs de ses variables de déplacement : dx et dy. Ensuite en (2) il y a vérification que la position est toujours à l'intérieur de l'écran. Si tel n'est pas le cas, et que la valeur du déplacement va dans le sens de l'éloignement de l'écran, cette valeur est inversée.

```
    void mouvement_formes( FORME* f )
    {
(1)     f->x += f->dx;
        f->y += f->dy;

(2)     if ( ((f->x <= 0) && (f->dx < 0)) | | ((f->x >= ECRAN_X) && (f->dx > 0)))
            f->dx = -f->dx;

        if (((f->y <= 0) && (f->dy < 0)) | | ((f->y >= ECRAN_Y) && (f->dy > 0)))
            f->dy = -f->dy;
    }
```

Fig. 13.9 : Modifier les coordonnées des formes pour les déplacer

Fonction « assemble_formes »

Comme les deux fonctions précédentes (figure 13.10 ci-dessous) elle prend en argument un pointeur sur une structure FORME, mais également un pointeur sur une BITMAP : chaque forme va être affichée selon ses paramètres de type, de position et de taille dans la bitmap de référence passée en argument. Il y a plusieurs types possibles et chacun correspond à un appel de fonction spécifique. Il faut donc faire des tests sur le type de chaque forme, ce qui se traduit par des expressions comme : si type égale 0 alors tracer un rectangle, si type égale 1 tracer un traingle, si type égale 2 tracer un cercle. Cela peut être

est remplacé par une figure équivalente, un interrupteur ou aiguillage nommé « switch » dans le langage C.

Instruction switch

Le switch permet un choix entre plusieurs groupes séparés d'instructions. Il est de la forme :

```
switch ( expression ) {
        case expression-constante : instructions ;
        case expression-constante : instructions ;
        default : instructions
}
```

Chaque cas est identifié par une valeur numérique entière, constante qui peut prendre l'aspect d'expressions constantes telles #define, enum etc. Si la valeur numérique de l'argument *expression* du switch correspond à la valeur numérique d'une des *expressions-constantes* des différents cas, ce sont les instructions de ce cas qui sont exécutées. Les suivantes également à moins d'isoler le bloc avec l'instruction « break ; » qui provoque la sortie du switch. Si l'argument *expression* du switch ne correspond à aucun cas, et si « default » est définie, ce sont les instructions du cas par défaut qui sont exécutées, sinon aucune [KERNIGHAN, 1995, p. 59] [BRAQUELAIRE, 1998, p.141].

```
void assemble_formes(FORME*f, BITMAP* bmp)
{
    switch(f->type){
        case rectang : rectfill( bmp,   f->x, f->y,
                                        f->x+f->tx, f->y+f->ty,
                                        f->couleur);
            break;
        case triang :   triangle( bmp, f->x, f->y,
                                        f->x+f->tx, f->y,
                                        f->x+f->tx/2 ,f->y-f->ty,
                                        f->couleur);
            break;

        case cercle :   circlefill( bmp, f->x, f->y, f->ty, f->couleur);
            break;
    }
}
```

Fig. 13.10 : Dessiner la forme dans la bitmap intermédiaire, le « buffer »

Dans la fonction « assemble_formes » trois cas sont définis qui correspondent aux constantes de la liste énumérée. Il n'y a pas de « case default : ». Le test du switch se fait sur la valeur affectée au champ « type » de chaque forme. Si type est égal à 0 c'est le cas « rectang » qui est sélectionné. Le bloc d'instruction consiste à tracer, à l'adresse de la bitmap passée en argument, un rectangle selon les autres paramètres de la structure forme dont l'adresse est bien sûr passée en argument. Si type est égal à 1 c'est la cas « triang » avec un tracé de triangle et si type égale 2 c'est le cas cercle avec l'instruction de tracer un cercle.

Pour le rectangle, la position (x, y) donne le point haut gauche et la position plus la taille donne le point bas droite (x + tx, y + ty).
Pour le triangle, le premier point est à gauche à la position (x, y). Le second est sur la même ligne horizontale avec (x + tx, y). Le troisième est au milieu du point de vue horizontal (x + tx / 2) et au-dessus du point de vue vertical avec (y - ty).
Pour le cercle le rayon est fourni par la taille en y (ty)

Sur la base de ce mouvement d'ensemble des formes nous avons développé un écran de veille pour une association qui souhaitait mettre en valeur un patrimoine de photos numériques prises lors de leurs activités.

2.3 Déplacement sur des images, modes d'affichages

Les formes du programme précédent sont des formes de couleur animées et projetées à l'écran. Mais il y a moyen de modifier le mode d'affichage afin par exemple de faire courir la forme sur une image et de voir l'image à travers elle comme à travers une serrure. L'environnement propose six modes d'affichage, chacun désigné par une macro constante :

DRAW_MODE_SOLID,	DRAW_MODE_MASKED_PATTERN,
DRAW_MODE_COPY_PATTERN,	DRAW_MODE_TRANS,
DRAW_MODE_SOLID_PATTERN,	DRAW MODE XOR

Le mode graphique est fixé dans le programme par un appel à la fonction :

```
void drawing_mode(int mode, BITMAP *pattern, int x_anchor, int y_anchor);
```

Chacun d'entre eux détermine un mode de remplacement des pixels lorsqu'ils sont modifiés par une fonction de tracé de formes géométriques (pixels, lignes, rectangles, triangles, cercles, polygones et même une fonction de remplissage « floodfill »). Nous n'allons pas les présenter tous mais seulement les deux dont nous avons besoin pour un tracé normal et un déplacement de formes géométriques sur des images.

DRAW_MODE_SOLID est le mode par défaut de coloration normale. La couleur des pixels concernés par un tracé est remplacée par la couleur passée en argument à la fonction de tracé.

DRAW_MODE_COPY_PATTERN est le mode dans lequel les pixels sont copiés à partir d'une image de référence, comme un plan de référence. Si par exemple un rectangle est tracé à l'écran, les pixels qui le constituent prennent la couleur des pixels aux positions correspondantes de l'image de référence. Dans ce cas la couleur passée en argument à la fonction de tracé est ignorée. Ce mode présente l'avantage d'être le plus rapide.

A titre indicatif, toujours dans le cadre d'une utilisation d'image d'arrière plan (pattern) il y a le DRAW_MODE_SOLID_PATTERN et le DRAW_MODE_MASKED_PATTERN.

Le premier consiste à comparer la couleur des pixels de l'image de référence avec la couleur utilisée comme masque (ce qui est de la couleur « masque » n'est pas affiché). Cette dernière est 0 en 256 couleurs et rose en vraie couleur (maximum de rouge et de bleu et

zéro vert). Si la couleur trouvée n'est pas celle de masque alors le pixel prend la couleur passée en argument à la fonction de tracé. Dans le cas contraire, il prend la couleur 0, noire. L'image de référence agit comme un filtre. Les formes ne sont visibles que sur les parties de couleur différentes de la couleur masque.

Le second, DRAW_MODE_MASKED_PATTERN est équivalent au précédent sauf que les pixels trouvés de la couleurs du masque dans l'image « pattern » seront laissés tel quel et non passés à zéro.

Ainsi dans le programme de la figure 13.7, pour faire courir les formes sur des images, c'est essentiellement à la ligne (4) la fonction de création de palette qui est remplacée par un ensemble d'instructions nouvelles ; les lignes suivantes sont introduites (figure 13.11) :

```
(1) BITMAP* image;

(2)  image = load_bitmap(".\\mon_image.bmp",pal);
     if(! image ){
          allegro_message("Erreur creation bitmap\n%s\n", allegro_error);
          return 1;
     }
(3)  set_palette(pal);
(4)  drawing_mode ( DRAW_MODE_COPY_PATTERN, image, 0, 0);
```

Fig. 13.11 : Charger une image dans le programme et changer le mode d'affichage

(1) Il est nécessaire d'ajouter au programme une bitmap pour récupérer un fichier d'image sur le disque dur. C'est la déclaration du pointeur BITMAP* à mettre soit dans la fonction « main » soit en globale.

(2) La récupération de l'image au format bitmap se fait via l'utilisation de la fonction « load_bitmap » fournie par l'environnement. Elle prend le chemin d'accès au fichier en argument et une palette dans laquelle sera stockée, en 8 bits, la palette des couleurs de l'image. Elle renvoie un pointeur BITMAP* qui contient l'adresse de l'image copiée en disponible en mémoire centrale (RAM).

(3) La palette de l'image est rendue active avec la fonction « set_palette ».

(4) Appel de la fonction « drawing_mode » qui 1) change le mode graphique et 2) prend pour image pattern de référence l'image juste chargée « image ».

Maintenant les formes courent sur l'image. Dans la réalisation du programme écran de veille « Vue » l'idée était que chaque forme puisse avoir une image pattern de référence différente, piochée au hasard dans un ensemble d'images à proximité du programme sur le disque dur. Le résultat donne un brassage des images assez surprenant. Sa qualité dépend beaucoup de la cohérence graphique de l'ensemble des images sur lequel il s'appuie.

3. Bouger à partir d'images

Nous avons abordé le mouvement de formes géométriques grâce à des fonctions de tracé, la technique du double buffering et les différents modes d'affichage qui permettent certains effets. Mais il y a une autre manière de bouger. Il y a le procédé d'animation classique, du type dessin animé, et son implémentation dans le cadre d'un programme. C'est le domaine des « sprites » selon la terminologie anglo-saxonne, un aspect essentiel de la programmation des jeux. Nous allons prendre pour base un bonhomme qui marche.

3.1 Sur place

Comme dans un dessin animé il s'agit d'une suite d'images qui décompose un mouvement et dont la succession va reproduire le mouvement. La figure 13.12 est la suite de dessins d'un bonhomme qui marche.

Algorithme de base d'une animation

Chaque pas du bonhomme est une petite image bitmap. Pour constituer l'animation il faut charger dans le programme la totalité des images puis les afficher les unes à la suite des autres. Le mieux est de stocker toutes les images dans un tableau, et dans l'ordre de l'animation, afin que les indices du tableau correspondent à l'ordre du déroulement de l'animation.
Après chargement des images dans le tableau, pour voir le bonhomme simplement marcher sans déplacement, l'algorithme sera :

1) Effacer l'image intermédiaire, le « buffer ».
2) Afficher l'image n_i courante.
3) Incrémenter n_i pour passer à l'image suivante. Si n_i égale l'indice de la dernière image, la valeur de l'indice de la première lui est affectée.
4) Afficher le buffer à l'écran.
5) Mettre le processus en boucle.

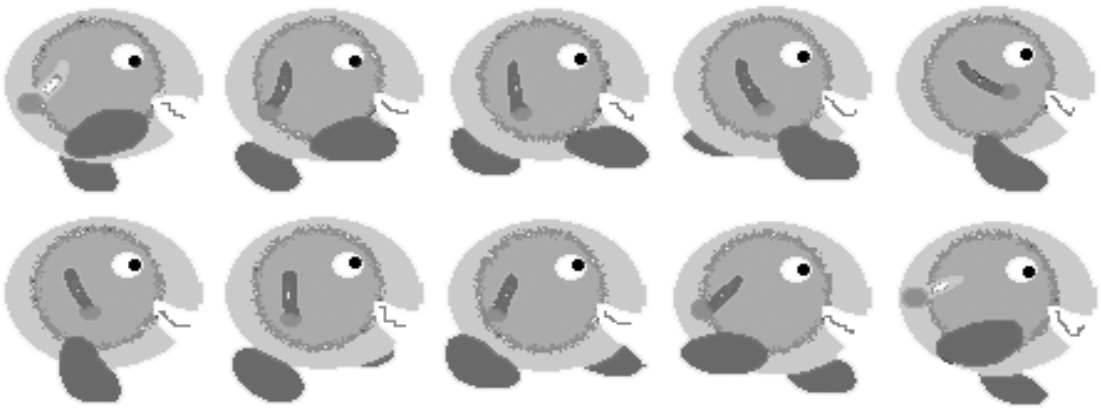


Fig. 13.12 : Décomposition d'un mouvement pour une animation

Bonhomme sur place

Sans déplacer le bonhomme l'algorithme peut s'écrire de la façon suivante (figure 13.13). Il est restitué pour ainsi dire textuellement aux lignes (5) à (10) :

```
(1)  if ( ! charger_dessins( ) ){
         allegro_message("Erreur pour charger_dessins\n%s\n", allegro_error);
         return 1;
     }
(2)  set_palette(pal);
(3)  page = create_bitmap(ECRAN_X, ECRAN_Y);

(4)  posx = (ECRAN_X/2)- (DESSIN[0]->w/2);
     posy = (ECRAN_Y/2)- (DESSIN[0]->h/2);
     dessin_cmpt = 0;

(5)  do{
(6)      clear(page);
(7)      draw_sprite(page, DESSIN[dessin_cmpt], posx, posy);
(8)      dessin_cmpt = (++dessin_cmpt)%10;
(9)      blit(page, screen, 0,0,0,0,ECRAN_X, ECRAN_Y);

(10) }while ( ! key[KEY_ESC] );
```

Fig. 13.13 : Un bonhomme qui marche

La construction s'appuie sur trois globales que nous déclarons au début du programme :

```
BITMAP* DESSIN[10];
BITMAP* page;
PALETTE pal;
```

Il y a le tableau des images de l'animation, l'image intermédiaire nommée « page » et une palette. D'autre part la figure 13.13 fait intervenir les variables « posx », « posy » pour la position, et « dessin_cmpt » pour la sélection d'une image.
La portion de code de la figure 13.13 peut être positionnée dans la fonction « main » telle que nous l'avons rencontrée plus haut, et ces variables être locales à la fonction « main ».

(1) Tout d'abord il faut pouvoir disposer des images dans le programme et donc les charger. C'est le rôle de la fonction « charger_dessins » (figure 13.14).

Cette fonction ne prend pas d'argument et renvoie une valeur entière.
Pour chaque image elle appelle la fonction « load_bitmap » (présentée à la ligne (2) de la figure 13.11). A chaque fois elle prend en argument le chemin d'accès au fichier bitmap. Ceux-ci ont été regroupés dans un dossier nommé « dessins » situé dans le même répertoire que le programme.

Ensuite il y a un test pour connaître si chacune des images du tableau a bien été correctement chargée. Si ce n'est pas le cas et qu'une des positions du tableau est nulle, la fonction retourne la valeur 0 si non elle retourne la valeur 1.

Ainsi l'appel de la fonction « charger_dessins » se traduit par une valeur de retour de 0 ou 1. Si c'est 0 le test est faux et les instructions sont effectuées : un message d'erreur est donné avant sortie du programme.

```
int charger_dessins(){
int i;
      DESSIN[0] = load_bitmap(".\\dessins\\bonh1.bmp", pal );
      DESSIN[1] = load_bitmap(".\\dessins\\bonh2.bmp", pal );
      DESSIN[2] = load_bitmap(".\\dessins\\bonh3.bmp", pal );
      DESSIN[3] = load_bitmap(".\\dessins\\bonh4.bmp", pal );
      DESSIN[4] = load_bitmap(".\\dessins\\bonh5.bmp", pal );
      DESSIN[5] = load_bitmap(".\\dessins\\bonh6.bmp", pal );
      DESSIN[6] = load_bitmap(".\\dessins\\bonh7.bmp", pal );
      DESSIN[7] = load_bitmap(".\\dessins\\bonh8.bmp", pal );
      DESSIN[8] = load_bitmap(".\\dessins\\bonh9.bmp", pal );
      DESSIN[9] = load_bitmap(".\\dessins\\bonh10.bmp", pal );

      for(i=0; i<10;i++){
            if( ! DESSIN[ i ] )
                  return 0;
      }
      return 1;
}
```

Fig. 13.14 : Récupérer les images de l'animation dans le programme

(2) Activation de la palette récupérée avec la série d'images (toute les images ont la même palette).

(3) Création de la bitmap intermédiaire pour le double buffer que nous avons mis à la taille de l'écran afin de pouvoir ensuite ajouter d'autres éléments à l'animation.

(4) Initialisation des variables, « posx » et « posy » sont pour la position des dessins. Ils ont tous la même taille et nous les avons centrés dans l'écran.
La variable « dessins_cmpt » est le compteur pour la succession des images en commençant de zéro jusqu'à neuf.

(5) et (10) Définition de la boucle « do-while » avec pour test d'arrête la pression de la touche échap.

Dans les instructions de la boucle nous retrouvons les étapes de l'algorithme avec :

6) Effacement du buffer.
7) Affichage de l'image courante avec la fonction d'affichage « draw_sprite » qui prend en argument l'image de destination, « page », l'image source DESSIN[i] et la position (posx, posy).
8) Incrémentation de un du compteur des images afin de passer à l'image suivante, mais avec un modulo 10 pour que les valeur tournent de 0 à 9.
9) Placage à l'écran de l'image intermédiaire « page », avec la fonction blit que nous avons déjà rencontrée.

Pour qu'il y ait un déplacement du bonhomme il suffit de modifier la position de l'image, les variables posx et posy, avant affichage dans l'image intermédiaire.

3.2 Autour de l'écran

Faisons courir le bonhomme autour de l'écran. Il y a quatre cas que la figure 13.15 résume.

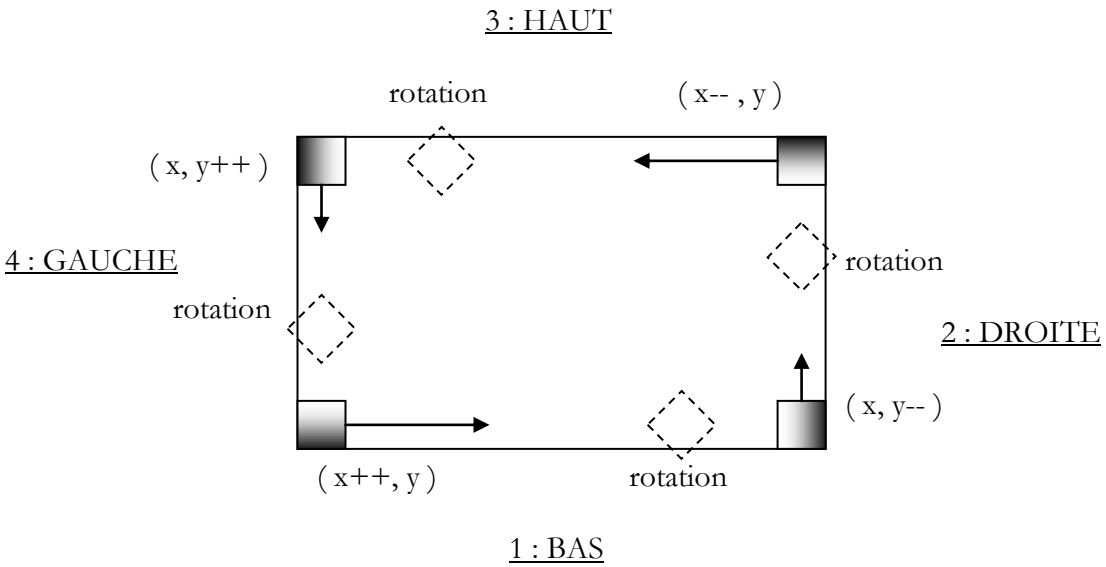


Fig. 13.15 : Machinerie à mettre en place pour faire tourner un bonhomme autour de l'écran

Le bonhomme part en bas, à gauche de l'écran. Il avance tout droit vers le bord droit de l'écran par l'incrémentation de sa position horizontale. Juste avant d'y arriver il pivote de 90° de façon à ce que ses pieds puissent adhérer au bord droit. Il avance ensuite vers le bord haut par décrémentation de sa position verticale. Avant d'y arriver il pivote à nouveau de 90°. Ce principe est répété pour les deux autres bords. Une fois en haut il avance par décrémentation de la position horizontale et sur le bord gauche après pivotement il avance par incrémentation de la position verticale. Le tout peut faire l'objet d'une boucle.

Trois variables et une énumération sont ajoutées au programme précédent. L'énumération définit quatre constantes, une pour chaque position, qui sont « bas », « droite », « haut », « gauche », une variable stocke la position courante « position », il y a une variable pour le pas d'avancement « pas » et une variable pour la rotation « rotation ». Les autres sont identiques au programme précédent. L'initialisation de l'énumération est faite en globale au début du programme de la façon suivante :

```
enum{ bas, droite, haut, gauche} ;
```

Enveloppe générale du processus

Le processus peut s'écrire ainsi (figure 13.16) :

(1) Tout d'abord il y a l'initialisation des variables. Comme le bonhomme part en bas, « position » prend la valeur « bas ». Il n'y a pas de rotation et « rotation » est à 0. Le pas d'avancement est fixé à 3.

Le bonhomme est à gauche de l'écran et c'est le point en haut à gauche de l'image qui détermine la position de l'image (figure 13.17)

Fig. 13.17 : Le point haut gauche de l'image détermine la position de l'image

Donc ses coordonnées sont 0 pour l'horizontale et la taille de l'écran moins la taille verticale de l'image pour la verticale.
La dernière variable est le compteur des images pour l'animation mis à 0.

```
(1)  rotation = 0;
     position = bas;
      pas = 3;
     posx = 0;
     posy = ECRAN_Y - DESSIN[0]->h;
     dessin_cmpt = 0;

(2)  do{
(3)       clear(page);
(4)       switch( position ){
               case bas :          /*  voir figure 13.18   */ break;
               case droite :       /*  voir figure 13.19   */ break;
               case haut :         /*  voir figure 13.20   */ break;
               case gauche :       /*  voir figure 13.21   */break;
          }
(5)       rotate_sprite(page, DESSIN[dessin_cmpt], posx, posy, itofix(rotation));
(6)       dessin_cmpt = (++dessin_cmpt)%10;
(7)       blit(page, screen, 0,0,0,0,ECRAN_X, ECRAN_Y);

(8)  }while ( ! key[KEY_ESC] );
```

Fig. 13.16 : Enveloppe de la machine à faire tourner le bonhomme autour de l'écran

(2) et (8) Boucle avec arrêt si la touche échap est pressée.

(3) Tout contenu de l'image intermédiaire d'assemblage est effacé.

(4) Un aiguillage qui reprend chacun des cas de la figure 13.15. L'avancement du personnage en fonction du bord de l'écran où il se trouve n'est pas le même dans chaque cas. Le test du switch est effectué sur la valeur de la variable « position ».

(5) A la sortie du switch l'image courante de l'animation du bonhomme est copiée sur l'image d'assemblage, le buffer intermédiaire. Nous avons choisi de faire pivoter l'image du bonhomme de façon à ce que ses pieds soient toujours contre le bord de l'écran. Il existe dans l'environnement une variante de la fonction « draw_sprite » que nous avons utilisée pour le bonhomme sur place. C'est la fonction « rotate_sprite » qui prend un argument

supplémentaire, le dernier :

```
void rotate_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y, fixed angle);
```

Cet argument est d'un type particulier à la librairie Allegro, le type « fixed ». Ce type décompose un entier codé sur quatre octets en deux parties de deux octets chacune afin d'avoir des nombres à virgule 16.16 (bits). Parmi les types définis par le C standard il existe plusieurs options pour les nombres à virgule baptisés « flottants ». Il y a le type « float » pour la simple précision, le type « double » pour les flottants double précision et le type « long double » pour une précision étendue. Le C définit des règles minimales pour leurs implémentations qui restent en partie dépendantes des environnements. A l'origine du type « fixed » dans l'environnement Allegro, l'objectif est certainement une optimisation de la rapidité des traitements. Il y a également la volonté d'une indépendance par rapport aux différentes implémentations des flottants. Une des vocations de la librairie est de pouvoir de fonctionner dans des environnements différents.

Cet argument, un flottant de type « fixed » est là pour donner des information d'angle. Le cercle est divisé en 256 unités et tourne dans le sens des aiguilles d'une montre. Pour remonter de 90° il faut passer la valeur -64, pour remonter de 180° c'est le double, -128 etc. Dans l'autre sens ce sont des valeurs positives.

(6) C'est l'instruction pour l'incrémentation du compteur d'images.

(7) Et pour finir, après assemblage dans le buffer, affichage à l'écran du résultat.

Détails des cas de chacun des bords de l'écran

Le premier cas est celui du bas avec la variable « position » initialisée avec la valeur de la constante « bas » (figure 13.18) :

```
    case bas :
(1)     posx += pas;
(2)     i = posx + DESSIN[dessin_cmpt]->h;

(3)     if ( i >= ECRAN_X){
            position = droite;
            posx = ECRAN_X- DESSIN[dessin_cmpt]->h;
            rotation = -64;
        }
(4)     else{
            i = ECRAN_X - i;
            if ( i <= 64*pas )
                rotation--;
        }
(5)     break;
```

Fig. 13.18 : Bonhomme en bas

(1) Incrémentation de la position horizontale courante avec l'addition du pas d'avancement à la variable posx.

(2) Variable outil « i » sert à stocker la position additionnée de la taille horizontale de l'image. Cette valeur est nécessaire pour savoir si le bord droit est atteint.

(3) Test sur la valeur de i pour savoir si le bord est atteint. S'il est atteint, c'est-à-dire si la valeur de i dépasse la longueur de l'écran, il faut passer au cas suivant. La variable position prend alors la valeur de la constante « droite ». La position horizontale sera fixe, égale au bord de l'écran moins la taille horizontale de l'image. Pour l 'affichage, il y a une rotation de 90° ce qui correspond au nombre -64 pour la fonction « rotate_sprite » que nous utilisons.

(4) En revanche si le bord de l'écran n'est pas atteint, la distance qui reste à parcourir est récupérée dans i (c'est la taille horizontale de l'écran moins la position de l'image additionnée de la taille horizontale de l'image stockée dans i). S'il reste 64 pas ou moins à parcourir pour atteindre le bord, la variable « rotation » est décrémentée de un.

(5) L'instruction « break » qui signifie interrompre, provoque la sortie du switch. Notons que le break, qui appartient au langage C et est un mot clé, peut également être utilisé pour la sortie d'une boucle for ou while. Il clôt chacun des quatre cas, et sans lui ils seraient tous effectués les uns à la suite des autres. Une instruction précieuse qu'il ne faut pas oublier.

Le cas suivant est celui de la variable « position » affectée de la valeur de la constante « droite » (figure 13.19) :

(1) Cette fois il s'agit de remonter le long du bord et c'est la variable y qui est décrémentée de la valeur de « pas ». La position en horizontale a été fixé à l'issue du cas précédent.

(2) Pas besoin de la variable i, les tests peuvent se faire directement sur la position verticale de l'image, la variable posy. Même principe que dans le cas précédent. Lorsque la position verticale sort de l'écran, il faut passer au cas suivant. La variable « position » prend la valeur de « haut », la rotation de l'image originale est de 128, soit 180° et la position en y devient fixe avec une valeur de 0 (le coin haut gauche de l'image touche le bord).

```
    case droite :
(1)     posy -= pas;
(2)     if ( posy <= 0){
            posy = 0;
            rotation = -128;
            position = haut;
        }
(3)     else{
            if ( posy <= 64*pas )
                rotation--;
        }
        break;
```

Fig. 13.19 : Bonhomme à droite

(3) Si le bord du haut n'a pas encore été franchi, il convient de regarder quelle distance en sépare encore l'image afin de commencer la rotation dès que la proximité sera de 64 pas. Le break clôt la séquence.

Le cas suivant est celui de la variable « position » affectée de la valeur de la constante « haut » (figure 13.20) :

(1) Presque identique au cas précédent, le déplacement est horizontal avec cette fois la variable posx décrémentée de « pas ».

(2) Si l'image sort de l'écran par la gauche, alors il faut passer au dernier cas, celui du bord gauche, « position » prend la valeur de « gauche ». La rotation est cette fois de 270° soit une valeur de -192, et la verticale reste identique avec posy à 0.

(3) Sinon, dès arrivée à 64 pas du bord la rotation commence avec « rotation » décrémentée de un à chaque nouveau pas.

```
      case haut :
(1)       posx -= pas;
(2)       if ( posx <= 0){
                posy = 0;
                rotation = -192;
                position = gauche;
          }
(3)       else{
                if ( posx <= 64*pas )
                rotation--;
          }
          break;
```

Fig. 13.20 : Bonhomme en haut

Le dernier cas est celui de la variable « position » affectée de la valeur de la constante « gauche » (figure 13.21) :

```
      case gauche :
(1)       posy += pas;
          i = posy + DESSIN[dessin_cmpt]->h;
(2)       if ( i >= ECRAN_Y){
             posy = ECRAN_Y - DESSIN[dessin_cmpt]->h;
             rotation = 0;
             position = bas;
          }
          else{
          if ( i >= ECRAN_Y - 64*pas )
              rotation--;
          }
          break;
```

Fig. 13.21 : Bonhomme à gauche

(1) A l'inverse du cas de droite, le bonhomme descend verticalement et la variable posy est incrémentée de « pas ». Par souci de lisibilité, i récupère la somme de la position courante et de la hauteur de l'image.

(2) Si i dépasse la taille verticale de l'écran, il passe au cas suivant, le premier. La variable « position » prend la valeur « bas ». Un tour complet a été effectué et « rotation » prend la valeur 0. La position verticale est bloquée à la hauteur de l'image au-dessus du bas de l'écran.

(3) Lorsque le bonhomme arrive à 64 pas du bord suivant, il entame sa rotation.

3.3 Dans une boîte en mouvement

Si maintenant le bonhomme est enfermé dans une boîte en mouvement, le principe précédent reste vrai. Comme autour de l'écran le bonhomme tourne dans sa boîte et il suffit d'ajouter le déplacement de la boîte à celui du bonhomme.

La boîte est un rectangle qui va fonctionner comme ceux que nous avons vus avec la présentation du double buffer. Toutes les variables nécessaires à la boîte sont regroupées dans une structure selon la déclaration globale suivante :

```
typedef struct BOITE{
        int x, y, tx, ty, px, py ;
}BOITE;
```

et un pointeur, global ou local à la fonction main est déclaré :

```
BOITE* boite ;
```

Les variables x et y correspondent à la position de la boîte, tx et ty à sa taille, px et py sont les pas d'avancement.

Par rapport à l'étape précédente il faut ajouter une fonction d'initialisation de la boîte (figure 13.22) :

Fig. 13.22 : Les champs de la structure BOITE sont initialisés

```
(1)  void initialiser_boite(BOITE* b)
     {
     int i;
(2)        b->tx = DESSIN[ 0 ]->w * ((rand()%5)+2 );
           b->ty = DESSIN[ 0 ]->h * ((rand()%5)+2 );

(3)        i = ECRAN_X- b->tx;
           b->x = rand()%i;
           i = ECRAN_Y- b->ty;
           b->y = rand()%i;

(4)        b->px = (rand()%3)-3;
           b->py = (rand()%3)-3;
     }
```

(1) La fonction « initialiser_boite » ne renvoie rien. Elle prend en argument un pointeur de type BOITE*, l'adresse d'une structure boîte afin d'attribuer à chaque champ une valeur.

(2) Tout d'abord c'est la taille de la boîte qui est initialisée et à partir de la taille de l'image du sprite. Elle est au minimum deux fois plus large et longue et au maximum cinq fois plus large et longue.

(3) Ensuite la position de la boîte est initialisée au hasard dans l'écran.

(4) Finalement ce sont les valeurs pour le déplacement de la boîte initialisées entre -3 et 3.

