Modélisation multi-échelle procédurale de scènes animées

Frank Perbet

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THÈSE
pour obtenir le grade de docteur de l'INPG
spécialité Imagerie Vision Robotique
préparée au sein du projet EVASION, laboratoire GRAVIR
dans le cadre de l'école doctorale
Mathématiques, Sciences et technologies de l'information, Informatique
présentée et soutenue publiquement le 26 février 2004 par

Frank Perbet

Modélisation multi-échelle procédurale de scènes animées

Directeur de thèse : **Marie-Paule Cani**
Co-directeur de thèse : **Przemyslaw Prusinkiewicz**

JURY

Augustin Lux,	Président
André Gagalowicz,	Rapporteur
Pascal Guitton,	Rapporteur
Marie-Paule Cani,	Directrice de thèse
Przemek Prusinkiewicz,	Co-directeur de thèse
Christophe Godin,	Examinateur
Fabrice Neyret,	Examinateur

Modélisation multi-échelle procédurale de scènes animées

Résumé

En synthèse d'images, les scènes tridimensionnelles animées sont de plus en plus riches et détaillées. Mais elles sont actuellement limitées dans les variations de leur échelle d'observation. Par exemple, créer des modèles permettant une ballade interactive passant continûment d'un atome à une galaxie pose de sérieux problèmes. L'objectif de cette thèse est d'y apporter une solution dans le cadre d'une visualisation temps-réel sur un matériel informatique standard.

Tout d'abord, nous montrons pourquoi la modélisation multi-échelle procédurale est particulièrement bien adaptée pour résoudre ce problème. Plus précisément, nous utilisons la modélisation par *complexification* qui décrit un modèle par une représentation grossière et par un ensemble de fonctions qui lui ajoutent localement des détails jusqu'à satisfaire la précision requise par des critères perceptuels. Nous introduisons une nouveau formalisme basé sur le langage C++ capable de décrire un large éventail de modèles 3D animés sur de grandes variations d'échelle. Nous proposons un outil générique qui implémente ce formalisme appelé DynamicGraph. Cet outil offre d'une part une interface graphique dédiée et d'autre part un algorithme de rendu temps-réel qui évalue efficacement la visibilité et de la précision requise. Nous illustrons par différentes études de cas le potentiel de cette approche.

Mots clefs : synthèse d'images, 3D, modélisation procédurale, multi-échelle, multi-résolution, arbre dynamique, temps-réel, visibilité

Multi-scale procedural modeling of animated scenes

Abstract

In computer graphics, 3D animated scenes are more and more rich and detailed. But the current techniques for managing such scenes cannot handle a wide range of observation scales. For example, creating models that allow to zoom from a galactic scale to an atomic scale is a very difficult task. Our goal is to address this problem in the scope of real-time visualization using standard computers.

First, we show that multi-scale procedural modeling is particularly well suited to solve this problem. More precisely, we use *complexification* modeling which describes a model by its coarse representation and a set of functions which adds local details until the required precision according to perceptual criteria. We introduce a new formalism based on the C++ language which is able to describe a large set of 3D animated models over a large range of scales. We propose a tool which implements this formalism called DynamicGraph. This tool consists of a specialized graphical interface and a real-time rendering algorithm which efficently evaluates the visibility and the required precision. Finally, we illustrate the potential of this approach by several case studies.

Keywords : computer graphics, 3D, procedural modeling, multi-scale, multi-resolution, dynamic graph, real-time, visibility

Préface

Vous avez dans les mains le résultat d'un peu moins de 4 ans de travail. C'est difficile de réaliser que toutes cette sueurs, tous ces doutes et ces remises en question, toutes ces prises de consciences, tout cela soit compressé en quelques dizaines de pages, une dizaine de milliers de mots...

Il me semble important de souligner que cette thèse puise son énergie dans la négation. Plûtot que de me baser sur l'existant et d'y apporter une amélioration, j'ai préféré utiliser toute l'indépendance dont j'ai disposé pour aller vers quelque chose de moins conventionnel. Le premier chapitre bibliographique de ce rapport de thèse est d'ailleurs essentiellement une critique reflétant l'énergie j'ai passé à traquer les défauts qui rendent inadéquates, pour le problème que je voulais résoudre, la plupart des approches existantes.

C'est épuisant de toujours repousser et contredire, et je crois que j'aurais pu passer toute ma thèse sur ce même rythme contradictoire. Heureusement, à la suite de mon séjour à Calgary, je me suis décidé à suivre une voie qui me paraissait aller suffisamment loin et dont, au moins, je ne percevais pas de contre-indication évidente. Les discussions avec Przemyslaw Prusinkiewicz et Christophe Godin ont été décisives dans ce nouvel élan qui allait conclure ma thèse.

Remerciements

Marie-Paule, tu m'as donné la possibilité de faire cette thèse. Je t'en remercie.

Marta, Christophe et Raph, vous avez été mes compagnons d'aventure. Vous resterez toujours associés à ces quatres ans que l'on a passé à INRIA.

Merci à Pascal Guitton et André Gagalowicz d'avoir accepté d'être rapporteur.

My journey in Calgary was a wonderful experience. Thanks Martin, Richard, Ying, Callum, Collin, Kay, Peter and all the graphics jungle lab.

Lars, our collaboration was short but intense !

Fredéric Boudon, il semblerait décidemment que nos destins soient liés...

Fabrice, tu as été pour moi une grande source d'inspiration.

Alors Alex Meyer, procédural ou précalculé ?

Patricia ton aide m'a été précieuse.

Joëlle, merci pour tout. Gilles, idem.

Benjamin et Sandrine, je me demande comment j'aurais fait sans vous !

Très chère famille, très chers amis, votre soutien est inestimable.

Laure et Philippe, ça me rend triste de quitter notre bureau.

Mais voila, toutes les bonnes choses ont une fin, et d'autre bonnes choses doivent bien avoir un commencement !

Table des matières

Chapitre 1

Introduction

Grâce à la synthèse d'images, l'utilisateur devient, via des périphériques comme l'écran et la souris, à la fois observateur et acteur de mondes de synthèse que l'on dit virtuels. Insatiable, nous voulons toujours plus d'objets, d'animations et d'interactions. Qu'elles soient d'intérieur ou d'extérieur, les scènes sont toujours plus complexes : les formes et les mouvements s'enrichissent et l'immersion n'en est que plus crédible.

Parmi les caractéristiques contribuant à la richesse d'un monde virtuel, la *précision* joue un rôle fondamentale (cf. figure 1.1). Celle-ci est la motivation d'un des plus beaux défis lancé à l'informatique graphique : la modélisation et le rendu *multi-échelle*. L'objectif de cette thèse est l'affichage de scènes tridimensionnelles avec de grande, variation, dans l'échelle d'observation.

Ce chapitre est composé de trois sections :

- la section 1.1 argumente la nécessité de créer des mondes virtuels observables à différentes *échelles* ;
- la section 1.2 décrit le présent document dans ses grandes lignes et expose ses principales *contributions* ;

1.1 La *multi-échelle* : une manière d'observer le monde

Dans cette section, nous nous attarderons d'abord sur les *limites* biologiques de nos perceptions. Nous décrirons ensuite une expérience imaginaire : un *voyage* à travers les échelles de l'univers. Ensuite, nous expliquerons (très succinctement !) comment l'Homme dépasse ses limites par l'utilisation d'*outils*.

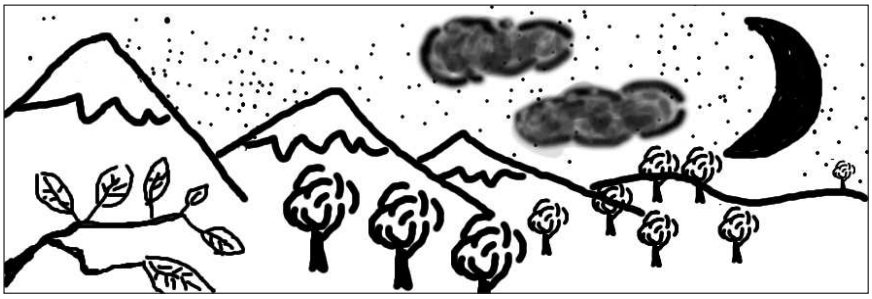


FIG. 1.1 – La richesse du monde réel tient en grande partie aux variations d'échelle qu'il offre à nos yeux.

1.1.1 Les limites de l'Homme

La réalité nous offre un spectacle d'une complexité infinie. Dommage que nous soyons si mal loti pour en profiter...

Limites corporelles

D'un point de vue biologique, nos limites sont incontournables. Nous nous déplaçons lentement. Dans nos moments de gloire aéronautique, nous fonçons à quelques dizaines de kilomètres par seconde, ce qui nous laisse à près d'un million d'année de l'étoile la plus proche après le soleil. Nos moyens de transport favoris, tels la marche à pied, le vélo ou la voiture, nous permettent de nous déplacer à des vitesses un milliard de fois inférieures à celle de la lumière.

L'œil est l'un des organes les plus sophistiqués du corps humain. Pourtant, nous sommes, pour ainsi dire, pratiquement aveugles. Notre champ de vision a un angle solide d'environ 180°. Malheureusement, nous n'y voyons précisément que dans un angle de 5°. Nous ne distinguons quasiment rien en dessous du millimètre. Les couleurs que nous percevons sont une très grossière approximation des rayons électromagnétiques qui nous entourent. Nous percevons l'écoulement du temps à peu près toujours au même rythme et les mouvements trop rapides nous sont imperceptibles.

Notre spectre de perception est ridiculement petit. Si nous n'avons pas conscience de ces limites, c'est qu'il en a toujours été ainsi.

Limites intellectuelles

Les limites de notre corps induisent irrémédiablement des limites intellectuelles. En effet, notre cerveau se forge en partie grâce aux signaux extérieurs qu'il reçoit. Par exemple, il est notoire que beaucoup de musiciens ont une bonne oreille. Il est bien sûr possible d'améliorer la qualité de son audition. Néanmoins, l'acuité d'une oreille favorise sans aucun doute le développement d'une certaine intelligence musicale[1].

La stimulation neuronale est la base de l'intelligence, c'est une forme évoluée du principe de "action-réaction". Avides de nouvelles sensations, nous cherchons, par le moyen de notre corps, cette nourriture de l'esprit. Il est décidément regrettable que nous soyons intrinsèquement contraints par un corps dont aux facultés de perception si limitées.

1.1.2 Une ballade

Débarrassons nous de nos contraintes corporelles, notre imagination est faite pour cela. Je vous invite à devenir un super-observateur. Imaginez un outil capable de se transformer soit en microscope, soit en télescope, en passant par tous les intermédiaires possibles. Rajoutez à cela une paire d'ailes : vous êtes le premier voyageur multi-échelle.

Plus gros

Notre voyageur est sur Terre, il regarde la Lune. Il décolle et s'en rapproche en quelques secondes, la voyant grossir pour progressivement occuper tout son champ de vision. Il s'arrête et regarde la terre. Il se dit qu'on l'a tellement vue à la télévision que ce n'est même pas impressionnant, alors il regarde le soleil et fait marche arrière, très vite, si bien que bientôt il voit aussi toutes les planètes du système solaire. Alors il fait un tour sur Jupiter parce que c'est la plus grosse mais comme il se rapproche il voit que ce n'est qu'une boule de gaz dans laquelle il s'enfonce comme dans un brouillard.

Il décide ensuite d'aller plus loin, toujours en marche arrière. Le brouillard de Jupiter s'estompe, le système solaire s'enfuit. La voie lactée est toujours visible, comme depuis la terre : c'est une vaste ceinture transparente. Une multitude d'étoiles entoure le voyageur, réparties aléatoirement autour de lui. Mais alors qu'il continue à reculer, il constate qu'il laisse derrière lui un amas d'étoiles dont il vient juste de sortir. Toutes les étoiles le constituant se rassemblent comme un nuage de points

[1] A l'autre extrême, une personne née sourde aura bien du mal à composer la moindre symphonie.

lumineux toujours plus dense. Se concentrant sur ce nuage, il oublie de regarder autour, ce qui est bien dommage, car il s'écarte de la galaxie et un nouveau nuage prend forme sous ses yeux, toujours plus dense, en forme de spirale.

Alors il visite d'autres galaxies, s'en rapprochant à des vitesses bien supérieures à celle de la lumière et ralentissant au fur et à mesure qu'elles se rapprochent de sorte qu'il ne faut que quelques secondes pour sauter d'une galaxie à une autre. Il regarde encore quelques objets célestes, nébuleuses, quasars et même quelques trous noirs. Alors il recule encore...

Plus petit

De retour sur terre, notre voyageur n'est pas devenu fou, ce qui est remarquable compte tenu toutes les choses qu'il vient de voir. Après cette longue ballade, il veut se reposer. Mais pendant qu'il dort, il se fait piquer par un moustique et se réveille. Le voyageur se dit qu'il a envie de voir à quoi ressemble l'insecte qui vient de s'envoler plein de son sang. Alors il rappetisse et s'envole à sa poursuite. Celui-ci, vu de près, n'est pas très joli, plutôt effrayant même.

Alors il se rapproche de son œil et devient si petit que le moustique lui-même ne peut le voir. L'œil du moustique devient une grande surface lisse au début mais très vite de nombreuses irrégularités apparaissent. En se rapprochant encore, une multitude de petites briques surgissent, les cellules. Notre voyageur en choisit une et entre dedans. Une immense machinerie est en marche et de grosses molécules vaquent à leurs occupations.

Le voyageur se concentre sur le noyau dans lequel il distingue un long filament. Il se rapproche et perçoit alors la structure du filament : celui-ci est enroulé sur lui même. Il se rapproche d'un enroulement et constate qu'un second niveau d'enroulement succède au premier, puis encore un... Enfin, il perçoit une structure hélicoïdale, l'ADN. De près, on distingue les molécules qui la composent.

En se rapprochant de celles-ci, il distingue les atomes de chaque molécule et constate que ceux-ci sont en fait formés de plusieurs petites boules : les neutrons et les protons. En se rapprochant encore, il constate que ces boules sont en faits constituées de trois petites boules, les quarks. Il décide alors de se rapprocher des quarks pour voir de quoi ils sont constitués...

1.1.3 Des outils pour se dépasser

Soyons clairs, la ballade virtuelle décrite plus haut n'a pas été réalisée au cours de cette thèse. Même si le travail présenté ici va dans cette direction, la route est longue. Avant de générer un monde aussi complexe, il faut nous munir des bons outils.

Surmonter nos limites

C'est grâce à l'outil que l'Homme se dépasse. Avec nos mains, nous avons aiguisé des pierres pendant plusieurs millions d'années. C'est le temps qu'il nous a fallu pour parvenir aux premières abstractions : "Je donne un coup de pierre là, puis un coup là, comme ça j'aurais plus de chance de ne pas casser la pierre avec le dernier coup que je donnerai là"... Il nous a fallu environ 10 millions d'années pour formuler une telle phrase.

A l'échelle du temps géologique, l'ordinateur vient tout se suite après la pierre. C'est l'outil le plus sophistiqué que nous ayons jamais conçu. Grâce à lui, on peut automatiser de nombreuses tâches ennuyeuses et se concentrer sur l'essentiel. Il nous permet de communiquer et de coopérer. Les jeux vidéos nous font vivre des expériences que la réalité ne peut pas nous donner : nous conduisons des engins imaginaires à des vitesses folles, construisons et détruisons des empires, devenons Dieu ou le Diable...

Bonne ou mauvaise, une chose est sûre : l'informatique élargit le spectre de nos perceptions. Elle met à notre disposition une gigantesque quantité de médias et nous aide, dans une certaine mesure, à ne pas nous perdre dans cet océan d'informations.

Submergé

Toujours plus, toujours plus vite. L'explosion des technologies "multimédias" a inexorablement conduit à une énorme production d'informations. La quantité de son, d'image et de texte produite durant les 50 dernières années dépasse toute celle produite depuis l'aube de l'humanité. Un tel raz-de-marée pose quelques problèmes à nos petits cerveaux.

Devant une telle affluence, notre réaction varie entre l'abrutissement et l'abstinence. Heureusement, ces deux extrêmes ne sont pas les seules solutions. Pour arriver à percevoir dans tout ce brouhaha un peu de pertinence, il nous faut prendre le temps de discerner, d'analyser et de prendre du recul.

Prendre du recul et aller à l'essentiel : nous devons apprendre à *naviguer*. Et bien sûr, pour nous aider dans cette tâche, de nouveaux outils doivent nous assister et incidemment former notre intelligence à ce nouvel exercice que l'on pourrait nommer : la perception multi-échelle.

Perception multi-échelle

La lutte pour la perception multi-échelle a déjà commencé. En fait, elle existe depuis que l'Homme existe : nos capacités d'abstraction sont l'une des caractéristiques de notre espèce. Simplement le besoin a grandi et plus que jamais, nous avons besoin de hiérarchiser nos pensées. Il nous faut acquérir la souplesse de sauter de concept en sous-concepts, d'argumenter le général et le détail, sans jamais perdre de vue l'ensemble.

Par exemple, peut-être vous souvenez-vous avoir appris à l'école que le corps est constitué de cellules. Puis quelques semaines plus tard est venu une seconde révélation : les cellules sont constituées de molécules qui sont elles-mêmes constituées d'atomes. Personnellement, je suis resté perplexe devant ce discours. Je ne remettais pas la chose en question, mais j'avais beaucoup de mal à imaginer mon corps immense comparé à un atome et ridiculement petit comparé à une galaxie.

Essayons d'imaginer l'outil adéquat dans la situation de l'écolier face à l'infiniment grand et l'infiniment petit. La difficulté ici n'est pas de se familiariser avec l'existence des galaxies et des atomes. En effet, quelques images ou schémas bien sentis permettent d'appréhender de tels objets *en les ramenant à notre échelle*. La difficulté réside dans le passage d'une échelle à l'autre. Comment rendre compte de variations d'échelle de plusieurs dizaines d'ordres de grandeur ?

Mais pour nous entraîner à naviguer entre le détail et le général, j'ai la conviction qu'une aide visuelle serait du plus grand secours. Pour revenir à l'exemple de l'écolier, un film similaire à la ballade décrite en 1.1.2 serait parfaitement adéquat. Un observateur se déplacerait de l'autre bout de l'univers jusqu'à une molécule ADN d'un arbre terrien.

1.2 Vue d'ensemble

Dans cette section, les principaux problèmes liés à la modélisation et à l'affichage de scènes avec de *grandes variations d'échelle* sont énumérés, ce qui nous conduit à préciser les objectifs de ce travail. Ensuite, les principales contributions de cet ouvrage sont brièvement décrites. Enfin, nous présenterons l'articulation du document.

1.2.1 Contexte

L'observation multi-échelle exacerbe les problèmes d'*aliasing*[2] et leurs conséquences néfastes. Dans le cas d'application avec de grandes variations d'échelle, les méthodes de simplification sont difficilement utilisables. En revanche, la modélisation procédurale, bien que délicate à mettre en oeuvre, est plus adéquate.

[2]Compte tenu de l'emploi très rare de la traduction française du terme *aliasing*, le terme anglais sera utilisé tout au long de ce document.

La source du problème : l'*aliasing*

Lors de la modélisation, un modèle 3D est la plupart du temps échantillonné dans l'espace à une certaine *résolution*. Lors d'une observation du modèle, celui-ci est usuellement projeté sur un écran, c'est dire une grille de pixels 2D à une autre résolution. Si le pas d'échantillonnage du modèle 3D après projection et à peu près celui de l'écran, l'affichage se passe plutôt bien.

Lors d'observations multi-échelles, on autorise la caméra à se rapprocher ou à s'éloigner du modèle. Ce genre de déplacement fait varier le pas d'échantillonnage du modèle 3D après projection. En conséquence, si celui-ci est constant dans l'espace 3D, il existe forcément des positions de caméra pour lesquelles l'écart entre la résolution du modèle projeté et celle de l'écran est grand. L'affichage souffre alors inexorablement des fameux problèmes d'*aliasing* [Whi81] et la qualité du rendu se dégrade.

De très nombreux travaux proposent des *méthodes multi-échelles* pour garder le pas d'échantillonnage de la projection du modèle 3D le plus proche possible du pas d'échantillonnage de l'écran.

Limitation des méthodes de simplification

De nombreuses méthodes multi-échelles supposent la connaissance du modèle à sa précision maximum pour en extraire automatiquement des versions *simplifiées*. Lors de l'affichage, on choisira le bon degré de simplification en fonction de la distance à la caméra. Ce sont les *méthodes de simplification* ; elles seront étudiées dans le chapitre 2.

Simplifier automatiquement un modèle en gardant de bonnes propriétés visuelles devient de plus en plus difficile à mesure que le modèle initial est précis. Symétriquement, les simplifications extrêmes sont de mauvaise qualité. De plus, le coût mémoire nécessaire à une description fine d'un modèle n'est pas acceptable lorsque qu'une grande précision est requise. Intuitivement, un modèle observé de 1000 fois plus près devra être 1000 fois plus précis, et donc environ 1000^3 fois plus volumineux en mémoire (dans le cas tridimensionnelles). Enfin, pour assurer un affichage efficace, ces méthodes reposent généralement sur des précalculs lourds limitant sérieusement les possibilités d'animation.

Modélisation procédurale : une approche laborieuse

Certaines méthodes décrivent un modèle par des fonctions plutôt que par des données brutes. Ce sont les méthodes *procédurales* ; elles seront étudiées dans le chapitre 3. Appliquées à la multi-échelle, elles permettent de ne pas décrire un modèle à son niveau le plus fin et d'échapper ainsi aux limitations des algorithmes de *simplification*.

Des algorithmes de *modélisation procédurale multi-échelle* existent notamment depuis 1975 [Man75] sous le nom de *fractales*. Les objets fractales sont très répétitifs et non-animés. De rares travaux ont récemment permis d'introduire l'animation et de diminuer la répétitivité en ajoutant plus de contrôle. Mais chacun d'entre eux est appliqué à un cas particulier et à chaque fois leur mise en œuvre est laborieuse.

Notre objectif

Notre objectif est simple et ambitieux : nous voulons modéliser et afficher des modèles 3D sans contraintes sur les variations d'échelle.

Nous désirons modéliser des phénomènes *massivement animés* (prairie et arbre sous le vent par exemple). Toute structure algorithmique trop statique, nuisant au possibilité d'animation, sera donc évitée.

Nous visons des performances temps-réel, mais aussi un contrôle fin sur la qualité requise lors de l'affichage. Ces deux critères sont inexorablement antinomiques. Lorsque la qualité augmente, les performances diminuent : c'est en quelque sorte le produit $Qualité \times Rapidité$ qui est constant. Nous voulons donc offrir le maximum de contrôle sur ce coefficient $Qualité \times Rapidité$ et notamment permettre de régler celui-ci pour un rendu temps-réel. Remarquons ici que le terme *qualité* n'est

pas défini rigoureusement car il dépend de valeurs perceptuelles difficilement quantifiables. Par conséquent, le terme *Qualité* × *Rapidité* est lui-même assez approximatif. Néanmoins, j'ai choisi de l'utilisé dans cette ouvrage puisqu'il exprime une balance certe imprécise mais bel et bien réelle.

1.2.2 Contributions

Très généralement, nos contributions sont une analyse du problème posé par la multi-échelle, l'introduction d'un formalisme pour la modélisation par complexification et une mise en pratique de ce concept.

Analyse du problème

Nous avons identifié les méthodes de simplification comme étant intrinsèquement incapables d'assurer globalement un rendu temps-réel de scènes complexes et animées avec de grandes variations d'échelle.

La réalisation d'un algorithme temps-réel de rendu de prairie sous le vent nous a fait prendre conscience des avantages de la *modélisation procédurale multi-échelle*.

En revanche, cette approche nécessitait jusqu'à présent une modélisation au cas par cas laborieuse compte tenu de l'absence d'outil dédié à cette tâche.

Représentation par *complexification*

Nous montrons que les problèmes d'affichage multi-échelle sont intimement liés au *langage descriptif* (ou représentation) choisi pour représenter les modèles.

Nous proposons un nouveau langage descriptif procédurale multi-échelle que nous appelons *représentation par complexification* permettant de décrire une très grande variété de formes tridimensionnelles.

Ce langage impose de décrire un modèle par un ensemble de fonctions ajoutant de la précision à une forme tridimensionnelle initiale. Ces fonctions enrichissent non seulement l'apparence d'un modèle, mais aussi son animation.

Un outil générique pour la modélisation multi-échelle

Un nouvel outil de modélisation, nommé Dynamic Graph, offre un environnement de travail facilitant la modélisation par complexification pour une grande classe d'objets tridimensionnelles. Deux intervenants extérieurs l'ont testé pendant deux mois et ont notamment réalisé un modèle d'arbre multi-échelle animé.

Nous proposons un nouvel algorithme de rendu qui permet un affichage des modèles avec de *grandes variations d'échelle*. Afin d'assurer un rendu efficace, il rejoue les fonctions ajoutant des détails données lors de la modélisation jusqu'à satisfaire la précision requise par l'observateur. Des réglages sur la précision permettent un excellent contrôle sur le coefficient *Qualité* × *Rapidité*. Un nouvel algorithme de détection d'occultation permet, *sans aucun précalcul*, d'ignorer les parties occultées de scènes animées.

1.2.3 Articulation

Processus de création

Cette thèse est bâtie autour du *processus de création* schématisé dans la figure 1.2. Celui décrit l'acte de modélisation de la façon suivante :

- le *créateur* imagine un objet ;
- grâce à une *interface*, il décrit cet objet par un langage descriptif ;
- la représentation de l'objet est alors *évaluée* pour satisfaire une observation particulière : celle de la caméra ;

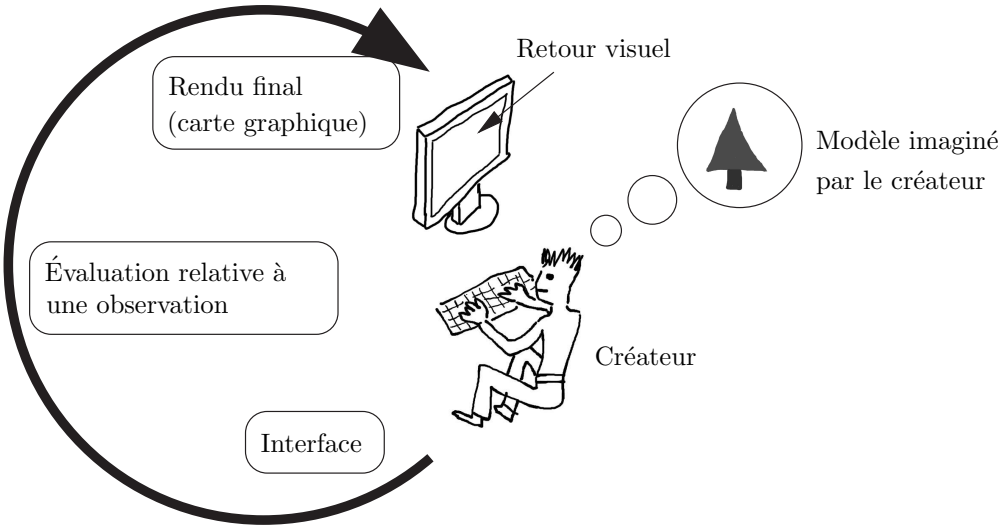


FIG. 1.2 – Cet ouvrage est entièrement pensé autour du *processus de création*. Ce schéma sera utilisé à plusieurs reprises et joue en quelque sorte le rôle de boussole.

- lors de cette évaluation, la nature de la représentation change et termine notamment sa course sous la forme de géométrie puis enfin de pixels ;
- le créateur peut alors contempler son œuvre dont il contrôle les paramètres ;
- il peut éventuellement modifier le modèle ou bien l'observation : chacune de ces deux actions déclenche une nouvelle évaluation.

Plan du document

Cette thèse est composée des chapitres suivants (cf. figure 1.3) :

Simplification : diverses méthodes existantes de simplification appliquées tout au long de la partie finale du processus de création sont étudiées. Nous concluons par l'incapacité de ces méthodes à assurer un rendu avec de grandes variations d'échelle ;

Complexification : une étude des méthodes de modélisation procédurale multi-échelle existantes est suivie de l'introduction du concept de *complexification*. Nous démontrons le besoin d'un outil générique pour la modélisation et le rendu par complexification.

Prairie animée en temps-réel : L'étude d'un cas particulier appuie la conclusion du chapitre précédent ;

Dynamic Graph : principe : nous présentons Dynamic Graph, un nouvel outil générique d'aide à la modélisation par complexification.

Dynamic Graph : évaluation : nous décrirons l'évaluation d'un modèle relativement à une observation et sous la forme d'un arbre dynamique évoluant au fil du temps ;

Dynamic Graph : modélisation : après une description de la partie *modélisation* de Dynamic Graph, nous présenterons un modèle d'arbre animé par le vent.

Conclusions et perspectives : tout est dit.

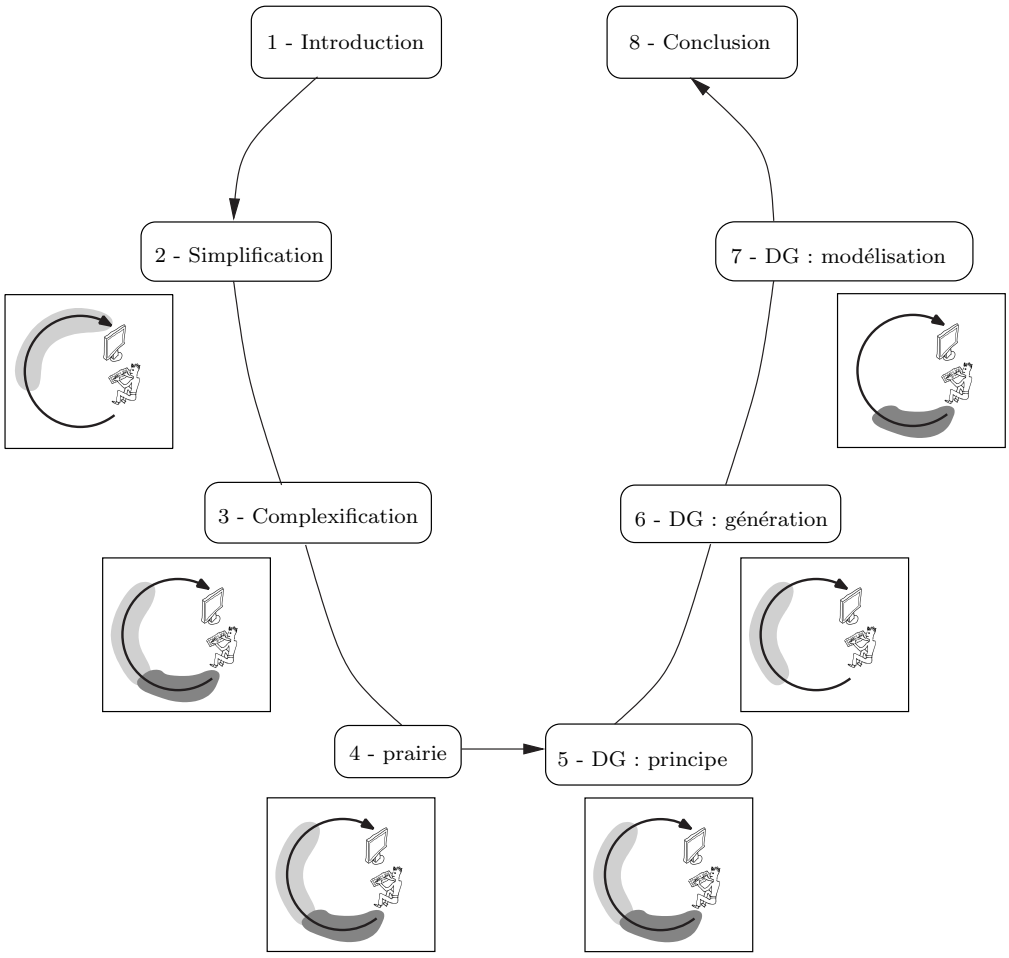


FIG. 1.3 – Ce schéma représente les différents chapitres de cette thèse et l'aspect du processus de création auquel ils s'intéressent. La *simplification* agit à la fin du processus, jusque dans la carte graphique. La modélisation par complexification, comme nous le verrons, prend sa source lors de la phase de modélisation, puis elle est évaluée pour une observation. La prairie est un exemple de complexification. Dynamic Graph est un outil générique de modélisation par complexification présenté en trois chapitres : principe, évaluation et modélisation.

Chapitre 2

Simplification

Ce premier chapitre d'état de l'art décrit les méthodes de *simplification*. Il motive ainsi les méthodes procédurales, plus rares et plus proches de nos propres travaux, dont l'état de l'art sera fait au chapitre suivant.

Les méthodes de simplification agissent à différents moments du processus de création (cf. figure 2.1). Nous les décrirons dans l'ordre inverse du flux d'information envoyé par le créateur. Nous commencerons par les méthodes de simplification prenant place au sein même de la carte graphique. Nous continuerons par celles s'appliquant à la représentation du modèle dans le processeur central.

Notons ici que cette partie, et tout ce mémoire de thèse en général, se place dans le contexte d'une utilisation intensive des cartes graphiques actuelles. Celles-ci sont toutes basées sur une architecture en *pipe-line* et sur un flux d'information monodirectionnel. Des formes géométriques simples sont décomposées en fragments qui vont s'amonceler sur le tampon de couleur [WND99]. Une connaissance de ces mécanismes est fortement recommandée pour la lecture de ce document.

2.1 Affichage graphique et multi-échelle

Tout à la fin du processus de création (cf. figure 2.1), au moment du rendu final, l'information envoyée par le créateur existe au sein de la carte graphique sous une forme brute : les fragments et les pixels (cf. figure 2.2). À ce niveau, les difficultés posées par l'affichage d'objets vus à différentes échelles sont connues sous le nom d'*aliasing*.

Après une rapide description des problèmes causés par l'*aliasing*, nous décrirons les méthodes d'*anti-aliasing*. Nous listerons ensuite d'autres techniques susceptibles de prendre en charge une partie du problème. Nous conclurons en identifiant les raisons pour lesquelles le matériel graphique

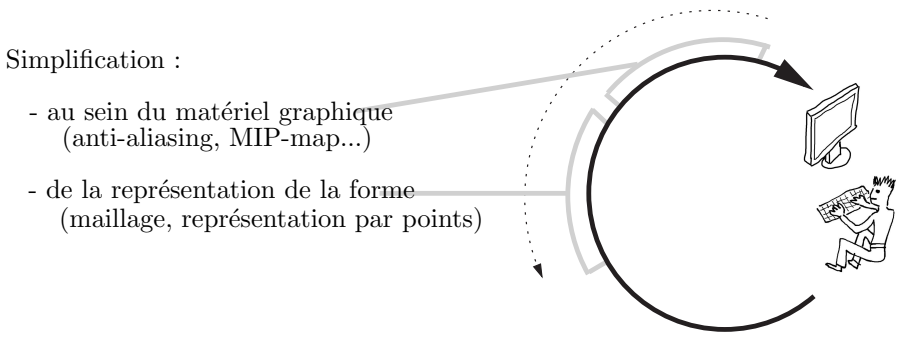


FIG. 2.1 – En partant d'une description des problèmes d'*aliasing*, nous décrirons les méthodes y remédiant en remontant le processus de création.

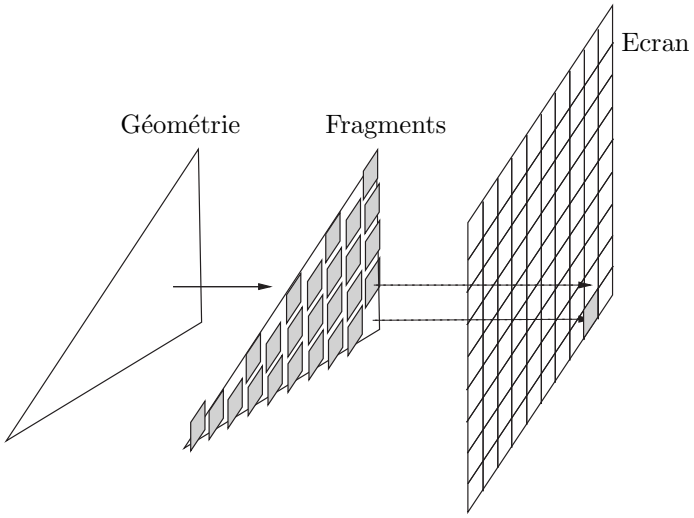


FIG. 2.2 – La géométrie est envoyée à la carte graphique. Elle est projetée et décomposée en fragments [WND99], c'est à dire rasterisée. Puis ces fragments sont envoyés sur l'écran.

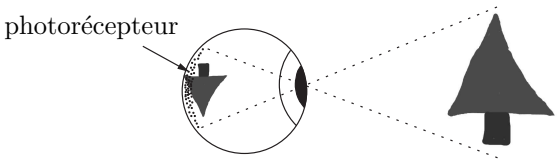


FIG. 2.3 – Les photo-récepteurs tapissant le fond de nos yeux (les cônes et les bâtonnets) transforment les signaux lumineux en signaux électriques qui sont envoyés au cerveau. Vu de la synthèse d'images, nos yeux sont de redoutables filtres supprimant en temps-réel la quasi-totalité des problèmes d'*aliasing* temporel et spatial.

ne peut pas, à lui tout seul, résoudre les problèmes posés par la multi-échelle.

2.1.1 L'*Aliasing* : la source du problème

Lorsque la résolution de la projection d'un modèle 3D sur l'écran est très différente de la résolution de la grille de pixels de l'écran, la qualité de l'affichage est parasitée. Lorsque cette Ce phénomène est appelé *aliasing*.

Réel et virtuel

Dans le monde réel, nos yeux traitent très efficacement tous les signaux qu'ils reçoivent. Les récepteurs tapis au fond de la rétine réagissent aux photons lui parvenant et filtre ce signal spatialement et temporellement (cf. figure 2.3). Ce filtrage, qui est une sorte de simplification, s'effectue à coût constant et ne dépend pas de la complexité du monde observé.

Dans les mondes virtuels que nous générons, cela ne se passe pas de la même manière. Tout d'abord, le monde n'existe pas au sens "physique" du terme. Il est remplacé par des données qui sont transformées tout le long du processus de création (cf. figure 2.1) pour finalement devenir des pixels sur l'écran. L'écran et la carte graphique sont en quelque sorte un *œil intermédiaire* qui observe le monde virtuel et qui est lui-même observé par le créateur. L'écran joue ici le rôle d'œil "intermédiaire". L'onde lumineuse devient une métaphore du flux d'informations que génère

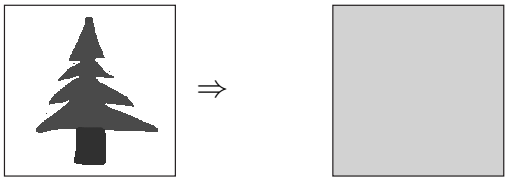


FIG. 2.4 – Le pixel doit avoir une couleur représentative des formes visuelles qu'il contient. Si vous mettez ce livre très loin, ces deux pixels deviendront similaires (cette affirmation est très probablement fausse, mais cela n'enlève rien au message qu'elle porte).

l'algorithme. Ce flux, juste avant de percuter l'écran, existe sous la forme de fragments, c'est-à-dire d'éventuels futurs pixels. Les difficultés liées à ce fonctionnement résident essentiellement dans la discrétisation de l'écran.

Revenons un instant au monde réel : les récepteurs de la rétine filtrent le signal lumineux quelque soit sa complexité. Cette intégration est essentielle et nous permet d'apprécier une valeur représentative d'un signal infiniment plus complexe issu de la réalité. En effet, les ondes lumineuses du monde réel sont beaucoup plus complexes que celle que nos yeux envoient au cerveau. Par exemple, lorsque nous regardons un tee-shirt rose, les photons qui viennent vers notre oeil sont descriptifs de détails très précis tel que les fibres ou même plus. En revanche, le signal que nos yeux envoient au cerveau est qu'une surface lisse et de couleur quasi-uniforme : les fibres ont disparues au profit d'un signal grossier mais néanmoins représentatif, c'est à dire rose.

Dans les mondes virtuels, ce filtrage est tout aussi importante et les pixels allumés sur l'écran doivent absolument être représentatifs du signal plus précis tel qu'il a été décrit par le créateur (cf. figure 2.4). D'une façon ou d'une autre, il faut donc retrouver, lors de l'observation, la précision qui correspond le mieux à cette observation. Pourtant, le créateur ignore à quelle précision va être observée la forme qu'il crée. Il doit donc faire en sorte que cette forme soit adaptative et puisse passer d'une précision à l'autre.

Sur l'écran

Venons en à des considérations plus techniques. La synthèse d'images, en un sens, n'est ni plus ni moins que l'art de discrétiser des fonctions sur une grille régulière : l'écran. La difficulté vient du fait que l'on ne désire pas vraiment calculer la fonction en chacun des *points* de l'écran, mais sur *la surface* des petits carrés que constitue chaque pixel. C'est donc une sorte d'intégrale des couleurs qu'il est nécessaire de faire en chacun des pixels.

Par exemple, si trop de fragments (de candidats au pixel) viennent sur un même pixel, de sérieux problèmes surgissent. Le pixel ne sait plus que faire de toute cette information et décide plus ou moins aléatoirement de celle qui sera finalement retenue. Il commettra ainsi une erreur à chaque affichage. De plus, rien n'assure que cette erreur reste la même d'un affichage à l'autre. Cela conduit irrémédiablement à des clignotements qui détériorent la qualité du rendu. On obtient alors *autre chose* que ce que l'on souhaitait, d'où l'utilisation de l'expression : problème d'*aliasing* (la base latine de "alias" signifiant "autre"). Une traduction française d'*aliasing* pourrait d'ailleurs être *altération*.

Errare humanum est

Pour lutter contre l'*aliasing*, de nombreuses méthodes ont été mises au point. Notamment, le sous-ensemble de ces méthodes traitant l'information quasiment au niveau des pixels est nommé *anti-aliasing*. Il est amusant de constater que toutes les méthodes luttant contre l'*aliasing* n'ont pas droit à cette noble appellation. Peut-être est-ce rassurant de cantonner ce problème loin de l'Homme dans le processus de création. Une façon de dire, peut-être, que «c'est la faute de l'ordinateur si ça ne marche pas»...

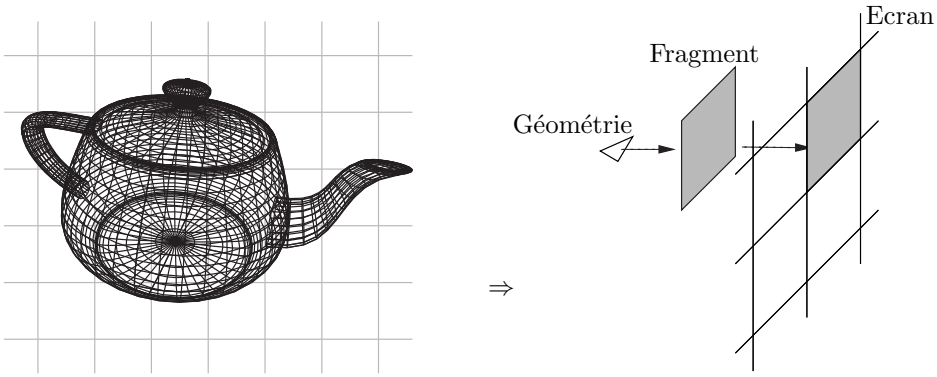


FIG. 2.5 – Exemple classique de rendu provoquant de sévères problèmes d'*aliasing*. La scène est sur-échantillonnée et la taille des triangles est bien inférieure à celle des pixels. Le fragment généré pour un triangle sera sur-dimensionné et ne sera pas un bon représentant du pixel.

Nous verrons dans ce chapitre comment, en voulant à tout prix éviter l'*aliasing*, de nombreuses méthodes ont remonté progressivement tout le processus de création pour finalement nous toucher directement. Bien entendu, ce n'est pas l'ordinateur le fautif. Les causes du problème prennent leur source dans notre cerveau, dans la simple manière dont nous élaborons un modèle virtuel observable à différentes échelles (cf. chapitre suivant).

2.1.2 Méthodes d'*anti-aliasing*

De nombreux fragments transitent vers le tampon de couleur dans l'espoir de devenir pixel. Quand plusieurs fragments revendiquent la place, les problèmes d'*aliasing* apparaissent. Afin d'y pallier, il faut trouver un moyen de relaxer un peu le critère de choix et de choisir non plus un fragment, mais plusieurs fragments dont l'influence est judicieusement pondérée.

Une approche naïve

Comment assurer un rendu correct d'une information échantillonnée trop finement pour les pixels de l'écran ? La réponse théorique, contrairement à sa mise en pratique, est simple : il suffit d'augmenter la précision de l'écran. Physiquement, bien sûr, c'est impossible. Mais on peut toujours imaginer des algorithmes permettant de simuler un écran virtuel avec des pixels plus petits que ceux de l'écran réel. Pour revenir à l'écrean réel, il suffirait de filtrer en intégrant tout les sous-pixels virtuels dans le pixel réel correspondant. Ainsi, après un filtrage adéquat de l'écran virtuel on retrouverait la taille de l'écran réel et on supprimerait les effets indésirables du sur-échantillonnage.

Mais le coût mémoire d'un tel écran virtuel ainsi que le coût temporel de la fonction de filtrage rend impossible une telle réalisation avec un facteur d'échelle trop important. Supposons par exemple que la géométrie soit 10 fois plus petite que la taille du cône de vision d'un pixel (cf. figure 2.5). Un écran virtuel devrait alors être 10 fois plus grand pour espérer obtenir une bonne valeur pour chaque pixel, ce qui amènerait sa taille à $10000 \times 10000 = 100Mo$[1]. Pour un facteur 100 (les feuilles d'une forêt vues de la montagne avoisinante), la taille mémoire devient $10Go$. Ces coûts mémoires rendent l'idée irréalisable.

Plus d'astuce

Certaines approches accumulent la valeur de fragments concurrents dans un même pixel de l'écran [HA90]. Celui-ci intègre successivement ces nouveaux fragments à la couleur du pixel. Ces techniques résolvent en partie les problèmes de coût exposés plus haut puisque l'écran garde la

[1] pour un octet par pixel, tel que c'est souvent le cas pour des images noir et blanc.

