École doctorale IAEM - DEA Informatique de Lorraine

Automates cellulaires dans le plan hyperbolique, réalisation d'un outil de simulation et de visualisation.

Rapport de Stage de DEA

soutenance le 1^{er} Juillet 2003

Composition du jury :

Membres permanents :	Dominique Méry
	Didier Galmiche
	Noëlle Carbonell
Représentant de la filière :	André Schaff

Nicolas Noble
nicolas@nobis-crew.org
encadré par Francine Herrmann au LITA

Université de Metz, Faculté des Sciences, Île du Saulcy 57045 Metz Cedex 02

Table des matières

Liste des figures

Introduction

Le sujet de ce stage porte sur les automates cellulaires dans le plan hyperbolique. Un Automate Cellulaire est un tableau d'automates à états finis identiques appelés des cellules interconnectées par un réseau régulier et uniforme. Les automates cellulaires ont été largement étudiés depuis plusieurs années [5] et sont le plus souvent étudiés dans les espaces euclidiens de dimension 1, 2, ou 3 où les cellules sont respectivement interconnectés sur une ligne, un plan ou l'espace 3D. Dans le plan euclidien, un automate cellulaire peut être vu comme un pavage régulier de polygones identiques. On peut construire dans l'espace euclidien des pavages réguliers par exemple de triangles, d'hexagones, d'octogones mais pas de pentagones. Dans [9], Margenstern a défini un automate cellulaire sur un pavage du plan hyperbolique et a établi un certain nombre de propriétés théoriques, algorithmiques et de complexité relatives à cet automate.

L'objectif de ce stage est de réaliser un outil de simulation et de visualisation de cet automate. Cet objectif nécessitait de comprendre les automates cellulaires dans un premier temps, puis de comprendre et intégrer leur fonctionnement dans le plan hyperbolique, et en particulier il a fallu établir certaines propriétés dans la géométrie hyperbolique, ainsi que concevoir, utiliser et programmer des structures de données adaptées à cette géométrie.

La suite de ce rapport sera constituée ainsi :

- Dans le premier chapitre, nous introduirons la géométrie hyperbolique et les méthodes utilisées pour construire un pavage régulier pentagonal : la pentagrille.
- Le chapitre deux sera consacré à l'étude de quelques propriétés mathématiques du plan hyperbolique qui nous seront particulièrement utiles pour représenter graphiquement notre pavage.
- Dans le chapitre trois, nous décrirons en détail la structure de données utilisée pour coder, numéroter les pentagones, et pour s'orienter dans le plan hyperbolique : l'arbre de Fibonacci.
- Le chapitre quatre est dédié à la notion d'automates cellulaires.
- Enfin, le chapitre cinq décrit l'implémentation de l'outil de simulation réalisé ainsi que les premiers résultats d'expérimentation.

Chapitre 1

Introduction à la Pentagrille

1.1 Géométrie Hyperbolique

1.1.1 Introduction historique

La géométrie hyperbolique a été introduite par Lobatchevski [8] en 1826, lorsqu'il tenta de démontrer le cinquième postulat d'Euclide à partir des quatre premiers, qui dit que "deux droites parallèles ne se rencontrent jamais", ce qui se formule autrement par "une droite et un point étant fixés, il existe une droite et une seule passant par ce point et parallèle à la droite". Lobatchevski découvrit que si l'on ne prend pas ce postulat pour hypothèse, alors d'autres géométries, qui respectent elles-mêmes d'autres postulats, apparaissent, notamment la géométrie hyperbolique.

Dans cette géométrie un peu spéciale, on continue de conserver les quatres premiers postulats, en particulier par deux points passe une et une seule droite. En revanche, le cinquième postulat n'est plus respecté, comme nous le verrons plus tard.

1.1.2 Le modèle du disque de Poincaré

La géométrie hyperbolique est relativement peu exploitée dans le monde de l'informatique. Cependant, elle offre un modèle radicalement différent du plan euclidien communément utilisé comme support. Nous allons maintenant donner quelques explications sur cette géométrie, et notamment sur le modèle en disque de Poincaré qui se situe dans le disque unité. Il existe cependant d'autres modèles, mais qui ne nous intéressent pas pour ce sujet. Dans le reste de ce document, les mots "géométrie hyperbolique" feront référence au disque de Poincaré.

Nous utilisons le cercle unité comme espace, l'origine étant le centre du cercle, et l'infini se situant sur les bords du cercle, que nous appellerons "horizon". Les lignes sont des arcs de cercles qui sont perpendiculaires à l'horizon. Vous pouvez voir par exemple sur la figure 1.1 que p, l, et m sont des droites hyperboliques. Comme pour la géométrie Euclidienne, on définit une droite par deux points. Et par deux points ne passera jamais qu'une seule droite hyperbolique. En effet, la contrainte d'être perpendiculaire à l'horizon crée naturellement cette propriété.

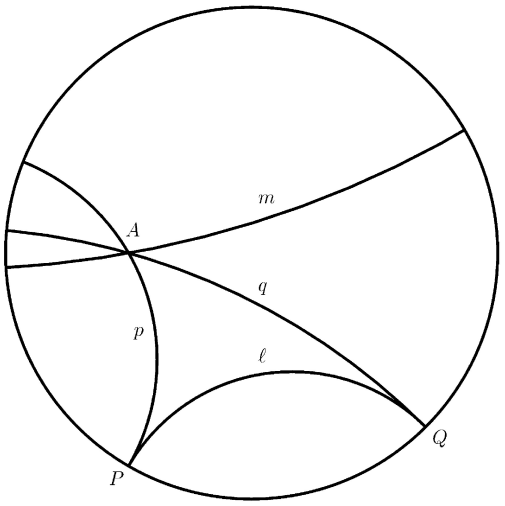

Figure 1.1 – Quelques droites hyperboliques

Nous pouvons noter quelques particularités dans la notion de géométrie hyperbolique, en particulier sur les droites parallèles. En effet, il nous faut revoir la définition de base que l'on connaît. Dans le plan, on sait que deux droites sont parallèles si elles ne se coupent jamais. En géométrie hyperbolique, ce n'est plus le cas. Deux droites sont parallèles si elles se rejoignent à l'infini, c'est-à-dire si elles possèdent le même point à l'horizon. Dans le cas de la figure 1.1, les droites q et l sont parallèles (elles ont le point Q en commun à l'horizon), ainsi que l et p (elles ont le point P en commun à l'horizon). Cependant, q et p ne sont **pas** parallèles, ce qui marque déjà une particularité par rapport au plan euclidien, où le parallèlisme est transitif.

L'autre particularité, c'est que contrairement au cinquième postulat d'Euclide, si l'on se donne une droite l et un point A, alors par A passe *deux* droites parallèles à l, et non plus une. On peut le voir toujours sur la figure 1.1 avec les droites p et q.

Enfin, il nous faut définir la réflexion dans cette géométrie, ce qui nous sera utile plus tard. Chaque droite est portée dans ce modèle par un arc de cercle euclidien, donc il est possible de déterminer le centre de ce cercle. Ensuite, on définit la réflexion hyperbolique : l'image M' d'un point M par rapport à une droite hyperbolique de centre O et de rayon R est tel que $\overrightarrow{OM}$ est colinéaire à $\overrightarrow{OM'}$, et $OM.OM' = R^2$. On peut voir en figure 1.2 un exemple de réflexion dans le plan hyperbolique tiré de la construction d'une œuvre d'Escher.

1.2 Définition de la Pentagrille

1.2.1 Construction

La géométrie hyperbolique va nous servir à construire la fameuse Pentagrille, qui nous servira ensuite de support pour notre automate cellulaire. La Pentagrille est un pavage régulier complet dont le symbole de

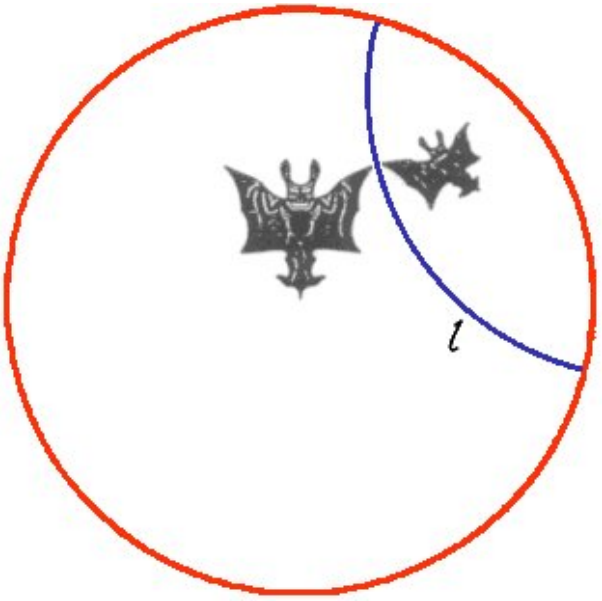

Figure 1.2 – Exemple de réflexion dans le plan hyperbolique : la petite chauve-souris est l'image de la grande par réflexion par la droite hyperbolique l

Schlafli est {5,4}, ce qui signifie que chaque noeud est entouré de 4 5-gones (c'est-à-dire, des pentagones) réguliers. Vous trouverez un exemple de la Pentagrille en figure 1.3.

Un pavage régulier et complet est un partitionnement de l'espace dans lequel on travaille, en utilisant une figure de base translatée ou réfléchie régulièrement afin de le recouvrir en entier. Chaque reproduction de la figure participe à son recouvrement total.

Dans le cas de la Pentagrille, nous partons de quatre pentagones réguliers (voir figure 1.4) et nous effectuons des réflexions de ceux-ci par rapport à leurs arêtes (voir figure 1.5 pour la première récursion). On continue à construire récursivement, en évitant bien évidemment de construire deux fois le même pentagone. Car, on peut voir sur la figure 1.5 que le pentagone $4,1$ est construit par la réflexion du pentagone $1,1$, ou par celle du pentagone $2,2$. Ainsi, l'affichage de la Pentagrille ne se fera jamais entièrement, mais seulement jusqu'à un certain niveau de récursion. De toute manière, rapidement, la récursion arrive à un niveau quasiment invisible sur l'écran, ce qui justifiera l'implémentation d'un zoom. Ainsi, sur la figure 1.3, la récursion a été effectuée sept fois.

Vous noterez que la numérotation est celle que nous allons conserver. Le premier chiffre désigne le numéro du pentagone dans le repère que nous allons construire, et le second désigne le quadrant du disque unité dans lequel ce pentagone réside. Ces quadrants sont numérotés de 1 à 4, dans le sens trigonométrique, en commençant par le quart nord-est.