Après l'initialisation il y a à prévoir le déplacement de la boîte avec une fonction spécifique. Nous avons regroupé l'affichage de la boîte avec la fonction de déplacement (figure 13.23) :

```
(1)  void deplace_boite(BITMAP* bmp, BOITE* b, int* tx, int* ty)
     {
(2)       b->x += b->px;
          b->y += b->py;

(3)       if ( ((b->x+b->tx > ECRAN_X) && (b->px > 0 )) || ((b->x < 0) && (b->px <
0)) )
               b->px = -b->px ;
          if ( ((b->y+b->ty > ECRAN_Y) && (b->py > 0 )) || ((b->y < 0) && (b->py <
0)) )
               b->py = -b->py ;

(4)       *tx = b->x + b->tx;
          *ty = b->y + b->ty;

(5)       rectfill(bmp, b->x, b->y, *tx, *ty, 255);
     }
```

Fig. 13.23 : Bouger la boîte

(1) C'est la fonction « deplace_boite ». Elle ne renvoie rien et prend quatre arguments.
Il y a la bitmap de référence « bmp » sur laquelle va être affichée la boîte après son déplacement. Ensuite vient la boîte de référence « b » concernée par les opérations, avec un passage par adresse. Pour finir deux entiers passés par adresse afin de pouvoir récupérer la somme « position de la boîte plus sa taille », c'est-à-dire les deux bornes droite et basse de la boîte, comme nous avions le bas de l'écran et la droite de l'écran. C'est pour éviter ensuite d'avoir à répéter cette adition à chaque test où une de ces deux bornes est nécessaire lorsque le bonhomme se déplace que nous avons opté pour cette écriture, afin de la simplifier et de la clarifier autant que possible.

(2) Pour commencer les positions en x et y de la boîte sont incrémentées des pas respectifs px et py de la boîte.

(3) Ensuite deux tests pour savoir si la boîte demeure dans l'écran ou pas : si oui aucune instruction et si non les valeurs des pas d'avancement sont inversées.

(4) Les valeurs des bornes horizontale et verticale de la boîte (position plus taille) sont affectées aux entiers passés en argument via les pointeurs d'entiers tx et ty.

(5) La dernière instruction est l'affichage de la boîte sous forme d'un rectangle, à sa position et en fonction de sa taille avec une couleur choisie arbitrairement, celle de la position 255 de la palette.

Ensuite, l'algorithme de l'animation reste le même que celui de la figure 13.16 avec juste quelques modifications indiquées sur la figure 13.24 :

```
(1)  int tx, ty ;
          (…)

     do{
          clear(page);
(2)       deplace_boite(page, boite, &tx, &ty);
(3)       posx += boite->px;
          posy += boite->py;

(4)       switch( position ){
          case bas :        /* voir figure 13.25  */   break;
          case droite :     /* voir figure 13.26  */   break;
          case haut :       /* voir figure 13.27  */   break;
          case gauche :     /* voir figure 13.28  */   break;
          }
          rotate_sprite(page, DESSIN[dessin_cmpt], posx, posy, itofix(rotation));
          dessin_cmpt = (++dessin_cmpt)%10;
          blit(page, screen, 0,0,0,0,ECRAN_X, ECRAN_Y);

   }while ( ! key[KEY_ESC] );
```

Fig. 13.24 : Modification de l'animation : le bonhomme court dans une boîte.

(1)(2) Premièrement, deux variables locales à la fonction, « tx » et « ty », sont ajoutées qui vont prendre les valeurs des bornes horizontale et verticale de la boîte avec l'appel de la fonction « deplace_boite » en (2). Les arguments de l'appel sont l'image intermédiaire du montage « page », la boîte concernée « boite » et l'adresse de ces deux variables afin de pouvoir les modifier.

(3) Les positions horizontale et verticale du bonhomme suivent celles de la boîte, ce qui se traduit par l'ajout des pas d'avancement de la boîte aux positions horizontale et verticale de l'image du bonhomme.

(4) De même que dans le cas du bonhomme autour de l'écran, le switch aborde les quatre cas possibles : bas, droite, haut et gauche. Le test du switch porte sur la position courante du bonhomme qui est mémorisée par la variable locale « position ».

En ce qui concerne les quatre cas des quatre positions, ils sont traités de façon identique à celle vue précédemment si ce n'est que :

- Le bord de l'écran horizontal gauche (valeur 0) est remplacé par a position en x de la boîte.
- Le bord de l'écran horizontal droite (valeur ECRAN_X) est remplacé par la variable locale « tx » initialisée lors de l'appel de la fonction « déplace_boite ».
- Le bord de l'écran verticale bas (valeur ECRAN_Y) est remplacé par la variable locale « ty » initialisée lors de l'appel de la fonction « déplace_boite ».
- Le bord de l'écran vertical haut (valeur 0) est remplacé par la position en y de la boîte.

A titre indicatif, voici le détail du code source de chacun des cas bas, droite, haut, gauche :

```
case bas :
    posx += pas;
    i = posx + DESSIN[dessin_cmpt]->h;

    if ( i >= tx){
        posx = tx - DESSIN[dessin_cmpt]->h;
        rotation = -64;
        position = droite;
    }
    else
        if ( i >= tx-(64*pas) )
            rotation--;
    break;
```

Fig. 13.25 : Bonhomme en bas de la boîte

```
case droite :
    posy -= pas;
    if ( posy <= boite->y){
        posy = boite->y;
        rotation = -128;
        position = haut;
    }
    else
        if ( posy <= boite->y + (64*pas) )
            rotation--;
    break;
```

Fig. 13.26 : Bonhomme à droite de la boîte

```
case haut :
    posx -= pas;
    if ( posx <= boite->x){
        posy = boite->y;
        rotation = -192;
        position = gauche;
    }
    else
        if ( posx <= boite->x + 64*pas )
            rotation--;
    break;
```

Fig. 13.27 : Bonhomme en haut de la boîte

Fig. 13.28 : Bonhomme à gauche de la boîte

```
case gauche :
    posy += pas;
    i = posy + DESSIN[dessin_cmpt]->h;
    if ( i >= ty){
        posy = ty - DESSIN[dessin_cmpt]->h;
        rotation = 0;
        position = bas;
    }
    else
        if ( i >= ty - 64*pas )
            rotation--;
    break;
```

3.4 Ressources images, les fichiers « .dat »

La figure 13.14 qui montre la fonction « charger_dessins » laisse supposer qu'un problème peut se poser si le nombre des images est élevé, avec des séquences d'animation différentes, des décors, des objets etc. Sans compter que cette question va s'aggraver en cas d'utilisation d'échantillons sonores, de fichiers midi, de fontes pour l'esthétique des caractères etc. C'est-à-dire tout ce qu'il est convenu d'appeler les « ressources » dans un programme.

A titre indicatif, mentionnons simplement que l'environnement que nous utilisons fournit la possibilité de regrouper toutes les ressources d'un programme quelle qu'en soit la nature (image, son etc.) dans un seul fichier suffixé « .dat ». Ce fichier est obtenu via un utilitaire de l'environnement nommé le « grabber ». Ce programme permet la saisie de chacune des ressources, de les nommer, les organiser.

Indépendamment de leur nature (images, palettes, sons, fontes, etc.) toutes les ressources apparaissent ensuite sous la forme d'un tableau pour le programmeur. Chacune d'entre elles correspond à un indice du tableau. En effet, à la création du fichier de ressources, une option permet de générer automatiquement une librairie qui comprend pour chaque ressource une macro au nom de la ressource et à la valeur numérique de son indice dans la le tableau final. La question de l'organisation de la mémoire pour la réalisation de tels fichiers est vraiment intéressante, surtout si l'on songe qu'au bout du compte, du point de vue du programmeur, des types différents se trouvent en quelque sorte regroupés dans un tableau unique !

Enfin, après compilation du programme en cours de réalisation, le fichier de ressources peut soit être incorporé au programme, c'est-à-dire littéralement fusionné avec le programme et le programme n'en a plus besoin pour fonctionner, soit rester indépendant et le programme doit le charger au moment de son utilisation pour fonctionner. Ce chargement peut être entier ou sélectif, au fur et à mesure de l'action dans le programme.

Chapitre 14

Croquis de mouvements

Nous avons abordé des techniques du mouvement des couleurs, le déplacement de formes géométriques, une petite séquence d'animation et l'incorporation d'une séquence d'animation dans le mouvement d'une forme géométrique. L'objectif de ce chapitre-ci est d'aborder le mouvement intrinsèque d'une forme elle-même et de sa construction. Pour ce faire nous présentons trois programmes. Le premier est extrait des créations de la communauté Allegro, les deux autres des cours de Pierre Audibert [AUDIBERT, 2000].

Le scénario d'une forme

La forme est considérée comme faisant l'objet d'un scénario, un scénario d'engendrement de l'image. Ce scénario peut être d'inspirations très diverses. « Bouger pour aller où ? » est une question que le mouvement du bonhomme pose rapidement. De même au niveau du pixel et de la forme géométrique : comment organiser les pixels entre eux et les trajectoires de couleurs avec eux, selon quelles idées les formes initiales vont-elles s'enchevêtrer, s'additionner, se rencontrer et les couleurs s'organiser ? La forme finale revient à poser le principe du développement et de formation, la forme finale revient à bâtir une histoire.

La notion d'histoire, dans le cadre robographique de la création artistique et ludique se prête à différents niveaux d'investissements sémantiques. L'espace robographique en relation avec la notion de forme se décompose selon notre approche en trois dimensions. Il y a la dimension de l'effet, celle du croquis et celle de l'histoire proprement dite, qui a vocation de raconter.

Effets, croquis ou histoire

Par définition, l'effet peut ne pas être intéressant. L'effet conserve une sorte de neutralité et ne se prête pas encore à un investissement tel qu'il attire l'attention d'une façon qui vient rompre cette neutralité un peu fade du départ. L'effet peut toutefois devenir intéressant. Dans ce cas il perd de cette neutralité pour entrer progressivement dans un projet, de façon inconsciente. Il devient alors comme une sorte de fragment de rêve dont on ne sait pas encore de quoi il parle ou il va parler. Nous entrons là dans le domaine du croquis. Le croquis est un effet sélectionné, C'est un effet qui commence à parler de quelque chose à celui qui programme. Le croquis est un premier engagement dans un projet d'histoire.
Il pose une question symétrique au potentiel d'idée initiale que nous avons suggéré au chapitre Deux avec la notion de projet et la constitution d'un domaine de connaissance propre au projet. Le potentiel d'idée initiale pose la question « Qu'est ce que ça pourrait être ? » à propos d'une formule qui ne désigne aucune réalité, « la mouche de gratté-collé » par exemple. A propos du croquis, qui lui a une existence et fait déjà l'objet d'un programme, la question sous-jacente devient « qu'est-ce que ça pourrait dire ou faire qui raconte quelque chose ? Le croquis est à proximité d'une histoire et l'on espère la découvrir. Souvent les deux questions vont se compléter et, lorsqu'un croquis et un potentiel d'idée initiale se connectent ensemble, elles vont faire naître une histoire. D'où l'importance pour la création de la curiosité et de l'étude d'effets, lorsque la lecture du code source des programmes est ouverte et rendue public.

Evolutivité des matériaux dans l'espace robographique

Les programmes que nous proposons comme illustrations dans ce chapitre-ci sont tous des croquis mais ils sont plus ou moins investis d'un désir d'histoire. Cela ne signifie pas

qu'ils soient plus aboutis les uns que les autres du point de vue d'un projet éventuel. C'est l'histoire intrinsèque des projets et de leurs développements qui finalement décide du degré de maturité d'un croquis par rapport à une histoire. Le croquis renvoie au degré de maturité de la conscience d'une histoire, moins il est élevé plus on est proche de l'effet, plus il est élevé plus on est proche de l'histoire. Comme une partie du processus est inconscient, c'est difficilement mesurable et il y a de l'incertitude. Il peut même arriver que le croquis s'affirme pleinement en tant que projet à part entière et sans processus de développement plus avant. Dans certains cas un simple croquis peut être plus intéressant et adapté qu'un grand projet.

Toutefois c'est une particularité dans l'espace robographique que l'évolutivité des matériaux sous forme de programmes et de code source. La matière proposée est toujours une matière en devenir potentiel. D'où l'importance de l'ouverture du code source qui facilite le développement des projets et coïncide avec cette réalité d'une matière évolutive. La question de l'auteur n'est plus réduite à celle de l'œuvre et de sa propriété en tant qu'objet unique, la question de l'auteur est celle de l'identité de sa démarche liée à la singularité de sa personnalité et de son histoire individuelle. Il y a une certaine forme d'évanescence du support informatique lié à la volatilité de la mémoire numérique. Le programme informatique apparaît comme l'expression d'une communication à court terme dans sa forme, selon une modalité hybride entre parole et matière.

1. Effet de feu

1.1 Principe

L'idée de feu s'appuie d'abord sur la création d'une palette de couleur : un dégradé du noir au rouge puis au blanc en passant par l'orange et le jaune. Les flammes ont une allure de dessin d'enfant, montent du bas de l'écran vers le haut, du jaune le plus clair vers le plus foncé, le rouge sombre et finalement disparaissent dans l'obscurité. Pour ce faire il y a trois principales étapes.

La première est la définition de la première ligne de feu, c'est-à-dire, puisque les flammes montent, la dernière ligne en bas de l'écran. Sur cette ligne des segments sont définis à partir de points pris au hasard comme centres des segments. La couleur des segments est un dégradé qui, de gauche à droite, part du rouge le plus sombre et rejoint à nouveau le rouge le plus sombre en passant par le blanc. Eventuellement les segments peuvent se superposer ce qui ajoute un peu de relief à la régularité du dégradé.

En second lieu il s'agit de faire monter les flammes. La ligne du bas va être recopiée sur la ligne au-dessus avec au passage un léger assombrissement de la couleur. Ce principe de recopie ligne à ligne en partant du haut et en descendant vers le bas avec assombrissement progressif de la couleur fait l'objet d'une boucle jusqu'à ce qu'il n'y ait plus de couleur sur la ligne tout en haut de la flamme à une certaine hauteur de l'écran.

Tel quel l'algorithme donne un dégradé vers le haut des couleurs de la palette à partir des

centres des segments de la ligne initiale et il n'y a pas de mouvement. Pour qu'il y ait un mouvement il faut modifier légèrement et au hasard la ligne initiale du bas, c'est l'objet de la troisième étape. Les segments sont définis par un point sur une ligne : une distance est retranchée à gauche et ajoutée à droite du point, ce qui donne le segment par rapport au point. Il suffit de modifier légèrement la position du point pour créer le décalage, base de l'animation de la flamme. La figure 14.1 récapitule le principe de cette représentation.

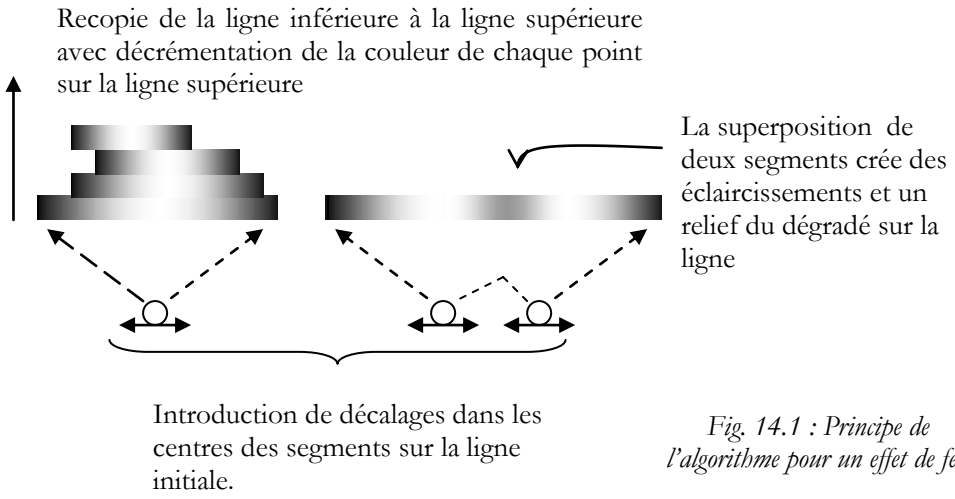


Fig. 14.1 : Principe de l'algorithme pour un effet de feu

1.2 Mise en forme et initialisations

Créer la palette

Il ne reste plus qu'à implémenter cette idée. La première fonction est celle de la création d'une palette (figure 14.2). C'est un dégradé constant presque identique à celui de la figure 10.2 vu au chapitre Dix.

(1) La palette « pal » est déclarée en globale au début du programme.

(2) La fonction « couleurs_flammes » ne renvoie aucune valeur et n'a pas de paramètre en argument.

(3) Le rouge augmente de 0 à 63 alors que vert et bleu restent à 0 (progression du rouge de plus en plus vif).

(4) Le rouge reste à 63 et le bleu à 0, le vert augmente progressivement de 0 à 63 (les orangers d'abord et jaunes ensuite de plus en plus vifs).

(5) Rouge et vert restent à 63, le bleu augmente progressivement de 0 à 63 (du jaune au blanc). La formulation est intéressante, plutôt que d'écrire « i-128 » c'est « i-192 », disons par coquetterie littéraire du domaine. Pour comprendre l'expression il ne faut pas oublier que seuls les six premiers bits comptent pour le codage de la couleur et également que 192 correspond à 1100 0000 en binaire. La soustraction 128-192 en binaire donne sur un octet non signé la valeur 192, la soustraction suivante 129-192 donne la valeur 193 soit en binaire 1100 0001, ce qui fait 1 pour les six premiers bits du codage de la couleur etc.

(6) Rouge, vert et bleu restent à 63, couleur blanche des indices 192 à 255 de la palette.

(7) Activation de la palette.

```
(1) PALETTE pal ;

(2)  void couleurs_flammes(void)
     {
     int i;

(3)        for (i=0; i<64; i++) {
                pal[i].r = i;
                pal[i].g = 0;
                pal[i].b = 0;
           }
(4)        for (i=64; i<128; i++) {
                pal[i].r = 63;
                pal[i].g = i-64;
                pal[i].b = 0;
           }
(5)        for (i=128; i<192; i++) {
                pal[i].r = 63;
                pal[i].g = 63;
                pal[i].b = i-192;
           }
(6)        for (i=192; i<256; i++) {
                pal[i].r = 63;
                pal[i].g = 63;
                pal[i].b = 63;
           }
(7)        set_palette(pal);
     }
```

Fig. 14.2 : Création de la palette aux couleurs de flammes

Initialiser les variables

Ensuite la structure de données essentielle est un tableau déclaré en global qui sert à stocker des positions sur une ligne afin de créer les segments. Nous l'avons appelé « braise » et fixé sa taille à 48 avec une macro pour pouvoir changer facilement ce nombre dans tout le programme. C'est la déclaration suivante :

```
#define NB_BRAISES  48
int braise[NB_BRAISES] ;
```

L'initialisation de ce tableau est pour chacune de ses positions un nombre pris au hasard entre 0 et la taille de la ligne qui est la taille horizontale de l'écran définie par la macro ECRAN_X.
Avec ce tableau est également en global une ligne temporaire en mémoire nommée « tmp ». C'est un pointeur de caractères non signés. Il va servir de tableau après allocation de mémoire à la taille voulue. C'est la définition :

```
unsigned char* tmp ;
```

Pour l'affichage nous avons opté pour un double buffering, avec une image intermédiaire toujours en globale « page » ce qui donne la déclaration :

```
BITMAP* page ;
```

Ainsi, avant d'entrer dans la boucle du processus d'animation, après la fonction « couleurs_flammes » pour la création de la palette, une fonction initialise le pointeur BITMAP *page, le pointeur unsigned char *tmp et le tableau d'entiers « braise » (figure 14.3) :

```
(1)  int initialisations(void)
     {
     int i;

(2)       page = create_bitmap(ECRAN_X, ECRAN_Y);
          tmp = (unsigned char *)malloc(sizeof(unsigned char) * ECRAN_X);
          if (!tmp || !page) {
               set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
               allegro_message("Pas assez de mémoire ?\n");
               return 0;
          }
          clear(page);

(3)       for (i=0; i<NB_BRAISES; i++)
               braise[i] = rand() % ECRAN_X;

(4)       return 1;
}
```

Fig. 14.3 : Initialisation des variables avant processus des flammes

(1) La fonction « initialisations » n'a pas de paramètre et renvoie un entier.

(2) Création de la bitmap « page » à la dimension de l'écran. Ensuite allocation de mémoire pour l'équivalent d'une ligne de l'écran codé en 8 bits (ECRAN_X fois la taille d'un caractère non signé). Au cas où un problème de mémoire surviendrait, test pour vérifier le bon fonctionnement des deux opérations. Si un problème est survenu un message d'erreur est envoyé via l'appel d'une fonction spécialisée « allegro_message » et ensuite la valeur de retour 0 provoque la sortie de la fonction. S'il n'y a pas eu de problème toutes les positions de la bitmap « page » sont mises à 0 avec l'appel de la fonction « clear ».

(3) Le tableau « braise » est un tableau d'entiers qui correspondent à des points sélectionnés au hasard sur une ligne. Pour avoir une position horizontale de chacun il suffit de tirer un nombre pseudo-aléatoire avec la fonction « rand » puis exercer un modulo avec la taille maximum de la ligne ECRAN_X. Ensuite affecter ce nombre à l'entier de la i-ème position du tableau « braise ».

(4) Tout a bien fonctionné et la fonction renvoie à la fonction appelante la valeur 1.

1.3 Processus générateur

Programme complet, boucle du processus

La figure 14.4 donne le détail de la fonction « main » :

(1) Après quelques initialisations d'usage propres à l'utilisation de la librairie Allegro, se trouve l'appel de la fonction « couleurs_flammes » pour la création de la palette qui nous intéresse.
Ensuite test sur la valeur de retour de la fonction « initialisations ». Si c'est 1, c'est-à-dire si tout s'est bien déroulé, le test est faux et le bloc est sauté. Sinon, le return provoque la sortie du programme.

(2) Boucle « tant que » du processus. Le test est l'appel d'une fonction qui renvoie simplement si oui (valeur 1) ou non (valeur 0) une touche du clavier a été pressée. Si oui, le test est faux et la boucle prend fin. Si non le test est vrai et la boucle continue.

```
    int main(void)
    {
        allegro_init( );
        install_keyboard( );
        if (set_gfx_mode(GFX_AUTODETECT, ECRAN_X, ECRAN_Y, 0, 0) != 0){
            allegro_message("Erreur pour le mode graphique\n%s\n", allegro_error);
            return 1;
        }
(1)     couleurs_flammes( );
         if ( ! initialisations( ) )
            return 1 ;

(2)     while ( ! keypressed ( ) ) {
(3)         ligne_initiale_de_feu( );
(4)         monte_flammes( );
(5)         blit(page, screen,0,0,0,0,ECRAN_X, ECRAN_Y);
        }
(6)     free( tmp);
        destroy_bitmap ( );
        return 0;
    }
    END_OF_MAIN( );
```

Fig. 14.4 : Programme pour un effet de feu

(3) Appel à la fonction « ligne_initiale_de_feu ». Le rôle de cette fonction est de modifier la ligne de base à partir de laquelle les flammes sont constituées. Le détail de cette fonction est donné avec la figure 14.5.

(4) Appel de la fonction « monte_flammes ». A partir du haut jusqu'en bas de l'écran, c'est la recopie de la ligne immédiate du bas à la lignes courante supérieures avec décrémentation progressive de la couleur comme l'indique plus bas la figure 14.6.

Détail de la fonction « ligne initiale de feu »

(1) « ligne_initiale_de_feu » ne renvoie rien et ne prend pas d'argument.

(2) Toutes les positions désignées par le pointeur tmp sont mises à 0.

(3) et (4) Boucles for imbriquées. Tout d'abord en (3) chaque position du tableau « braise » va être convoquée via l'incrémentation de la variable i. Pour chaque indice i, braise[i] contient une valeur de position en x sur une ligne. Pour faire un segment il suffit de lui retrancher une valeur à gauche et de l'ajouter à droite. La taille du segment est fixée à 40 pixels.
En (4) l'objectif est de parcourir chaque segment. Pour ce faire la valeur de départ à gauche est récupérée dans la variable « dx » avec braise[i] – 20. Le test d'arrêt est effectué sur la valeur de fin du segment à droite avec braise[i]+20. Le segment est de -20 à 20 pixels à partir de la position braise[i] ; dx est incrémentée de un en un et parcourt toutes les positions du segment.

```
(1)  void ligne_initiale_de_feu(void)
     {
     int i, dx;

(2)        for ( i=0; i<ECRAN_X; i++ )
               tmp[ i ] = 0;

(3)        for ( i=0; i<NB_BRAISES; i++ ) {
(4)             for ( dx = braise[ i ] – 20; dx < braise[ i ] + 20; dx++ )

(5)                  if ((dx >= 0) && (dx < ECRAN_X))
                         tmp[ dx ] = MIN( tmp[ dx ] + 20 – ABS( braise[ i ] – dx ), 192 );

(6)                  braise[ i ] += (rand() & 7) - 3;
(7)                  if (braise[ i ] < 0)
                         braise[ i ] += ECRAN_X;
                     else
                         if (braise[ i ] >= ECRAN_X)
                             braise[ i ] -= ECRAN_X;
               }
(8)        memcpy(page->line[ECRAN_Y-1],tmp, sizeof(unsigned char)*ECRAN_X);
}
```

Fig. 14.5 : Fonction pour établir et modifier la ligne de base sur laquelle s'appuie l'édifice des flammes

(5) Si la valeur de dx fait que dx est toujours dans la ligne et ne sort pas de l'écran, la position courante dans le segment va être affectée d'une valeur pour la couleur du pixel.

La formule pour obtenir cette valeur utilise la macro « MIN » qui donne la valeur minimum entre deux valeurs : MIN(a, b) donne a si a<b et b sinon.

Le deuxième argument de MIN est 192, le premier se décompose en trois parties :

$$\underbrace{tmp[\,dx\,]}_{3} + 20 - \underbrace{ABS(\,braise[\,i\,] - dx\,)}_{1}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{2}$$

1) donne la distance de dx au centre du segment braise[i] en positif à gauche et négatif à droite, mais restitué en valeur absolue.
2) Comme 20 est la distance maximum d'une extrémité au centre, en soustrayant la valeur absolue de la distance de dx au centre, plus dx est près des bords plus le résultat est petit et plus dx est près du centre plus le résultat est grand. Le résultat va de 0 à 20 et de 20 à 0.
3) Uniquement pour le premier segment toutes les positions de tmp sont à 0. Mais ensuite des segments peuvent se superposer comme nous l'avons indiqué dans la figure 14.1. La valeur de chaque position de tmp est additionnée ce qui surélève d'autant les valeurs calculées en 2), et crée un effet de superposition, comme des collines à partir du mouvement de crescendo et decrescendo.

   C'est-à-dire que l'augmentation de l'intensité de la couleur dépend du rapport entre le nombre de « braises » et la taille horizontale de l'écran. Le rapport est ici prévu à l'origine pour un écran tout petit de 320 pixels sur 200 pixels.

Si le résultat final de ce calcul est inférieur à 192, il est affecté à la position dx de tmp, sinon tmp prend une valeur maximum de 192

(6) Ensuite la position du centre de segment braise[i] est légèrement modifiée au hasard dans la fourchette de – 3 à 3 pixels, afin de faire bouger la ligne et de créer l'animation souhaitée.

(7) L'écran est considéré comme circulaire, c'est-à-dire que si braise[i] dépasse à gauche de l'écran il est renvoyé à droite et inversement s'il dépasse à droite il est renvoyé à gauche de l'écran.

(8) La dernière instruction est la recopie de cette nouvelle ligne de base, en bas de l'écran, sur la bitmap intermédiaire « page ». Pour ce faire la fonction « memcpy », qui fait partie de la librairie standard du C <string.h>, est employée. Cette fonction copie à l'adresse du premier argument la taille de mémoire passée en troisième argument et à partir de l'adresse passée en deuxième argument. En l'occurrence la ligne tmp est recopiée toute entière à l'adresse de la dernière ligne de la bitmap « page ».

Détail de la fonction « monte_flammes »

Après création de la ligne de base il reste à faire monter la flamme. C'est le rôle de la fonction « monte_flammes » de la figure 14.6.

(1) « monte_flammes » ne prend pas de paramètre et ne renvoie rien.

(2) Boucles for imbriquées. La première examine les lignes, la seconde les pixels de chaque ligne.

(3) Le principe est de partir à la hauteur maximum des flammes et de descendre ligne par ligne. A chaque ligne, il s'agit de copier la ligne de dessous (y+1) sur la ligne courante y. Pour ce faire la valeur de couleur de chaque pixel de la ligne de dessous est récupérée dans la variable c. Si elle est supérieure à 0 elle est décrémentée de 1. La valeur c obtenue est affectée à la même position horizontale sur la ligne courante y. C'est ainsi que le dégradé vers le haut des flammes est obtenu. C'est également ce qui donne la hauteur maximum des flammes. La valeur de couleur est 192 au maximum et elle décroît à chaque ligne copiée de un en un jusqu'à 0.

```
(1)  void  monte_flammes ( void )
     {
     int y,x,c;

(2)        for ( y = 192;  y < ECRAN_Y – 1; y++ )
                for ( x = 0; x < ECRAN_X; x++ ) {

(3)                  c = page->line[ y+1 ][ x ];
                     c=( c > 0 ) ? - -c : 0;
                     page->line[ y ][ x ] = c;
                }
     }
```

Fig. 14.6 : Faire monter les flammes

Maintenant il est possible de faire traverser des flammes à notre bonhomme du chapitre précédent, voire de le faire courir dans une boite enflammée ou au contraire de le faire s'enfuir à la vue des flammes, d'être poursuivi par des flammes. Les flammes peuvent aussi perdre leur caractère de flammes ordinaires et changer de couleur ce qui est assez amusant et puis il y a à voir l'étude de flammes sur une surface circulaire etc.

2. Croquis de sable

2.1 Principe

Dans l'espace du pixel, il s'agit de décrire une avalanche de cubes, tous de même taille. Comme le montre la figure 14.7, il s'agit de passer d'une répartition instable à un tas homogène faisant une pente de 45°.

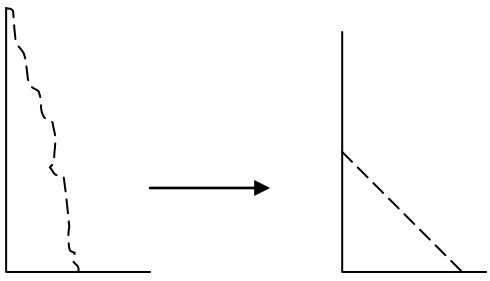


Fig. 14.7 : Avant et après l'avalanche de sable

Le moyen utilisé pour décrire le tas est tout simplement un tableau d'entiers comme l'indique la figure 14.8 qui suit. En effet le tas correspond à une superposition de lignes de cubes. Chaque indice du tableau correspond à une ligne et la valeur à cet indice est le nombre de cubes sur la ligne.