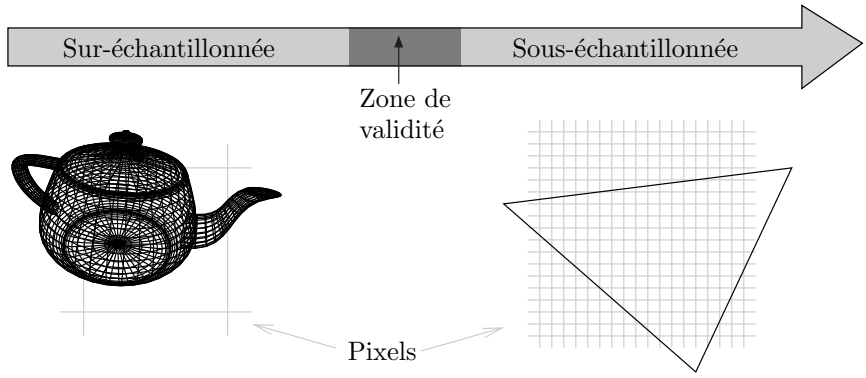


FIG. 2.6 – La projection d'un objet sur l'écran peut être *sous-échantillonnée* ou *sous-échantillonnée*. Les méthodes d'anti-aliasing permettent d'agrandir la zone de validité : même si un triangle est un peu trop petit, son affichage sera de bonne qualité. Malheureusement, cet agrandissement est très limité est permet au maximum de gagner un facteur d'échelle de l'ordre de la dizaine.

même taille. Néanmoins, il est délicat de savoir quelle importance donner à tel ou tel fragment, puisque l'information sur la couverture du pixel est perdue (les fragments sont comme des points, il n'ont pas de taille).

Il est possible de garder une bonne approximation de l'information de couverture en générant des fragments dédiés à une sous-partie d'un pixel. Notamment, le A-buffer [Car84] propose de découper un pixel en sous-parties (le plus souvent $4 \times 4 = 16$). De cette façon, il devient possible de mieux quantifier l'influence de chaque fragment et de calculer une couleur finale relativement correcte.

L'implémentation de ce genre de techniques ne supporte le temps-réel que lorsqu'elle est directement coulée dans le silicium. De fait, les cartes graphiques récentes proposent des méthodes efficaces améliorant très sensiblement la qualité du rendu. Malheureusement, le facteur d'échelle gagné est de l'ordre de la dizaine, bien loin des buts que nous nous sommes fixés dans le chapitre 1. Dans le cas contraire, la mémoire et les coûts de calcul explosent.

Les méthodes d'*anti-aliasing* ont évidemment leur utilité et participent notamment à combattre d'autre phénomènes d'aliasing qui ne sont pas liées à la multi-échelle (tel que le crénelage). Cependant, elle ne sont pas suffisantes, à elles-seules, pour afficher des scènes complexes observables à différentes échelles.

Sous-échantillonnage

Une autre forme d'*aliasing* sévit tous les jours sur nos écrans, mais elle n'a ordinairement pas le privilège d'être étiquetée comme produit de l'*aliasing* : il s'agit du sous-échantillonnage (c'est-à-dire l'inverse du sur-échantillonnage). Par exemple, prenez un lapin rouge. Vu à une certaine distance, le lapin rouge ressemble à un lapin rouge, ce qui est très souhaitable. Mais rapprochons nous maintenant, si bien que tout l'écran devient uniformément rouge. On ne voit plus un lapin, ni ses poils que l'on aurait dû voir pourtant. On voit *autre* chose : du rouge. Donc, si l'on se réfère à l'étymologie du mot, c'est bien d'*aliasing* dont il s'agit.

Sans doute l'appellation paraît inappropriée car, finalement, on s'attend bien à voir un écran rouge, comment en serait-il autrement ? Si les poils n'ont pas été modélisés, l'ordinateur ne pourra pas les inventer. C'est donc moins *surprenant* de voir un écran rouge que le désagréable clignotement de l'*aliasing* dû au sur-échantillonnage. Mais mis à part cette considération émotionnelle, il s'agit bien, scientifiquement, des deux faces du même problème.

Il n'est pas vraiment de remède miracle au sous-échantillonnage. Les modèles géométriques courants sont implicitement créés pour être observés à une certaine échelle. Si l'on observe de

plus près, on peut toujours filtrer et simplifier l'information. Mais si l'on observe de plus loin, on touche aux limites du modèle et incidemment à la "stupidité" d'un ordinateur : celui-ci est bien incapable d'automatiquement enrichir une forme visible sans qu'on lui ai dit comment faire. Un modèle pré-échantillonné n'est donc valable que dans une certaine *zone d'observation* en dehors de laquelle il devient caduque (cf. figure 2.6).

2.1.3 Techniques diverses

Avant de conclure sur l'affichage multi-échelle au sein des cartes graphiques dans la sous-section suivante, nous expliquons ici deux approches communément utilisées pour réduire les problèmes d'*aliasing*. Elles ne sont pourtant pas étiquetées comme méthodes d'*anti-aliasing*. Il s'agit du *MIP-mapping* (discrétisé) et de la programmabilité (procédurale).

MIP-map

Les textures sont des tableaux[2] échantillonnés régulièrement. Elles représentent la plupart du temps certaines propriétés de la fonction d'éclairage tel que le spectre de diffusion (autrement dit, les couleurs) ou une direction privilégiée (les normales). Une texture est appelée par un fragment qui, nous l'avons vu plus haut (cf. figure 2.2), peut représenter une zone plus ou moins vaste. Par exemple, un fragment peut représenter à lui seul toute la texture. Il est donc nécessaire d'intégrer l'information afin d'obtenir une valeur du fragment correcte, c'est-à-dire représentative de toute la portion de texture qu'il recouvre.

De nombreuse méthodes opèrent un pré-traitement sur ces tableaux afin de construire une hiérarchie de textures plus ou moins fine. Ces opérations sont connus sous le nom de *MIP-mapping*[Wil83]. L'acronyme MIP vient de l'expression latine "multim in parvo" qui veut dire "beaucoup de chose dans une petite place". Pour les tableaux à une et deux dimensions, le MIP-mapping réduit efficacement les parasites dus à un mauvais échantillonnage. Les récentes fonctions de filtrage anisotrope ajoutent encore à la qualité du rendu.

Les méthodes de MIP-mapping ressemble aux méthodes d'*anti-aliasing* dans la mesure où elles s'appliquent à des données *régulièrement* échantillonnées. Compte tenu du fait qu'il s'applique à des données statiques (les textures ne sont pas modifiées à chaque affichage), on peut décrire le MIP-mapping comme une sorte de pré-*anti-aliasing* local. Le terme "local" vient du fait que ces méthodes ne s'appliquent pas à tout l'écran, mais à chacune des textures des différents objets constituant la scène.

Programmabilité

Les fragments issus d'une primitive graphique sont autant d'approximations de la fonction d'éclairage du matériau. Performance oblige, il n'est pas question de s'amuser à décrire la micro-structure d'une surface et à l'intégrer lors de l'affichage. Un éventail de fonctions d'éclairage permet de calculer efficacement quelques comportements simplistes (éclairage diffus, spéculaire). Cette modélisation procédurale de l'éclairage a fait ses preuves. Mais elle est insuffisante pour représenter des éclairages complexes et réalistes.

La sophistication de ces fonctions d'éclairage a récemment fait un bond en avant avec les *fragment-programs* [WND99]. Ces micro-programmes permettent notamment d'augmenter l'éventail des fonctions d'éclairage. Ils le font de manière *procédurale* alors que les textures que nous venons d'évoquer s'attaquent au même problème (enrichir la fonction d'éclairage) mais de manière *discrétisée*.

Plus généralement, la programmabilité des cartes graphiques récentes offre une souplesse qui peut être utilisée pour mieux gérer la multi-échelle. Par exemple, [DVS03] utilise les *vertex-programs* pour déterminer quelle hiérarchie de point doit être utilisée pour l'affichage courant. [BS04] propose une génération de surfaces de subdivision avec un *fragment-program*. Même si

[2] Généralement, les textures sont des tableaux à une ou deux dimensions. Plus de dimensions sont permises, mais les coûts mémoire associés limitent leur utilisation (la mémoire est chère...).

les traitements réalisés en *hardware* restent relativement simples, il est clair que ces approches procédurales sont très prometteuses.

2.1.4 Limitations intrinsèques des cartes graphiques

Le matériel graphique fait parfois la pluie et le beau temps en synthèse d'images. En divisant par cent le coût de certaines fonctions, il peut éventuellement rendre complètement obsolètes certains algorithmes ou au contraire les remettre aux goûts du jour. Indéniablement, la recherche en infographie dépend, dans une certaine mesure, des possibilités offertes par les cartes graphiques (de même qu'elle dépend de toute l'architecture d'un ordinateur). Mais il est important de garder la tête froide : voyons pourquoi on ne peut pas tout attendre d'un matériel spécialisé.

Simple et efficace

Plus l'information migre le long du processus de création, moins celle-ci est sophistiquée. Par exemple, en fin de course, l'information devient une vaste énumération de pixels mis à jour plusieurs dizaines de fois par seconde. En début de cycle, l'idée d'un modèle qui germe dans notre cerveau est une information de nature bien plus sophistiquée et concise qu'une simple énumération. La création revient à trouver un moyen de transformer la représentation du modèle tout en la gardant fidèle à l'idée de départ.

En fin de parcours, la carte graphique s'occupe du traitement final de l'information. Compte tenu de la simplicité de la représentation finale, les traitements sont très basiques (projection, rasterisation). En revanche, ils sont redoutablement efficaces et traitent une très grande quantité d'informations. Ce mélange d'efficacité et de simplicité est la nature profonde des cartes graphiques.

Face à la multi-échelle

Dans cette section, nous avons vu que les cartes graphiques étaient d'un grand secours face au problème posé par la multi-échelle. Mais, de par leur nature "simple et efficace", elles ne prendront jamais complètement en charge un problème aussi complexe que celui du rendu multi-échelle de scène complexe animée.

Bien qu'elles soient parfois utilisées pour des tâches qui n'ont rien de visuel [BFGS03], elles ne peuvent offrir autant de souplesse qu'un processeur universel. Certains pensent que ces cartes sont les prémices d'un nouveau type de processeur universel basé sur un flux d'information intense et redirectionnel. Ces considérations sont alléchantes et laissent entrevoir une toute autre utilisation des cartes graphiques (mais peut-être ne seront-elles alors plus appelées comme ça).

Plutôt que de spéculer sur l'avenir des GPU, revenons en à la multi-échelle. J'affirme que la complexité algorithmique nécessaire au rendu multi-échelle de scènes complexes et animées dépasse de loin les possibilité des cartes graphiques actuelles. Évidemment, plus il sera possible de migrer des fonctionnalités au sein du GPU, plus efficace sera l'affichage. Néanmoins, il restera toujours un noyau algorithmique qui ne pourra être réalisé que sur un support suffisamment complexe et souple tel que le processeur universel. L'une des contributions de cette thèse est l'identification de ce noyau ainsi qu'une tentative d'implémentation (chapitre 3).

2.2 Méthode de simplification générique

Plus en amont dans le processus de création, pendant ou juste après la modélisation, une forme visible peut être décrite par de nombreuses représentations : maillage, volume, surfels... Généralement, pour chacune de ces représentations, plusieurs méthodes de *simplification* permettent un affichage multi-échelle.

Avant tout, nous commencerons par définir ce que nous entendons par *simplification*. Nous décrirons ensuite rapidement les différentes méthodes de simplification générique. Ensuite, nous décrirons les limites de ces méthodes dans le cadre de la modélisation et du rendu de scènes animées avec de grandes variations d'échelle.

2.2.1 Définition

Les méthodes de simplification calculent *automatiquement* des versions plus grossière d'un modèle décrit à son *niveau le plus fin*.

Une connaissance du niveau le plus fin

Nous caractérisons les méthodes de *simplification* par le fait qu'elles prennent en donnée d'entrée un modèle supposé être au *niveau le plus fin* pour calculer des versions plus grossière (moins précise). La quasi-totalité des modeleurs classiques (Maya ou 3DSMAX par exemple) produisent justement des modèles décrits à leur niveau le plus fin. Par conséquent, les méthodes de simplification sont très majoritaires dans l'ensemble des travaux proposant une approche multi-échelle.

Automatisme

Les méthodes de simplification se veulent *automatiques* : l'utilisateur fournit simplement le modèle en entrée et récupère le résultat. Ce but est louable car les méthodes de simplification interviennent une fois la modélisation terminée[3].

2.2.2 Classification

De nombreuses représentations permettent de décrire des formes visibles. Les méthodes de simplification sont ici classées en fonction de la représentation à laquelle elles s'appliquent. Cette classification restera très générale : mon propos est plutôt de monter l'inadéquation de l'approche d'un point de vue global.

Représentations maillées

Les maillages sont la représentation la plus utilisée et détiennent haut la main le record du nombre d'algorithmes de simplification [Hop97, RB93]... Je vous renvoie à l'excellent [LRC+02] qui recense la quasi-totalité des méthodes de simplification de maillages.

De notre point de vue, les principales caractéristiques de ces méthodes sont :

précision uniforme/variable : certaines méthodes produisent une hiérarchie de maillages dont chaque élément a une précision uniforme. D'autres méthodes plus complexes permettent de faire varier la précision sur un même objet.

transition continue/discrète : le passage d'une précision à l'autre peut être discret ou bien être interpolé continûment.

Seuls les algorithmes à précision variable peuvent supporter un affichage avec de grandes variations d'échelle. La gène visuelle occasionnée par des transitions discrètes [Red97] limite leur emploi intensif.

Représentations volumiques

Les représentations volumiques [KK89, Ney98] décrivent une forme visible par un échantillonnage spatial de la fonction d'éclairage. Des versions plus efficaces ont été proposées [LL94], parfois supportées par le matériel graphique [MN98].

En fonction du type d'information stocké dans un texel, les représentations volumiques permettent des simplifications plus ou moins importante. Par exemple, le codage de l'opacité d'un texel permet d'engendrer des versions grossières du modèle de bonne qualité [HHK+95]. Les représentations volumiques supportent donc mieux les simplifications extrêmes. En revanche, le passage à la 3D entraîne souvent des coûts mémoires nettement supérieurs aux représentations maillées.

[3] Il existe toutefois, dans les modeleurs du commerce, des méthodes non automatisées n'offrant qu'une aide à la création de niveaux de détail discret. Ces travaux seront discutés dans le chapitre suivant.

Représentations alternatives

Le terme *représentations alternatives* [Ney01] caractérise des méthodes de représentations originales, par exemple :

surfels : de récent travaux [RL00, DVS03] représentent des objets par de petits éléments de surface et en affichent plus ou moins selon la précision requise ;

imposteurs : l'approximation de forme $3D$ par des images simplifie énormément la complexité d'une scène [DDSD03, CBL99, MS95, LS97].

lightfield : [LH96, GGSC96] propose un codage global de la fonction d'éclairage. Conceptuellement proche des représentations volumiques, ces méthodes nécessitent énormément de mémoire pour une excellente qualité de rendu.

2.2.3 Limitation

Dans le cadre de scènes animées observées avec de grandes variations d'échelle, la simplification souffre principalement de trois limitations.

Simplification extrême

La simplification extrême de formes lest un problème très délicat. Dans la grande majorité des cas, la qualité de l'approximation se dégrade fatalement à partir d'un certain degré de simplification. Imaginons par exemple un livre dont une face (la couverture) est rouge et l'autre face est verte. Très peu d'algorithmes, lors d'une simplification extrême, réduiront le livre à une sorte de point dont la couleur variera selon le point de vue.

Certaines représentations s'en sortent mieux que d'autres. Notamment, les représentations volumiques ou les *light fields* prenant en compte la transparence sont très robustes. En revanche, les coûts mémoires nécessaire pour décrire de grandes scènes deviennent vite insupportables. En fait, cette robustesse est réalisée au détriment de tout codage ingénieux du modèle. En effet, l'intelligence d'une représentation repose sur la connaissance du "type" de la forme (lisse, chaotique, transparent...). Mais ce type à la fâcheuse tendance de changer complètement lors de passages à l'échelle (une prairie vue de très loin est une surface lisse).

La simplification automatique révèle ici ses limites : sa méconnaissance sémantique des structures qu'elle manipule. Non seulement, comme nous venons de le voir, la nature de l'objet lui échappe, mais elle ignore aussi l'importance que nous, humains, accordons à cet objet. Par exemple, il nous est bien égal qu'un algorithme approxime brutalement un caillou au bord de la route. En revanche, il devra prêter une grande attention à la simplification des yeux humains. Nous remarquerions en effet la moindre dissymétrie, même de loin. En fait, la simplification de forme visible peut supporter une part d'automatisme mais elle ne pourra *jamais* fonctionner pour un large type d'objets et pour des grandes variations d'échelle sans une intervention humaine. Nous seuls sommes capables de décréter ce qui est important à nos yeux.

Coût mémoire

La connaissance explicite d'une scène à son niveau le plus fin limite drastiquement tout codage ingénieux. Nul besoin d'aller à des échelles astronomiques, les moindres paysages un peu trop volumineux sont très vite impossibles à stocker. En effet, les données d'entrée (le modèle à sa précision maximum) se chiffrent rapidement en dizaines ou centaines de gigaoctets.

Certaines méthodes de pagination de donnée transforment le disque dur en une extension plus lente mais plus volumineuse de la mémoire vive [LP02]. Les structures de données utilisées sont de plus en plus sophistiquées et accélèrent l'accès aux données. Mais cela ne fait que repousser le problème. D'ailleurs, on peut toujours décréter que l'on veut traiter des scènes de taille infinie (comme les fractales).

Le constat est clair : en utilisant une description au niveau le plus fin, on limite la taille des scènes visualisables. Cette limite est floue et dépend notamment de la représentation choisie. Il

est donc difficile de quantifier à partir de quand ce handicap devient trop pesant. Mais dans une optique de rendu avec de grandes variations d'échelle, la simple existence de cette limite est très rebutante.

Animation du modèle

Les algorithmes de simplification temps-réel utilisent systématiquement un précalcul dont le but est de permettre un rendu efficace. Ces précalculs représentent toujours plus ou moins une représentation du modèle à différentes résolutions. Si la forme du modèle change (par exemple, lors d'une animation), il doit donc remettre à jour toute l'information plus grossière qui dépend de la zone qui s'est déformée. Le coût de cette mise à jour est souvent incompatible avec un affichage temps-réel.

En pratique, l'utilisation d'algorithmes de simplification limite beaucoup les possibilités d'animation. Dans certaines scènes très statiques, cela ne pose pas vraiment de problème (comme les villes). Mais c'est incompatible avec notre objectif : le rendu de scènes *animées* avec de grandes variations d'échelle (dans le film «Dark city», les bâtiments changent de place et de forme toutes les nuits...)

2.2.4 Bilan

Les limitations décrites ici motivent une représentation par complexification, décrite dans le chapitre suivant.

Pour une utilisation locale

Les contraintes sur la taille de la scène, la mauvaise qualité des simplifications extrêmes et l'animation sont autant de limites aux méthodes de simplification générique. Cette approche est inapte à assurer le bon déroulement de la modélisation et du rendu de scènes animées avec de grandes variations d'échelle. Cela dépend de ce que l'on entend par "grandes variations d'échelle". Disons qu'à partir de certaines amplitudes dans les variations, la simplification échoue inexorablement.

Je précise ici que ce constat ne met pas ces méthodes hors course, loin de là. Cela les assigne simplement à une marge de manœuvre plus locale. La simplification n'est pas la bonne approche pour résoudre le problème. En revanche, il me paraît clair que la bonne approche, quelle qu'elle soit, devra utiliser la simplification pour arriver à ses fins.

Vers une représentation plus sophistiquée

Pour de nombreuses raisons, les algorithmes de simplification ne sont pas aptes à faire un rendu multi-échelle animé de scène complexe. Mais pourquoi ne pas retourner la critique ? Et si c'était la *représentation* utilisée pour décrire des scènes complexes qui n'était pas à la hauteur des algorithmes de simplification ?

Il est remarquable que les méthodes de simplification prennent toutes des données d'entrée de nature très *simplistes*. Dans la grande majorité des cas, les représentations utilisées sont des énumérations de points, éventuellement agrémentées d'une information de voisinage (graphe d'adjacence, grille). La plupart du temps, aucune information *sémantique* n'existe et on ne sait pas si on a affaire à un lapin ou à un tractopelle.

Pourquoi ne pas adopter une représentation plus *complexe* ?

Chapitre 3

Complexification

Comme nous l'avons vu à la fin du chapitre précédent, la meilleure manière d'aborder la modélisation est l'affichage multi-échelle de scènes animées et d'introduire la multi-échelle le plus tôt possible dans le processus de création. C'est la conception même du modèle qui est multi-échelle (cf. figure 3.1).

Afin d'éviter une description explicite au niveau le plus fin, le modèle est décrit par une représentation complexe partant de sa forme la plus grossière et ajoutant de la précision : c'est la *complexification*.

La structure de ce chapitre est particulière. En effet, l'une des contributions de cette thèse est une analyse des besoins qu'engendre la complexification. Ce chapitre contient donc des parties d'état de l'art *et* des contributions. Voici précisément comment se déroule ce chapitre :

Modélisation procédurale multi-échelle : une nouvelle définition du terme *modélisation procédurale* est proposée. Intégrée dans le contexte de la modélisation multi-échelle, elle devient *la complexification*.

Fractales : cette section est un état de l'art des premières modélisations par complexification (cf. [EMP+98],page 436) : les fractales. Elle introduit les recherches de Przemyslaw Prusinkiewicz et Christophe Godin sur «le monde platonique».

Cas particuliers : de nombreux travaux existants ont, de près ou de loin, utilisé une approche par *complexification*. Nous les étudierons et conclurons par une contribution : l'identification des besoins communs à tous ces travaux.

3.1 Modélisation procédurale multi-échelle

Dans cette section, nous essairons de mieux comprendre ce qu'est la *modélisation procédurale*, si présente dans ce mémoire. Après une (tentative de) définition basée sur celle donnée dans [EMP+98], son apport dans le cadre d'un rendu multi-échelle sera discuté.

3.1.1 Programme *vs.* structure de données

Reprise de nombreuses fois en synthèse d'image, la définition de "modélisation procédurale" est plutôt obscure. Néanmoins, il semble bien que l'élément essentiel soit la distinction entre *programme* et *données*.

Définition de référence

Dans le livre «Texturing & Modeling, A Procedural Approach» [EMP+98], David Ebert donne une définition de *procédural* : «the adjective *procedural* is used in Computer Sciences to distinguish entities that are described by program rather than by data structures». Cette définition oppose

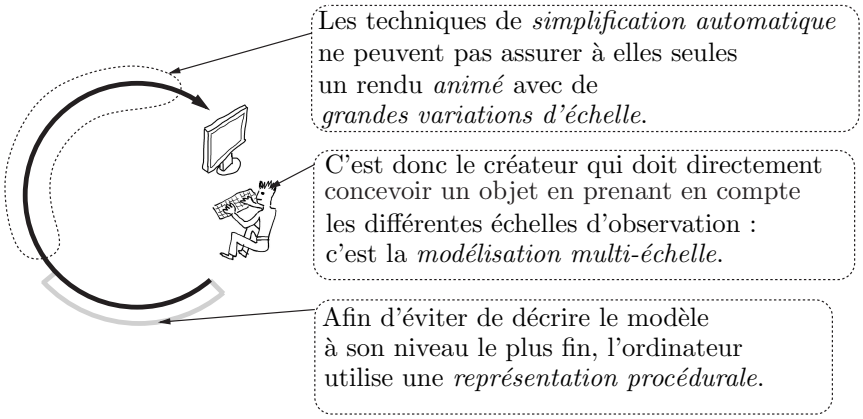


FIG. 3.1 – Ce chapitre propose une étude des différentes représentations procédurales multi-échelles existantes et de leurs applications.

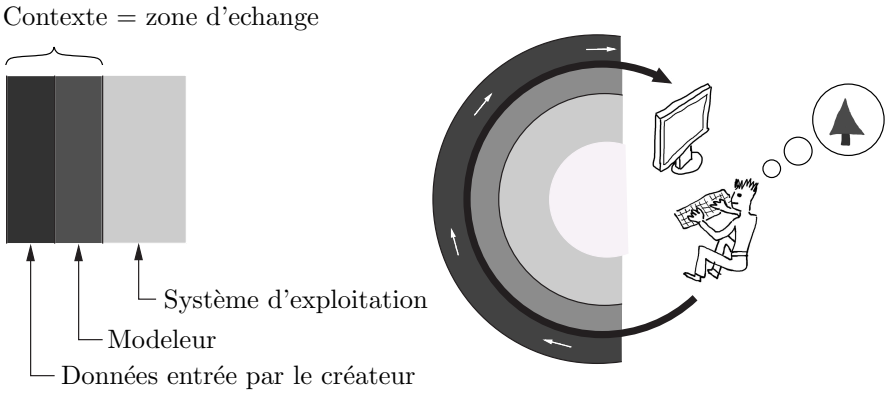


FIG. 3.2 – Durant le processus de création, un *programme* traite les idées du créateur (les *données d'entrée*) et les transforme en pixels. On peut considérer que tout le système d'exploitation fait partie du programme, mais cela n'a que peu d'influence : on se focalise sur, le contexte, c'est à dire la *frontière* entre l'outil et les données entrées par l'utilisateur.

deux types d'informations : actives (le code) et passives (les données). La délimitation entre ces deux types d'informations est malheureusement très floue.

L'auteur de la précédente définition est d'ailleurs explicite quant à cette imprécision. Il prend notamment l'exemple d'une fonction faisant partie du programme. L'information la constituant est considérée comme étant de type *actif*. Mais ne peut-on pas considérer que les *paramètres* de cette fonction sont en fait une information de type *passif* ? Il semble qu'ici encore, la vérité soit une question d'échelle : elle dépend de l'observation... Avant de m'essayer à mon tour à une définition, j'aimerais m'attarder un peu sur le contexte particulier dans lequel on se place : celui de la modélisation. Pour cela, il nous faut donc répondre aux questions suivantes : dans le cadre de la modélisation en synthèse d'images, qu'est-ce que le *programme* ? Où sont les *données* ?

Le programme

Reprenons ici le processus de création : le créateur envoie des informations dans l'ordinateur qui sont transformées en pixels sur l'écran. Le *programme*, en synthèse d'images, est toute l'information qui décrit ces transformations. Très concrètement, c'est le code des programmes de modélisation et d'affichage ou plus généralement, toute information décrivant ces *capacités de traitement de*

l'information (cf. figure 3.2).

Les données d'entrée

les *données d'entrée* sont, comme leur nom l'indique, données au programme par le créateur dans la phase de modélisation. Elles sont l'information *imprévisible* de la modélisation. Elles sont donc entièrement représentées par les actions physiques qu'opère le créateur devant son poste : bouton X, clic droit... On pourrait même remonter encore et déclarer qu'elles sont représentées dans la tête du créateur, mais elle est difficilement accessible ici... La représentation qui nous intéresse est celle du qui est retenue par le programme au moyen d'un *langage descriptif*.

Le langage descriptif est la grammaire utilisée pour représenter les données d'entrée dans le programme. Le choix de ce langage est d'une importance capitale. Par exemple, un modeleur peut mémoriser les opérations effectuées grâce à un maillage. Dans ce cas, chaque opération est oubliée et seul le résultat (le maillage) est stocké. Dans d'autres cas, ce sont les opérations qui sont stockées et la forme ne prend de sens que comme la succession d'application de ces opérations à une forme initiale. Il devient alors possible de remonter l'historique de la création et éventuellement faire des modifications "dans le passé" (comme le permet par exemple Maya avec son Dependency Graph [Ali03]).

3.1.2 Modélisation procédurale : définition

On peut donc opposer deux types d'informations : les données (le modèle 3D) et le programme (le modeleur). Une fois ce contexte situé, nous voila maintenant mieux préparés pour une nième définition de la *modélisation procédurale*.

Complexité du langage descriptif

Dans l'exemple précédent, on distingue différentes potentialités des langages descriptifs : un maillage permet moins de chose que la liste de toute les opérations effectuées par le créateur. Ceci n'est finalement que justice car mémoriser toutes les opérations effectuées par le créateur nécessite un langage *complexe* : il faut pouvoir y stocker tous les types d'opérations, leur domaine d'application, la date...

Définition : *la complexité d'un langage descriptif est la taille de sa sémantique relativement à celle du programme qui l'utilise.*

L'intelligence est assimilable à la prise d'initiative, l'indépendance. Par exemple, comment dessiner une forme représentée par un maillage ? Il faut extraire et afficher tous les triangles un par un et les envoyer à la carte graphique. Un autre langage descriptif pourrait faire preuve de plus de finesse et intégrer dans sa représentation la façon dont elle s'affiche. Le programme principal n'aurait plus qu'à demander à une forme décrite avec ce langage de s'afficher, sans s'occuper des détails de l'opération. La complexité d'un langage descriptif est donc similaire à la concision de l'échange entre programme et donnée. Cette concision cache des processus complexes réalisés de façon relativement autonome par les données d'entrée.

Modélisation procédurale

Définition : *une modélisation est dite procédurale lorsque la complexité du langage descriptif est grande.*

Remarquons d'emblée que c'est une définition nuancée : plus la complexité du langage descriptif est grande, plus la modélisation est procédurale. L'idée intuitive, derrière cette définition, est de donner plus de *pouvoir* aux données d'entrée, plus d'*expressivité*. Ceci est à mon avis essentiel puisque ces données d'entrée sont les représentantes de notre intelligence au sein du programme !

Je suis conscient du manque de formalisme que l'on peut reprocher à cette définition, notamment dans l'utilisation du mot "sémantique" utilisé dans la définition de la compléxité. J'espère néanmoins avoir donner l'idée principale : rendre le modèle plus intelligent. Cette intelligence est absolument nécessaire dans le cadre de la multi-échelle où un modèle n'est plus une simple "statue",

mais une forme en perpétuel mouvement, transformée d'une part par des méthodes d'animation, mais aussi par les mouvements de l'observateur.

3.1.3 Modélisation par complexification

Quels liens unissent la multi-échelle et la modélisation procédurale ? Est-ce que la modélisation procédurale n'est qu'une approche parmi d'autres pour aborder le problème des scènes observées avec de grandes variations d'échelle ? Nous allons voir ici que ces deux concepts sont en fait inséparables. La problématique de la multi-échelle, devant l'échec de la simplification, mène *naturellement* à la modélisation procédurale. La méthode résultante, la modélisation procédurale multi-échelle, est nommée *complexification*.

Plus de souplesse

Nous avons vu dans le chapitre 2 qu'il était impossible d'espérer trouver des solutions aux problèmes posés par *une* méthode de simplification. Mais rien n'empêche de compliquer un peu le langage descriptif d'un modèle et de permettre que celui-ci englobe plusieurs sous-modèles. Ainsi, au lieu d'un seul modèle simplifié sur une grande plage d'échelle, plusieurs sous-modèles sont simplifié localement (cf. figure 3.3). On peut alors choisir des représentations et les algorithmes de simplification les plus adaptés à chaque sous-modèle. Notons que cette approche permet d'afficher plusieurs fois le même objet, puisque celui-ci n'est plus noyé dans une immense simplification : cette technique couramment utilisée se nomme *instanciation* [EMP+98]. Du coup, les problèmes de coût mémoire et de simplification extrême sont moins limitants.

Mais comment le créateur fait pour permettre au modèle de passer d'un sous-modèle à l'autre quand l'observateur se rapproche ? Supposons que l'on dispose d'un modèle de forêt observable de $50m$ à $1km$ ainsi que d'un modèle d'arbre observable de $1m$ à $50m$. Sans même considérer des critères perceptuels complexes, le problème consistant à enrichir un modèle n'a rien d'évident. En effet, le modèle de la forêt peut très bien faire abstraction de l'individu et n'avoir alors que peu d'informations sur la manière dont un arbre va la remplacer. La solution consiste à décrire des *fonctions de transition* permettant de passer d'un modèle à l'autre et d'assurer le lien entre deux hiérarchies distinctes (cf. figure 3.3).

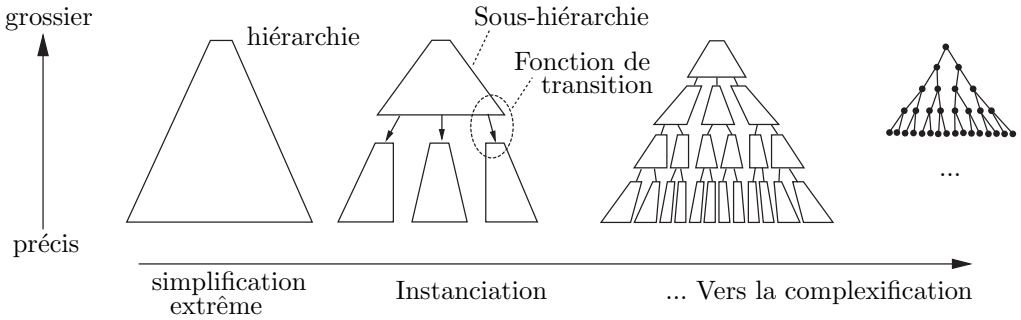


FIG. 3.3 – D'un modèle brutalement simplifié vers une représentation par complexification, la démarche est très naturelle. Le découpage et la réduction des plages de changement d'échelle permettent d'utiliser *différentes représentations adaptées* et de réduire les problèmes inhérents à la simplification.

Fonctions de transition

Ces *fonctions de transition* ont pour but de briser la lourde hiérarchie des niveaux de détail précalculés. Ainsi, elles doivent être très flexibles et ne pas reposer sur de grandes quantités d'in-

formations. Pour le passage des forêts aux arbres, on peut imaginer une fonction qui va repérer, dans le modèle de la forêt, la place et l'espèce des arbres qui apparaissent.

Cette information peut être codée au sein de la forêt, mais de façon très concise. En tout cas, il est hors de question de décrire, dans le modèle de la forêt, chaque arbre entièrement et dans toute sa précision. On retomberait alors dans une représentation au coût mémoire exorbitant en "recollant" les sous-hiérarchies de chaque sous-modèle entre elles. Par contre, il est envisageable de stocker la place de chaque arbre de la forêt ainsi que d'éventuels paramètres (sa taille, son espèce, le nombre de branches...). Les fonctions de transitions consistent en fait en un *paramétrage* des modèles précis par le modèle grossier. Le modèle, lors d'un passage à un niveau plus précis, ne représente plus la forme brute, mais les paramètres qui vont permettre de l'afficher. On est bien en présence de *modélisation procédurale*.

Une description complexe mais dynamique

Le créateur, à cause des contraintes inhérentes à la simplification, est amené à représenter un modèle par plusieurs sous-modèles reliés entre eux par des *fonctions de transition*. D'une part, il ne manquera pas de remarquer que celles-ci sont difficiles à réaliser. En effet, il devra sans doute sophistiquer un peu le langage descriptif du modèle : celui-ci doit maintenant représenter non seulement des formes visuelles, mais aussi les fonctions permettant de les paramétrer. D'autre part, le créateur regrettera d'avoir été contraint à quitter les méthodes de simplification. En effet, celles-ci permettent de passer automatiquement d'une précision à l'autre. Désormais, ce passage est manuel et le bon déroulement des *fonctions de transition* est à sa seule et unique charge.

Néanmoins, notre créateur remarquera aussi que l'utilisation de *fonctions de transition* a décidément plusieurs aspects très positifs. Tout d'abord, elles répondent à leurs objectifs initiaux : elles réduisent les coûts mémoire et permettent des simplifications locales de bonne qualité. De plus, le paramétrage des sous-modèles, nécessaire aux fonctions de transition, peut être utilisé à d'autres fins. Notamment, ces nouveaux degrés de liberté permettent d'*animer* le modèle. Par exemple, revenons aux forêts et supposons que le modèle d'arbre dont on dispose peut changer de taille. Cette caractéristique, initialement prévue pour assurer une population d'arbres de taille variée, peut aussi servir à faire pousser les arbres au cours du temps.

Vers la complexification

La possibilité de faire pousser des arbres est certes une nouveauté en comparaison avec les méthodes de simplification pures qui permettent très difficilement l'animation. En revanche, cela est insuffisant pour rendre le mouvement des arbres sous le vent, par exemple. Du coup, afin de relaxer encore le modèle et d'y ajouter plus de degrés de liberté, notre créateur téméraire ajoutera des fonctions de transitions un peu partout. Il réduira en conséquence les phases de simplification et augmentera les possibilités d'animation (cf. figure 3.3). Ce faisant, il modélisera d'une façon bien particulière. Il commencera par la forêt vue de très loin et rajoutera des fonctions de transitions enrichissant progressivement la forme pour la rendre à chaque fois plus précise. Il pratiquera la modélisation par *complexification*.

Définition : *La modélisation par complexification est une description d'un modèle par un ensemble de fonctions décrites dans un langage descriptif procédural (i.e. complexe). Ces fonctions ajoutent localement de la précision à une forme initiale grossière. Lors d'un affichage, elles sont rejouées jusqu'à satisfaire les critères d'observation (précision, visibilité...).*

Au premier abord, complexification semble prometteuse. Mais il y a bien sûr beaucoup de problèmes à résoudre. Tout d'abord, comme nous l'avons déjà vu en 2.2.4, la simplification ne doit pas être mise de côté. En revanche, son utilisation, dans le cadre d'une application avec de grandes variations d'échelle, doit rester locale. Ensuite, la complexification nie l'automatisme ainsi que l'universalité et pousse vers des algorithmes toujours plus nombreux et spécialisés.

Historiquement, les premiers modèles par complexification sont les objets fractales qui sont étudiés dans la prochaine section. Plus récemment, quelques travaux assez récents utilisent la

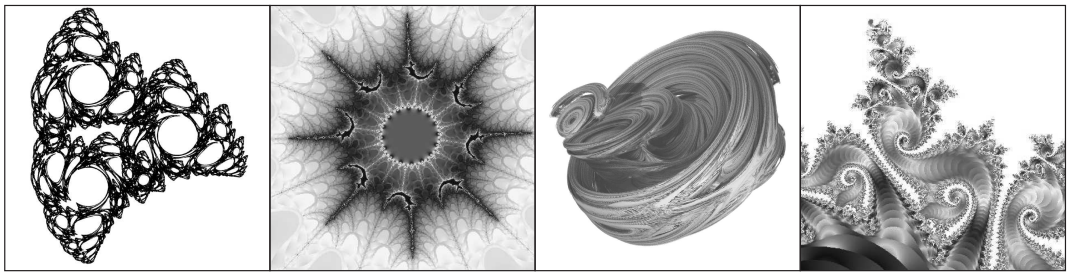


FIG. 3.4 – Les fractales ont décidément quelque chose d'intrigant. Lorsqu'on s'en approche ou s'en éloigne, l'animation engendrée est d'un effet très curieux. C'est un peu comme si notre cerveau pressentait la simplicité de la définition mais se laisser pourtant submergé par une apparente complexité.

complexification afin de modéliser des phénomènes précis, dans des conditions d'observations particulières. Ils seront étudiés dans la section 3.3.

3.2 Fractales

Les fractales sont très polémiques en synthèse d'images : elles ont suscité beaucoup de critiques et leur utilité a été sévèrement remise en cause. De notre point de vue, elles sont pourtant extrêmement intéressantes : ce sont les premiers objets modélisés par *complexification*.

Nous commencerons cette section par une vue d'ensemble des fractales. Nous décrirons ensuite les méthodes utilisées pour accélérer le rendu d'objet fractal. Nos terminerons par la descriptions du "Platonic World".

3.2.1 Vue d'ensemble

Benoît Mandelbrot a découvert les objets fractales : des «objets infiniment complexes et auto-similaires à différents niveaux». Cette définition, proposée par Mandelbrot lui-même, ne le satisfait pas entièrement [Man75]. D'autres définitions existent, mais finalement, comme tant d'autres termes, la nature précise du mot "fractale" nous a un peu échappé. Entre considération esthétique et rigueur mathématique, que sont réellement ces objets ? Quelle est leur utilité ?

Un luxe

La visualisation fractale a eu son heure de gloire et la voilà maintenant réduite à remplir les fonds d'écran. À un curieux engouement esthétique a succédé un réel désintéressement scientifique. Les fractales ont néanmoins trouvé des applications en statistique, en compression et en traitement du signal [HRSV01].

Il n'en reste pas moins l'intérêt esthétique qu'elles suscitent (cf. figure 3.4). Les images générées ont quelque chose d'un peu mystique au point que nombreux sont ceux qui déclaraient que l'univers était fractal. Dernièrement, certains logiciels ont permis de zoomer sur des fractales en temps-réel [xao, eas]. Ces logiciels, très spécialisés, permettent de très grandes variations d'échelle dont la limitation est la précision numérique des flottants. Naviguer dans une fractale est une expérience très particulière : la complexité d'un univers engendré par une simple équation peut être étonnante et les "paysages" auto-similaires (et pourtant subtilement différents) sont très attrayants, presque hypnotiques.

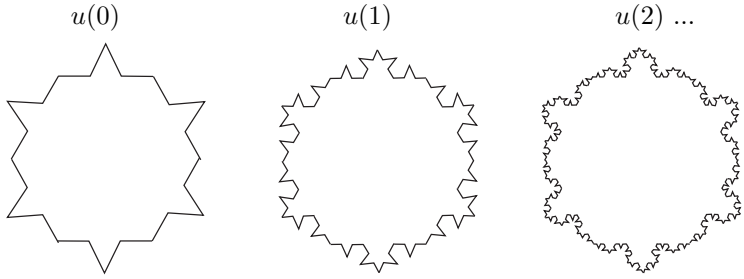


FIG. 3.5 – La construction de certaines fractales peut être vue comme un ajout successif de précision.

Un bon rapport qualité prix

Chacun définit le mot fractale de son propre point de vue. Un mathématicien (mais pas tous) dira que c'est un objet dont la dimension de Hausdorff est supérieure à sa dimension topologique. Un artiste dira que c'est une jolie image répétitive à différentes échelles... Définir de tels objets est délicat. Pourtant, il existe une caractéristique commune à toutes les fractales : la complexité de l'information générée. Lorsqu'on la visualise, la complexité d'une fractale *paraît* être bien plus grande que la complexité de sa définition. En d'autres termes, une fractale est une fonction mathématique simple dont l'observation (i.e. la visualisation) est disproportionnellement complexe. Un bon rapport $\frac{qualité}{prix}$ en quelque sorte...

Du point de vue de la multi-échelle, cette abondante complexité est très séduisante : les fractales nous offrent des univers qui nous apparaissent comme infiniment complexes. Ici, il n'est plus besoin de décrire le modèle à son niveau le plus fin pour le simplifier (cf. chapitre 2), avec tous les problèmes que cela engendre. Une simple formule décrit implicitement la scène à toutes les précisions possibles. En effet, le calcul d'une fractale est généralement itératif et ajoute à chaque pas un peu plus de précision : il suffit donc d'arrêter l'itération à une certaine valeur pour obtenir la précision correspondante.

Quelques faiblesses

Les fractales souffrent de quelques handicaps assez rebutants :

Irréel : Les fractales synthétisent des univers fictifs bien loin de la réalité. La nature offre pourtant quelques similitudes qu'il est possible d'approximer grâce aux fractales sur quelques échelles. Par exemple, un arbre, ces branches et ces sous-branches sont des objets très similaires qui peuvent être représentés par une même fonction. Mais, à la lumière des connaissances actuelles, ce genre de symétrie ne se répère jamais sur plus de 3 ou 4 niveau.

Peu paramétrable : Créer une fractale revient à définir des règles et à appuyer sur un bouton "départ". Une fois la machine lancée, «alea jacta est» : aucune influence n'est possible. Certes le coefficient $\frac{taille\ de\ l'objet}{taille\ de\ la\ description}$ est énorme. Mais c'est au prix d'une très faible paramétrabilité. Le moindre changement dans l'équation initiale peut engendrer des changements aussi importants qu'inattendus ;

Pas d'interactivité : Une fractale supporte très mal la moindre interactivité et décrit en quelque sorte des mondes "morts" qui n'offre aucune réaction à des stimuli extérieurs. C'est en fait une conséquence du faible nombre de paramètres qui la définit.

Synthétiser des mondes sur lesquels on n'a aucun contrôle est un exercice lassant. De plus, d'un point de vue scientifique, l'Homme doit plier l'informatique à ses désirs, et non la laisser générer ce que bon lui semble. En fait, la modélisation d'objet fractal et parfois très hasardeuse : il s'agit d'essayer de comprendre ce qui se passe lorsque l'ordinateur devient imprévisible. D'ailleurs, les liens très forts entre les sciences du chaos, des systèmes déterministes et des fractales ne sont

pas anodins. Pourtant, malgré leur irréalisme et leur aspect incontrôlable, les objets fractales ont soulevé de nombreux problèmes très intéressants.

3.2.2 Calculabilité

Les fractales sont les premiers univers virtuels dans lesquels on peut zoomer infiniment. Pour la première fois, ce n'est plus le modèle qui est limitant dans la navigation multi-échelle, mais l'algorithme de rendu. Voyons comment celui-ci a été modifié pour permettre un rendu efficace de fractale.

Précision itérative

Les fractales sont infiniment précises. Les méthodes utilisées pour visualiser une fractale consistent à complexifier (amplifier) l'information jusqu'à ce que la précision devienne suffisante. La représentation d'une fractale n'est donc pas la forme proprement dite, mais la façon dont elle se construit. Cette propriété est absolument prépondérante lors de son évaluation (i.e. dérivation). De ce point de vue, une fractale n'est ni plus ni moins que la limite d'une *suite convergente* (cf. figure 3.5).

Les suites utilisées sont *récursives*, ce qui permet de déduire un niveau de détail du pas de calcul antérieur. La représentation sous forme de suites récursives des formes tridimensionnelles est providentielle dans le cadre de la multi-échelle. En effet, l'observateur peut se rapprocher, c'est-à-dire rajouter des itérations, ou bien reculer, c'est-à-dire revenir aux itérations précédentes (que l'on a gardé en mémoire). De cette façon, la forme n'est jamais décrite explicitement au niveau le plus fin : seule une évaluation plus ou moins précise permet d'engendrer la forme adéquate.

Évaluation paresseuse

La représentation sous forme de suite convergente permet donc de ne calculer que la précision nécessaire à l'observation. En effet, il suffit de trouver le bon n tel que $u(n)$ satisfasse la précision requise par l'observateur (cf. figure 3.5). Mais les critères d'arrêt, dans l'espace tridimensionnel, ne sont pas aussi simples qu'un seuil entier jusqu'où la fractale est uniformément calculée. En effet, les formes visibles n'ont pas la même précision dans une même image donnée (cf. figure 1.1). De plus, certaines parties ne sont pas visibles pour diverses raisons : hors-champ, occultées...