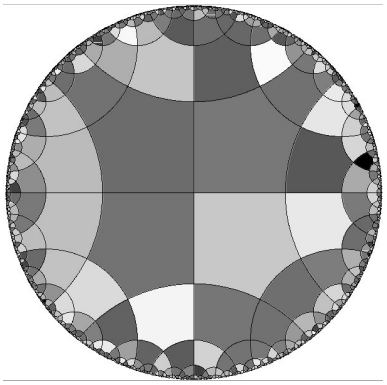

Figure 1.3 – La Pentagrille

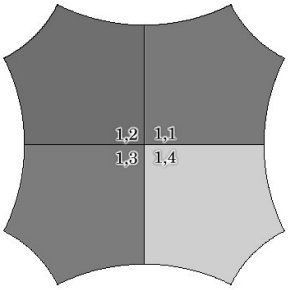

Figure 1.4 – Les 4 pentagones de départ

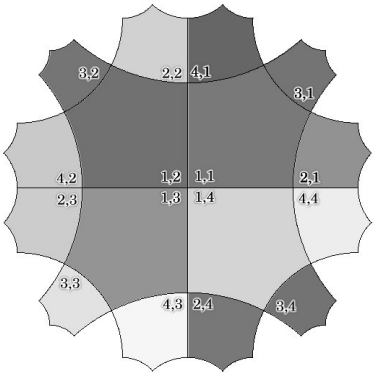

Figure 1.5 – La première récursion de la construction

1.2.2 Numérotation des pentagones

Pour identifier les pentagones, Margenstern dans [9] propose de les numéroter suivant une structure de données un peu spéciale : l'arbre de Fibonacci. Nous détaillerons cette structure un peu plus loin dans ce document. Par contre nous pouvons déjà voir la corrélation existant entre l'arbre de Fibonacci (voir figure 1.6) et la pentagrille numérotée (voir figure 1.7)

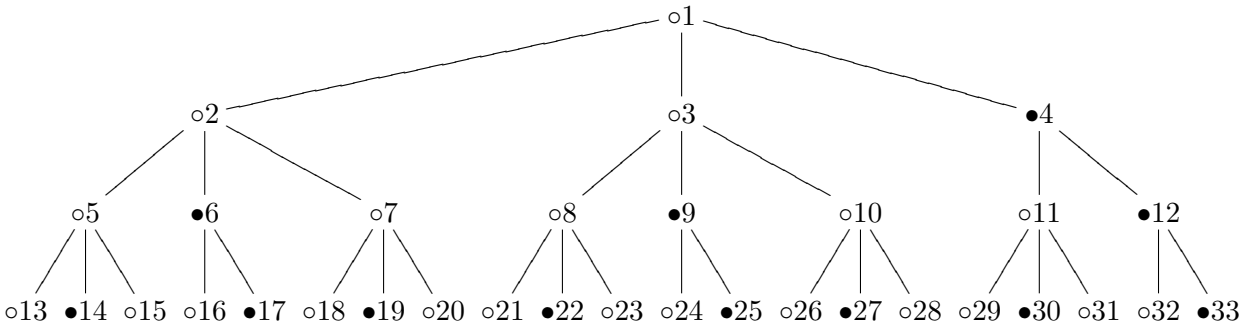

Figure 1.6 – Arbre de Fibonacci

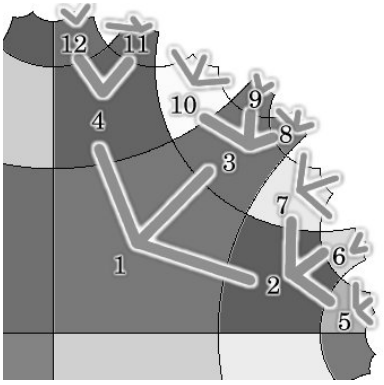

Figure 1.7 – Quart de Pentagrille superposé à l'arbre de Fibonacci

Il est important de noter tout de suite que l'arbre de Fibonacci ne coïncide qu'avec un seul quart de la Pentagrille. Ainsi, nous reproduirons la structure de données en quatre exemplaires afin d'obtenir une structure de données recouvrant totalement la Pentagrille, et de ce fait, nous avons besoin de repérer dans quel quart de la Pentagrille nous nous situons. C'est pourquoi nous numérotons les pentagones suivant un couple (n, q), n désignant le numéro du pentagone suivant l'arbre de Fibonacci, et q désignant le quart de la Pentagrille dans lequel il réside.

Nous avons donc dans ce chapitre expliqué comment construire un pavage pentagonal régulier du plan hyperbolique et comment nous pouvons numéroter chaque pentagone.

Chapitre 2

Quelques propriétés mathématiques du plan hyperbolique

Nous nous intéressons ici principalement aux propriétés nous permettant de représenter la pentagrille sur une sortie graphique.

Dans le cadre du travail dans le plan hyperbolique, nous noterons U le cercle unité de centre $(0, 0)$ et de rayon 1.

2.1 Notes générales

Dans le plan hyperbolique, les coordonnées du premier pentagone se calculent à partir des constantes suivantes :

$$x = \sqrt{\frac{cos\left(\frac{2\pi}{5}\right)}{1 + cos\left(\frac{2\pi}{5}\right)}}$$

$$u = \frac{x(1 + x^2)}{1 + x^4}$$

$$v = \frac{x(1 - x^2)}{1 + x^4}$$

Et donc, les 5 points du premier pentagone sont $O(0, 0)$, $A(x, 0)$, $B(u, v)$, $C(v, u)$, et $D(0, x)$.

2.2 Equation d'une droite hyperbolique

Nous allons discuter ici de la manière d'écrire l'équation d'une droite hyperbolique passant par deux points. Il faut noter qu'une droite dans le plan hyperbolique se représente par un arc de cercle dans le plan euclidien.

Rappel : l'équation d'un cercle $\mathcal{C}$ de centre Ω (ω_x, ω_y) s'écrit simplement :

$$(x - \omega_x)^2 + (y - \omega_y)^2 = R^2$$

que nous allons utiliser sous sa forme développée :

$$x^2 - 2\omega_x x + y^2 - 2\omega_y y = R^2 - \omega_x^2 - \omega_y^2$$

2.2.1 Equation d'une tangente à un cercle

Propriété:

L'équation de la tangente à un cercle $\mathcal{C}$ de centre Ω (ω_x, ω_y) en un point P $(x_p, y_p) \in \mathcal{C}$ est $x(\omega_x - x_p) + y(\omega_y - y_p) + x_p(x_p - \omega_x) + y_p(y_p - \omega_y) = 0$.

Demonstration:

Nous allons chercher l'équation de la tangente à un cercle $\mathcal{C}$ de centre Ω (ω_x, ω_y) en un point P (x_p, y_p).

- Nous savons que $P \in \mathcal{C}$, donc

$$x_p^2 - 2\omega_x x_p + y_p^2 - 2\omega_y y_p = R^2 - \omega_x^2 - \omega_y^2$$

- Le vecteur directeur d'un diamètre passant par P a pour coordonnées :

$$\overrightarrow{P\Omega} \begin{pmatrix} \omega_x - x_p \\ \omega_y - y_p \end{pmatrix}$$

- Le vecteur directeur d'une droite passant par P et un point M (x, y) a pour coordonnées :

$$\overrightarrow{PM} \begin{pmatrix} x - x_p \\ y - y_p \end{pmatrix}$$

- Calculons le produit scalaire de $\overrightarrow{P\Omega}$ avec $\overrightarrow{PM}$ afin de placer (PM) perpendiculairement à $(P\Omega)$:

$$\begin{aligned}
(\omega_x - x_p)(x - x_p) + (\omega_y - y_p)(y - y_p) &= 0 \\
\omega_x x - \omega_x x_p - x_p x + x_p^2 + \omega_y y - \omega_y y_p - y_p y + y_p^2 &= 0 \\
x(\omega_x - x_p) + y(\omega_y - y_p) + x_p^2 + y_p^2 - \omega_x x_p - \omega_y y_p &= 0 \\
x(\omega_x - x_p) + y(\omega_y - y_p) + x_p(x_p - \omega_x) + y_p(y_p - \omega_y) &= 0
\end{aligned}$$

□

2.2.2 Equation de la tangente à U en un point

Propriété:

L'équation de la droite tangente au cercle unité U en un point P (x_p, y_p) est $xx_p + yy_p = 1$

Demonstration:

Nous cherchons l'équation de la droite tangente à U en un point $P\ (x_p, y_p)$. Notons que P est sur le cercle U donc que $x_p^2 + y_p^2 = 1$. Il s'agit simplement d'un cas particulier de l'équation précédente et nous obtenons :

$$\begin{aligned} x(0 - x_p) + y(0 - y_p) + x_p(x_p - 0) + y_p(y_p - 0) &= 0 \\ xx_p + yy_p - x_p^2 - y_p^2 &= 0 \\ xx_p + yy_p &= x_p^2 + y_p^2 \\ xx_p + yy_p &= 1 \end{aligned}$$

□

2.2.3 Equation d'une droite hyperbolique

Propriété:

La droite hyperbolique $\mathcal{D}$ passant par les points $A\ (x_a, y_a)$ et $B\ (x_b, y_b)$ a pour équation, si $x_a y_b \neq x_b y_a$, $(x - w_x)^2 + (y - w_y)^2 = R^2$ avec

$$\begin{cases} \omega_x &= \dfrac{y_a(1 + x_b^2 + y_b^2) - y_b(1 + x_a^2 + y_a^2)}{2(x_a y_b - x_b y_a)} \\ \omega_y &= \dfrac{x_a(1 + x_b^2 + y_b^2) - x_b(1 + x_a^2 + y_a^2)}{2(x_a y_b - x_b y_a)} \\ R &= \sqrt{\omega_x^2 + \omega_y^2 - 1} \end{cases}$$

Demonstration:

Finalement, nous allons calculer l'équation d'une droite hyperbolique $\mathcal{D}$ passant par les points $A\ (x_a, y_a)$ et $B\ (x_b, y_b)$. Il s'agit en fait d'un cercle de rayon R et de centre $\Omega\ (\omega_x, \omega_y)$ (à déterminer). La particularité de ce cercle est qu'il est perpendiculaire à U, c'est-à-dire que les tangentes des deux cercles aux points d'intersections sont perpendiculaires. Notons $P\ (x_p, y_p)$ un des deux points d'intersections de $\mathcal{C}$ et de U.

$$\begin{aligned} A \in \mathcal{C} \Rightarrow \quad x_a^2 - 2\omega_x x_a + y_a^2 - 2\omega_y y_a &= R^2 - \omega_x^2 - \omega_y^2 \quad (1) \\ B \in \mathcal{C} \Rightarrow \quad x_a^2 - 2\omega_x x_a + y_a^2 - 2\omega_y y_a &= R^2 - \omega_x^2 - \omega_y^2 \quad (2) \end{aligned}$$

$$\begin{aligned} P \in U \Rightarrow \quad x_p^2 + y_p^2 &= 1 \\ \Rightarrow \quad y_p^2 &= 1 - x_p^2 \quad (3) \\ P \in \mathcal{C} \Rightarrow \quad x_p^2 - 2\omega_x x_p + y_p^2 - 2\omega_y y_p &= R^2 - \omega_x^2 - \omega_y^2 \quad (4) \end{aligned}$$

La tangente à $\mathcal{C}$ en P est perpendiculaire à la tangente à U en P. Calculons les vecteurs directeurs de, respectivement, la tangente à $\mathcal{C}$ en P et à U en P :

$$T_{\mathcal{C}} = \begin{pmatrix} \omega_y - y_p \\ -\omega_x + x_p \end{pmatrix}$$

$$T_U = \begin{pmatrix} y_p \\ -x_p \end{pmatrix}$$

D'où le produit scalaire de $T_{\mathcal{C}}$ et T_U :

$$\begin{array}{rrrcll} T_{\mathcal{C}} \perp T_U \Rightarrow & y_p(\omega_y - y_p) - x_p(-\omega_x + x_p) & = & 0 & \\ \Rightarrow & y_p^2 - y_p\omega_y + x_p^2 - x_p\omega_x & = & 0 & (5) \end{array}$$

Maintenant, développons nos diverses équations :

$$\begin{array}{rrll} (3) \rightarrow (4) & \Rightarrow & 1 - 2\omega_x x_p - 2\omega_y y_p = R^2 - \omega_x^2 - \omega_y^2 & (6) \\ (3) \rightarrow (5) & \Rightarrow & 1 - x_p^2 - y_p\omega_y + x_p^2 - x_p\omega_x = 0 & \\ & \Rightarrow & \omega_x x_p + \omega_y y_p = 1 & (7) \\ (7) \rightarrow (6) & \Rightarrow & R^2 - \omega_x^2 - \omega_y^2 = -1 & (8) \\ (8) \rightarrow (1) & \Rightarrow & x_a^2 - 2\omega_x x_a + y_a^2 - 2\omega_y y_a = -1 & \\ & \Rightarrow & \omega_x x_a + \omega_y y_a = \dfrac{1 + x_a^2 + y_a^2}{2} & (9) \\ (8) \rightarrow (2) & \Rightarrow & x_b^2 - 2\omega_x x_b + y_b^2 - 2\omega_y y_b = -1 & \\ & \Rightarrow & \omega_x x_b + \omega_y y_b = \dfrac{1 + x_b^2 + y_b^2}{2} & (10) \end{array}$$

Maintenant, (9) *&* (10) forment un système d'équations permettant de résoudre le couple d'inconnues (ω_a, ω_b). Avec des méthodes de résolutions classiques, on obtient :

$$\left\{ \begin{array}{rcl} \omega_x & = & \dfrac{y_a(1 + x_b^2 + y_b^2) - y_b(1 + x_a^2 + y_a^2)}{2(x_a y_b - x_b y_a)} \\ \omega_y & = & \dfrac{x_a(1 + x_b^2 + y_b^2) - x_b(1 + x_a^2 + y_a^2)}{2(x_a y_b - x_b y_a)} \end{array} \right.$$

□

Enfin, il nous reste à écrire le rayon R, qui vaut $\sqrt{\omega_x^2 + \omega_y^2 - 1}$ à partir de l'équation (8).