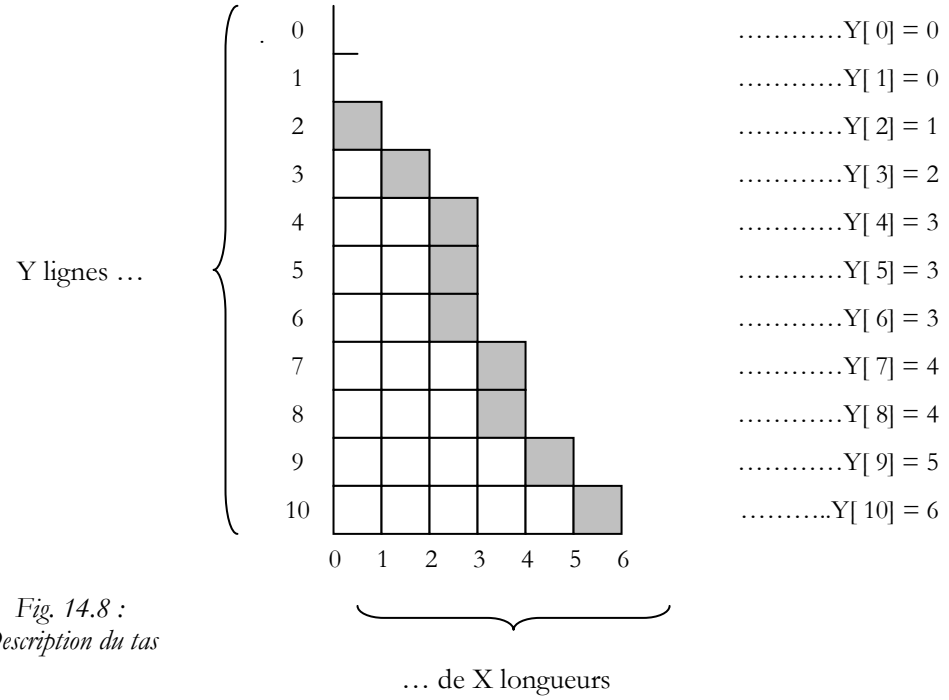


Fig. 14.8 : Description du tas

L'objectif final est de réduire les écarts afin d'avoir une liste du type Y[0] = 0, … $Y[n_1]=1$, $Y[n_2]=2$, $Y[n_3]=3$, $Y[n_4]=4$ etc. ce qui donne le tas équilibré avec sa pente à 45° après la chute des cubes.

Le sens de la chute est un glissement de la surface de la pente et les cubes roulent vers le bas. Deux situations sont isolées, la falaise et le surplomb.

Deux mouvements de base : falaise et surplomb

La falaise survient lorsque plusieurs lignes consécutives de cubes ont la même longueur. C'est-à-dire lorsque Y[i] = Y[i + 1] (figure 14.9 a)

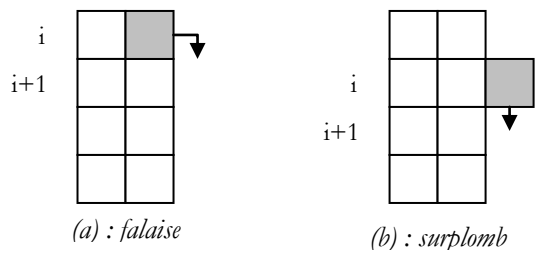


Fig. 14.9 : Deux mouvements de base

(a) : falaise *(b) : surplomb*

Le surplomb se produit lorsque le nombre de cubes de la ligne courante est supérieur à celui de la ligne qui suit soit lorsque Y[i] > Y[i + 1] (figure 14.9 b).

L'objectif est 1) de repérer les falaises afin de provoquer des surplombs en poussant le premier cube de la falaise et 2) de repérer les surplombs et de faire avancer les cubes en surplomb jusqu'à ce qu'ils provoquent une falaise.

2.2 Mise en forme et initialisations

Le tas de cubes

Question structure de données, le programme s'appuie simplement sur un tableau d'entier s«Y » de taille « YMAX » (facilement modulable) selon la déclaration suivante :

```
#define YMAX 100
int Y[YMAX];
```

En ce qui concerne l'affichage, nous avons opté pour un double buffering qui donne une animation plus fluide, d'où une bitmap de montage déclarée en globale. La palette est également déclarée en globale et deux macros donnent la taille souhaitée pour les grains d'où les déclarations :

```
BITMAP *page;
PALETTE pal;
#define pasx     4
#define pasy     4
```

Avant le processus d'avalanche lui-même il faut un tas de cubes. Pour simplifier nous allons juste traiter un versant de montagne et le tas s'appuie contre un bord de l'écran (figure 14.10).

```
(1)  void init_montagne(void)
     {
     int i,j ;
     int c=1;
     int hasard;

(2)        Y[ 0 ]=1;
           for( i=1; i < YMAX; i++ ){
                hasard=rand( )%2;
                if( ! hasard )
                     Y[ i ]=Y[ i-1 ];
                else
                     Y[ i ]=Y[ i-1 ]+rand( )%2;
           }
(3)        for( i=0; i < YMAX; i++ )
                for( j=1; j <= Y[ i ]; j++ )
                     carre( page, j, i, c );
     }
```

Fig. 14.10 : Créer le tas initial de cubes

(1) La fonction « init_montagne » ne prend pas d'argument et ne renvoie rien. Elle utilise quatre variables locales, des entiers. L'entier « c » correspond à une couleur de la palette, il est initialisé arbitrairement sur 1.

(2) Il s'agit de l'initialisation du tableau Y. La première position, le cube du haut est forcée à un. Il y a un cube tout en haut pour commencer. Ensuite boucle for pour passer en revue toutes les autres positions du tableau. L'entier « hasard » prend une valeur de 0 ou de 1 selon le retour de l'appel au générateur de nombres pseudo-aléatoires « rand ». Si c'est 0, la i-eme position de « Y » prend la même valeur que la valeur de la position précédente, ce qui va dans le sens de créer une falaise. Si « hasard » est égal à 1 alors la position courante Y[i] prend pour valeur la valeur précédente à laquelle est ajouté soit 0, soit 1. Ce second appel à la fonction « rand » n'est pas nécessaire, c'est juste afin d'augmenter la probabilité d'avoir des grandes falaises.

(3) Ensuite il reste à afficher le tas. C'est l'objectif de la boucle for imbriquée. Les lignes sont données par les indices i du tableau Y et le nombre de cubes par ligne est donné par la valeur contenue à la position i du tableau. Ainsi pour parcourir dans l'ordre le tas en passant par chaque cube, on prend chaque ligne i, première boucle, et pour chaque ligne on avance de 0 à la fin de la ligne qui est la valeur contenue dans Y[i] (deuxième boucle). Enfin, chaque position (i, j) détermine la position d'un cube. Il est affiché sous la forme d'un carré de couleur c avec l'appel à la fonction « carre » (figure 14.11).

fonction « carre »

Fig. 14.11 : Chaque cube ou grain est représenté par un petit carré

```
void carre(BITMAP* bmp, int x,int y, int c)
{
int xe, ye;

        for( xe = pasx*x; xe > pasx*(x-1); xe-- )
                for ( ye = pasy*y; ye < pasy*(y+1); ye++ )
                        putpixel(bmp, xe, ye, c);

}
```

La fonction « carre » ne renvoie rien. Elle prend quatre arguments qui sont une bitmap de référence, deux entiers « x » et « y » pour la position respectivement horizontale et verticale et un entier « c » pour la couleur.

La taille du carre (ou rectangle) dépend des valeurs données aux macros « pasx » et pasy » qui définissent ses côtés. Ils sont coloriés avec la fonction putpixel qui parcourt tous leurs pixels.

Par rapport au bord gauche de l'écran, le bord droit du carré est à x*pasx. Le bord gauche du carré en est à (x – 1)*pasx. La valeur « xe » part de la droite du carré et le parcourt en revenant vers la gauche. La position verticale du carré est à y*pas du haut de l'écran et il va jusque pas*(y+1).

2.3 Processus générateur

Deux modalités d'avalanche

Il y a deux modalités de glissement pour l'avalanche : soit l'écroulement a lieu grain par grain, du haut vers le bas jusqu'à ce que la tas n'ait plus ni falaise ni surplomb, soit tous les grains en surplomb ou sur une falaise descendent quasi simultanément. Les résultats visuels ne sont pas identiques. Grain par grain le tas se défait assez méthodiquement et l'on a l'impression de voir le tas initial descendre alors que celui de l'équilibre monte. Le phénomène paraît un peu plus réaliste, plus fluide, avec l'affaissement de toutes les falaises ensembles. La figure 14.12 donne la version grain par grain de l'avalanche et la figure 14.13 en donne la version toutes les falaises ensembles.

Avalanche grain par grain

```
(1)  int avalanche_grain_par_grain(void)
     {
     int c = 150;
     int cmpt=0;
     int i, k;

(2)        for( i = 0; i < YMAX - 1; i++ )
               if( ( Y[ i ] > Y[ i+1 ] )  ||  ( Y[ i ]==Y[ i+1 ] )  && ( Y[ i ]!=0 ) ){

(3)                    k = ( Y[ i ] > Y[ i+1 ] ) ? 0 : 1 ;

(4)                    carre( page, Y[ i ], i , 0 );
                       carre(page, Y[ i ]+k, i + 1, c);

(5)                    Y[ i+1 ]++;
                       Y[ i ]- -;
(6)                    cmpt++;
               }
          return cmpt;
     }
```

Fig. 14.12 : En commençant par le haut, chaque grain descend jusqu'en bas

(1) La fonction « avalanche_grain_par_grain » ne prend pas d'argument mais renvoie une valeur entière. Elle utilise quatre variables locales qui sont des entiers. « c » est pour la couleur des grains et est initialisée arbitrairement à 150. « cmpt » va prendre la valeur renvoyée par la fonction. Cette variable est initialiée à 0. Si des opérations ont lieu elle prend une valeur positive sinon elle reste à 0. « i » et « k » sont des variables outils pour la boucle et le déplacement des cubes.

(2) La boucle for indique que l'on parcourt toutes les positions du tableau Y. Toutes les instructions dépendent d'un test if sous lequel elles sont regroupées. Le test est décomposé en trois volets. Les deux premiers décrivent respectivement la situation du surplomb (figure 14.9 b) et celle de la falaise (figure 14.9 a). Comme les longueurs des lignes

changent avec l'avancement du processus, le dernier test sert uniquement à borner le processus lorsqu'il n'y a plus de cube sur la ligne. Ce sont donc surtout les deux premiers tests qui sont importants. Cela peut s'énoncer de la façon suivante : si l'on est soit dans le cas d'un surplomb, soit dans le cas d'une falaise et qu'il y a encore des cubes sur la ligne, alors les instructions du bloc sont effectuées.

(3) La variable k sert au déplacement horizontal d'un cube. De deux choses l'une, soit le cube est en surplomb et il faut descendre mais il n'y a pas d'avancée horizontale. Soit le cube est le premier d'une falaise et il doit se décaler vers la droite de 1, afin de se mettre en surplomb pour commencer sa descente. L'utilisation d'un test conditionnel permet d'initialiser la variable k de la façon suivante : « est-ce que l'on est en surplomb ? Si oui k prend une valeur de 0 (pas d'avancée horizontale) et sinon, k prend une valeur de 1 (avancée horizontale).

(4) Lorsque la valeur k est fixée, le mouvement est possible. L'appel de la fonction « carre » est pour l'affichage du mouvement. D'abord le cube courant de fin de la i-ème ligne du tas est effacé. Ensuite il est affiché avancé horizontalement de « k » et verticalement de 1.

(5) Ce résultat, qui modifie l'allure du tas, doit être répercuté sur les données du tableau Y. Puisque le cube est descendu de un la ligne suivante est plus longue de un cube, d'où l'incrémentation de un de la position suivante du tableau. Inversement la ligne que le cube a quitté est plus courte de un d'où la décrémentation de un de la longueur de la ligne courante.

(6) Pour finir, le rôle de la variable cmpt est de témoigner du passage par le bloc d'instructions et elle est incrémentée de un. Ce passage signifie qu'il y a encore des falaises ou des surplombs. Dès qu'il n'y en a plus le tas a trouvé sa forme finale, la variable cmpt qui est initialisée à 0 au début de la fonction mais n'entre plus dans le bloc et conserve sa valeur de 0, cmpt est renvoyée pour indiquer la fin du processus à la fonction appelante.

Avalanche simultanée des falaises

Une légère modification permet d'obtenir le même résultat final mais avec un mouvement d'ensemble. Il s'agit d'activer toutes les falaises et surplombs, plutôt que d'attendre la chute complète de chaque grain. Cela crée une situation qui n'est pas envisagée précédemment.

En particulier il y a à éviter de creuser la falaise. C'est-à-dire qu'après le déclenchement d'un surplomb, les cubes suivants de la falaise restent à leur place ou avancent uniquement s'ils sont déjà en surplomb. En quelque sorte il y a un seuil pour le déclenchement de nouveaux surplombs, ce ne peut être qu'après la position horizontale courante de valeur Y[i], c'est-à-dire que le seuil est la taille de la ligne plus un. Pour ce faire une variable est utilisée qui interdit le déclenchement de surplombs si la position horizontale Y[i] est inférieure au seuil, comme l'indique la figure 14.13.

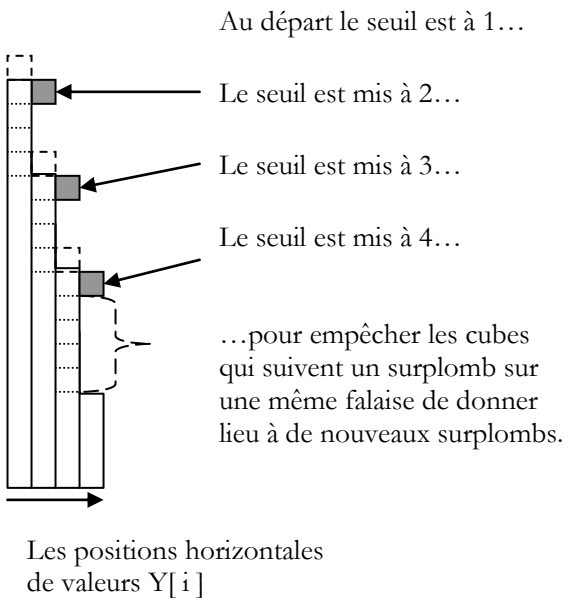


Fig. 14.13 : Contrôler les surplombs.

Voici le détail des modifications apportées dans le code source (figure 14.14) :

```
    int avalanche_falaises_ensembles(void)
    {
    int c = 150;
    int cmpt=0;
    int i, k;
(1) int seuil=1;

        for( i = 0; i < YMAX - 1; i++ )
            if( ( Y[ i ] > Y[ i+1 ] ) || ( Y[ i ]==Y[ i+1 ] ) && ( Y[ i ]!=0 ) &&
(2)             ( Y[ i ]>= seuil ) ){

(3)             seuil = Y[ i ]+1 ;

                k = ( Y[ i ] > Y[ i+1 ] ) ? 0 : 1 ;

                carre( page, Y[ i ]+, i, 0 );
                carre( page, Y[ i ]+k, i+1, c );

                Y[ i+1 ]++;
                Y[ i ]- -;
(4)             i++;
                cmpt++;
            }
        return cmpt;
}
```

Fig. 14.14 : Chaque falaise s'effondre simultanément

(1) Déclaration de la variable locale « seuil » qui va servir de seuil. Le premier seuil est 1.

(2) La seconde modification concerne le test d'accès aux instructions du bloc. Il ne suffit plus que le cube courant soit un surplomb ou une falaise et non nul, il faut en plus qu'il soit supérieur au seuil courant.

(3) Si le cube courant satisfait à ces contraintes, il y a entrée dans le bloc et tout de suite le seuil est mis à jour et prend la valeur horizontale immédiatement supérieure à celle de Y[i]. Afin d'éviter une confusion rappelons que « Y[i+1] » est le cube de la ligne suivante, celui du dessous, alors que « Y[i] +1 » est la position horizontale supérieure de un.

(4) Nouveauté également l'incrémentation de un de l'indice i des lignes. En effet, une fois le cube déplacé, il a avancé de un et se trouve sur la ligne suivante en i+1. Pour éviter de retomber sur lui l'itération suivante de la boucle, il faut avancer maintenant de un pour que la boucle qui va encore ajouter un à i passe effectivement au cube suivant.

Le croquis de sable est une bonne piste pour l'étude d'un déplacement d'ensemble de cubes en tas. Une sorte de tas ambulant qui pourrait se mouvoir de façon fluide mais sans s'éparpiller, dans un roulement et effondrement continus, épousant toutes sortes de reliefs et pouvant prendre n'importe quelle direction. A voir également un travail original sur la couleur…

3. Vers une histoire de danse à partir de flammes

Il s'agit d'un croquis dont l'identité initiale est la représentation de la propagation d'un feu de forêt. Après discussion avec des amis, observations du résultat et quelques modifications du programme original, l'idée est venue d'un travail en direction de la danse. C'est-à-dire que ce croquis est un projet potentiel, même si l'idée de projet une fois née du croquis peut évoluer et aboutir finalement à se passer du croquis de départ. Actuellement ce projet est en suspens. Le croquis permet d'en conserver la mémoire et se pencher sur le croquis suscite une dynamique de créativité. Nous partons de l'algorithme initial, ensuite nous apportons la modification qui nous oriente vers un questionnement sur la danse.

3.1 Principe

C'est un automate cellulaire. Nous avons rencontré ce sujet au chapitre Un à propos de la réalisation d'un logiciel de musique et au chapitre Sept avec les systèmes génératifs. L'automate cellulaire fait évoluer un plan en soumettant chaque position qui le constitue à des lois de transition. Ces lois se fondent sur le voisinage des positions entre elles, nombre de voisins, qualité des voisins etc. Là il s'agit de représenter un phénomène de propagation, le feu dans une forêt.

L'espace est celui d'une matrice. Au départ il est vide et toutes les positions sont à zéro. Des arbres y sont disposés, c'est-à-dire que des positions prennent une valeur, par exemple un. Des foyers d'incendies sont ajoutés et des positions sur la valeur un des arbres prennent une valeur autre, par exemple deux. Tout est alors prêt pour la mise en œuvre du processus qui est très simple : toutes les positions de la matrice sont parcourues, si la position est un arbre et qu'il y a une position voisine sur feu, elle passe à la valeur feu ; en revanche, si la position est à feu, et bien elle passe à cendre et prend une autre valeur disons 3. A partir de ce petit moteur plusieurs extensions sont possibles. C'est souvent le cas avec des automates cellulaires. L'automate cellulaire est une figure attachante pour l'imagination parce qu'elle soumet facilement des énigmes. Par exemple ici la quête d'un feu infini.

3.2 Mise en forme et initialisations

Tout d'abord, la forme de l'espace est une matrice de int « P » (Plan) et sa taille est définie par deux macros. Le fonctionnement réclame une seconde matrice « NP » (Nouveau Plan) en miroir de la première afin de stocker les résultats de la transition pour l'ensemble des positions. Ce sont au début du programme les déclarations :

```
#define TX          200
#define TY          150
int P[TX][TY];
int NP[TX][TY];
```

Les différentes valeurs possibles pour chaque position sont isolées avec trois macros :

```
#define BOIS        1
#define FEU         2
#define CENDRE      3
```

Sur le plan visuel les arbres sont représentés par des carreaux de couleur sur l'écran et l'affichage est facilité par une série de macros utilitaires. Avec la taille de l'écran, il y a la définition d'une longueur « pasx » et d'une hauteur « pasy » des carreaux. Elles correspondent au rapport entre la taille de l'écran et la taille de la matrice :

```
#define ECRAN_X     800
#define ECRAN_Y     600
#define pasx    ECRAN_X/TX
#define pasy    ECRAN_Y/TY
```

Ensuite deux macros donnent la coïncidence sur l'écran des positions de la matrice (Compte tenu de pasx et pasy) :

```
#define Y(n)    (n)*pasy
#define X(n)    (n)*pasx
```

Reste encore deux macros pour la définition d'une densité des arbres lors de l'initialisation de la forêt. Toutes ces valeurs sont modifiables facilement pour toutes sortes de tests :

```
#define MAXI        1000
```

```
#define SEUIL          0.6*MAXI
```

Enfin, trois variables globales. Une palette, une image intermédiaire (Nous optons pour un double buffering) et un entier pour compter le nombre des arbres en feu.

```
PALETTE pal;
BITMAP *page;
int cmpt_feu;
```

Initialisation des bois

L'initialisation des bois consiste à parcourir la matrice P et pour chaque position attribuer soit la valeur BOIS en fonction de la valeur de SEUIL, soit la valeur 0. C'est la fonction « init_bois » figure 14.15 :

Fig. 14.15 : Une certaine densité de bois au départ

```
(1)  void init_bois( void )
     {
     int i, j, hasard;

(2)        for ( i = 0; i < TX; i++ )
                for ( j = 0; j < TY; j++ ){

(3)                hasard = rand( )%MAXI;
                   if ( hasard < SEUIL )
                        P[ i ][ j ] = BOIS;
                   else
                        P[ i ][ j ] = 0;
                }
     }
```

(1) « init_bois » ne prend pas d'argument et ne renvoie rien.

(2) Boucles for imbriquées qui donne accès à chaque position de la matrice.

(3) Affectation de la valeur de retour de l'appel de la fonction « rand » à la variable « hasard ». Ensuite comparaison de hasard à la valeur de SEUIL. Si hasard est inférieur alors la position (i, j) dans la matrice prend la valeur BOIS, sinon elle prend la valeur 0.

Initialisation du feu

En ce qui concerne le feu un front est allumé en bordure gauche de l'écran. C'est la fonction « init_feu » (figure 14.16) :

(1) « init_feu » ne prend pas d'argument mais renvoie un entier.

(2) La boucle for permet de parcourir une colonne. La position horizontale i reste toujours la même, à 1 du bord, et toutes les positions verticales sont passées en revue. Pour chacune, si elle vaut BOIS elle passe à FEU et une variable compte tous les passages à FEU.

(3) Deuxième boucle dont la vocation est la recopie de la matrice P dans la matrice NP à l'aide de la fonction « memcpy » déjà utilisée avec la fonction « ligne_initiale_de_feu » figure 14.5. La copie se fait ligne par ligne. Rappelons en effet que la première taille d'une matrice donne le nombre de lignes et la seconde le nombre de colonnes. Le nombre des lignes correspond ici à la taille horizontale et le nombre de colonne à la taille verticale.

```
(1)  int init_feu(void)
     {
     int i, j, cmpt;

(2)        for ( i = 1, j = 0, cmpt = 0; j < TY; j++ )
               if ( P[ i ][ j ] == BOIS ){
                    P[ i ][ j ] = FEU;
                    cmpt++;
               }
(3)        for ( i = 0; i < TX; i++ )
               memcpy( NP[ i ],P[ i ], sizeof( int )*TY );

(4)        return cmpt;
}
```

Fig. 14.16 : La ligne du bord gauche de la matrice est mise à FEU

(4) Pour finir, le compte du nombre de feux est renvoyé à la fonction appelante.

3.3 Processus génératif

Fonction de propagation

Le feu se communique des cellules sur FEU aux cellules sur BOIS voisines. Les cellules sur FEU passent ensuite sur CENDRE. La fonction de propagation est la suivante (figure 14.17) :

(1) « propagation » ne prend pas d'argument et ne renvoie rien.

(2) Outre les variables i et j utilitaires pour passer en revue les positions de la matrice avec les boucles for imbriquées, quatre autres variables entières vont servir à désigner les positions adjacentes (est, nord, ouest, sud) à la position (i, j) courante.

(3) Première boucle for qui passe en revue toutes les lignes de la matrice. Les positions adjacentes (i + 1) et (i – 1) sont conservées respectivement dans les variable « ia » et « ib ». C'est surtout pour une question de clarté du code. Notons que nous optons pour un écran circulaire, c'est-à-dire que le débordement d'un côté se retrouve de l'autre côté.
Lorsque (i + 1) déborde à droite de l'écran le reste de la division par la taille horizontale de la matrice ramène ce voisin de i à celui tout à gauche, et sans débordement ça ne change rien.
Lorsque (i – 1) déborde à gauche de la matrice il faut d'abord rajouter la taille horizontale de la matrice avant d'opérer le modulo avec la taille horizontale de la matrice, ce qui renvoie complètement à droite de l'écran. De même, ça ne change rien si (i – 1) ne sort pas de l'écran.

(4) Deuxième boucle for, imbriquée dans la première, avec cette fois les positions adjacentes verticales stockées respectivement dans ja et jb pour vers le bas et vers le haut. Le principe de contrôle est le même que pour la position horizontale, mais cette fois c'est la hauteur TY qui est utilisée.

```
(1)  void propagation(void)
     {
(2)  int i, ia, ib, j, ja, jb;

(3)        for ( i = 1; i < TX; i++ ) {
               ia = ( i+1 )%TX;
               ib = ( ( i -1 ) + TX )%TX;

(4)            for ( j = 0; j < TY; j++ ) {
                   ja = ( j +1 )%TY;
                   jb = ( ( j -1 ) + TY )%TY;

(5)                if ( P[ i ][ j ]== BOIS ) {
                       if(  ( P[ ia ][ j ] == FEU )  || ( P[ i ][ jb ] == FEU ) ||
                            ( P[ ib ][ j ] == FEU )  || ( P[ i ][ ja ] == FEU ) ){

                           NP[ i ][ j ] = FEU;
                           cmpt_feu++;
                       }
                   }
                   else if ( P[ i ][ j ] == FEU ) {
                       NP[ i ][ j ] = CENDRE;
                       cmpt_feu--;
                   }
               }
           }
(6)        for( i = 0; i < TX; i++ )
               memcpy(P[ i ], NP[ i ], sizeof( int )*TY );
     }
```

Fig. 14.17 : Processus de propagation du feu

(5) Au centre de l'imbrication des deux boucles, le test pour connaître la valeur de la position courante (i, j) si sa valeur est BOIS un second test est effectué pour connaître si une cellule voisine est sur FEU. Si oui, la cellule de la position courante passe sur FEU à la même position dans la matrice miroir NP qui accueille les résultats. Ensuite le compteur de feux est incrémenté de un.
En revanche si la position courante n'est pas sur BOIS mais sur FEU alors elle prend la valeur de CENDRE dans la matrice miroir NP et le compteur est diminué de un.

(6) Une fois que toutes les positions ont été regardées et les valeurs modifiées sauvées dans la matrice miroir NP il faut recopier les résultats obtenus de la matrice NP vers la matrice P afin de recommencer le processus avec la nouvelle configuration. La recopie est faite ligne par ligne et avec la fonction memcpy, comme en (3) sur la figure 14.16 avec la fonction « init_feu ».

Boucle du processus

La boucle du processus, à placer dans la fonction « main » après les initialisations d'usage, est la suivante (figure 14.18) :

(1) La variable globale pour le comptage des feux est initialisée à 0.

(2) Comme souvent c'est une boucle while qui se termine avec la pression de la touche échap du clavier de l'ordinateur. Dans cette boucle il y a deux tests.

```
    (…)

(1) cmpt_feu=0;
(2) while ( ! key[KEY_ESC] ){

(3)       if ( key[KEY_ENTER]){
              init_bois();
              cmpt_feu = init_feu();
          }
(4)       if(cmpt_feu){
              propagation();
              affiche_plan(page);
              blit(page,screen, 0,0,0,0, ECRAN_X,ECRAN_Y);
          }

    (…)
```

Fig. 14.18 : Mise en œuvre dans la fonction main

(3) Le premier test a pour objectif de permettre des initialisations et de relancer l'action lorsqu'un feu est fini. Il porte sur la pression de la touche « entrée ». Si oui la première fonction appelée est init_bois pour l'initialisation de la forêt. Ensuite c'est la fonction « init_feu » et le nombre de feux allumés renvoyés est affecté à la variable « cmpt_feu ».

(4) Si la variable cmpt_feu est différente de zéro, c'est-à-dire s'il y a des feux allumés, la propagation a lieu. Le calcul est fait une fois et la matrice P modifiée est affichée dans la bitmap intermédiaire « page » . C'est le rôle de la fonction « affiche_plan » (figure 14.19, détails plus bas). Enfin la bitmap « page » est affichée à l'écran. Lorsqu'il n'y a plus de feu, l'action s'arrête sur la dernière image et peut être relancée avec une nouvelle initialisation en pressant la touche entrée.

```
void affiche_plan(BITMAP *bmp)
{
int i, j;

    clear( bmp );
    for ( i = 0; i < TX; i++ )
        for ( j = 0; j < TY; j++ )
            rectfill( bmp, X( i ),Y( j ), X( i+1 ),Y( j+1 ), P[ i ][ j ] );
}
```

Fig. 14.19 : Affichage de la matrice dans une bitmap

Fonction d'affichage

La fonction « affiche_plan » prend une bitmap de référence. Toutes les positions de la matrice P sont passées en revue et chacune d'elles est représentée par un rectangle de longueur pasx et hauteur pasy aux positions « i * pasx » et « j * pasy » définies par les macros X et Y. La couleur est celle de la position. Notons qu'il n'y a besoin que de trois couleurs, celle du fond, celle des bois, celle du feu et celle des cendres. Elles vont correspondre aux indices 1, 2, 3 d'une palette à créer avant la boucle.

Evolution du croquis, idée du Tango

L'idée est de faire se rencontrer deux feux de nature opposée et complémentaire ; ce que l'un a consommé sert à rallumer l'autre et réciproquement. D'où la notion de danse, et particulièrement de danse à deux. Mais c'est actuellement juste à l'état d'intuition, comme un sujet en suspens, un projet qui méritera un jour d'être concrétisé : un « droit de rêver » selon l'expression de Gaston Bachelard.

Pour conserver cette ouverture potentielle nous avons fait quelques tests simples à partir du programme précédent. La surface de la matrice est divisée en deux territoires et un nouvel état possible est ajouté, baptisé POUSSE. Le principe des transitions est le suivant :
Si la position est sur BOIS et qu'il y a des positions voisines à FEU, mise à FEU.
Si la position est sur FEU, mise à CENDRE .
Si la position est sur CENDRE et qu'il y a des positions voisines à POUSSE mise à POUSSE.
Si la position est sur POUSSE, mise à BOIS.
Les modifications à apporter au programme sont mineures. L'initialisation partage la matrice en deux, d'un côté c'est BOIS et la mise à feu est avec FEU de l'autre c'est CENDRE et la mise à feu est avec POUSSE. La propagation traite les deux cas l'un à la suite de l'autre :

```
if (P[ i ][ j ] == BOIS) {
        …
}
else if P[ i ][ j ] == FEU{
        …
}
if (P[ i ][ j ] == CENDRE) {
        …
}
else if P[ i ][ j ] == POUSSE{
        …
}
```

Le résultat varie avec les initialisations. Parfois il y a des phénomènes ondulatoires, une mise en boucle infinie d'un ensemble de séquences. Du point de vue esthétique l'affichage joue beaucoup. Selon la forme et le nombre des cellules le résultat n'est pas le même. En particulier il peut être intéressant de travailler davantage l'animation lorsque le nombre de cellules est peu important, afin d'avoir un déplacement progressif de la forme vers ses nouvelles coordonnées plutôt qu'un saut. Il y a également le travail sur les couleurs à envisager. A tester aussi un nombre plus grand de populations de cellules reliées entre elles et une initialisation plus fine de chaque territoire…

4. Entre explosions et fantômes

Ce croquis suggère une curieuse matière un peu visqueuse qui apparaît sous la forme d'une explosion, s'écoule le long de l'écran et se stabilise en bas dans une sorte de flaque qui prend progressivement l'allure d'une barre, comme une épée de Jedaï dans la Guerre des Etoiles. Eventuellement, si l'écoulement n'arrive pas à son terme, la masse en forme de fuseau se resserre et s'efface progressivement, puis disparaît comme un fantôme.

En modifiant quelques paramètres on obtient toutes sortes de variations. Cet effet est pressenti pour le Brodock à plumeau mentionné au chapitre deux à propos de la liaison entre des idées et la conception d'un programme (le Brodock à plumeau est une sorte de bouée hérissée de poils explosifs, quelque chose comme un mollusque à réaliser dans le cadre d'une galerie de logiciels monstrueux).