Il est donc nécessaire de réaliser une *évaluation paresseuse* qui calcule la fonction fractale *le moins possible* de façon à ne générer que l'information utile à une observation[1]. Une *évaluation paresseuse* consiste à ne calculer l'information que là où elle est pertinente (cf. John C. Hart dans [EMP+98]). De cette façon, l'algorithme de rendu devient *output sensitive* : son coût dépend de la complexité de l'observation (entre autres : nombre de pixels sur l'écran) et non de la complexité de la fractale[2].

La grande majorité des fonctions fractales est bidimensionnelles. En conséquence, les évaluations paresseuses mises en œuvre sont généralement très simples. D'une part, la précision est uniforme (critère d'arrêt global). D'autre part, la visibilité consiste juste à ne pas engendrer ce qui est en dehors de l'écran (pas d'occultation). En trois dimensions, ces problèmes sont plus délicats. Les problèmes de précision [Red97] et de visibilité [Dur99] ont été essentiellement étudiés dans le cadre d'approche par simplification.

Cohérence temporelle

La nature très continue de la majorité des observations faites par l'Homme permet d'optimiser les calculs en utilisant la *cohérence temporelle*. En effet, la quasi-totalité des méthodes de visualisation impose des mouvements de caméra suffisamment lents pour que l'image générée à un pas de temps ressemble à celle du pas du temps précédent. Il est donc possible d'accélérer le calcul

[1] On peut parler ici d'évaluation paresseuse *conservatrice* : le moins possible, mais suffisamment.

[2] Remarquons qu'un algorithme n'est jamais complètement *output sensitive* ou *input sensitive*, mais toujours entre les deux. Le but du jeu est de faire pencher la balance du bon côté (en l'occurrence : *output sensitive*).

d'une image en se basant sur sa similarité avec l'image précédente. C'est une sorte d'évaluation paresseuse *temporelle*.

L'utilisation de la cohérence temporelle est un concept très général utilisé dans plusieurs domaines et pour de nombreuses applications. Néanmoins, son utilisation est relativement rare dans le cadre de la modélisation fractale. En effet, l'optimisation par cohérence temporelle nécessite la connaissance de la forme visible au pas de temps précédent, puisque c'est sur ces données que l'optimisation est fondée. Or, une fractale est généralement générée à la volée, rendant ainsi très volatile les informations qui la décrivent.

En effet, lorsqu'une scène est décrite par des fonctions de construction, sa définition est *implicite*. Sa description *explicite* (géométrie, matériaux) n'existe souvent que lors d'une observation (d'une évaluation). Parfois, elle n'a même aucune existence dans la mémoire de l'ordinateur, mais n'apparaît que de façon très fugace au sein même du processeur (dans la la pile d'appels par exemple).

Il faut donc faire un effort particulier pour garder la description *explicite* en mémoire. Cet effort consiste généralement à coder une structure de donnée qui stocke et rend accessible cette information ancienne (c'est notamment le cas de XaoS [xao]). Remarquons que la problématique est subtilement différente pour les algorithmes utilisant des descriptions explicites. En effet, ceux-ci offrent la possibilité de *modifier* l'information existante, ce qui est déjà une forme d'optimisation par cohérence temporelle.

3.2.3 Modélisation fractale et langage descriptif

La modélisation fractale est généralement un acte délicat. Trouver des méthodes simples de construction dont peuvent résulter des observations très complexes n'est pas chose aisée. C'est en tout cas moins intuitif que de décrire une forme à une précision donnée[3]. La plupart du temps, c'est une idée qui jaillit sans prévenir, ou un coup de chance, voire même un "bogue". Plutôt qu'un acte de création, les fractales semblent être issues du hasard. Pourtant, certains outils proposent un cadre formel permettant leur modélisation.

Classement

On peut distinguer deux grandes catégories de modeleurs de fractales :

Spécialisé : de très nombreux logiciels permettent la visualisation à plusieurs échelle de fractales très particulières : les ensembles de Mandelbrot, Julia, Markus-Lyapunov. Ces fractales sont très peu paramétrables et l'aspect "modélisation" de ces outils est clairement limité ;

Générique : des représentations plus générique permettent suffisamment de degré de liberté pour rendre l'espace de création intéressant. Les plus couramment employés sont les L-systèmes et les IFS. Remarquons que ces langages permettent bien plus que la modélisation de fractales, mais sont néanmoins très fréquemment utilisé pour cela.

Compte tenu des objectifs de cette thèse, nous nous sommes plus particulièrement intéressé aux méthodes génériques.

Ces langages descriptifs ont des propriétés différentes :

IFS : iterative function system [DHN85, TT95]. C'est un ensemble fini d'opérateurs contractants (usuellement des transformations affines) dont l'application répétée fait converger le domaine initial vers un *attracteur* (la fractale) ;

L-système : système de Lindenmayer [LR76]. C'est essentiellement une grammaire. Un axiome définit la fractale sous sa forme la plus grossière. Des règles de production vont itérativement récrire l'axiome, ajoutant à chaque fois plus de précision. Ce fonctionnement est souvent appelé *système de réécriture* puisqu'il consiste à récrire une chaîne sur elle-même. Au final, des règles d'interprétation permettent à la chaîne produite de "s'exprimer".

[3] comme c'est très majoritairement le cas dans les modeleurs actuels.

Le système de réécriture des L-systèmes consiste, pour un certaine type de règle, à rajouter de l'information. L'idée d'ajouter de l'information est bien adapté à la multi-échelle : on va ajouter de la précision en fonction de la multi-échelle. Cette similitude entre croissance et précision, mise en évidence dans [Coe00, Smi84], fait des L-systèmes un langage bien adapté à une description multi-échelle.

Présentation des L-systèmes

Créer une fractale revient à trouver des fonctions simples dont la composition donnera lieu à des observations intéressantes. Très vite, les L-systèmes (et les grammaires en générale) se sont avérées pratiques pour coder ce genre de construction. En effet, lorsqu'une production crée de nouveaux symboles, ceux-ci peuvent soit être interprétés (observés), soit engendrer de nouveaux symboles (par l'application d'autres règles).

Plus généralement, les langages descriptifs utilisés pour décrire des fractales ne sont ni plus ni moins que de *petits langages de programmation*. «Pourquoi petit ?». De nombreuses contraintes pèsent sur les langages descriptifs de fractales :

Simple : les langages descriptifs sont souvent développés localement dans un but précis. Il n'est donc pas besoin d'atteindre des sommets de perfectionnement et c'est souvent le minimum requis qui est implémenté ;

Restreint : les fonctions d'interprétation et de dérivation d'une fractale sont massivement appelées lors du calcul de l'affichage. Pour assurer de faibles temps de calcul, l'approche courante consiste à proposer un éventail restreint de fonction étant chacune rapidement évaluable ;

Concis : le coefficient $\frac{taille\ de\ l'objet}{taille\ de\ la\ description}$ est historiquement l'une des motivations principales suscitant l'intérêt des fractales. Les langages descriptifs restent fidèles à leur origine et proposent des syntaxes très concises.

Ces considérations évoquent un curieux équilibre entre créativité et expressivité[4]. Les L-systèmes ne seraient-il pas trop contraint pour la modélisation de scènes tridimensionnelles ? Ici, les résultats parlent d'eux même : dans un cadre non multi-échelle, la variété et la richesse des modèles crées sont époustouflantes [PHM99, CXGS02].

De plus, certains travaux s'attachent à donner aux L-systèmes le plus d'expressivité possible :
- les sub-L-systems [Mec98] permettent de constituer un modèle très varié constitué de plusieurs sous-modèles "emboîtés" ;
- le langage L+C [KP03] se propose d'augmenter l'expressivité des L-systems en se rapprochant du langage C++.

3.2.4 Platonic World

Le Platonic World [PGF01] consiste en une formalisation de la modélisation par complexification. Le langage descriptif utilisé est un L-système. Ce travail étant étroitement lié au mien, il est maintenant décrit et discuté.

Concept

Le monde platonique est un monde où tout existe sous forme "d'idée" et où ces idées ne prennent une apparence palpable dans le mode réel que si elles sont observées. En langage informatique, cela signifie qu'il ne faut afficher que ce qui est visible à la bonne précision. Pour arriver à cela, les traditionnelles méthodes de visibilité sont appliquées. C'est la grande série des "*culling*" qui énumère tous les critères pour lesquels une forme visible doit être acceptée ou rejetée.

L'originalité de la méthode de Przemyslaw Prusinkiewicz et de Christophe Godin vient du fait que des graphes multi-échelles y sont représentés par des L-systèmes. Les objets modélisés avec Platonic World sont essentiellement des fractales. Un modèle d'arbre a aussi été écrit (à rapprocher

[4] L'art naît-il de la contrainte ?

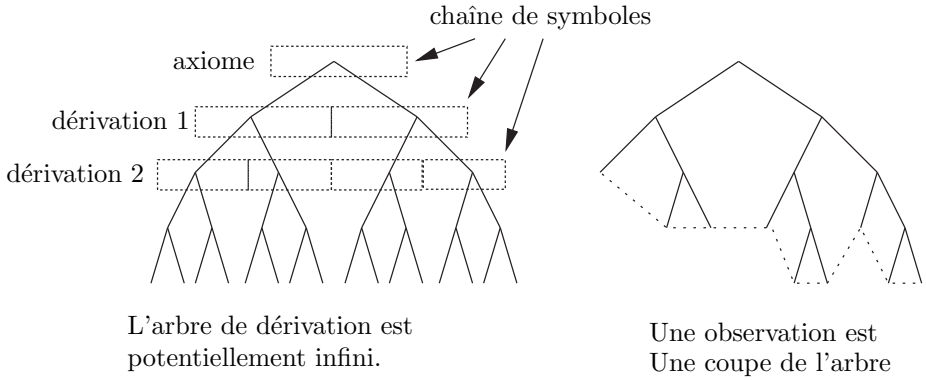


FIG. 3.6 – L'évaluation d'un L-système est ici représentée par un arbre de dérivation (éventuellement cyclique). La chaîne de caractères d'un L-système peut être dérivée à l'infini, mais une observation correspond à une coupe de cet arbre.

avec [WP95, LCV03]), ce qui illustre le fait que le Platonic World n'est pas restreint aux objets auto-similaires à différentes échelles.

Du point de vue de la multi-échelle, c'est l'approche des fractales qui est retenue : un objet est représenté implicitement par les fonctions qui vont progressivement enrichir sa précision. Platonic World représente à ma connaissance la première formalisation du concept de *Modélisation multi-échelle procédurale*. L'échelle d'un objet est décrite comme une fonction de sa distance à la caméra. Des boîtes englobantes hiérarchiques sont utilisées pour déterminer si un objet est hors champs ou non. L'évaluation du L-système est représentée par un arbre potentiellement infini. Une observation y est caractérisée par une *coupe* de l'arbre. Une *coupe* d'un arbre est un sous-arbre comprenant la racine (cf. figure 3.6).

Dans Platonic World, Lorsque l'observation change, la forme change aussi. De quelle manière doit-on mettre à jour l'arbre qui la représente ? Deux approches sont fondamentalement opposées :

Top-down : La forme est mise à jour en partant de sa forme grossière en descendant vers les productions menant à plus de précision. De cette façon, les symboles correspondants à des noeuds grossiers peuvent être inutilement parcourus. En revanche, les détails qui n'existent plus au nouveau pas de temps ne sont pas parcourus inutilement.

Bottom-up : La modification a lieu par le bas. Les formes précises qui ne sont plus requises sont supprimées et les nouveaux détails sont ajoutés là où nécessaire.

D'un point de vue théorique, ces deux approches ont chacune leurs avantages et leurs inconvénients. Principalement, remarquons que la méthode *Bottom-up*, pour être efficace, implique une bonne cohérence temporelle lors d'une animation (du modèle lui-même ou de la caméra). Le choix de l'une ou l'autre des approches n'est donc pas évident et la question est à ma connaissance laissée en suspens.

Discussion de l'implémentation actuelle

Les mondes générés avec Platonic World sont en deux dimensions. Même si la méthode reste conceptuellement la même en trois dimensions, certaines difficultés supplémentaires viennent se greffées aux problèmes.

- le calcul de visibilité est remarquablement plus complexe lorsque la profondeur est permise (à cause des problèmes d'occultation) ;
- l'utilisation du matériel graphique devient incontournable pour obtenir des résultats interactifs ;
- les méthodes de modélisation sont beaucoup plus complexe qu'en deux dimensions ;

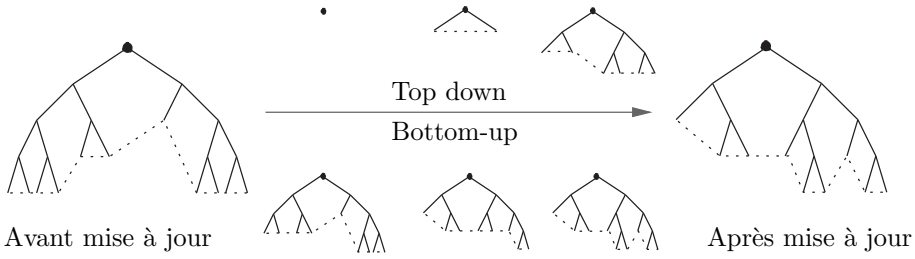


FIG. 3.7 – Lors d'une mise à jour de la figure, deux approches s'opposent. La méthode *Top-down* commence par la racine et descend vers les feuilles. La méthode *Bottom-up* commence par les feuilles et remonte vers la racine (elle redescend cependant lorsque des détails sont ajoutés).

Les mondes générés par le Platonic World ne sont pas animés. Encore une fois, cela ne changerait rien au concept originel. Cependant, la formalisation des parcours *Top-down* et *Bottom-up* n'a pas été pensée pour des objets animés. En effet, l'intérêt du parcours *Bottom-up* implique l'utilisation d'un modèle statique. Dans le cas d'objet animé à tous les niveaux de détail, le parcours *Bottom-up* serait impraticable. Nous voyons les parcours *Top-down* et *Bottom-up* comme deux méthodes parmi d'autres pour *mettre à jour* un modèle en fonction du temps et de la position de la caméra.

Le langage L-système est un langage simple, minimaliste. En un sens, il est au langage de programmation ce que les cartes graphiques sont au CPU : une version spécialisée, simple et rapide[5]. Le système d'interprétation implémenté dans L-Studio est celui de la "tortue", correspondant à des *push/pop* de matrices en OpenGL. Un catalogue de formes tridimensionnelles est disponible pour l'affichage. Les fonctions d'interprétation et les formes disponibles sont implémentées dans le programme principal. L'ajout ou la modification de nouveaux éléments requiert une re-compilation de ce dernier, entravant ainsi gravement les possibilités d'extension. Il est souhaitable, dans *un contexte algorithmique aussi délicat que la modélisation multi-échelle*, d'avoir plus de marge de manœuvre qu'un simple éventail d'objets aux repères tridimensionnels variables. Je défend dans ce manuscrit l'hypothèse selon laquelle les limitations drastiques des langages simples brident la créativité et contraignent le "champ des possibles".

3.3 Loi du cas particulier

Contrairement aux méthodes de simplification, la complexification ne se prête guère à la généralisation et à l'universalisation. Ainsi, les programmes adhérant à cette philosophie sont tous appliqués à des phénomènes très spécifiques : c'est la *loi du cas particulier*. Nous nous proposons ici d'énumérer ces travaux afin d'en extraire un besoin commun. L'énumération commencera par les travaux orientés modélisation. Ensuite, une partie sera réservée aux applications non-académiques. Enfin, nous nous intéresserons aux algorithmes plus orientés rendu temps-réel.

3.3.1 Travaux orientés vers modélisation

Avant de décrire les rares travaux permettant la description de formes tridimensionnelles du plus grossier au plus fin, le cas bidimensionnel est rapidement survolé. Seront ensuite analysés les langages de description, les interfaces graphiques et enfin les outils de modélisation par contraintes (CAO).

[5]Il n'est d'ailleurs pas impossible qu'un formalisme proche des L-systèmes soit un jour implémenté au sein des cartes graphiques.

Deux dimensions

De nombreuses applications 2D utilisent des représentations procédurales. Par exemple, le très répandu format *vectoriel* (*postcript*) est un codage de l'image utilisant des fonctions plutôt qu'une grille de points. Ce codage est très utilisé en cartographie où les deux représentations sont en concurrence : *matricielle versus vectorielle*[6]. De façon plus générale, une simple page *HTML* ou le moindre système de fenêtrage est une image procédurale. En deux dimensions, les représentations procédurales sont décidément bien implantées.

Les travaux multi-échelles sont plus rares. Par exemple, les textures de Perlin offrent la possibilité de garder une représentation procédurale [Per02, PV95]. Mises à part les approches fractales discutées dans la section 3.2, la modélisation multi-échelle 2D n'est en fait guère plus développée que la multi-échelle 3D. Il est bien sûr possible de réaliser des animations 2D multi-échelles avec la quasi-totalité des outils de modélisation. Mais aucun d'entre eux, à notre connaissance, n'est prévu pour cela.

Langage descriptif expressif

J'aimerais ici parler de Slithy [ZS03], une bibliothèque écrite Python permettant de réaliser des présentations similaire à "Powerpoint". En effet, cet outil peut tout à fait être analysé sous l'angle de la modélisation multi-échelle procédurale de scènes massivement animées. Il suffit pour s'en convaincre de voir les présentations qui l'utilisent.

Python permet une incroyable expressivité. Notamment, il est *réflexif* [Gra95], ce qui signifie que le type "code python" existe dans Python[7]. Cette "réflexivité" est en quelque sorte le "comble" de la modélisation procédurale : le langage descriptif est aussi complexe que le langage de programmation.

Ce bel avantage masque malheureusement deux faiblesses :

lenteur : l'affichage de présentations pourtant relativement simples est parfois beaucoup plus lent que ce que l'on pourrait espérer. Python n'est pas un bon langage pour modéliser des formes complexes en temps-réel ;

austérité : écrire une présentation en Slithy est un travail acharné. L'interface graphique n'est ni plus ni moins que votre éditeur de texte favori. De plus, il faut terminer puis relancer le programme pour juger du résultat de quelques modifications dans le code. On est loin ici des menus déroulants de *Powerpoint*.

Les interfaces textuelles, aussi austères soient-elles, dépassent en expressivité n'importe quelle interface graphique. Voyons quels langages du même acabit existent en trois dimensions.

Langages descriptifs tridimensionnels

Il est difficile de recenser les langages permettant une modélisation par complexification en trois dimensions. Non pas parce qu'ils sont peu nombreux, mais parce que tous les langages, dans une certaine mesure, le permettent. Voici donc un petit échantillon représentatif des travaux existants :

shaders : les langages CG ou GRAPE [Nvi02, NF87] offrent la possibilité de la décrire de façon très expressive la fonction d'éclairage. Éventuellement, ils peuvent être utilisés pour faire varier cette dernière en fonction de l'échelle d'observation ;

modeleurs : les modeleurs Maya, 3DSMAX ou Blender [Ali03, Dis03, Ble03] ouvrent leurs fonctionnalités via un langage de script. L'expressivité de celui-ci peut être utilisée pour une description par complexification.

Les interfaces textuelles sont actuellement plus expressives que les interfaces graphiques. Entre autres, elles ne sont pas focalisées sur des cas particuliers. Bien entendu, cette permissivité est gagnée au prix d'un confort d'utilisation très médiocre. On peut considérer que l'interface graphique est le plus haut niveau d'abstraction que peut atteindre un langage de modélisation multi-échelle

[6] La cartographie semble être une application fort intéressante de la modélisation par complexification.
[7] Le langage a "conscience" de lui-même.

procédurale. Dans ce cas, quelques étapes restent encore à parcourir avant la réalisation de cet objectif.

Modélisation interactive

De récents travaux [BPF+03, KN02] proposent pour la première fois des interfaces graphiques permettant la modélisation multi-échelle de forme tridimensionnelle. Uniquement axés sur la *modélisation*, ces systèmes ont certains points communs :

- Ils se basent sur une intervention humaine massive. D'une certaine façon, les seules fonctions automatiques sont la répercussion des modifications à travers les différentes échelles de représentation.
- Ces algorithmes sont chacun spécialisés : l'un pour les arbres [BPF+03], l'autre pour les cheveux [KN02]. Ceci appuie l'hypothèse de la loi du *cas particulier* inhérente à toute méthode de complexification.
- À la fin de la modélisation, la forme est stockée à son niveau le plus fin en vu de son rendu. En effet, les représentations intermédiaires sont *structurelles* et ne possèdent pas leur propre représentation.

Ces travaux pionniers en modélisation multi-échelle pourraient éventuellement se prolonger vers le rendu :

- en effet, on pourrait imaginer que les descriptions intermédiaires possèdent chacune leur propre représentation visuelle ;
- au lieu d'afficher l'arbre à son niveau le plus fin, l'arbre serait alors afficher à des niveaux plus grossiers correspondant à sa decription struturelle multi-échelle.

De cette façon, le moteur de rendu n'aurait qu'à rejouer ces fonctions et à s'arrêter à la bonne précision. Par exemple, un bonsai vue de loin serait uniquement rendu par son enveloppe globale (le houppier). Cette approche est très similaire à celle utiliser pour les arbres décrits dans le chapitre 7. Du point de vue de la complexification, malgré ces quelques lacunes, ces travaux vont décidément dans le bon sens.

Entre simplification et complexification

Les surfaces de subdivision [Loo94] peuvent être vues comme des méthodes de complexification. Malheureusement, l'ajout de détails n'est souvent réalisé que dans un seul but : celui de lisser la forme. Même si d'autres schémas de subdivision existe, ils souffrent la plupart du temps d'un manque de contrôle certain : c'est le même schéma de subdivision qui est répété à l'infini.

D'autres techniques permettent d'ajouter des détails plus variés à un maillage [EDD+95] ou un volume [GLDH97]. Elles s'appuient sur des méthodes par ondelettes [SDS96] ou en sont conceptuellement proche. Ce codage permet un certain dynamisme dans l'affichage : on peut passer d'une représentation à l'autre de façon *continue* en rajoutant progressivement l'influence d'un coefficient.

Malheureusement, le langage descriptif utilisé est très simple. En effet, le choix très restreint d'une base de fonctions d'ondelette impose une discrétisation fine de l'espace. Il devient alors difficile de garder une information sémantique forte : on manipule des tableaux de coefficients. En pratique, toutes ces méthodes partent d'une information représentée à sa précision maximum qui est décomposée en ondelette. A notre connaissance, personne ne s'est intéressé à la génération procédurale de coefficients d'ondelette.

De même que pour la *simplification*, ces méthodes sont absolument nécessaires à la réalisation de scènes tridimensionnelles multi-échelles à un niveau local, mais elles s'adaptent mal, pour l'instant, à de très grandes variations d'échelle.

3.3.2 Travaux non-académiques

De nombreux travaux sont réalisés en dehors du circuit académique. Obéissant à d'autres contraintes que ce dernier, peu d'informations sont disponibles. Il est néanmoins possible de dé-

FIG. 3.8 – Quelques (magnifiques !) rendus réalisés avec Pandromeda. Après expérimentation du logiciel, il est clair que les temps de calcul de ses images sont bien loin du temps-réel.

SpeedTree Gscape Blueberry3D

FIG. 3.9 – La végétation est un terrain de prédilection pour les méthodes de complexification. Ces images tirées de produits commerciaux en sont la preuve.

duire certaines propriétés par une simple observation. Voici donc une liste des produits dont j'ai connaissance agrémentée de quelques commentaires.

Les intervalles d'observation donnés sont *approximatifs* mais donnent une bonne estimation de ce que permet le modèle. Vous pourrez trouver plus d'informations sur le site très complet de VTerrain [vte].

Pandromeda [moj, MKM89]

Ken Musgrave est certainement l'un des plus fervents adeptes de la modélisation procédurale multi-échelle. Son travail sur Mojo Word en est l'indiscutable preuve (cf. figure 3.8). Mojo Word se distingue par des interfaces graphiques très soignées et bien pensées (ce qui n'a rien d'évident en modélisation procédurale). Les méthodes employées semblent être massivement orientées vers la génération procédurale de textures. Les temps de calcul sont assez lents (du moins, plus lent que ce que l'on pourrait espérer). Les résultats, moyennant un peu de patience, peuvent être vraiment spectaculaires. Cet outil a été conçu pour être très modulaire, près à accueillir de nouveaux *plugins*. Très paramétrable, Mojo Word est néanmoins spécialisé pour le rendu de vaste terrain. L'intervalle d'observation est assez large, permettant de zoomer de plusieurs kilomètres à quelques mètres. Dans l'ensemble, c'est un logiciel très prometteur orienté vers la modélisation procédurale multi-échelle.

Speedtree [spe], Blueberry3d [Wid], Gscape [gsc]

Ces logiciels concurrents (cf. figure 3.9) proposent tous un rendu de végétation en temps-réel très convaincant. Ils sont basés sur une utilisation intensive de *Billboards* semi-transparents. Dans un certain intervalle d'observation, le problème de rendu de végétation semble être en grande partie résolu. Blueberry3d se distingue par un intervalle d'observation plus grand. Notamment, une vidéo montre un zoom impressionnant sur une forêt (de $1000m$ à $1m$).

FIG. 3.10 – Quelques images du jeu *Black and white* dans lequel vous êtes un "Dieu multi-échelle".

XaoS [xao], EasyFractal [eas]

Xaos est un logiciel spécialisé dans le rendu de l'ensemble de Mandelbrot. C'est à mon sens un petit bijou algorithmique et une référence en la matière. L'algorithme est basé sur une cohérence temporelle forte assurée par une méthode bottom-up pour le rendu de l'image suivante. EasyFractal semble faire plus ou moins la même chose, mais je n'ai pas pu l'expérimenter.

En vrac

Le jeu Black and White [Stu01] propose de devenir Dieu. Ce dernier, comme tout Dieu qui se respecte, peut tout voir et tout faire. En conséquence, un travail exceptionnel a été réalisé pour permettre de grandes variations d'échelle (cf. figure 3.10).

Virtual Skipper [vsk] est un jeu de course de bateau dont le rendu de l'océan, multi-échelle et procédurale, et impressionnant.

Celestia [cel] est un rendu de l'Univers (notre univers). Esthétiquement, c'est sans prétention. Par contre, c'est du vrai temps-réel avec des zooms de la galaxie vers le système solaire et la Terre (vue satellite). C'est la preuve que la multi-échelle a sa place dans les logiciels pédagogiques.

Descensor [des] est un logiciel qui propose de générer des villes procéduralement. La distance d'observation varie de $1km$ à $1m$. Lors d'un *zoom* arrière, des imposteurs simplifient la géométrie. Très convaincant dans le discours, ce logiciel passe assez mal le cap de la démonstration.

Bilan

Tous ces produits appuient la loi du "cas particulier". En effet, aucun logiciel ne prétend faire un rendu multi-échelle générique. Tous sont spécialisés (villes, fractales, planète). Chaque modèle est observable dans des intervalles bien délimités. La politique de Mojo Word sur les *plugins* est une façon de proposer un cadre commun à l'expression des nouveaux résultats en modélisation procédurale. En effet, aucun format de fichier n'est actuellement capable de représenter un large éventail de données multi-échelles. C'est une sorte de crise d'identité : tous ces travaux ont besoin d'une bannière commune pour aller plus loin dans la variété et l'amplitude des variations d'échelle.

3.3.3 Travaux académiques

La recherche en synthèse d'image a récemment engendré une série de travaux s'appuyant sur la modélisation procédurale multi-échelle. Puissance des machines ou maturité des recherches, le temps-réel est désormais accessible, même avec de grandes variations d'échelle. C'est sans doute l'un des tournants historiques de la multi-échelle : les limitations semblent maintenant venir de l'algorithmique plus que du matériel. L'avenir s'annonce radieux...

Arbres

Les arbres et les forêts sont les travaux les plus nombreux. Weber [WP95] est sans doute le premier à proposer des arbres multi-échelles. Alexandre Meyer utilise des fonctions d'éclairage

procédurales et multi-échelle [MN00], puis une représentation à base d'image [MNP01] pour un rendu plus efficace. Xavier Baele [Bae03] utilise la programmabilité du *vertex-engine* pour générer les branches à la bonne précision. Un soin particulier est apporté aux transitions, réalisées par un fondu enchaîné des modèles 3D dans l'espace image. Thomas Di Giacomo [GCF01] permet une animation d'arbre multi-échelle basculant entre méthodes *réactives* et *prévisibles* selon la position de l'observateur. Marc Stamminger [SD01] utilise un rendu par point (avec lequel il modélise aussi d'autres types d'objets procéduraux). Olivier Deussen affiche des végétations complexes [DCSD02], mais on est plus ici dans une optique de *simplification*.

Prairie

Nous avons réalisé, au début de ma thèse, un programme permettant le rendu temps-réel de prairie animée. L'approche est multi-échelle et procédurale, aussi bien dans la forme que dans l'animation : des imposteurs se transforment en géométrie pendant que des primitives de vent balayent le terrain. A la première version [PC01] a succédé une seconde mouture, améliorant très sensiblement les résultats [GPR$^+$03]. Programme fondateur de cette thèse, il est décrit précisément dans le chapitre suivant.

[BLH02] propose un rendu de prairie à base de points. Je pense que le rendu par points est utile dans le cas de formes pour lesquelles l'information de voisinage est très chaotique (comme les feuilles d'un buisson par exemple). L'aspect longiligne d'un brin d'herbe me semble donc mal se prêter à ce genre de représentation.

Terrain

Les rendus de terrain ont tout d'abord été traités par des méthodes de simplification [LRC$^+$02]. Mais de nombreuses applications ne tirent aucun profit à la connaissance du terrain au niveau le plus fin. De plus, le dynamisme de la représentation nécessaire au changement rapide de point de vue a orienté certaines méthodes vers des techniques de complexification [SD01, MKM89].

Océan

Certains travaux proposent un affichage temps-réel des surfaces procédurales. Ils sont très remarquables car la modélisation procédurale est usuellement utilisée pour modéliser des formes structurées de façon hiérarchique. Damien Hinsinger modélise un océan en temps-réel [HNC02]. La projection inverse de la grille de l'écran sur l'océan et les fonctions d'animation multi-échelles adaptative participent à rendre ce travail très original. C'est à mon goût l'un des plus beaux exemples de réalisation procédurale multi-échelle.

Ville

Longtemps abordées par les techniques de simplification, les villes de Yoah Parish sont entièrement procédurales. Elles profitent de la sémantique forte inhérente à tout son modèle procédurale pour faire des approximations géométriques intelligentes. Toujours dans le milieu urbain, certains travaux proposent des solutions au problème posé par la modélisation de foules multi-échelles. Les approches retenues sont actuellement très axées sur la *simplification* [TLC02, WS02].

Poils et cheveux

Les poils et les chevelures sont des structures très réfractaires aux méthodes de simplification (forme complexe et très animée). Jerom Lengyel [LPFH01] permet le rendu (et l'animation) de poils courts en temps-réel par volumes approximés par tranche (une approche simple et redoutablement efficace). La détection de contour permet l'ajout de la précision sur la silhouette, là où notre œil en requiert le plus. Florence Bertails [BKCN03] modélise, anime et affiche des mèches de cheveux multi-échelles. La méthode d'animation utilisée est remarquable puisqu'elle adapte sa complexité

à la précision requise par l'observation (de loin ou au repos, seules quelques mèches sont prises en compte).

3.3.4 Des besoins similaires

Le petit nombre de travaux cités ici et leur caractère très spécifique à une application particulière montrent la pauvreté, voire l'inexistence, des outils génériques de modélisation par complexification. Il est donc nécessaire d'envisager des outils plus évolués. A la lumière des travaux existants, recensons les principaux besoins afin de tracer les grandes lignes d'un modeleur multi-échelle procédural.

Outils génériques existants

Le but de tous ces programmes est toujours le même : *n'afficher que ce que l'on voit*. Ce but se décompose en sous-objectifs ; c'est la série des *cullings* : *back-face culling*, *view frustrum culling*, *precision culling*, *occultation culling*, ... De nombreux graphes de scène proposent des algorithmes pour répondre à toutes ces façons de réduire la forme visible à l'essentiel [Inv, Per, Jav, Opeb, Opec, Opea, Giz, Ren, Wor]. Quelques fonctionnalités multi-échelles sont disponibles comme des algorithmes de rendu de terrain multi-échelle ou la gestion de hiérarchie de niveaux de détail statique. Néanmoins, tous ces outils sont pensés pour la représentation de scènes connues à l'avance et supportent mal *la génération procédurale à la volée*.

Précision et visibilité

La *complexification* entraîne notamment l'utilisation intensive de *fonctions de transition*. Implémenter une transition est le genre de code qu'il est possible d'écrire en quelques minutes de façon très naïve, mais qui prend *beaucoup* plus de temps si l'on désire la contrôler subtilement. Les transitions utilisent notamment la *précision*, notion vague et toujours délicate à coder de façon générique.

La quasi-totalité des algorithmes de visibilité [MKM89] est basée sur des précalculs lourds et elle est donc inadapté à la modélisation par complexification. Pour bénéficier d'un algorithme de suppression des faces occultées, la seule solution consiste actuellement à le coder soi-même pour chaque objet modélisé (comme dans [SD01]). En conséquence, il serait souhaitable mettre à disposition un système de visibilité générique applicable aux méthodes procédurales. Les autre "*culling*" (*back-face*, *view frustrum*) posent moins de difficultés techniques mais bénéficieraient tout autant d'une implémentation générique.

Modélisation

Le rendu de scènes procédurales multi-échelles produit un flux d'information soutenu et très volatile, s'adaptant au moindre mouvement de la caméra. Ce flux doit être canalisé lors du rendu (précision, visibilité) afin de garder des performances correctes. Remarquons que le temps d'affichage d'un objet est essentiel à sa modélisation. En effet, la répercussion immédiate des modifications faites par le créateur est garante d'un meilleur confort de modélisation[8].

Notamment, l'utilisation de langages descriptifs *complexes* doit être accompagnée d'outils adéquats. En effet, plus un langage est expressif, plus il est possible de commettre des erreurs. Par exemple, de nombreux langages nécessitent même des débogueurs tant ces erreurs peuvent être complexes et difficilement repérables. En synthèse d'images, les formes modélisées sont affichées sur l'écran, ce qui est parfois suffisant pour déterminer l'origine d'une erreur. De plus, certaines modélisations procédurales utilisent des langages très répandus (Python, C++) et bénéficient alors d'environnements existants. Néanmoins, ceux-ci sont mal adaptés à la *complexification* et à son flux d'information dense et versatile.

[8] Cette condition est nécessaire, mais pas suffisante.

Des outils d'aide à la modélisation seraient d'un grand secours ici. Le flux d'information peut être visualisé de façons très diverses. De nombreux travaux montrent les niveaux de détail utilisés par de fausses couleurs. De même, un rendu de la scène vue par un deuxième observateur s'avère souvent très utile pour mieux observer le comportement de certaines transitions. De plus, la complexification implique une connaissance forte sur les objets manipulés. Cette connaissance peut être mise à profit pour créer des micro-environnements de développement parfaitement adaptés au type de l'objet observé.

Plus de richesse

Les travaux cités en 3.3.3 bénéficieraient tous plus ou moins de l'utilisation d'un outil factorisant les problèmes communs (précision, visibilité, cohérence temporelle, persistance, environnement de modélisation...). Mais ils bénéficieraient aussi de leur simple mise en commun au sein d'une même scène tridimensionnelle. En effet, les scènes que l'on pourrait réaliser en rassemblant les travaux existants laissent rêveur : des forêts, des prairies, des montagnes, des océans, des villes... En pratique, une telle mise en commun est délicate. Certes, les programmes sont quasiment tous écrits en C++, mais la mise en commun de deux programmes dont les architectures sont radicalement différentes peut être un véritable calvaire.

Ce problème délicat peut être allégé par l'utilisation d'un *format commun*. Par exemple, il est facile de réunir deux modèles VRML dans un même modeleur pouvant lire ce format. Malheureusement, aucun format standard de modèles 3D ne permet aujourd'hui de coder aisément des représentations procédurales. Nous voilà donc confrontés à une requête essentielle : un langage capable de décrire des modèles procéduraux multi-échelles. Malheureusement, les formats disponibles actuellement sont soit peu expressifs (comme les L-système), soit inefficaces (comme Python). Il semble bien difficile de se passer de l'expressivité et de la puissance de langage tel que le C++.

Mise en commun des intervalles d'observation

Le *format commun* décrit plus haut permettrait non seulement de rassembler différents modèles à la même échelle, mais il permettrait aussi une mise en commun multi-échelle. On pourrait alors, par exemple, utiliser un modèle de planète et le connecter à des modèles de prairie, de forêt et d'océan. De cette façon, on cumulerait les intervalles de validité des sous-modèles. Il deviendrait possible de créer petit à petit des modèles couvrant des intervalles d'observation gigantesques et offrant, pour la première fois, de grandes variations d'échelle et *une grande variété au sein de ces différentes échelles*. En effet, les seuls modèles permettant actuellement de grandes variations d'échelle sont les fractales, très monotones dans leur différentes résolutions.

Cette mise en commun multi-échelle n'est pas aussi évidente qu'une mise en commun "mono-échelle". En effet, les *fonctions de transition*, évoquées dans la section 3.1 , ne sont pas des formules magiques et peuvent être difficiles, voire impossibles à coder. Afin de faciliter leur écriture, les sous-modèles doivent notamment être très paramétrables. Ils doivent offrir suffisamment de degrés de liberté pour pouvoir d'abord se fondre dans la représentation grossière puis faire apparaître continûment leurs détails. Malheureusement, lors de la réalisation d'un seul sous-modèle, il est très facile de négliger ces degrés de liberté nécessaires à une future mise en commun multi-échelle. Ici encore, un environnement de travail peut imposer une certaine discipline ou tout du moins faciliter le plus possible le paramétrage d'un modèle.

Cahier des charges

Dans cette thèse, nous proposons un nouveau modeleur générique multi-échelle présenté dans les chapitres 5, 6 et 7. Nous avons identifié une liste de fonctionnalités dont ce dernier doit être muni :

Formalisme : afin d'épargner au créateur la mise en oeuvre de structures de donnée complexes nécessaires à la représentation de scène représentée par complexification, il est nécessaire de proposer un squelette algorithmique. Si celui-ci est suffisamment souple, le créateur n'aura plus qu'à remplir les parties pertinentes décrivant l'essence même de l'objet modélisé.

Visualisation : de la modélisation par complexification résulte une information dense et volatile. Pour maîtriser ce flux, il est important de bien le visualiser pour mettre en évidence des comportements maladifs par exemple.

Contrôle et interaction : la modélisation procédurale, en allant plus loin dans l'abstraction de la représentation d'une forme, permet de générer celle-ci automatiquement. Mais il important de garder un bon contrôle sur cet automatisme et de pouvoir modifier localement une forme, même si elle est générée automatiquement.

Précision : pour savoir quand et où la précision d'un objet doit être augmenté, le créateur doit pouvoir se reposer sur des méthodes efficaces et adaptables à de nombreux cas. De cette façon, il pourra se concentrer sur les mécanismes permettant d'augmenter la précision, et non sur le calcul de celle-ci.

Visibilité : les scènes tridimensionnelles multi-échelle pouvant se révéler très complexe, il est impératif de détecter les parties non-visibles de celle-ci (hors-champ ou occultée). Pour cela, un calcul de détection d'occultation est essentiel et permet de s'approcher encore plus de l'objectif idéal : ne calculer que se que l'on voit.

Sachant que la modélisation par complexification est dirigée par la loi du *cas particulier*, les choix réalisés pour cet environnement ne doivent en aucun cas restreindre l'éventail des modèles possibles. Clairement, on prend le risque d'imposer un certain style de modélisation mais ce constat est valable pour n'importe quel outil de modélisation. De plus, toutes les fonctionnalités listées plus haut peuvent paraître ambitieuses. Pourtant, l'objectif à atteindre est finalement relativement modeste : il s'agit de rendre *meilleures* les conditions actuelles de modélisation par complexification. Ces conditions étant actuellement très médiocres (quasi-inexistantes), il y a beaucoup de place pour l'amélioration.

Avant de passer à la description de nos contributions à ce problème, le chapitre suivant revient sur l'étude préliminaire d'un cas particulier : l'animation temps-réel de prairies agitées par le vent.

Chapitre 4

Étude de cas : prairie animée en temps-réel

Je vais présenter ici un algorithme permettant le rendu et l'animation de prairies animées sous le vent. Ce travail est un cas typique de modélisation par *complexification* (cf. chapitre précédent). Sa description nous aidera à mieux comprendre l'intérêt de l'outil générique pour la modélisation par complexification décrit dans les chapitres suivants.

Une seconde version de l'algorithme des prairies a été implémentée par Sylvain Guerraz, François Faure et moi. Les différences avec la première en disent long sur l'art et la manière de bien aborder la modélisation par complexification. Ce chapitre est donc composé de deux sections :

Première version : une description de l'article [PC01], suivie d'une critique ;

Seconde version : les améliorations apportés dans [GPR+03] sont analysées et discutées ;

Perspectives : un bilan est tiré de ces travaux.

4.1 Prairie, première édition

Durant la première partie de ma thèse, j'ai terminé la mise au point d'un algorithme de rendu de prairies animées en temps-réel conçu pendant mon DEA [PC01].

Nous présenterons d'abord le contexte scientifique. Nous décrirons ensuite les trois niveaux de détail utilisés, puis la méthode d'animation et enfin les transitions entre niveaux de détail. Cette section se termine par une analyse critique des résultats.

4.1.1 Contexte

Visuellement, le cerveau humain semble avoir un besoin insatiable de *complexité*. Ceci est particulièrement vrai quand on regarde les images synthétisées par ordinateur : les détails (si possibles animés) sont essentiels pour s'immerger dans un monde virtuel.

Motivations

Mais cette complexité est difficilement réalisable dans le cadre d'applications interactives comme les jeux vidéos[1]. Grâce à la puissance des cartes graphiques et à la simplicité et le statisme de la géométrie des environnements artificiel (i.e. réalisé par l'Homme), les scènes d'intérieurs sont rendues en temps-réel avec une bonne qualité. A l'extérieur, les modèles actuels sont généralement moins séduisants, spécialement pour les scènes naturelles. En effet, la richesse de ces dernières est limitée, et elles sont très difficilement animables en temps-réel.

[1] Ce travail a été réalisé en collaboration avec l'entreprise Infogrammes dans le cadre d'un projet PRIAM intitulé «Scènes naturelles animées et interactives pour le jeu vidéo»

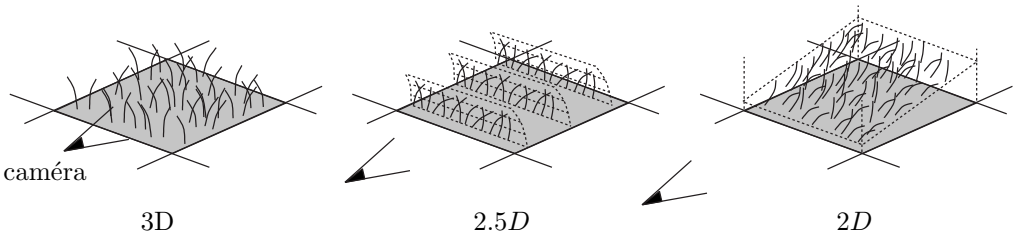


FIG. 4.1 – Trois niveaux de détail sont utilisés pour afficher la prairie.

De plus, comme on l'a vu dans le chapitre précédent, les travaux qui vont dans cette direction sont rares. Le travail présenté ici est parmi les tous premiers à relever ce nouveau et très excitant défi. Nous proposons un algorithme permettant le rendu et l'animation d'une prairie. Les applications visées sont les jeux vidéos, c'est pourquoi le modèle a été conçu pour être observé par un marcheur parcourant la prairie (le cas général étant, comme nous le verrons, plus délicat).

Trois niveaux de détail

Afficher chaque brin d'herbe individuellement anéantirait tout espoir d'interactivité. Comme expliqué au chapitre 2, les méthodes de simplification sont inefficaces devant une géométrie si complexe et animée. Nous avons vu dans le chapitre 3 que les scènes complexes sont mieux représentées par des niveaux de détails de nature différentes (cf. sous-section 3.1.3). Notre solution s'appuie sur l'utilisation simultanée de trois représentations : de la géométrie 3D, des textures volumiques et des textures 2D (cf. figure 4.1).

Pour assurer des performances constantes, lors de mouvements de la caméra, certaines zones de la prairie passent d'un niveau de détail à l'autre. Afin d'éviter toute discontinuité nuisible à la qualité de l'affichage, des fonctions de transitions permettent de passer continûment d'un niveau de détail à l'autre sans altérer la qualité du rendu.

Animation

Les *fonctions d'animation* que nous utilisons dans notre travail puisent leur inspiration dans trois travaux :

- Jakub Wejchert et David Haumann [WH91] décomposent le champ de vitesse du vent en fonctions de base. Leur approche phénoménologique permet un meilleur contrôle et des coûts plus faibles que les méthodes de simulation de fluide. En revanche, les performances étaient encore loin du temps-réel.
- Jos Stam [Sta97] précalcule différents modes vibratoires d'arbre pour les rejouer ensuite en temps-réel. L'animation est très réaliste mais n'offre malheureusement pas beaucoup de contrôle.
- Fabrice Neyret [Ney95] modélise des prairies sous le vent grâce à une représentation volumique qu'il répète sur un terrain. La prairie est animée par déformation du volume.

Nous avons choisi d'animer la prairie par des *primitives de vent*, plus généralement appelées *fonction d'animation*. Elles envoient à chaque instance de chaque niveau de détail une information de mouvement procédural. Elles offrent un contrôle interactif à l'utilisateur.

4.1.2 Trois niveaux de détail

Sont décrits ici les modèles géométriques utilisés pour représenter l'herbe. Trois niveaux de détail sont utilisés. Bien que, par soucis de clarté, la modélisation du mouvement sera décrite plus tard, il est important de signaler que c'est durant la modélisation géométrique que les paramètres utiles à l'animation doivent être considérés.