2.2.4 Cas particulier

Il est important de noter que cette propriété n'est vraie que pour $x_a y_b \neq x_b y_a$, c'est-à-dire pour A et B n'appartenant pas à un diamètre de U. Dans le cas contraire, il s'agit d'une droite classique dans le plan euclidien passant par les deux points A et B.

2.3 Réflexions dans le plan hyperbolique

Propriété:

Le point $P\ (x, y)$, a pour image $P'\ (x', y')$ par réflexion par rapport à la droite hyperbolique $\mathcal{C}$ de centre Ω et de rayon R, avec

$$\begin{cases} x' = \dfrac{(x - \omega_x)}{k} + \omega_x \\ y' = \dfrac{(y - \omega_y)}{k} + \omega_y \end{cases}$$

Demonstration:

Soit une droite hyperbolique $\mathcal{C}$ de centre Ω et de rayon R, et un point $P\ (x, y)$, avec $P \neq \Omega$. Soit la droite euclidienne D, diamètre de $\mathcal{C}$ passant par P.

Le point $P'\ (x', y')$, image de la réflexion de P par $\mathcal{C}$ est tel que :

$$P' \in D$$
$$|\Omega P| \times |\Omega P'| = R^2$$

Donc, Ω, P et P' sont alignés, et on a les rapports $\overrightarrow{\Omega P} = k\overrightarrow{\Omega P'}$ et $|\Omega P| = k|\Omega P'|$. Ce qui nous fait $|\Omega P| \times \dfrac{|\Omega P|}{k} = R^2$ et finalement, $k = \dfrac{|\Omega P|^2}{R^2} = \dfrac{(x - \omega_x)^2 + (y - \omega_y)^2}{R^2}$ On obtient enfin le vecteur $\overrightarrow{\Omega P'}$:

$$\overrightarrow{\Omega P} = \begin{pmatrix} x - \omega_x \\ y - \omega_y \end{pmatrix} \quad \overrightarrow{\Omega P'} = \begin{pmatrix} x' - \omega_x \\ y' - \omega_y \end{pmatrix}$$

$$\begin{cases} x' = \dfrac{(x - \omega_x)}{k} + \omega_x \\ y' = \dfrac{(y - \omega_y)}{k} + \omega_y \end{cases}$$

□

Nous venons de présenter des équations nécessaires pour l'implémentation de la représentation graphique du pavage pentagonal nommé Pentagrille.

Chapitre 3

Construction de l'arbre de Fibonacci

L'arbre de Fibonacci et un nouveau système de coordonnées décrits tous deux dans [9] par Margenstern vont nous être très utile pour naviguer à l'intérieur de cette Pentagrille. En effet, nous avons une structure bijective entre les noeuds de l'arbre de Fibonacci et chaque pentagone de la pentagrille. Nous allons voir cela plus en détail.

On ne construit ici qu'un seul arbre de Fibonacci, représentant un quart de la Pentagrille. Puis nous étendrons cette représentation à tout le disque unité.

3.1 Code de Fibonacci

3.1.1 Rappel

La suite de Fibonacci est la suivante :

$$\begin{cases} Fib(0) &= 1 \\ Fib(1) &= 1 \\ Fib(n) &= Fib(n-1) + Fib(n-2)\ \forall n > 2 \end{cases}$$

Voici les quelques premiers nombres de cette suite :
1, 1, 2, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

3.1.2 Codage

Une propriété bien connue, est que tout nombre n peut s'écrire de manière unique comme combinaison linéaire de nombres de Fibonacci à coefficients binaires et n'ayant pas deux coefficients successifs égaux à 1. Il est possible de calculer ce codage d'un nombre n quelconque suivant la méthode : on prend le plus grand nombre de Fibonacci F n'excédant pas n, on émet un 1 et on calcule $n' = n - F$. Puis on passe au nombre de Fibonacci précédant en émettant des 0 tant que ceux-ci sont supérieurs à n'. Dès que l'on retrouve un nombre de Fibonnaci inférieur, on émet un 1 et on recommence la manoeuvre jusqu'à ce que l'on atteigne $Fib(1)$. Exemple :

$$200 = 1 \cdot 144 + 0 \cdot 89 + 1 \cdot 55 + 0 \cdot 34 + ... + 0 \cdot 2 + 1 \cdot 1$$
$$200 = 1 \cdot Fib(10) + 0 \cdot Fib(9) + 1 \cdot Fib(8) + 0 \cdot Fib(7) + ... + 0 \cdot Fib(2) + 1 \cdot Fib(1)$$

donc le codage de Fibonacci de 200 sera

1010000001

Dans le reste de ce document, nous noterons $Rep(n)$ la représentation binaire en code de Fibonacci de n, et $\#Rep(n)$ la taille de cette représentation.

3.1.3 Propriétés

Propriété:

En suivant la méthode décrite précédemment, le codage de Fibonacci d'un nombre n ne contient jamais deux 1 consécutifs.

Demonstration:

En effet, si le codage de n contient deux 1 consécutifs aux positions i et $i+1$, c'est que l'on a dans la décomposition les nombres $Fib(i)$ et $Fib(i+1)$. Or, d'après la définition de la suite de Fibonacci, $Fib(i+2) = Fib(i+1) + Fib(i)$. De plus, le codage préconise d'utiliser toujours les nombres les plus grands. Donc, lors du codage du nombre, l'agorithme aurait dû utiliser $Fib(i+2)$ au lieu de $Fib(i+1) + Fib(i)$. □

De ce principe élémentaire, on peut en déduire une manière naturelle de compter dans ce codage. Il suffit en effet de compter "normalement" en binaire en sautant toutes les combinaisons qui amèneraient à juxtaposer deux 1.

n	codage	n	codage	n	codage	n	codage
1	1	6	1001	11	10100	16	100100
2	10	7	1010	12	10101	17	100101
3	100	8	10000	13	100000	18	101000
4	101	9	10001	14	100001	19	101001
5	1000	10	10010	15	100010	20	101010

Figure 3.1 – Exemple de codage

3.1.4 Longueur du code

Une propriété établie au 19^{eme} siècle permet de connaître la longueur de la représentation d'un nombre de Fibonacci : soit n un nombre quelconque. Alors $\#Rep(n) = O(log_{\varphi}(n))$ où φ est le nombre d'or, à savoir $\frac{1+\sqrt{5}}{2}$

3.1.5 Propriété du code

Propriété:

Soit n un nombre de Fibonacci, alors $n = Fib(\#Rep(n))$.

Demonstration:

Soit n un nombre de Fibonacci. Alors il existe un entier i tel que $n = Fib(i)$. Et ainsi, la représentation de Fibonacci de n sera un 1 suivi de $i-1$ zéros, le 1 étant en position i. Donc $\#Rep(n) = i$, et $n = Fib(\#Rep(n))$. □

3.2 Arbre de Fibonacci

3.2.1 Définition

Pour coder la structure de données qui va supporter le graphe de voisinage dans la pentagrille, nous allons utiliser un arbre dit de Fibonacci. Détaillons la structure de cet arbre. Chaque Nœud peut être un 2-Nœud ou un 3-Nœud, c'est-à-dire posséder 2 ou 3 fils. Un 2-Nœud aura obligatoirement comme fils un 3-Nœud et un 2-Nœud. Un 3-Nœud aura obligatoirement comme fils deux 3-Nœuds et un 2-Nœud. Pour construire notre arbre, la racine est un 3-Nœud, qui possède comme fils, de gauche à droite, deux 3-Nœuds et un 2-Nœud. Ensuite, chaque 3-Nœud de l'arbre aura comme fils, de gauche à droite, un 3-Nœud, un 2-Nœud et un 3-Nœud, et chaque 2-Nœud aura comme fils, de gauche à droite, un 3-Nœud et un 2-Nœud.

La figure 3.2 présente un exemple de représentation de cet arbre, muni d'une numérotation de haut en bas et de gauche à droite, les points blancs (∘) étant des 3-Nœuds et les points noirs (•) étant des 2-Nœuds.

3.2.2 Propriétés sur les dimensions

Le nœud racine possède 3 fils. Ne nous occupons pas pour l'instant de leur ordre. Au niveau 1, nous avons donc 1 seul 3-Nœud, et au niveau 2, nous avons un 2-Nœud et deux 3-Nœuds. Chaque 3-Nœud engendrera deux 3-Nœud et un 2-Nœud à son tour, et chaque 2-Nœud engendrera un 3-Nœud et un 2-Nœud. Donc, au niveau 3, nous aurons cinq 3-Nœuds, (avec $5 = 2 \times 2 + 1 \times 1$) et trois 2-Nœuds (avec $3 = 1 \times 2 + 1 \times 1$), soit un total de huit nœuds sur le niveau 3.

Par intuition, on voit que l'arbre épouse la suite de Fibonacci. Voyons cela en détail.

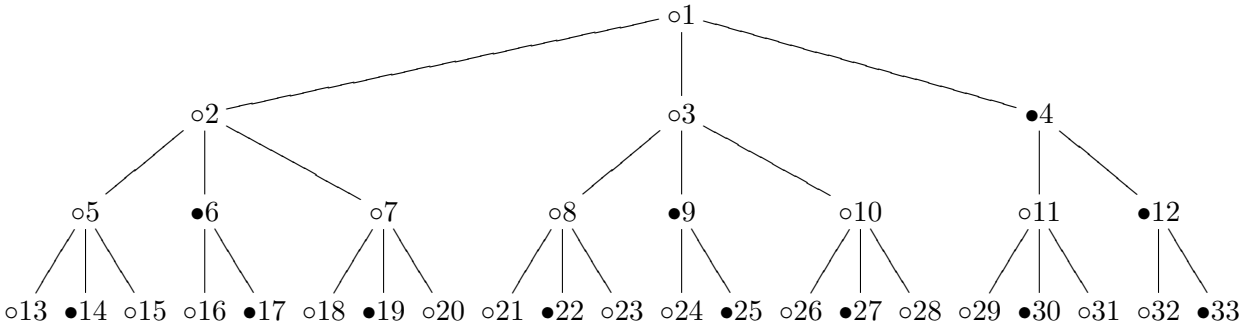

Figure 3.2 – Arbre de Fibonacci

Propriété:

Si u_n est le nombre de 2-Nœuds au niveau n, et v_n le nombre de 3-Nœuds au niveau n, alors

$$\begin{cases} u_n &= Fib(2n-3) \\ v_n &= Fib(2n-2) \end{cases}$$

Demonstration:

Soit u_n le nombre de 2-Nœuds au niveau n, et v_n le nombre de 3-Nœuds au niveau n. Nous savons déjà que $u_1 = 0$ et que $v_1 = 1$. Maintenant, connaissant les règles qui régissent l'arbre, nous avons : $u_{n+1} = u_n + v_n$ et $v_{n+1} = u_n + 2v_n$. Nous allons tenter de prouver par récurrence la propriété $P(n)$ suivante :

$$\begin{cases} u_n &= Fib(2n-3) \\ v_n &= Fib(2n-2) \end{cases}$$

Il faut que nous définissions $Fib(-1) = 0$, ce qui vérifie $Fib(1) = Fib(0) + Fib(-1)$. Vérifions l'hypothèse de base $P(1)$:

$$\begin{cases} u_1 &= Fib(-1) &= 0 \\ v_1 &= Fib(0) &= 1 \end{cases}$$

Supposons que $P(i)$ vrai $\forall i \leq n$, et montrons que $P(n+1)$ vrai. Calculons u_{n+1} et v_{n+1} :

$$\begin{array}{rcl} u_{n+1} &=& u_n + v_n \\ &=& Fib(2n-3) + Fib(2n-2) \\ &=& Fib(2n-1) \\ &=& Fib(2(n+1)-3) \end{array}$$

$$\begin{array}{rcl} v_{n+1} &=& u_n + 2v_n \\ &=& Fib(2n-3) + 2Fib(2n-2) \\ &=& (Fib(2n-3) + Fib(2n-2)) + Fib(2n+2) \\ &=& Fib(2n-1) + Fib(2n-2) \\ &=& Fib(2n) \\ &=& Fib(2(n+1)-2) \end{array}$$

Donc $P(n)$ vrai $\forall n \in \mathbb{N}^*$.