4.1 Principe

A la base il y a simplement une concentration de vecteurs, tous de sens différents, mais qui, à partir d'un centre commun donnent lieu à des projections de particules et un rayonnement. Les particules projetées sont ensuite soumises à pesanteur. Elles s'alourdissent petit à petit et l'ensemble finit par s'effondrer vers le bas. Les particules rebondissent en bas, remontent et retombent jusqu'à stabilisation.

4.2 Mise en forme et initialisations

Les vecteurs sont réalisés sous la forme d'une structure qui comprend quatre floats et un entier. Les quatre floats sont pour la précision de calculs en nombre à virgule. Ils concernent les deux points de coordonnées du vecteur. Le premier comme origine (x, y) et le second comme distance qui lui est ajoutée. L'entier « couleur » stocke une couleur. Le type « VECTEUR » est défini comme suit au début du programme :

```
typedef struct {
    float x, y, dx, dy;
    int couleur;
} VECTEUR;
```

Ensuite il y a besoin d'un nombre de vecteurs. Le plus simple est d'avoir un tableau de structures VECTEUR. Il faudra pouvoir tester le programme avec des nombres différents de vecteurs, d'où la définition d'une macro, ce qui donne :

```
#define NB_VECTEURS 2000
VECTEUR      VECT[NB_VECTEURS];
```

Notons également le besoin de nombres aléatoires à virgule. Pour ce faire la macro « frandom » est définie de la façon suivante :

```
#define frandom( n )      ( ( (float)rand( )  / ( float )RAND_MAX ) * ( n ) )
```

La valeur de retour de la fonction « rand » est convertie en float par un « cast » ensuite elle est divisée par la valeur maximum que peut retourner « rand », elle-même convertie en float (la macro RAND_MAX est définie dans la librairie utilitaire <stdlib.h>). On a donc un nombre compris entre 0 et 1. Pour finir il est multiplié par « n » afin d'avoir un nombre entre 0 et « n » avec la précision d'un flottant.

Comme pour les autres programmes il y a les deux macros pour la taille de l'écran, le pointeur « BITMAP *page » et une palette « pal ».

Initialisation des vecteurs

L'initialisation des vecteurs se fait en deux étapes. La première consiste à donner des valeurs de position et de distance. La seconde est un petit calcul qui permet de se situer sur un disque de rayon aléatoire (figure 14.20) :

```
(1)  void init_explosion(int x, int y)
     {
     int i;
     float lon, lx, ly, dist;

(2)        for( i = 0; i < NB_VECTEURS; i++ ) {
                VECT[ i ].x = x;
                VECT[ i ].y = y;
                VECT[ i ].dx = frandom( 2 ) - 1;
                VECT[ i ].dy = frandom( 10 ) - 5;

 (3)            lx = VECT[ i ].dx;
                ly = VECT[ i ].dy;
                lon = sqrt( lx*lx + ly*ly );
                if( lon != 0.0 )
                     lon = 1.0 / lon;

                dist = frandom( 2.5 );
                VECT[ i ].dx *= lon*dist;
                VECT[ i ].dy *= lon*dist;
(4)             VECT[ i ].couleur = 255;
           }
     }
```

Fig. 14.20 : Initialisation des vecteurs

(1) La fonction « init_explosion » ne renvoie rien mais a deux paramètres, deux entiers. En fait c'est une position qui lui est passée. La fonction utilise quelques variables locales, un entier et quatre float.

(2) Boucle qui passe chaque vecteur en revue. Le premier point est d'attribuer des valeurs de position et de distance à la position pour chacun des vecteurs. Les vecteurs prennent tous la même position (x, y) passée en argument à la fonction. La distance est aléatoire dans la fourchette de – 1 à 1 pour dx et de – 5 à 5 pour dy.

(3) A cette étape de la fonction, par rapport au centre en (x, y), l'ensemble des vecteurs est enfermé dans un rectangle de deux unités horizontales sur 10 verticales. Ce n'est pas inintéressant mais l'objectif est ici de passer en coordonnées polaires et de se situer à l'intérieur d'un disque.
Les variables « lx » et « ly » sont utilitaires juste pour simplifier l'écriture. Tout d'abord, c'est le calcul de la longueur à partir du théorème de Pythagore. La fonction « sqrt » renvoie la racine carré de ($dx^2 + dy^2$), résultat stocké dans « lon ». Si ce résultat n'est pas nul, il est inversé et avec multiplication par dx on obtient le cosinus de l'angle et le sinus en multipliant par dy. La variable « dist » donne un rayon dans la fourchette de 0 à 2,5 et on passe en coordonnées polaires. Comme tous les vecteurs ont même origine (x, y) ils sont tous enfermés cette fois dans le disque de centre (x, y) et de rayon 2,5.

(4) En ce qui concerne la couleur nous avons opté pour que tous les vecteurs prennent la même couleur initiale, la position 255 de la palette. Le souhait est d'avoir au centre une couleur très claire et qui décroisse avec l'éloignement.

Initialisation de la palette

En ce qui concerne la palette il n'y a pas tellement de commentaires techniques à faire (figure 14.21). Elle est inspirée de celle du feu vue précédemment (figure 14.2) avec une variation : le rouge est aléatoire. Du coup, en partant du rouge vers le blanc, on obtient des verts-jaunes et de reflets de bleus très clairs à proximité du blanc.

```
(1)  void palette(void)
     {
     int i;
(2)        pal[ 0 ].r=0;
           pal[ 0 ].g=0;
           pal[ 0 ].b=0;
(3)        for ( i=1; i < 64; i++ ) {
                pal[ i ].r=rand( )%64;
                pal[ i ].g=0;
                pal[ i ].b=0;
           }
(4)        for ( i=64; i<128; i++ ) {
                pal[ i ].r=rand( )%64;
                pal[ i ].g=i - 64;
                pal[ i ].b=0;
           }
(5)        for ( i=128; i < 256; i++ ) {
                pal[ i ].r=rand( )%64;
                pal[ i ].g=63;
                pal[ i ].b=( ( i – 128 ) >> 1);
           }
(6)        set_pallete( pal );
     }
```

Fig. 14.21 : Palette, dégradé fixe pour le vert et le bleu avec valeurs aléatoires de rouge

(1) La fonction « palette » ne prend pas d'argument et ne renvoie rien.

(2) la position 0 est initialisée avec la couleur noire.

(3) Des positions 1 à 63 le rouge prend une valeur au hasard entre 0 et 63. Le vert et le bleu restent à 0.

(4) Des positions 64 à 127 toujours pareille pour le rouge, le vert augmente de 1 en un et le bleu reste à 0.

(5) Des positions 128 à 256 le rouge est toujours aléatoire, le vert est fixé à 63 et le bleu augmente de 1 en 1 par pas de deux.

Opérateur de traitement des bits : à gauche « << » et décalage à droite « >> »

Ces opérateurs << et >> décalent leur opérande de gauche du nombre de bits donné par l'opérande de droite. Par exemple soit x un octet tel que : 0000 0001.
« x<< 2 » donne 0000 0100.
Toutes les valeurs des bits sont décalées de 2 vers la gauche et les bits de l'extrémité à droite sont remplis avec des 0 ; cela revient à multiplier x par 2^2.
Dans l'autre sens « x >>2 » revient à notre position de départ, toutes les valeurs des bits sont décalées de 2 vers la droite et les valeurs de l'extémité gauche sont remplacées par des 0, ce qui est équivalent à une division par 2^2. [KERNIGHAN, 1995, p. 49] [BRAQUELAIRE, 1998, p. 120]

Ainsi l'expression i >> 1 revient à diviser i par deux et dans la fonction il faudra qu'il augmente de 2 pour avancer de 1.

4.3 Processus générateur

Boucle du processus

Commençons par la boucle du processus placée dans la fonction main du programme (figure 14.22).

```
    (…)
(1) palette ( );
    page = create_bitmap ( ECRAN_X, ECRAN_Y );
    clear ( page );
    init_explosion ( ECRAN_X/2, ECRAN_Y/2 );

(2) do {
        if ( key[ KEY_ENTER ] )
(3)         init_explosion ( rand( )%ECRAN_X, rand( )%ECRAN_Y );

(4)     bouge_vecteurs ( page );
        efface_progressif ( page );
        blit(page, screen, 0, 0, 0, 0, ECRAN_X, ECRAN_Y);

(5) } while(!key[KEY_ESC]);
    (…)
```

Fig. 14.22 : Détail de la boucle d'action au début du programme

(1) Tout d'abord il y a les initialisations des globales, à savoir la palette et la page intermédiaire du montage de l'image. Ensuite un appel à la fonction « init_explosion » avec en argument la position du centre de l'écran afin de démarrer le programme sur une explosion au centre de l'écran.

(2) Le type de boucle choisi est do-while de façon à ce qu'il y ait au moins une explosion même dans le cas ou la touche Echap aurait déjà été pressée, c'est en effet le test d'arrêt de la boucle ligne (5).

(3) A chaque pression de la touche Entrée il y a une nouvelle initialisation avec une position aléatoire dans l'écran.

(4) Après initialisation, vient le mouvement, avec l'itération d'une part de la fonction « bouge_vecteurs » qui anime les vecteurs, et d'autre part, l'effet de disparition progressive des couleurs avec la fonction « efface_progressif ». Les modifications de l'image sont faites en mémoire avec la bitmap page et pour finir l'image obtenue est plaquée à l'écran avec la fonction « blit ».

Animation des vecteurs

Il s'agit des instructions suivantes de la fonction « bouge_vecteurs » (figure 14.23) :

(1) La fonction utilise une variable « poids » ici déclarée globale afin éventuellement de pouvoir modifier sa valeur à partir d'une autre fonction dans le programme. « poids » va servir à accentuer le déplacement vers le bas de l'écran en y.

(2) « bouge_vecteurs » prend une bitmap de référence en argument et ne renvoie pas de valeur. La déclaration locale d'un pointeur de type VECTEUR* est motivée afin de soulager l'écriture. Il est initialisé avec la première adresse du tableau VECT, la première position.

(3) Tous les vecteurs sont passés en revue, à noter l'incrémentation parallèle du pointeur.

(4)(5) Pour chacun d'entre eux il y a d'abord les modifications de la position verticale et ensuite les modifications de la position horizontale en (5). Dans les deux cas la distance est ajoutée à la position courante. Vient ensuite un test sur les bords de l'écran.
Le pourtour de l'écran fait l'objet d'une marge de deux unités réservées au fonctionnement de la fonction suivante « efface_progressif ».
Tous dépassements, gauche-droite ou haut-bas, se traduisent par le ralentissement du déplacement. La valeur donnée est arbitraire, obtenue par tâtonnement, elle pourrait faire l'objet de modification à la volée.
Du point de vue vertical le déplacement vers le bas en dy est augmenté du « poids », ce qui accélère sa chute et crée un effet de pesanteur.

(6) Pour finir, le vecteur qui est visualisé par le déplacement d'un point est affiché dans la bitmap de référence selon le principe que nous avons détaillé au chapitre Onze.

```
(1)  float poids = 0.1;

(2)  void bouge_vecteurs(BITMAP *bmp)
     {
     int i;
     VECTEUR *v = VECT;

(3)       for ( i = 0; i < NB_VECTEURS;  i++, v++ ) {

(4)            v->y += v->dy;
               if ( v->y > ECRAN_Y-2 ) {
                    v->y = ECRAN_Y-2;
                    v->dy = -( v->dy ) / 1.5;
               }
               else if ( v->y < 2 ) {
                    v->y = 2;
                    v->dy = -( v->dy ) / 1.5;
               }
               else
                    v->dy += poids;

(5)            v->x += v->dx;
               if ( v->x > ECRAN_X-2 ) {
                    v->x = ECRAN_X-2;
                    v->dx = -( v->dx ) / 1.5;
               }
               else if( v->x < 2 ) {
                    v->x = 2;
                    v->dx = -( v->dx ) / 1.5;
               }

(6)            bmp->line[ ( int ) v->y ][ ( int ) v->x ] = v->couleur;
       }
     }
```

Fig. 14.23 : Déplacement et affichage des particules

Disparition progressive des couleurs

Pour chaque position de l'image, il s'agit de faire la moyenne des couleurs des positions est, nord, ouest et sud et de diminuer le résultat progressivement jusqu'à 0 (figure 14.24).

(1) « efface_progressif » ne renvoie pas de valeur et a comme paramètre unique une bitmap de référence.

(2) Pour chaque position de la bitmap passée en argument, une variable « t » stocke tout d'abord l'addition des quatre voisines et la divise par quatre. Ensuite l'expression conditionnelle décrémente t de 2 si t est supérieur à 2 et lui attribue 0 sinon.

(3) Pour finir, t qui est de type int est casté en unsigned char et affectée à la position courante (x, y).

```
(1)  void efface_progressif(BITMAP *bmp)
     {
     int x, y, t;

          for (  y = 1; y < ECRAN_Y-1; y++ )
               for ( x = 1; x < ECRAN_X-1; x++ ){

(2)            t = ( bmp->line[ y-1 ][ x ] + bmp->line[ y+1 ][ x ] +
                     bmp->line[ y ][ x-1 ] + bmp->line[ y ][ x+1 ]  ) / 4;
               t = ( t < 2 ) ? 0 : t - 2 ;
(3)            bmp->line[ y ][ x ] = ( unsigned char ) t ;
               }
}
```

Fig. 14.24 : Effet de résorption des couleurs

Chapitre 15

Histoires : le non-calculable

L'oignon n'est pas une boîte. En lui, ce qui contient s'identifie exactement avec ce qui est contenu, selon un paradoxe pelliculaire qui offre c'est sûr, une image de prédilection pour le géomètre, pour le philosophe et pour l'artiste. Dans l'oignon, en effet, l'écorce est le noyau : plus de hiérarchie possible, désormais, entre le centre et la périphérie. Une solidarité troublante, basée sur le contact – mais aussi sur d'inframinces interstices –,noue l'enveloppe et la chose enveloppée. Le dehors, ici, n'est qu'une mue du dedans.

Didi-Huberman

Expérience robographique, ouverture polymorphe

Nous sommes partis de la couleur, de sa définition et du pixel. La question de la création d'ensembles de couleurs et des palettes est venue juste ensuite, puis la définition d'une surface en mémoire avec l'étude de l'image bitmap. Couleurs et surface donnent la possibilité de tracer des formes, et d'animer ces formes, de réaliser des croquis. Cependant les figures que nous avons montrées ne sont pas des dessins, elles n'appartiennent pas à cette nature plastique. Nous avons montré du code source et des algorithmes et nous avons fait de la programmation.

L'expérience des structures de données ainsi que celle des algorithmes nous a fait sortir progressivement du champ de l'image. Ce qui a trait à l'image est devenu, dans l'espace robographique, cas particulier de types, structures de données et algorithmes. Mais si les algorithmes graphiques sont des figures non graphiques, en revanche elles se prêtent à l'imagination. Il reste l'image du point de vue de la représentation imaginaire et de l'intériorisation des mécanismes de l'écriture.

C'est pourquoi, après quelques réalisations de croquis, tout est prêt pour s'engager plus avant dans une narration plus complète, une histoire. Mais l'histoire n'est pas simplement un croquis plus développé et plus important en poids de programmation. D'ailleurs une histoire peut faire l'objet d'un programme assez léger. Ce qui différencie le simple effet de l'histoire n'est pas de l'ordre du calculable.

Ce chapitre est comme un carrefour à deux directions, elles arrivent après incorporation de l'expérience robographique. L'une ramène au début de l'ouvrage, avec les questions relatives à la notion de projet, et de modélisation. L'autre accueille la nature polymorphe de l'espace robographique et se prolonge en décuplant les possibilités d'ouverture. Entre les deux, et c'est l'objectif surtout de ce chapitre-ci, une question se pose assez directe maintenant : comment faire du non-calculable un calcul ?

Du non-calculable au calcul

Au chapitre Six nous avons proposé une propriété de programme dite de *réfraction*. La réfraction est un regard porté sur le programme en tant qu'objet indépendant, qui, pris comme phénomène, peut être considéré auprès du public comme doté de propriétés non programmées. En effet cette capacité dite de réfraction, aux prises avec l'imaginaire de chacun, n'est pas observable comme « hardware » ou « software » et à ce titre elle ne fait pas partie de l'objet informatique ni d'un savoir direct sur ces composants. Mais elle relève de ce qui est non calculable par nature dans l'opération réalisée. A savoir tout ce qui a trait au sujet et au désir, c'est-à-dire au principe même de l'engendrement, que ce soit dans l'acte de conception ou, une fois le programme conçu, sa mise en situation publique, son utilisation et son fonctionnement.

Le non-calculable désigne toutefois deux pôles. Celui en aval de l'objet qui concerne l'objet fini, et celui en amont de l'objet dont procède sa définition. Dans les deux cas nous constatons que le non-calculable, ce qui relève de l'investissement psychique de celui qui fait ou regarde, s'exprime néanmoins par un calcul. La question se pose ainsi de faire du non-calculable un calcul.

Plaisir de calculer l'incalculable

Le programme, langage semi-autonome en apparence, reste attaché à la parole qui énonce, qui prononce. Les représentations, ce ne sont pas seulement celles des données en machine, mais également celles nées de l'imagination de celui qui conçoit. C'est ce qui scelle la notion d'écriture plutôt littéraire que nous attachons au programme qui, de façon générale, raconte.

Ainsi, le programme peut être une interprétation bien réelle de quelque chose qui existe d'abord dans l'imaginaire de celui qui le réalise. Mais en même temps cette réalisation contraint l'imaginaire. Au début de la démarche de création il n'y a d'ailleurs pas grand chose de visible du point de vue d'une relation possible entre l'imaginaire et la réalisation d'un programme. Mais progressivement se façonne une sorte de perception à partir des expériences de réalisation de programmes. L'intérêt s'éveille petit à petit et se porte sur des potentiels attractifs de développement. Bientôt une sorte de relief différencie entre eux les programmes réalisés : il y en a qui plaisent plus que d'autres. Des compositions qui « fonctionnent » apparaissent. Elles fonctionnent parce qu'elles apportent du plaisir, tant pour le concepteur que pour le public, et selon la nature des relations que le concepteur souhaite entretenir avec un public.

Rationalisation a posteriori de la démarche

Ces compositions peuvent ensuite être élaborées en « germes de création ». Un germe de création est une base de code source réalisée à partir d'une idée et disponible pour être développée et interprétée. Le germe de création est comparable, dans le domaine musical, à une partition de Jazz disponible pour être jouée, interprétée, arrangée. Les germes de création sont l'objet de la constitution d'un répertoire. Mais ce n'est pas un répertoire d'œuvres finies. C'est un répertoire d'appel à réalisations d'œuvres diversifiées. C'est aussi un ensemble de repères et de références dans une perspective de développement.

Le premier point abordé explicite la démarche, qui est de l'ordre du non calculable, à travers le principe de l'oignon ainsi que la relation que cette démarche entretient avec le calcul des matériaux informationnels. Le second point est une approche méthodologique possible à travers la notion de germe de création. Il s'agit de soulever le thème de la construction et de la réalisation d'un espace de développement. Quant à la nature essentiellement polymorphe de l'espace robographique, elle fait l'objet du chapitre suivant.

1. Principe de l'oignon

1.1 La démarche est non-calculable

L'oignon présente cette particularité d'être constitué de pelures enveloppées les unes dans les autres depuis le centre jusqu'à la périphérie. Elles sont emboîtées de l'intérieur vers l'extérieur et chacune définit un espace ayant sa propre dimension qui correspond aussi à un stade de développement de l'oignon. Dans le fond il y a là un modèle de

développement dont l'action et la réflexion peuvent s'inspirer. Le principe de l'oignon est une figure que nous aurions pu ajouter en particulier auprès de la spirale finalisée du chapitre Trois (figure 3.5). Le principe de l'oignon se différencie de la spirale finalisée, dans la mesure où il n'y a pas de finalité ferme même si celle-ci peut apparaître à un moment donné.

Tension vers la réalisation

Nous retrouvons effectivement des problématiques que nous avons abordées aux chapitres Deux et Trois avec les questions de projet et de modélisation. Nous étions arrivés à la notion de connaissance définie entre action et information par la médiation du modèle. Deux grandes lignes apparaissent alors, l'une couvrant l'ouverture sur le monde, l'autre correspondant à la descente dans la réalisation d'un programme (figure 3.1). Selon cette vision nous avons ensuite proposé quelques formes et approches de la modélisation (figures 3.3, 3.4, 3.5). Chacune d'elles fait valoir la modélisation dans le processus de réalisation, à la fois comme une compétence et comme une méthodologie. C'est-à-dire que la modélisation nous apparaît comme doublée d'une manière d'être qu'il s'agisse de l'acquisition des connaissances ou de la réalisation du programme.

Par là le modèle prend l'aspect d'un chantier permanent. Cette permanence n'est pas l'impossibilité d'arriver à un résultat, elle n'est pas l'expression d'un enlisement même si celui-ci est possible. Tous les chantiers n'aboutissent pas. La permanence du chantier est soumise à une tension vers la réalisation. Cette tension vers la réalisation est l'un des trois aspects principaux que nous avons définis au chapitre Deux qui entrent dans la composition d'un projet : idée initiale, espace de travail et tension vers la réalisation.

Création évolutive

La tension vers la réalisation est ce qui pousse vers l'obtention d'un résultat, mais en même temps c'est aussi ce qui motive l'exploration de résultats différents. C'est à la fois ce qui provoque l'ouverture sur des préoccupations diverses et la spécialisation vers un objectif particulier. Ces deux tendances ne sont pas antinomiques mais au contraire s'appellent et se nourrissent l'une de l'autre : l'acte de création et de réalisation est évolutif. La maîtrise de cette évolution fonde la pratique, et réciproquement, on peut dire également que la pratique est l'expression d'une maîtrise de cette évolution.

Si l'on entendait par maîtrise, au sens fort, une sorte de toute puissance, il serait possible d'évoquer entre le processus évolutif de la création et celui de la réalisation, une certaine forme d'ambivalence. Les tendances de l'ouverture peuvent entrer en conflit avec celles de la maîtrise. Il y a une dynamique entre ces deux bords. Cette dynamique caractérise un auteur. L'engagement dans la création et la réalisation d'un programme informatique fait l'objet d'une analyse des motivations et de son fonctionnement : l'évolution c'est celle de celui qui conçoit.

Exemple du bonhomme, « La vie mode d'emploi »

A titre d'exemple mentionnons celui du bonhomme que nous avons utilisé avec la présentation d'un certain nombre de techniques du mouvement au chapitre Douze.
Au début nous faisons le programme d'un bonhomme qui marche sur place. Ensuite le bonhomme marche et se déplace. Sa trajectoire est de courir autour de l'écran. Du coup il

est comme dans une boîte, l'idée vient de le faire courir dans une boîte et d'attribuer du mouvement à cette boîte. Chaque idée est traduite par la réalisation du programme correspondant. Les programmes font l'objet d'expériences sensibles. On voit le bonhomme qui court sur place, autour de l'écran, dans sa boîte. Ces expériences entrent en résonance avec l'imaginaire. A chaque fois le cercle de la maîtrise exprimée par le programme s'élargit si l'ouverture est acceptée.

Le bonhomme, au départ simple illustration de quelques points précis de programmation, peut devenir un projet. Une suite possible est d'expérimenter la possibilité d'avoir de nombreuses boîtes avec pour chacune un bonhomme dedans. Nous avons vu précédemment le procédé des boîtes avec le déplacement de formes géométriques. Les boîtes peuvent même éventuellement se déplacer sur des images. Ensuite il peut être amusant de voir ce que donnent différentes échelles pour les bonhommes et leurs boîtes, c'est-à-dire d'avoir des petits et des grands bonhommes dans des boîtes proportionnées. De plus imaginons les bonhommes différents, une recherche graphique est par là motivée. Les questions d'avoir plusieurs bonhommes par boîte, puis un scénario de relations entre les bonhommes de boîte à boîte, peuvent être abordées. Il y a un embryon d'histoire possible. D'ailleurs il rappelle une autre histoire, le roman de George Perrec « La vie mode d'emploi ». Cette histoire lue ou relue est incorporée. Avec cette référence du roman d'autres idées vont apparaître comme des pièces d'un puzzle encore inexistant mais se révélant pourtant, et progressivement, un véritable projet va se mettre en place, il aura un point d'ancrage avec la littérature. Un domaine de connaissances propres à ce projet commence à se constituer.

1.2 Métamorphoses de l'objet avec la démarche

Par des apports successifs, la création évolue. Mais ce ne sont pas seulement les idées qui évoluent ; le matériau algorithmique lui-même doit évoluer. Notamment, et ce n'est pas toujours trivial, lorsque les structures de données fondamentales choisies pour le programme se révèlent insuffisantes pour supporter l'apport des idées nouvelles. La maturation du programme peut alors se traduire par une succession de programmes, chacun correspondant à une phase de son évolution liée à l'évolution du projet.

Afin d'illustrer ce point nous allons nous reporter au projet Kandoscope de « tableau métamorphique » évoqué à propos de la différenciation entre notre notion de projet et celle de résolution de problème au chapitre Deux.

Projet métamorphique, illustration de la démarche de création évolutive

La base du programme est l'animation de formes simples, cercles, rectangles et triangles. Chacune est soumise à une trajectoire et l'ensemble compose des figures selon le nombre des formes mises en mouvement. Tout d'abord le moteur de l'animation est paramétré de façon à réaliser des figures-types qui sont découvertes expérimentalement, par tâtonnements. Ensuite le moteur s'enrichit progressivement d'options nouvelles. Elles apportent des directions à explorer et des gammes supplémentaires de paramétrages. En quelque sorte, au fur et à mesure de l'évolution du projet, le potentiel esthétique du moteur s'élargit.
Avec l'élargissement esthétique apparaît la nécessité de refondre le programme du point de

vue des structures de données. L'idée s'approfondit et se développe à travers des programmes-fragments, et il s'agit à un moment donné de les incorporer tous dans un nouveau programme unique. Ainsi, développer le programme et développer l'idée du programme se fait simultanément de façon réciproque. Il eut été impossible de prévoir l'évolution et l'avancement du programme, et donc de rationaliser autrement son développement. L'idée se développe et le programme avec elle sur la base expérimentale des réalisations effectuées. On ne peut savoir réellement ce qu'est le programme qu'après l'avoir fait et avoir constaté un fonctionnement non seulement du point de vue de la machine, mais aussi du point de vue d'une sorte de catharsis, du ressenti, des aspects psychiques et psychologiques associés à la conception.
Dans cette perspective de création évolutive, l'idée du programme ne peut être atteinte que par cette évolution et il n'y a, dans un premier temps exploratoire, aucun autre moyen que celui d'un cheminement pour accéder aux idées.

Toutefois l'accumulation des expériences de programmes et d'idées, jointe à l'élargissement de la maîtrise des réalisations successives, produit une sorte de style propre de la démarche. Ce style se traduit au fur et à mesure par une compétence qui est comme l'intuition de programmes à faire et de comment les faire. C'est-à-dire que la probabilité d'avoir des idées de programmes réalisables avec une moindre exploration augmente avec une certaine forme de maturité qui provient de la pratique. Mais, ce que nous entendons par pratique, précisément, réclame toujours sa part d'exploration. C'est ce dont témoignent un intérêt renouvelé régulièrement pour des figures nouvelles et le principe de l'évolution qui est imprédictible par définition.

Historique de l'évolution du Kandoscope

Le développement du Kandoscope a été momentanément interrompu. Toutefois trois moteurs ont été réalisés. Ils expriment chacun un chapitre constitué de plusieurs étapes dans la recherche et la conception.

Phase 1

La première phase est aussi celle de la mise en place du premier moteur baptisé « flux ». L'idée initiale est de tester des courbes de Bézier pour la réalisation de trajectoires d'objets. Le moteur réalisé s'avère posséder trois niveaux d'expérimentations possibles (figure 15.1).

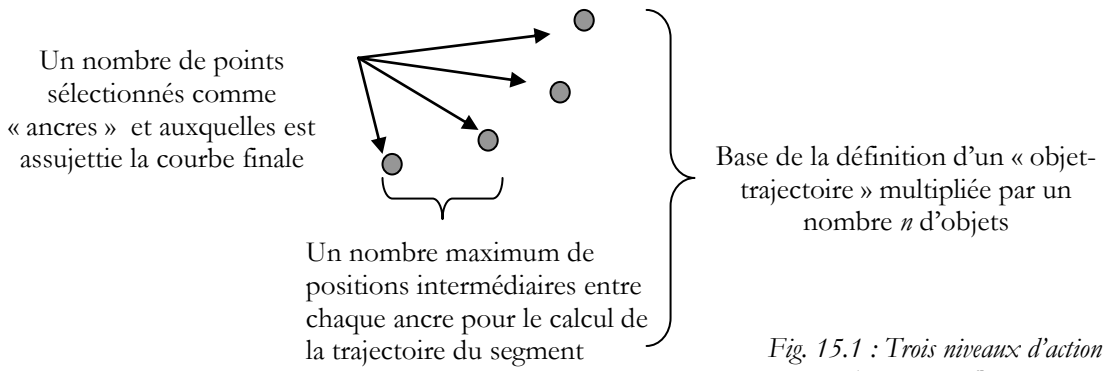


Fig. 15.1 : Trois niveaux d'action du moteur « flux »

Pour une trajectoire il y a d'abord un ensemble d' « ancres » qui sont des positions dans l'écran. Entre chacune de ces positions est opéré un calcul de positions intermédiaires via le principe des courbes de Bézier. On obtient des différences avec le contrôle du nombre des ancres et celui du nombre des positions intermédiaires. La trajectoire est visualisée avec l'animation d'une petite pastille de couleur qui suit de façon permanente la courbe résultante.

En multipliant le nombre des « objets-trajectoires » actifs simultanément on obtient un mouvement d'ensemble surprenant, comme une sorte de scintillement et selon la palette des couleurs (figure 15.2).

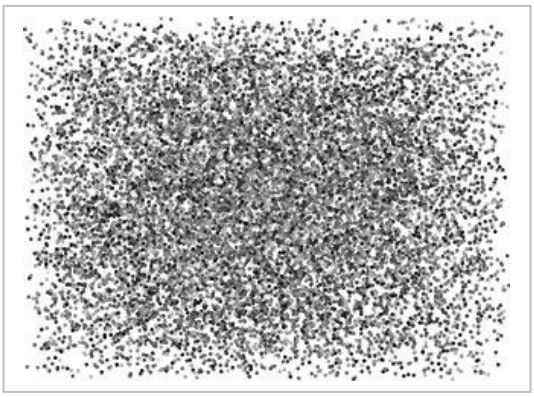


Fig 15.2 : Jouer sur la quantité des objets

Phase 2

Après l'expérimentation d'une masse d'objets représentés par des pastilles de couleurs, soumis à trajectoire fixe et activés simultanément, il y a l'idée de varier les formes et les tailles et d'explorer des ensembles moins nombreux d'objets.

Les objets sont alors dotés de caractéristiques sonores, et avec elles il y a l'introduction du temps. Les trajectoires sont réduites à de simples allées et venues d'un point à un autre (figure 15.3).