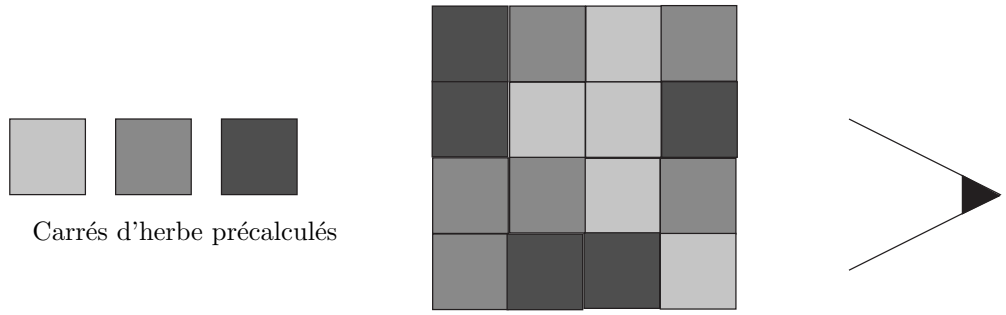


FIG. 4.2 – Le terrain est constitué d'un certain nombre de carrés d'herbe dont la répartition des brins est précalculée. Placés aléatoirement sur le terrain, ils brisent la répétitivité d'un placement régulier.

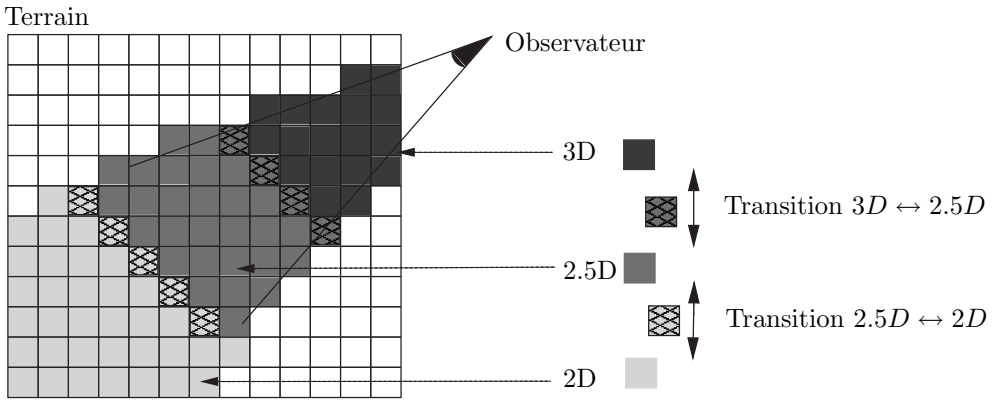


FIG. 4.3 – Les carrés d'herbe visibles du terrain peuvent être représentés par trois niveaux de détail différents. Ces niveaux de détail sont déterminés en fonction de la distance à la caméra. Entre chaque zone, des carrés d'herbe transitent d'un niveau à l'autre.

Le terrain, support de la prairie

Le terrain est une carte de hauteurs et peut donc être divisé en petits carrés de terrain. Un carré est rempli d'herbe : on parlera dans la suite de *carrés d'herbe*. Un certain nombre de carrés d'herbe est précalculé : la position des brins d'herbes, leur taille et leur couleur sont connues à l'avance. En précalculant 2 ou 3 carrés d'herbe, en les plaçant et en les tournant aléatoirement, la répétition n'est pas visible (cf. figure 4.2).

Les carrés d'herbes, selon leur distance à la caméra, utilisent différentes représentations alternatives de l'herbe. Lorsque la caméra bouge, certains carrés d'herbe passent d'une représentation à une autre et subissent une transition. Trois représentations alternatives sont utilisées. Un carré d'herbe est ignoré s'il ne se trouve pas dans le champ de vision (cf. figure 4.3).

Premier niveau : 3D

Chaque brin d'herbe est représenté par une chaîne de lignes (GL_LINES) dont la l'épaisseur varie en fonction de la hauteur.

Le brin est illuminé procéduralement : une couleur verte lui est attribuée (variant plus ou moins au hasard). La couleur est plus foncée à la base du brin afin de rendre compte (tout à fait empiriquement) du fait que moins de lumière parvient au sol. Cette représentation est la seule qui décrit la forme d'un brin d'herbe individuellement par des primitives 3D, c'est pourquoi elle est

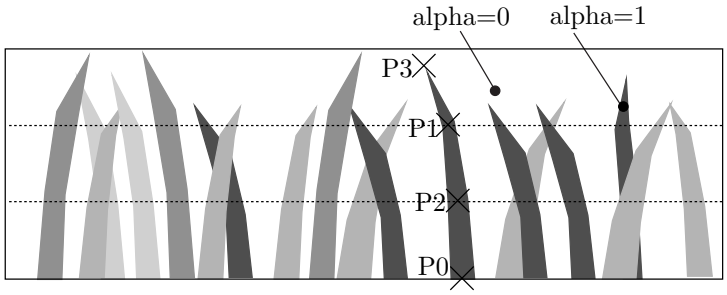


FIG. 4.4 – Exemple (schématique) de texture utilisée pour le niveau 2.5D. La position de chaque brin d'herbe de la texture est connue grâce à une liste de points (les croix sur la figure). Cette information est nécessaire aux transitions vers la représentation 3D.

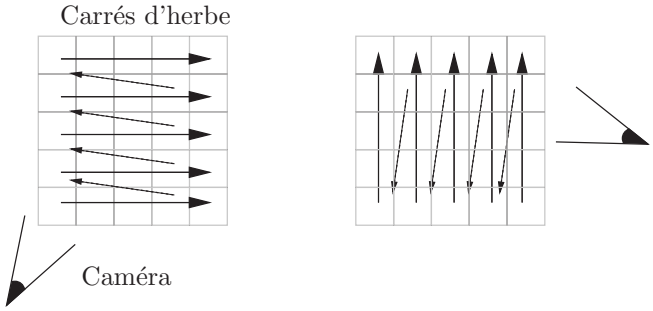


FIG. 4.5 – Dans le cas simple d'une répartition sur une grille régulière (comme c'est le cas du modèle de la prairie), un tri peut être effectué en temps linéaire.

appelée *la représentation 3D* (cf. figure 4.1).

Deuxième niveau : 2.5D

Lorsqu'un carré d'herbe est plus loin, il devient trop coûteux de dessiner chaque brin d'herbe. Le carré d'herbe est alors découpé en tranches horizontales (cf. figure 4.1). Des bandes de polygones semi-transparents subdivisent chacune de ces tranches. Ces polygones semi-transparent sont texturés avec une image RGBA (cf. figure 4.4). Le canal Alpha code la transparence. Chaque tranche est plantée verticalement parallèlement au côté du carré qui qui fait le plus face à la caméra.

La transparence est permise grâce à l'utilisation de la fonction *glBlendFunc*. Cette fonction est un peu capricieuse et requiert un tri des polygones semi-transparents de l'avant vers l'arrière. Compte tenu de la structure régulière du terrain, ce tri est réalisable via un simple parcours intelligent, prenant en compte la position de la caméra (cf. figure 4.5).

Ce niveau de détail, compte tenu de sa construction en tranches, offre une bonne qualité de rendu lorsque l'observateur regarde la prairie d'un angle rasant (comme un marcheur). L'approximation d'un volume 3D à base de tranches de texture 2D nous a amené à appeler, très informellement, cette représentation *2.5D*.

Troisième niveau : 2D

La représentation utilisée ici est une texture représentant de l'herbe directement appliquée sur le sol, c'est pourquoi elle a été nommé *2D* (cf. figure 4.1). Quelques textures de cette représentation sont créés par un rendu en projection orthogonale d'un carré d'herbe 3D quelconque (cf. figure 4.6).

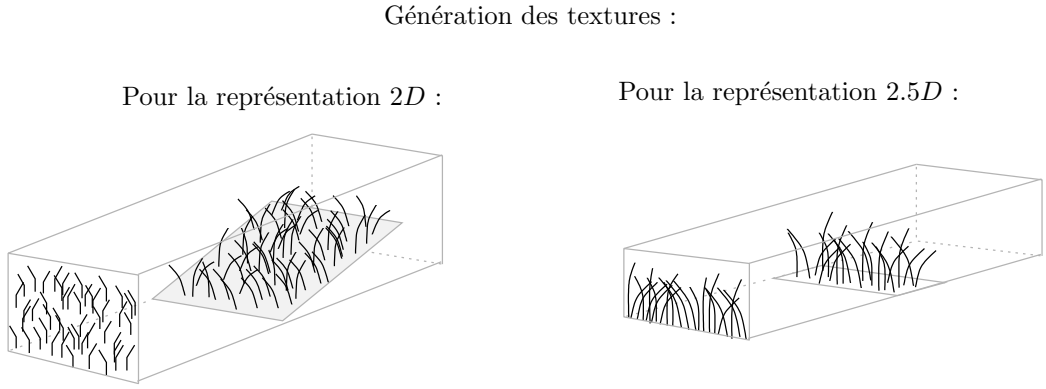


FIG. 4.6 – Les textures des niveaux 2.5D (resp. 2D) sont générées à partir d'une tranche d'un carré d'herbe (resp. d'un carré d'herbe) projetée orthogonalement sur un rendu *offscreen*.

En fait, cette texture est légèrement surélevée et semi-transparente afin que l'on puisse distinguer la couleur du sol. Elle est aussi légèrement inclinée et montre la plus forte pente à la caméra. De cette façon, on approxime mieux le volume que devrait occuper l'herbe.

4.1.3 Transition

Une attention particulière a été portée sur la continuité des transitions. En effet, le marcheur (portant la caméra) va se déplacer dans la prairie. Des transitions entre niveaux de détail doivent être déclenchées pour que la qualité visuelle et le coût d'affichage ne soient jamais dégradés. Nous considérons que les discontinuités visuelles durant le passage d'un niveau de détail à un autre nuisent énormément au confort visuel. Les transitions mises en œuvre entre chaque niveau sont donc des *métamorphoses* continues d'un niveau à l'autre. L'état de complexion d'une transition se mesure par son *avancement* qui n'est rien d'autre qu'un coefficient d'interpolation.

En plus de cette continuité temporelle, les transitions 3D $\leftrightarrow$ 2.5D assurent aussi une continuité "spatiale" : chaque brin d'herbe sur la texture a un homologue 3D (cf. figure 4.4). Nous verrons comment la construction des carrés d'herbe permet cette correspondance, puis nous décrirons les transitions.

Construction d'un carré d'herbe

Chaque brin d'herbe dans une texture 2.5D a un homologue 3D. Il est imaginable d'assurer cette correspondance en construisant, pour chaque carré d'herbe, autant de textures que nécessaire. Mais cela mènerait très vite à un coût mémoire exorbitant.

Lors d'une phase de précalcul, un certain nombre de tranches d'herbe sont générées. Chaque tranche d'herbe contient un ensemble de brins d'herbe dont les caractéristiques sont tirées au hasard : taille, couleur, position de racine et la courbure.

Les textures utilisées pour la représentation 2.5D sont chacune générées à partir de l'une de ces tranches précalculées (cf. figure 4.6). Ces textures sont obtenues par un rendu *offscreen* de la tranche correspondante avec une caméra orthogonale. Chaque carré d'herbe précalculé est alors constitué de tranches placées aléatoirement (cf. figure 4.7).

Ainsi, avec peu de textures (i.e. de tranches d'herbe), il est possible de générer combinatoirement un grand nombre de carrés d'herbe différents. La correspondance des brins d'herbe un à un est assurée par construction de la façon suivante. Un certain nombre de texture sont initialement générées au début du programme. Les tranches d'herbe sont ensuite générées à partir de ces textures.

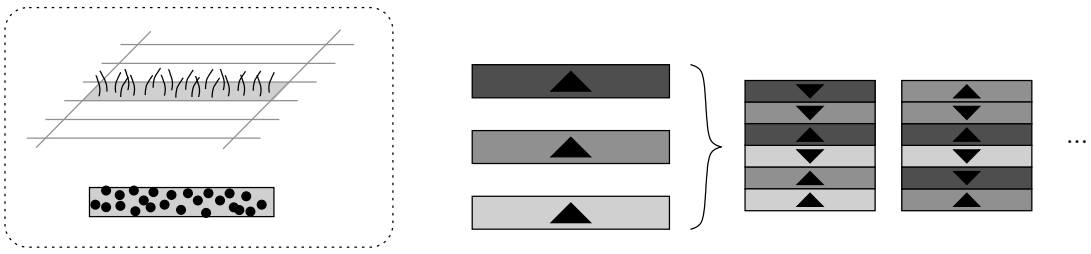


FIG. 4.7 – Chaque carré d'herbe est constitué de plusieurs tranches d'herbe. Comme chaque tranche d'herbe utilise une texture, il est important d'économiser de la mémoire en gardant le nombre de tranche le plus petit possible. En fait, seules quelques tranches sont générées. Pour garder un aspect aléatoire, un carré d'herbe est constitué par une certaine combinaison de ces tranches.

Transition 3D ↔ 2.5D

Lorsqu'un carré d'herbe bascule du premier niveau vers le deuxième, chaque brin d'herbe 3D rejoint son homologue dans les textures 2.5D (cf. figure 4.8). Ceci est réalisé par une simple interpolation du brin durant son mouvement (ce dernier sera décrit en 4.1.4). En particulier, le brin d'herbe glisse sur le sol pour épouser sa nouvelle forme. Ce mouvement est invisible pour un œil non averti car il a lieu à distance suffisante.

Lorsque l'interpolation est terminée, il faut "sauter" d'une représentation à l'autre. Ce passage ne peut pas être réalisé de façon totalement discontinue. En effet, compte tenu des représentations différentes (texture et géométrie), il existe toujours une différence visuelle non négligeable entre les deux niveaux. Pour éviter tout artefact, un fondu enchaîné est réalisé[2].

Remarquons que les distances où ont lieu les transitions sont déterminées de façon complètement empiriques. En effet, il nous a semblé préférable, plutôt que d'établir un espace de mesure à la norme discutable, de laisser le maximum de degrés de liberté dans le paramétrage des transitions. Ces degrés de liberté étant modifiés à la volée, la phase de calibration est très rapide. Les critères de calibration sont tout simplement la qualité du rendu, en éternelle opposition avec les performances.

Transition 2.5D ↔ 2D

Pour basculer de la représentation 2.5D à la représentation 2D, les textures 2.5D s'enfonce dans le sol pendant que la représentation 2D pousse (cf. figure 4.9). Aucune association de brin d'herbe n'est assurée, mais à la distance où intervient cette transition, il n'est de toute façon pas possible de distinguer un brin d'herbe individuellement.

Transition asynchrone (hystérésis)

L'avancement des transitions ne dépendait initialement que de la position de la caméra. En conséquence, pour certaines positions de la caméra, l'avancement d'une transition pouvait rester à mi-chemin entre un niveau de détail et le suivant. Ces positions intermédiaires ne sont pas souhaitables pour la simple raison qu'elles sont plus coûteuses qu'un niveau déterminé et offrent une moindre qualité de rendu. En effet, les états transitoires n'existent que pour assurer la continuité et l'esthétisme est maximal aux états "de repos" (sans transition).

Ceci a tout naturellement mené à des transitions asynchrones qui, une fois déclenchées, évoluent inexorablement jusqu'à leur complexion. Pour limiter les problèmes d'oscillations entre des

[2]La fonction $glPolygonOffset$ évite les clignotements inhérents à toutes surfaces affichées sur le même plan avec OpenGL.

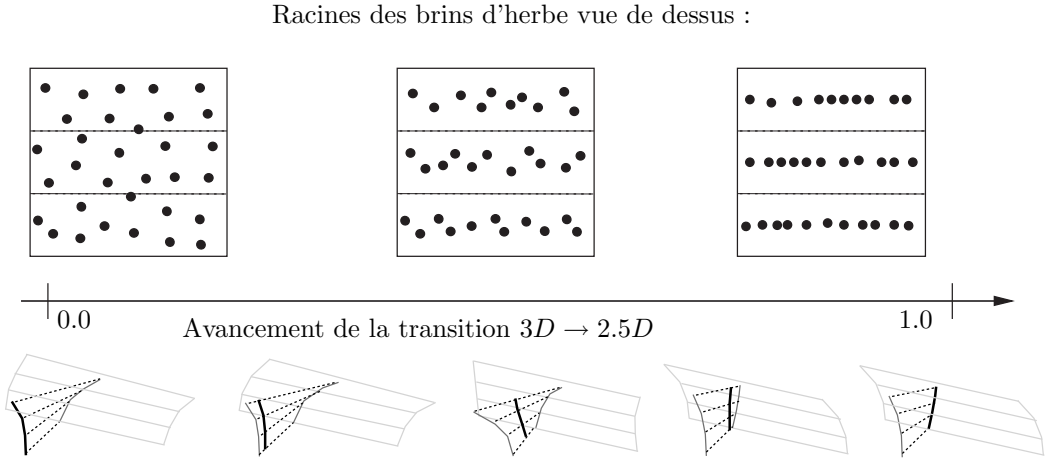


FIG. 4.8 – Lors d'une transition entre le niveau 3D et 2.5D, chaque brin d'herbe d'une tranche rejoint, en cours d'animation, son homologue dans la texture.

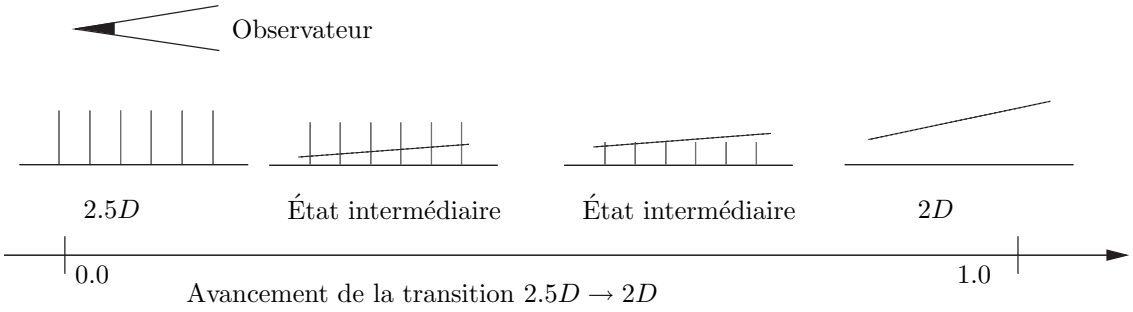


FIG. 4.9 – Les transitions du niveau 2.5D vers le niveau 3D sont assurées par une simple astuce : lorsque qu'une représentation "pousse", l'autre "s'enterre". Cette technique très simple permet d'assurer un confort perceptuel tout à fait acceptable.

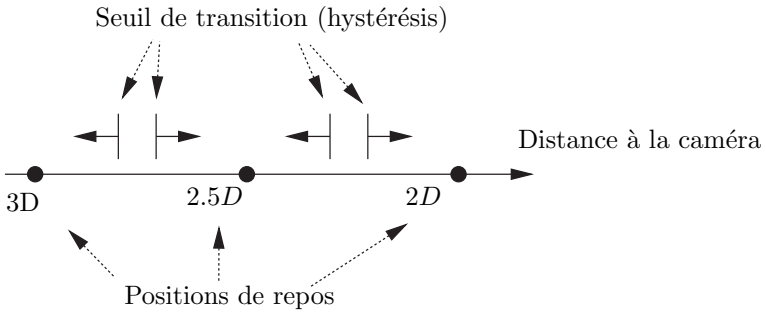


FIG. 4.10 – Les transitions se déclenchent lorsque la distance de la caméra au carré d'herbe passe par certain seuils. Pour éviter les oscillations entre deux niveaux de détails, les seuils sont différents selon le sens de la transition.

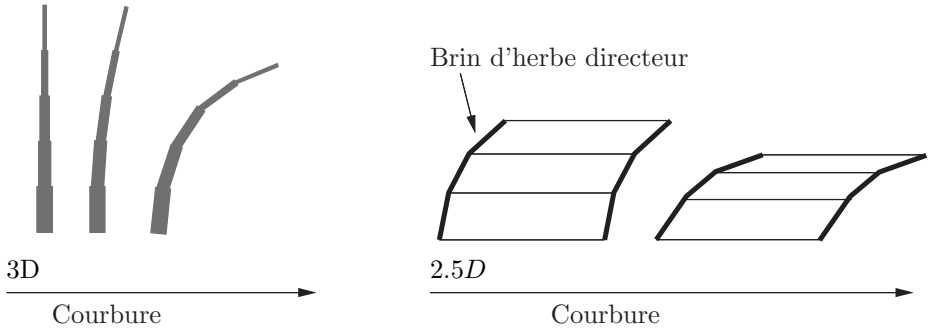


FIG. 4.11 – Exemple de mouvements possibles en variant la courbure .

différents niveaux de détail, des seuils de déclenchements différents sont utilisés d'un niveau à l'autre (cf. figure 4.10). Cette technique est appelé transition par *hystérésis* par [LRC+02].

4.1.4 Fonctions d'animation

Avant de décrire la méthode d'animation et de donner quelques exemples, il est nécessaire de décrire les degrés de liberté que proposent les trois représentations. L'animation de la prairie reviendra à modéliser l'action des "primitives de vent" sur ces degrés de liberté.

Récepteurs

L'animation consiste à simuler la courbure que subit un brin d'herbe dans une direction donnée sous l'action du vent. Afin de permettre cette action, les différentes représentations offrent un certain nombre de degrés de liberté :

3D : un brin d'herbe est paramétré par sa courbure et son orientation (cf. figure 4.11) ;

2.5D : les deux côtés des polygones semi-transparents peuvent bouger de la même manière que les brins d'herbe 3D. C'est pourquoi ils sont appelés *brins directeurs*. Étant le support de tout le polygone, leurs animations induisent un mouvement à chaque brin d'herbe dessiné sur la texture ;

2D : dans cette version, la représentation 2D n'est pas paramétrée (donc pas animée).

L'ensemble des paramètres de chaque instance d'un niveau de détail est appelé un *récepteur* : c'est lui qui reçoit l'information envoyée par les primitives de vent.

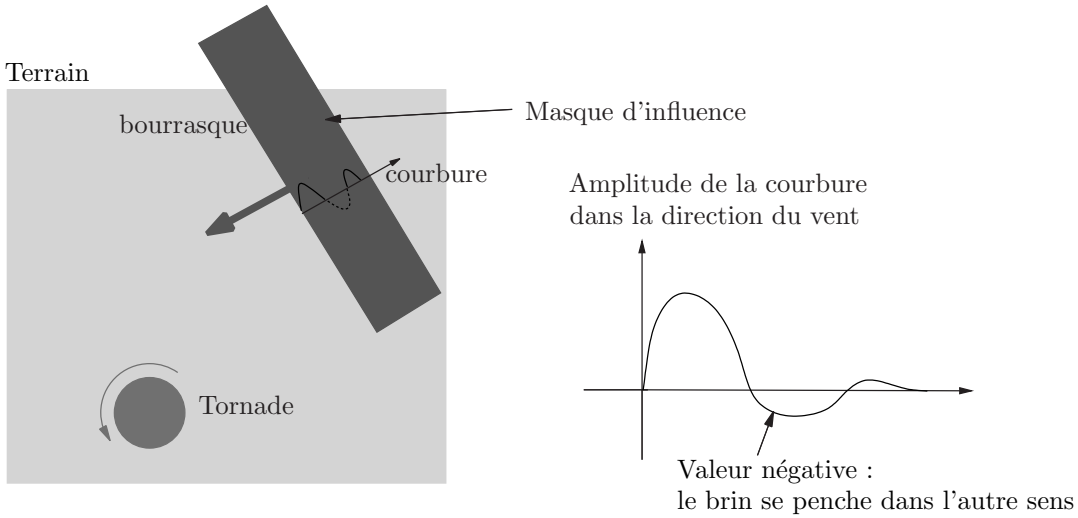


FIG. 4.12 – L'influence des fonctions d'animation est bornée par un masque d'influence. Au sein de ce masque, les fonctions dictent le mouvement que l'herbe doit suivre.

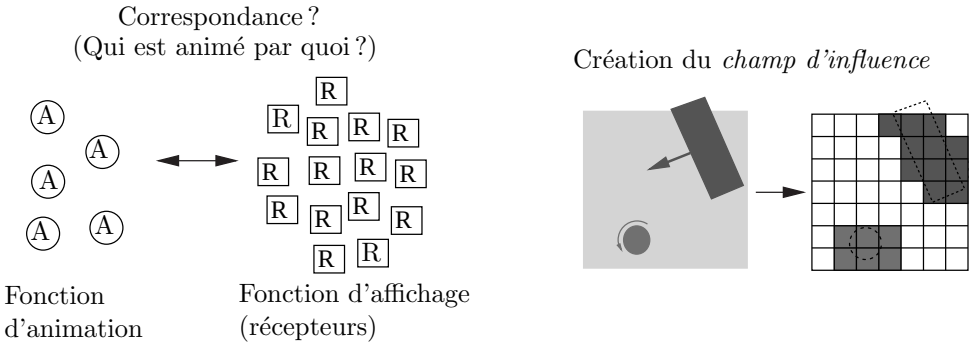


FIG. 4.13 – Chaque *récepteur* peut être animée ou non par des *fonctions d'animation* (bourrasque, tornade). Il est donc nécessaire d'établir une mise en correspondance coûteuse. Un calcul d'intersection entre chaque carré d'herbe et les masques d'influences permet de diminuer ce coût.

Action du vent

Des *fonctions d'animation*, ou *primitives de vent*, modifient la prairie en simulant différents types de vent tels que des bourrasques, une brise légère ou un tourbillon (cf. figure 4.12). Pour de bonnes performances, plutôt que de simuler le vent comme un fluide et de déterminer son action sur chaque brin d'herbe, les fonctions d'animation modélisent *directement* l'action du vent sur l'herbe.

Zone d'influence

Ces fonctions ne s'appliquent pas à tous les brins d'herbes mais seulement à ceux compris dans un certain *masque d'influence*. chaque primitives de vent a son propre masque d'influence représenté par une forme géométrique 2D simple. A chaque pas de temps, tous les carrés d'herbe visibles de la prairie détectent quelles sont les primitives de vent susceptibles de les influencer en calculant leur intersection avec le masque de chaque primitive (cf. figure 4.13).

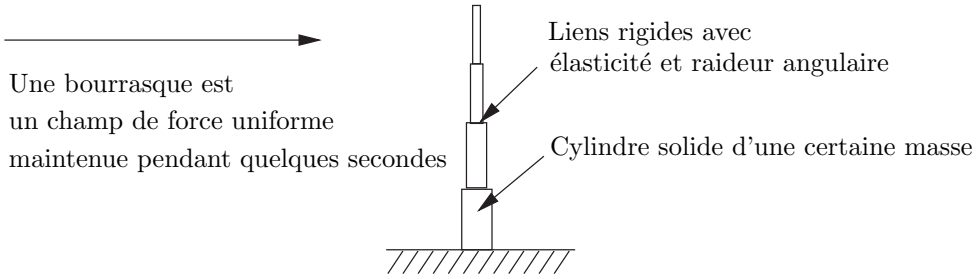


FIG. 4.14 – Une simulation physique permet de déterminer le comportement d'un brin d'herbe. Le mouvement résultant est stocké pour être ensuite rejoué en temps-réel.

Différentes primitives de vent

Une bourrasque est définie par un masque d'influence rectangulaire qui avance sur la prairie (cf. figure 4.12) dans une certaine direction principale. Chaque brin d'herbe (ou brin directeurs) influencé se courbe dans la direction principale et oscille jusqu'à revenir à sa position de repos Il est possible de déclencher une bourrasque de vent sur commande.

Les positions possibles d'un brin d'herbe soumis à une bourrasque sont précalculées dans un tableau de positions indexées par la *courbure*. Il est envisageable de calculer ces positions à l'aide d'un programme de simulation physique. Nous avons utilisé le programme ABDULA, réalisé par François Faure [Fau99] (cf. figure 4.14).

La brise est modélisée par un petit mouvement aléatoire qui influence toute la prairie. L'amplitude évolue selon la formule $ampMax \times sin(\theta_0 + \omega \times t)$. Pour chaque brin d'herbe, la valeur de ω évolue lentement et aléatoirement afin que le mouvement ne ressemble pas à celui d'un métronome. De même, la valeur de $ampMax$ (l'amplitude maximale) évolue pour briser toute sensation de répétitivité.

D'autres primitives ont été modélisées : une tornade, les mouvements induis par un hélicoptère... Grâce à leur nature procédurale, il est très facile de rajouter de nouvelles primitives de vent. Néanmoins, la principale limitation de notre algorithme réside en la manière dont sont combinées les fonctions d'animation.

Combinaison des influences

En pratique, plusieurs primitives de vents peuvent être activées en même temps sur la prairie et passée sur les mêmes endroits. Nous les combinons de façon très simple en leur donnant une priorité. La primitive de vent qui a la plus haute priorité annule les influences des autres primitives. Par exemple, une tornade annulera les effets d'une bourrasque qui annulera les effets de la brise légère.

4.1.5 Résultats

Nous avons présenté une solution au problème des prairies animées sous le vent. Ce travail à fait l'objet d'une publication à *Interactive 3D Graphics* [PC01].

Performances

Le coefficient *Qualité* × *Rapidité* de notre système dépend de nombreux paramètres :
- taille du carré d'herbe ;
- nombre de brins d'herbe par carré d'herbe ;
- nombre de segments par brin d'herbe ;
- nombre de polygones dans la représentation 2.5D ;
- position des frontières de transitions (cf. figure 4.3) ;

Param. \ Qualité	Basse	Moyenne	Haute
Nb. de brins par carré	160	320	500
Intervalle $2.5D$ (distance min/max au marcheur)	3-8	2-12	3-20
Nb. segments par brins	3	4	8
Approx. nb. brins par image	100.000	500.000	1.000.000
Fréquence d'affichage sur une SGI 02	5 Hz	4 Hz	2 Hz
Fréquence d'affichage sur une ONYX	25 Hz	12.5 Hz	8 Hz

FIG. 4.15 – Le coefficient $Qualité \times Rapidité$ dépend de nombreux paramètres. Il est possible, pour chaque plateforme, de les *calibrer* expérimentalement pour trouver le meilleur compromis pour un besoin donné.

Le tableau de la figure 4.15 donne un aperçu de différents paramétrages possibles et des performances résultantes.

L'utilisation de trois représentations *de nature différente* est bien entendu le point clef de ce travail. Elle illustre parfaitement la "loi du cas particulier" exposé en 3.3 : des choix adaptés à un phénomène particulier s'avère bien plus souple et efficace que des algorithmes universels. La gestion combinée de différentes représentations, d'animation procédurale et des transitions continues montrent une nouvelle voie dans la modélisation multi-échelle : la *complexification* (cf. chapitre 3).

Qualité

L'animation, dans son extrême simplicité, est pourtant la grande force du programme (cf. figure 4.16). En effet, cette vaste étendue d'herbe se déplaçant plus ou moins harmonieusement est du meilleur effet sur l'œil humain. L'aspect des brins d'herbe, bien que simplifié à l'extrême, reste crédible. C'est la preuve que l'Homme, dans certains cas, n'a pas besoin d'un réalisme très poussé pour se laisser convaincre.

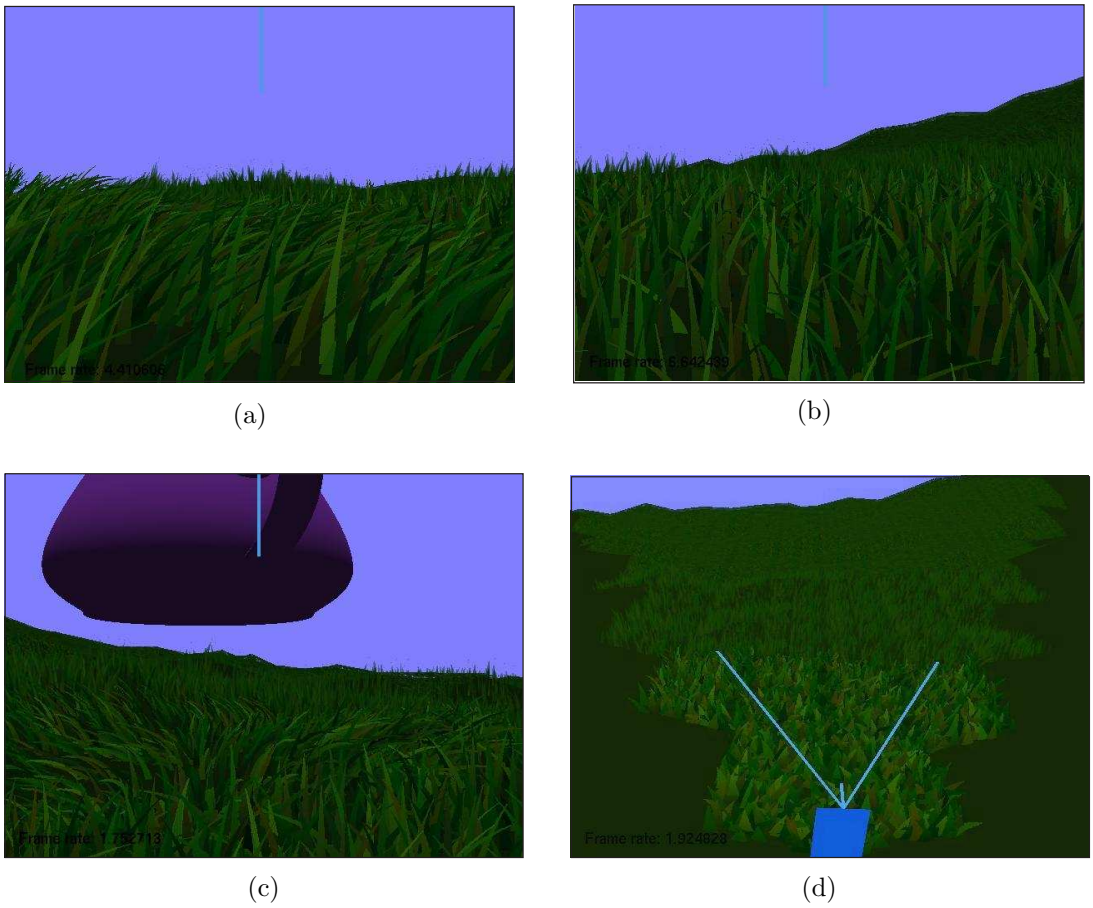
Ce premier résultat soulève des questions d'ordre perceptuel. Était-il possible de prévoir que l'emploi des trois niveaux de détails et des fonctions d'animation donnerait d'aussi bons résultats ? La réponse me semble être négative. La méthode utilisée ne supporte pas de mesure de qualité *a priori* pour la simple raison que l'espace dans lequel une telle mesure prendrait effet n'existe pas avant la création du modèle (ou du moins, il est beaucoup trop complexe). Contrairement à de nombreuses méthodes, on doit créer l'espace de mesure avant d'espérer en tirer quoique ce soit. Cette approche, comme tout notre programme, est extrêmement *empirique*.

Aléatoire

Insistons sur l'aspect pseudo-chaotique de la prairie : c'est lui qui, malgré l'irréalisme du modèle, trompe l'œil humain. Nous sommes en effet exceptionnellement doués pour détecter la répétitivité, qu'elle soit spatiale ou temporelle. Toute monotonie détériore l'esthétique d'une scène.

Des fonctions aléatoires aux lois bien choisies sont utilisées à de nombreux endroits : la couleur, la direction par défaut, l'amplitude maximum, la répartition, la taille... Chaque brin d'herbe est d'une certaine façon unique. Les fonctions d'animation sont aussi "randomisées" dans leur vitesse et leur placement.

Plus généralement, le bon usage de fonctions aléatoires est un facteur essentiel à la qualité de tout modèle procédural décrivant des scènes naturelles.



(a) (b)

(c) (d)

FIG. 4.16 – Quelques captures d'écran de la première version des "prairies sous le vent". (a) : une bourrasque ; (b) : au repos ; (c) : La théière volante génère des coups de vent ; (d) : vue d'ensemble montrant les différents niveaux de détail sous un nouvel angle.

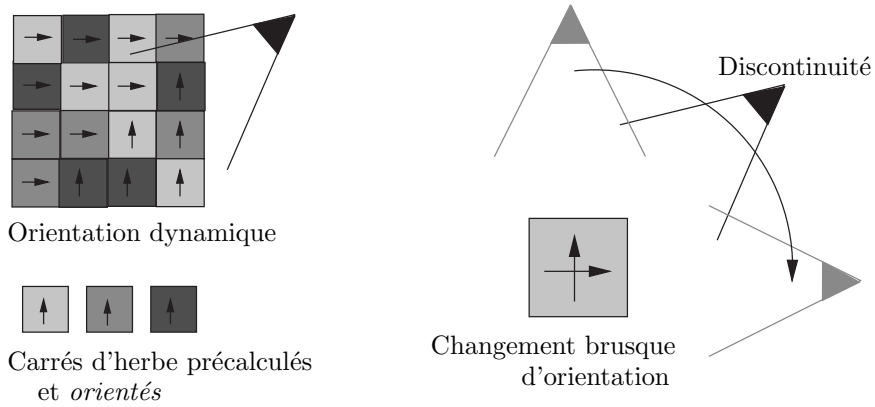


FIG. 4.17 – Les carrés d'herbe, étant constitués de tranches d'herbe orientées, sont eux-mêmes orientés. Si le marcheur tourne autour d'un carré d'herbe sans le quitter des yeux, l'orientation d'un carré d'herbe (en particulier celle des tranches texturées du niveau 2.5D) va changer brusquement, résultant en une discontinuité de l'affichage.

4.1.6 Limitations

Cette version de la prairie souffre de nombreuses limitations que nous décrivons ici. Nous verrons dans la section 4.2 comment elles ont été abordées dans la seconde version du programme.

Combinaison des influences

La méthode de combinaison des différentes influences des primitives de vent est insuffisante. Une simple priorité ne peut pas, par exemple, tenir compte de l'effet *cumulé* de différentes primitives.

Discontinuité dans le cas d'observation "rotative"

La figure 4.2 a été simplifié : les carré d'herbes sont en fait orientés. La détermination de leur orientation à la volée peut parfois entraîner une discontinuité dans la représentation (cf. figure 4.17). Utiliser un simple fondu enchaîné pour remédier à cette discontinuité de l'affichage n'est pas évident compte tenu de la géométrie entre-croisée. En effet, pour trier les polygones (nécessaire pour un rendu avec transparence), il faudrait décomposer chaque polygone semi-transparent. Nous verrons dans la section 4.2 comment ce problème a été corrigé.

Densité

La densité de l'herbe est quasi-constante sur tout le terrain. Il est possible de décréter que certains carrés d'herbe sont en fait désertiques, mais l'effet ne serait pas convaincant car la prairie serait comme crénelée. Une densité variant continûment est nécessaire si on veut représenter un paysage plus réaliste. Malheureusement, les textures précalculées engendrent une contrainte très forte : il doit y avoir autant de brins d'herbe dans une tranche que dans la texture associée. Il est envisageable de construire plusieurs types de texture correspondant à des densités différentes. La difficulté viendrait ensuite de leur assemblage.

Angle de vue

Seul un angle de vue rasant offre une bonne qualité de rendu. Ceci est dû aux niveau 2.5D et 2D qui sont prévus pour être observés d'un angle bien particulier. Cette limitation est sans-doute la plus nuisible. Même si l'on restreint l'observateur à un marcheur (disons 2 mètres au dessus de la prairie), cela ne l'empêchera pas de voir des portions de prairie avec un angle de vue important

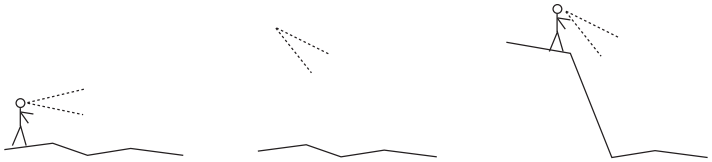


FIG. 4.18 – La qualité du rendu dépend de l'angle de vue. Cette contrainte est très contraignante. Notamment, elle limite les variations d'altitude du terrain.

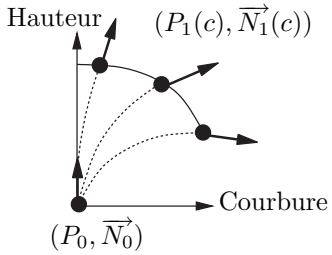


FIG. 4.19 – Les brins d'herbe sont générés par des fonctions splines. Une taille approximativement constante est assurée par une *fonction de rapetissage*. Cette fonction précalculée donne la valeur de P_1 et N_1 en fonction de la courbure.

si le terrain est très dénivelé (cf. figure 4.18). Ainsi, pour assurer une bonne qualité de rendu, non seulement on doit contraindre l'observateur à l'état de marcheur, mais il est aussi nécessaire de limiter la taille des bosses du terrain... Ces contraintes deviennent vite très restrictives.

4.2 Prairie, seconde édition

La première version des prairies a donné suite à une nouvelle mouture principalement développée par Sylvain Guerraz [GPR+03] lors de son stage de magistère. J'ai participé à l'encadrement, principalement effectué par François Faure. Les différentes représentations et leurs animations restent sensiblement les mêmes. En revanche, la flexibilité du modèle a été améliorée, permettant de repousser certaines limitations inhérentes à la première version.

4.2.1 Rendu

Voici les améliorations portant sur le rendu de la prairie.

Génération à la volée

L'utilisation de tableaux de position, utilisés dans la première version, limite la variété des mouvements (cf. sous-section 4.1.2). L'apparence d'un brin d'herbe étant simple et variant peu, il est facile de générer ce dernier à la volée sans jamais stocker sa géométrie. La courbe d'un brin d'herbe est donc calculée grâce à une simple courbe de Hermite.

Il faut cependant faire attention à conserver approximativement la taille du brin d'herbe pour n'importe quelle paramétrisation (tout droit, courbé, couché). Les fonctions permettant de conserver la longueur d'une courbe sont rares et coûteuses. Par soucis d'efficacité, une taille à peu près constante est assurée par une fonction de "rapetissage" (cf. figure 4.19).

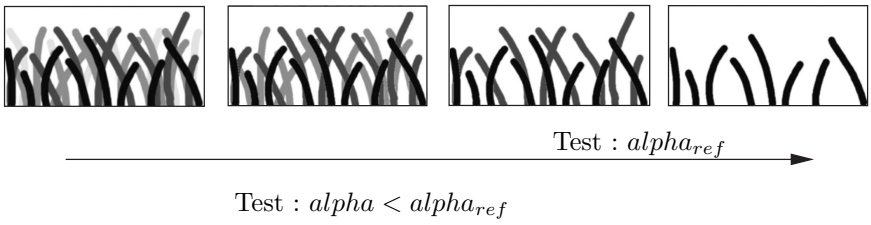


FIG. 4.20 – En variant la valeur de $alpha_{ref}$, la densité des textures varie dynamiquement.

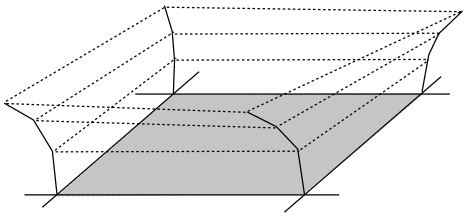


FIG. 4.21 – Au lieu de plusieurs tranches, les polygones texturés sont désormais autour des carrés d'herbe (ceux-ci sont par conséquent plus petit). Remarquons que cela ne résoud pas le problème d'une vue non-rasante : l'observateur, dans ce cas, verra des trous dans le terrain.

Paramétrage des textures

Les textures utilisées sont désormais paramétrées par un facteur alpha. Celui-ci, grâce à l'utilisation du GL_ALPHA_TEST, permet d'afficher plus ou moins de brins d'herbe, selon la valeur du seuil alpha (cf. figure 4.20). Ce système a été utilisé avec succès pour les textures 2D et 2.5D. Il permet, grâce à une carte de densité, d'obtenir des terrains dont la répartition d'herbe est très variable (cf. figure 4.24).

C'est donc en ajoutant un degré de liberté dans le rendu des textures qu'il a été possible de varier la densité. L'ajout de ce degré de liberté est réalisé au détriment de l'association des brins d'herbe 3D et 2.5D. Cela signifie que lors d'une transition entre ces deux représentations, la correspondance des brins d'herbe n'est plus assurée.

Ce choix allège énormément la conception des transitions : il n'est plus nécessaire de garder les coordonnées des brins d'herbe dessinés dans les textures (cf. figure 4.4). Cette perte entraîne une légère dégradation du rendu : durant le déplaçant de l'observateur, l'observateur attentif distingue une onde de discontinuité au niveau des transitions entre la 3D et la 2.5D. Même si cela s'annonce délicat, il semble possible de conserver la correspondance des brins d'herbe tout en gardant un paramétrage de la densité. Affaire à suivre...

Positionnement des textures 2.5D

Afin de remédier au problème de "l'observation rotative", les polygones texturés *entourent* les carrés d'herbe (cf. figure 4.21). De cette façon, la géométrie ne dépend moins de l'angle d'observation. Pour une qualité similaire, le nombre de polygones utilisé est très légèrement supérieur. En effet, l'orientation est moins optimisée pour l'observateur. En revanche, la caméra peut maintenant se déplacer sans contrainte dans la prairie, sans soulever le problème des "l'observation rotative" évoqué plus haut.

Sous-niveaux de détail

Au sein d'un même niveau de détail, la géométrie est échantillonnée plus ou moins finement selon la distance à la caméra (cf. figure 4.22). Cela permet une meilleure adaptation de la précision

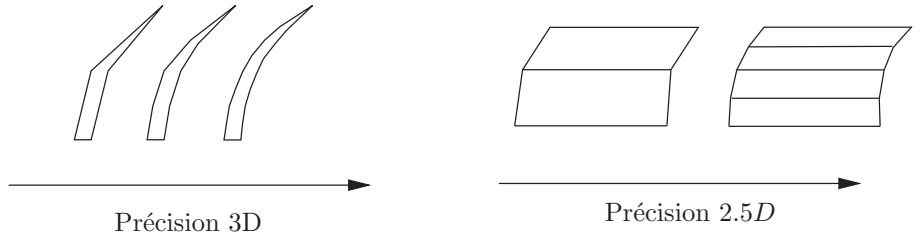


FIG. 4.22 – Dans la seconde version des prairies, l'échantillonnage de la géométrie varie au sein d'un même niveau de détail.