Propriété:

Soit $c_n = u_n + v_n$ le nombre de nœuds présents sur le niveau n de l'arbre, alors nous avons

$$c_n = Fib(2n-1)$$

Demonstration:

Par définition

$$\begin{cases} u_n &= Fib(2n-3) \\ v_n &= Fib(2n-2) \end{cases}$$

Donc, par définition de la suite de Fibonacci, nous avons $c_n = Fib(2n-3) + Fib(2n-2) = Fib(2n-1)$. □

Nous allons maintenant calculer le nombre de nœuds d'un arbre de Fibonacci complet possédant n niveaux.

Propriété:

Soit $S(n)$ le nombre de nœuds de l'arbre de Fibonacci complet. Alors $S(n) = Fib(2n) - 1$.

Demonstration:

Prouvons déjà le lemme $P(n)$ suivant par récurrence :

$$\sum_{i=0}^{n} Fib(i) = Fib(n+2) - 1$$

Vérifions $P(0)$:

$$\sum_{i=0}^{0} Fib(i) = Fib(0) = Fib(2) - 1$$

Supposons ensuite $P(i)$ vrai $\forall i \leq n$, et montrons que $P(n+1)$ vrai.

$$\begin{aligned}\sum_{i=0}^{n+1} Fib(i) &= Fib(n+1) + \sum_{i=0}^{n} Fib(i) \\ &= Fib(n+1) + Fib(n+2) - 1 \\ &= Fib(n+3) - 1\end{aligned}$$

Donc $P(n+1)$ est vrai et nous avons $P(n)$ vrai $\forall n \in \mathbb{N}$.

Ainsi, pour terminer, soit le nombre de nœuds de l'arbre jusqu'au niveau n

$$S(n) = \sum_{i=1}^{n} c(n)$$

Alors on a :

$$\begin{aligned}S(n) = \sum_{i=1}^{n} c(i) &= \sum_{i=1}^{n} Fib(2i-1) \\ &= \sum_{i=1}^{n} Fib(2i-2) + Fib(2i-3) \\ &= \sum_{i=0}^{n-1} Fib(2i) + Fib(2i-1) \\ &= \sum_{i=-1}^{2n-2} Fib(i) \\ &= Fib(-1) + \sum_{i=0}^{2n-2} Fib(i) \\ &= Fib(2n) - 1\end{aligned}$$

□

De la sorte, on sait qu'au niveau n, le premier et le dernier élément possèdent respectivement les numéros $Fib(2n-2)$ et $Fib(2n) - 1$.

Propriété:

Si un nœud n se situe sur le bord gauche de l'arbre de Fibonacci, alors $n = Fib(\#Rep(n))$.

Demonstration:

Soit un nœud n de l'arbre de Fibonacci. Alors il se situe sur le niveau i de l'arbre. Si en plus, il est situé sur le bord gauche, il est le premier élément de ce niveau i, et donc, $n = Fib(2i-2)$. Or, on a vu précédemment que si n était un nombre de Fibonacci, alors $n = Fib(\#Rep(n))$.

3.3 Graphe de Fibonacci

Nous allons maintenant étendre l'arbre de Fibonacci en un graphe afin de posséder une structure de données se calquant à la pentagrille. C'est-à-dire que chaque nœud aura 5 voisins exactement. Chaque nœud du graphe de Fibonacci correspond à un pentagone de la pentagrille présentée précédemment, et les cinq nœuds voisins dans le graphe ont les cinq mêmes numéros que les cinq voisins dans la pentagrille. Voici en figure 3.3 une représentation de ce graphe. Les lignes pointillées sont les liens rajoutés à l'arbre de Fibonacci classique. Notez que la racine est reliée à deux graphes de Fibonacci voisins, et que chaque nœud du bord est relié à un nœud du bord du graphe voisin.

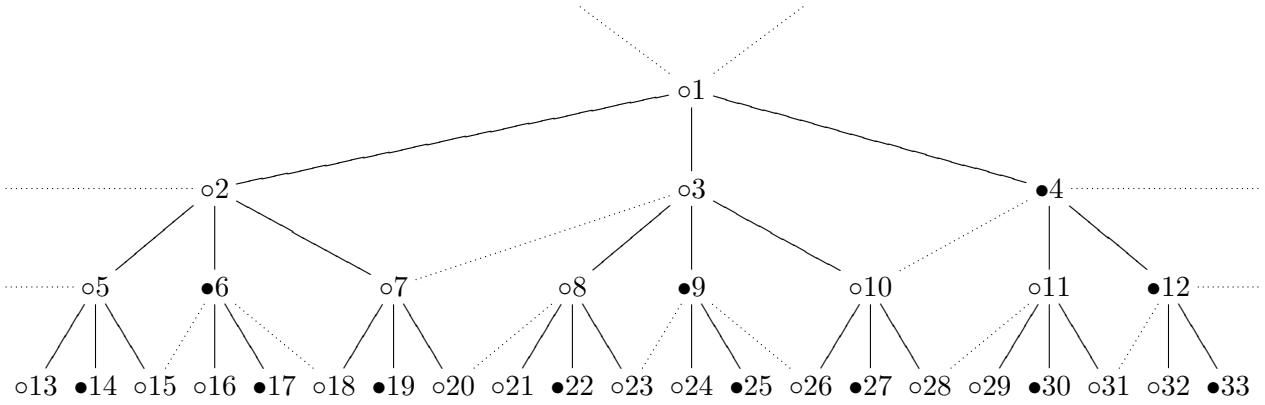

Figure 3.3 – Graphe de Fibonacci

En fait, grâce au codage de Fibonacci, on peut déterminer directement la nature d'un noeud et l'ensemble de ses voisins. Déjà, on peut voir que, pour prendre le père, il suffit de retirer les deux derniers chiffres binaires, chaque fils d'un nœud donné se voit numéroté avec son père en préfixe puis de deux chiffres binaires, dans l'ordre 00, 01 et 10. Ensuite, il suffit de regarder les deux derniers chiffres binaires. En effet, si la représentation binaire se termine par 01, il s'agit d'un 2-Nœud (puisqu'il est le deuxième fils), et automatiquement, on peut déterminer ses deux voisins supplémentaires. Tout cet algorithme décrivant le calcul du père, des fils et des différents voisins (ainsi que la majorité des démonstrations de ce chapitre)

en fonction de la représentation de Fibonacci a été décrit dans [9] On peut aussi déterminer si un nœud se situe sur la bordure gauche ou la bordure droite de manière directe avec les propriétés que l'on a déterminés précédemment : pour un nœud n situé sur la bordure gauche, alors $Fib(\#Rep(n)) = n$ et pour un nœud situé sur la bordure droite, $Fib(\#Rep(n+1)) = n+1$ (c'est-à-dire qu'il est le prédécesseur d'un nœud situé sur la bordure gauche). Enfin, il y a deux types de 3-Nœuds : ceux qui se terminent en 00 et ceux qui se terminent en 10 ; ce qui change aussi le voisin supplémentaire. De ce fait, toutes les données dont nous avons besoin pour déterminer le voisinage d'un nœud se calculent directement, car elles découlent de la représentation de Fibonacci du numéro du nœud, qui se calcule en $O(log_{\varphi}(n))$ (on considère la suite de Fibonacci précalculée, ou au moins en cache, comme nous le verrons dans l'implémentation plus loin).

Chapitre 4

Automates Cellulaires

4.1 Automates Cellulaires classiques

4.1.1 Définition d'un Automate Cellulaire

Un automate cellulaire se définit par un quadruplet (E, H, S, f), avec E, son espace de travail (ou pavage) ; H un voisinage, c'est-à-dire une liste de déplacements relatifs dans l'espace E ; la taille de H $|H|$ est le nombre de voisins de chaque cellule ; S est l'ensemble des états qu'une cellule peut prendre ; et f est une fonction de transition définie de $S^{|H|}$ vers S qui permet de calculer le nouvel état de chaque cellule à partir des états des celulles du voisinage H.

On notera que dans la liste des voisins peut se situer la cellule "0", c'est-à-dire la cellule de base elle-même. On notera aussi que, si la grande majorité des automates cellulaires fonctionne avec un espace E qui est un pavage cartésien, il existe des automates cellulaires sur des pavages différents, comme par exemple à base de triangles ou d'hexagones. Et dans ces cas précis, il suffit de définir des axes afin de permettre des déplacements relatifs, et l'on peut écrire un voisinage correct.

Les particularités d'un automate cellulaire sont les suivantes : l'état de chaque cellule ne va dépendre que de l'état de ses voisines. On calcule le nouvel état de l'automate en parallèle, c'est-à-dire que l'on applique la fonction de transition f à chaque temps t de l'évolution de l'automate sur toutes les cellules en même temps, et en parallèle. C'est pour cela que l'on qualifie les automates cellulaires de "synchrones".

4.1.2 Un exemple dans le plan euclidien : le jeu de la vie

Un exemple courant d'automate cellulaire est le jeu de la vie. Cet automate se définit avec $d = 2$, $H = \{(-1,-1),(-1,0),(-1,1),(0,1),(1,1),(1,0),(1,-1),(0,-1),(0,0)\}$, $S = \{0,1\}$, et

$$f(v1, v2, v3, v4, v5, v6, v7, v8, c) = \begin{cases} 1 \text{ si } c = 1 \text{ et } 2 \leq \sum_{i=1}^{8} v_i \leq 3 \text{ ou } c = 0 \text{ et } \sum_{i=1}^{8} v_i = 3 \\ 0 \text{ sinon} \end{cases}$$

ou, dit dans un langage courant, une cellule vaut 0 si elle est morte, 1 si elle est en vie, et elle naît si elle a exactement 3 voisins, reste en vie si elle a 2 ou 3 voisins, et meurt ou reste morte dans tous les autres cas.

Le jeu de la vie a au départ été défini par Conway en 1970 [11] dans l'idée de faire un jeu. Il s'est finalement révélé être un outil théorique intéressant. En particulier, Conway a prouvé l'universalité de ce jeu, c'est-à-dire que l'on peut construire une machine de Turing universelle à l'aide de ce jeu.

4.1.3 Etat de l'art des simulateurs

Il existe de nombreux simulateurs d'automates cellulaires disponibles, principalement développés pour des simulations physiques, ou pour de l'informatique appliquée, mais aussi certains à buts ludiques ou pédagogiques. En particulier, on peut citer le simulateur de Dana Eckart [1], qui fonctionne dans des espaces euclidiens, et qui utilise un langage appelé Cellang de modélisation des fonctions de transitions. Elle a développé notamment un outil parallèle de visualisation et de simulation d'automates parallèles. On peut citer aussi les travaux de Freiwald U. et Weimar J.R., qui présentent JCAsim [12], un outil de visualisation développé en java, et qui essaie d'introduire une notion de programmation objet dans la définition des fonctions de transition. JCASim permet également de simuler tout automate cellulaire de l'espace euclidien de dimension quelconque. Mais bien sûr, ces outils ne permettent pas de travailler dans le plan hyperbolique.