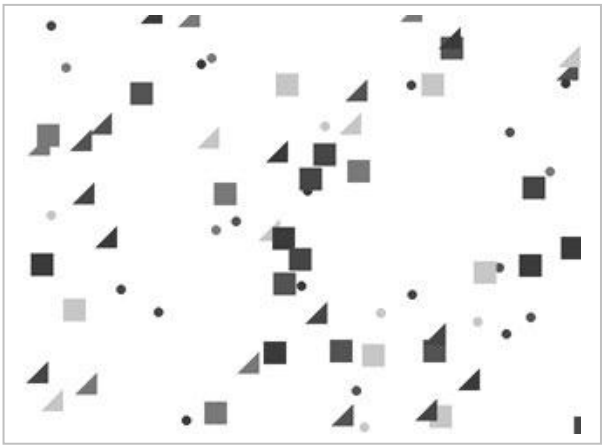


Fig. 15.3 : Paramètres de son, de temps et de formes d'objet

Phase 3

Vient ensuite un approfondissement de la question du temps, en relation avec l'élaboration de la figure d'ensemble. Chaque objet possède une horloge interne sur laquelle se cale le déclenchement de comportements.

La forme donnée dans le temps passe par l'imagination de micro-scénarii, volontairement définis, des mouvements des objets, comme des mouvements musicaux (figure 15.4).

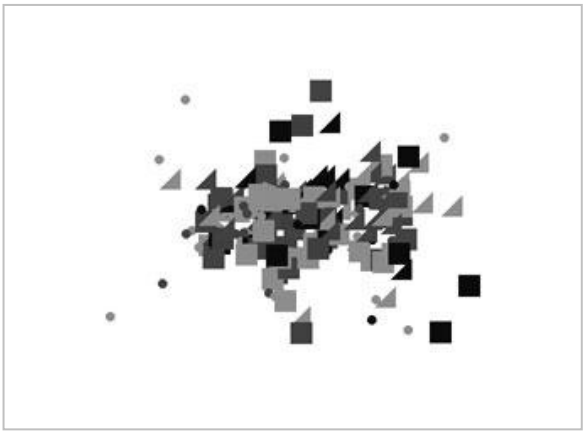


Fig. 15.4 : Paramètres de temps et de formes du temps

Phase 4

Toujours avec le même moteur, il y a réalisation d'un appareil à faire des bulles, comme celui des enfants avec de l'eau savonnée. C'est l'étude de chorégraphies de mouvements d'objets, de la forme des ensembles.

Les paramètres sont ceux de positions initiales, de nombre d'objets, de nombre d'ancres, de formes des objets, de tailles des objets et de temps séquentiel propre à chaque objet (figure 15.5).

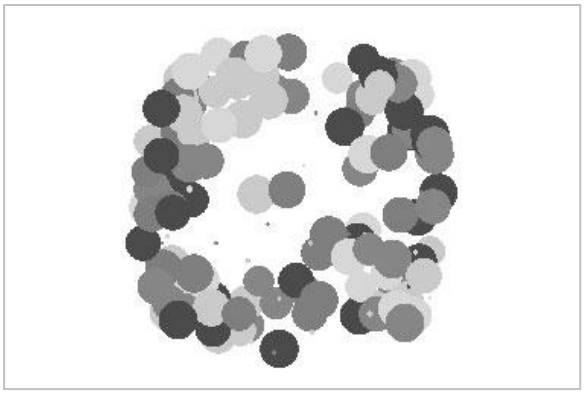


Fig. 15.5 : Paramétrage qui permet d'obtenir des bulles rondes et des bulles carrées ou triangulaires

Phase 5

Toujours avec le moteur « flux », réalisation d'une sorte de soleil.

Après l'étude des mouvements d'objets en nombre restreint, contraints à des initialisations particulières, retour aux grands nombres d'objets de petites tailles et apparition de nouvelles figures d'ensemble.

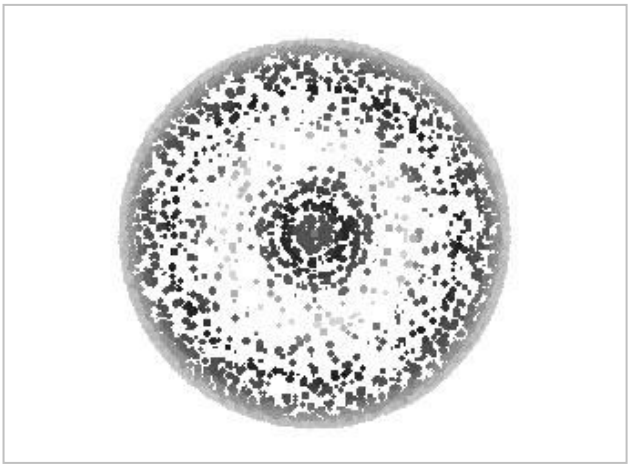


Fig. 15.6 : Paramètres de nombre d'objet, de tailles, de nombre d'ancres et d'initialisation.

Phase 6

A cette phase, la multiplicité des expérimentations pousse à reconstituer le moteur originel. Il s'agit d'adapter les structures de données pour un éventail plus large de possibilités. Ces possibilités nouvelles font partie de l'imaginaire développé avec le programme au fur et à mesure des expérimentations du moteur initial.

La principale modification est de faire de chaque trajectoire un objet graphique autonome, visualisable. Les trajectoires ne sont plus réduites à la notion de trajet pour des formes. Il est possible de les voir indépendamment des formes qui les parcourent.

Les trajectoires sont représentées comme des lignes de métro. Chaque ancre est une station et correspond à un rond affublé de son numéro d'ordre. Le type d'objet précédent, une forme soumise à une trajectoire, peut néanmoins être présent, conjugué avec sa trajectoire. Son mouvement propre est alors mis en évidence compte-tenu du calcul, en courbe de Bézier, des positions intermédiaires entre deux ancres (figure 15.7).

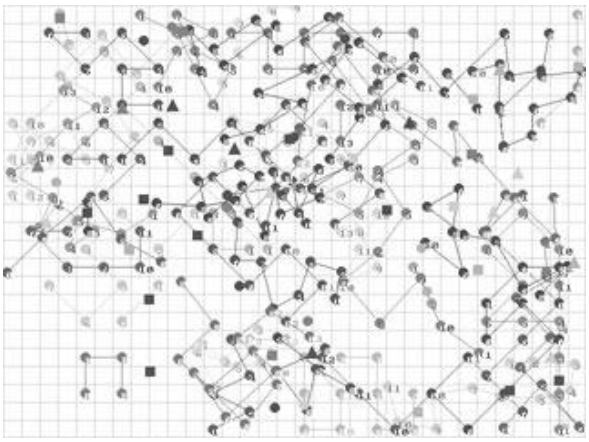


Fig. 15.7 : Apparition d'un nouveau type d'objet à travers la visualisation, pour chaque trajectoire, de l'ensemble des relais reliés les uns aux autres

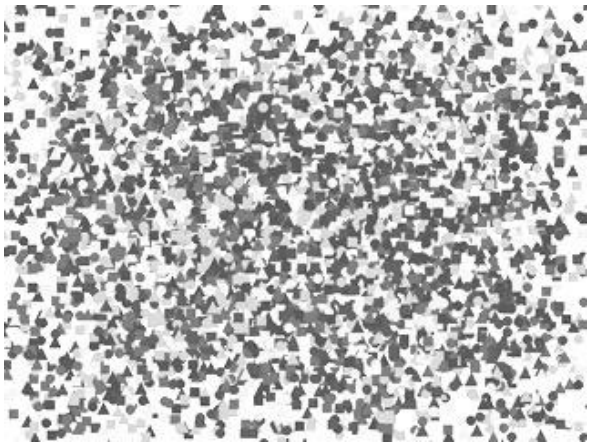


Fig. 15.8 : Exploration de la seconde génération du moteur

Phase 7

Le nouveau moteur incorpore le précédent. Tout ce que le précédent pouvait faire, en principe le nouveau peut le faire. D'où l'exploration de figures réalisées avec le précédent moteur, en particulier sur un grand nombre d'objets-formes soumis à trajectoire. Mais, c'est étonnant, le résultat n'est pas le même. L'effet produit ressemble à du vent qui soufflerait sur des feuilles innombrables, avec quelque chose d'aquatique dans la souplesse du mouvement. En réalité ce qui est différent c'est la démarche exploratoire, ainsi que l'interface des variables du moteur qui n'est pas neutre du point de vue des manipulations. Bien entendu, on obtient exactement le même résultat qu'avec le moteur précédent si le paramétrage du moteur est équivalent. C'est la démarche pour paramétrer qui est différente avec des conséquences sur les explorations qui sont effectuées (figure 15.8).

Phase 8

Apparition de la troisième génération du moteur maintenant nommé « métro ». Avec lui encore une nouvelle espèce d'objets graphiques. Cette espèce est le détachement de la vision, à la phase 6, des trajectoires en tant que lignes fixes. Ces lignes ont maintenant la possibilité de s'animer, elle acquièrent de l'autonomie. La forme est de taille variable et soumise à trajectoire. Elle peut ou non accompagner la ligne. Les lignes finalement se transforment en machines « pédalo-arachno-mécaniques ». Chaque ligne est enfermée dans un espace clos en forme de cercle ou de rectangle et dans cet espace chaque station de la trajectoire se déplace et rebondit sur les bords de l'espace. L'espace est lui-même en mouvement. Le nombre de lignes simultanées est variable. Quelques stations peuvent être fixes et les traits qui unissent les stations sont élastiques, éventuellement escamotables, ainsi d'ailleurs que les numéros d'ordre (figure 15.9).

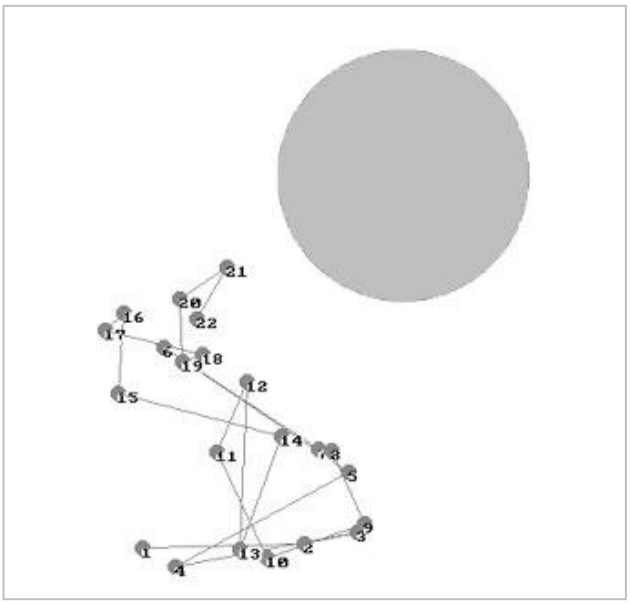


Fig. 15.9 : Les machines pédalo-arachno-mécaniques

Phase 9

L'exploration du moteur « métro » conduit à scinder le fonctionnement du programme en plusieurs registres d'activités superposées. C'est-à-dire à superposer des plans d'activité des différentes espèces d'objets. Il y a les formes soumises à trajectoires, les trajectoires fixes et les machines pédalo-arachno-mécaniques. De plus des modes graphiques d'affichage différents peuvent également être superposés. Il y a des formes qui courent sur des images et d'autres qui sont de couleur. Ainsi peut-on faire fonctionner ensemble des populations de petites formes à trajectoire simplifiées avec des formes plus grosses de couleurs ou d'images et des machines pédalo-arachno-mécaniques. Cependant il n'y a pas d'interaction comportementale entre les espèces visuelles (figure 15.10)

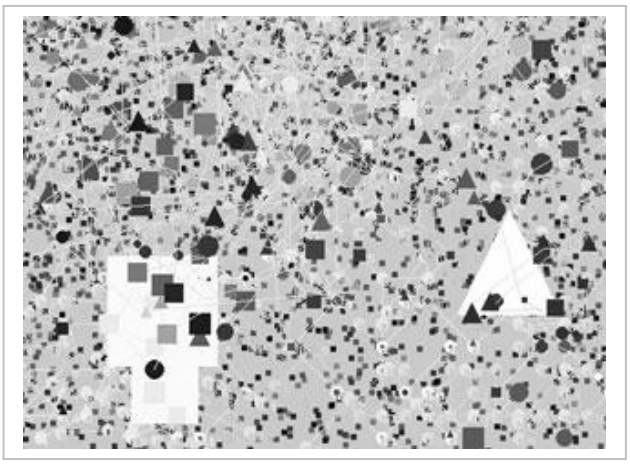


Fig. 15.10 : Simultanéité des espèces de formes sans interaction comportementale

1.3 Résistance à propos des matériaux algorithmiques

Les expérimentations et le projet du Kandoscope sont finalement momentanément immobilisées à cause d'un débordement du point de vue de la gestion des innombrables paramètres du programme. Le programme a été pensé au fur et à mesure par séquences et les moteurs fonctionnent bien mais seulement une fois qu'ils sont paramétrés. Ils ne sont pas équipés pour générer des paramétrages. Sans entrer dans le détail du code, mais à titre indicatif, afin d'illustrer le nombre des paramètres, la figure 15.11 présente la fonction d'initialisation d'un modèle de paramétrage pour un fonctionnement du programme.

```
    void modele0( MODELE*cm ){

(1)     construct_moteur ( cm, 0, NON, OUI, 0, TRAJ_MAX, 0,  0, 0, 20, NON,  NON
);

(2)     construct_aff_metros_immobiles( cm->m, DRAW_MODE_COPY_PATTERN,
                                                A[PAT]      );

(3)     construct_rls_metro(cm->m, 32, ALTERNE_FORK(cheval,roi,OUI),
ALTERNE-1,                              ALTERNE, A[PAT],
DRAW_MODE_COPY_PATTERN,
                            ALTERNE, 1, 1, 1,  1, 20, 20, 0, 0, 0, 0, 0, NON, NON,
NON,                            NON, NON, NON, NON, NON );

(4)     construct_traj_objet( cm->m, NULL, DRAW_MODE_SOLID, OBJ_ALT,
```

Fig.15.11 : Complexité du paramétrage née d'une surabondance de variables

La fonction « model0 » détermine un état particulier du programme selon des affectations de valeurs choisies à un ensemble fixé de paramètres. Il y a en tout 55 variables réparties sur quatre fonctions distinctes. La première concerne des paramètres généraux du moteur sur les différents secteurs d'activité, chacun disposant ensuite de ses propres structures de données (essentiellement des structures et des tableaux). La seconde concerne les paramètres des métros immobiles découverts à la phase 6 de l 'évolution du programme. La troisième concerne les machines pédalo-arachno-mécaniques de la phase 8 et la quatrième les formes soumises à trajectoire qui sont à l'origine du programme. Une fois le programme affublé de toutes ces valeurs il tourne en fonction d'elles mais sans possibilité d'évolution.

Effectivement l'attention a surtout porté sur la définition des bonnes structures de données pour l'obtention des figures imaginées. C'est en ce sens que la bonne organisation interne du programme, a été recherchée. La transversalité du fonctionnement, le processus métamorphique proprement dit, correspond à l'étape à venir. Il faut d'abord réduire la lourdeur obtenue du fait des nombreuses possibilités de figures, c'est-à-dire également assurer une bonne maîtrise des différents plans de fonctionnement. Cette lourdeur ralentit les expérimentations rendues de plus en plus compliquées. Peut-être que la mise en place algorithmique nécessiterait de tester un langage de programmation objet, afin d'utiliser un certain nombre des possibilités du C++ par exemple. A moins qu'il ne s'agisse tout simplement de la difficulté inhérente à la définition d'une pensée évolutive.

Le calcul de la beauté

Il y a, c'est évident, un désir esthétique, c'est-à-dire un plaisir complexe assez particulier, que nous appelons beauté, et qui est recherché. Dans le cadre de la réalisation d'un programme c'est la calcul de la beauté qu'il faut prendre en compte. Evidement, nous nous heurtons d'une part au problème de calculer le non-calculable et d'autre part à la définition du désir lui-même, tel par exemple qu'il recherche ce que d'une façon ou d'une autre il connaît déjà (comment dès lors apprécier une nouveauté ?).

Dans le Kandoscope c'est surtout la question des métamorphoses des figures d'ensemble qui pose aujourd'hui problème. La beauté des métamorphoses ne se laisse pas facilement approcher. Il y a une grande résistance du matériau algorithmique qui ressort avec un trop grand nombre de paramètres à activer. On s'y perd, d'autant qu'il est impossible de se représenter un résultat sans en faire d'abord l'expérience. Comment imaginer la continuité d'une métamorphose réalisée à travers une discontinuité d'étapes inconnues ? Et surtout il faut que le programme, dont l'identité est fixée, ne se déroule pas exactement de la même façon à chaque nouveau lancement. Il ne s'agit pas de faire un film d'animation.

Mais il serait injuste d'attribuer à la seule matière algorithmique et ses difficultés associées, tout le poids de la résistance. La résistance est également de l'ordre du psychologique. En effet la tendance est souvent de rechercher le plaisir que l'on connaît déjà, ce qui se traduit éventuellement par un refus inconscient de ce que cette matière algorithmique peut produire de surprenant. La notion de beauté ou de plaisir ressentis personnellement font ainsi l'objet d'une évolution : comment pourrait-on apprécier quelque chose qu'on ne connaît pas ? Il faut bien développer une perception de quelque chose à propos duquel on éprouve de l'intérêt et accepter le renouvellement de l'ouverture qui permet l'évolution.

1.4 Compositions « fonctionnantes »

Nous avons mentionné la notion de « formes fonctionnantes » au chapitre Six avec une référence à Paul Klee pour lequel le tableau *fonctionne* et dispose d'une autonomie propre. Nous disons de même pour le programme qui a une identité et génère une cohérence.

Le principe de Pinocchio

A un moment donné le programme est détaché du concepteur. On est tenté parfois de penser qu'il se détache de lui-même comme un être accompli. C'est le principe de Pinocchio. Ce détachement entre le programme et le concepteur fait apparaître la conception comme une sorte de dialogue à trois bords. Il y a le dialogue de soi à soi, mais également de soi à l'intuition d'autrui auquel le programme s'adresse en partie, et tout l'échange de soi aux limites de la matière informationnelle, comme il est reconnu qu'il existe un échange entre un musicien instrumentiste et son instrument. Cette matière résiste. Elle n'est pas facile à maîtriser et le travail prend la forme d'un dialogue avec la difficulté. Un équilibre est toujours à trouver entre ces composantes. Le principe de Pinocchio qui coïncide avec l'idée d'une autonomie du programme est défini par cet équilibre. C'est lorsqu'un équilibre est trouvé que le programme « fonctionne ». Il suscite alors des réactions diverses auprès du public. Son développement est considéré comme terminé. C'est le cas des programmes que nous avons présentés au chapitre Sept.

Cette fin du programme ne signifie pas que le projet soit satisfait du point de vue du désir qu'il recouvre. D'ailleurs, soulignons que les « erreurs » ou maladresses qui entrent dans le résultat final contribuent aussi au cachet général du programme du point de vue de son autonomie. En fait devant un programme, de la nature de ceux auxquels nous nous intéressons, il n'y a pas d'a priori sur ce qu'il serait mieux de faire, il n'y a pas d'a priori ni de règle sur la notion de progrès à faire. Les progrès sont ceux de la démarche, compte-tenu des connaissances dont dispose le concepteur ou qu'il décide d'acquérir. L'enjeu n'est pas un enjeu de puissance et souvent le choix esthétique est dicté par la recherche d'une intelligence dans les relations entre les formes et les objets imaginés. Dans ce domaine d'une créativité associée à la programmation informatique, faut-il le souligner, la puissance de l'ordinateur ne garantit ni la qualité des propositions, ni l'intérêt que l'on peut éprouver pour elles.

2. Germes de création

Nous avons vu que ce qui est non-calculable c'est la démarche elle-même. De ce fait, l'histoire, vécue à travers l'élaboration du modèle, est caractérisée par une permanence du commencement. En quelque sorte, raconter une histoire revient à raconter son commencement ; d'où le principe des germes de création. Le germe de création c'est un commencement toujours possible.

2.1 Point de départ exploratoire

Comme des standards de jazz

Nous avons évoqué les germes de création au chapitre Trois avec cette idée du modèle, objet permanent d'une interprétation et d'une élaboration. A partir de modèles qui existent, une extrapolation est toujours possible. Il est toujours possible de reprendre à son compte une dynamique de développement d'un modèle. Nous avons comparé cette attitude à la pratique, dans le domaine du jazz, de l'interprétation de « standards ». Les standards sont les morceaux d'un répertoire, des références qui se sont constituées avec le jazz lui-même. Les interprétations sont innombrables et les standards deviennent quasi méconnaissables parfois. Les standards de jazz peuvent être utilisés comme des points de départ exploratoires dans la musique.

Synthèse minimaliste d'un modèle

Le modèle, associé à un langage de programmation, est un point de départ exploratoire et le germe de création est la version minimaliste codée d'un modèle de quelque chose. C'est en quelque sorte l'expression d'une synthèse épurée du modèle initial. Cette synthèse est restituée sous la forme d'un squelette algorithmique codé dans un langage de programmation, qui consigne les opérations essentielles à mettre en œuvre pour le fonctionnement de l'idée.

Clef d'accès et de transversalité

L'identification et la réalisation de ces germes constituent des repères. L'ensemble de ces repères établit un domaine de connaissances et de références pour la créativité d'une personne et son expression individuelle à travers la programmation informatique.

Le modèle est toujours le modèle de quelque chose c'est-à-dire qu'il renvoie aux connaissances propres de ce qui fait l'objet du modèle. En ce sens le germe de création est une clef d'accès pour des domaines de connaissances existant, et ils peuvent être très divers (mathématiques, art, vie artificielle, etc.). En réalité il n'y a pas de limite aux sujets qu'il est possible d'aborder et la programmation peut se conjuguer avec les connaissances en général.

Le germe de création peut initialement appartenir à un domaine de connaissances et être transposé ou transporté dans un autre. Par exemple l'automate cellulaire est une figure que l'on peut envisager de multiples façons du point de vue de la connaissance en général (géographie, art, mathématiques, informatique etc.). Ainsi le germe de création n'est pas seulement une clef d'accès pour le domaine dont il est issu, mais il est comme une sorte de pollen appelé à se transporter dans différentes zones de préoccupations. Une dynamique de recherche peut lui être associée.

2.2 Plusieurs niveaux d'approche au regard de la programmation

Du point de vue de la programmation, la notion de germe de création peut être envisagée sur plusieurs plans. Nous en avons repéré quatre qui sont : les plans de l'intériorisation des figures de la programmation, l'existence de paradigmes utilitaires et créatifs qui interviennent dans l'organisation pratique, et, déjà mentionnés, les apports extérieurs de connaissances et d'idées.

Intériorisation de figures de programmation

Au plan de l'intériorisation sont désignés particulièrement les types et structures de données comme fondements scénaristiques associés à la compétence de les réinterpréter toujours dans des situation nouvelles. *Arbres dans… graphes pour…* c'est un apprentissage qui passe par l'étude des types et structures de données corrélés avec un traitement spécifique. C'est la lecture et l'expérimentation de code source avec l'acquisition progressive de l'art et de la manière de réduire toutes sortes d'idées à l'emploi nécessaire des figures de base fondamentales de l'informatique.

Paradigmes utilitaires et créatifs

Au plan pratique le germe renvoie à des « paradigmes » en programmation. Les paradigmes désignent des séquences typiques de code que l'on rencontre pratiquement dans tous les programmes ou qui sont communes à une catégorie de programmes. Ce sont des actions caractéristiques comme ouvrir un fichier, sauvegarder, imprimer, créer une fenêtre etc. Les paradigmes sont généralement solidaires d'un environnement de développement dont c'est la raison d'être de les répertorier, compte-tenu du projet global de programmation dans lequel ils s'inscrivent. Par exemple la librairie Allegro est un environnement de

programmation initialement dédié à la programmation de jeux. Elle fournit donc toutes les fonctions de base répertoriées comme nécessaires dans ce domaine, qu'il s'agisse du son, du graphisme, des questions d'interface etc..
En fait on pourrait distinguer deux aspects dans les paradigmes. Le premier est simplement utilitaire, considéré comme nécessaire, et sans intervention sérieuse au niveau scénaristique. Le second est en relation avec une gamme de scénarii possibles. Dans un jeu par exemple il est souvent question de détecter des collisions d'objets divers. La détection de collisions peut donc faire l'objet d'un paradigme en quelque sorte spécialisé au niveau scénaristique.
Toujours dans les jeux il y a parfois le souhait de disposer de modèles réalistes dans des situations typiques, des modèles de physique peuvent être sollicités. C'est-à-dire que le paradigme n'est pas seulement une situation typique de programmation mais également renvoie à une situation typique dans l'approche du domaine et des connaissances associées. Les paradigmes peuvent ainsi être évalués et classés en fonction de leurs pertinences du point de vue de la créativité et des connaissances du domaine concerné. Eventuellement, au plan de la créativité, le paradigme retourne la question « Comment ça pourrait s'intégrer dans… ? ». A ce titre il trouve sa place dans les réalisations, et nous le considérons éventuellement comme un élément de vocabulaire algorithmique.

Apports extérieurs de connaissances

Au plan des apports extérieurs de connaissances le germe de création est, comme nous l'avons indiqué, une clef d'accès et de transversalité pour le domaine des connaissances dont il est issu et ceux vers lesquels la curiosité peut conduire. Sur le registre pratique où nous sommes, il convient d'ajouter tout ce qui relève des effets et des croquis. Les exemples de code accessibles à l'appropriation et l'interprétation sont des apports extérieurs de connaissances qui contribuent à l 'élaboration d'un vocabulaire algorithmique.

Idées et références

Il reste le plan plus personnel des idées. Il rejoint la dimension exploratoire du germe, qu'il s'agisse de l'élaboration d'un germe, par exemple à partir d'un croquis, ou de l'extrapolation d'un germe existant, pioché dans une bibliothèque de germes. Cette bibliothèque est à constituer progressivement par l'auteur lui-même. C'est son fond de références conceptuelles lié à la programmation et édifié au fur et à mesure de ses recherches.

3 Exemple de racine automate

Le germe de création récapitule un ensemble de connaissances exprimées avec des structures de données claires, dans un code source minimaliste qui fonctionne. A titre indicatif en voici un qui a été particulièrement important dans notre démarche de recherche initiale. C'est le principe d'automate cellulaire à la racine de plusieurs programmes : Dialectic, présenté au chapitre Un, Graphitos et Graphitor présentés au

chapitre Sept. Il est également utilisé dans La Vénus du mythe des races métalliques présenté au chapitre Deux, à titre expérimental, comme générateur de nombres imprédictibles, quoique non aléatoires, pour le déplacement des êtres du programme.

3.1 Principe

Il s'agit de faire évoluer un plan, à partir de l'application de lois de transition à toutes les positions qui le constituent. Au commencement les position font l'objet d'une initialisation. Ensuite en fonction des lois de transitions les valeurs de chaque position évoluent et l'ensemble du plan avec elles. Le plan est visualisable à chaque étape de ce développement.

3.2 Mise en forme et initialisation

Le plan est défini par une matrice d'entiers. Une seconde matrice-miroir est là pour le calcul à chaque étape de temps. Ces matrices, déclarées en globales ont la même taille définie par deux macros TX et TY. Ces macros sont éventuellement relatives à la taille de l'écran, elle-même définie par deux macros. L'affichage du plan représenté par les matrices prend la forme d'un carrelage. Deux macros sont définies par le rapport entre la taille de l'écran et la taille des matrices de sorte que chaque carreau a « pasx » sur « pasy » de côté. Pour trouver la position d'un carreau à partir d'une position dans la matrice il suffit de multiplier par pasx pour l'horizontale et pasy pour la verticale. Afin d'alléger l'écriture deux macros sont définies pour cette opération (figure 15.12).

```
#define ECRAN_X  800
#define ECRAN_Y  600

#define TX      (ECRAN_X/2)
#define TY      (ECRAN_Y/2)
#define pasx    ECRAN_X/TX
#define pasy    ECRAN_Y/TY
#define X(n)    (n)*pasx
#define Y(n)    (n)*pasy

int MAT[TX][TY];
int SAV[TX][TY];
```

Fig. 15.12 : Déclarations au début du programme

La fonction d'initialisation minimale consiste premièrement à s'assurer que toutes les positions de la matrice initiale « MAT » sont à 0 puis à mettre quelques positions sur la valeur un. Nous avons choisi un petit carré au centre de la matrice, connaissant l'efficacité de cette initialisation du point de vue visuel (figure 15.13).

```
(1)  void init (void)
     {
(2)      memset(MAT[ 0 ], 0, sizeof(int)*TX*TY);

(3)      MAT[TX/2][TY/2]=1;
         MAT[(TX/2) +1][TY/2]=1;
         MAT[TX/2][(TY/2) +1]=1;
         MAT[(TX/2) +1][(TY/2) +1]=1;
     }
```

Fig 15.13 : Fonction d'initialisation

(1) La fonction « init » ne prend pas d'argument et ne renvoie rien.

(2) Utilisation de la fonction « memset » de la librairie standart <string.h> qui permet, avec p1, p2, p3 pour ses arguments, de mettre p3 fois la valeur p2 à partir de l'adresse p1. En l'occurrence toute la matrice d'entiers « MAT » est remplie de 0.

(3) Au centre de la matrice MAT quatre positions sont mises à 1.

3.3 Processus générateur

Après l'initialisation la boucle du déroulement est constituée de trois étapes. En premier il y a le calcul du nouvel état de la matrice. Les résultats de ce calcul sont sauvegardés dans la matrice intermédiaire SAV. Juste après le calcul a lieu l'affichage des nouveaux résultats. Pour finir il y a la recopie des résultats obtenus de la matrice SAV dans la matrice MAT. Dans la fonction main cela donne la portion de code suivant (figure 15.14) :

Fig. 15.14 : Boucle du processus dans la fonction main du programme.

```
    (...)
(1) bmp = create_bitmap(ECRAN_X,ECRAN_Y);
    degrade_variable(pal);

(2) init( );
    while( ! key[KEY_ESC]) {
        calcul();
        affiche(bmp);
        copie();
    }
```

(1) En ce qui concerne l'affichage nous avons opté pour un double buffering d'où, dans la fonction main, la création d'une bitmap à la taille de l'écran qui va servir pour constituer l'image finale avant plaquage en une seule fois à l'écran. Ensuite création d'une palette de couleurs avec la fonction « degrade_variable » vue au chapitre Dix.

(2) Appel de la fonction « init » avant entrée dans la boucle. Le test d'arrêt de la boucle repose sur la pression de la touche Echap. Ensuite, comme indiqué précédemment, appels des trois fonctions « calcul », « affiche » et « copie ».

Détail de la fonction de calcul

La fonction « calcul » est presque identique à celle de la figure 7.11 (a) abordée au chapitre Sept avec Graphitos.