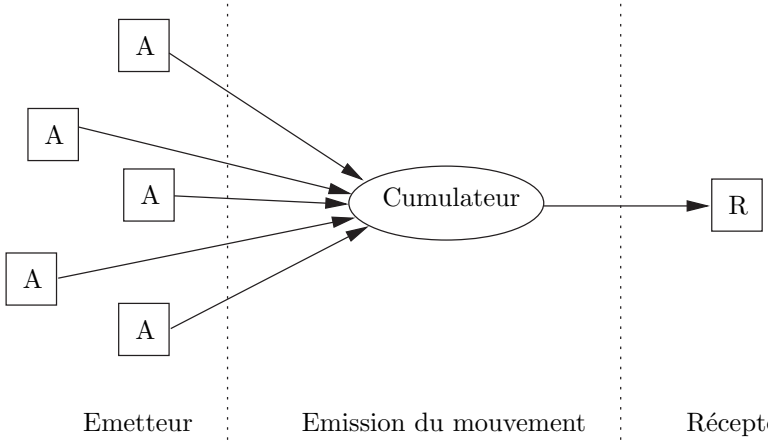


FIG. 4.23 – Le cumulateur combine les différentes influences des fonctions d'animation appliquées à un récepteur. Par défaut, il propose de pondérer les influences aux moyens de simples coefficients.

et ainsi un meilleur *Qualité* × *Rapidité*.

4.2.2 Animation

Un effort a aussi été porté sur les fonctions d'animation.

Moteur d'animation procédurale

Tout d'abord, un moteur d'animation procédurale a été développé. Il gère la vie des fonctions d'animation. Son rôle est le suivant :

- il génère aléatoirement des fonctions d'animation (des bourrasques par exemple) ;
- il introduit un *cumulateur* (cf. figure 4.23) qui combine les effets des différentes fonctions d'animation afin de les transformer en une expression compréhensible par les fonctions d'affichage (l'herbe).
- il optimise la création du *champ d'influence* (cf. figure 4.13) grâce à un rendu *offscreen* basse-résolution réalisé sur la carte graphique.

Traces de pas

Une fonction d'animation d'*écrasement d'herbe* a été ajoutée. Elle permet de simuler procéduralement l'herbe écrasée (par un personnage par exemple) en la couchant temporairement sur le sol (cf. figure 4.24). Pour plus d'informations sur cette fonction, référez vous à [GPR+03].

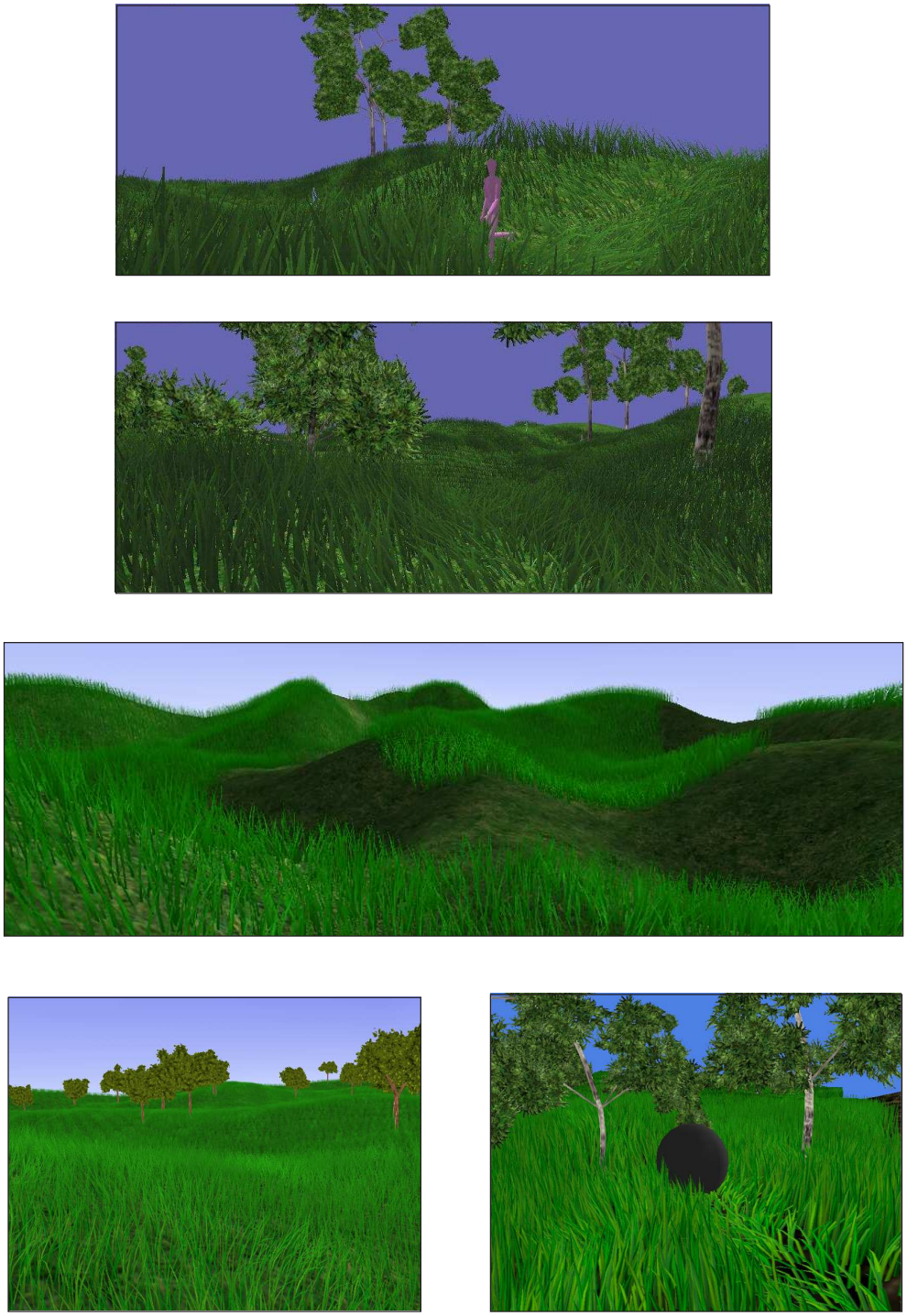


FIG. 4.24 – Voici quelques captures d'écran tirées de la seconde version de la prairie. Le rendu d'arbres a été réalisé par Infograme lors d'une collaboration PRIAM. Leur animation a fait l'objet d'une publication [GCF01] conjointe entre notre équipe et Infogrammes.

4.3 Perspectives

Nous décrirons d'abord quelques améliorations qu'il serait souhaitable d'ajouter à la seconde version des prairies. Nous décrirons ensuite en quoi ces travaux sont une expression du besoin d'un modeleur par complexification (qui fait l'objet du chapitre 5).

4.3.1 De futures améliorations

La prairie dans sa version actuelle est efficace et simple à utiliser. Elle a été combinée à une technique d'animation d'arbres dans [GCF01]. Il est envisageable de l'intégrer à un moteur de rendu dédié au jeu. Pourtant, il reste quelques point à éclaircir.

Mieux définir les échelles d'observation

De nombreuses contraintes sont posées sur l'observateur. Celui-ci doit observer la prairie ni de trop loin (pas plus de quelques centaines de mètre), ni de trop près (pas à moins de un mètre). De plus, notre marcheur ne peut pas observer la prairie d'en haut. Ce sont beaucoup de limites pour de la recherche qui se proclame multi-échelle et il est clair qu'ici, beaucoup reste à faire.

Notons néanmoins que ces limites existent forcément quelque soit le modèle. A l'exception des fractals (malheureusement très peu variées), il n'existe pas de modèles permettant n'importe quelle échelle d'observation. En fait, il serait peut-être souhaitable de mieux préciser les plages d'observation valide de chaque modèles multi-échelle. Une telle mesure permettrait de placer chaque modèle sur un puzzle pour modéliser des phénomènes de plus en plus grand sur des échelles variants toujours plus.

Diversité et texture

Toutes ces prairies manquent de fleurs : une plus grande variété de végétation améliorerait beaucoup la qualité du rendu. Pour cela, une très grande cohérence entre la géométrie et les textures devra être assuré. En effet, il n'est pas question de faire apparaître et disparaître de gros pétales rouges lors d'une transition.

Les textures ont la fâcheuse tendance d'être très peu *paramétrables* et en conséquence très statiques. Les problèmes de variété et de densité en sont des conséquences directes. La solution mise en œuvre pour améliorer le contrôle sur la densité consiste justement à *paramétrer* la texture avec le canal alpha.

De nombreuses méthodes proposent de générer des textures procédurales. Notamment, la programmabilité des cartes graphiques, même embryonnaire, permet de plus en plus de souplesse. Les textures pourront bientôt être composées de sous-textures, engendrées par des fonctions mathématiques simples, des fonctions de bruits... En bref, les textures procédurales ont de beaux jours devant elles. Leur utilisation pour la prairie est très prometteuse.

4.3.2 Vers Dynamic Graph

Les difficultés rencontrées durant le développement de la prairie sont la motivation initiale d'un outil générique pour la modélisation procédurale multi-échelle.

Une modélisation par complexification

Tout d'abord, la prairie, que ce soit dans la première ou la seconde version, est bien un exemple de modélisation par *complexification*. Cela peut ne pas paraître évident compte tenu du fait qu'elle n'a pas été présentée de son niveau le plus grossier vers le niveau le plus fin. Pourtant, la diversité des représentations utilisées, la complexité du langage descriptif (le C++) et la quantité très faible de précalculs pour la version finalement retenue sont des signes qui ne trompent pas.

Remarquons que durant la conception de la première version, toute la structure de la prairie reposait sur des précalculs lourds (chaque brin d'herbe de la prairie existait à son niveau le plus

fin). Cela n'a pas été évident d'assouplir ces précalculs pour quitter doucement la *simplification* et se rapprocher de la *complexification*. Cependant, la complexité du modèle et de son animation n'offrait pas d'autre choix que d'aller dans cette direction. La seconde version de la prairie est d'ailleurs un pas de plus vers la complexification.

Analyse du temps de développement

Lors de la modélisation de la prairie, le temps passé à améliorer son esthétisme a été quasiment nul ! En un sens, ceci est rassurant : je ne suis pas un artiste. Dans le cadre de ma thèse, mon rôle n'est pas de réaliser de belles choses, mais de proposer des moyens pour y arriver.

Néanmoins, la question se pose : où est la difficulté ?

- une grande partie de l'énergie a servi à trouver une représentation adaptée. Il a fallu abandonner toute représentation maillée pour finalement aboutir à des fonctions évaluées *à la volée* ;
- la réalisation des transitions, déjà délicate, a été compliquée par les problèmes d'oscillation évoqués en 4.1.3 ;
- pour déboguer, il a fallu programmer des moyens de restreindre l'affichage à certains niveaux de détail, ou à un certains individus ;
- des difficultés dans la gestion des primitives de vent de la première version a conduit à la réalisation *un moteur d'animation procédurale* (cf. sous-section 4.2.2).

En fait, les seuls représentants du temps passé spécifiquement sur la prairie sont les transitions (ce qui est normal, compte tenu de ce qui a était dit en 3.1). En somme, disons que 10% du temps qui a été nécessaire à la réalisation des prairies a directement servi ce but. C'est peu.

La seconde version

La première version a été développée en six mois par une personne. La seconde version a profité de l'expérience acquise durant la première ainsi que du moteur d'animation procédurale. Pour des résultats bien meilleurs, cette dernière a été implémentée en trois mois par un stagiaire niveau maîtrise et comprend une interface graphique dédiée (cf. figure 4.25). Remarquons que l'utilisation de transitions n'assurant pas la correspondance des brins d'herbe un à un a notablement allégé la modélisation. En revanche, la mise en commun des arbres et de la prairie pour [GPR+03] a demandé une certaine énergie.

Vers un outil générique

La prairie, comme toutes les réalisations procédurales multi-échelles, a un besoin crucial d'un environnement de modélisation. La plupart des critères énumérés dans 3.3.4 sont ici valides :

- calcul de la précision ;
- facilité de mise en place des fonctions de transition entre différentes représentations ;
- mise en commun des modèles ;
- outils de visualisation et de débogage des modèles.

De façon moins évidente :

- persistance de l'information : les questions se sont posées pour les traces de pas ;
- détermination des objets occultés : ceci n'a pas été abordé dans ce chapitre, mais un algorithme de détection d'occultation améliorerait les performances de notre algorithme ;

Le dernier point non abordé est celui de la cohérence temporelle. La première version de la prairie est basée sur une information dynamique mise à jour à chaque pas de temps : les carrés d'herbe visibles restent en mémoire de façon permanente. Dans la seconde version, la génération des brins d'herbe est réalisée à la volée (cf. sous-section 4.2.1), sans aucune mémoire de l'état précédent. Certaines fonctions d'animation nécessitant la connaissance de cet état, il n'a pas été évident de rendre permanentes certaines données (c'est en fait le moteur d'animation qui assure cette permanence). Ici, une bonne gestion de *la durée de vie* de l'information aurait été très utile.

La réalisation des prairies animées est la première étape d'une pensée qui, me semble-t-il aujourd'hui, ne pouvait aboutir que dans la réalisation de Dynamic Graph.

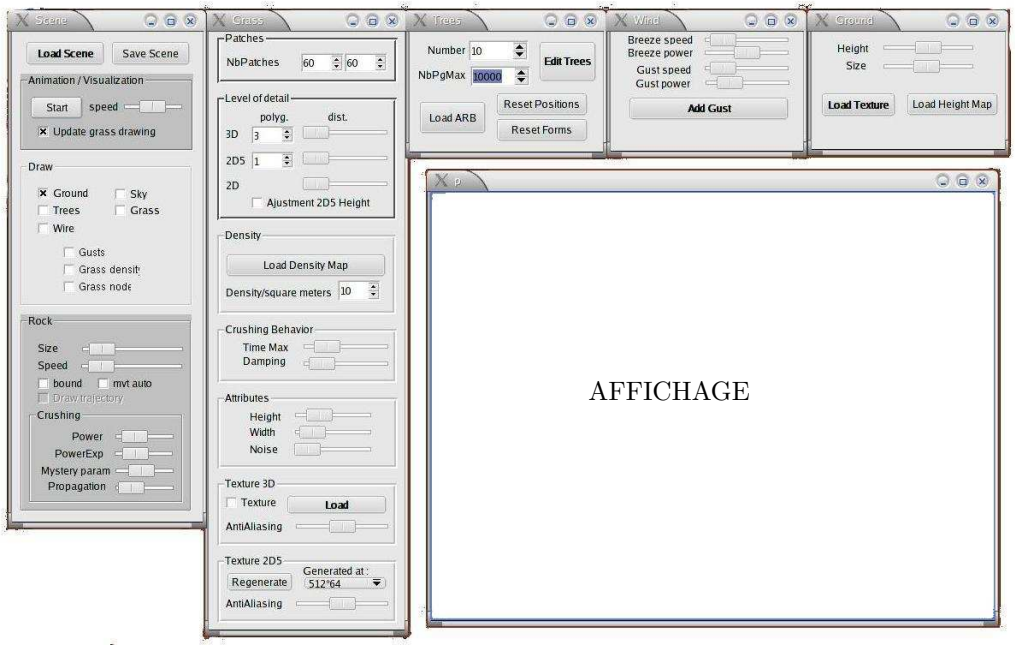


FIG. 4.25 – L'interface graphique de la seconde édition de la prairie permet de régler de très nombreux paramètres : densité de l'herbe, déclenchement des transitions, contrôle des fonctions d'animation.

Chapitre 5

Dynamic Graph : un outil générique pour la modélisation multi-échelle

Ce chapitre propose une vue d'ensemble de Dynamic Graph, un outil de modélisation par complexification générique que j'ai conçu et développé durant ma thèse. Il introduit les chapitres suivants :

- le chapitre 6 détaille le coeur de Dynamic Graph, c'est-à-dire les mécanismes nécessaires à l'évaluation et l'affichage du modèle ;
- le chapitre 7 prend le point de vue du créateur et décrit l'acte de modélisation ainsi que les résultats obtenus.

Voici un survol des différentes sections de ce chapitre.

Introduction à Dynamic Graph : Bien que la prairie n'ait pas été réalisée avec Dynamic Graph, elle est une transition idéale qui reflète parfaitement le cheminement de ma pensée. Ce chapitre débute donc par la description du modèle de prairie tel qu'il aurait été réalisé avec Dynamic Graph. Cela permettra d'introduire le langage descriptif de Dynamic Graph basé sur un nouveau concept : les *amplifieurs*.

Évaluation du modèle : Après cette introduction à Dynamic Graph, nous donnerons un premier aperçu des mécanismes complexes qui assurent la transformation du modèle décrit par le créateur en pixel sur l'écran. Cette transformation est appelée *évaluation*. Notez que dans tout ce document, le terme *évaluation* doit être compris comme un synonyme de *calcul* (et non de validation).

Bilan intermédiaire : la section 5.3 fait un bilan de ce chapitre et introduit le contenu des chapitres 6 et 7.

5.1 Introduction à Dynamic Graph

La prairie, décrite au chapitre précédent, est ici reformulée avec le langage descriptif que propose Dynamic Graph. Cette section propose donc une entrée en matière basée sur cet exemple[1]. Nous verrons tout d'abord comment voir la prairie comme une modélisation par complexification et nous introduirons le concept d'*amplifieur* qui est le fondement du langage descriptif de Dynamic Graph. Enfin, nous aborderons le problème complexe de la *persistance d'information*.

[1] Afin d'éviter toute confusion, je tiens à signaler que la prairie n'a pas été implémentée avec Dynamic Graph. Néanmoins, l'adaptation ne poserait aucun problème.

5.1.1 Vue de la complexification

Une forme visible peut être vue comme une forme grossière auquel on ajoute progressivement et localement des détails. Une fonction qui ajoute des détails est appelée un *amplifieur*. Une scène 3D pourrait donc être décrit par une forme grossière et un ensemble d'amplifieur. Mais Dynamic Graph va plus loin dans l'abstraction de la représentation du modèle : le créateur ne donne pas des amplifieurs, mais des *générateurs d'amplifieurs*. Ceux-ci ont pour fonction de créer des amplifieurs automatiquement. Voyons plus précisément la nature de ces générateurs d'amplifieurs dans le cas de la prairie.

Amplifications successives

Imaginons que la prairie soit initialement définie par une forme grossière : le carré d'herbe 2D. Celle-ci est *enrichie* par les représentations 2.5D et 3D en fonction de la position de la caméra (cf. figure 5.1).

Dans Dynamic Graph, la description du fonctionnement d'un niveau de détail est un *générateur d'amplifieurs*. Le rôle du créateur, lors de la modélisation est principalement de créer ces générateurs. Ensuite, lors d'une observation, Dynamic Graph augmente automatiquement la précision à différents endroits en créant une multitude d'amplifieurs générés par les générateurs.

Il est important de bien distinguer ces deux entités :

un amplifieur ajoute de la précision à un certain endroit du modèle. Dans la prairie, chaque brin d'herbe et généré par un amplifieur "3D".

un générateur d'amplifieurs génére automatiquement des amplifieurs. Le créateur, lorsqu'il invente un nouvel objet, le décrit par des générateurs d'amplifieurs, et non pas directement par des amplifieurs.

Le langage descriptif d'un objet décrit avec Dynamic Graph est donc essentiellement un ensemble de générateur d'amplifieurs. Cette représentation est un pas de plus dans l'abstraction d'un modèle : celui-ci est représenté non plus par les fonctions qui le construisent, mais par des fonctions qui génèrent des fonctions qui le construisent. Remarquons que cette approche très procédurale permet de mieux capturer la répétitivité d'un scène. Par exemple, dans le cas des prairies, les brins d'herbe ne sont pas modelisés un a un : un générateur d'amplifieurs crée de multiples instances d'amplifieur (qui représente chacun un brin d'herbe).

De façon très simplifiée, un amplifieur a deux fonctions possibles :
- il peut afficher une partie du modèle ;
- ou bien il peut déclencher des fonctions de transition pour ajouter de la précision à cette partie du modèle.

Ajout de précision

Un amplifieur est un objet capable de se dessiner ou bien de créer plus d'information (de s'amplifier). Les fonctions de transitions sont réalisées par la création de nouveaux amplifieurs. L'idée générale de cette reformulation est la suivante (cf. figure 5.1) :
- si un amplifieur 2D est suffisamment précis, il se dessine, sinon il engendre des sous-amplifieurs 2.5D ;
- si un amplifieur 2.5D est suffisamment précis, il se dessine, sinon il engendre des sous-amplifieurs 3D ;
- les amplifieurs 3D n'ont pas d'autre choix que de s'afficher car ils ne savent pas générer de sous-amplifieurs (aussi appelés amplifieur fils).

5.1.2 Modélisation de la prairie

Dynamic Graph propose un langage descriptif basé sur le C++ permettant la description d'une scène par un ensemble de générateurs d'amplifieurs.

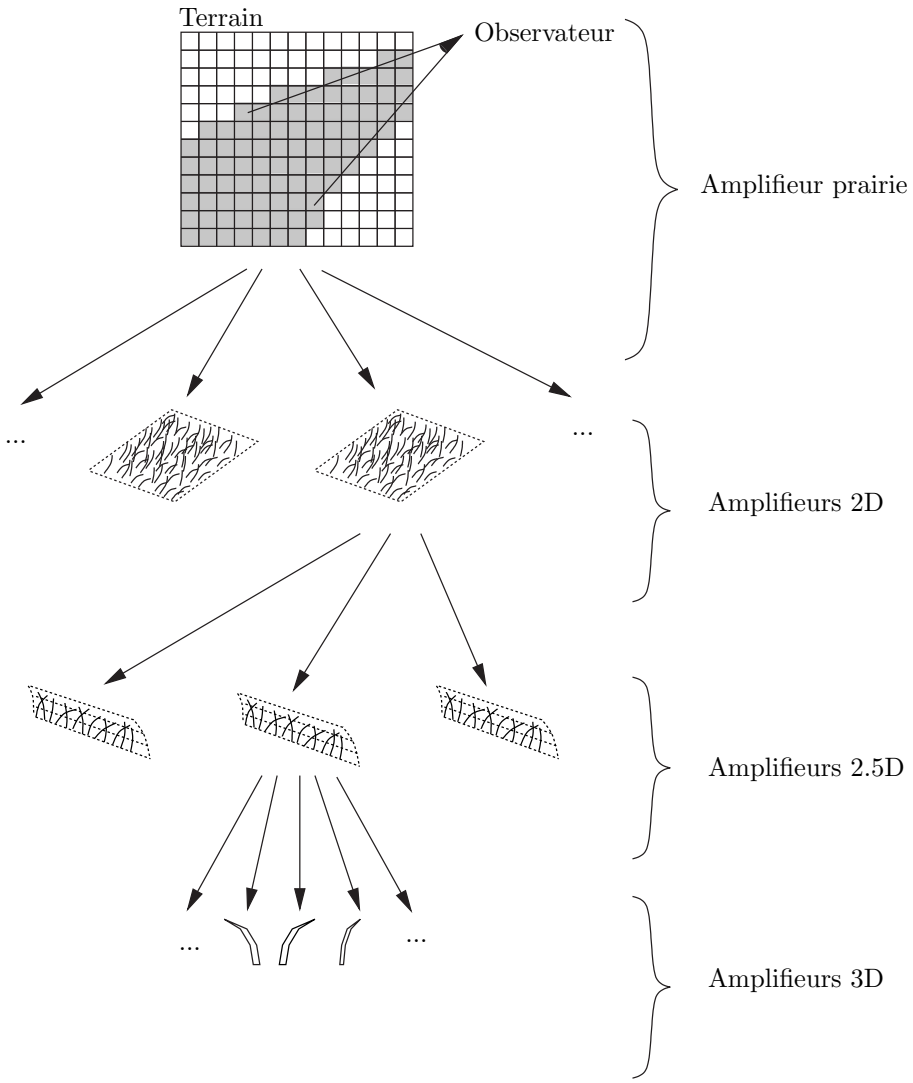


FIG. 5.1 – Un premier amplifieur “prairie” engendre les amplifieurs 2D à condition que ceux-ci soient dans le champ de vue. Ensuite, en fonction de la précision, les amplifieurs vont continuer leur ajout de précision (leur amplification) ou bien dessiner la forme qu'ils représentent.

Création d'un nouveau générateur d'amplifieurs

Avec Dynamic Graph, le créateur doit fournir des générateurs d'amplifieurs qui seront transformés en amplifieurs (i.e. instance de générateur d'amplifieurs) lors d'une évaluation. Très concrètement, le langage de modélisation est le C++ auquel quelques contraintes sont ajoutées. Pour chaque amplifieur, il faut remplir le corps de certaines fonctions :

constructeur : les constructeurs sont appelés lors de nouvelles amplifications (de construction de sous-amplifieurs). ;

affichage : si la précision est suffisante, un amplifieur s'affiche ;

amplification : si elle n'est pas suffisante, il faut enrichir la forme et continuer l'amplification par la génération de sous-amplifieurs (par des appels du constructeur).

Cas des prairies

Afin de fixer les idées, voici, pour chaque générateur d'amplifieurs, le rôle de chacune des fonctions :

générateur d'amplifieurs "prairie" : constructeur : construit une prairie avec un certain pavage, un certain type d'herbe et de nombreux autres paramètres donnés par le créateur... Cet amplifieur est l'amplifieur axiome : c'est le premier a être généré. Il détermine la distance maximale d'observation : si elle est dépassé (i.e. si l'observateur s'éloigne encore), le phénomène d'aliasing par sur-échantillonement (cf. 2.1.2) du modèle apparaît ;

affichage : affiche le terrain ;

amplification : produit les carrés d'herbe 2D s'ils sont dans le champ de vue.

générateur d'amplifieurs 2D : constructeur : construit un carré d'herbe basé sur une certaine combinaison de tranches d'herbe (cf. figure 4.7) ;

affichage : affiche la texture 2.5D (éventuellement en cours de transition) ;

amplification : génère les amplifieurs 2.5D correspondant aux tranches d'herbe du carré.

générateur d'amplifieurs 2.5D : constructeur : construit une tranche d'herbe et calcule l'animation ;

affichage : affiche les polygones semi-transparents correspondant à la tranche d'herbe (éventuellement en cours de transition) ;

amplification : construit les brins d'herbe associés à la tranche d'herbe.

générateur d'amplifieurs 3D : constructeur : construit d'un brin d'herbe et calcule l'animation ;

affichage : affiche d'un brin d'herbe (éventuellement en cours de transition) ;

amplification : aucune amplification n'est codée. Cela détermine la distance minimale d'observation : si elle est dépassé (i.e. si l'observateur se rapproche encore), le phénomène d'aliasing par sous-échantillonement (cf. 2.1.2) du modèle apparaît.

5.1.3 Processus de création

La représentation que fournit le créateur lors de la modélisation est donc un ensemble de *générateurs d'amplifieurs*. Ceux-ci, durant l'évaluation, vont engendrer des *amplifieurs* jusqu'à satisfaire la précision requise par l'observation. Ces amplifieurs envoient finalement des commandes à la carte graphique afin d'afficher le modèle.

Outils de modélisation

Très concrètement, l'acte de modélisation se passe de la façon suivante : chaque générateur d'amplifieurs est écrit en C++ par le créateur. Différents outils d'aide au développement sont fournis et simplifient la modélisation : ils seront discutés dans le chapitre 7.

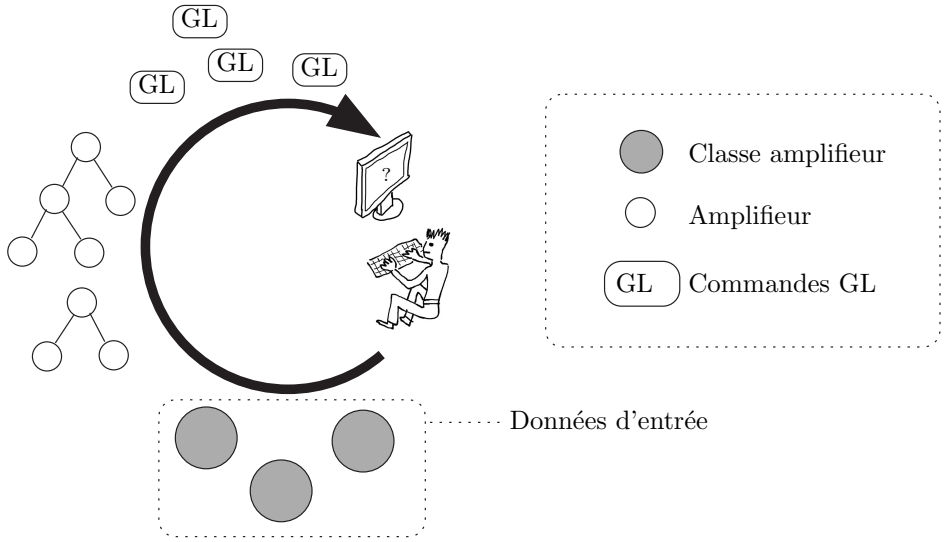


FIG. 5.2 – La nature du langage descriptif varie au fil de l'évaluation. Le langage descriptif initial est une sorte de C++ spécialisé pour la description de générateurs d'amplifieurs. Au fil de l'évaluation, ces classes génèrent des instances de classes, des commandes OpenGL et enfin des pixels sur l'écran.

Évaluation

Une fois les générateurs d'amplifieurs fournis (cf. figure 5.2), la scène est évaluée "à la volée" en fonction de l'observation. Cette évaluation entraîne une production de nouveaux amplifieurs à partir de l'amplifieur axiome. Ils sont stockés dans un arbre appelé arbre d'évaluation. A chaque nouvelle observation (i.e. à chaque pas de temps) la scène est réévaluée et engendre un nouvel arbre d'évaluation.

Rendu

Au fil de l'évaluation, les amplifieurs envoient des commandes à la carte graphique pour dessiner le résultat final, en l'occurrence la prairie sous le vent. Des critères de précision et de visibilité sont utilisés pour n'envoyer que les commandes nécessaires.

5.2 Vue d'ensemble

Dans cette section, la partie évaluation est décrite dans son ensemble. Tout d'abord, les *amplifieurs*, fondement de Dynamic Graph, sont détaillés. Nous verrons ensuite comment l'observation d'une forme engendre un *arbre d'évaluation*. Ensuite, nous motiverons et décrirons l'*organiseur*, dont le but est de garder un maximum de contrôle sur la génération de l'arbre d'évaluation. Enfin, nous verrons quel support Dynamic Graph fournit pour l'animation et porterons notamment notre attention sur la *mémoire* nécessaire à la cohérence temporelle.

5.2.1 Amplifieurs

Fonctions paramétrables

Lors d'une modélisation avec Dynamic Graph, le créateur écrit des fonctions qui vont enrichir la forme : les générateurs d'amplifieurs. Ces derniers sont en fait des fonctions qui renvoient des fonctions :

$$u \overset{g}{\longmapsto} (x \overset{g_u}{\longmapsto} f_u(x))$$

Le générateur g renvoie, en fonction du *contexte* u, la fonction g_u qui va ajouter de la précision à la forme. Le contexte représente en quelque sorte toutes les variables qui sont *inconnues* lors de la modélisation.

Il existe plusieurs raisons pour lesquelles une variable est inconnue :

- dans Dynamic Graph, un modèle est représenté par ajout de précision successif. le contexte peut représenter la partie de la forme qui va être amplifier : il indique où et comment il faut enrichir le modèle ;
- le contexte peut aussi représenter tous les événements qui sont *imprévisibles* lors de la modélisation tel que le déclenchement d'une bourrasque de vent par exemple.
- l'observateur, ainsi que tous les paramètres influençant l'affichage, font aussi partie du contexte : la position de la caméra, le temps qui passe, la précision requise... Tous ces paramètres changeront le comportement des amplifieurs et ne sont connus que lors de l'affichage.

La modélisation proposée par le langage descriptif de Dynamic Graph impose en quelque sorte un niveau d'abstraction supplémentaire. Plutôt que d'appliquer plusieurs fois la même fonction (ou amplification) à une forme, le créateur choisit de décrire une façon automatique pour appliquer ces fonctions : les générateur d'amplifieurs. Dynamic Graph est donc d'autant plus performant que les fonctions sont répétitives.

Une représentation orientée-objet

Pour des raisons de performance, le langage C++ a été choisi comme fondement du langage descriptif. Nous discuterons des conséquences de choix dans la section 8.1. Un générateur d'amplifieurs g est une classe dont chaque instance d'objet est un amplifieur g_u. Le contexte u représente principalement les paramètres du constructeur. C'est la représentation utilisée par Dynamic Graph : une forme visible est un ensemble de *classes* écrites dans un langage orienté-objet : le C++.

f_u est un objet-fonction qui ajoute de la précision à la forme. Il est appelé *amplifieur*. La classe F qui le génère est un *générateur d'amplifieurs*. Dans Dynamic Graph, le créateur s'exprime donc en définissant de nouveaux *générateurs d'amplifieurs*.

Les paramètres contenus dans u, c'est-à-dire les paramètres du constructeur du *générateur d'amplifieurs*, font partie du *contexte*. Le *contexte* représente les données qui dépendent de l'*observation* (comme la position de la caméra). Lors de la modélisation, ce sont donc les "inconnues" dont la valeur ne sera révélée que lors d'une observation particulière.

On distingue généralement deux parties dans un objet : les variables et les fonctions. Dans Dynamic Graph, c'est la partie fonctionnelle des objets qui est la plus importante. Un objet est avant tout *un ensemble de fonctions* qui vont être évaluées pour répondre à une *observation*.

Les variables membres, c'est-à-dire la zone mémoire dynamique de l'objet, doivent être considérées comme un support des fonctions. Concrètement, elles servent la plupart du temps à stocker les résultats intermédiaires des fonctions. Le résultat final d'un amplifieur est l'affichage de la forme enrichie.

Génération de nouveaux amplifieurs

La description d'un modèle consiste essentiellement en la donnée des générateurs *amplifieurs*. Un autre type de classe est cependant nécessaire : les *fonctions d'échange*. Elles produisent des objets très particuliers dont le but est d'assurer le bon déroulement des opérations. Notamment, la fonction de *production de l'axiome* produit l'*amplifieur axiome* qui représente la scène à sa précision la plus grossière.

L'algorithme de rendu récupère ces classes pour *évaluer* la forme visible en fonction d'une *observation*. L'observation fournit aux *générateurs d'amplifieurs* les paramètres nécessaires à la production des *amplifieurs*. C'est l'évaluation de ces *amplifieurs* qui va afficher la forme visible (cf. figure 5.3).

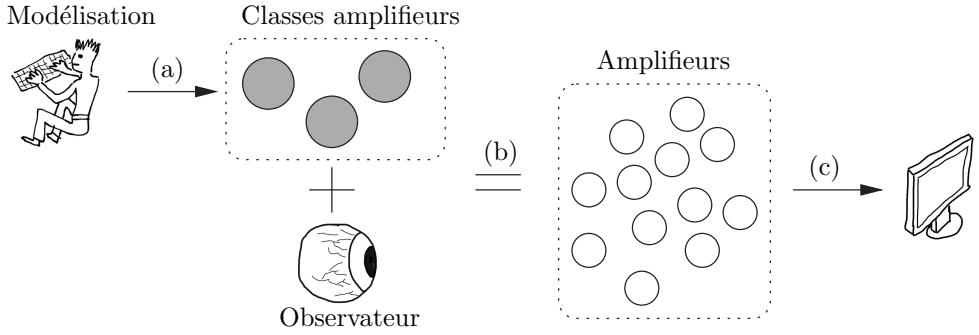


FIG. 5.3 – (a) Le créateur, durant la phase de modélisation, produit les *générateurs d'amplifieurs*. (b) Celles-ci produisent, grâce aux données fournies par l'observation, les objets *amplifieurs*. (c) Les *amplifieurs* produisent l'image.

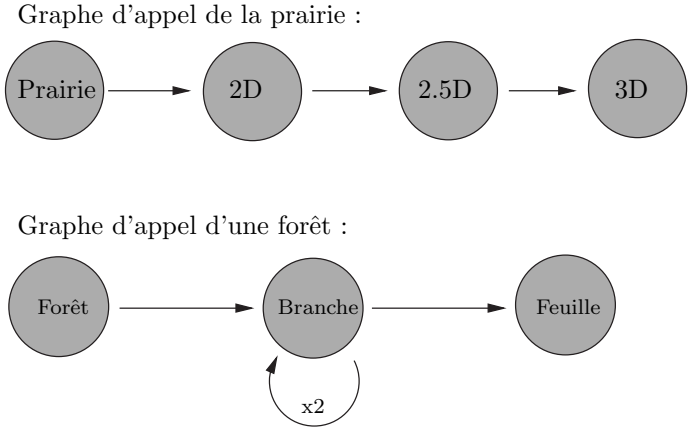


FIG. 5.4 – Les générateurs d'amplifieurs s'appellent les unes les autres. De fait, Dynamic Graph permet une représentation très concise des structures répétitives. Par exemple, le générateur de branche, en s'appellant lui-même, permet de décrire un arbre très succintement en utilisant sa structure "pseudo-fractale".

Lorsqu'un amplifieur ne suffit pas à la précision requise par une observation, il peut appeler d'autres amplifieurs qui vont continuer le travail qu'il n'a pas pu mener à terme. C'est naturellement au créateur de spécifier la manière dont un amplifieur peut générer des amplifieurs fils. La succession de ces appels définit un *arbre d'évaluation* (cf. figure 5.4).

5.2.2 Arbre d'évaluation dynamique

Dans Dynamic Graph, chaque observation (i.e. affichage) nécessite une nouvelle évaluation de la forme. Une forme est évaluée par l'évaluation de tous les amplifieurs qui vont lui ajouter de la précision. Ces amplifieurs sont stockés dans un *arbre d'évaluation*. La génération de cet arbre s'arrête lorsque les critères de précision et de visibilité sont remplis.

Observation

Dans Dynamic Graph, on peut considérer la forme comme une fonction de l'observation. L'observation est notamment caractérisée par la position de la caméra et le temps. Plus généralement, l'observation est l'ensemble de tous les paramètres qui influent sur le rendu. Par exemple, on peut

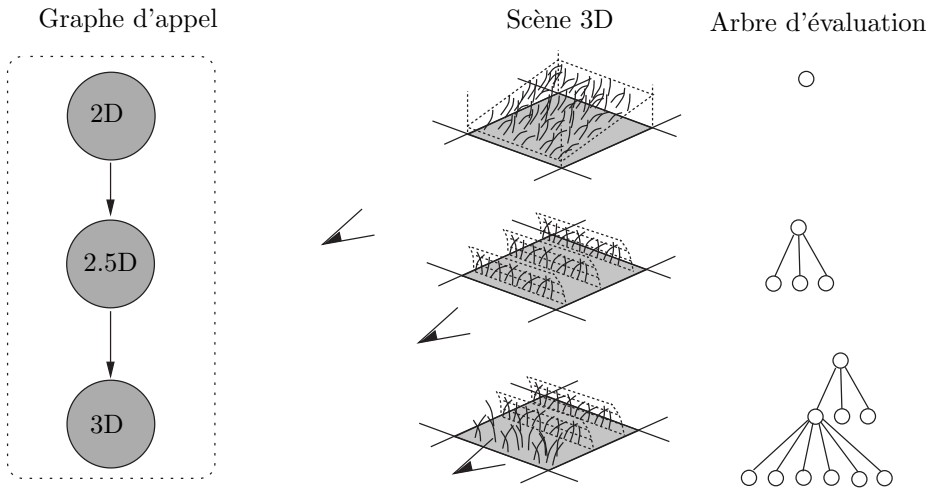


FIG. 5.5 – Lorsque l'observateur se rapproche, l'arbre d'évaluation se développe. Il s'arrête lorsque l'objet est suffisamment précis ou non-visible.

décider d'augmenter ou de diminuer le coefficient $Qualité \times Rapidité$. La précision fait donc aussi partie de l'observation. Une forme peut être observée par plusieurs caméras. Dans ce cas, toutes les caméras font partie de l'observation.

Vous l'avez compris, l'observation est utilisée ici dans un sens très large. Du point de vue de la modélisation, l'observation représente indirectement toutes les inconnues, c'est-à-dire les paramètres des constructeurs des *générateurs d'amplifieurs*.

Construction de l'arbre d'évaluation

Lorsqu'un amplifieur ne suffit pas à la précision requise par une observation, nous avons vu qu'il peut appeler d'*autres* amplifieurs. La description proposée par Dynamic Graph impose que ces *autres* amplifieurs soient en fait engendrés par l'amplifieur en manque de précision. Ce choix est d'une importance capitale : il impose une description multi-échelle hiérarchique (ce choix sera discuté dans la section 8.1). Chaque noeud de l'arbre est un amplifieur et les fils d'un noeud sont les amplifieurs que celui-ci a créé. Cet arbre est appelé l'*arbre d'évaluation* (cf. figure 5.5).

Décomposition de l'évaluation

L'arbre d'évaluation représente en fait *le résultat* de l'évaluation. Plus précisément, il représente les étapes intermédiaires vers le résultat final : l'affichage sur l'écran.

Pour mieux comprendre cet arbre, on peut imaginer une personne faisant un immense calcul mathématique (une grosse équation) sur un tableau. Ce calcul étant très complexe, il préfère décomposer le calcul en plusieurs sous-problèmes. De cette façon, lorsqu'il sera bloqué sur un sous-problème, il pourra continuer sur un autre. Pour faire cela, il n'utilisera pas un tableau mais plusieurs petites ardoises. Lorsqu'il reprendra une ardoise, il pourra continuer le calcul qu'il y avait dessus car toutes les étapes y sont gardées en mémoire.

Dans l'arbre d'évaluation, chaque noeud est un *amplifieur*. Chaque noeud (i.e. amplifieur) de l'arbre d'évaluation est "l'ardoise" qui permet de retenir les calculs. Un nom plus précis de cet arbre serait donc : *arbre de résultats partiels de l'évaluation*. Pour simplifier, et par "abus de langage", nous employons le terme : *arbre d'évaluation*.

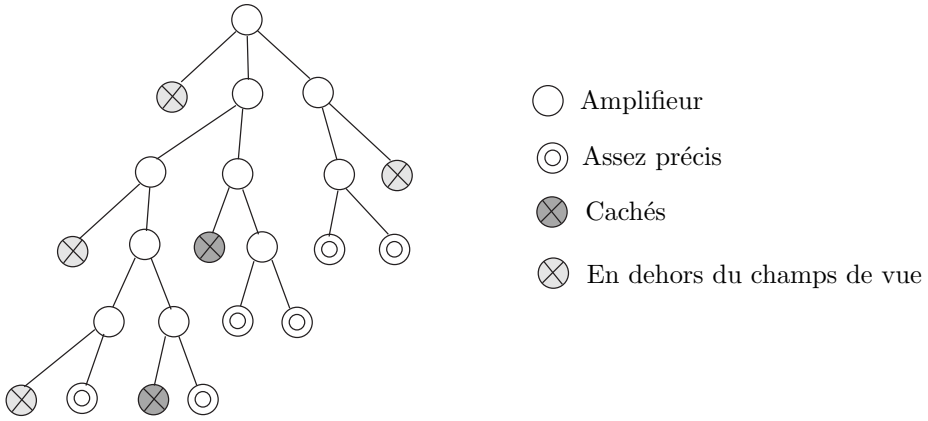


FIG. 5.6 – En fonction de l'observation, l'arbre d'évaluation va se générer jusqu'à sa complexion, c'est-à-dire quand toutes les conditions d'arrêt des feuilles sont positives. Comme les conditions d'arrêt dépendent de l'observation, l'arbre d'évaluation est différent à chaque affichage.

Critère d'arrêt

On peut, en connaissance d'une observation, calculer la précision d'un amplifieur. La précision joue un rôle essentiel dans les critères d'arrêt de l'évaluation de la forme. De plus, elle permet le calcul de *la maturité*. Cette dernière est un simple coefficient facilitant la réalisation de transitions continues entre niveaux de détail (c'est-à-dire entre amplifieur père et fils).

La forme contenue par un amplifieur peut ne pas être visible pour deux raisons :
- il est hors-champ ;
- il est caché par un autre objet.

D'une part, un simple calcul d'intersection entre la boîte englobante de l'amplifieur et la pyramide de vue est réalisé. D'autre part, un algorithme de visibilité est mis en oeuvre pour détecter les faces occultées afin ne pas afficher la géométrie cachée par d'autres géométries.

La construction s'arrête quand tous les amplifieurs sont soit assez précis, soit non visibles (cf. figure 5.6).

5.2.3 Une évaluation assistée

L'évaluation du modèle dans le but de son affichage est un procédé complexe assuré par Dynamic Graph. Avant de détailler dans le chapitre suivant les subtilités de cette évaluation, nous donnons ici un premier aperçu rapide.

Un *organiseur*, au moyen de *visiteur* (voir plus bas), contrôle l'ordre de l'évaluation de chaque amplifieur. Ce contrôle est nécessaire pour de nombreuses raisons, notamment à cause des contraintes émanant de l'algorithme de visibilité. Ce contrôle nécessite un échange complexe entre l'organiseur (via les visiteurs) et les amplifieurs.

Visiteur

Un visiteur est une fonction (un objet) qui parcours l'*arbre d'évaluation* et applique un traitement sur tous les noeuds (les amplifieurs). Chaque amplifieur est spécialisé grâce au polymorphisme et au mécanisme des fonctions virtuelles [Cha98]. Le traitement appliqué aux noeuds est donc différent pour chaque couple (visiteur,amplifieur).