4.2 Automates Cellulaires dans le plan hyperbolique

Les premiers papiers sur les automates cellulaires dans le plan hyperbolique datent de 2001, où Margenstern a défini dans [10], [9], des automates cellulaires dans le plan hyperbolique et a obtenu un certain nombre de résultats théoriques intéressants dans cet espace, en particulier il a été établi une méthode pour trouver une solution polynomiale dans le temps et dans l'espace au problème 3-SAT.

En 2003, la construction d'un automate universel dans le plan hyperbolique a été donnée par Herrmann et Margenstern dans [3]

En théorie, dans le plan hyperbolique, $P = NP$. Tout du moins, l'implémentation complète de l'automate décrit par Morita et Margenstern en [10] afin de résoudre un problème 3-SAT pourrait être faite sur l'outil développé, et vérifier ainsi les théories mises en place.

4.2.1 Définition

On ne peut plus suivre totalement la définition classique d'un automate cellulaire ici. En effet, E est bien la pentagrille telle que nous l'avons définie précédemment, S et f restent respectivement les états et la fonction de transition, mais le voisinage H n'est pas définissable complètement de manière mathématique ici : il nous faut rajouter une notion provenant du graphe de Fibonacci. En effet, on a vu que le voisinage est une liste de coordonnées relatives à une cellule quelconque de l'automate. Or, il n'y a pas de déplacements relatifs définis dans la pentagrille. Nous n'avons que les coordonnées absolues à notre disposition. En revanche, on sait définir le voisinage en absolu d'une cellule, à partir du graphe de Fibonacci, avec des notions comme Père, Fils, Cousin, etc... Donc, on définira H uniquement sous la forme $H = \{i | 1 \leq i \leq 5\}$, et l'on saura, lorsque l'on y fera référence, qu'il s'agit du voisinage défini par le graphe de Fibonacci, avec éventuellement l'usage des primitives telles que $Pere(v)$ ou $Fils1(v)$.

4.2.2 Etat de l'art des simulateurs

Contrairement aux automates cellulaires classiques, il n'y a que très peu de travaux établis sur ce sujet, et encore moins d'applications concrètes permettant de simuler un automate cellulaire dans le plan hyperbolique. Imai et Ogawa [2] se sont basés sur un pavage de quadrangles et ont effectués quelques recherches sur le jeu de la vie, mais n'ont pas utilisé la pentagrille, et ont utilisé un jeu de la vie à 5 états. Par ailleurs, Herrmann et Margenstern [4] ont développé un simulateur d'automate cellulaire sur la pentagrille, mais sans outil de visualisation graphique.

4.2.3 Le jeu de la vie

Dans le cas euclidien, le jeu de la vie est défini sur un voisinage de Moore composé de 8 cellules, la naissance lors de la présence de 3 voisins, la survie entre 2 et 3 voisins, et la mort dans tous les autres cas. Dans le plan hyperbolique, il nous faut réinventer le jeu de la vie.

En ce qui nous concerne, nous allons définir notre jeu de la vie sur la pentagrille avec deux états, la naissance lors de la présence de 2 voisins, et la survie entre 2 et 4 voisins. On se rendra compte rapidement qu'il existe plusieurs possibilités d'établir des jeux de la vie sur la pentagrille. Nous pouvons déjà établir une notation afin de classifier un tout petit peu ceux-ci : Notons un jeu de la vie sur la pentagrille $\{p, q\}$, p étant le nombre maximum de voisins pour la survie, et q la quantité minimum de voisins nécessaires pour la survie, ainsi que la quantité nécessaire pour la naissance. Le jeu de la vie que nous avons décrit au début de ce paragraphe est donc un $\{4, 2\}$. Vous trouverez quelques images de ce jeu de la vie sur le chapitre décrivant l'implémentation, en figure 5.6, page 31

Parmi les quelques configurations testées, ce jeu de la vie était le seul à rester à peu près stable quant au nombre de cellules occupées. D'autres configurations ont été testées. La configuration proche de celle du jeu de la vie dans le plan euclidien, $\{3, 2\}$ avait tendance à mourir relativement vite, et à ne laisser que quelques survivants. Une troisième configuration, $\{4, 1\}$, avait tendance à envahir l'espace.

Le problème avec la pentagrille, c'est que visuellement, elle n'est pas régulière : plus les cellules s'approchent de l'horizon, plus leur taille se réduit. Ainsi, autant pour le jeu de la vie dans le plan euclidien il fut possible de visualiser par hasard les configurations qui se répètent et se translatent, autant en ayant la pentagrille comme support, il est difficile de visualiser à l'œil nu la progression de notre environnement. Ainsi, nous pouvons dors et déjà définir plusieurs extensions possibles de ce projet : une fonction de déplacement, permettant de changer la numérotation des cellules à volonté, et ainsi tenter de suivre une configuration en la centrant toujours dans la partie de la pentagrille qui reste correctement visualisable à l'œil nu ; et une fonction de recherche de configurations identiques, qui tenterait de retrouver des similitudes dans l'automate cellulaire par translation et rotation à partir d'une de ses configurations précédentes.

Chapitre 5

Implémentation

5.1 Objectifs

Il était demandé de réaliser un outil de simulation d'un automate cellulaire dans le plan hyperbolique, ainsi que son outil de visualisation associé, le tout fonctionnant en mode client-serveur et écrit en C++. En outre, l'idée était de se libérer des contraintes imposées par les grands nombres générés par la suite de Fibonacci. En effet, on atteint rapidement des grandeurs astronomiques qui dépassent la capacité habituelle des entiers classiques de la machine sur 32 bits.

L'architecture client-serveur permet essentiellement d'avoir l'algorithme de l'automate cellulaire qui tourne sur une machine puissante, pas forcément dotée d'écran, et d'avoir une ou plusieurs sorties écran sur des machines client qui viennent se connecter en TCP/IP sur le serveur afin d'y récupérer les données concernant l'état en cours de l'automate. Le client peut ensuite afficher l'automate suivant diverses méthodes.

5.2 Analyse

5.2.1 Vue d'ensemble

Ce projet nécessite une programmation relativement étalée sur plusieurs sujets ; il a donc été découpé en unités indépendantes. Voici en figure 5.1 une vue d'ensemble du modèle client-serveur.

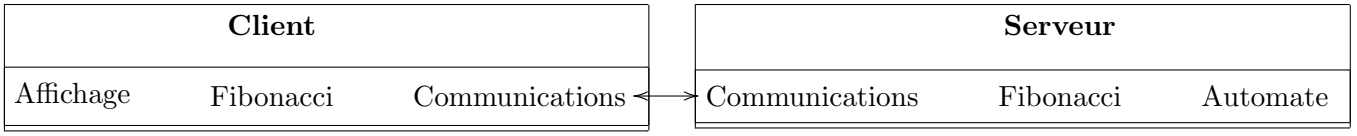

Figure 5.1 – Modèle global

La partie communications du serveur est beaucoup importante que celle du client. En effet, elle va gérer plus d'un client à la fois. Sa structure est explicitée en figure 5.2.

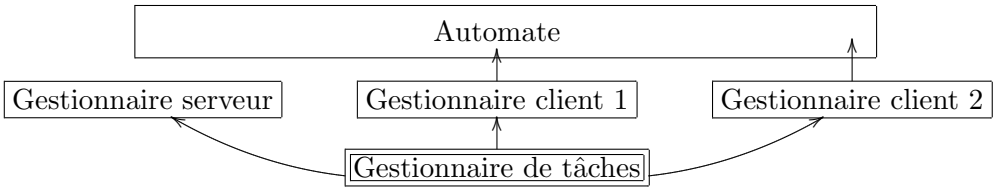

Figure 5.2 – Serveur

Nous pouvons voir que l'automate fonctionne de manière autonome ; chaque gestionnaire de client va aller l'interroger pour obtenir l'état en cours de la simulation. Il s'agit donc là que d'un accès en lecture seule. Le rôle du gestionnaire serveur est d'attendre de nouveaux clients. Ensuite, un gestionnaire de tâches va coordonner les entrées/sorties réseau.

L'automate en lui-même possède une structure relativement simple. Il se base sur un graphe de Fibonacci dont les cellules sont stockées dans un arbre rouge et noir. La figure 5.3 tente de décrire la hiérarchie des classes C++ qui interviennent.

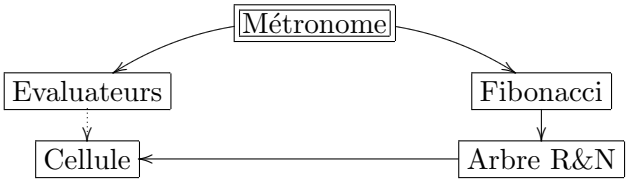

Figure 5.3 – Automate

Enfin, la figure 5.4 présente une vue de l'architecture de tout le logiciel.

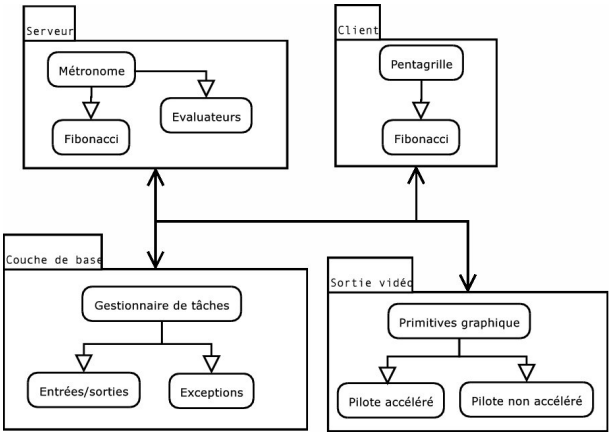

Figure 5.4 – Architecture

Nous allons maintenant analyser en détail les points essentiels des différentes parties présentées.

5.2.2 Les états de l'automate

Pour éviter de se limiter quant aux nombres d'états, ainsi qu'à leurs types, il nous faut prévoir un nombre intéressant d'états différents, sans pour autant les fixer. De plus, à terme, on voudrait éventuellement garder un certain historique des états, afin de pouvoir "revenir en arrière" lors de la simulation.

De ce fait, nous allons utiliser de simples entiers sur 32 bits, ce qui permet d'obtenir déjà plus de 4 milliards d'états différents. Pour l'historique, il suffit que chaque nœud possède un tableau circulaire d'états. L'indice de l'état en cours est obtenu en effectuant un modulo de la taille du tableau sur le temps.

Les évaluateurs présentés figure 5.3 jouent le rôle de la fonction de transition. Ils permettent donc de calculer le nouvel état d'une cellule en fonction des états des cellules voisines au temps précédent. La particularité ici est qu'il peut y avoir plusieurs évaluateurs, organisés en pile. L'idée est que pour un automate très complexe, comme le modèle 3-SAT, il peut être plus facile d'instaurer des règles de transition par défaut, qu'on laissera en bas de la pile, et rajouter par dessus des règles plus spécifiques. Un évaluateur devra alors renvoyer *vrai* ou *faux* s'il a pris en charge la cellule qu'il lui a été demandé d'évaluer. On descend ainsi la pile des évaluateurs jusqu'à ce qu'il y en ait un qui prenne en charge la cellule. La figure 5.5 montre un exemple simple d'une pile d'évaluateurs.

```
Si (Cellule = 5) alors
    Cellule := 7 ;
    renvoyer vrai ;
Sinon renvoyer faux ;
```

```
Si (Somme des voisins ≥ 10) alors
    Cellule := Somme des voisins / 2 ;
    renvoyer vrai ;
Sinon renvoyer faux ;
```

```
Cellule := 0 ;
renvoyer vrai ;
```

Figure 5.5 – Pile d'évaluateurs

5.2.3 La structure de données

La structure supportant la Pentagrille est elle-même relativement complexe. Une simple lecture de la définition nous indique clairement que les coordonnées des voisins sont calculées de manière absolue, et nous donnent des nombres, et éventuellement des indications comme "bord gauche" ou "bord droit". Il n'y a rien permettant l'accès direct à un nœud. De plus, on tentera d'élaguer régulièrement les nœuds qui sont devenus totalement inactifs, c'est-à-dire, dont les voisins sont tous à 0, et que l'historique du nœud est passé entièrement à 0, ce qui fait que la structure sera creuse, et que tous les nœuds vivants ne sont pas forcément atteignables à un instant donné. Ainsi, il nous faut une structure de données qui nous permette de stocker des nombres, et de les retrouver facilement et rapidement.