Dans la boucle la première action est le calcul de la nouvelle matrice en fonction de l'initialisation et des lois de transition (figure 15.15) :

(1) Dans le corps de la double boucle qui permet d'accéder à chaque position des matrices MAT et SAV, la première fonction appelée est la fonction « compte » qui renvoie le nombre de positions à 1 trouvées autour de la position courante. Cette fonction est identique à celle de la figure 7.11 (b) si ce n'est que la matrice nommée « INITIAL » est ici

nommée « MAT ». Il s'agit juste d'une série de tests sur les huit positions voisines du type :

```
if (MAT[ i+1 ][ j ] ==1)
        comptevoisin++;
```

Les tests sur les positions voisines sont effectués à partir de la position courante (i , j) passée en argument. Après revue des huit positions voisines la fonction « compte » renvoie la valeur de la variable « comptevoisin ».

```
    void calcul(void)
    {
    int i, j;
    int voisin;

        for ( i = 1; i < TX - 1; i++ )
            for ( j = 1; j < TY - 1; j++ ){
(1)             voisin = compte( i , j );

(2)             if ( voisin < 2 || voisin > 3 )
                    SAV[ i ][ j ] = 0;
                else
                    SAV[ i ][ j ] = 1;
            }
    }
```

Fig. 15.15 (a) : La loi de transition, calcul de la nouvelle matrice

```
if ( MAT[ i ][ j ] == 1 ){
    if ( voisin < 2 || voisin > 3 )
        SAV[ i ][ j ] = 0;
    else
        SAV[ i ][ j ] = 1;
}
if ( MAT[ i ][ j ] == 0 ){
    if (  voisin !=0 &&
        ( voisin < 2 || voisin > 3 ) )
        SAV[ i ][ j ] = 1;
    else
        SAV[ i ][ j ] = 0;
}
```

(b) : une modification de la loi de transition

(2) Une fois connu le nombre des positions voisines sur 1, arrive le test pour la transition. Sur la figure 15.15(a) il est très simple : si le nombre de voisins est inférieur à 2 ou supérieur à 3 la position courante prend la valeur 0 dans la matrice SAV du résultat et sinon elle prend la valeur 1.
Une variante est indiquée en (b) quasi identique à celle de la figure 7.11(a) du programme Graphitos. Dans le cadre d'un germe de création elle donne une piste pour l'élargissement du potentiel des transitions dans le programme.

Détail de la fonction d'affichage

L'affichage va consister à passer toutes les positions en revue et de les colorier lorsqu'elles sont à un (figure 15.16) :

(1) La fonction « affiche » prend un pointeur BITMAP* comme paramètre. C'est la bitmap de référence utilisée en mémoire avant l'affichage à l'écran. Nous avons bloqué la couleur avec une variable locale arbitrairement affectée de la valeur 150. En fait il y a beaucoup a imaginer du point de vue des couleurs. C'est juste pour souligner ce point que nous avons utilisé une variable plutôt que de passer 150 en dur à la fonction de dessin.

(2) S'il n'y a pas eu de changement après le calcul que la fonction « calcul » vient d'effectuer, il n'y a pas lieu non plus de redessiner la position. Le mot clé « continue » est

une instruction qui permet de passer immédiatement à la position suivante.

(3) S'il y a eu changement et que SAV est sur 1 un rectangle est tracé à la position courante, moyennant l'utilisation des macros X et Y. Lorsque SAV est sur 0 la couleur de la position courante prend la couleur du fond.

Fig. 15.16 : L'affichage

```
(1)  void affiche(BITMAP *bmp)
     {
     int i, j;
     int couleur = 150;

          for ( i = 0; i < TX; i++ )
               for ( j = 0; j < TY; j++ ){
(2)                 if ( SAV[ i ][ j ] == MAT[ i ][ j ])
                         continue;

(3)                 if ( SAV[ i ][ j ] == 1 )
                         rectfill ( bmp, X( i ), Y( j ), X( i+1 ), Y( j+1 ), couleur );
                    else
                         rectfill ( bmp, X( i ), Y( j ), X( i+1 ), Y( j+1 ), 0 );
               }
(4)       blit ( bmp, screen, 0, 0, 0, 0, ECRAN_X, ECRAN_Y );
     }
```

(4) Lorsque l'image est prête elle est plaquée à l'écran via un appel à la fonction « blit ».

La fonction de recopie

Pour finir il ne reste plus qu'à recopier la matrice SAV dans la matrice MAT afin de pouvoir reprendre le calcul à partir ces nouvelles valeurs.

Fig. 15.17 : Recopier les valeurs résultantes

```
void copie(void)
{
        memcpy( MAT[ 0 ],SAV[ 0 ], sizeof( int )*TX*TY );
}
```

Il suffit d'un appel à la fonction « memcpy » de la librairie <string.h>.

Chapitre 16

Polymorphisme robographique

Ouverture polymorphe née de l'expérience robographique

L'expérience robographique nous sort du cadre de l'image seule, mais, par le biais de l'imagination, nous confronte à des problématiques d'écriture, tant du point de vue de l'intériorisation des mécanismes de la programmation que des investissements narratifs. L'expérience robographique pose la question de faire du non calculable un calcul et elle apparaît finalement indissociable d'une démarche personnalisée. Cette démarche personnalisée se rationalise et se construit à travers la découverte et la mise en place de repères méthodologiques ; nous avons suggéré le principe des germes de création qui permettent d'arrimer un espace de développement et de le rationaliser selon une dynamique de créativité et de recherche. Les germes de création peuvent théoriquement être ouverts sur tous les domaines et disciplines. Ce potentiel en fait des clefs d'accès et de transversalité. L'objectif de ce chapitre-ci est de préciser la nature essentiellement polymorphe de l'espace robographique.

La notion de polymorphisme existe en informatique et concerne le modèle de données d'un langage de programmation[1]. Il s'agit d'un *mécanisme de polymorphisme de types et de fonctions* qui élargit les possibilités d'écriture et de conceptualisation fondées uniquement sur le typage strict de certains langages. Un objet polymorphe peut changer de type pendant l'exécution du programme et les résultats des opérations effectuées sur cet objet dépendent de la forme prise dynamiquement par l'objet, en fonction du contexte dans lequel il se trouve. Par exemple les différentes interprétations d'un clic souris sur un objet diffèrent selon qu'il s'agisse d'une icône, d'une fenêtre, d'une ligne dans un menu etc. Les actions effectuées seront différentes. L'icône s'ouvre, la fenêtre passe sur le haut de la pile, l'action du menu est déclenchée. Jacques Longchamp précise l'intérêt technique du polymorphisme : « Cette approche évite de recourir aux fonctions universelles traitant tous les cas possibles ou aux fastidieuses suites de tests visant à déterminer dans quel cas de figure on se trouve et la fonction spécialisée à appeler » [LONGCHAMP, 1989, p.172]. Cependant ce n'est pas le polymorphisme technique inhérent au modèle de données d'un langage de programmation que nous proposons d'aborder. Le polymorphisme auquel nous nous référons est présent quel que soit le langage de programmation mis en oeuvre.

Au chapitre Cinq nous avons évoqué le « sphinx-machine » avec une double réalité du programme qui s'exerce, d'une part sur le registre mythologique, à travers un auteur dans une expérience de langage et de devenir, et d'autre part au plan de sa concrétisation matérielle dans le monde. Le programme est manifeste objectivement et concrétisé matériellement par la machine. L'idée exprimée par l'écriture du programme se traduit en fin de compte par un robot qui fonctionne et l'ensemble des programmes concernent une multitude de domaines d'activités.

Ainsi le premier point abordé est-il celui d'une ouverture généralisée. Mais cette ouverture

[1] A propos du polymorphisme et du concept associé de virtualité dans le C++ voir [LAPOINTE, 1995, p. 101 à 111]. Sur la question d'une approche objet de la programmation en langage C qui conduise au polymorphisme voir [BRAQUELAIRE 1998, p. 520].

est aussi sa propre limite du fait de l'exercice d'un matériau unique, le matériau informationnel. La similarité du matériau informationnel à travers des champs différenciés est évoquée et en dépit de l'évidence apparente, nous souhaitons faire sentir et rendre effectivement perceptible cette unité d'ensemble du matériau informationnel à travers des sujets diversifiés ; c'est pour le plaisir aussi, afin de conclure cette troisième partie, que nous retournons à la programmation avec des propositions d'ouverture sur le son, un germe de création à propos du texte et un système d'organisation inspiré initialement de mouvements collectifs des poissons ou des oiseaux.

1. **Ouverture généralisée**

1.1 Distance de l'écriture aux sujets traités

L'expérience de la programmation élargit considérablement les perspectives de création et de créativité. Elles ne sont pas limitées à des travaux sur l'image. Dans l'espace robographique le questionnement sur les images est une possibilité parmi d'autres mais le robographe est diversement connoté. Il peut disposer de composantes sonores, visuelles, graphiques, relationnelles, textuelles etc. Il n'y a pas de règle sur ce qu'il est ou doit être. Il est relié à l'inspiration, la sensibilité et la culture de l'auteur du programme. La programmation est à mettre au service de la curiosité et de l'expérimentation. L'écriture de programmes peut se détacher d'un certain nombre de classements, d'idées, d'habitudes ou de traditions selon tel ou tel domaine particulier d'activité ; elle peut au contraire s'y attacher, mais dans les deux cas il y a une distance introduite par cette écriture. Le polymorphisme naît de cette distance.

1.2 La forme issue de la démarche d'un auteur

Dans l'espace robographique la notion de forme est premièrement à envisager de façon pluridisciplinaire et cette pluridisciplinarité a des propriétés comme celle d'un ensemble par rapport aux parties. Elle relève de la démarche de l'auteur. Par exemple nous avons vu la notion de germe de création qui est une approche possible pour la recherche et l'exploration. Cette approche est une clé d'accès et de transversalité. Elle ouvre des domaines d'activités, comme le fait la figure de l'automate cellulaire que nous avons évoquée. Egalement elle découvre des perspectives de recherche par un jeu de transpositions interdisciplinaires. Il y a pourtant une spécificité du champ de l'espace robographique. Cette notion de germe de création permet à un auteur de conférer sa propre identité à ce champ par la démarche qu'il adopte. La forme apparaît alors directement reliée à cette démarche de l'auteur, qu'il s'agisse de la réalisation ou de l'interprétation d'un germe, du fait de l'appropriation du germe. Le germe est une forme et cette forme une idée.

1.3 Diversifications pragmatiques

D'autre part, quels que soient les questionnements mis en œuvre, ils passent par la modélisation selon des attitudes pragmatiques qui en elles-mêmes peuvent être très diversifiées. La modélisation s'achève finalement dans l'écriture du programme à travers un langage de programmation. A ce niveau encore, il est question de méthodologie, mais il n'y a pas d'obligation sur l'attitude à adopter pour coder le programme pourvu que soit réussie la réalisation du programme.

1.4 Une ouverture qui est sa propre limite

Sur des registres différents, qu'il s'agisse de la création, de la réalisation et des méthodes, l'expérience de la programmation dans l'espace robographique peut prendre une infinité de formes et probablement s'intéresser à n'importe quel sujet. Elle apparaît essentiellement polymorphe. En ce sens nous avons un moyen d'expression. Ce moyen d'expression s'appuie sur la conquête de l'espace robographique et parle, dans la distance de l'écriture aux sujets traités, des positionnements plus intimes qui sont adoptés par le concepteur comme par le lecteur.

En théorie n'importe quel sujet peut être abordé, toutefois ce support de la programmation opère une transposition du sujet abordé vers l'espace robographique. L'écriture construit et réalise un programme et cet objet est manifeste concrètement dans la machine. En tant que tel il acquiert des propriétés d'autonomie que nous avons évoquées avec le principe de Pinocchio. En sorte qu'il ne s'agit pas uniquement du sujet traité mais également de l'espace robographique et de l'idée que l'auteur du programme s'en fait.

2. Exemple d'ouverture sur le son

Au chapitre Onze nous avons présenté l'image telle qu'elle se présente en mémoire sous la forme d'une matrice d'octets qui constitue un ensemble de pixels. Le son apparaît également en mémoire mais il est constitué d'échantillons et non pas de pixels. Les échantillons sont stockés dans un tableau d'octets et les informations codées sont relatives aux fréquences et à l'amplitude. Il faut encore ajouter la durée du son, s'il est stéréo ou non et la fréquence d'échantillonnage qui est le nombre total d'échantillons en une seconde. Ensuite viennent éventuellement des informations qui concernent des propriétés musicales comme la possibilité de constituer des boucles de lecture du son.

Nous allons frayer rapidement un chemin jusqu'à être en mesure d'entrer dans la création sonore.

2.1 Du son dans une surface mémoire

Codage d'un son échantillonné

L'environnement Allegro code un son échantillonné, un « sample », à l'aide d'une structure (figure 16.1) :

```
    typedef struct SAMPLE
    {
(1)     int bits;
(2)     int stereo;
(3)     int freq;
(4)     int priority;
(5)     unsigned long len;
(6)     unsigned long loop_start;
        unsigned long loop_end;
        unsigned long param;
(7)     void *data;
    } SAMPLE;
```

Fig. 16.1 : Structure de données son dans l'environnement Allegro

(1) Le champ « bits » code l'amplitude 8 ou 16 bits
(2) Si le son est stéréo la valeur donnée est 1, 0 sinon
(3) Le champ « freq » désigne la fréquence d'échantillonnage. Il s'agit du nombre total d'échantillons en une seconde, par exemple 22050 hertz correspond à 22050 échantillons pour le codage d'une seconde de son. Le plus souvent les fréquences d'échantillonnage sont 11025, 22050, 44100 hertz.
(4) La variable « priority » est utilisée pour classer plusieurs sons joués simultanément par ordre d'importance. Ceci afin de prévenir le dépassement éventuel du nombre de voix maximum de la polyphonie autorisée par la carte son. Dans ce cas les sons moins importants, d'après le scénario, cèdent la place aux autres.
(5) La variable « len » designe la longueur du son en nombre d'échantillons.
(6) Deux variables, « loop_start » et « loop_end » sont pour la mise en place d'une boucle à l'intérieur du son. La variable « param » est utilisée en interne pour le fonctionnement du driver.
(7) C'est là, sous une forme identique à celle de l'image, que sont stockés tous les échantillons qui composent le son.

Allocation de mémoire pour un son

La fonction de création pour l'espace mémoire d'un son est la suivante (figure 16.2) :

(1) La fonction « create_sample » a quatre paramètres qui donnent la définition en 8 ou 16 bits de l'amplitude, la stéréo ou pas (1 ou 0), la fréquence d'échantillonnage et la longueur total du son en nombre d'échantillon. Cette fonction renvoie un pointeur du type SAMPLE* qui contient l'adresse de la zone mémoire concernée. Ce pointeur est tout d'abord déclaré local à la fonction.

(2) L'adresse d'une zone de mémoire qui couvre la taille d'un SAMPLE lui est attribuée.

En cas de problème rencontré lors de l'exécution de la fonction « malloc » la valeur NULL est renvoyée.

(3) Chaque champ de la structure SAMPLE à l'adresse contenue dans le pointeur « spl » est initialisé en fonction des valeurs passées en argument lors de l'appel de la fonction. Par défaut, le champ « priority » est initialisé à 128, une boucle éventuelle fait la taille du son complet, et le paramètre interne pour le driver est mis à 0.

(4) C'est la ligne qui réserve l'espace mémoire pour l'ensemble des échantillons constitutifs du son. La taille totale est le nombre d'échantillons « len » multiplié par 1 si on est en 8 bits et par 2 si on est en 16 et multiplié à nouveau par 1 si on est en mono et par deux si on est en stéréo.

```
(1)  SAMPLE *create_sample(int bits, int stereo, int freq, int len)
     {
     SAMPLE *spl;

(2)        spl = malloc(sizeof(SAMPLE));
            if (!spl)
                 return NULL;

(3)        spl->bits = bits;
           spl->stereo = stereo;
           spl->freq = freq;
           spl->priority = 128;
           spl->len = len;
           spl->loop_start = 0;
           spl->loop_end = len;
           spl->param = 0;

(4)        spl->data = malloc(len * ((bits==8) ? 1 : sizeof(short)) * ((stereo) ? 2 : 1));
           if ( ! spl->data) {
                 free(spl);
                 return NULL;
           }

 (5)       memset(spl->data, 0, len * ((bits==8) ? 1 : sizeof(short)) * ((stereo) ? 2 : 1));

 (6)       lock_sample(spl);
 (7)            return spl;
     }
```

Fig 16.2 : Créer l'espace du son en mémoire

(5) Pour finir tous les octets à l'adresse « spl->data » sont initialisés à 0 ce qui donne un silence.

(6) La fonction « lock_sample » verrouille l'espace de mémoire pour des raisons internes aux fonctionnements des différentes plateformes et systèmes d'exploitations à partir desquels la librairie Allegro peut fonctionner.

(7) L'adresse de l'espace mémoire du son, à savoir l'ensemble des échantillons qui le constituent en fonction des paramètres passés, est renvoyée à la fonction appelante.

Lecture du son

L'environnement fournit plusieurs approches pour la lecture d'un son qui incluent des effets sonores. Mais la plus simple est un appel à la fonction :

```
int play_sample(const SAMPLE *spl, int vol, int pan, int freq, int loop);
```

Cette fonction prend alors l'adresse du son échantillonné en argument, le volume souhaité entre 0 et 255, le panoramique de 0 à 255 avec 128 au milieu, la fréquence exprimée en valeurs relatives avec 1000 pour le son tel qu'il a été échantillonné, 2000 pour le double etc. Si le dernier paramètre « loop » est mis à un la lecture boucle le son sur lui-même.
La fonction renvoie un numéro qui désigne une voix de la carte son utilisée et un nombre négatif si aucune voix n'est disponible.

Pour arrêter le son :

```
void stop_sample(const SAMPLE *spl);
```

Pour modifier des paramètres de lecture du son au besoin pendant la lecture elle-même :

```
void adjust_sample(const SAMPLE *spl, int vol, int pan, int freq, int loop);
```

2.2 Organiser un flux sonore continu

Structure de données pour un flux continu

Le son échantillonné, stocké dans un tableau, est dans la mémoire centrale de l'ordinateur. Si le son est trop long ce n'est pas possible. De plus un programme peut générer du son et dans ce cas il ne s'agit pas d'un son préexistant échantillonné mais d'un son de synthèse. Dans ces deux cas il est nécessaire d'organiser un flux sonore à partir d'un buffer qui est un son intermédiaire en mémoire centrale. Le buffer est rempli au fur et à mesure de l'avancée du son dans le temps.

Imaginons un tableau d'une taille « t * 2 », il est divisé en deux parties égales et la première partie est remplies avec le début du son. Au début ce sont les échantillons contenus dans la première partie du tableau qui sont joués. Dans le même temps la seconde partie du tableau est préparée avec la suite du son à jouer. Dès que la lecture du son arrive dans la seconde partie du tableau, c'est la première partie qui est remplie avec la suite du son. Lorsque la lecture arrive à la fin du tableau, elle reprend au début de la première partie du tableau et la seconde partie est remplie avec la suite du son etc.

Dans cette perspective, la structure SAMPLE est intégrée dans une structure AUDIOSTREAM qui apporte des informations supplémentaires (figure 16.3) :

(1) Variable qui désigne la voix utilisée par la carte son.
(2) Structure SAMPLE qui correspond au buffer son.
(3) La taille totale voulue pour le buffer.

(4) Indique éventuellement si chaque moitié du buffer total est fragmentée en sous-parties.
(5) Désigne la partie du buffer à remplir.
(6) Désigne la partie active, en cours de lecture.
(7) Verrouillage de la zone en cours de lecture.

Fig. 16.3 : Double buffering sonore

```
    typedef struct AUDIOSTREAM
    {
(1)     int voice;
(2)     struct SAMPLE *samp;
(3)     int len;
(4)     int bufcount;
(5)     int bufnum;
(6)     int active;
(7)     void *locked;
        } AUDIOSTREAM;
```

Trois fonctions organisent le flux sonore continu

La première fonction initialise la lecture à partir des paramètres voulus pour le son et renvoie l'adresse d'une structure AUDIOSTREAM dont tous les champs ont été paramétrés :

```
AUDIOSTREAM *play_audio_stream(int len, bits, stereo, freq, vol, pan);
```

Le premier paramètre concerne la taille du buffer. En fait ce buffer est doublé en mémoire. Le second définit l'amplitude en 8 ou 16 bits. Le troisième détermine s'il y a ou pas stéréo. Le quatrième concerne la fréquence d'échantillonnage. Le cinquième donne le volume et le dernier le panoramique.

Après appel de cette fonction tout est près pour la lecture du son. Cependant le buffer est vide et il convient de le remplir régulièrement avec des informations de son. Pour ce faire il faut disposer de l'adresse d'un bloc de mémoire qui corresponde à une partie du buffer. C'est le rôle de la fonction :

```
void *get_audio_stream_buffer(AUDIOSTREAM *stream);
```

Cette fonction retourne un pointeur générique de type void* c'est-à-dire l'adresse d'un bloc de mémoire réservée en fonction de la taille voulue pour le buffer et selon les paramètres d'amplitude et de stéréo. Dans le programme ce buffer peut ensuite être rempli des informations que l'on souhaite diriger sur la carte son. Pendant la lecture du son, s'il n'y a rien à changer dans le buffer, la fonction retourne NULL. Mais dès qu'une moitié du buffer est terminée, l'adresse d'un nouveau bloc est renvoyée afin que la suite du son puisse être stockée dans cette partie du buffer. Evidement cela nécessite d'appeler régulièrement cette fonction pendant la lecture du son afin de contrôler son évolution.

Notons que cette fonction travaille en partenariat avec une troisième au sein d'une boucle dans le programme :

```
void free_audio_stream_buffer(AUDIOSTREAM *stream);
```

Cette troisième fonction est appelée juste après qu'une partie du buffer soit rafraîchie avec de nouvelles informations sonores. Elle actualise tous les paramètres nécessaires afin que la prochaine partie du buffer soit effectivement la suivante.

Mise en place du flux sonore continu

La mise en place dans un programme de ce mécanisme de lecture du son est illustrée avec la figure 16.4 :

```
(1)  #define AMPLITUDE       16
     #define FREQ_ECH        44100

                (…)
(2)  AUDIOSTREAM *stream;
                (…)
(3)  if ( install_sound( DIGI_AUTODETECT, MIDI_NONE, NULL ) != 0 )
          ERREUR( "Carte son" );

(4)  stream = play_audio_stream( TAILLE_BUFFER, 16, 1, 44100, 30, 128 );
     if ( ! stream )
        ERREUR( "Création du buffer" );

(5)  while ( ! keypressed ( ) ) {

(6)       unsigned int *p = ( unsigned int* )get_audio_stream_buffer ( stream );
(7)       if ( p ) {

(8)            création_son ( p );
(9)            free_audio_stream_buffer( stream );
          }
     }
(10)      stop_audio_stream( stream );
                (…)
```

Fig. 16.4 : Mise en place d'un double buffering sonore

(1) Deux macros sont définies, une pour l'amplitude et une pour la fréquence d'échantillonnage.

(2) Déclaration soit en global, soit dans la fonction appelante d'un pointeur sur une structure AUDIOSTREAM.

(3) Fonction d'initialisation en relation avec le driver de la carte son et qui donne accès à son utilisation dans le programme.

(4) Appel de la fonction « play_audio_stream » avec les valeurs souhaitées passées en argument. TAILLE_BUFFER est une macro dont la valeur est de préférence une puissance de deux, 1024 ou 2048 par exemple. Elle définit la taille du buffer. L'amplitude est ici 16 bits, la stéréo est souhaitée, la fréquence d'échantillonnage 44100 hertz, le volume est à 30 et le pan au milieu.

(5) La boucle à l'intérieure de laquelle le son est joué et traité. Cette boucle prend fin avec la pression de la touche Echap.

(6) La fonction « get_audio_stream_buffer » renvoie NULL si la partie active du buffer est toujours en lecture et l'autre préparée. Elle renvoie l'adresse d'un bloc de mémoire si une partie du buffer doit être remplie.

(7) Si l'adresse du bloc mémoire désignée par le pointeur « p » n'est pas nulle, il faut le remplir des informations sonores souhaitées.

(8) La fonction « creation_son » prend l'adresse du bloc en argument et prépare une petite sonnerie de téléphone à partir de sinusoïdes (détails plus bas).

(9) La fonction « free_audio_stream_buffer » actualise la nouvelle situation, afin de permettre un accès au bloc suivant avec l'avancement de la lecture.

(10) Au sortir de la boucle toute la mémoire utilisée est libérée avec l'appel à la fonction « stop_audio_stream ».

2.3 Sonorités, perspectives de morphing

Création de sonorités

Nous voici arrivés maintenant au lieu de la création du son. Présentée à titre indicatif, la fonction « creation_son » est une petite sonnerie de téléphone (figure 16.5).

Admettons que le cercle représente une seconde de temps. De ce fait un angle de base est obtenu en divisant la circonférence du cercle par la fréquence d'échantillonnage. Si l'on est en 44100 hertz, cela donne PI*2/44100. Imaginons un point « pt » qui tourne à une vitesse constante autour de ce cercle. Selon l'angle donné à son pas d'avancement, il va pouvoir faire entre 0 et 44100 tours du cercle en une seconde. Le nombre de tours par seconde détermine des fréquences. Moins il y a de tours, plus la fréquence est basse ainsi que la sonorité ; inversement, plus le nombre de tours est élevé, plus la fréquence est haute et la sonorité aigue. Le nombre de tours par seconde se traduit par l'angle de base multiplié par la fréquence, c'est-à-dire « (PI*2/fréquence échantillonnage) * fréquence sonore » . Il s'agit d'un angle et le son à telle ou telle fréquence peut s'exprimer à travers toutes les valeurs d'une fonction sinus, ces valeurs, comprises entre 0 et 1 étant multipliées par l'amplitude. D'où essentiellement la formule :
« amplitude * sin((PI*2/fréquence échantillonnage) * fréquence sonore) » qui permet de récupérer les valeurs pour chaque étape de temps d'un son à une fréquence donnée.

Ce principe est mis en pratique dans la fonction « creation_son » :

(1) Macro qui donne la valeur PI.

(2) La fonction « creation_son » prend la partie à remplir du buffer en argument et son paramètre est un pointeur d'entiers non signés.

```
(1) #define PI    3.14159265

(2)  void creation_son (unsigned int *p)
     {
(3)  static int       freq = 440;
     static float  angle = 0;
     static int  pas = 250;

(4)  float anglebase;
     double d;
     int i, signe;

(5)        freq+=pas;
           anglebase = ( 2 * PI / FREQ_ECH )*freq;
(6)        for ( i=0; i<TAILLE_BUFFER; i++, angle+=anglebase) {
                d=sin(angle);
                p[ i ] = ( AMPLITUDE * d );
           }
(7)        if(freq > 2000 || freq < 0){
                freq-=pas;
                signe = (pas >0)? 1 : -1;
                pas= (rand()%450)+50;
                pas*= -signe;
           }
     }
```

Fig. 16.5 : Créer du son selon le même langage que celui utilisé pour l'image

(3) Trois variables « static ». Ce sont des variables locales à la fonction qui ont la particularité de conserver leurs valeurs entre deux appels de la fonction. Elles ne sont pas détruites automatiquement avec le retour de la fonction.
« freq » désigne une fréquence sonore, « angle » une position par rapport au cercle et son origine, « pas » détermine une modification de la fréquence de façon à ce qu'à chaque appel de la fonction la fréquence soit différente.

(4) Viennent ensuite trois variables, « anglebase » va correspondre au pas d'avancement de l'angle selon la fréquence, « d » récupère le sinus de l'angle. « i » et « signe » sont deux variables-outils.

(5) La fréquence est modifiée selon un pas initial de 250. « anglebase est calculé selon la formule circonférence du cercle divisée par la fréquence d'échantillonnage et multipliée par la fréquence sonore.

(6) Boucle for pour le calcul des échantillons à stocker dans le buffer « p ». La boucle passe en revue chaque position selon la taille du buffer déterminée par la macro TAILLE_BUFFER. Chaque position du buffer prend pour valeur le sinus de l'angle multiplié par l'amplitude. Lors du premier appel de la fonction la variable « angle » est à 0. Ensuite elle est incrémentée par pas de « anglebase », calculé précédemment avec la fréquence voulue pour le son. Une fois la boucle terminée, le buffer contient un son déterminé par une fréquence unique.

(7) Il s'agit de limiter l'évolution de la fréquence dans une fourchette entre 0 et 2000. S'il y a débordement dans un sens ou dans l'autre la progression est inversée et un nouveau pas est défini aléatoirement.

Morphing son-image

Le matériau informationnel traduit le son et l'image en structures de données proches sinon identiques. Si l'on songe aux expressions « void *data » de la figures 16.1 pour le son échantillonné et « void *dat » de la figure 11.2 pour l'image bitmap, cela fait naître des interrogations et des désirs d'expérimentation.

Par exemple ce qui normalement est de l'image peut être envoyé à la carte son et réciproquement. Le résultat se heurte à ce qui distingue bien entendu le son et l'image du point de vue des rapports entre les données et du contenu organisationel surajouté aux données. Les échantillons ne sont pas des pixels et réciproquement. Toutefois il est sans doute possible d'envisager des modalités de glissement de l'un à l'autre. Un type de filtrage particulier qui permette de passer d'un son à une image et d'une image à un son[1]. Imaginons un son qui s'éteigne et se désagrège progressivement alors qu'apparaît une image à l'écran ou au contraire une image qui disparaît doucement en même temps que se constitue une sonorité. Des résultats harmonieux ne sont pas garantis mais l'idée est séduisante.

Chercher du son dans les images

Y-a-t-il des possibilités d'extractions sonores liées à des images ? Comment faut-il s'y prendre pour chercher du son dans des images ? Quelles trajectoires pour le son dans l'image ? Approcher le son par l'image reviendra à récupérer des éléments de l'organisation de l'image pour générer du son.

Faire des images avec du son

Utiliser la cohérence intrinsèque du son dans la structuration d'images, dans leurs développements cinématiques et historiques. Par exemple le son peut être utilisé comme système de commande, éventuellement silencieux, pour le développement cinématique d'une image. On peut imaginer de programmer un système d'animation d'une image inspiré de la reconnaissance vocale. L'idée serait de mettre en scène des personnages dotés d'oreilles c.à.d « pilotés » par des musiques en fonctions des fréquences et des motifs liés à une analyse structurelle du son (Transformée de Fourier).

[1] A ce sujet peuvent être consultés [ROADS, 1998, p. 313] et les travaux de Vincent Lesbros [LESBROS, 1995, p. 40]

3. Implémentation d'un robographe textuel

Afin d'illustrer l'ouverture vers le texte en tant que matière informationnelle, voici une version germe de création du programme Discours coton présenté au chapitre Sept. L'objectif est la génération de texte à partir d'une librairie de mots.

3.1 Principe

S'appuyer sur des protophrases. Ce sont des modèles de phrases littéralement utilisés comme des moules à phrases. Chaque protophrase est un syntagme constitué de primitives qui sont des catégories de mots. La génération automatique consiste à prendre pour chaque catégorie un mot au hasard. Voici un exemple de protophrase :

Sélectionner dans une liste	:	verbe à l'infinitif
Ponctuation	:	« , » (une virgule)
Sélectionner dans une liste	:	verbe à l'infinitif
Ponctuation	:	« , » (une virgule)
Sélectionner dans une liste	:	verbe à l'infinitif
Expression fixe	:	« et »
Sélectionner dans une liste	:	verbe à l'infinitif
Expression fixe	:	« tous les »
Sélectionner dans une liste	:	nom masculin pluriel
Expression fixe	:	« de la »
Sélectionner dans une liste	:	nom féminin commençant avec une consonne
Sélectionner dans une liste	:	adjectif féminin singulier
Ponctuation	:	« . » (un point)

3.2 Mise en forme et initialisation

Le premier point est la construction d'une librairie. Pour chaque catégorie un tableau regroupe les mots disponibles et chaque mot est implémenté par une chaîne de caractères, à savoir un « char* ». Ainsi, la structure de données pour la librairie est un ensemble de tableaux de chaînes de caractères. Nous avons déjà rencontrés ce type sous la forme « char* line[ ] » au chapitre Onze à propos de l'implémentation d'une image bitmap. Il a servi pour définir l'espace mémoire de l'image. Nous allons l'utiliser maintenant du point de vue de l'écriture des mots et de leur stockage.