Un visiteur sait comment stimuler un *amplifieur* : de leur échange naît l'information désirée. Voici les visiteurs couramment utilisés dans Dynamic Graph :

générateur : ce visiteur construit l'arbre ;

destructeur : ce visiteur détruit un arbre d'évaluation ;

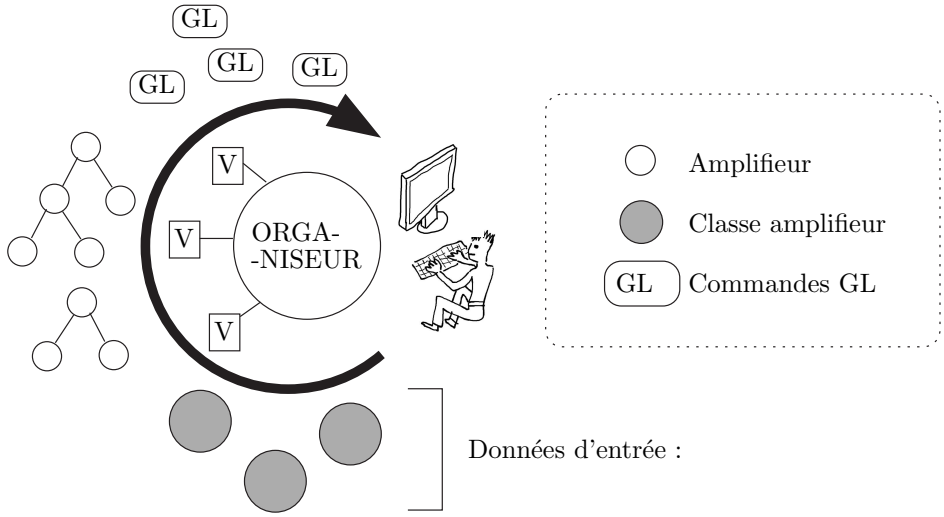


FIG. 5.7 – On reprend ici la figure 5.2 en rajoutant l'organiseur et ses visiteurs. L'organiseur s'occupe du bon ordonnancement des tâches nécessaires aux contraintes émanant de certaines fonctionnalités (tel que l'algorithme de détection d'occultation).

redessinateur : il redessine les informations contenu dans un arbre déjà généré ;

visualiseur : il affiche l'arbre de façon formelle (noeud et arête), ce qui est utile lors du debogage du modèle.

Le générateur est le visiteur le plus important : c'est lui qui, à mesure qu'il parcourt l'arbre d'évaluation, construit ce dernier. Cela peut paraître paradoxal de parcourir un arbre en même temps qu'on le construit. Pour que le générateur puisse visiter un arbre comme s'il était déjà construit, il génère lui-même les fils des amplifieurs qu'il veut visiter plus profondément. Ce faisant, le générateur se maintient dans l'illusion de l'existence d'un arbre infini. Cet algorithme est inspiré des méthodes de la *théorie des jeux* telle que le *branch and bound* [BCL+99].

L'organiseur

L'un des rôles fondamental de l'algorithme de rendu est de bien distribuer le temps de calcul et d'activer tel ou tel amplifieur selon des critères de performances particuliers (que nous verrons en détails plus tard). L'algorithme de rendu est pour cette raison appelé l'*organiseur* : il organise l'activité pour optimiser les performances.

On peut ici reprendre les notions qui ont été nécessaires à la définition de la *modélisation procédurale* et en particulier, discuter du découpage entre "programme" et "donnée" (cf. sous-section 3.1.1). L'organiseur, avec ces visiteurs, est la partie du programme qui se trouve à la frontière des données d'entrée. Les générateurs d'amplifieurs, les amplifieurs et enfin les commandes OpenGL produites sont les données utilisateur à leur différentes représentations au fil de l'évaluation (cf. figure 5.7).

Évaluation par morceau

Comme les amplifieurs génèrent d'autres amplifieurs, on pourrait croire qu'une fois l'évaluation commencée, il est impossible de l'arrêter. Ce comportement est celui des fonctions récursives : les appels s'enchaînent sans jamais rendre la main jusqu'à ce que tous les critères d'arrêt soient satisfaits (cf. figure 5.8). Dynamic Graph propose un comportement similaire à ces fonctions récursives mais permet un bien meilleur contrôle de leur exécution.

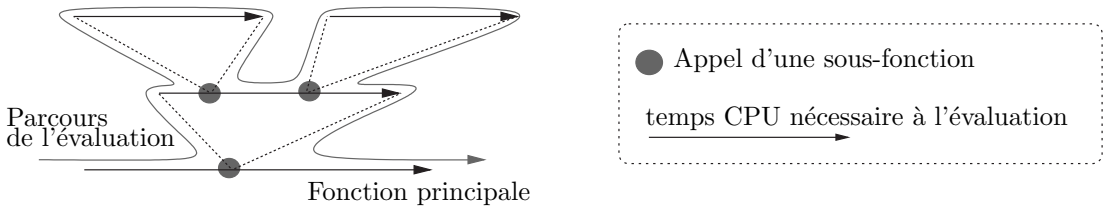


FIG. 5.8 – Dans Dynamic Graph, la construction de l'arbre d'évaluation est parfaitement contrôlée par l'organiseur. Contrairement à de simples fonctions récursives, Dynamic Graph permet de rendre la main à l'organiseur après la fin de l'évaluation de chaque amplifieur. De cette façon, l'organiseur a tous les pouvoirs pour diriger les opérations dans un ordre adéquat.

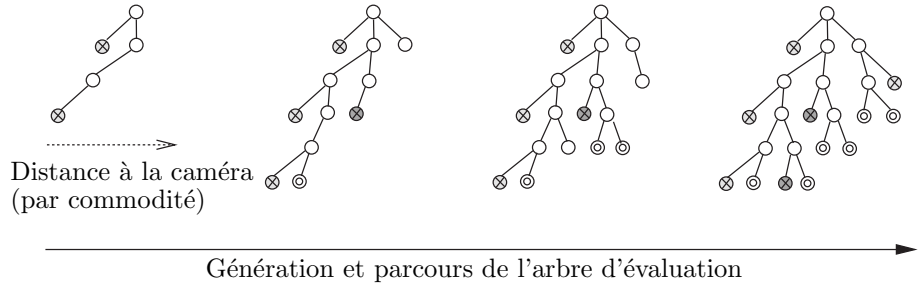


FIG. 5.9 – Un parcours trié en profondeur d'abord assure un rendu de l'avant vers l'arrière. Un noeud envoie ses commandes OpenGL dès que possible. Les feuilles visibles représentées par deux cercles imbriqués sont les amplifieurs les plus susceptibles de s'afficher. Sur ce schéma, par commodité de représentation, les amplifieurs à droite de l'arbre sont les plus distants de la caméra.

Ce contrôle est essentiel pour diverses raisons. Notamment, l'ordre d'affichage des amplifieurs est important : l'algorithme de visibilité qui sera expliqué dans la section 6.3 nécessite un rendu de l'avant vers l'arrière. On pourrait construire tout l'arbre d'évaluation d'abord pour le trier et l'afficher ensuite. Malheureusement, ce même algorithme nécessite d'afficher les amplifieurs *le plus tôt* durant la construction : le premier amplifieur est rendu alors que la construction vient à peine de commencer.

Pour résoudre ce problème, l'arbre est construit *approximativement* de l'avant vers l'arrière (d'autres contraintes peuvent peser sur la construction). Lorsqu'un amplifieur est prêt à être affiché, l'organiseur le place dans une liste d'attente qui rétablit un ordre plus strict. Pour réaliser cela, un amplifieur, lorsqu'il génère des amplifieurs fils, construit ces derniers sans les évaluer. L'organiseur choisira plus tard s'il décide de les évaluer ou s'il est temps de lancer l'affichage de certains amplifieurs pour cela.

Échange entre le modèle et le programme

L'*organiseur* doit être capable de communiquer avec les amplifieurs directement ou via un visiteur. Les *fonctions d'échange* s'occupent d'une partie de ce travail : elles fournissent notamment l'amplifieur axiome et assurent aussi certaines initialisations sans lesquelles le modèle ne peut pas être évalué. De plus, Dynamic Graph sait comment communiquer avec chaque générateur d'amplifieurs car ceux-ci héritent d'une classe virtuelle connue par l'organiseur.

Voici les fonctions que l'*organiseur* est susceptible d'appeler sur chaque *amplifieur* :
- initialisation ;
- fonctions diverses de navigation au sein d'un arbre (indiquer le fils, le père...) ;
- génération des amplifieurs fils ;

– affichage de la forme ;
– calcul de la visibilité ;
– calcul de la précision.

Dans certains cas, ces fonctions sont virtuelles pures, ce qui signifie que le créateur doit obligatoirement donner le code de cette fonction. Dans d'autres cas, ces fonctions sont virtuelles "tout court", ce qui signifie qu'un comportement par défaut est disponible. Pour des raisons diverses (performances, qualité, contrôle), le créateur peut choisir de personnaliser ces dernières pour un *amplifieur* particulier.

5.2.4 Animation

Dynamic Graph propose une évaluation de la forme visible à la volée. Afin de permettre l'optimisation par cohérence temporelle, des mémoires à plus ou moins long terme sont proposées au créateur. Un exemple simple et concret motive ce besoin.

Mémoire de l'information

À chaque pas de temps, un nouvel arbre d'évaluation est entièrement régénéré (i.e. la scène est entièrement reconstruite). En effet, la structure très dynamique de Dynamic Graph impose une évaluation à la volée. Utilisée naïvement, cette régénération supprime toute possibilité d'optimiser le programme par cohérence temporelle. Afin de ne pas passer son temps à recalculer la même fonction mille fois, Dynamic Graph permet une certaine mémoire des opérations effectuées aux pas de temps précédents.

Cette mémoire est principalement implémentée via un *tampon d'arbres* qui stocke les n derniers arbres d'évaluation. Chaque arbre d'évaluation a accès aux arbres plus anciens et peut donc retrouver des résultats effectués récemment. Plus généralement, Dynamic Graph permet d'utiliser toute une panoplie de mémoires à plus ou moins long terme et propose différentes fréquences de calculs selon le type de fonction.

Voyons maintenant au travers d'un exemple simple une utilisation possible de la mémoire et de la cohérence temporelle.

Exemple : la brise légère

L'animation est calculée durant la construction d'un objet. Les fonctions d'animation modifient certaines données présentes dans les amplifieurs. Elle incrémente une position : $x = x + v.\delta t$. Dynamic Graph propose des fonctionnalités avancées dans la gestion de la *persistance* des données. C'est un point essentiel du fonctionnement de Dynamic Graph, mais il est difficile à motiver de façon générale. Je vais donc l'introduire par un exemple.

On peut imaginer implémenter un mouvement par défaut de l'herbe, comme une brise légère[2]. Cette fonction d'animation est réalisée par la donnée d'une position du brin d'herbe au repos : la brise légère induit de légères oscillations autour de cette position. Compte tenu de la génération à la volée, cette méthode pose un problème a priori anodin : où stocker cette position au repos ?

Durée de vie d'une information

Voici deux façons extrêmes et inappropriées d'accéder à cette information :

permanent : mémoriser les positions au repos de tous les brins d'herbe induirait une description partielle au niveau le plus fin à éviter absolument ;

éphémère : une fonction pseudo-aléatoire peut générer à la volée et à chaque pas de temps une position au repos pour chaque brin d'herbe. Ceci implique, pour tous les brins d'herbe, un calcul reproduit à l'identique à chaque pas de temps. De plus, l'information de cette position

[2]La brise légère anime donc tous les brins d'herbe, mais elle laisse aux autres primitives de vent la possibilité d'influencer. Du point de vue de l'amplifieur, on peut parler d'animation interne (la brise légère) et d'animation externe (les autres primitives de vent).

au repos sera stocké dans chaque amplifieur à chaque pas de temps, résultant en un gaspillage de place mémoire.

Dans le cas de description au niveau le plus fin, le problème de la durée de vie ne se pose pas : toute l'information existe à tous les pas de temps. En revanche, dans le cas d'une génération à la volée, la gestion de la persistance mémoire est beaucoup plus délicate. Dans l'exemple cité plus haut, il serait souhaitable, dans l'idéal, de n'allouer la position au repos d'un brin d'herbe qu'une fois par *apparition* sur l'écran : si, d'une image à l'autre, le même brin est visible, la même information peut être utilisée. Dynamic Graph permet ce genre de mécanisme et propose une gamme de mémoires dont la durée de vie varie de *permanente* à *éphémère*. Ces fonctionnalités sont une première forme simpliste d'optimisation par *cohérence temporelle* et sont le fondement de toute utilisation plus complexe de ce dispositif.

5.3 Bilan intermédiaire

Dynamic Graph

Les présentations sont faites. Dynamic Graph propose une nouvelle représentation d'une forme consistant en une description du plus grossier au plus fin. Le langage descriptif de Dynamic Graph est le C++ dont les caractéristiques orientées-objet sont utilisées pour permettre la communication entre l'algorithme de rendu et l'objet lui-même.

Compte tenu de cette représentation originale, la modélisation et le rendu sont très spécifique. En conséquence, aucun outil existant nous a semblé propice à prendre en charge une part des difficultés. C'est pourquoi Dynamic Graph est présent quasiment tout au long du processus de création[3].

La suite

Dans le chapitre 6, la phase d'évaluation sera détaillée. Une attention particulière sera portée sur la façon dont sont traités les deux critères d'arrêt : la précision et la visibilité.

Le chapitre 7 aborde Dynamic Graph du point de vue de l'utilisateur. J'y discute des outils proposés pour faciliter le développement de modèles et des résultats obtenus.

[3]Sauf au stade final : l'utilisation des cartes graphiques, quelque soit l'application, est incontournable lorsque le temps-réel est visé.

Chapitre 6

Dynamic Graph : génération

Nous reprenons ici les concepts expliqués en 5 en les abordant sous un angle plus technique. Plus précisément, nous décrirons les algorithmes mis en œuvre pour manipuler l'arbre d'évaluation. Les critères d'arrêt de précision et de visibilité feront l'objet d'une attention particulière.

La structure de ce chapitre est la suivante :

Évaluation : nous décrirons dans cette section la partie la plus algorithmique de Dynamic Graph : la génération de l'arbre d'évaluation et les différentes techniques utilisées pour la rendre accessible et efficace ;

Précision : nous détaillerons ici la notion de précision qui sert notamment dans le critère d'arrêt de la génération de l'arbre d'évaluation. Nous introduirons aussi la *maturité* qui est une normalisation de la précision pour la rendre plus utilisable par le créateur lors d'une transformation continue d'un niveau de détail à l'autre ;

Visibilité : nous expliquerons l'algorithme de détection d'occultation ;

6.1 Évaluation

Nous verrons dans cette section les mécanismes complexes mis en œuvre durant la génération de l'arbre d'évaluation. Nous verrons ensuite comment Dynamic Graph assure une certaine connaissance des informations anciennes, notamment grâce à l'utilisation d'identifieurs. Nous décrirons enfin l'arbre permanent qui, contrairement à un arbre d'évaluation, contient des informations permanentes. Enfin, nous dresserons un bilan de la méthode d'évaluation et proposerons deux améliorations potentielles.

6.1.1 Génération de l'arbre d'évaluation

La génération de l'arbre d'évaluation est dirigée par l'organiseur, via un visiteur particulier : le générateur. L'évaluation d'un amplifieur est réalisée de façon séquentielle : elle ne se termine qu'après plusieurs phases d'exécution. Pour permettre cela, un dialogue a lieu entre le générateur et un amplifieur : le générateur donne des requêtes auxquelles l'amplifieur essaie de répondre.

Organiseur et visiteur

À chaque pas de temps, l'évaluation de la forme est effectuée par la création d'un arbre d'évaluation. La création de cet arbre est rythmée par l'organiseur. D'emblée, on distingue deux sortes d'événements temporels (cf. figure 6.1) :

évaluation : une évaluation commence au début du calcul de la forme pour une nouvelle observation et s'achève lorsque l'arbre d'évaluation est complètement généré ;

étapes de l'évaluation : l'organiseur cadence l'évaluation en étapes de calcul entre lesquelles il reprend la main.

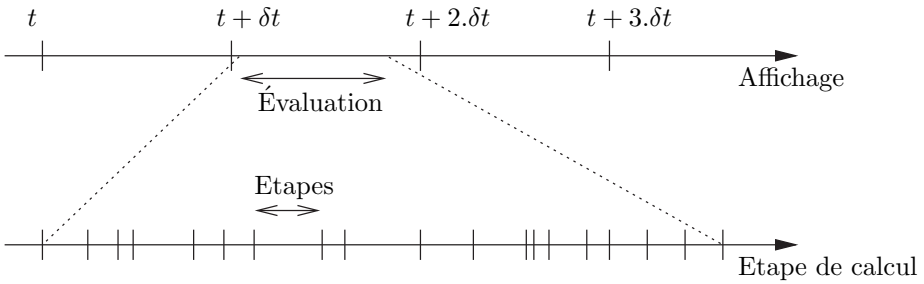


FIG. 6.1 – On appelle *pas de temps* les dates auxquelles une nouvelle observation est calculée. Chaque affichage est le fruit d'une évaluation, laquelle est divisée en *étapes* par l'organiseur.

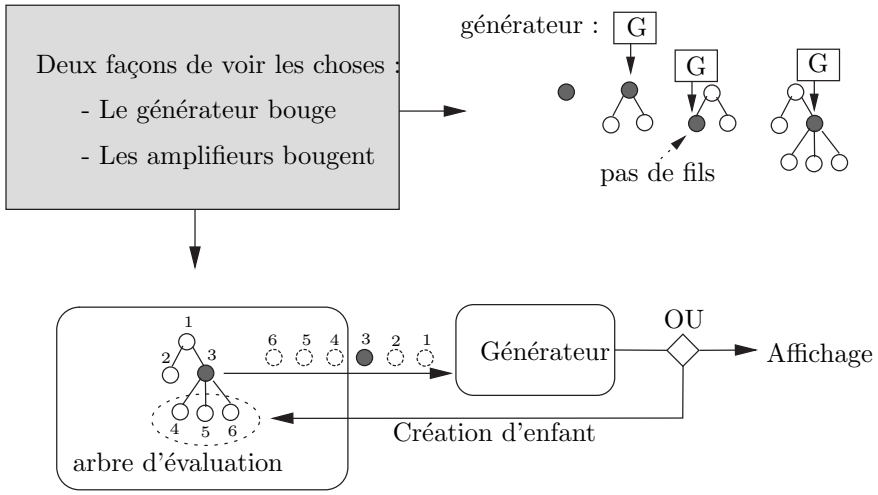


FIG. 6.2 – Il est plus facile de rendre compte de l'architecture en *pipe-line* lorsque ce sont les noeuds (les amplifieurs) qui bougent. Attention, ceci n'est qu'une vue de l'esprit. Concrètement, ni le visiteur ni les noeuds ne changent de place en mémoire.

Le *générateur* est le visiteur qui s'occupe de "développer" l'arbre d'évaluation jusqu'à sa complexion. On peut voir un visiteur est une entité qui se déplace sur un graphe pour appliquer un traitement à chaque noeud. Mais dans Dynamic Graph, il est plus intuitif de voir le phénomène "dual" : ce sont les noeuds du graphe qui vont visiter le visiteur (cf. figure 6.2).

Cette différence est importante pour décrire l'évaluation avec une architecture en *pipe-line*. Plutôt que de visualiser un arbre dont les noeuds ne bougent pas, il est préférable d'imaginer qu'un amplifieur, lors de son traitement, entame un sorte de pèlerinage qui le mènera éventuellement à la carte graphique.

Évaluation d'un amplifieur

Comme on l'a vu, un amplifieur, lorsqu'il est évalué, ajoute de la précision à la forme. L'*organiseur* s'occupe de lancer le calcul des amplifieurs via le visiteur générateur. Afin de bien contrôler l'évaluation, l'organiseur reprend la main entre deux évaluations. Ceci est essentiel pour assurer un certain parallélisme entre le processeur central et la carte graphique (nous verrons en section 6.3 pourquoi un tel parallélisme est nécessaire).

Comme toute action sur un ordinateur, l'évaluation d'un amplifieur n'est pas instantanée et il faut un certain nombre de cycles d'horloge pour que le processeur la termine. Lorsqu'un amplifieur est construit, il est dans son *état initial* : seule l'initialisation de l'évaluation a lieu. Lorsque

l'évaluation est terminée, l'amplifieur est dans son *état final* : il a terminé tous ses calculs.

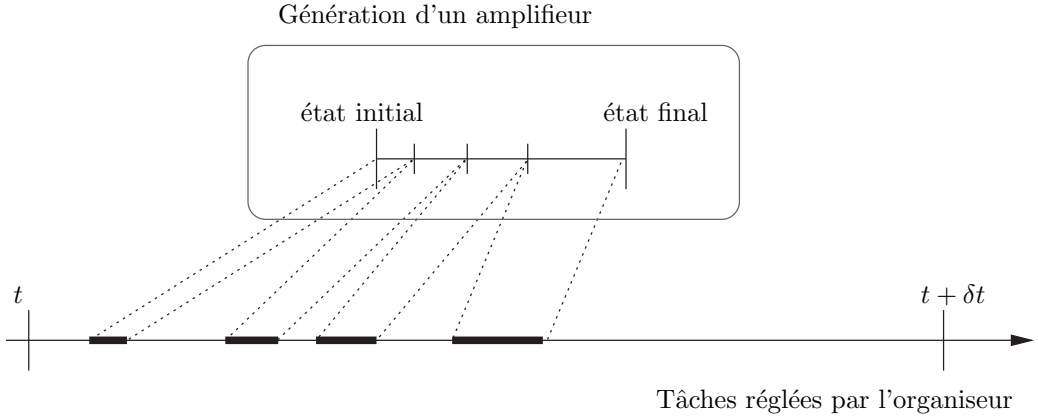


FIG. 6.3 – L'organiseur cadence le générateur pour que celui-ci évalue un amplifieur en plusieurs fois. En conséquence, l'organiseur reprend non seulement la main à la fin d'une évaluation, mais éventuellement plusieurs fois durant une même évaluation.

Évaluation séquentielle

Cette évaluation séquentielle est mise en œuvre pour permettre à un amplifieur d'attendre la fin de tâches parallèles sans bloquer le processeur courant. En effet, imaginons que l'évaluation d'un amplifieur requiert une certaine information qui n'est pas encore prête : l'amplifieur devient alors *bloquant*. Plutôt que d'attendre inutilement, l'amplifieur peut rendre la main à l'organiseur qui va aller "stimuler" d'autres amplifieurs *non-bloquants*.

En théorie, Dynamic Graph est donc parfaitement conçu pour tourner en parallèle sur plusieurs processeurs. Pourtant, ce n'est pas ce type de parallélisme qui est visé en premier. Les évaluations séquentielles sont initialement prévues pour assurer une bonne synchronisation entre le processeur principal et le *processeur graphique* (cf. section 6.3).

Les amplifieurs bloquants sont référencés dans une liste appelée la *salle d'attente* (cf. figure 6.4). Elle réinjecte les amplifieurs qui se sont débloqués vers le générateur. Une priorité est attribuée aux amplifieurs débloqués afin de libérer plus vite d'autres amplifieurs.

Si tous les amplifieurs sont bloquants, l'organiseur attend. Il est donc, par exemple, plus important de libérer un amplifieur dont on estime que les enfants sont nombreux. En effet, tout nouvel amplifieur est par défaut non bloquant. Plus le nombre d'amplifieurs non-bloquants est grand, plus l'organiseur peut organiser son travail efficacement. Remarquez que ce mécanisme permet de diminuer les durée de blocage, mais non de les supprimer.

Requêtes du visiteur

On peut formaliser l'échange complexe qui a lieu entre un visiteur et un amplifieur de la façon suivante : pour déclencher l'évaluation d'un amplifieur, le visiteur lance une requête. Lorsque l'amplifieur lui rend la main, le visiteur vérifie si la requête est satisfaite. Dans le cas contraire, il envoie cet amplifieur dans la *salle d'attente*.

Différentes sortes de requêtes sont possibles. La plus courante est la requête de *complexion* : "Dessine la forme ou génère des enfants si nécessaire". D'autres requêtes sont possibles. Par exemple, le générateur peut forcer certains calculs : "génère tes enfants quoiqu'il arrive".

On peut représenter les différentes fonctions qu'un amplifieur peut réaliser par un graphe de dépendance des résultats qu'il peut produire (cf. figure 6.5). Au cours d'une même observation, l'amplifieur se souvient des fonctions qu'il a déjà exécutées. De cette façon, lors d'une reprise des calculs, l'évaluation peut reprendre là où elle s'était arrêtée.

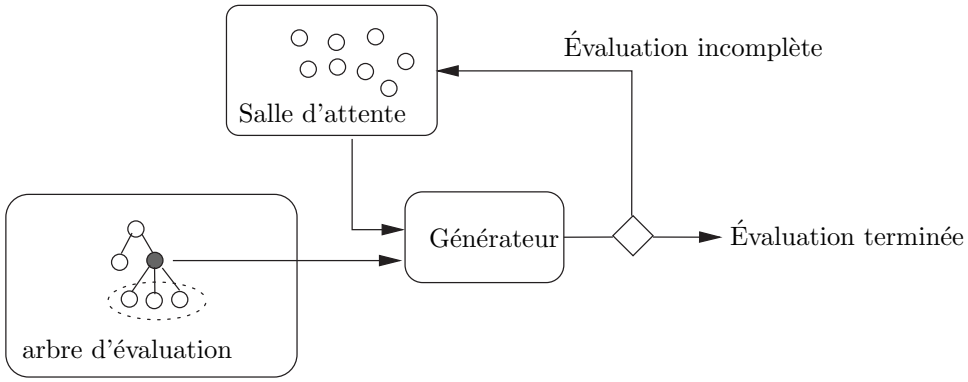


FIG. 6.4 – La salle d'attente récupère les références des amplifieurs dont l'évaluation est incomplète et attend le bon moment pour les remettre dans le circuit.

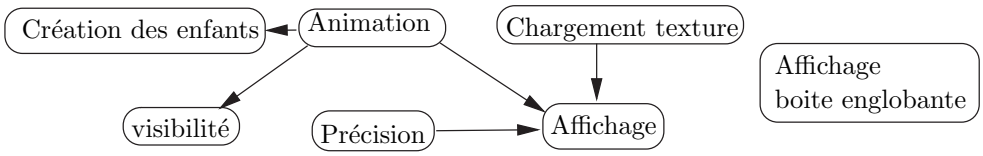


FIG. 6.5 – Au contact d'un visiteur, un amplifieur peut exécuter différentes fonctions. Celle-ci dépendent les unes des autres. Par exemple, l'affichage ne peut être effectué qu'après le calcul de l'animation.

Généralement, le générateur demande la requête de *complexion* à tous les amplifieurs. La complexion d'un amplifieur entraîne souvent la création de nouveaux amplifieurs qui doivent être évalués à leur tour. Lorsque tous les amplifieurs satisfont la requête de *complexion*, l'évaluation est terminée.

6.1.2 Mémoire et identifiant

Dans Dynamic Graph, chaque arbre d'évaluation connaît son ancêtre (notamment pour permettre l'optimisation par cohérence temporelle). Ceci est réalisé grâce au *tampon d'arbres* et aux *identifiants* attribués à chaque noeud. Chaque amplifieur a une date de naissance et de mort définissant sa durée de *vie*, notamment utilisée pour stocker des informations sur une longue durée.

Tampon d'arbres

Le tampon d'arbres stocke les n derniers arbres d'évaluations. n est fixé par le créateur ou l'application, selon les besoins de l'un ou de l'autre. Cette valeur peut être modifiée dynamiquement durant l'exécution du programme.

Son implémentation est basé sur l'utilisation de "tableaux rotatifs". Ces tableaux sont tout simplement des tableaux normaux dont l'accès est rendu cyclique en prenant les index modulo n.

Lorsque le programme débute, les premières cases du tableau, initialement vides, sont successivement remplies par un *arbre d'évaluation*. Lorsque le tableau est plein, les cases les plus vieilles sont libérées pour laisser place aux nouveaux arbres.

Chaque arbre d'évaluation est connecté à son ancêtre (l'arbre du pas de temps précédent). La connexion est réalisée dans les deux sens, ce qui signifie que chaque arbre est connecté à son ancêtre *et* à son successeur (s'ils existent).

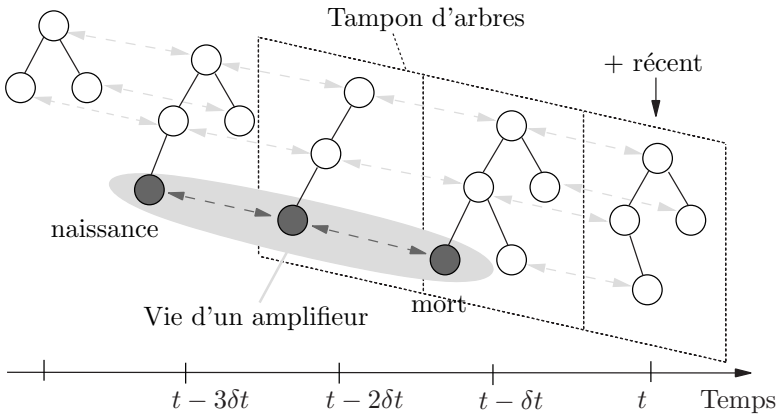


FIG. 6.6 – Chaque amplifieur de chaque arbre d'évaluation stocké dans le tampon d'arbres a connaissance de son successeur et de son ancêtre. Ceci lui permet d'avoir accès aux informations à des temps différents (généralement passés).

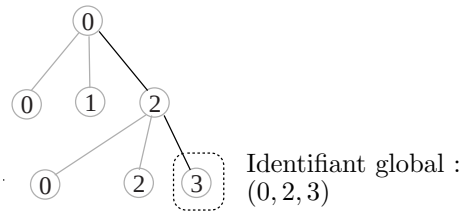


FIG. 6.7 – Un identifiant global est une liste d'identifiants locaux représentant le chemin de la racine à l'amplifieur identifié. Un identifiant local est unique parmi les identifiant des amplifieurs frères. Un identifiant global est unique parmi les identifiant des amplifieurs de l'arbre.

Identifiant global

Lorsque l'on dit qu'un arbre est connecté à son ancêtre, cela signifie que chacun de ses noeuds (i.e. amplifieurs) est connecté à son noeud ancêtre (cf. figure 6.6). Cette connexion est réalisée durant la création d'un nouvel arbre (c'est-à-dire durant une nouvelle évaluation). Le principe est simple : chaque noeud a un *identifiant global* le distinguant de tous les autres noeuds d'un *même* arbre. Lorsqu'un noeud est créé, il vérifie si un noeud du même identifiant existe dans l'arbre précédent : s'il existe, la connexion est créée. Ceci baigne chaque amplifieur dans une structure de liste doublement chaînée.

Une recherche d'identifiant global dans un arbre est une opération dont le coût n'est pas négligeable. Si l'on considère que l'accès à un fils est réalisé en coût constant, le coût moyen d'une recherche est proportionnel à la hauteur de l'arbre précédent. La réaliser lors de la création de chaque nouveau noeud grèverait considérablement les performances. L'utilisation d'*identifiant local* permet de palier ce problème.

Identifiant local

Un *identifiant local* permet de distinguer un amplifieur parmi ces frères. Concrètement, un identifiant local est un entier non signé. Un identifiant global est une liste d'identifiants locaux décrivant le chemin à suivre pour aboutir au noeud en question (cf. figure 6.7). Chaque noeud de l'arbre contient un conteneur vers ces noeuds fils. Ce conteneur est un conteneur associatif [D'A02] dont la clef (unique) est l'identifiant local. Accéder au fils $i \in \mathcal{N}$ revient donc à chercher l'existence de cette clef dans le conteneur associatif.

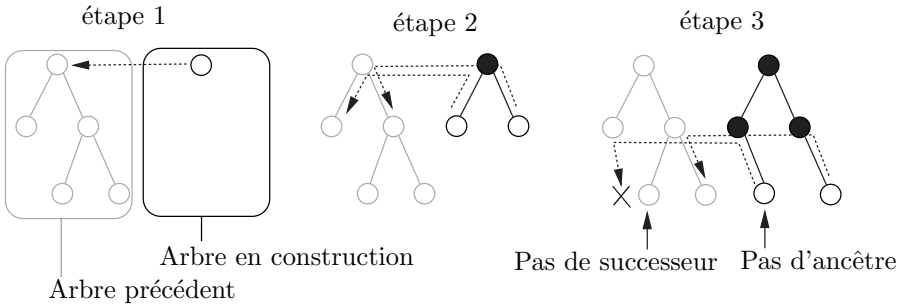


FIG. 6.8 – L'ancêtre d'un noeud est "le fils de l'ancêtre de son père". Cette formulation ne fait intervenir qu'une recherche d'identifiant local, c'est-à-dire la recherche d'un fils particulier parmi ses frères. Nous verrons en 7.2 que le coût de cette recherche dépend des choix que fait le créateur. Dans le meilleur des cas, elle est effectuée en coût constant.

Pour les raisons de performance identifiées plus haut, les identifiants locaux sont utilisés pour faire des recherches *relatives*. Pour comprendre intuitivement le mécanisme, on peut le comparer aux chemins de fichiers sous UNIX. Il est en effet souvent plus efficace d'accéder à un fichier par un chemin relatif comme :

"../source/toto.cpp"

que par un chemin absolu comme :

"/home/evasion/perbet/cpp/dg/model/source/toto.cpp"

La recherche de l'ancêtre est réalisée au cours de l'évaluation. Elle est initialisée avec la racine du nouvel arbre : celui-ci est spécifiquement connecté à la racine de l'arbre précédent par l'organiseur. Ensuite, chaque nouvel amplifieur a pour ancêtre (au sens temporel) "le fils de l'ancêtre de son père" (cf. figure 6.8). Comme les connexions sont réalisées au fur et à mesure, on est assuré que le père d'un amplifieur connaît déjà son ancêtre.

Destruction

À chaque pas de temps, lorsque le tampon d'arbres est plein, l'arbre le plus vieux est détruit. C'est un visiteur particulier qui s'occupe de ça : le *destructeur*. Celui-ci parcourt l'arbre des feuilles vers la racine en utilisant le *retour* d'un parcours en profondeur d'abord.

Il détruit tous les fils de l'amplifieur qu'il visite (la racine a un traitement particulier). De plus, si l'amplifieur n'a pas de successeur, il appelle sa fonction de mort. Si l'amplifieur à un successeur, il supprime proprement ce lien (dans les deux sens).

Vie et mort

La figure 6.6 montre la vie et la mort d'un amplifieur. La naissance est par définition une construction sans ancêtre et la mort une destruction sans successeur. Lors de la naissance (resp. de la mort) d'un amplifieur, un constructeur (resp. un destructeur) spécial est appelé.

Cette fonctionnalité permet d'initialiser de l'amplifieur. Par exemple, une texture peut être allouée lors de la naissance et désallouée lors de la mort. Plus généralement, la naissance et la vie d'un amplifieur permet de stocker n'importe quelle information *mortelle* pendant plusieurs pas de temps. L'information mortelle dure plus longtemps que les informations *éphémères* qui sont stockées dans un amplifieur : celles-ci sont accessibles n pas de temps ou n est la taille du tampon d'arbres En revanche, l'information *mortelle* dure moins longtemps que l'information *permanente* qui est décrite dans la sous-section suivante.

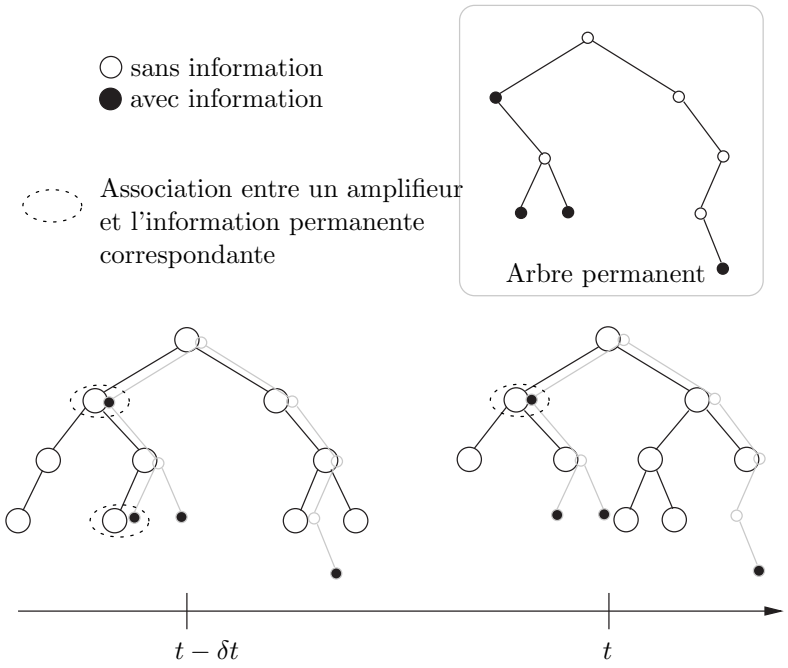


FIG. 6.9 – Cette figure montre l'arbre dynamique à deux pas de temps successif avec, en grisé, l'arbre permanent. L'arbre permanent ne change pas dans le temps. Les données permanentes sont associées à un amplifieur qui, à un instant donné, peut exister ou non. Certains noeuds de l'arbre permanent sont vides et ne servent qu'à atteindre des noeuds non vides.

6.1.3 Arbre permanent

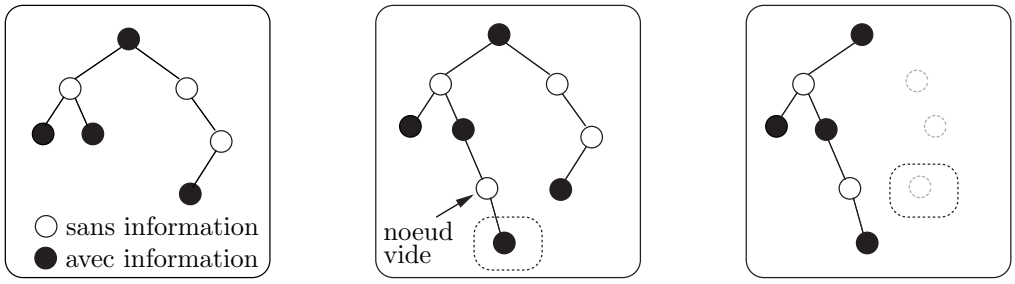
Les arbres d'évaluation, très dynamiques, sont complétés par un *arbre permanent*, représentant en quelque sorte la partie statique de l'évaluation. Il sert à stocker toutes les informations qui ne sont pas animés et ne nécessitent donc pas d'être recalculer à chaque pas de temps. Cet arbre ne dépend pas de l'observation et peut donc être très grand ; en conséquence, un effort particulier est fait pour maintenir sa taille la plus petite possible.

Description

Les informations stockées dans les noeuds sont très volatiles : leur durée de vie est celle de la taille du tampon d'arbres. Afin de donner la possibilité de stocker des informations de façon permanente, un arbre statique complète le tampon d'arbres dynamiques. Dans le cas de la prairie, par exemple, la courbure et la direction d'un brin d'herbe sont stockées dans l'arbre dynamique, alors que la couleur du brin d'herbe est stockée dans l'arbre permanent. Il permet d'associer à des amplifieurs particuliers des données supplémentaires. Il est appelé l'*arbre permanent*.

Les noeuds qui le composent, parce qu'ils représentent la partie permanente de l'information, sont appelés *amplifieurs permanents*. On peut représenter cet arbre comme un arbre partiellement superposable aux arbres d'évaluations mais beaucoup moins dense que ceux-ci (cf. figure 6.9).

Rappelons que l'arbre d'évaluation est potentiellement infini. Seuls les critères d'arrêt qu'engendre une *observation* permet de limiter son expansion. L'*arbre permanent* n'est a priori pas sujet à cette restriction : les informations qu'il contient sont *permanentes*, que l'amplifieur soit observé ou non. Elles ne peuvent être supprimées que par une intervention explicite (soit du créateur, soit de l'organiseur).



Arbre permanent avant les modifications

Opération d'*ajout* (et création de noeud vide)

Opération de *suppression* (En cascade)

FIG. 6.10 – L'arbre permanent, lors d'ajout ou de suppression d'informations, rajoute ou supprime des noeuds vides afin de garder la taille minimum requise pour accéder à toutes les informations permanentes. Remarquons qu'une conséquence de cela est que chaque feuille de l'arbre contient toujours de l'information.

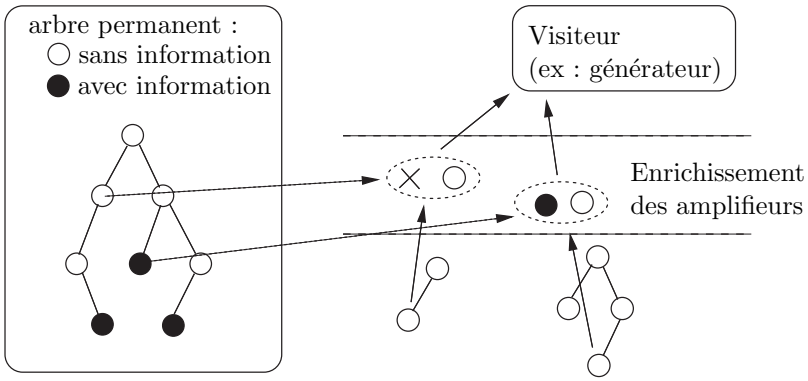


FIG. 6.11 – Avant d'être traité par le visiteur, l'amplifieur est enrichi de ses informations permanentes (si celles-ci existent).

Taille minimale

Si chaque amplifieur demande un homologue dans l'arbre permanent, la taille de celui-ci explose. Son utilisation doit donc rester exceptionnelle : seul quelques amplifieurs spéciaux peuvent l'utiliser.

De plus, afin de minimiser la mémoire occupée par l'arbre permanent, il est nécessaire de garder sa taille la plus petite possible. Pour cela, on s'assure que ce dernier est le plus petit graphe contenant tous les *amplifieurs permanents* (cf. figure 6.10). Concrètement, cela évite de garder en mémoire des informations concernant une forêt dans une zone désertique.

Amplifieur enrichi

Lorsqu'un visiteur parcourt l'arbre d'évaluation, il parcourt en même temps l'*arbre permanent*. Ainsi, lorsqu'un amplifieur s'apprête à être visité par le générateur, il est enrichi d'une référence vers ses éventuelles informations permanentes (cf. figure 6.11).

Lors de l'évaluation de l'amplifieur, le générateur met à disposition de celui-ci la zone de mémoire permanente. Elle est utilisée pour stocker des informations importantes. Dans un jeu, par exemple, cette mémoire peut servir à se souvenir qu'un objet particulier a été déposé ici par le joueur.

6.1.4 Bilan

La structure *dynamique* de Dynamic Graph permet de représenter des structures en perpétuelles transformations, parfaitement adéquate à une représentation par complexification. Néanmoins, on est en droit de se demander si cette structure n'est pas *trop* dynamique. Une nouvelle conception du tampon d'arbres *local* semblerait apporter une réponse. Enfin, nous verrons que le parcours de l'arbre d'évaluation dépend de l'échelle d'observation et limite ainsi les zooms très grands ; nous verrons comment la méthode de *congélation*, apporterait une solution partielle.

Dynamisme de la structure

La complexification entraîne une vision très dynamique de la forme affichée : celle-ci, en plus de son animation, s'adapte sans cesse aux mouvements de l'observateur. Pour représenter une forme aussi mouvante, il est nécessaire d'utiliser une structure qu'il est facile de mettre à jour. Dynamic Graph choisit une solution extrême : à chaque pas de temps, tout est régénéré. C'est un pari risqué : tout sur la *génération à la volée*.

La souplesse de la structure choisie est un avantage. La conception d'un algorithme traitant de scènes statiques a souvent quelques difficultés lorsqu'il faut passer à l'animation[1]. Dynamic Graph prend le parti inverse : la conception est entièrement pensée pour des structures animées. Lors de la modélisation de structure statique avec Dynamic Graph, il est toujours possible d'utiliser l'optimisation par cohérence temporelle. Nous verrons dans le chapitre 7 comment les informations *éphémères*, *mortelles* et *permanentes* peuvent être utilisée pour cela.

Régénération *vs.* modification

On peut tout de même reprocher à Dynamic Graph de faire beaucoup de calculs inutils lors de la modélisation de scènes statiques. Notamment, il devient inutile d'assurer la liaison entre un ancêtre et son successeur. Dynamic Graph propose généralement plus que ce que désire le créateur : c'est une façon d'être sûr que ce dernier ne se sente pas limité. En revanche, cela entraîne un coût supérieur à l'approche classiquement utilisée : la modification d'une structure statique.

Il est très délicat d'argumenter de façon générale des avantages et des inconvénients d'une structure statique (modifiée) ou d'une structure dynamique (générée à la volée). Ceci dépend notamment des fonctions que code le créateur au sein d'un générateur d'amplifieurs : certaines supportent très bien les modifications, d'autres moins. Dynamic Graph permet suffisamment de souplesse (grâce aux informations mortelles et permanentes) pour laisser le choix au créateur. Néanmoins, le comportement par défaut est celui de la régénération.

La modification est souvent plus efficace que la régénération lorsqu'il existe une forte similarité entre deux images (i.e. observation). Cette contrainte n'est pas toujours assurée. Même si l'observateur évolue très souvent de façon continue, il arrive fréquemment qu'il se déplace trop vite et que les opérations de modification deviennent ainsi trop coûteuses. La régénération est plus souple à ce niveau et permet généralement des mouvements de caméra moins contraints.

Travaux futurs : tampon d'arbres local

L'une des plus grandes limites de l'algorithme actuel est l'impossibilité d'assigner des *fréquences* de mise à jour différentes pour chaque générateur d'amplifieurs. Une telle fonctionnalité est similaire à celle de la plateforme de simulation OpenMask [Mar02] ou aux MTG [GC98]. Elle permettrait par exemple d'insérer des objets statiques avec une fréquence de mise à jour nulle. Une telle structure semble réalisable grâce à une conception plus locale du tampon d'arbres (cf. figure 6.12).

Cette conception est à vrai dire très séduisante. Elle permettrait aux visiteurs d'ignorer les noeuds n'ayant pas besoin de mise à jour. De plus, la recherche de l'ancêtre serait alors immédiate. Il serait très intéressant de coder cette solution et de la confronter à l'implémentation actuelle.

[1] C'est le cas de la majorité des algorithmes de simplification.

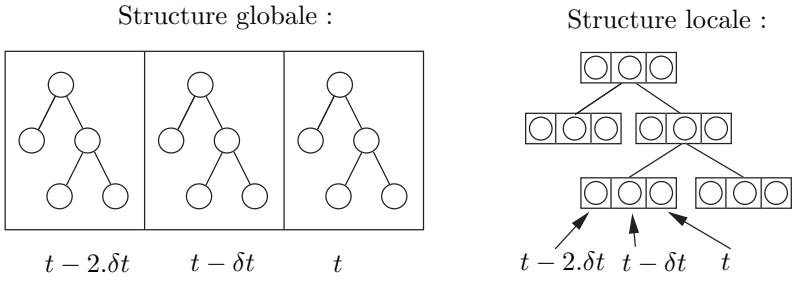


FIG. 6.12 – Pour que chaque noeud d'un graphe se souvienne de son ancêtre, deux structures sont possibles. La structure globale de Dynamic Graph stocke n arbres représentant les n dernières observations. La structure locale ne représente qu'un seul arbre, mais au sein de chaque noeud, n amplifieurs sont stockés (représentant les n dernières observations).