Donc, la solution adoptée sera de stocker les nœuds de la Pentagrille dans un arbre rouge et noir, et l'on considérera qu'un nœud non présent aura l'état arbitraire 0. L'utilisation de l'arbre rouge et noir se fait de manière transparante. Le programmeur qui s'adresse à la structure du graphe de Fibonacci ne sait pas quel est le support qui est derrière. Ainsi, il sera possible éventuellement de changer cette structure pour une autre, s'il advient qu'elle est plus efficace.

Les arbres rouges et noirs ont été décrits par Cormen dans [7]. Il s'agit d'une structure en arbre binaire de recherche dont chaque nœud contient une information supplémentaire, sa couleur, qui peut valoir soit rouge soit noir. En contrôlant la manière dont les nœuds sont colorés sur n'importe quel chemin allant de la racine aux feuilles, les arbres rouges et noirs garantissent qu'aucun des chemins n'est deux fois plus long que n'importe quel autre, ce qui rend l'arbre approximativement équilibré. Le temps d'accès d'un arbre rouge et noir équilibré est en $O(log(n))$.

5.2.4 La question de la précision

La grandeur des nombres de Fibonacci, ainsi que la précision nécessaire pour les calculs dans le plan hyperbolique rendent totalement inutiles les conteneurs classiques du C++.

Il fut donc choisi d'utiliser des librairies de calcul en précision arbitraire, telle que la librairie Gnu MP, permettant de stocker de grands entiers, de faire des calculs sur des flottants avec une précision arbitraire, possédant une interface C++, et surtout bénéficiant d'une notoriété incontestée en la matière.

5.2.5 Le système client/serveur

C'est un point relativement important dans cette application. Il faut pouvoir gérer un nombre arbitraire de clients, sans pour autant occuper trop de ressources système.

De plus, lorsque l'on analyse le modèle du serveur, il est divisé en deux modules : l'aspect calculatoire qui va occuper un processus à part, et l'aspect communication réseau. En temps normal, dans un schéma de programmation d'un serveur classique, on va utiliser un processus distinct par connexion entrante. Mais dans ce cas, la partie communication risque d'occuper en permanence quatre ou cinq processus, forçant ainsi le processeur à délaisser la partie calculatoire.

Ainsi, on va construire un modèle de serveur un peu différent de la normale, en limitant l'ensemble des communications à un seul processus. Pour ce faire, on construit un mini gestionnaire de tâches, avec comme modèle de tâche par exemple, une communication vers l'extérieur. La commutation des tâches se fera non pas de manière arbitraire, mais sur des points très précis de la programmation.

5.2.6 Affichage

L'affichage se doit d'être rapide, même si les calculs dans le plan hyperbolique sont coûteux en temps machine. De plus, il faut que le code soit portable, et si possible, ait le même aspect, quelque soit la plateforme qui lance l'application.

C'est pourquoi j'ai décidé de me reposer sur deux librairies reconnues robustes et rapides pour l'affichage : SDL (Simple DirectMedia Layer, *http ://www.libsdl.org*, librairie OpenSource initialement dévelopée par Loki) et OpenGL (librairie graphique 2D/3D développée par SGI), et d'utiliser les cartes graphiques accélératrices qui peuvent nous aider à afficher rapidement notre Pentagrille.

En effet, ce nouveau type de matériel permet l'affichage de textures, en utilisant non pas le processeur central, mais le processeur de la carte graphique. De ce fait, en précalculant les différents pentagones à afficher sous la forme de textures, on pourra afficher la pentagrille en entier, et en assignant à chaque pentagone une couleur individuelle.

5.3 Implémentation et algorithmes

La totalité de ce projet contient plus de 11000 lignes de code, pour un total d'environ 260Ko répartis dans près de 100 fichiers. Ce projet est divisé en trois parties. Examinons-les en détail.

5.3.1 La couche de base : le gestionnaire des tâches

L'implémentation de cette couche représente à elle seule environ 60 fichiers différents, pour 6000 lignes de code et 140Ko. Elle contient non seulement le gestionnaire des tâches, mais aussi toutes les fonctions vitales au système, comme des primitives permettant l'affichage et la conversion aisée des grands nombres de la librairie Gnu MP, ou encore divers conteneurs sur les entrées/sorties, qui facilitent la tâche de programmation. Cette couche est une véritable librairie indépendante, qui pourra être réutilisable dans une autre application.

Elle est elle-même divisée en trois couches distinctes. La couche 1 contient les classes d'Exceptions de base, permettant aux tâches d'envoyer des messages au gestionnaire de tâches. La couche 2 contient une encapsulation d'entrées/sorties diverses, afin de fournir un mécanisme non bloquant transparent. Ces classes vont envoyer de manière transparente des messages au gestionnaire de tâches afin éventuellement d'interrompre une tâche qui tenterait d'effectuer un appel bloquant. Et enfin la troisième couche contient le gestionnaire de tâche en lui-même, le squelette d'une classe virtuelle Task, et quelques tâches de base pouvant servir à tout moment, comme une tâche de recopie d'une entrée vers une sortie.

Il est à noter que le système d'entrée/sortie de la couche 2 mélange aussi bien les fichiers physiques présents sur le disque dur que des sockets TCP/IP vers l'extérieur, ou même des entrées/sorties virtuelles, telles que des tampons internes. Ainsi, la tâche de recopie va-t-elle pouvoir par exemple recopier indifféremment un fichier présent sur le disque vers un socket, ou un tampon contenant le résultat d'une requête précédemment remplie et calculée par une autre tâche. La programmation en classes virtuelles C++ permet une programmation générique.

Le gestionnaire des tâches intégré est le résutat d'un dialogue avec Alexander V. Lukyanov. Auteur du client ftp *lftp*, il a intégré dans son logiciel un gestionnaire de tâche indépendant. La discussion a été très instructive, et m'a permis de concrétiser certains mécanismes qui me paraissaient encore obscurs dans l'intégration d'un tel système. L'idée de base est la suivante : une librairie comme la libpthreads ne connaît pas le logiciel sur lequel elle va devoir opérer. La commutation de tâche est faite en interne par le noyau

du système d'exploitation, et de plus, il est courant que les machines avec lesquelles nous travaillons sont monoprocesseur. Si l'on considère en plus que la partie TCP/IP doit rester insignifiante vis-à-vis des calculs de l'automate, si l'on doit lancer l'automate sur une grosse machine parallèle, il vaut mieux que la partie communication n'occupe que très peu de ressources du système, voire, qu'un seul processeur[1]

Ainsi, si l'on considère que le temps de calcul entre deux transferts TCP/IP est insignifiant, les points critiques se situent au niveau de ces fameux transferts. Il suffit donc d'inverser le schéma classique de vision d'un système multitâche : on ne définit plus des zones "à risques" ou "sections critiques", mais des points de commutations, situés soit implicitement, par l'usage d'entrées/sorties bloquantes, soit explicitement, par l'usage par le programmeur d'une exception signifiant une mise en attente d'un évènement. De ce fait, le gestionnaire des tâches va simplement attendre qu'une tâche s'arrête naturellement, ou s'interrompe par une exception. Si c'est par une exception, le gestionnaire va l'analyser, et déterminer s'il s'agit d'une exception de demande de "mise en attente", ou s'il s'agit d'une erreur système. S'il s'agit d'une erreur, il va tuer la tâche, en détruisant simplement l'objet qui la contient. Sinon, il met la tâche dans sa liste en attente, liste organisée suivant l'algorithme Round Robin, et va organiser une attente sur les différents évènements qui sont demandés par toutes les tâches. Dès qu'un ou plusieurs évènement surviennent, il relancera la tâche correspondante, en respectant la priorité Round Robin si plusieurs tâches ont l'opportunité de se réveiller en même temps.

En clair, une tâche va être segmentée naturellement par le programmeur en blocs fonctionnels, les délimitations de ces blocs étant des entrées/sorties sur le réseau. De ce fait, on sait que chacun de ces blocs s'exécutera d'une seule traite, ce qui rend la gestion des interblocages implicites : un bloc ne pourra jamais se faire interrompre en plein milieu de son exécution, et rend ainsi inutile l'utilisation d'un système d'exclusion mutuelle.

Et donc, l'implémentation obtenue marche bien, pour un système de tâches légers, n'occupant pas plus de quelques méga-octets en mémoire, ne dupliquant pas arbitrairement les données des tâches, et n'obligeant pas le programmeur à gérer des interblocages éventuels que l'utilisation d'une librairie plus lourde aurait forcément entraîné.

5.3.2 La couche intermédiaire : la sortie vidéo

Cette couche est programmée en 25 fichiers, sur 3000 lignes de code, pour 75Ko. Elle contient un ensemble de primitives permettant la sortie vidéo en OpenGL et SDL. En effet, dans un souci de limitation des dépendances sur trop de librairies externes, j'ai programmé des routines d'affichage de texte, de chargement et de sauvegarde d'images sur disque, et de création et d'affichage de formes géométriques complexes. En effet, vu qu'aucune librairie existante ne faisait exactement ce dont j'avais besoin, j'ai commencé à programmer les fonctions de bases pour afficher des dessins arbitraires sur l'écran. Puis, plutôt que de mixer ce code avec une librairie de texte différente, j'ai utilisé ce que j'avais déjà programmé pour créer l'affichage du texte.

Elle est elle-même divisée en trois couches distinctes, deux étant assimilées à des pilotes graphiques, et une au-dessus, permettant des opérations plus lourdes. Il existe en effet principalement deux méthodes

[1] Cela n'est valide que pour une machine parallèle à mémoire partagée

pour effectuer une sortie graphique en utilisant SDL et/ou OpenGL : avec ou sans accélération matérielle. Il est nécessaire d'implémenter les deux méthodes, avec l'opportunité de changer de l'une à l'autre par un paramètre utilisateur, car si l'on tente d'utiliser l'affichage accéléré sur une machine qui ne possède pas d'accélération matérielle, le résultat obtenu sera radicalement l'inverse de celui escompté. De plus, la structure virtuelle permet de créer un effet de rétroaction : il est possible de créer un cadre de dessin auxiliaire, non destiné à l'affichage, ou utilisé de manière temporaire, en se basant sur le pilote non accéléré, même si c'est le pilote accéléré qui tourne sur l'affichage principal. Ces deux pilotes ont à peu près les mêmes fonctions de base, c'est-à-dire afficher des formes géométriques, et des caractères.

La couche supérieure, elle, va utiliser les briques élémentaires fournies par les pilotes, afin d'effectuer des affichages plus complets, comme des lignes de texte, des fenêtres, des boutons et autres. Le tout fonctionne exactement de la même manière sous Windows, Linux, et Solaris, plateformes qui ont été testées.

5.3.3 La couche haute : l'automate

Il y a 15 fichiers qui composent cette dernière couche, comprenant 2000 lignes de code, pour 45Ko. Tout comme les autres couches, celle-ci a une structure interne découpée en modules fonctionnels.

Tout d'abord, les nombres des Fibonacci sont calculés dans un tableau virtuel, dont la taille augmente en fonction des besoins de logiciel. Ainsi, cette structure contient toujours l'essentiel des nombres nécessaires, et une fois calculés, fonctionne en accès direct, tout comme un tableau.