Précisions sur les caractères et chaînes de caractères

Dans le langage C une lettre, un caractère, sont codés sur un octet et correspondent au type « char ». Une constante de type caractère apparaît entourée par deux apostrophes. ' a ' est la constante caractère *a*, c'est la lettre *a* dans un programme. Pour obtenir des mots il faut assembler des caractères, d'où la construction d'une « chaîne de caractères ».

La chaîne de caractères est une suite de caractères terminées par le caractère nul : ' \0 '. Ce

caractère vaut zéro. La suite est délimitée par des guillemets, par exemple "bonjour" est une chaîne de caractère. En fait cette suite de caractères correspond à un tableau de huit caractères, sept lettres plus le caractère nul « \0 » qui signale la fin de la chaîne. C'est un tableau de char. L'espace, qui est une ponctuation, est aussi un caractère. De ce fait une chaîne de caractère peut contenir une phrase entière ou plusieurs. "Il fait beau. Je prends ma canne et mon chapeau." est une chaîne de caractères.
Il y a trois formes sous lesquelles se présentent les chaînes de caractères dans un programme.

1. Une *constante chaîne de caractère* comme : "bonjour". En fait elle est mémorisée comme un tableau de caractère. Il contient les caractères de la chaîne et se termine par un ' \0 ' qui en marque la fin [KERNIGHAN, 1995, p.29].

2. Un *tableau de caractère* du genre :
   char chaine1[10] = {' b ', ' o ', ' n ', ' j ', ' o ', ' u ', ' r ', ' \0 ' } ;
   ou encore
   char chaine1[ ] = "bonjour" ;

   Remarquons que dans le premier cas « chaine1[10] » il y a deux cases disponibles en plus (initialisées à zéro). Dans le second, la taille du tableau « chaine1 » est automatiquement ajustée sur le nombre de caractères de la chaîne qui lui est affectée à l'initialisation. Le ' \0 ' final est implicite. La taille du tableau « chaine1 » est alors de huit caractères.

3. Le *pointeur de caractères* : char* chaine2 = "bonjour" ;
   Cette dernière formulation de la chaîne de caractère est à rapprocher de l'équivalence établie entre tableaux et pointeurs. Dans ce cas toujours, la chaîne est implicitement terminée par le caractère nul ' \0 '.

Le tableau de pointeurs de caractères va permettre de stocker les mots par catégorie. Chaque catégorie a son tableau de char* initialisé directement dans le programme. Une sentinelle est placée à la fin du tableau. C'est une astuce pour fournir un test d'arrêt à la fin du tableau alors que le nombre de ses éléments n'est pas connu. La sentinelle est simplement la valeur 0 castée en char*, c'est-à-dire la valeur NULL mais dans le type char* (figure 16.6)

```
char*adj_f_p[ ]={
"transcendantales",
"probables",
"banales",
"barbares",
"baroques",
"cognitives",
…… ,

( ( char* ) 0 ) } ;
```

Fig. 16.6 : Construire une liste selon une catégorie de mots

Compte-tenu du nombre important de catégories de mots ainsi parfois que du nombre éventuellement élevé de mots par catégorie, il est plus clair dans le programme d'utiliser un fichier par catégorie. De ce fait les données du programme se trouvent partagées sur plusieurs fichiers.

Mot-clé « extern »

Pour qu'une donnée définie sur un fichier soit connue sur un autre fichier il faut indiquer à cet autre fichier que la donnée en question est externe, faute de quoi elle n'est pas reconnue au moment de la compilation. C'est le rôle du mot-clé « extern » qui permet à une fonction dans un fichier donné de se référer à des données externes, déclarées sur d'autres fichiers en dehors de toute fonction [BRAQUELAIRE 1998, p26, 71] [KERNIGAHN 1995, p31].

Voici l'ensemble des catégories de mots du programme. Chacune correspond à un tableau de char* qui est défini dans un fichier dédié :

```
extern char*adj_f_p[ ];          // adjectif féminin pluriel
extern char*adj_m_p[ ];          // adjectif masculin pluriel
extern char*adj_f_s[ ];          // adjectif féminin singulier
extern char*adj_m_s[ ];          // adjectif masculin singulier
extern char*passif_f_s[ ];       // passif féminin singulier
extern char*passif_m_s[ ];       // passif masculin singulier
extern char*passif_m_p[ ];       // passif masculin pluriel
extern char*nom_f_cons[ ];       // nom féminin singulier qui commence par une consonne
extern char*nom_m_cons[ ];       // nom masculin singulier qui commence par une consonne
extern char*nom_f_voy[ ];        // nom féminin singulier qui commence par une voyelle
extern char*nom_m_voy[ ];        // nom masculin singulier qui commence par une voyelle
extern char*nom_f_p[ ];          // nom féminin pluriel
extern char*nom_m_p[ ];          // nom masculin pluriel
extern char*nom_prop[ ];         // nom propre
extern char*nombres[ ];          // nombre
extern char*adverbe[ ];          // adverbe
extern char*amorce[ ];           // diverses formules
extern char*infinitif[ ];        // verbe à l'infinitif
extern char*verb_pr_3s[ ];       // verbe présent 3e  pers. singulier
extern char*verb_pr_3p[ ];       // verbe présent 3e  pers. pluriel
extern char*se_verb_pr_3s[ ];    // verbe présent 3e  pers. singulier commençant par « se »
extern char*se_verb_pr_3p[ ];    // verbe présent 3e  pers. pluriel commençant par « se »
extern char*verb_cond_3s[ ];     // verbe conditionnel, 3e  pers. singulier
extern char*verb_pr_3s_obj[ ];   // verbe présent 3e  pers. singulier suivi adverbe
```

Toujours au niveau des structures de données, il reste à stocker le résultat qui est une ou plusieurs phrases générées par le programme. Il s'agit d'un char* déclaré en global. Les fonctions qui l'utilisent sont regroupées dans un seul fichier. Le nombre total de protophrases disponibles dans le programme est accessible via une macro ce qui donne les deux déclarations suivantes :

```
#define NB_PROTO                8
char *LISTE;
```

Faire une protophrase

La première fonction est celle de la construction d'une protophrase. Sur le même modèle nous avons plusieurs fonction qui déterminent des protophrases différentes (figure 16.7).

```
(1)  void fabrique_proto_1( void)
     {
(2)  char *mot[14];
     int i;
(3)       mot[0]  = "La";
          mot[1]  = select(nom_f_cons);
          mot[2]  = select(adj_f_s);
          mot[3]  = "qui";
          mot[4]  = select(verb_pr_3s_obj);
          mot[5]  = "dans les";
          mot[6]  = select(nom_m_p);
          mot[7]  = select(adj_m_p);
          mot[8]  = "peut";
          mot[9]  = select(infinitif);
          mot[10] = "l'";
          mot[11] = select(nom_f_voy);
          mot[12] = select(adj_f_s);
          mot[13] = ".";

(4)       for( i=0; i<14; i++)
               ajoute_liste(mot[i]);
     }
```

Fig. 16.7 : Fonction pour la construction d'une protophrase

(1) La fonction « fabrique_proto_1 » n'a pas de paramètre et ne renvoie rien.

(2) Son principe est de stocker un ensemble de mots selon la définition d'une protophrase. Le stockage des mots est effectué à l'aide d'un tableau de char* de la taille du nombre des éléments qui composent le syntagme. Il est nommé « mot ».

(3) Chaque position du syntagme correspond à la position équivalente du tableau par numéro d'ordre. Certaines positions sont affectées d'un mot fixe, d'une ponctuation ou d'un article suivi d'une apostrophe. D'autres sont piochées dans la liste du genre voulu avec un appel à la fonction « select » qui prend en argument la liste dans laquelle sélectionner un mot avec le nom du tableau correspondant.

(4) A l'issue du remplissage du tableau « mot » il s'agit d'ajouter chaque mot de la phrase obtenue au texte en cours d'élaboration. C'est le rôle de la fonction « ajoute_liste » qui prend en argument un char* et est appelée pour chaque mot du syntagme via une boucle for.

Détail de la fonction « select »

La sélection d'un mot dans une liste s'effectue de la façon suivante (figure 16.8) :

(1) La fonction « select » prend un pointeur de pointeur sur char en paramètre, à savoir

une double indirection avec un char**. Ceci afin de pouvoir passer en argument un tableau de char* qui est lui-même une double indirection, un char**. La fonction renvoie une chaîne de caractères

(2) Il n'y a pas dans le programme de référence à la taille des listes et le nombre des éléments n'est pas connu. Ainsi le premier point est de compter le nombre des éléments de la liste. La variable « i » est incrémentée de 0 à la fin de la liste.

Fig. 16.8 : Sélection d'un mot dans une liste

```
(1)  char* select(char** type_mot)
     {
     int i;
(2)        for ( i = 0 ; type_mot[ i ] != NULL; i++)
                ;
(3)        i = rand% i ;
(4)        return type_mot[ i ];
     }
```

(3) Ensuite un nombre pseudo-aléatoire modulo i est donné par la fonction « rand » et affecté à i.

(4) Ce nombre détermine le mot de la liste que retourne la fonction avec l'expression « type_mot[i] »

Détail de la fonction « ajoute_liste »

Une fois l'ensemble des mots du syntagme de la protophrase réuni dans le tableau « mot » de la fonction « frabrique_proto_1 », il reste à copier dans l'ordre chacun de ces mots à la suite du texte en cours d'élaboration et stocké dans la variable globale de type char* « LISTE » (figure 16.9) :

(1) La fonction « ajoute_liste » ne renvoie rien et a un char* en paramètre. Il s'agit du mot à ajouter à la liste des mots du texte en cours.

(2) Deux variables sont nécessaires, une pour avoir la taille du texte déjà réalisé et l'autre pour avoir la taille du mot à ajouter de façon à pouvoir augmenter la taille du bloc de mémoire en conséquence.

(3) Initialisation de la variable « t2 » avec un appel à la fonction « strlen » de la librairie standard du C <string.h>. Cette fonction retourne la longueur en nombre de caractères de la chaîne passée en argument (sans compter le caractère nul de la fin).

(4) S'il s'agit du premier mot de la liste LISTE prend la taille « t2 » du premier mot et une adresse mémoire lui est affectée.

(5) Si LISTE est déjà une adresse mémoire, des mots lui ont été affectés et il est nécessaire de connaître sa taille afin de pouvoir l'augmenter en proportion du nouveau mot à ajouter. C'est ce qui motive l'appel à la fonction « strlen » qui prend LISTE en argument et affecte

la valeur de retour à la variable « t1 ». La longueur total du texte, t1, est augmentée de 1 afin de mettre le caractère nul nécessaire à la fin de la chaîne de caractères. La taille du bloc de mémoire à l'adresse de LISTE est ensuite réallouée selon le nouveau nombre total des caractères (t1+t2) requis par le texte.

```
(1)  void ajoute_liste(char* mot)
     {
(2)  int t1=0, t2;

(3)       t2=strlen(mot);

(4)       if( ! LISTE)
               LISTE = calloc(sizeof(char),t2);
          else{
(5)            t1=strlen(LISTE)+1;
               LISTE = (char*)realloc(LISTE,sizeof(char)*(t1+t2));
(6)            if( mot[0] != '.' && mot[0] != ',' && LISTE[t1-2] != 39 ){
                    LISTE[t1-1]= 32;
                    LISTE[t1]='\0';
               }
               else
(7)                 LISTE[t1-1]='\0';
          }
(8)       strcat(LISTE,mot);
     }
```

Fig. 16.10 : Ajouter chaque mot au texte

(6) Il s'agit de gérer les espaces qui séparent les mots. Il y en a toujours un sauf dans le cas de ponctuation et d'une apostrophe. Précisons que chaque lettre fait l'objet d'un codage et peut être traduite par un nombre entier[1]. Ainsi la valeur ASCII de l'apostrophe est 39 et celle de l'espace est 32. Si le mot n'est pas un point, s'il n'est pas une virgule et s'il n'est pas une apostrophe, un espace est ajouté à la dernière position de la chaîne liste, juste après le dernier caractère et juste derrière lui est indiquée la fin de la chaîne avec le caractère nul.

(7) Si en revanche le mot est un point, une virgule ou une apostrophe, il n'y a pas d'espace à ajouter et la fin de la chaîne est juste derrière le dernier caractère, indiquée par le caractère nul.

(8) Pour finir, une fois l'espace mémoire préparé et la jonction entre la liste existante et le nouveau mot éclaircie, il reste à copier le mot à la fin de la liste. Pour ce faire c'est un appel à la fonction « strcat » de la librairie standard <string.h>. Cette fonction prend deux chaînes de caractères en argument et concatène la seconde à la suite de la première à partir du premier caractère nul qu'elle rencontre., et sous réserve que l'espace mémoire soit suffisant.

[1] C'est le codage ASCII qui est utilié. Initialement ce codage définit 127 caractères de base codés sur 7 bits. Il contient les lettres non accentuées, les chiffres, les caractères de ponctuation, les caractères de contrôle, un certain nombre de caractères spéciaux (« & » et « \ » par exemple). Viennent ensuite des extensions du codage ASCII codées sur 8 bits ; la plus répandue est la norme OS/LATIN-1 qui permet le codage de nombreuse langues en Europe.

Le principe de la fabrication des protophrases est mis en place ainsi que l'élaboration progressive du texte final à travers une liste de mots. Il reste à implémenter la mise en œuvre par le programme.

3.3 Processus générateur

Le principe génératif est mis en scène de la façon suivante (figure 16. 11):

Fig. 16.11 : Mise en œuvre du processus génératif

```
(1)  while(!key[KEY_ESC]){

(2)       if(key[KEY_ENTER]){
               degrade_pas_variable(pal);
               clear(screen);
(3)            construit_texte(irand(5)+1);
(4)            affiche_texte();
          }
     }
```

(1) Au sein d'une boucle while infinie sauf pression sur la touche Echap :

(2) si pression sur la touche Entrée (enter), une nouvelle palette de couleur est créée, l'écran est effacé,

(3) La fonction « construit_texte » est appelée. Cette fonction prend un nombre aléatoire de protophrases en argument dans la fourchette de 1 à 5.

(4) Lorsque le texte est créé il est affiché à l'écran avec un appel à la fonction « affiche_texte ».

Détail de la fonction « construit_texte »

Cette fonction permet de sélectionner plusieurs protophrases et de les enchaîner en une seule liste, le texte résultant (figure 16.12) :

(1) La fonction « construit_texte » prend comme paramètre le nombre total de protophrases à additionner pour faire un texte.

(2) Le premier point consiste à libérer la mémoire occupée par le texte précédent, si ce texte existe à savoir que la variable LISTE n'est pas nulle.

(3) Ensuite, les protophrases sont sélectionnées au hasard selon le nombre souhaité passé en argument et compte-tenu du nombre total des modèles dans le programme. Pour ce faire une boucle do-while est utilisée.

(4) Dans la boucle un nombre aléatoire est tout d'abord obtenu dans la fourchette du nombre maximum de protophrases défini avec la macro NB_PROTO (Nous avons déjà rencontré la macro « irand ». Elle fait correspondre à « irand(n) » « rand()%n »). Ce

nombre est affecté à la variable « choix ».

(5) Un switch, fondé sur la valeur de la variable « choix », dirige vers l'appel de la fonction de fabrication de protophrase correspondante.

(6) Le nombre de protophrases qui restent à fabriquer et ajouter à la liste est diminué de 1.

```
(1)  void construit_texte(int nombre_proto)
     {
     int choix;

(2)        if( LISTE )
               free(LISTE);
           LISTE = NULL;

(3)        do{
(4)             choix = irand(NB_PROTO);
(5)             switch(choix){
                     case 0 : fabrique_proto_1(); break;
                     case 1 : fabrique_proto_2(); break;
                     case 2 : fabrique_proto_3(); break;
                     case 3 : fabrique_proto_4(); break;
                     case 4 : fabrique_proto_5(); break;
                     case 5 : fabrique_proto_6(); break;
                     case 6 : fabrique_proto_7(); break;
                     case 7 : fabrique_proto_8(); break;
                     default : fabrique_proto_2(); break;
                }
(6)             nombre_proto--;
(7)        }while(nombre_proto > 0);

(8)        correction_majuscules();
     }
```

Fig. 16.12 : Création du texte par addition de protophrases

(7) L'action reprend tant qu'il y a des protophrases à ajouter à la liste, c'est-à-dire que le nombre des protophrases à faire est supérieur à 0.

(8) Une fois le texte terminé, il y a à corriger les majuscules après les points. C'est le rôle de la fonction « correction_majuscules ».

Détail de la fonction « correction_majuscules »

Cette fonction parcourt la chaîne de caractères LISTE et à chaque fois qu'un point est trouvé, la première lettre du premier mot suivant est mise en majuscule (figure 16. 13) :

(1) La fonction « correction_majuscules » ne renvoie pas de valeur et n'a pas de paramètre.

(2) Deux variables locales sont utilisées. La première « p » est un char* initialisé sur l'adresse de LISTE qui va servir à parcourir la liste. La seconde « flag » sert à marquer la rencontre éventuelle d'un point.

(3) (6) Boucle do-while avec pour test d'arrêt le fait d'arriver sur le caractère nul de fin de chaîne. L'expression « *p++ » incrémente p de 1 en 1 et est équivalente à *(p++). La valeur de *p++ est toutefois celle du caractère pointé *avant incrémentation de p* ; « ++ » placé après p ne le modifie qu'après utilisation du caractère pointé. Lorsque *p++ est égal à NULL c'es-à dire 0, le test devient faux (renvoie de la valeur 0) et la boucle s'arrête.

```
(1)  void correction_majuscules(void)
     {
(2)  char *p = LISTE;
     int flag = TRUE;

(3)       do{

(4)            if( *p == '.' )
                    flag=TRUE;

(5)            if ( flag == TRUE && isgraph ( *p ) ){
                    flag = FALSE;
                    if ( islower ( *p ) )
                              *p -= 32;
               }
(6)       }while ( *p++ );
     }
```

Fig. 16.13 : Correction des majuscules

(4) La première instruction teste si à la position courante de la chaîne il y a un point. Si oui, la variable flag est mise sur TRUE (la valeur -1 dans l'environnement Allegro). Cette variable est initialisée à TRUE afin de repérer la première lettre du texte qui doit prendre une majuscule et n'est pas précédée d'un point.

(5) Deuxième test : si flag est sur TRUE, un point a été rencontré et il faut mettre en majuscule la première lettre qui arrive. Il faut attendre cette première lettre et c'est le rôle de l'appel de la fonction « isgraph » de la librairie <ctype.h> standard du C. Cette fonction retourne vrai si son argument est un caractère affichable différent d'un espace et faux sinon. Si flag est TRUE et « isgraph » retourne vrai alors les instructions sont effectuées : flag reprend la valeur FALSE et si la lettre courante n'est pas déjà une majuscule elle est transformée en majuscule en modifiant simplement sa valeur ASCII (il y a un décalage de 32 entre les lettres majuscules et les minuscules).

Détail de la fonction « affiche_texte »

Une fois le texte mis en forme et corrigé, il n'y a plus qu'à l'afficher, c'est le rôle de l'appel de la fonction « affiche_texte » (figure 16.14) :

(1) La fonction « affiche_texte » n'a pas de paramètre et ne renvoie aucune valeur.

(2) Plusieurs variables locales sont nécessaires à son fonctionnement. Trois chaînes de caractères dont une, « tmp », qui est initialisée avec une taille de deux caractères (un caractère plus le caractère nul). En fait le texte va être affiché lettre par lettre. La variable

« tmp » va permettre d'isoler chaque lettre et d'utiliser une fonction de la librairie Allegro pour l'affichage d'une chaîne de caractère. La chaîne de caractère « p » est initialisée sur LISTE et va servir à parcourir le texte. Les variables « x » et « y » correspondent à la position dans l'écran. « tx » va servir à délimiter la taille horizontale pour le texte. Comme le texte sera toujours relativement court il n' y a pas de limite verticale. Les variables « pasx » et « pasy » déterminent les pas d'avancement horizontaux et verticaux pour chaque lettre. La variable « color » est pour la couleur.

```
(1)  void affiche_texte( void )
     {
(2)  char *c, *p=LISTE;
     char *tmp=malloc(sizeof(char)*2);
     int x,y,tx,color=0;
     int i,pasx,pasy;

(3)       x=ECRAN_X/8;
          tx=ECRAN_X-x;
          pasx = 8;
          pasy = pasx+8;
          y = x*2;
          tmp[ 1 ]='\0';

(4)       do{
(5)            if(*p==32){
                    c=p+1;
                    for( i=0; (*c != 32) && (*c != 0); c++,i++)
                         ;
                    if( x + (i*pasx) >tx ){
                         x=ECRAN_X/8;
                         y+=pasy;
                         p++;
                    }
               }
(6)            *tmp = *p;
               textout( screen, font, tmp, x, y, ( (++color)%255 )+1 );
               x+=pasx;
               rest( 10 );
(7)       }while( *p++ && !key[KEY_ESC] );
(8)       free( tmp );
     }
```

Fig. 16.14 : Affichage du texte à l'écran

(3) La variable x, position horizontale, est initialisée pour la marge gauche sur un huitième de l'écran. Pour la marge droite également avec la taille de l'écran moins un huitième. La marge en haut est deux fois la marge horizontale. Il n'y a pas de marge en bas mais les textes restent au centre de l'écran. Le pas d'avancement est de 8 pixels à l'horizontal. Il correspond à la taille de chaque lettre de la police utilisée (une police de base comme celle du « Bios » et fournie par l'environnement). Il est le double en vertical.

(4) (7) Boucle do-while. Le test d'arrêt est du type de celui vu avec la fonction précédente

« correction_majuscules » et comporte un test supplémentaire sur l'état de la touche Echap afin de pouvoir quitter le programme sans attendre la fin de cette boucle si demandé par l'utilisateur.

(5) Le premier point est de savoir dès le début de chaque mot s'il va tenir sur la ligne ou pas, et s'il faut passer à la ligne suivante en incrémentant y. Pour ce faire, à partir de chaque espace trouvé (valeur ASCII 32), le nombre de lettres est compté jusqu'à l'espace suivant ou la fin éventuelle du texte. C'est le rôle des variables « i » et « c » dans la boucle for. Une fois ce nombre obtenu, il est multiplié par le pas d'avancement, additionné à la position horizontale courante et le résultat est comparé à la borne horizontale droite. S'il est supérieur, il faut passer à la ligne c'est-à-dire augmenter y de pasx et faire revenir la position horizontale sur la marge de gauche.

(6) Quoiqu'il en soit, la lettre courante est affectée à la première position de la chaîne tmp. Ensuite appel de la fonction « textout » qui peut afficher une chaîne de caractère dans une bitmap. Cette fonction prend plusieurs arguments et voici son prototype :

```
void textout(BITMAP *bmp, const FONT *f, const char *s, int x, y, int color);
```

Le premier argument est la bitmap de référence, le second la police utilisée, le troisième la chaîne de caractère à afficher, les quatrième et cinquième la position initiale et le dernier la couleur. Pour notre appel la fonction prend la bitmap écran, screen, la fonte « font » par défaut, la chaîne de caractère tmp c'est-à-dire la lettre courante à la position (x, y) et avec une couleur qui évolue de 1 à 255 et selon la palette.
Une fois cet appel effectué et la lettre affichée, la position en « x » est incrémentée de pasx. Un temps est ajouté afin de voir le mouvement d'affichage.

(8) Après fin de la boucle l'espace mémoire alloué à la variable tmp est libéré.

4. Vers des curiosités robographiques : un robographe organisationnel

Il s'agit d'un germe de création réalisé sur la base d'un programme original de Robin Fercoq : « Volpurna ». Le principe fondamental du programme est extrait et restitué sous une forme minimaliste, ce qui en fait une source d'inspiration et de réflexion possible, en accord avec son auteur[1] qui met son propre code source à disposition du public.

[1] Le logiciel original ainsi que sa présentation sont actuellement disponibles à l'adresse : http://www.teleferique.org/stations/Fercoq/Archives/Volpurna/

4.1 Principe

La présentation faite par l'auteur est la suivante :

> « Volpurna simule l'agitation de particules en interaction. Chaque particule peut choisir soit de suivre son propre déplacement rectiligne soit d'adopter le mouvement de la particule la plus proche au-delà d'une certaine distance. La probabilité de suivre plutôt son « instinct » ou plutôt le mouvement des autres est réglée par l'utilisateur, de même que la distance minimum d'interaction. En manipulant ces paramètres globaux et leur variabilité inter-individuelle il est possible d'explorer continuellement différents régimes collectifs, moléculaires, chaotiques, individualistes ou totalitaires »

L'auteur s'interroge sur l'articulation entre mouvements individuels et mouvements collectifs. De ce fait la nature exacte du logiciel est hybride. C'est un travail qui n'est réservé ni au visuel, ni à la vie artificielle, ni à la physique ou aux mathématiques, il prend des accents éventuellement politiques, philosophiques et critiques… Le modèle du fonctionnement est le suivant :
Il y a une population d'objets capables de se déplacer et sensibles à l'attraction des objets alentours en fonction de la distance ou de la proximité. La sensibilité à l'attraction collective est dosée avec le mouvement horizontal de la souris. Totalement à gauche le collectif supprime toute possibilité de mouvement individuel. Totalement à droite les objets n'ont aucune interaction. Le mouvement vertical de la souris contrôle la distance des interactions possibles. Dans les deux cas toutes les graduations intermédiaires sont permises.

4.2 Mise en forme et initialisation

La structure de données est la suivante : un objet est défini par une structure dont les champs correspondent aux paramètres de ses propriétés (position, déplacements, couleur etc.). Le nombre des objets est défini par une macro et tous les objets sont rangés dans un tableau sous la forme d'une structure (figure 16.15) :

Fig. 16.15 : Définition d'un tableau de structures pour l'ensemble des objets

```
(1)  #define NB_OBJ          300

(2)  struct{

(3)       int x, y;
(4)       int mdx,mdy;
(5)       int adx,ady;
(6)       int resdx,resdy;
(7)       int mcol,acol,rescol;

(8)  }obj[NB_OBJ];
```

(1) La macro NB_OBJ donne un nombre d'objets qui sont des structures.

(2) (8) La structure pour les objets est définie au moment de la déclaration du tableau

« obj » de la taille de NB_OBJ. Elle n'a pas de nom et ne sera à aucun moment nécessaire en dehors du tableau.

(3) Les variables « x » et « y » déterminent la position de l'objet à l'écran.

(4) « mdx » et « mdy » sont les variables pour le vecteur dit « maître » à savoir la direction par défaut que prend l'entité lorsqu'elle n'est soumise à aucune influence.

(5) « adx » et « ady » correspondent à la direction courante d'une entité soumise ou non à influence.

(6) « resdx » et « resdy » sont des variables qui stockent des valeurs intermédiaires de directions pendant le calcul des influences mutuelles entre entités.

(7) Schéma identique à celui du déplacement pour la couleur avec « mcol » pour la couleur maîtresse, « acol » pour la couleur courante et « rescol » pour les calculs intermédiaires.

En dehors du tableau de structures quelques macros habituelles, pour la taille de l'écran et l'obtention d'un nombre pseudo-aléatoire avec un modulo.

```
#define ECRAN_X      800
#define ECRAN_Y      600
#define X2           ( ECRAN_X/2 )
#define Y2           ( ECRAN_Y/2 )
#define irand( n )   ( rand( )%( n ) )
```

La fonction d'initialisation

A l'aide d'une boucle « for » chaque champ reçoit une valeur aléatoire. Cette valeur est puisée dans la fourchette de l'écran pour la position, dans une fourchette entre -5 et 5 pour les vecteurs, et dans la fourchette de 1 à 255 pour la couleur (figure 16.16).

```
void initialisation(void)
{
int n;
    for(n=0;n<NB_OBJ;n++){
        obj[n].x=irand(ECRAN_X);
        obj[n].y=irand(ECRAN_Y);
        obj[n].mdx=obj[n].adx=irand(11)-5;
        obj[n].mdy=obj[n].ady=irand(11)-5;
        obj[n].mcol=obj[n].acol=irand(255)+1;
    }
}
```

Fig. 16.16 : Des valeurs initiales pour chaque objet

4.3 Processus générateur

Après création d'une palette et création d'une bitmap pour un affichage en double buffering, la fonction d'initialisation est appelée. Ensuite vient la boucle motrice avec pour test d'arrêt la pression sur la touche Echap. L'action se décompose en deux fonctions, sans appel à des sous fonctions hormis les fonctions d'affichage de l'environnement (figure 16.17).

```
(…)
faire_palette(gPal);
bmp=create_bitmap(ECRAN_X,ECRAN_Y);

initialisation();
while(!key[KEY_ESC]){
      calcul();
      affiche (bmp);
}
```

Fig. 16.17 : La boucle du processus générateur après initialisation

Détail de la fonction « calcul »

Le premier point de l'action consiste à trouver pour chaque entité celle qui est la plus proche, compte-tenu de la position verticale de la souris. Pour calculer la distance entre deux points dans l'écran le théorème de Pythagore est utilisé. Avec la distance horizontale et la distance verticale entre chaque entité, on finit par trouver l'hypoténuse la plus courte. Le second point, selon cette fois la position horizontale de la souris, est, soit de prendre comme résultat les valeurs du vecteur maître de l'entité en question, soit les valeurs du vecteur courant de celle qui lui est la plus proche (figure 16.18) :

(1) La fonction « calcul » ne renvoie pas de valeur et n'a pas de paramètre. Elle utilise quelques variables locales, des entiers.

(2) La première boucle for donne accès à chaque entité.
La première instruction est un test sur la position horizontale. La variable « mouse_x » fait partie de l'environnement Allegro et donne la position horizontale courante de la souris à l'écran ; la position verticale est de même donnée par la variable « mouse_y ». Si la valeur renvoyée par la fonction « rand » avec un modulo sur la taille de l'écran est inférieure à la position horizontale de la souris, les instructions du bloc sont effectuées.
Les instructions consistent à prendre comme résultat pour l'entité « n » ses propres valeurs maîtresses de déplacement et de couleur. Ainsi plus la souris sera du côté droit plus la probabilité d'exécuter les instructions sera grande, à savoir que l'entité soit dirigée par ses propres valeurs maîtresses. C'est ce qui exprime la suprématie de l'individuel sur le collectif.
L'instruction « continue » fait ensuite passer à l'entité suivante, le reste des instructions étant devenu inutile (pas d'influence).
Inversement, plus la souris est à gauche moins le bloc est accessible, et il ne l'est pas du tout avec « mouse_x » égale à 0, ce qui exprime les degrés de suprématie du collectif sur l'individuel.

(3) Dans le cas où le bloc précédent n'est pas exécuté, il s'agit de trouver l'entité la plus proche de l'entité n parmi les autres. La variable locale « mindist » est initialisée sur le carré de la distance horizontale la plus grande, la taille de l'écran. Chaque distance plus petite trouvée sera stockée à la place de la précédente de façon à ce que reste la plus petite à la fin.
Par ailleurs la variable « resobj » destinée à conserver le numéro d'indice de l'entité trouvée la plus proche est initialisée en dehors d'un indice de tableau afin éventuellement de signifier qu'aucune entité n'a été trouvée.