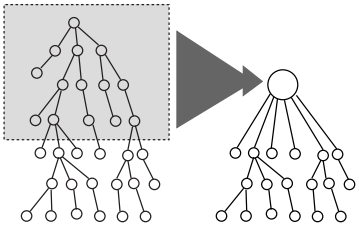


FIG. 6.13 – Lorsque le graphe devient trop haut, on pourrait "compresser" la partie haute.

Travaux futurs : la congélation

La régénération à chaque pas de temps de *tout* l'arbre d'évaluation entraîne un coût qui dépend de l'échelle d'observation : plus la scène est précise, plus l'arbre est grand et plus le temps de parcours augmente. Même si l'arbre à son niveau grossier est très mince, il faut au moins parcourir toute la hauteur. Ceci pose problème pour les *très* grandes variations d'échelle. En fait, nous n'avons pas pu observer ce phénomène car nous avons été limités par les problèmes d'imprécisions numériques. En effet, lorsque la caméra se rapproche trop d'un objet, la taille des détails observés et trop petite pour être représenté sur nombre constant de bits (typiquement 32 ou 64).

Ce n'est pas parce qu'un problème est caché par un autre que l'on peut l'ignorer, voyons les solutions qui s'offrent à nous. On peut envisager de congeler la partie grossière de l'arbre en une sorte "d'amplifieur axiome compressé" (cf. figure 6.13). Ce dernier pourrait, par exemple, représenter la partie congelée (forcément très lointaine) par un imposteur. Mais alors, l'animation des parties grossières du graphe seront ignorées. Cette approche ne pose pas de problème pour les zooms "allé simple" ou l'observateur se rapproche sans cesse sans jamais reculer beaucoup.

On peut imaginer que les générateurs d'amplifieurs sachent non seulement ajouter de la précision en créant des amplifieurs fils, mais aussi *enlever de la précision en créant leur amplifieur père*[2]. On touche ici à la limite intrinsèque de la *complexification* : celle-ci ajoute de l'information mais ne sait pas en enlever. Les langages descriptifs simples utilisés dans les méthodes de simplification autorisent la diminution de la précision. Mais dans le cas du langage descriptif complexe de Dynamic Graph, je vois mal comment une telle prouesse peut être réalisée...

[2]On est très proche, ici, du problème inverse, bien connu dans le domaine des fractales [HRSV01].

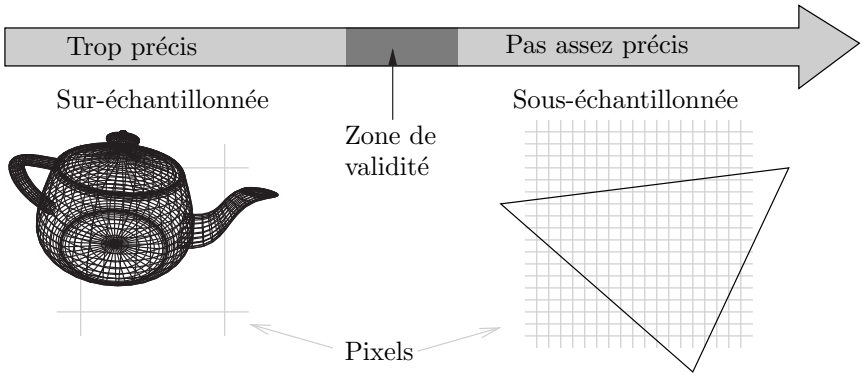


FIG. 6.14 – La précision est représentée par un réel. Une valeur positive signifie que la forme est sous-échantillonnée. Une valeur négative signifie que la forme est sur-échantillonnée. La valeur nulle représente l'échantillonnage idéal. Le but du jeu est ici de se rapprocher au maximum de la valeur nulle.

6.2 Précision

La fin de ce chapitre est dédié aux critères d'arrêt de la génération de l'arbre d'évaluation de Dynamic Graph : la précision et la visibilité. Dans cette section, la *précision* sera définie. Ensuite, nous ferons une première incursion du côté de la modélisation et nous présenterons la *maturité* comme une transformation pratique de la précision. Enfin, nous conclurons sur ces deux concepts avant de passer au second critère d'arrêt : la visibilité

6.2.1 Précision

Nous définirons tout d'abord la précision de manière formelle. Nous la reformulerons ensuite de façon approximative, efficace et suffisante aux besoins de Dynamic Graph. Nous verrons enfin comment la précision intervient lors de la génération de l'arbre d'évaluation.

Définition théorique

La *précision* est une notion très délicate à définir. Nous avons vu en 2.1 qu'un signal pouvait être *sur-échantillonné* ou bien *sous-échantillonné*. Dans Dynamic Graph, la *précision* est représentée par un nombre réel mesurant la qualité de l'échantillonnage (cf. figure 6.14).

Plus formellement, on peut considérer qu'un amplifieur $\mathcal{A}$ affiché envoie un ensemble de primitives graphiques :

$$\mathcal{P}_{\mathcal{A}} = \{primitives\ graphiques\ envoyées\ par\ \mathcal{A}\}$$

Pour chaque primitive graphique $p \in \mathcal{P}_{\mathcal{A}}$, on peut déterminer la surface de l'écran $s(p)$ qu'elle va remplir en nombre de pixels. Cette valeur peut être infiniment grande (on ne considère pas le *view frustrum culling*). Elle peut être infiniment petite (pour un triangle infiniment petit par exemple).

Soit S_{ideale} la taille de triangle considérée comme parfaite (cf. [WW03]). Cette valeur dépend notamment de l'application visée. Pour fixer les idées, pour une *qualité* maximum, on a $S_{ideale} \simeq 10 \times 10\, pixels$.

Pour chaque primitive graphique $p \in \mathcal{P}_{\mathcal{A}}$, on peut alors exprimer l'aire "normalisée" : $S(p) = \frac{s(p)}{S_{ideale}}$. Cette fonction vaut 1 lorsque la primitive a la taille idéale.

On peut calculer l'*aire normalisée moyenne* de toutes les primitives graphiques envoyées à la

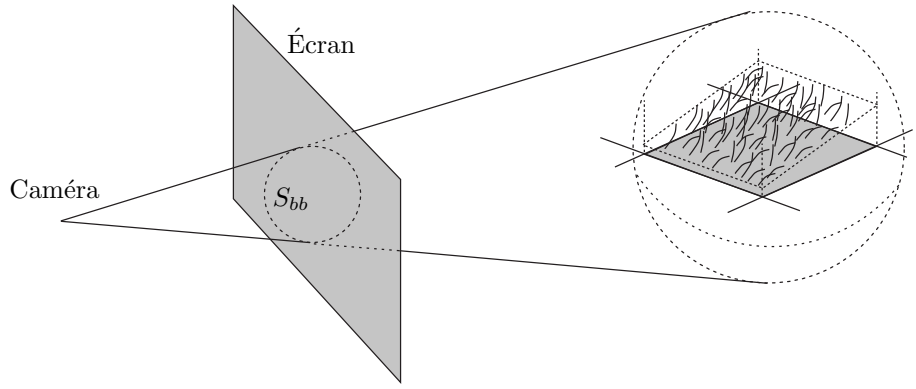


FIG. 6.15 – L'aire projetée S_{bb} de la boîte englobante d'un objet donne une idée approximative de la taille qu'il occupe sur l'écran. En diminuant le coefficient $C_{créateur}$, on peut prendre en compte le fait que l'objet a beaucoup de polygones (ou qu'il est texturé avec une image de grande résolution). Dans le cas de l'amplifieur 2D de la prairie, la valeur $C_{créateur}$ est petite car le polygone est texturé (ce qui, visuellement, est équivalent à une subdivision plus fine du polygone).

carte graphique par un amplifieur :

$$S_m(\mathcal{A}) = \frac{1}{card(\mathcal{P}_\mathcal{A})} \sum_{p \in \mathcal{P}_\mathcal{A}} S(p)$$

S_m est à valeur dans $\Re^+$.

Le but de Dynamic Graph est donc de générer des amplifieurs tels que $S_m(\mathcal{A}) = 1$. Afin de bien résoudre cette équation, il est préférable de mieux conditionner l'équation en prenant son logarithme : $log_2(S_m(\mathcal{A})) = 0$.

La définition théorique de la précision d'un amplifieur est :

$$precision(\mathcal{A}) = log_2(S_m(\mathcal{A}))$$

Remarquons que lorsque les primitives sont deux fois plus grandes que la taille idéale, on a :
$precision(\mathcal{A}) = 1$
Lorsqu'elles sont deux fois plus petites, on a :
$precision(\mathcal{A}) = -1$

En pratique

En pratique, le calcul précis de la surface de chaque primitive est inutilement coûteux. En effet, l'expérience montre qu'une grossière estimation est largement suffisante pour obtenir des résultats satisfaisants.

Ainsi, le calcul de $S_m(\mathcal{A})$ est ramené à la formule suivante :

$$S_m(\mathcal{A}) = \frac{C_{créateur}.S_{bb}}{S_{ideale}}$$

S_{bb} est la surface projetée d'une boîte englobante sphérique de l'amplifieur. Comme nous le verrons dans le chapitre suivant, la boîte englobante est spécifiée par le créateur lors de la modélisation. $C_{créateur} \in [0, 1]$ est un coefficient de calibration qui reflète la discrétisation de l'amplifieur. Par exemple, un maillage sphérique très échantillonné aura un coefficient $C_{créateur}$ inférieur à 1. À l'opposé, un maillage sphérique peu échantillonné aura un coefficient $C_{créateur}$ proche de 1.

Ces deux coefficients sont donnés par le créateur pour chaque *générateur d'amplifieurs* (cf. figure 6.15). C'est donc ce dernier qui spécifie entièrement la fonction de précision. Par expérience, la

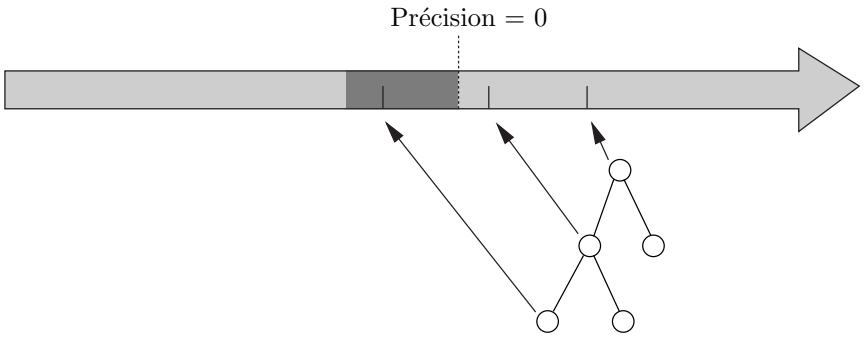


FIG. 6.16 – Les amplifieurs génèrent des fils tant qu'ils n'atteignent pas la première précision négative.

gène occasionnée par la détermination d'une boîte englobante sphérique et du coefficient $C_{créateur}$ sont minimes (en tout cas négligeable par rapport au reste du modèle).

Critère d'arrêt

La fonction $precision(\mathcal{A})$ est celle utilisée dans le critère d'arrêt de précision de chaque amplifieur $\mathcal{A}$. Le test de précision est très simple : si la précision est négative, cela signifie qu'elle est suffisante (cf. figure 6.16).

Remarquons, qu'une simple variation de S_{ideale} permet de diminuer la précision (et donc la qualité) tout en augmentant les performances (moins de géométrie est générée). Dynamic Graph propose donc un contrôle total sur le coefficient $Qualité \times Rapidité$.

Des contrôles plus fins sont permis : la valeur de S_{ideale} peut varier par *générateur d'amplifieurs*. Ceci permet la calibration relative de modèles d'origine différente lors de leur mise en commun.

6.2.2 Transition et maturité

Nous verrons ici comment la précision est transformée en une valeur utilisable par le créateur : la maturité. La maturité est une sorte de normalisation de la précision idéale pour interpoler continûment l'ajout de détail d'un amplifieur. Nous verrons deux versions de la maturité : synchrone et asynchrone.

La précision : une variable continue

Le rôle que joue la précision dans les critères d'arrêt est discret : l'amplifieur génère des fils ou non. La nature continue de la précision est utilisée pour assurer le passage continu d'un amplifieur à ses amplifieurs fils. Pour reprendre le vocabulaire utilisé dans le chapitre 4, les transitions entre niveaux de détail doivent être continues.

Dynamic Graph accorde une grande importance à cette continuité. De nombreux critères plus ou moins importants sont garants du "bon" déroulement d'une transition. Nous plaçons la continuité parmi les plus importants : le moindre *popping* visuel détériore très notablement la qualité du rendu.

Soyons clairs, Dynamic Graph n'assure pas automatiquement ces transitions. Il propose simplement au créateur un coefficient d'interpolation qui permettra à ce dernier d'ajouter la précision continûment. Ce coefficient est appelé *maturité*. Lorsque la maturité est nulle, les détails codés par l'amplifieur sont indiscernables. Lorsque la maturité vaut un, l'amplifieur a ajouté toute la précision dont il était capable.

Remarquons qu'il est essentiel que lors de l'apparition d'un amplifieur fils, la maturité de ce dernier soit nulle. Dans le cas contraire, on distinguera un *pop* de la forme car celle-ci sera subitement enrichie.

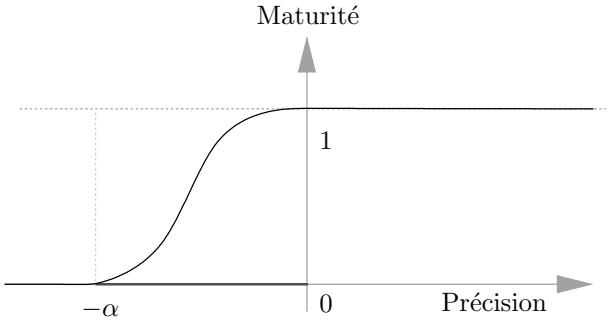


FIG. 6.17 – Avec la maturité synchrone, un amplifieur passe d'une maturité 0 à 1 par une simple fonction croissante sur l'*intervalle d'amplification* $[-\alpha, 0]$. Lorsque l'observateur se rapproche, la précision augmente et la maturité "glisse" vers sa valeur maximum.

Maturité synchrone

La première définition de la maturité qui vient à l'esprit est une simple transformation de la précision $maturite = f_{maturite}(precision)$ (cf. figure 6.17). Cette fonction est principalement définie par α déterminant la taille de l'*intervalle d'amplification*.

Cette taille dépend de l'amplifieur et de la précision qu'il couvre. C'est au créateur de régler le coefficient α (par calibrage expérimental). Celui-ci doit-être suffisamment petit sans quoi un amplifieur peut apparaître avec une transition non-nulle (cf. figure 6.18). Ceci entraîne des discontinuités de l'affichage à proscrire absolument. Dans le cas de la prairie, on aura par exemple la représentation 2.5D qui apparaîtra soudainement sur la représentation 2D sans "pousser" continûment.

Cette version de la maturité est appelée *maturité synchrone* car pour une position de la caméra correspond une valeur unique de la maturité pour chaque amplifieur. Le principal problème lié à cette maturité est qu'elle autorise un amplifieur à rester dans des positions intermédiaires (comme $maturite = 0.5$). Or, dans certains cas (cf. sous-section 4.1.3), l'esthétique des positions intermédiaires peut être inférieure à celle des positions de repos ($maturite = 0.0$ ou $maturite = 1.0$). Nous en avons vu un exemple pour les états de transition entre 2D et 2.5D dans le cas de prairie. Pour remédier à ce problème, Dynamic Graph propose des *transitions asynchrones*.

Maturité asynchrone

La maturité asynchrone dépend du temps : une fois déclenchée (quelque soit le sens), elle se termine forcément. Pour cela, il faut définir deux valeurs de précision p_{min} et p_{max} telles que :

$$p_{min} < p_{max}, p_{min} \in [-\alpha, 0], p_{max} \in [-\alpha, 0]$$

Ensuite, de même que pour les prairies, on déclenche la montée si $p > p_{max}$ et la descente si $p < p_{min}$. La montée est réalisée avec une certaine vitesse $v_{maturite}$. Du coup, à chaque pas de temps, on a :

$$maturite(t) = maturite(t - \delta t) \pm v_{maturite}.\delta t$$

On s'assure toujours que $maturite(t) \in [0, 1]$.

Le principal problème avec cette technique est que la maturité a de "l'inertie". En conséquence, la forme se modifie toujours même lorsque la caméra ne bouge pas. Ceci n'a pas d'importance pour la plupart des objets animés car cela passe inaperçu : l'animation des objets "cache" l'animation

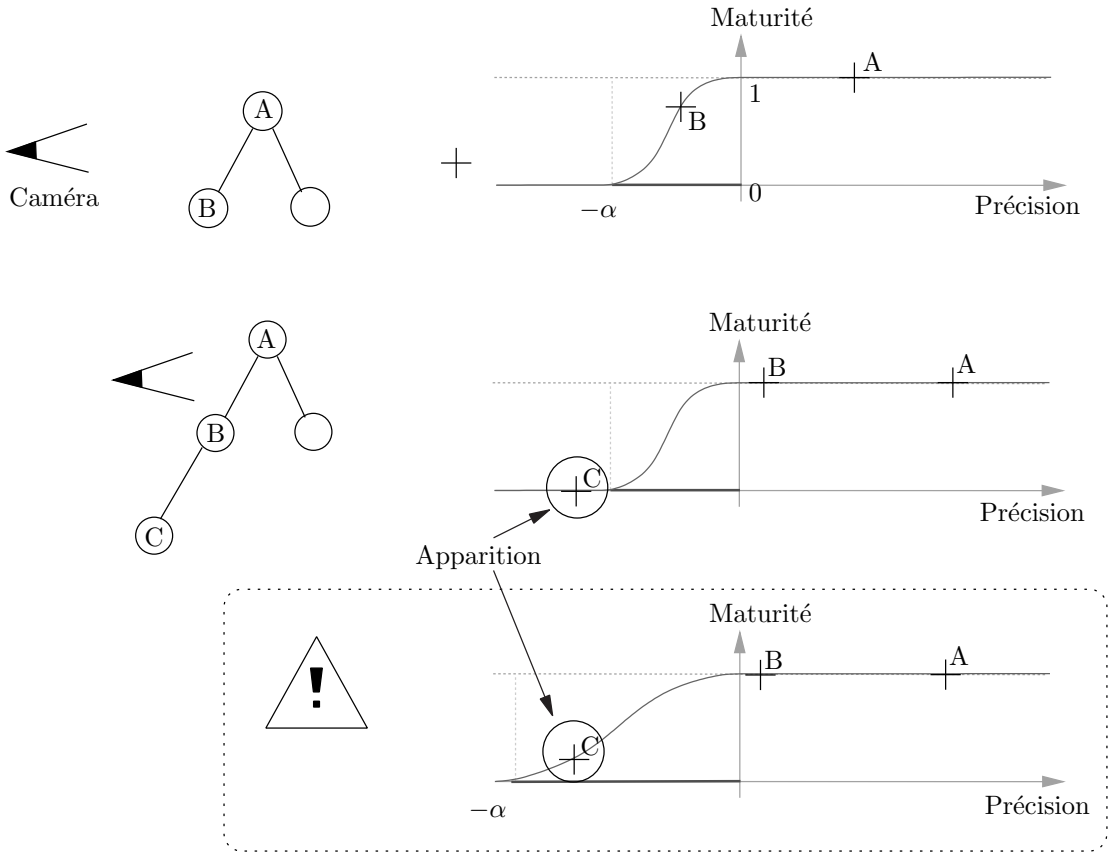


FIG. 6.18 – Lorsque la caméra se rapproche, un nouvel amplifieur naît. Si tout se passe bien, la précision de celui-ci sera suffisamment faible pour que que sa maturité soit nulle. En conséquent, l'ajout de détail est initialement indiscernable. Mais si α est mal réglé par le créateur, un amplifieur peut naître avec une maturité non nulle, ce qui engendrera un ajout soudain de précision.

due à la maturité. En revanche, pour les objets non-animés, cela nuit indéniablement à la qualité de l'effet visuel.

L'astuce ici consiste à faire varier la vitesse $v_{maturite}$ en fonction des mouvements de la caméra (ou tout autre action "diversive"). Dès que la caméra bouge "sensiblement", la vitesse $v_{maturite}$ augmente. En fait, pour détecter les mouvements de la caméra, on peut utiliser la dérivée de la précision :

$$\frac{\delta precision(t)}{dt} = \frac{precision(t) - precision(t - \delta t)}{\delta t}.$$

On a alors (en gardant les seuils de déclenchement p_{min} et p_{max}) :

$$maturite(t) = maturite(t - \delta t) + f_{vitesse}(precision(t) - precision(t - \delta t)).\delta t$$

La fonction $f_{vitesse}$ détermine comment on doit interpréter les variations de la précision (cf. figure 6.19). Cette méthodet fonctionne mais donnent des résultats médiocres pour certains types d'observation "brutales". Par exemple, si le marcheur court très vite vers la prairie, les niveaux de détails n'auront pas le temps de s'adapter. Pour éviter ce genre de problème, un mécanisme de détection de discontinuité est mis en place : si la précision varie plus qu'un certain seuil, on revient l'espace d'un pas de temps vers une maturité synchrone.

FIG. 6.19 – Deux exemples de fonction $f_{vitesse}$. À gauche, la fonction permet deux vitesses : lente pour les faibles variations et rapide pour les grandes variations. À droite, c'est le même principe en plus continu.

6.2.3 Bilan

Nous verrons tout d'abord dans ce bilan comment la maturité pourrait être étendue pour assurer la prise en charge de contraintes plus complexes. Ensuite, nous dresserons un bilan de la précision et conclurons en affichant Dynamic Graph comme un véritable laboratoire perceptuel dédié à la multi-échelle.

Maturité : une variable réactive

Peut être qu'il existe une fonction de maturité valide pour tous les types d'observations. Mais peut-être sommes-nous là aussi victimes de la loi du *cas particulier*. En fait, il semblerait que chaque générateur d'amplifieurs nécessite une fonction personnalisée qui dépend de la manière dont on l'observe, de son type d'animation.

On peut considérer la maturité comme une *variable réactive*, c'est-à-dire une variable dont la valeur dépend des valeurs précédentes. Dynamic Graph propose différentes fonctions par défaut que le créateur peut choisir selon le *générateur d'amplifieurs* modélisé.

Mais surtout, la connaissance des pas de temps précédents permet au créateur d'écrire ses propres variables réactives. On peut imaginer, par exemple, des amplifieurs dont la maturité doit fuir certaines valeurs $v_i \in [0, 1]$ pour lesquelles le résultat n'est graphiquement pas de bonne qualité.

La fonction de maturité pourrait à la limite être conçue comme le résultat d'une "simulation physique" complexe dont les forces tendraient à minimiser les désagréments de l'affichage. La connaissance des valeurs au pas de temps précédent font de Dynamic Graph l'outil idéal pour réaliser ce genre de simulation dans un monde procédural généré à la volée.

Plus sur la précision

La précision utilisée dans Dynamic Graph joue parfaitement son rôle : via une simple calibration visuelle pour déterminer un bon $C_{créateur}$, un critère d'arrêt automatique libère le créateur de tâches fastidieuses. La maturité met à la disposition de ce dernier un moyen simple de réaliser des interpolations durant l'ajout de détails et peut être configurée de différentes manières (synchrone, asynchrone).

Remarquons aussi que la précision caractérise ici tout l'amplifieur. Pourtant, on peut très bien imaginer un amplifieur complexe au sein duquel la précision varie de manière non-uniforme. Par exemple, un amplifieur peut représenter un terrain multi-résolution, auquel cas sa précision varie entre les parties proches et l'horizon. Plus généralement, Dynamic Graph implique une décomposition hiérarchique des amplifieurs et ne propose aucune facilité pour les structures continues telles qu'un tronc d'arbre (1D), un terrain (2D) ou du brouillard (3D). Nous discuterons de ces limitations dans le chapitre suivant et montrerons comment Dynamic Graph supporte aussi (mais moins bien) ces structures.

Dynamic Graph : un "laboratoire perceptuel"

Dynamic Graph, dans sa forme actuelle, ne résout pas les nombreux problèmes perceptuels posés par la synthèse d'image et la multi-échelle en particulier. Certes, le travail effectué sur la maturité apporte quelques éléments de réponse et évite notamment les *états intermédiaires*. Néanmoins, de nombreux critères n'ont pas été pris en compte : la vitesse des objets, leur importances dans la scène, l'influence du *frame-rate*, l'excentricité... Pour plus de précision sur ces critères, veuillez consulter l'excellent mémoire de thèse réalisé par Martin Reddy : *Perceptually Modulated Level of Detail for Virtual Environnments* dont certains éléments ont été repris dans le livre *Level of Detail for 3D graphics* [LRC+02].

J'aimerais présenter ici Dynamic Graph comme un "laboratoire perceptuel". En effet, Dynamic Graph offre un champ d'étude perceptuel très complet et offre une très grande souplesse. En dehors du coeur algorithmique (les arbres d'évaluations), la quasi-totalité des fonctionnalités proposées par défaut sont redéfinissables en C++ lors de la création d'un modèle. Les vitesses et les accélérations des objets sont facilement calculables grâce au tampon d'arbres. La sémantique très forte apportée par la modélisation procédurale permet l'utilisation de critère cognitif. Enfin, les possibilités d'animation et les grandes variations d'échelle permettent la création de cadres d'expérimentations très variés.

6.3 Visibilité

La visibilité est, avec la précision, un critère d'arrêt essentiel dans la génération de l'arbre d'évaluation de Dynamic Graph. Nous exposerons tout d'abord le contexte scientifique puis un algorithme original de détection d'objet caché. Nous présenterons ensuite l'algorithme puis détaillerons la plus grande difficulté de sa mise en oeuvre : assurer un parallélisme efficace entre la carte graphique et le processeur central. Nous conclurons ensuite en insistant sur le rôle essentiel de la carte graphique dans cet algorithme.

6.3.1 Présentation

Avant de décrire le contexte scientifique, je justifierai la présence de cette méthode de détection d'occultation dans ce mémoire portant sur la multi-échelle, et non la visibilité. Notamment, nous verrons comment un triple concours de circonstances favorables "m'a interdit" de ne pas implémenter cet algorithme. Ensuite, nous comparerons rapidement ce travail à l'existant par l'étude bibliographique [COCSD03]. Nous énumérerons ensuite les différents travaux portants sur la visibilité *et* la multi-échelle.

Hors-sujet ?

La présence de cet algorithme dans cette thèse peut laisser perplexe. En effet, pourquoi parler de visibilité dans un travail sur la multi-échelle ? Au premier regard, cela peut paraître tout simplement hors-sujet. Pourtant, la visibilité et la multi-échelle sont toutes deux motivées par un même but : «n'afficher que ce que l'on voit».

Néanmoins, bien que mues par le même désir, les techniques mises en œuvres dans ces deux méthodes sont radicalement différentes. En fait, l'implémentation de l'algorithme de détection d'occultation de Dynamic Graph est le fruit d'un concours de trois circonstances favorables. De ce coup de chance est né un algorithme simple et efficace permettant un calcul de visibilité temps-réel sur des scènes complexes et animées.

Un coup de chance

La première circonstance favorable est le besoin. La principale raison pour laquelle Dynamic Graph nécessite un algorithme de détection d'occultation est son efficacité. Sur certaines scènes

très complexes (comme le cube de Sirpienski), 80% des objets dessinés par un algorithme de base n'étaient pas visibles.

La seconde circonstance est l'apparition d'une nouvelle extension sur les nouvelles cartes graphiques : GL_ARB_occlusion_query. Cette extension s'est démocratisée en 2002 et elle est désormais sur toutes les cartes graphiques. Elle permet de compter le nombre de pixels qui atteignent l'écran[3]. L'une des applications immédiates de cette extension est un calcul de *détection d'occultation* (ou *calcul de visibilité*) : lorsque le nombre de pixels allumés par un objet vaut zéro, ce dernier n'est pas visible. On peut donc l'ignorer puisqu'il ne contribuera pas à l'image finale.

La troisième circonstance favorable est l'utilisation de la modélisation procédurale multi-échelle. En effet, tout était prêt, dans Dynamic Graph, à accueillir l'algorithme de visibilité décrit plus loin. Les boîtes englobantes hiérarchiques étaient déjà utilisées pour le calcul de précision et le *view frustrum culling*. L'absence de précalcul, imposée par la modélisation procédurale, montrait presque la solution du doigt... Je ne pouvais pas ne pas implémenter cet algorithme.

Contexte

Pour un état de l'art exhaustif sur le problème de la visibilité, je vous renvoie à la thèse de Frédo Durand [Dur99]. Plus récemment, une excellente étude bibliographique [COCSD03] recense de très nombreuses méthodes. Surtout, elle propose une taxonomie très précise permettant une rapide et précise description des différents algorithmes de visibilité (section 2 : Classification).

Selon cette taxonomie, voici une description de l'algorithme implémenté dans Dynamic Graph :

point based **:** le calcul de la visibilité est effectué pour chaque point de vue ;

image precision **:** le calcul de visibilité est réalisé dans l'espace image ;

generic scenes **:** l'algorithme n'est pas limité à un type de scène particulier.

[COCSD03] propose d'autres critères descriptifs :

conservative **:** Dynamic Graph affiche plus que le minimum requis ;

tightness of approximation **:** les primitives graphiques sont regroupées par amplifieur : aucune occultation au sein d'un même amplifieur n'est détectée. Les amplifieurs interpénétrants peuvent aussi détériorer la qualité du résultat ;

all **:** Tous les objets sont pris en compte pour tester la visibilité ;

generic occluders **:** dans Dynamic Graph, l'*occluder* est tout simplement le *Z-buffer* qui se rappelle de la plus petite distance à la caméra des objets déjà dessinés. Cela signifie que l'*occluder* évolue au cours du calcul de visibilité : plus on dessine d'objet, plus le *Z-buffer* se "remplie" et plus les distances minimales des objets à la caméra sont petites. Remarquons que l'affichage est réalisé *en même temps* que le calcul de visibilité.

3D **:** l'algorithme est entièrement pensé en trois dimensions ;

special hardware requirement **:** la détection d'occultation utilise massivement l'extension GL_ARB_occlusion_query ;

no precomputation **:** Aucun précalcul n'est requis ;

dynamic scenes **:** les scènes traitées sont animées.

Visibilité et multi-échelle

Un dernier critère manque peut-être à l'appel : l'utilisation de méthodes multi-échelles. En effet, certains travaux se démarquent par l'utilisation simultanée d'un algorithme de visibilité et d'une méthode multi-échelle. À notre connaissance, la totalité d'entre eux est basée sur des méthodes de simplification hiérarchique [YSM03, ESSS01, GLY+03, ISGM02, ACW+99]. Les précalculs effectués limitent ou interdisent l'animation. En effet, lorsque le modèle change trop, il faut remettre à jour la hiérarchie.

[3] Plus précisément, GL_ARB_occlusion_query compte le nombre de fragments franchissant le *depth test* et le *stencil test*.

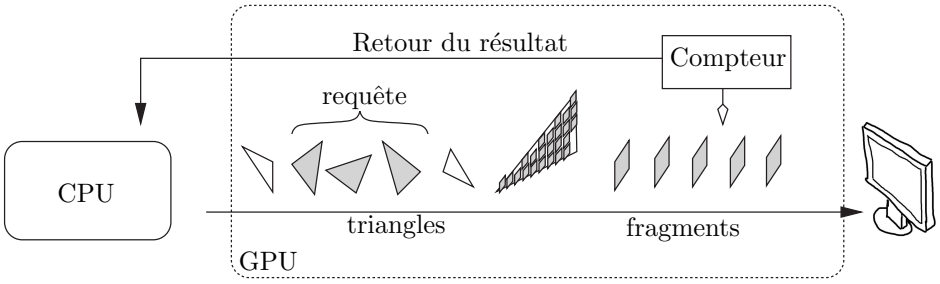


FIG. 6.20 – Le CPU envoie au GPU des primitives graphiques qui sont décomposées en fragment. Un compteur compte ces derniers et renvoie le résultat au CPU. Le problème de l'asynchronicité souhaitée vient du fait que le CPU rester bloquer pendant que la requête traverse toute le pipe-line de la carte graphique et lui revienne.

Dans la catégorie des algorithmes alliant visibilité et multi-échelle, mon travail est le premier utilisant une approche procédurale par complexification. Par conséquent, compte tenu de l'aspect dynamique de cette approche, il est aussi le premier à permettre une animation *massive*, en ce sens que tous les niveaux de la hiérarchie peuvent être animés.

6.3.2 Algorithme

Sont expliquées ici les grandes lignes de cet algorithme de détection de face cachée. Celui-ci est basé sur la fonction GL_ARB_occlusion_query permettant de compter les pixels allumés par la boite englobante de la forme contenu dans un amplifieur. Avec un rendu de l'avant vers l'arrière, il est alors possible de déterminer quel objet est au moins partiellement visible. Ensuite, nous verrons pourquoi et comment limiter le nombre d'appels à la fonction GL_ARB_occlusion_query.

Calcul du nombre de pixels allumés

La fonction GL_ARB_occlusion_query n'est en fait qu'un simple compteur de fragments non rejetés (cf. figure 6.20). Le début et la fin du compte sont spécifiés par des commandes OpenGL s'insérant dans le flux d'information qui part vers la carte graphique. Le compteur est situé après le *depth test*.

Lorsque l'on affiche un objet en l'entourant de commandes de début et de fin de requête[4], la fonction
GL_ARB_occlusion_query compte le nombre de pixels qui s'allument. Si l'on prend soin de désactiver l'écriture dans le tampon de couleur et de profondeur, il est possible de réaliser ce test sans influencer l'affichage.

En affichant la boîte englobante d'un objet, on obtient une borne supérieure du nombre de pixels qu'aurait affiché cet objet. L'utilisation de boîtes englobantes entraîne donc une perte de précision, mais aussi un gain de rapidité. En effet, celles-ci sont plus grossières et donc beaucoup plus rapides à afficher que les objets qu'elles englobent. De plus, dans Dynamic Graph, rappelons-le, on ne connaît même pas l'objet que l'on va afficher : celui-ci dépend de la précision.

Rendu de l'avant vers l'arrière

La fonction GL_ARB_occlusion_query nous donne un moyen de calculer une *sur-estimation* de la surface réellement affichée de l'objet *occulté*. Pour avoir un algorithme conservatif, il faut que l'obstacle utilisé soit en quelque sorte une *sous-estimation* de l'obstacle réel. Dans notre cas, l'obstacle est le *Z-buffer*. Celui-ci, durant tout son remplissage, est une sous-estimation du *Z-buffer* final obtenu à la fin de l'affichage.

[4]

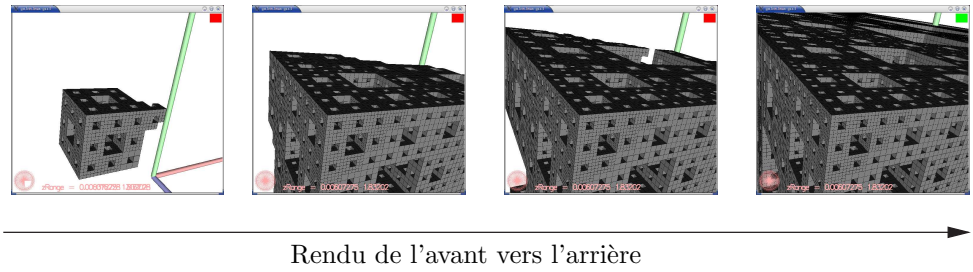


FIG. 6.21 – Dynamic Graph trie les amplifieurs de l'avant vers l'arrière lors de l'affichage.

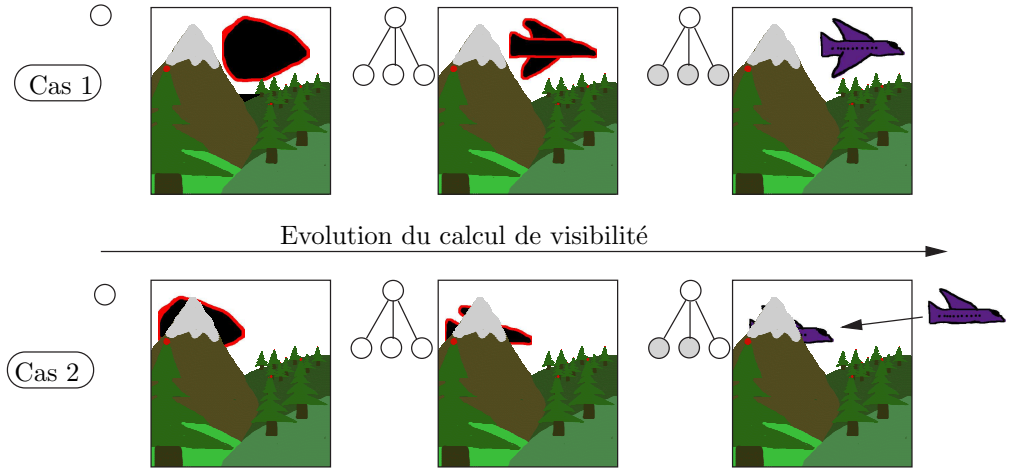


FIG. 6.22 – Les tests de visibilité s'effectuent sur une scène déjà partiellement dessinée. Avant de continuer l'amplification d'un objet, on teste la visibilité de sa boîte englobante. Si celle-ci est visible, l'amplification continue, sinon elle s'arrête. Lorsque la précision d'un amplifieur visible est suffisante, celui-ci est affiché.

L'idéal serait de lancer les requêtes de visibilité sur le *Z-buffer* final. Ceci est malheureusement impossible car on ne peut remplir un *Z-buffer* qu'en évaluant la scène. Or, le calcul de visibilité est justement nécessaire à l'évaluation de cette scène : le problème "se mord la queue". On pourrait imaginer utiliser le *Z-buffer* au pas de temps précédent. Celui-ci est effectivement souvent une bonne approximation de la valeur courante. Malheureusement, assurer un calcul conservatif dans ces conditions est très difficile[5].

Afin de permettre un calcul de la visibilité *en même temps* que le calcul de la scène, celle-ci est rendue de l'avant vers l'arrière grâce à un parcours adéquat (cf. figure 6.21 et 5.9). Le but du jeu est de remplir le *Z-buffer* aussi vite que possible avec des valeurs aussi proches de la caméra que possible. De cette façon, le *Z-buffer* sera un *occluder* de bon qualité (i.e qui cachera le plus de chose possible). Avant d'enrichir un *amplifieur*, sa boîte englobante sera testée sur le *Z-buffer* courant (cf. figure 6.22). Les fils ne seront générés que si le nombre de pixels allumés par la boîte englobante n'est pas nul.

Limitation du nombre d'appels à GL_ARB_occlusion_query

Lors de l'évaluation d'un *amplifieur*, le *générateur* peut évaluer les fonctions *draw* ou *drawBoundingBox* (ou les deux). Le but de l'*organiseur* est ici assez complexe : il doit bien sûr minimiser

[5] On peut imaginer "reculer" le *Z-buffer* par son déplacement maximum possible, auquel cas de nombreux amplifieurs cachés ne seront pas détectés

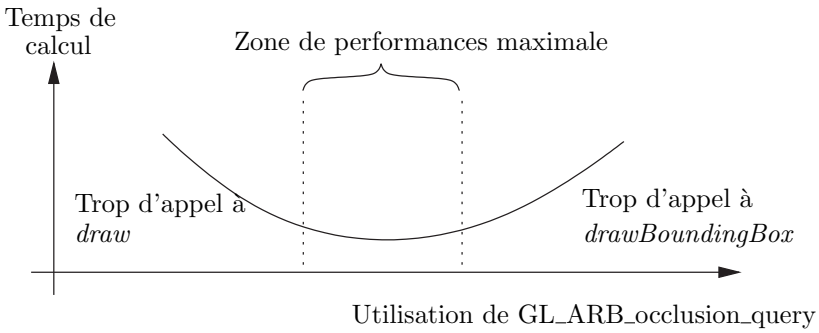


FIG. 6.23 – Comme toute méthode d'optimisation basée sur des tests, il faut s'assurer que le coût de ces derniers ne soit pas au bout du compte nuisible aux performances. En d'autres termes, il faut s'assurer que le test ne coûte pas plus cher que ce qu'il permet d'économiser.

le nombre d'appels à la fonction *draw*. Pour cela, il va utiliser les fonctions *drawBoundingBox* qui, via une utilisation de GL_ARB_occlusion_query, détecte si l'amplifieur est occulté.

Mais il doit aussi minimiser le nombre d'appel à la fonction *drawBoundingBox*, notamment très coûteuse au niveau du taux de remplissage. Le problème ressemble à une minimisation (cf. figure 6.23). Une façon d'aborder ce problème est de donner une valeur de pertinence aux requêtes de visibilité dans le but d'en ignorer certaines. Remarquons que, conservativité oblige, tout requête non pertinente est considérée comme positive : l'objet est par défaut visible.

De nombreux critères déterminent la pertinence d'une requête :

Cohérence temporelle : un objet "très" visible à un pas de temps a de grande chance d'être visible au pas de temps suivant ;

Amplifieurs grossiers : les amplifieurs très grossiers qui sont dans le champ de vision ont de fortes chances d'être au moins partiellement visibles. De plus, leurs boîtes englobantes étant énormes (plus grosses que l'écran), le surcoût engendré par le taux de remplissage peut être extrêmement important. Il est donc bénéfique d'ignorer les amplifieurs jusqu'à une certaine profondeur dans l'arbre ;

Amplifieurs fins : durant l'évaluation d'un amplifieur, il est possible d'estimer, en se basant sur les données du pas de temps précédent, la taille du sous arbre dont il est la racine (grâce à la cohérence temporelle et à la précision). Si cette taille est très faible, il est inutile de réaliser un test de visibilité.

Ces critères sont combinés pour former une heuristique décidant si la requête doit être lancée ou non. Une analyse précise n'a pas été réalisée et très clairement, cette heuristique est très perfectible. Actuellement, une somme pondérée de ces critères, avec des coefficients choisis empiriquement, est comparée à un seuil : le test de visibilité n'est réalisé que si la pertinence est suffisante.

6.3.3 Parallélisme GPU/CPU

La subtilité majeure de l'algorithme présenté ici est d'assurer une bonne assynchronisation entre la carte graphique et le processeur central. Nous verrons tout d'abord pourquoi une approche naïve nuit fatalement aux performances. Ensuite, nous énumérerons les moyens existants de synchronisation entre le CPU et le GPU. Enfin, nous détaillerons la mise en oeuvre du parallélisme et discuterons des auto-occultations.

Utilisation naïve

L'utilisation de GL_ARB_occlusion_query requiert quelques précautions d'utilisation. En effet, imaginons une utilisation naïve d'une requête de visibilité :

– l'utilisateur affiche la boîte englobante d'un objet en activant le calcul de visibilité ;

– il demande ensuite le résultat, soit le nombre de pixels dessinés ;
– puis, en fonction du résultat, il peut décider, par exemple, de dessiner l'objet en question.

Cette approche est à proscrire. En effet, si l'utilisateur demande le résultat de la requête juste après l'avoir effectuée, il forcera la géométrie de la boîte englobante à traverser tout le pipe-line *sans que celui-ci pourrait continuer à se remplir*. La carte graphique se videra pendant que le processeur attend sans rien faire. Le temps moyen de traversée de la carte est non négligeable[6]. En conséquence, il n'est pas tolérable de laisser inactif le GPU et le CPU à chaque requête de visibilité.

Moyen de synchronisation

Si l'on compare le CPU avec un énorme carrefour complexe et très fréquenté, le GPU est une autoroute à sens unique. Suivant cette métaphore, l'utilisation naïve décrite plus haut revient à laisser entrer quelques voitures sur l'autoroute, puis à la fermer et à attendre que les voitures arrivent à destination avant de la rouvrir. L'accumulation de voitures à l'entrée de l'autoroute paralyserait complètement le grand échangeur (le CPU), contraint d'attendre sans bouger que l'autoroute soit de nouveau ouverte.

Il existe actuellement trois moyens pour contrôler la synchronisation entre un CPU et la carte graphique :

glFinish : cette fonction envoie toutes les commandes en attente et se termine une fois le *pipeline* graphique vidé (on vide l'autoroute) ;

glFlush : permet d'envoyer à la carte graphique toute les commandes en attente (on ouvre l'autoroute et toutes les voitures qui attendaient peuvent avancer) ;

GL_NV_fence : cette extension permet d'envoyer des bannières dans le flux d'informations et de contrôler leur état d'avancement (on fait entrer des voitures spéciales de différentes couleurs dans l'autoroute et on peut demander aux différents péages s'ils ont vu les voitures passer).

L'utilisation de *glFinish* est à proscrire (le CPU et le GPU attendent). L'utilisation des *GL_NV_fence* est prometteuse mais n'a pas été étudiée ici. La solution choisie consiste à utiliser *glFlush* couplé à une estimation du temps de traversée du GPU.

Mise en œuvre du parallélisme

Pour remédier au temps perdu par une utilisation naïve, il convient avant tout de toujours envoyer de nouvelles informations au GPU. Pour cela, il est absolument nécessaire de ne pas demander le résultat de la requête de visibilité juste après l'avoir exécuté par un *glFlush*. Chaque amplifieur en attente devient alors bloquant, c'est-à-dire que le générateur ne peut plus les visiter avant que le résultat des requêtes soient disponibles. Si éventuellement tous les noeuds sont bloquants, alors la seule solution est d'attendre que les réponses des requêtes soient disponibles.

De plus, il n'est pas conseillé d'appeler *glFlush* trop souvent car le bus aime bien les gros paquets d'informations. Il faut donc envoyer des *groupes* de requêtes. La taille des groupes ne doit pas être trop grande, sans quoi tous les noeuds risquent de devenir bloquants. Cette situation arrive parfois en début de création de graphe lorsque les choix sont peu nombreux. C'est une raison de plus pour développer les tous premiers niveaux du graphe sans tester la visibilité.

Dans Dynamic Graph, le CPU utilise le générateur pour visiter les noeuds non bloquants pendant que le GPU calcule les requêtes de visibilité. Les réponses, une fois rapatriées sur le CPU, vont débloquer les noeuds bloquants et le générateur pourra alors les visiter.