Ensuite, il a été défini une structure générique permettant de représenter un graphe de Fibonacci. Cette structure est indépendante du contenu des cellules. Elle utilise là encore des conteneurs virtuels, qui une fois dérivés, permettent au programmeur d'y stocker ce qu'il le désire. Un arbre rouge et noir sert de conteneur pour toutes ces cellules, et permet ainsi une recherche rapide et efficace. Lorsqu'une cellule est créée, elle recherche automatiquement ses voisines dans la structure, et si elle les trouve, se lie à elles, en leur signalant sa présence. Ainsi, la recherche du voisinage à travers l'arbre rouge est noir est-elle faite une seule fois à la création d'une cellule, et est constamment mise à jour de manière automatique lorsque ses voisines se créent ou se détruisent. Cette structure d'arbre est commune au code du serveur et du client. En effet, tous deux ont besoin de cette structure, comme nous allons le voir.

Côté serveur, l'automate cellulaire est défini en deux parties.
- Le métronome qui contient la pentagrille précédente, et coordine les changements d'états dans l'automate, mais aussi l'élagage des cellules mortes.
- Les cellules de l'automate, conteneurs dérivés des cellules de la pentagrille, qui contiennent leurs états. Le système de changements d'état est lui aussi une classe virtuelle. Cette classe, appelée évaluateur, contient la fonction de changement d'état, et éventuellement des données auxiliaires.

Ainsi, il est possible de programmer n'importe quel automate cellulaire, comme le jeu de la vie, ou bien le simulateur 3-SAT proposé par Margenstern. De plus, il pourrait être intéressant dans une extension future de programmer un évaluateur fonctionnant à l'aide d'un langage du type de Cellang [1], afin de s'affranchir de la recompilation du serveur à chaque tentative de programmation d'un nouvel automate.

Côté client, un usage intensif de la mémoire est effectué : en effet, je précalcule tous les pentagones jusqu'à un niveau de récursion donné, et je stocke en mémoire les formes géométriques obtenues. Ceci a l'énorme aventage de pouvoir afficher l'intégralité de la pentagrille en quelques centièmes de secondes, une fois le précalcul terminé. Par contre, l'occupation mémoire monte très vite en flèche, surtout lorsqu'on a la mauvaise idée d'essayer d'afficher la pentagrille en entier avec un niveau d'affichage élevé : environ 12 pour obtenir près de 500Mo d'occupation mémoire. Mais de toute manière, un tel niveau d'affichage pour la pentagrille en entière est totalement inutile : en effet, les pentagones devenant de plus en plus petits vers l'horizon, il est impossible de les distinguer. Par contre, en zoomant sur une partie de la pentagrille, et même à des niveaux élévés de récursion, vu que l'affichage ne comporte qu'un nombre relativement restreint de pentagones, il est possible de rester dans des limites tout à fait raisonnables d'occupation temporelle et spatiale.

De plus, la présence d'un tel recouvrement visuel à l'aide d'un pavage de formes arbitraires permet de retrouver immédiatement le pentagone situé en dessous du curseur de la souris, sans strictement aucun calcul supplémentaire. Il suffit de parcourir la liste des pentagones affichés (en laissant de côté ceux qui sont d'une taille trop petite pour être sélectionnable à la souris) et effectuer un test d'appartenance très simple.

La création du recouvrement va utiliser la structure en arbre de Fibonacci, et non pas en graphe. En ne suivant que les arêtes "Fils", nous sommes sûrs de n'obtenir qu'une et une seule fois chaque nœud. Comme la pentagrille et l'arbre de Fibonacci possèdent la même structure de voisinage, il nous suffit de créer les pentagones d'après l'ordre d'arrivée des nœud. Et à l'aide d'une numérotation adéquate, on peut arriver à faire coïncider la numérotation des arêtes d'un pentagone avec la numérotation de l'ordre des voisins du graphe. Par exemple, si un nœud n de l'arbre possède deux fils, et sont les 3^{eme} et 4^{eme} voisins, alors on sait que l'on pourra générer les deux pentagones correspondant aux deux fils par réflexion du pentagone P associé à n suivant ses arêtes numéro 3 et 4.

Enfin, pour respecter un souci du détail, les pentagones s'affichent en utilisant des calculs fait en précision arbitraire grâce à la librairie Gnu MP, et une insistance particulière a été apportée sur la gestion correcte d'algorithmes de clôture/fenêtre afin de respecter toujours l'intégrité géométrique de la pentagrille. De la sorte, nous pouvons gérer un système de zoom tout à fait efficace, car il suffit de changer les coordonnées de la clôture pour obtenir immédiatement le résultat dans la fenêtre graphique. Bien évidemment, cela nécessitera de recalculer les textures des pentagones pour cette nouvelle clôture, mais une fois cette opération effectuée, l'affichage pourra reprendre immédiatement. Pour zoomer, l'utilisateur n'a qu'à définir les coordonnées de la nouvelle clôture avec la souris, en traçant un rectangle sur la zone de son choix. On peut voir un exemple de zoom sur la figure A.4, page 36

5.3.4 Résultats pratiques

La figure 5.6 montre un petit exemple d'évolutions du jeu de la vie.

J'ai aussi rajouté en appendice quelques images supplémentaires provenant du logiciel en train de tourner. On notera qu'aller jusqu'au niveau 9 de récursion pour l'affichage revient à avoir $4Fib(17) = 16724$ pentagones. Leur calcul nécessite environ 45 secondes sur l'ordinateur sur lequel j'ai effectué mes tests, un Athlon XP 2100+. Une fois ces précalculs effectués, le logiciel est capable d'afficher l'intégralité de ces 16724

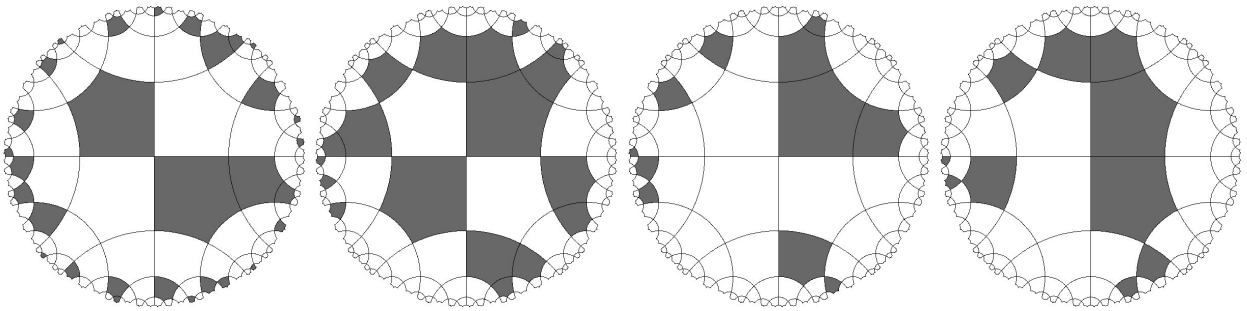

Figure 5.6 – Evolutions d'un jeu de la vie

pentagones en 7 dixièmes de seconde, soit environ 14 images par seconde. Lors d'un zoom, on améliore ces performances pour obtenir 25 images par seconde, soit 4 dixièmes de secondes par image.

A ce niveau-ci de récursion, le temps est plus consommé par le parcours de l'arbre rouge et noir que par l'affichage. En effet, le nombre de pentagones affichés en zoom est considérablement moindre par rapport au nombre de pentagones affichés sur l'écran complet : seule une trentaine subsistent lors du zoom. Il resterait donc encore quelques optimisations à faire éventuellement au niveau de l'affichage, mais les performances générales sont déjà très suffisantes.

Conclusion

Au cours de cette étude, j'ai présenté un automate cellulaire pentagonal dans le plan hyperbolique défini dans [9]. J'ai vérifié certaines propriétés de l'espace hyperbolique, de la pentagrille et de l'arbre de Fibonacci.

J'ai effectué un état de l'art des outils de simulations des automates cellulaires. Puis j'ai conçu et programmé un outil de simulation et de visualisation de l'automate cellulaire dans le plan hyperbolique. Enfin, j'ai expérimenté quelques algorithmes simples comme le jeu de la vie et de la mort qu'il a fallu redéfinir sur ce nouvel espace, et j'ai présenté mes résultats expérimentaux.

Un certain nombre de problèmes restent à résoudre, et permettent d'ouvrir notre étude sur de nouvelles perspectives. En particulier, notre simulateur doit permettre d'identifier les configurations spécifiques du jeu de la vie et de la mort (glisseurs, canons, etc...) afin de les redéfinir dans l'espace hyperbolique. Il devra également permettre d'explorer le diagramme-temps de cet automate cellulaire et de mettre en évidence l'apparition de motifs fractals.

Une autre perspective d'étude est de développer dans cet automate spécifique certains algorithmes bien connus dans les automates définis dans le l'espace euclidien.

Il pourrait aussi être intéressant de créer et d'implémenter un langage similaire à Cellang [1], afin de pouvoir programmer la fonction de transition de notre automate de manière plus intuitive qu'en C++.

Enfin, un objectif est de paralléliser le programme de simulation pour améliorer les performances de notre outil.

L'ensemble de tous ces travaux supplémentaires pourrait éventuellement faire l'objet d'une étude ultérieure, sous la forme d'une thèse par exemple.

Appendice A

Quelques photos d'écran

Voici quelques prises de vue du logiciel en lui-même.