(4) Chaque entité regarde toutes les autres. C'est la raison d'être de la deuxième boucle for. Chaque entité « n » regarde toutes les autres « i ». D'ailleurs la première instruction consiste à passer i si i et n sont la même entité.

```
(1)  void calcul(void)
     {
     int dx, dy, n, i;
     int resobj, dist, mindist;

(2)        for ( n=0 ; n < NB_OBJ; n++ ){

                if(irand(ECRAN_X)<mouse_x){
                     obj[n].resdx=obj[n].mdx;
                     obj[n].resdy=obj[n].mdy;
                     obj[n].rescol=obj[n].mcol;
                     continue;
                }
(3)             mindist=ECRAN_X*ECRAN_X;
                resobj=-1;

(4)             for ( i = 0 ; i < NB_OBJ ; i++ ){
                     if ( i == n )
                          continue;
(5)                  dx = abs ( obj[ n ].x - obj[ i ].x );
                     if ( dx > X2 )
                          dx = ECRAN_X - dx;

                     dy = abs ( obj[ n ].y - obj[ i ].y );
                     if( dy > Y2 )
                          dy = ECRAN_Y - dy;

(6)                  dist = dx*dx+dy*dy;
                     if( dist <= mindist && dist > mouse_y*mouse_y ){
                          resobj = i;
                          mindist = dist;
                     }
                }
(7)             if ( resobj >= 0 ){
                     obj[ n ].resdx = obj[ resobj ].adx;
                     obj[ n ].resdy = obj[ resobj ].ady;
                     obj[ n ].rescol = obj[ resobj ].acol;
                }
           }
     }
```

Fig. 16.18 : Les relations entre les entités

(5) Il s'agit de calculer la distance de l'entité n à chaque entité i. La distance horizontale est donnée par la soustraction des coordonnées horizontales de chaque entité, de même pour la distance verticale avec les coordonnées verticales. Puisqu'il s'agit de comparer les distances indépendamment des orientations, ce sont les valeurs absolues de l'objet n à l'objet i pour l'horizontale et la verticale qui sont retenues. Précisons que l'espace de l'écran est considéré comme circulaire. Les bords se rejoignent et se touchent. Sortir à

gauche fait rentrer à droite et réciproquement, et de même entre haut et bas. Si donc la distance trouvée est supérieurs à la moitié de l'écran c'est que les deux entités sont plus proches en passant « par derrière ». Cette nouvelle distance est obtenue en soustrayant l'ancienne à la distance maximum qui est la taille de l'écran. Ces aspects sont identiques pour les distances horizontale et verticale.

(6) La variable « dist » stocke le carré de l'hypoténuse et elle est suivie d'un test à deux conditions. Tout d'abord si la valeur de dist est inférieure à la valeur stockée dans mindist c'est que l'on a trouvé plus près. Mais également il y a une autre contrainte qui varie avec la position verticale de la souris.
La variable mouse_y donne la position verticale courante de la souris à l'écran. Ainsi il ne suffit pas que la distance entre deux entités soit plus courte il faut également qu'elle soit supérieure au carré de la position verticale de la souris. Il s'agit de trouver l'entité la plus proche après cette distance là.
Si tel est le cas, alors les instructions sont de mémoriser la valeur de l'indice de l'entité i dans la variable resobj et de conserver cette distance trouvée comme la nouvelle plus petite distance.

(7) Une fois la distance de l'entité n à toutes les entités i comparées, le résultat, l'entité i la plus proche de n se trouve dans la variable resobj à moins qu'aucune entité n'est été trouvée, ce qui est possible si « mouse_y » est très élevé avec la souris en permanence vers le bas de l'écran. Dans ce cas resobj est toujours égale à -1. Dans le cas contraire, c'est la confrontation entre les deux entités.
L'entité n qui regarde les autres prend les valeurs courantes de l'entité i la plus proche d'elle comme résultat dans les champs prévus de la structure.

Détail de la fonction « affiche »

L'affichage est assez sobre. Les entités sont réduites à de petits carrés (figures 16.19) :

```
(1)  void affiche (BITMAP*bmp)
     {
     int n;
          clear(bmp);
(2)       for ( n = 0 ; n < NB_OBJ ; n++ ){
               obj[ n ].adx=obj[ n ].resdx;
               obj[ n ].ady=obj[ n ].resdy;
               obj[ n ].acol=obj[ n ].rescol;

(3)            obj[ n ].x = ( obj[ n ].x+obj[ n ].adx+ECRAN_X )%ECRAN_X;
               obj[ n ].y = ( obj[ n ].y+obj[ n ].ady+ECRAN_Y )%ECRAN_Y;

(4)            rectfill ( bmp, obj[ n ].x, obj[ n ].y, obj[ n ].x+5, obj[ n ].y+5,obj[ n ].acol );
          }
(5)       vline ( bmp,mouse_x, 0, ECRAN_Y-1, 255 );
          hline ( bmp, 0, mouse_y, ECRAN_X-1,255 );
          blit ( bmp, screen, 0, 0, 0, 0, ECRAN_X, ECRAN_Y );
     }
```

Fig. 16.19 : Affichage final des entités, ainsi que la mire pour la position de la souris.

(1) La fonction « affiche » ne renvoie pas de valeur. Elle a un pointeur BITMAP* comme paramètre. C'est l'image bitmap de référence en mémoire avant copie dans la mémoire vidéo pour l'écran. Le premier appel de fonction consiste à effacer le contenu de cette bitmap.

(2) Boucle for qui passe en revue toutes les entités. Pour chacune d'entre elles le résultat du calcul est copié dans la valeur courante pour les variables de déplacement et de couleur.
(3) La nouvelle position (x, y) est calculée en respectant la circularité de l'écran.

(4) Affichage de l'entité sous la forme d'un petit carré de cinq pixels de côté de la couleur courante.

(5) Après la boucle et l'affichage des entités, affichage de la position souris à travers une ligne horizontale pour la position horizontale et une ligne verticale pour la position verticale. Enfin placage à l'écran du résultat avec appel à la fonction « blit ».

CONCLUSION

Jouer d'un langage de programmation

(…) sans prétendre clore la discussion (dont nous venons d'esquisser les grandes lignes) par une décision définitive et absolue, nous pouvons risquer cette proposition : « Au commencement était l'action. »

Sigmund Freud

Dans la première partie de cet ouvrage nous avons exposé notre problématique de création : de façon directe et simple il s'agit de programmer pour le plaisir comme on fait de la musique. Cette première partie manifeste la dynamique de l'idée avec les positionnements et les orientations qui en découlent. Elle s'appuie sur un ensemble de tâtonnements expérimentaux. L'important est la construction de l'action avec tout ce qu'il y a d'engagement ontologique. Dans la seconde partie, c'est le passage à l'acte proprement dit avec l'exposition de réalisations. Ces réalisations sont encadrées par deux espaces de réflexion qui en sont issus : une phénoménologie des programmes et l'objectif de développer un imaginaire sur la base du matériau informationnel avec le concept de robographe. La troisième partie a trait à l'élaboration théorique de la pratique qui progresse entre investissement poétique et maîtrise de la matière informationnelle : l'espace robographique proprement dit.

Articuler idée et action

Cheminement individuel, modèle ouvert

Tout au long de l'ouvrage il s'agit d'un cheminement. Un certain nombre des points abordés au début sont retrouvés à la fin selon un angle de vue qui s'est modifié. Par exemple les notions de projet et de modélisation qui réapparaissent enrichies de l'expérience robographique au chapitre Quinze. Pour autant, le développement proposé ne correspond pas à la réalité chronologique de la découverte expérimentale, et les différentes étapes présentées sont de pures représentations qui mettent en scène le processus de recherche et d'évolution que nous avons voulu exprimer.

En effet, c'est un principe évolutif de la conceptualisation qu'il nous a paru finalement important de former, et non un modèle de démarche ou une méthode. Toutefois, il y a bien modélisation de la démarche de créativité et elle n'est pas laissée au hasard. Une méthodologie lui est associée afin d'assurer sa survie et son développement. Disons qu'il s'agit d'un « modèle ouvert »[1].

Exposer une pratique de la créativité est revenu à comprendre un devenir, une mise en marche. C'est pourquoi nous avons souhaité retrouver dans la structuration même de notre développement, ce principe du devenir à l'oeuvre. Ainsi à partir du chapitre Huit, avec l'action de développer un imaginaire, l'organisation des chapitres suivants s'est faite essentiellement par un processus de scissiparité ; chaque chapitre est né au sein du chapitre précédent. D'où l'idée de l'oignon dont Didi Huberman dit que « le dehors n'est qu'une mue du dedans » [DIDI-HUBERMAN, 2000, p.20].

[1] Cette idée de « modèle ouvert » peut être rapprochée du principe d'un « univers de finalités ouvertes » tel qu'il est décrit par Israël Kirzner pour le domaine économique [KIRZNER, 1992, avant-propos p. XV].

Acte de conscience

A partir de l'expérience d'un devenir, fondement d'une pratique, nous avons relié notre prise de conscience de l'espace robographique au domaine du langage poétique élargi à des processus de programmation informatique.

Si nous reprenons les termes de Gaston Bachelard, nous avons fait le pari qu'un langage de programmation pris dans une volonté poétique pouvait être relié avec « l'acte de conscience » :

> « Cet acte, nous ne l'étudierons que dans le domaine du langage, plus précisément encore, dans le langage poétique, quand la conscience imaginante crée et vit l'image poétique. Augmenter le langage, créer du langage, valoriser le langage, aimer le langage. Voilà autant d'activités où s'augmente la conscience de parler. » [BACHELARD, 1960, 1999, p. 5].

En fait, nous nous sommes intéressés à l'articulation de l'idée et de l'action, ainsi qu'à leur confrontation, d'où, en exergue de cette conclusion, la citation de Sigmund Freud. C'est une parole qui clôt le livre « Totem et Tabou » en éveillant précisément cette question de « la séparation nette et tranchée que nous établissons entre pensée et action » [FREUD, 1913, 1965, p. 240].

Rencontrer le public

Or l'action peut être abordée comme ce qui nous relie aux autres. L'action individuelle n'est pas close sur elle-même. Dans le cadre de notre démarche, elle se prolonge en direction du public : chercheurs, artistes, amateurs, entrepreneurs ou spectateurs. Cette idée actuelle, qui pose la programmation non plus uniquement comme une activité scientifique mais aussi comme une activité d'écriture aux plans ludique, littéraire et artistique, permet de toucher le public selon plusieurs registres. Nous avons expérimenté plusieurs directions au cours de la recherche.

Toucher le public

En premier lieu, il y a la recherche associée à une volonté de transmettre la programmation comme moyen d'expression, mais aussi de recherche, une clef d'accès et de transversalité. Ce peut être à partir d'un point d'ancrage comme la programmation de jeux. L'espace de la programmation de jeux permet un travail technique, créatif et ouvert sur toutes sortes de thématiques et d'expérimentations. De plus il y a moyen d'adhérer à des projets techniques de développement informatique comme celui de la communauté Allegro. Ensuite, il y a le domaine des réalisations artistiques avec ses créneaux spécialisés et ses états d'âme particuliers. De là se présente le travail à la commande qui renvoie au monde de l'entreprise, notamment au domaine de la communication d'entreprise et du design publicitaire. Peut également être abordé le secteur de l'activité socioculturelle et des associations qui oscillent entre art et social. En gros, la programmation peut être traitée comme l'est la musique : recherche et technologie, réalisations et compositions, commandes d'œuvres ou d'illustrations dans une industrie, enseignement, animation socioculturelle.

Transmettre la programmation

L'étude d'un langage de programmation n'est pas une évidence. La matière informationnelle constituée de types, structures de données et algorithmes n'est pas une matière simple, c'est un agrégat de concepts. Les livres d'algorithmique parlent d'arbres, de listes, de graphes etc. sans expliquer pourquoi, c'est-à-dire sans montrer sérieusement dans quelles situations ils peuvent être utilisés et comment leur donner du sens.
On trouve le plus souvent des algorithmes de tri et les objectifs sont toujours implicites ; des objectifs peu intuitifs, difficiles à retrouver et à comprendre, comme une entrave à la transversalité des connaissances. A propos de la programmation et de l'algorithmique, la plupart des publications supposent de la part des lecteurs qu'ils ont déjà toutes les clefs d'accès (Plus souvent la maîtrise de l'anglais !) à savoir un projet dans lequel la programmation n'est plus qu'une formalité. Dans ces conditions la programmation n'est accessible toujours qu'aux mêmes personnes ayant une formation commune dans le domaine scientifique. Si l'on en juge par les publications ajoutons que les objectifs, tout implicites qu'ils sont, semblent demeurer identiques ou évoluer très lentement et que l'élargissement de la pratique de la programmation paraît difficile y compris pour ceux qui s'y attachent.

Pourtant se rapprocher de l'idée informatique et s'approprier le langage de cette expression ne manque pas d'intérêt. Comment fait-on pour comprendre le fonctionnement d'un programme, le déroulement d'un algorithme ? Quelles sont les difficultés intellectuelles avec lesquelles on peut avoir à faire ? Il n'y a pas d'emboîtement immédiat entre la réalité de ces formes et l'intellect. Une lecture est nécessaire. Une reconstruction intérieure jusqu'à ce que l'ensemble n'ait plus de zone d'ombre. Il ne s'agit pas d'acquérir des automatismes, il s'agit d'un apprentissage pour jouer de ces formes. On peut comparer avec l'apprentissage d'un instrument de musique ; ce n'est cependant pas le même plaisir, ce n'est pas la même nature d'énigme, ce n'est pas le dressage des mêmes muscles ; c'est aussi difficile et peut donner autant de joie.

Souvent il s'agit de jouer avec la matière de son incompréhension. Constater qu'un algorithme fonctionne et rester sans comprendre, c'est-à-dire rester dans l'impossibilité d'accéder à l'intuition qui permettrait d'en avoir l'idée, c'est éventuellement entrer dans le désir de programmer si l'on accepte le jeu de l'énigme. L'incompréhension stimule l'expérimental, la réalisation de tests. Elle précède la compréhension. C'est tout le rôle de l'énigme, avec la motivation et la joie qui l'accompagnent. Ce qui est sûr c'est qu'il y a un véritable plaisir de jouer.

Réalisations artistiques, « Démo », installation

La mise en scène de la visibilité de l'œuvre compte pratiquement pour l'œuvre elle-même dans le milieu artistique. C'est finalement ce qui lui donne son statut et bien souvent c'est celui-ci qui définit la notion d'œuvre. C'est dire à quel point les modalités de la visibilité de l'œuvre sont importantes dans le milieu artistique et imaginer une œuvre revient souvent à imaginer les modalités de sa visibilité.

En ce qui nous concerne nous avons participé à l'élaboration progressive d'un concept dans l'air du temps et qui fait son chemin depuis : le concept de « démo ». Voici un texte original du collectif d'artistes « Téléférique » dont nous avons contribué à démarrer les activités au cours de notre recherche :

> « Démo », ce projet est né d'une constatation évidente aujourd'hui. La notion d'exposition se trouve reconfigurée par les expériences récentes des œuvres sur le web qui induisent des modes de présentation différents de l'accrochage. Ainsi certains artistes produisent des programmes informatiques qui fonctionnent tant comme des outils que des œuvres : des matrices dont l'utilisation en live devient un mode de visibilité. Ce type d'œuvres numériques nécessite d'être manipulé, configuré sur une machine pour être montré. Nous appelons « DEMO » plutôt qu' « EXPO » leurs présentations. En parallèle au home studio où chacun travaille chez soi sur son ordinateur, la démo pour Téléférique fonctionne comme un lieu de rendez-vous, un échange entre les artistes du collectif, le public et les espaces d'accueil. (…)

Ce qui est important est la volonté de rencontre physique avec le public, l'œuvre apparaissant comme le prétexte de cette rencontre. Ce n'est pas seulement l'œuvre qui est montrée mais l'auteur en personne qui est là pour animer son œuvre et dialoguer avec le public. Cet aspect est vraiment intéressant, et très stimulant pour la réalisation des projets.

En ce qui nous concerne les propositions sont plus de l'ordre d'une certaine forme de poésie que de « l'Installation » qui souvent recouvre des moyens techniques plus importants. Il est vrai que cette poésie qui consiste essentiellement à donner du sens à son action nous importe beaucoup, et que nous chercherons certainement à la préserver s'il advenait que nous explorions des gammes de moyens comme la robotique et des systèmes interactifs à base de capteurs.

C'est ce qui est ressorti d'une action menée dans un jardin public à Belleville avec une association d'artistes « La Graine d'Or » lors de la Fête des Jardins. Nous avons mis en scène « les fleurs du multimédia ». Les fleurs, de vraies fleurs offertes finalement aux personnes, étaient présentées sur le mode graphique, sous forme d'images intégrées dans des structures de bois comme sur des tiges. Les images étaient issues d'un logiciel, une sorte de mandala numérique, qui a fonctionné en parallèle et à partir duquel bon nombre de discussions ont commencé. Ainsi étaient réunis en direct, quoique dans des lieux différents, un programme informatique, des images avec une installation plastique, et des fleurs véritables offertes finalement après une mise en scène sous forme de jeu qui permettait de circuler entre les trois espaces.

Contact avec l'entreprise

L'entrepreneur est un artiste en son genre. Ce qui l'intéresse surtout c'est la réalisation d'un projet et sa mise en place. Le profit n'est pour lui qu'une condition de viabilité du fonctionnement mais ce n'est pas nécessairement un but en soi. L'entrepreneur a autant de plaisir à monter quelque chose que l'informaticien à réaliser un programme ou le musicien à composer une œuvre. Nous avons rencontré une telle personne : José Moreno fondateur de plusieurs entreprises dans le domaine du design publicitaire et du graphisme. Voici dans les grandes lignes les différents points que nous avons commencés d'examiner ensemble.

Les possibilités expressives de l'écriture informatique, permettent de dégager deux orientations pour la conception et la production. Il y a d'une part la recherche artistique, avec notamment l'inscription d'œuvres dans l'architecture et l'urbanisme via des écrans plats géants, et d'autre part un prolongement avec la réalisation de produits publicitaires : écran de veille, carte de visite ou de vœux, jeux, tableaux etc. La liste des types de produits paraît ouverte et dépendre de l'imagination dans ce domaine à développer. Il reste à

envisager un outil de production et d'imaginer un profil d'informaticien « créatif », comme c'est déjà le cas des graphistes par exemple.

Galerie d'algorithmes scénaristiques

On discute avec un client d'un logiciel publicitaire pour lui. Derrière soi on a un ensemble de possibilités scénaristiques que l'on peut réinterpréter avec des situations spécifiques. S'il s'agit d'un travail sur le texte on cherche dans la génération de textes. On dispose déjà d'un ou plusieurs algorithmes, si l'idée retenue s'appuie sur un principe adaptatif ou de métamorphose on dispose d'un modèle de réseau de neurones ou d'un algorithme génétique ou encore d'un modèle personnel etc. Reste ensuite à décider, selon le type de produit retenu (écran de veille, jeu etc.) et avec le choix d'un algorithme fondamental, du scénario « de surface » et de son habillage selon les objectifs à atteindre ou à respecter, des éléments de la charte graphique et.

C'est-à-dire que face au client on dispose déjà d'un ensemble de possibilités scénaristiques informatiques et que l'on peut en imaginer la mise en scène dans telle ou telle situation. Ainsi, afin de ne pas partir de rien à chaque nouvelle commande il est important de disposer d'un fond de culture algorithmique et de l'enrichir continuellement. C'est un ensemble de germes de créations qui est à constituer progressivement.

Un cahier des charges technologique, un outil de développement

A la galerie d'algorithmes scénaristiques s'ajoute un environnement de programmation. C'est-à-dire l'outil de production à proprement parler. Les contraintes de base sont les mêmes que celles à l'œuvre pour les jeux. Disons qu'il faut que l'environnement de développement réunisse :
fiabilité maximum du produit, efficacité graphique et sonore, efficacité pour la gestion de tout ce qui a rapport aux interfaces standards et système (installation du logiciel, ouverture et sélection de fichiers, sauvegarde, impression éventuelles etc.), éventuellement la portabilité (Windows 98, Millenium, Nt, 2000, XP, Mac Os, Linux …), éventuellement également les possibilités de réseaux (pour des jeux en ligne ou des scénarios distribués sur plusieurs machines).

Contact avec le milieu associatif

Pendant la recherche, nous avons créé une association afin de promouvoir la pratique de la programmation. En fait cette action nous a permis de pénétrer le milieu associatif très riche et prolifique des quartiers de l'Est parisien. Essentiellement notre proposition de départ est d'envisager la pratique de la programmation informatique au même titre que la pratique d'un instrument de musique. Un des objectifs est d'initier à la programmation informatique des personnes de formation non nécessairement scientifique et technique.

Outre ces aspects de formation, le secteur associatif a permis également d'identifier quelques produits possibles. Par exemple la réalisation de tracts informatiques pouvant être projetés lors de manifestations publiques ou sous forme d'écrans de veille, c'est-à-dire le logiciel support de message éventuellement à caractère publicitaire. Egalement la notion de décor informatique, tout à fait passionnante à réaliser dans le cadre de spectacle ou pour l'animation de salles de spectacles.

Ce qui nous paraît le plus intéressant à l'issu de ces différentes expériences de rencontre, c'est la diversité des personnes rencontrées et avec elles des projets possibles. Dans chaque milieu rencontré, universitaire, artistique, entrepreneurial et associatif, il y a des personnes intéressées. C'est-à-dire que des passerelles sont possibles entre des domaines d'activités différents et généralement envisagés séparément les uns des autres. La programmation telle que nous l'avons envisagée dans ce travail permet de réunir des publics différents.

BIBLIOGRAPHIE

AHO Alfred, ULLMAN Jeffrey, *Concepts fondamentaux de l'informatique,* Dunod, Paris 1993.

AHO Alfer, SETHI Ravi, ULLMAN Jeffrey, *Compilateurs, principes, techniques et outils, cours et exercices,* Dunod, Paris 2000.

ALLIOT Jean-Marc, SCHIEX Thomas, *Intelligence artificielle et informatique théorique,* Cépaduès éditions, Toulouse 1993.

AUDIBERT Pierre, *Ordre et chaos, fractales, pavages,* Université de Paris 8, département Informatique, support de cours 2000.

AUDIBERT Pierre, *Algorithmes et programmation, applications en combinatoire, probabilités, théorie des nombres, géométrie,* Université de Paris 8, département Informatique, support de cours 1995, 1997, 1999.

AUROUX Sylvain, *La révolution technologique de la grammatisation,* Mardaga, Paris1994.

BACHELARD Gaston, *La poétique de la rêverie,* Presses Universitaires de France, Paris 1999, première édition 1960.

BACHELARD Gaston, *Le droit de rêver,* Presses Universitaires de France, Paris 1970.

BALMARY Marie, *La divine origine, Dieu n'a pas créé l'homme,* Grasset, Paris 1993.

BALMARY Marie, *Abel ou la traversée de l'Eden,* Grasset, Paris 1999.

BALPE Jean-Pierre, *L'art et le numérique*, Hermès, Paris 2000.

BENASAYAG Miguel, AKDAG Herman, SECROUN Claude, *Peut-on penser le monde ?,* Editions du Félin, Paris 1997.

BERTALANFFY, *Théorie générale des systèmes,* Dunod, Paris 1993, première édition 1968.

BOULEAU Nicolas, *Philosophie des mathématiques et de la modélisation, du chercheur à l'ingénieur,* Harmattan, Paris 1999.

BOURDIN Jean-Jacques, *De la droite discrète,* Université de Paris 8, département Informatique, support de cours 1996.

BRAQUELAIRE Jean-Pierre, *Méthodologie de la programmation en C,* Masson, Paris 1998.

BRET Michel, *Images de synthèse,* Dunod, Paris 1988.

BRETON Philippe, *Une histoire de l'informatique,* Seuil, Paris 1990.

CAILLAUD Bernard, *La création numérique visuelle, aspect du Computer Art depuis ses origines,* éditions Europia, Paris 2001.

CAGE John, *Je n'ai jamais écouté aucun son sans l'aimer : le problème avec les sons, c'est la musique*, édition La main courante, La Souterraine, France 1998.

CAPLAT Guy, *Modélisation cognitive et résolution de problèmes,* Presses polytechniques et universitaires romandes, Lausanne 2002.

CARDON Alain, *Conscience artificielle et systèmes adaptatifs,* Eyrolles, Paris 2000.

CHAZAL Gérard, *Le miroir automate : introduction à une philosophie de l'informatique,* Champ Vallon, Paris 1995.

CLANCY Michael, LINN Marcia, *Conception d'un programme, études de cas en Pascal,* Dunod, Paris 1993.

CHARBONNEL Jacquelin, *Langage C, les finesses d'un langage redoutable,* Armand Colin, Paris 1992.

DANCE Peter, *Faux animaux, escroqueries et mystifications,* Pierre Horay, Paris 1978.

DEJOURS Christophe, *Souffrance en France, la banalisation de l'injustice sociale,* Seuil, Paris 1998.

DELEUZE Gille, *Critique et clinique,* Minuit, Paris 1993.

DELEUZE Gille, GUATTARI Félix, *L'anti-Œdipe,* Minuit, Paris 1972.

DIDI-HUBERMAN Georges, *Etre crâne, lieu, pensée, sculpture,* éditions de minuit, Paris 2000.

DRILLON Jacques, *Traité de la ponctuation française,* Gallimard, Paris 1991.

DROUILLON Frédéric, *Exercer un art systémique à travers la narration informatique,* mémoire de D.E.A Information et Communication option Créations Multimédias, Université de Paris 8 Vincennes Saint-Denis, 1998.

DROUILLON Frédéric, "Narrativité autonome sonore et visuelle : la Vénus du mythe des races métalliques" in CNRS, Laboratoire de Mécanique et d'Acoustique, *Journées d'Informatique Musicale 98, textes des publications,* publication du LMA n°148, Mai 1998, ISBN 2-909669-12-2.

DROUILLON Frédéric, *Musicalité d'une morphogénèse appliquée,* Maîtrise sciences et techniques Information et Communication option Hyperdocuments et Multimédias, Université de Paris 8 Vincennes Saint-Denis, 1997.

DROUILLON Frédéric, RIDEAU Stéphane, " Sculpture informatico-musicale " in CNRS – URA 1526 Groupe de Recherche en Informatique, Image, Instrumentation de Caen, *Troisièmes Journées d'Informatique Musicale JIM 96,* Les cahiers du GREYC n°4, Caen 1996.

FEYERABEND Paul, *Contre la méthode, esquisse d'une théorie anarchiste de la connaissance,* Seuil, Paris 1979.

FREUD Sigmund, *Totem et tabou,* Payot, Paris 1965, première édition 1913.

FROIDEVAUX Christine, GAUDEL Marie-Claude, SORIA Michèle, *Types de données et algorithmes,* Ediscience international, Paris 1993.

GANASCIA, Jean-Gabriel, *L'âme-machine, les enjeux de l'intelligence artificielle,* Seuil, Paris 1990.

GIRARD Jean-Yves, *La machine de Turing,* Seuil, Paris 1995.

GOODY Jack, *La raison graphique, la domestication de la pensée sauvage*, Minuit, Paris 1979.

HEUDIN Jean-Claude, *La vie artificielle,* Hermès, Paris 1994.

HOLTZ-BONNEAU Françoise, *L'image et l'ordinateur,* Aubier, Paris 1986.

HUYGHE François-Bernard, *La langue de coton,* Robert Laffont, Paris 1991.

KANDINSKY Wassily, *Point et ligne sur plan,* Gallimard, Paris 1991, première édition 1926.

KIRZNER Israël, " La publicité dans un univers de finalités ouvertes " in Ekelund Robert, Saurman David, *Publicité et économie de marché,* Editions Litec, Paris 1992.

KERNIGHAN Brian, RITCHIE Dennis, *Le langage C,* Masson, Paris 1995, première édition 1978.

KERNIGHAN Brian, PIKE Rob, *La programmation en pratique,* Vuibert Informatique, Paris 2001.

KISSELEVA Olga, *Cyberart, un essai sur l'art du dialogue,* Harmattan, Paris 1998.

KLEE Paul, *Théorie de l'art moderne,*Denöel, Paris 1985, première édition 1956.

KLEIN Etienne *L'unité de la physique,* Seuil, Paris 2000.

KOESTLER Arthur, *Le cri d'Archimède,*Calman-Levy, France 1965.

LALANDE André, *Vocabulaire technique et critique de la philosophie,* PUF, Paris 1993.

LAPOINTE Pierre-Nicolas, *Pont entre C et C++,* AddisonWesley France, Paris 1995.

LEGER Fernand, *Fonctions de la peinture,* Gallimard, Paris 1997, première édition 1965.

LEMOIGNE Jean-Louis, *La modélisation des systèmes complexes,* Dunod, Paris 1995.

LESBROS Vincent, *Atelier Incrémentiel pour la Musique Expérimentale,* Thèse de Doctorat, Université de Paris 8, Département Informatique, 1995.

LIPSCHUTZ Seymour, *Mathématiques pour informaticiens,* McGraw-Hill International, Angleterre 1983.

LONCHAMP Jacques, *Les langages de programmation, concepts essentiels, évolution et classification,* Masson, Paris 1989.

MERCIER Philippe, *Assembleur, une découverte pas à pas,* Marabout Informatique, Alleur (Belgique) 1989.

MOLES Abraham, *Art et ordinateur,* éditions Blusson, Paris 1990.

MORIN Edgar, *Science avec conscience,* Seuil, Paris 1990.

OSTOLAZA Julio Fernandez, BERGARECHE Alvaro Moreno, *La vie artificielle,* Seuil, Paris 1997.

PEREC George, *La vie mode d'emploi,* Hachette, Paris 1978.

PIGNON Dominique, " Les machines molles de Von Neumann " in John Von Neumann, *L'ordinateur et le cerveau,* réédition chez Flammarion, Paris 1996.

RENNARD Jean-Philippe, *Vie artificielle, où la biologie rencontre l'informatique,* Vuibert, Paris 2002.

RIDEAU Stéphane, *Morphogenèse des contenus organisationnels des hyperdocuments,* Thèse de Doctorat, Université de Paris 8, Département Hypermédia, 1999.

ROADS Curtis, *L'audionumérique,* Dunod, Paris 1998.

SCHWACH Victor, *Micropsychologie des rapports homme/machine dans la vie quotidienne,* Thèse de Doctorat, Université des Sciences Humaines de Strasbourg, Laboratoire de Sociologie de la Culture Européenne, 1995.

SEARLE John, *L'intentionalité, essai de philosophie des états mentaux,* Editions de Minuit, Paris 1985.

SEDGEWICK Robert, *Algorithmes en langage C,* InterEditions, Paris 1991.

SERRES Michel, *Hermés V, Le passage du Nord-Ouest,* Minuit, Paris 1980.

SIMON Herbert, *Sciences des systèmes, sciences de l'artificiel,* Dunod, Paris 1991, première édition 1969.

SIMONDON Gilbert *Du mode d'existence des objets techniques,* Aubier, Paris1989, première édition 1958.

SOURIAU Etienne, *Vocabulaire d'esthétique,* Presses Universitaires de France, Paris 1990.

TODOROV Tzvetan, *Qu'est-ce que le structuralisme ? _ Tome 2 Poétique*, Seuil, Paris 1968.

VACQUIN Monette, *Main basse sur les vivants,* Fayard, Paris 1999.

VERNANT Jean-Pierre, *Mythe et pensée chez les Grecs, étude de psychologie historique,* La Découverte, Paris 1996, première édition 1985.

VON NEUMANN John, *L'ordinateur et le cerveau,* Flammarion, Paris1996, première édition 1957.

VON NEUMANN John, *Théorie générale et logique des Automates,* Champ Vallon, Paris 1996, première édition New York 1951.

WATZLAWICK Paul, *Le langage du changement,* Seuil, Paris 1980.

WALLISER Bernard, *Systèmes et modèles, introduction à l'analyse de systèmes,* Seuil, Paris 1997.

WINOGRAD Terry, FLORES Fernando, *L'intelligence artificielle en question,* Puf, Paris 1986.