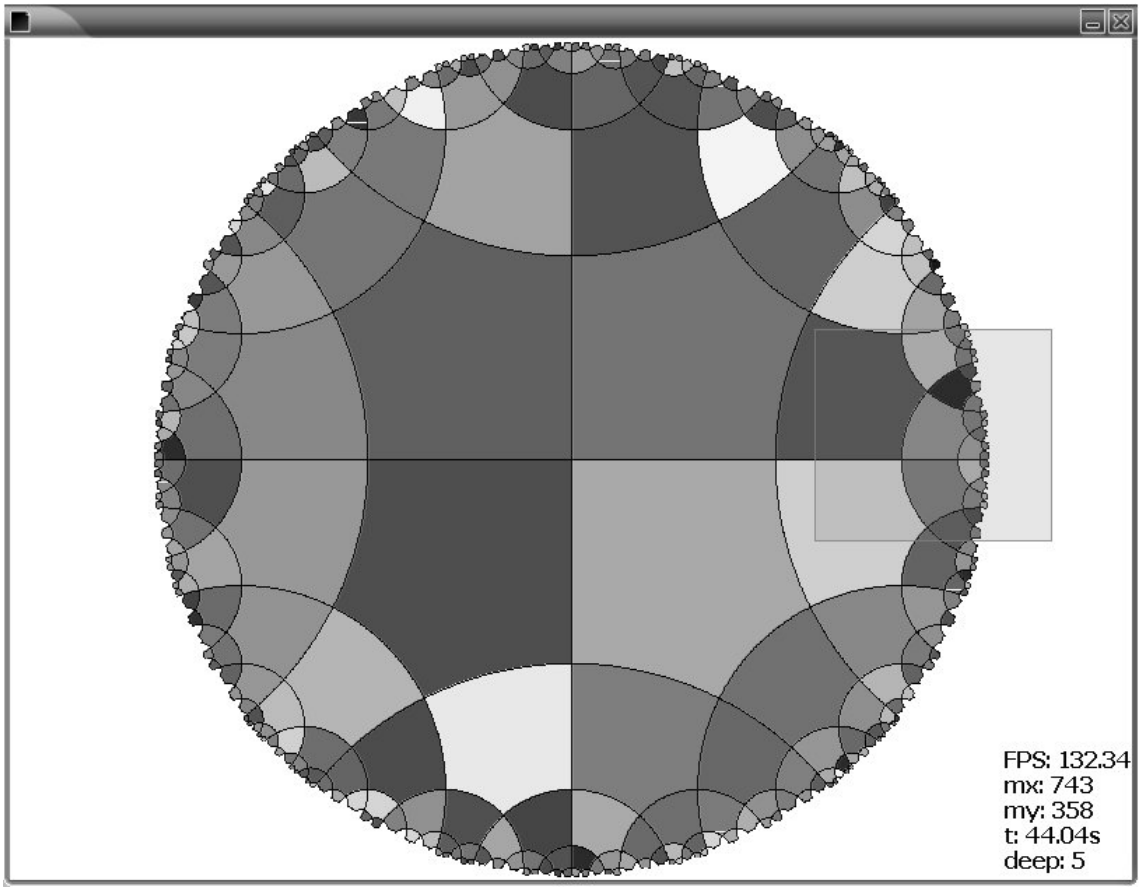

Figure A.1 – Sélection d'une zone à zoomer avec la souris

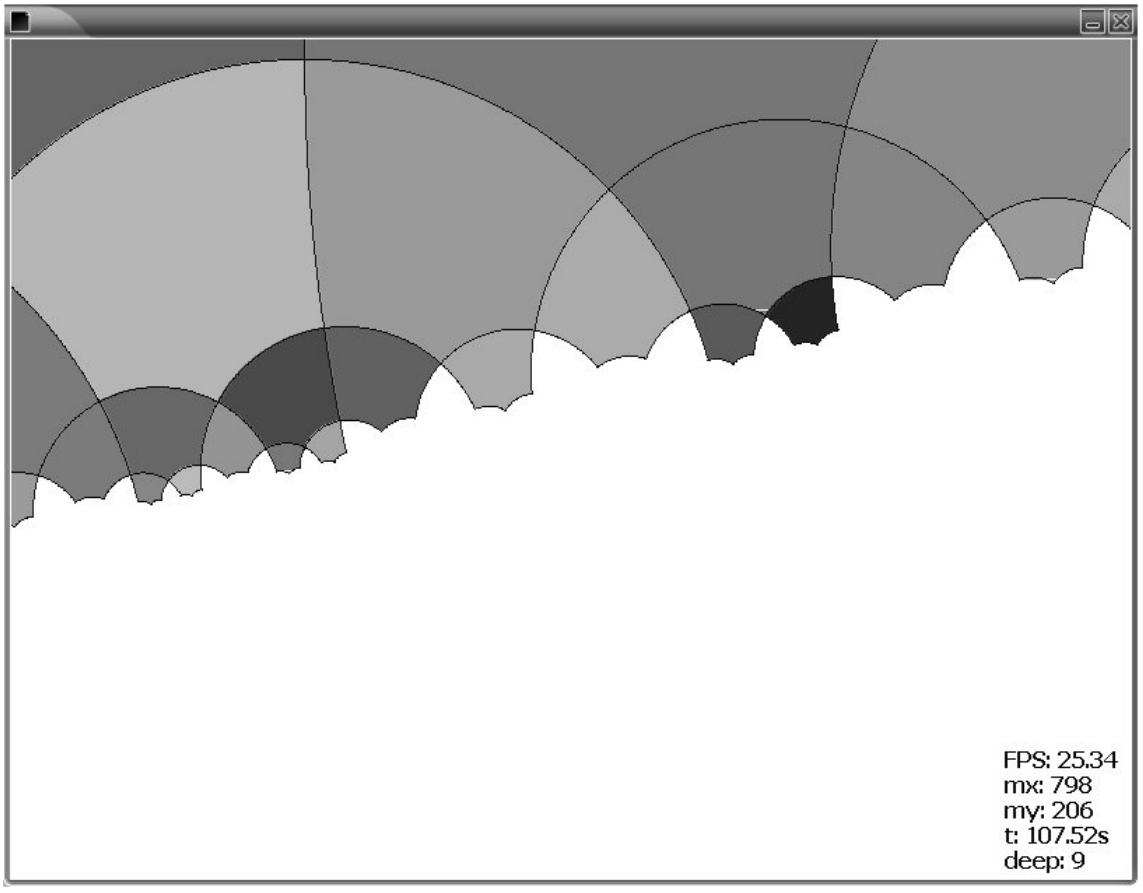

Figure A.2 – Affichage zoomé de la pentagrille au niveau 9

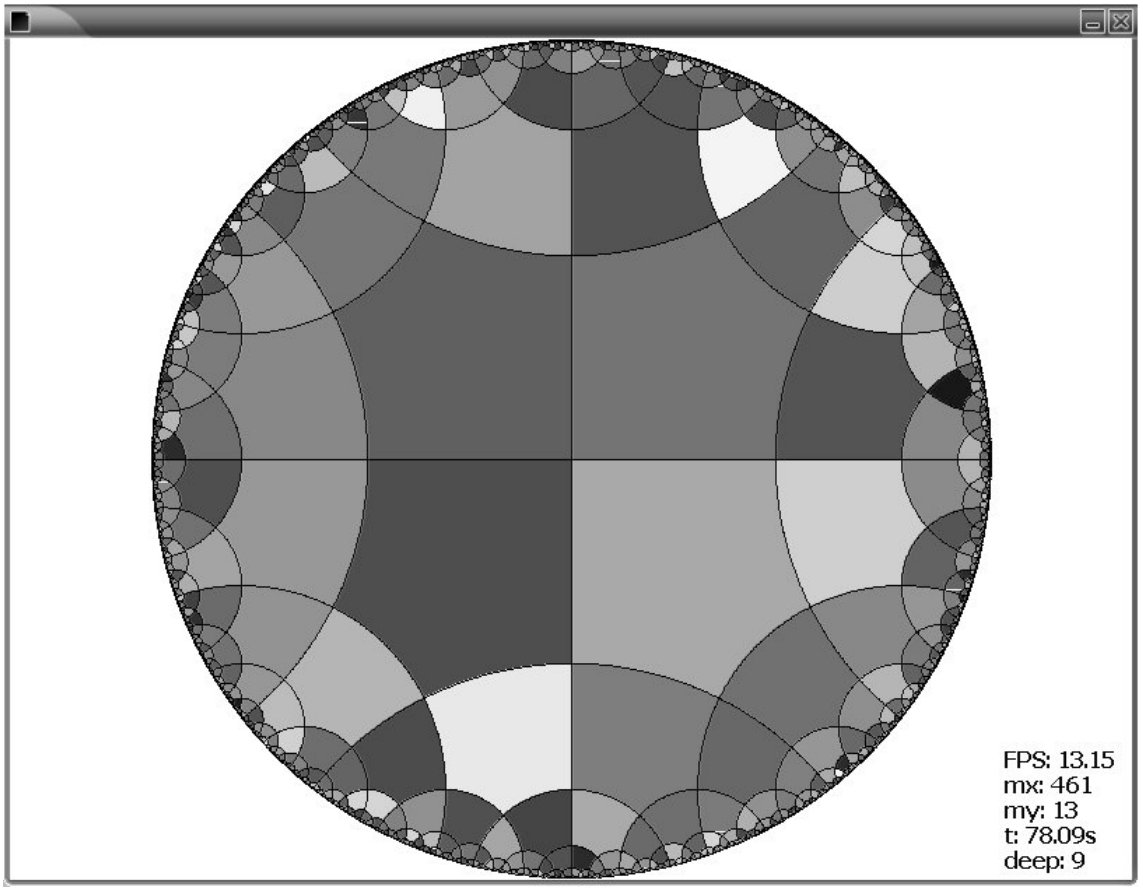

Figure A.3 – Affichage complet de la pentagrille au niveau 9

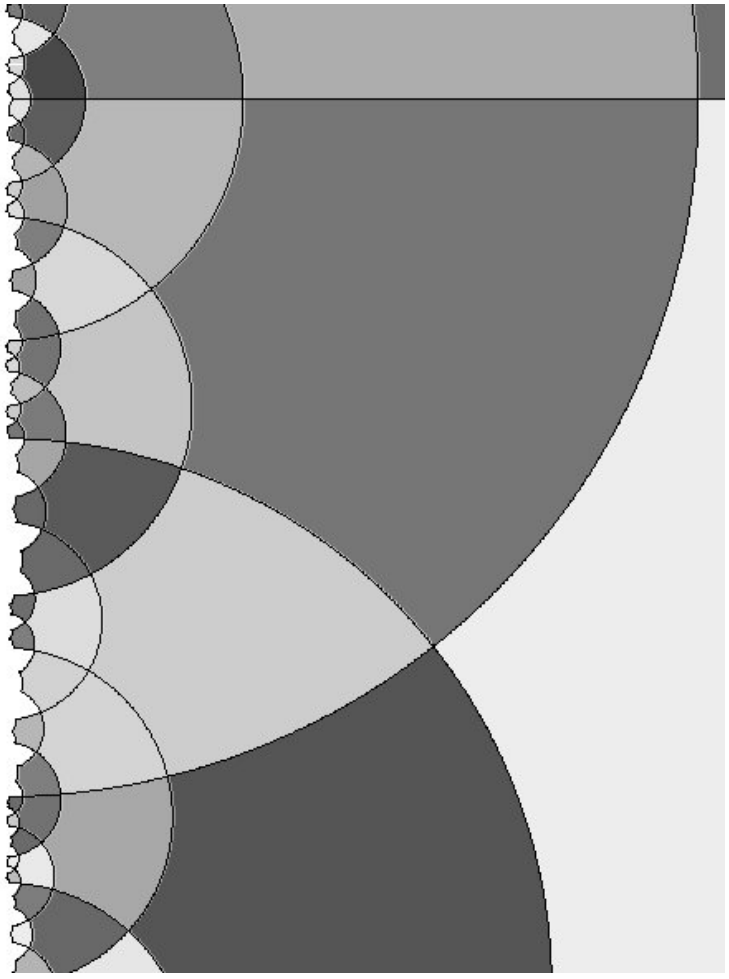

Figure A.4 – Exemple de zoom

Bibliographie

[1] Eckart J.D. *A cellular automata simulation system*, http ://www.cs.runet.edu/d̃ana/cellular.html, Radford University (1995).

[2] Imai K. Ogawa H. *A simulating tool for hyperbolic cellular automata and its application to construct hyperbolic cellular automata which can simulate logical circuits*, CA2000, Osaka, Japon (2000).

[3] Herrmann F., Margenstern M. *A universal cellular automaton in the hyperbolic plane*, Theoretical Computer Science, vol. 296 (2003), pages 327-364.

[4] Herrmann F., Margenstern M. *An interactive processing of cellular automata in the hyperbolic plane*, The 6th Conference on Systemics, Cybernetics and Informatics, SCI 2002, Orlando, Floride (2002).

[5] Mazoyer J., Delorme M. *Cellular Automata : A Parallel Model*, Kluwer Academic Publishers, vol 460 (1999).

[6] Mazoyer J. *Computations on Cellular Automata*, Rapport de recherche n. 98-34, publication du LIP, ENS Lyon (1998).

[7] Cormen T., Leiserson C., Rivest R., Stein C. *Introduction à l'Algorithmique*, Dunod (2002).

[8] Ramsay A., Richtmyer R.D. *Introduction to hyperbolic geometry*, Springer Berlin (1995).

[9] Margenstern M. *New tools for cellular automata in the hyperbolic plane*, Journal of Universal Computer Science, vol.6, num.12 (2000) pages 1226-1252.

[10] Margenstern M., Morita K. *NP-problems are tractable in the space of cellular automata in the hyperbolic plane*, Theoretical Computer Science, vol. 259 (2001).

[11] Durand B, Róka Zs. *The Game of Life : universality revisited* Rapport de recherche n. 98-01, publication du LIP, ENS Lyon (1998).

[12] Freiwald, U. Weimar J.R. *The Java based cellular automata simulation system - JCASIM*, Future Generation Computer Systems 18 (2002), pages 995-1004.

[13] Dijkstra J., Timmermans H. *Towards a multi-agent model for visualizing simulated user behavior the assessment of design preformance*, Automation in Construction 11 (2002), pages 135-145.

«

Cette documentation a été réalisée sous Linux à l'aide de LaTeX et cvs.»