Auto-occultation

Mais le problème ne s'arrête pas là. Il est nécessaire de limiter les auto-occultations au sein d'un même groupe de requêtes. Afin d'illustrer cette nécessité, penchons nous sur l'exemple suivant : le

[6] Les données sont quasi-inexistantes sur le temps que met une information à traverser le *pipe-line* graphique et le nombre d'étages dont est composé ce dernier est une donnée que les constructeurs ne publient pas.

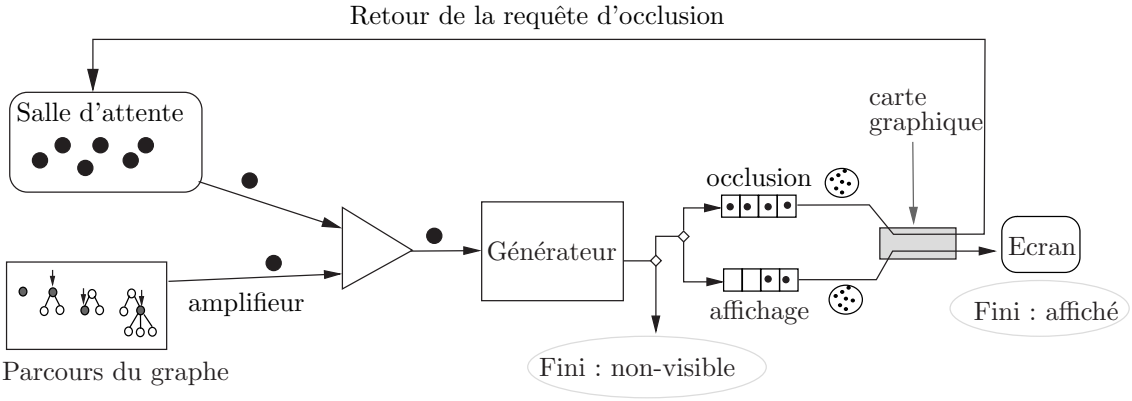


FIG. 6.24 – Afin d'assurer un parallélisme efficace entre le GPU et le CPU, la génération de l'arbre d'évaluation est jalonnée de files d'attente. Notamment, les files d'attentes de l'affichage et de l'occultation ont pour premier objectif de grouper les commandes OpenGL. De plus, la file d'attente de l'affichage applique un tri local d'avant vers l'arrière pour compléter le tri réalisé lors du parcours du graphe.

visiteur laisse les noeuds A et B en attente le temps que leur calcul de visibilité se termine. Pendant ce calcul, A et B n'influent pas sur le Z-buffer (grâce à *glDepthMask*) mais comptent juste le nombre de pixels qu'allument leurs bounding-box. Que se passe-t-il si A occulte complètement B ? Comme les visibilités de A et B sont évaluées en même temps, cette occultation ne sera pas détectée, amenant des calculs inutiles (mais conservativement sans risque).

Il est donc nécessaire de bien répartir les membres d'un groupe de requêtes de visibilité selon leur placement sur l'écran. Idéalement, si les membres sont mutuellement non-recouvrants, le test de visibilité sera optimal. En pratique, il serait trop coûteux d'assurer cette condition et il est suffisant de bien répartir les requêtes sur l'écran pour obtenir de bonne performance. Cette répartition est actuellement implicitement et partiellement réalisé par le tri en profondeur. Cette méthode simpliste devrait pouvoir être améliorer par une meilleur connaissance de la répartition des boîtes englobantes sur le plan de l'image.

6.3.4 Bilan

Nous décrirons tout d'abord les répercussions importantes de cet algorithme sur l'architecture de Dynamic Graph. D'un point de vue plus abstrait, nous décrirons la carte graphique comme l'observateur du monde virtuel contenu dans les modèles de Dynamic Graph. Plus concrètement, nous discuterons rapidement des performances. Enfin, nous ajouterons une note sur l'utilisation de la transparence avec cet algorithme de visibilité.

Un pipe-line complexe

Au bout du compte, les contraintes qu'entraîne le parallélisme entre le processeur central et la carte graphique compliquent notablement le *pipe-line* d'évaluation. Sans rentrer dans les détails, la figure 6.24 donne un aperçu simplifié de la structure utilisée. Le rôle de la *salle d'attente* est essentiel : elle permet aux amplifieurs d'attendre sagement que leur requête de visibilité soit terminée.

La carte graphique observe le modèle

Fondamentalement, l'algorithme de rendu de Dynamic Graph est l'esclave de la carte graphique (cf. figure 6.25). Ceci diffère radicalement des approches pour lesquelles on envoie toutes

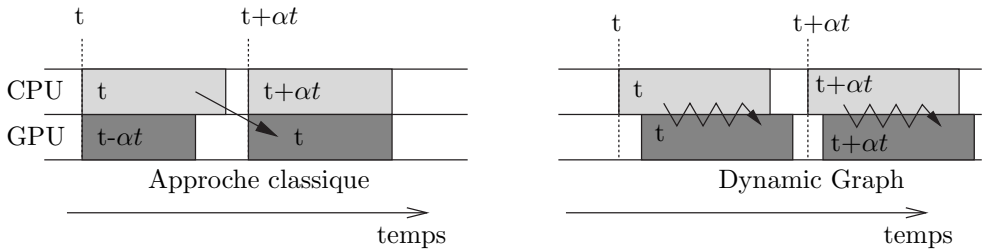


FIG. 6.25 – L'échange complexe que Dynamic Graph fait avec la carte graphique permet d'afficher un modèle en même temps qu'il est animé (évalué). Plus classiquement, les méthodes d'animation affichent usuellement le résultat d'une animation au pas de temps suivant. De ce point de vue, notre approche réduit la latence [LRC+02] de l'affichage.

les informations d'un bloc. Ici, l'algorithme de détection d'occultation amène un échange plus complexe.

Cette évolution est en fait très naturelle. Rappelons que c'est l'observateur qui limite la génération de l'arbre d'évaluation par des critères de visibilité et de précision. Ne pourrait on pas considérer que la carte graphique est cet observateur ? C'est en quelque sorte elle qui *observe* les données virtuelles du modèle. On peut représenter la carte graphique comme un œil intermédiaire (dont l'écran est la rétine) entre le modèle et le créateur. Il est tout naturel que Dynamic Graph interroge la carte graphique pour lui demander son avis : "vous en voulez encore ou on s'arrête là ?". D'ailleurs, il est probable que le calcul de précision ou de *view frutrum culling* bénéficie aussi un jour des avantages de l'accélération matérielle par un retour de l'information du GPU vers le CPU.

Performance

Les résultats sont graphiquement corrects, ce qui est tout à fait normal puisque l'algorithme est conservatif. La figure 6.26 donne un aperçu des résultats grâce à une seconde caméra.

Dans le cas du cube de Sirpienski, une mesure du taux de remplissage a été effectuée (cf. figure 6.27). Le taux de remplissage mesure le nombre de fragments passant nécessaire au rendu final. Dans un cas idéal, un seul fragment serait envoyé par pixel. Mais ce chiffre n'est jamais égalé notamment parce qu'un grand nombre de fragments inutils doit être produit pour obtenir le résultat final. Au final, on peut mesurer le nombre total de fragments générés en *équivalent écrans*. Par exemple, un taux de remplissage égal à 12*écrans* signifie que l'équivalent de 12 écrans a été rempli par les fragments durant *un* affichage.

Le taux de remplissage engendré par l'affichage lui-même est très stable et reste entre 2*écrans* et 5*écrans* (cette valeur dépend beaucoup du modèle). Lorsque la caméra est très proche du cube, le taux de remplissage engendré par les boîtes englobantes monte jusqu'à 12*écrans*. Au total, le taux de remplissage est monté jusqu'à 17*écrans* pour une observation très proche. Sans l'algorithme de détection d'occultation, dans les même conditions, le taux de remplissage vaut 200*écrans* (soit un facteur 10) ! ! Clairement, le jeu en vaut la chandelle.

Néanmoins, un taux de remplissage de 12 peut être jugé trop élevé. Nous estimons que les heuristiques utilisées pour décider de l'exécution ou non du test de détection d'occultation sont actuellement très basiques. Une étude plus poussée améliorerait sans aucun doute ces résultats.

Le plus surprenant vient du fait que, d'une part, ce taux de remplissage énorme ne nuit pas à un affichage interactif. Ceci est certainement dû au fait que l'affichage des boîtes englobantes et très élémentaire (aucun éclairage, géométrie grossière). Il est possible que des techniques de *early Z test* (cf. [WW03]) accélèrent très sensiblement les calculs. De plus, il est clair que la puissance des cartes graphiques actuelles joue un rôle décisif.

Les performances de cet algorithme dépendent du type de scène affiché. Actuellement, seul une dizaine de modèles relativement simples ont été réalisés. Pour chacun d'eux, le gain est évident.

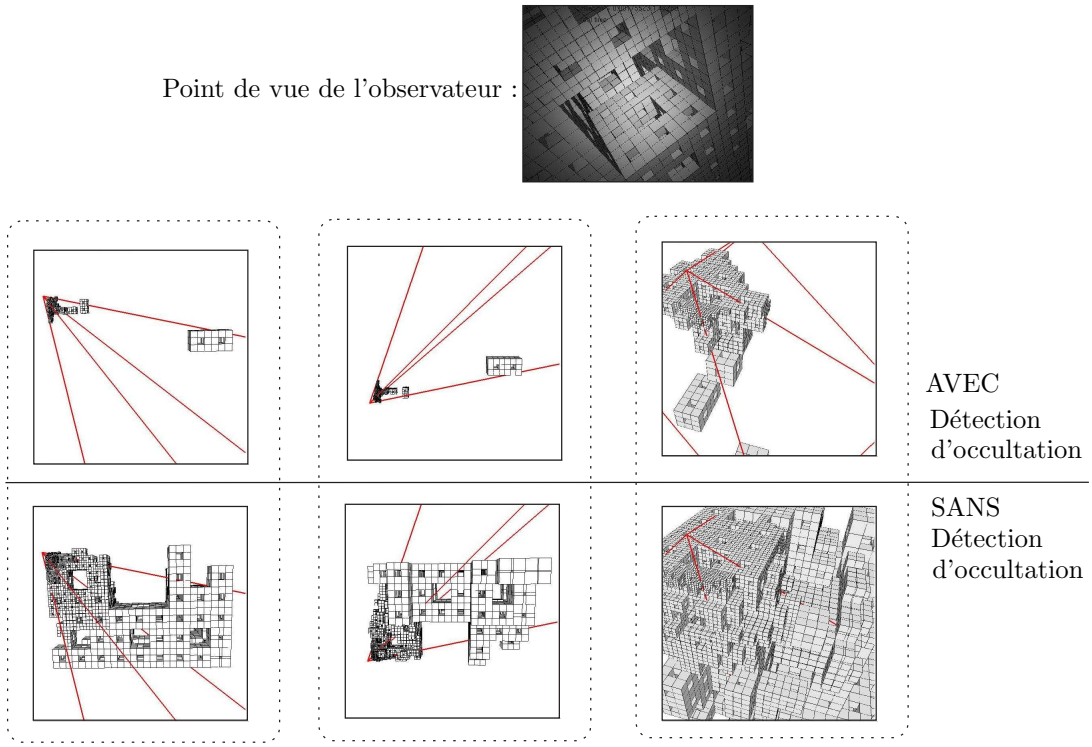


FIG. 6.26 – Sous différents angles, on apprécie la quantité d'informations supprimées.

Il est difficile de prédire les résultats dans le cas général. Il est pourtant clair que plus le taux de remplissage de la scène est grand, plus les performances seront bonnes. Par conséquent, plus la scène est complexe et volumique, plus le gain sera marqué. Quant à la comparaison de cet algorithme avec les algorithmes existants, le constat est simple : plus la scène affichée sera vaste, précise, chaotique et animée, plus l'algorithme présenté ici montrera sa supériorité.

Transparence

La transparence n'est actuellement pas permise en même temps que l'utilisation de l'algorithme de visibilité. Cela est dû au fait qu'il est actuellement impossible de mettre à jour le *Z-buffer* en utilisant la fonction *glBlendFunc* avec un rendu de l'avant vers l'arrière. Suite à une discussion avec Fabrice Neyret, il semblerait qu'une extension très simple puisse remédier à ce problème. Il suffirait d'ajouter une nouvelle sorte de test *alpha* du type :

`si` $(A_{screen}$ `>` $A_{threshhold})$ `alors` $z_{screen} \leftarrow z_{fragment}$

La valeur $A_{threshhold}$ détermine la valeur à partir de laquelle on considère le pixel comme opaque. Cette extension est simple et prometteuse mais reste à être implémenter en hardware.

6.4 Bilan

Les entrailles de la bête sont désormais connues, voyons comment on peut la dompter.

Une structure souple, dynamique et performante

Dynamic Graph offre une structure très dynamique régénérée à chaque pas de temps. Néanmoins, contrairement à la plupart des méthodes procédurales "à la volée", des mécanismes permettent d'accéder aux valeurs des pas de temps précédent.

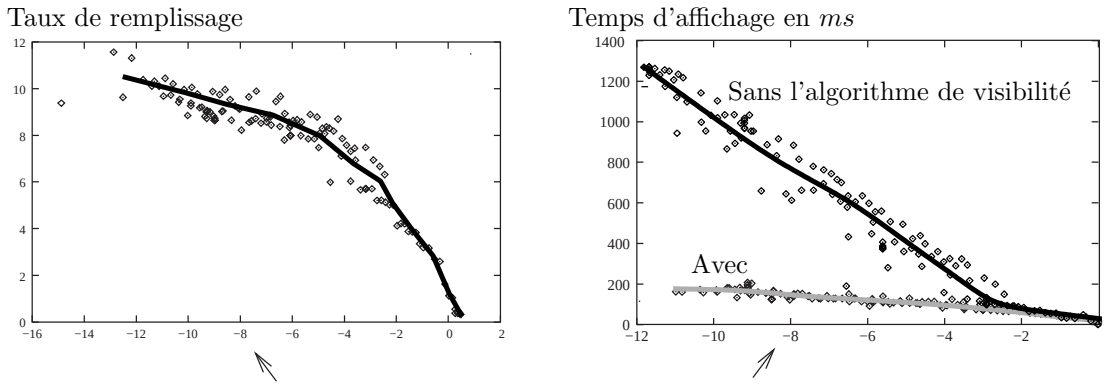


FIG. 6.27 – Les mesures sont effectuées en fonction de l'échelle d'observation (plus précisément, en fonction de la distance à l'objet le plus proche). Le modèle utilisé est le cube de Sirpienski. À gauche, la courbe représente le taux de remplissage engendrée par *les boîtes englobantes uniquement*. À droite, des mesure du *frame-rate* avec et sans l'algorithme de visibilité ont été effectuées.

La forme affichée est une fonction de l'observateur, en perpétuelle adaptation aux variations de ce dernier. Des critères de précision et de visibilité assurent que seule l'information nécessaire à l'observation est générée.

La structure en *pipe-line* de la carte graphique influe notablement l'architecture de Dynamic Graph. Des méthodes de parallélisme dédiées permettent au processeur central d'utiliser la puissance de ces dernières sans pour autant nuire aux performances.

Vers la modélisation

Il peut paraître un peu ambitieux de proposer une solution recouvrant *tout* le processus de création. Généralement, un travail de recherche est plus local et s'attarde exclusivement sur la modélisation ou l'animation ou le rendu, mais pas sur tous ces domaines à la fois !

Tout d'abord, précisons que le processus de création que j'ai développé dans Dynamic Graph est très simpliste. Notamment, comme nous le verrons bientôt, la phase de modélisation est réalisée essentiellement par une interface textuelle. J'aimerais préciser aussi que j'aurais aimer me raccrocher à des outils existants. Mais l'originalité du langage descriptif de Dynamic Graph impose, en amont et en aval du processus de création, des outils très spécifiques. Nous venons de voir les particularités de l'algorithme de rendu (en aval). Voyons maintenant les particularités de l'algorithme de modélisation (en amont).

Dynamic Graph propose une représentation plus fondamentale dans laquelle l'affichage ne joue finalement qu'un second rôle. La modélisation revient essentiellement à décrire le modèle de façon minimaliste par des paramètres pertinents : plutôt que la surface, c'est le "squelette" qui est décrit. Le créateur manipule ensuite ces paramètres pour les animer ou les rendre plus précis.

Voyons maintenant, dans le chapitre suivant, comment aborder cette modélisation complexe.

Chapitre 7

Dynamic Graph : vue du créateur

Ce chapitre décrit Dynamic Graph du point de vue du créateur. C'est donc la partie modélisation qui est détaillée ici. La structure de ce chapitre est la suivante :

Environnement de modélisation : la section 7.1 décrit l'environnement graphique de Dynamic Graph ;

Création d'un amplifieur : la section 7.2 décrit la tâche principale du créateur durant la modélisation : la création d'un amplifieur ;

Arbres animés : la section 7.3 termine ce chapitre par la description du modèle le plus sophistiqué réalisé avec Dynamic Graph : des arbres agités par le vent.

7.1 Environnement de modélisation

Dans cette section sont détaillées les différents fonctionnalités que Dynamic Graph propose pour simplifier la modélisation.

7.1.1 Sous un autre angle

Les formes que Dynamic Graph génère s'adaptent à l'observation. Dans le meilleur des cas, on ne distingue pas cette adaptation lorsque l'on prend le point de vue de l'observateur : elle est invisible, camouflée. De plus, chaque mouvement de l'observateur provoque une transformation du modèle. il est parfois "congeler" le modèle pour pouvoir l'observer sous tous les angles. De cette façon, il est plus facile de déboguer et de trouver d'éventuels défaut.

Afin de permettre cela, Dynamic Graph propose, en plus de l'*observation principale*, une *seconde observation*, c'est-à-dire une observation pour laquelle la forme affichée reste adaptée à l'observation principale (cf. figure 7.1).

7.1.2 Sélection multi-échelle

La sélection est une opération de base dans toute modélisation. Dans Dynamic Graph, une sélection est un ensemble d'amplifieurs. La persistance de l'information est assurée par l'*arbre permanent* : les amplifieurs sélectionnés sont inscrits dans ce dernier.

Trois opérations de sélection classiques sont possibles :

- création d'une nouvelle sélection constituée d'un ensemble d'amplifieurs ;
- intersection d'une nouvelle sélection avec la sélection courante ;
- union d'une nouvelle sélection avec la sélection courante.

Deux autres opérations plus originales sont propres à Dynamic Graph :

hiérarchique : il est impossible de sélectionner graphiquement un amplifieur $\mathcal{A}$ qui n'affiche pas de géométrie. En revanche, il est possible de sélectionner un des amplifieurs contenu dans le sous-arbre de l'arbre d'évaluation (cf. section 6.1) dont $\mathcal{A}$ est la racine. Ensuite, il suffit

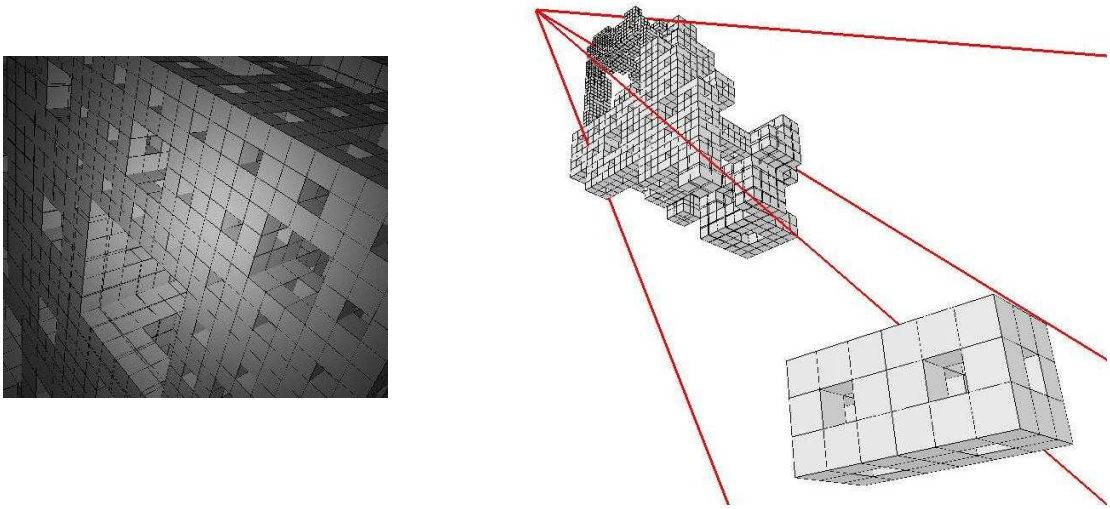


FIG. 7.1 – Une seconde observation permet un regard critique sur le modèle.

de remonter d'amplifieur père en amplifieur père pour trouver $\mathcal{A}$. Dynamic Graph propose donc de transformer une sélection courante de façon hiérarchique : la nouvelle sélection est constituée par les père (resp. les fils) des amplifieurs de la sélection courante.

temporelle : le tampon d'arbre (cf. section 6.1) garde une mémoire des n derniers graphes d'évaluation. Pour accéder à un amplifieur ancêtre, il faut sélectionner l'amplifieur présent et remonter le temps (tant que le tampon d'arbre le permet). Dynamic Graph propose de transformer une sélection courante de façon temporelle : la nouvelle sélection est constituée par les ancêtres (resp. les successeurs) des amplifieur de la sélection courante.

7.1.3 Le *DebugPanel*

Dynamic Graph propose une boîte de dialogue composée de plusieurs champs de curseurs et de boutons : le *debugPanel* (cf. figure 7.2). C'est un outil d'aide à la modélisation.

Cet outil est un assistant très pratique à la modélisation avec Dynamic Graph. Avec une ligne de commande, on peut initialiser un curseur et lui donner un nom (par exemple : "taille cube" ou "couleur"). Avec une autre ligne de commande, il est possible d'appeler cette valeur. Durant l'exécution du programme, il devient alors possible de changer la valeur du curseur et d'en apprécier les conséquences.

7.1.4 Micro-interface graphique

Dans les modeleurs classiques, il est possible de sélectionner une partie du maillage et de l'éditer. Cette opération est impossible en modélisation procédurale puisque aucune représentation géométrique n'est imposée. Néanmoins, afin de faciliter l'édition, le créateur peut, dans chaque amplifieur, coder une micro-interface graphique. S'il fait cela, il pourra, après avoir sélectionné un amplifieur, visualiser ses caractéristiques et interagir avec lui (cf. figure 7.3).

Concrètement, les interfaces graphiques sont uniquement basées sur OpenGL. Elles intègrent une gestion des évènements souris et clavier. Aucune bibilothèque d'interface n'a malheureusement été utilisée. Il serait intéressant d'associer à chaque amplifieur une interface dédié simple et facilement implémentable (des sortes de *DebugPanel* personnalisés).

L'utilisation la plus simple consiste à afficher, sous forme de texte, les caractéristiques de l'amplifieur sélectionné. Une utilisation plus complexe permettra une visualisation graphique éventuellement tridimensionnelle de certaines caractéristiques complexes de l'amplifieur. Enfin, il est

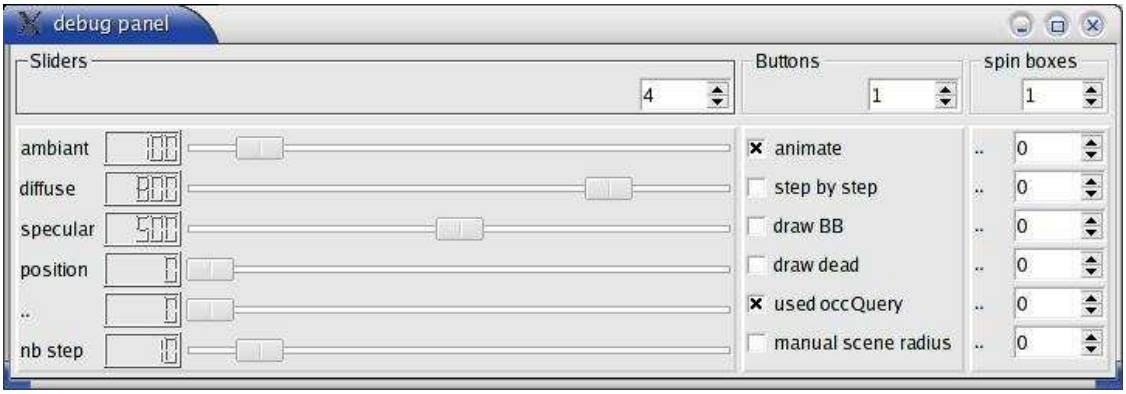


FIG. 7.2 – Le *debugPanel* offre la possibilité de déclarer et de modifier dynamiquement une infinité autant de valeur que le souhaite le créateur. C'est en quelque sorte la version la plus primitive qui soit d'une interface graphique procédurale. Cet outil est très utile pour contrôler les très nombreux degrés de liberté qu'offre la modélisation procédurale.

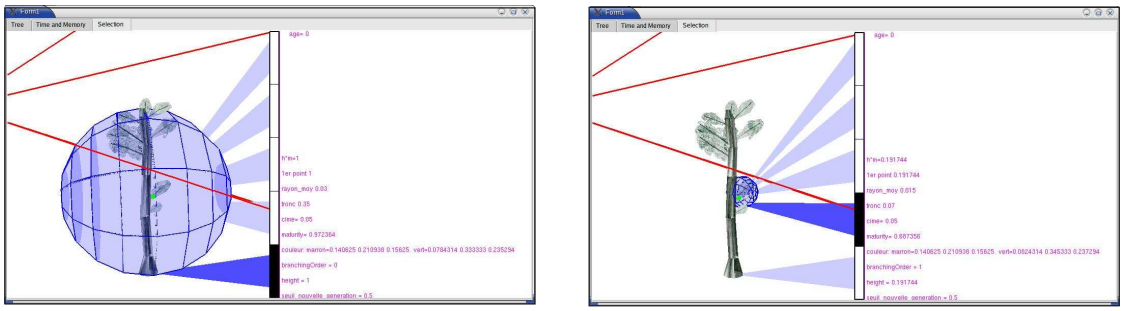


FIG. 7.3 – Les micros-interfaces locales permettent d'interagir avec un amplifieur particulier.

possible, via les fonctions *événements*, d'interagir avec l'objet et, par exemple, de changer certaines valeurs à la volée.

7.1.5 Mesure de performance

Lors d'une modélisation procédurale assistée par Dynamic Graph, les coûts mémoires et les temps de calcul du rendu dépendent essentiellement des fonctions que le créateur a codé dans les amplifieurs. Il est donc très utile de pouvoir rapidement faire des mesures afin d'éviter de coder des amplifieurs trop gourmands en mémoire ou bien trop lents à calculer. Pour cela, Dynamic Graph fournit une visualisation de courbes de performances.

Plusieurs mesures de consommation de mémoire sont effectuées à chaque pas de temps :
- occupation mémoire de tout l'arbre d'évaluation ;
- occupation mémoire moyenne pour chaque type d'amplifieur ;
- occupation mémoire individuelle pour chaque amplifieur.

Plusieurs mesures de temps de calcul sont réalisées :
- temps de calcul global d'une image ;
- temps de calcul moyen par pour chaque type d'amplifieur ;
- temps de calcul individuel pour chaque amplifieur.

7.1.6 Visualisation de l'arbre d'évaluation

De l'évaluation de la scène résulte l'*arbre d'évaluation*. La visualisation de ce dernier (cf. figure 7.4) s'avère dans certains cas fort utile. Par exemple, on peut sélectionner les amplifieurs directement dans cette fenêtre et naviguer plus facilement dans la hiérarchie de l'arbre.

De plus, en attribuant des couleurs particulières aux noeuds, il est possible de représenter certaines statistiques intéressantes sur les amplifieurs :
- les valeurs de visibilité ou de précision,
- la valeur de la maturité,
- la place mémoire ou le temps de calcul nécessaires à l'évaluation,
- ...

D'autres types de visualisation peuvent être définis. On peut imaginer, dans le cas particulier des prairies par exemple, afficher la densité de brins d'herbe de chaque amplifieur. Ici encore, Dynamic Graph ne propose pas de solution clef en main, mais une visualisation très souple et très paramétrable.

7.1.7 Temps réel, temps virtuel

Compte tenu du flux très dynamique d'informations durant l'exploration d'une scène animée multi-échelle, il est vital de maîtriser l'écoulement du temps. Pour cela, Dynamic Graph offre un paramétrage très souple du couple $(f_{virtuel}, f_{reel})$:
- La fréquence d'affichage f_{reel} est caractérisée par le temps entre deux affichages. De façon imagée, f_{reel} est mesuré par un chronomètre que tiendrait le créateur devant son écran ;
- La fréquence de l'animation $f_{virtuel}$ est caractérisée par le temps virtuel qui s'est écoulé entre deux images. De façon imagée, $f_{virtuel}$ est mesuré par une horloge virtuelle que l'on pourrait afficher au sein même de la scène tridimensionnelle.

Dynamic Graph peut forcer la valeur de $f_{virtuel}$ sans tenir compte de f_{reel}. L'inverse est aussi possible grâce à une régulation très simple de la fonction de précision, via la valeur S_{ideale} (cf. section 6.2). Il est possible d'asservir ces deux valeurs. Par exemple, un asservissement $f_{virtuel} = f_{reel}$ est caractéristique d'un affichage *temps-réel*.

Il est aussi possible de contrôler manuellement f_{reel} : c'est le *pas à pas*. Il permet de faire avancer l'animation image par image et de détecter, par exemple, l'éventuelle apparition de bogue. Toujours dans la même optique, Dynamic Graph permet de *décomposer* f_{reel} et de générer l'arbre d'évaluation progressivement. Chaque étape de l'organiseur est alors déclenchée manuellement.

Il est possible de passer d'un mode à l'autre durant l'exécution du programme.

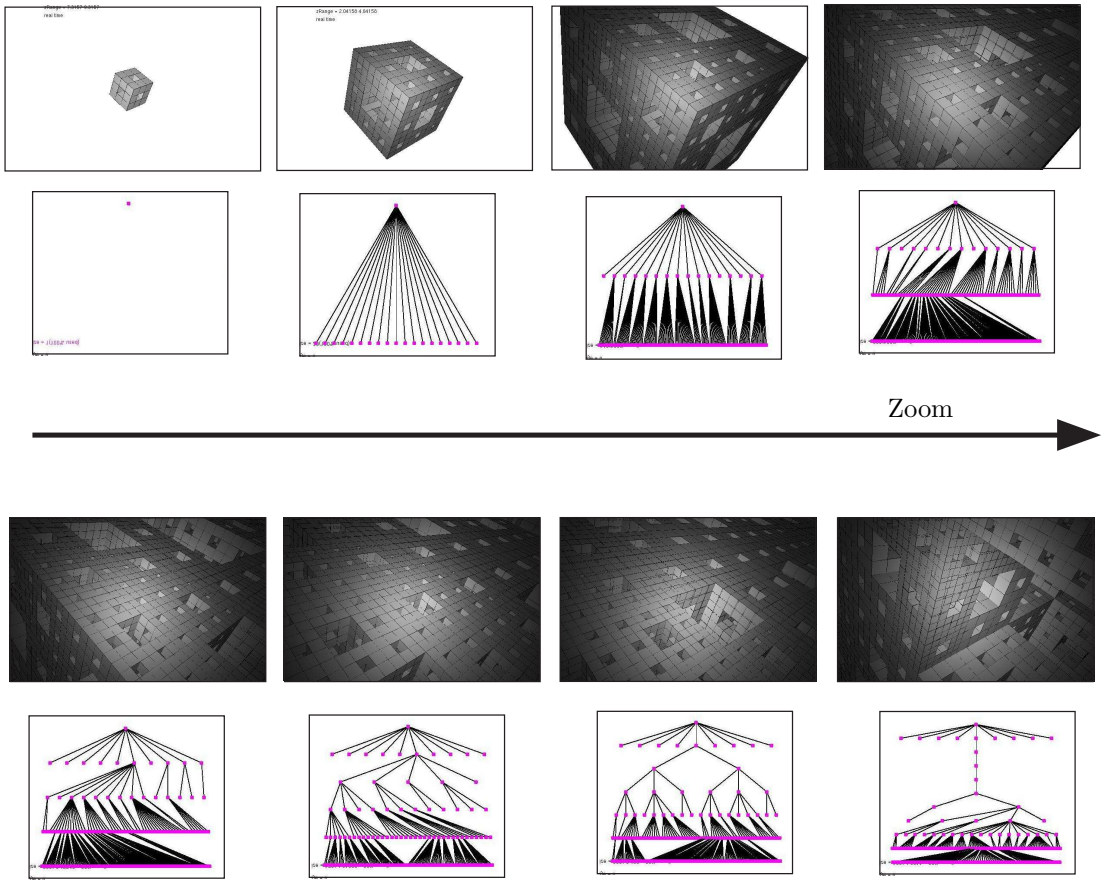


FIG. 7.4 – La visualisation de l'arbre d'évaluation donne un bon aperçu de la complexité de la scène.

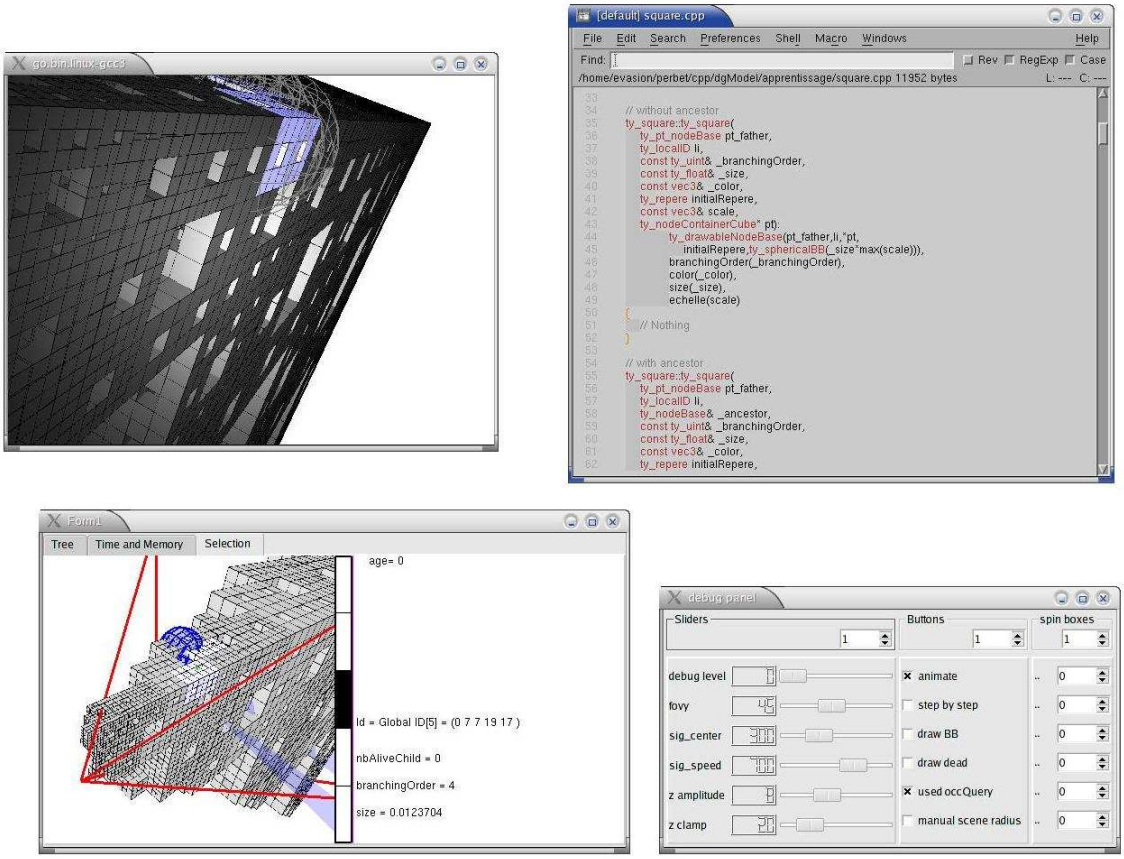


FIG. 7.5 – Exemple typique de modélisation avec Dynamic Graph. En haut à gauche, la fenêtre affiche l'objet tel qu'il est vu par l'observateur principal. En bas à gauche, une seconde observation permet de contrôler la manière dont le modèle s'adapte au point de vue ; la sélection permet d'accéder aux micro-interfaces graphiques. En bas à droite, le *debugPanel* permet de changer dynamiquement certaines valeurs. En haut à droite, un éditeur de texte permet la modélisation des amplifieurs.

7.2 Création d'un amplifieur

Les amplifieurs sont la base du langage descriptif de Dynamic Graph. Nous allons voir ici les détails de la création d'un générateur d'amplifieurs dans le cas simple du cube de Sirpienski. Nous décrirons tout d'abord rapidement l'environnement de travail. Nous détaillerons ensuite les étapes nécessaires à la création d'un amplifieur. Enfin, nous nous attarderons sur quelques points sensibles auxquels le créateur doit prêter une attention particulière.

7.2.1 Cadre de travail

Compte tenu de l'aspect très particulier de la modélisation que propose Dynamic Graph, nous décrivons ici le cadre de travail d'un point de vue très général. Du point de vue du Dynamic Graph, un modèle (i.e. une forme tridimensionnelle) est une librairie dynamique (sous linux, un fichier dont l'extension et *so*). Les fonctionnalités décrites en 7.1 assistent le créateur dans sa tâche. Lors de la modification d'un modèle, Dynamic Graph recharge dynamiquement la librairie sans cesser son éxécution afin de rendre compte des modifications le plus vite possible.

Le projet

Très concrètement, comment se passe une session de modélisation ? Tout d'abord, il faut créer le "projet" correspondant à un modèle. Ce projet va générer l'infrastructure nécessaire à la réalisation du modèle : un dossier, un fichier *Makefile* et le fichier principal. Ce fichier principal contient les fonctions d'échange (cf. sous-section 5.2.3).

Par exemple, c'est ici que la fonction de production de l'axiome est définie. D'autre fonctions d'échange permettent d'initialiser la librairie dynamique en éxécutant une fonction particulière lors du chargement. Il est aussi possible d'éxécuter une fonction avant chaque affichage. Parallèlement à ces fonctions d'initialisation existe des fonctions de terminaison.

L'interface

La figure 7.5 décrit une configuration classique de l'environnement. La modélisation se passe essentiellement dans l'éditeur de texte. Remarquons que l'éditeur de texte est parfaitement indépendant de Dynamic Graph. Dans la figure 7.5, l'éditeur utilisé est *NEdit* augmenté de quelques macro-fonctions dédié à Dynamic Graph.

Une classe C++

Un amplifieur est une *transformation* de la forme tridimensionnelle qui consiste à ajouter de la précision. Concrètement, il est représenté par une classe C++ héritant d'une classe virtuelle. Celle-ci impose au créateur de remplir certaines fonctions :

constructor : construit une instance de classe (un amplifieur) ;

createChild : construit les instance filles (amplifieur plus précis) ;

draw : envoie les commandes OpenGL pour dessiner la forme ;

drawBoundingBox : envoie les commandes OpenGL pour dessiner la boîte englobante.

Ces fonctions seront détaillées dans la sous section suivante 7.2.2.

Compilation à la volée

Lorsque le créateur veut afficher le modèle qu'il a écrit dans son éditeur préféré, il lance une compilation. Évidemment, il lui faudra sûrement corriger quelques erreurs. Lorsque la compilation réussit, une librairie dynamique est créée (ou modifiée si elle existe déjà).

Dynamic Graph peut charger à la volée des modèles (i.e. des librairies dynamiques). Par exemple, lors d'une modification de la librairie dynamique, Dynamic Graph détectera cette dernière et fera la mise à jour automatiquement. Ce comportement engendre quelques soucis techniques,

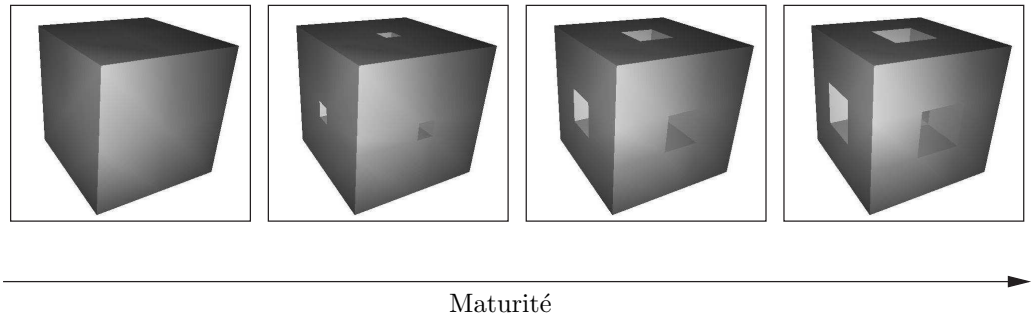


FIG. 7.6 – La fonction *draw* de l'exemple du cube de Sirpienski dépend de la maturité de l'amplifieur. Cet objet est le seul utilisé lors de l'affichage de la fractale.

mais il est néanmoins tout à fait naturel de la part d'un modeleur : lorsque l'on modifie un objet, on veut en voir les répercussions immédiatement.

La latence entre une modification et sa répercussion sur l'écran est égale au temps de compilation. Dans le cas du cube de Sirpienski, cette durée vaut environ 5 secondes (sur une machine standard achetée en 2003).

7.2.2 Détail des fonctions

Voyons maintenant plus précisément le contenu de chaque fonction de l'amplifieur : le constructeur, la fonction d'affichage, les boîtes englobantes et la génération des fils. Nous terminerons par quelques résultats de modélisation de fractales.

Constructeur

Le rôle du constructeur est principalement d'initialiser les valeurs des variables membres. Voici à quoi ressemble l'entête du constructeur :

```
square(int localID,
       float _size,
       repere localRepere)
```

les paramètres caractérisant un amplifieur sont sa taille et son repère local (exprimé dans le repère de la caméra). L'identifieur local est aussi nécessaire : rappelons qu'il permet de distinguer un amplifieur parmi ses frères.

Le constructeur est appelé durant la création de l'amplifieur axiome et à chaque génération d'amplifieurs fils.

Fonction d'affichage

Dans le cas du cube de Sierpinski, la fonction *draw* dessine l'objet représenté sur la figure 7.6. Cet objet est défini procéduralement : il est généré et envoyé à la carte graphique à la volée.

La fonction d'affichage envoie les commandes OpenGL à la carte graphique. Elle n'a aucun paramètre et utilise généralement les données stockées dans l'amplifieur. Par exemple, dans le cas du cube de Sierpinski, la fonction est la suivante :

Les variables *globalRepere* et *size* sont des variables membres de l'amplifieur. La fonction *maturity()* renvoie la maturité de l'amplifieur.

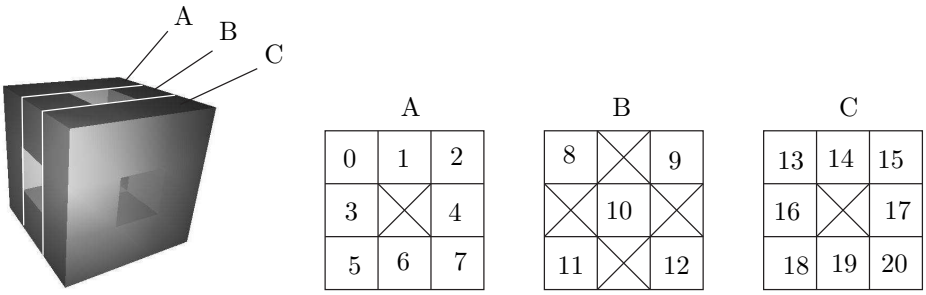


FIG. 7.7 – Les sous cubes de l'élément de base sont numérotés. La fonction *getRepere* donne la translation entre le repère de l'élément (centré sur celui-ci) et les 20 sous repères.

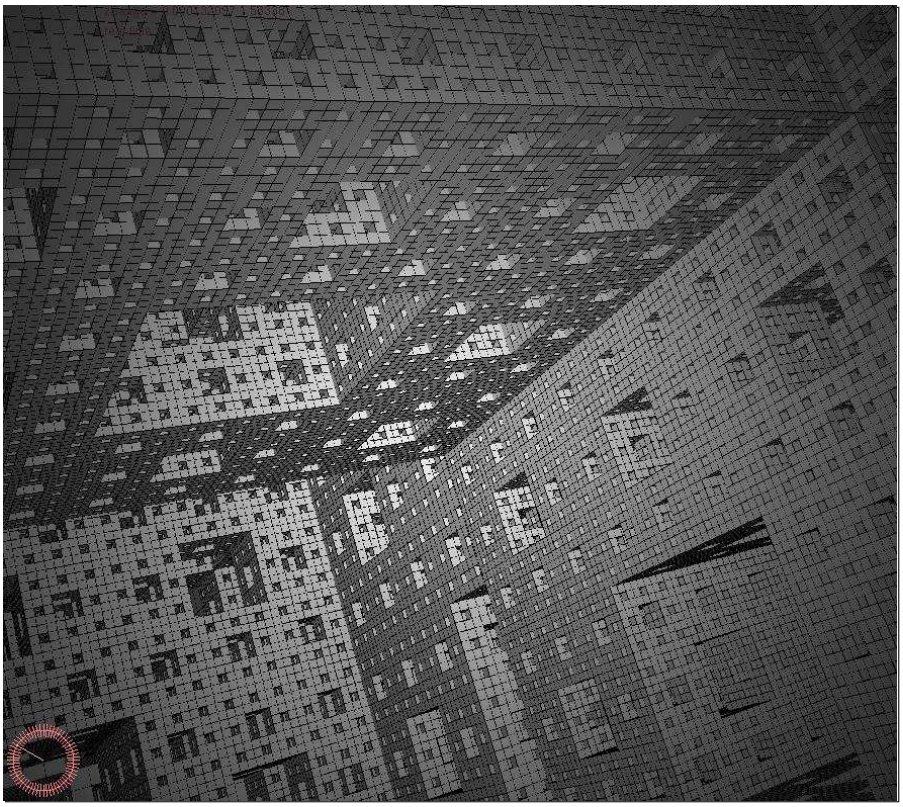


FIG. 7.8 – Pour une qualité aussi haute, la fréquence d'affichage est d'environ $1Hz$. Remarquons tout de même que la taille des détails observés, pour un cube initial mesurant un mètre, sont de l'ordre du micromètre.

```
void draw()
{
    glPushMatrix();
    glMultMatrix(globalRepere);
    draw_elem(size,maturity());
    glPopMatrix();
}
```

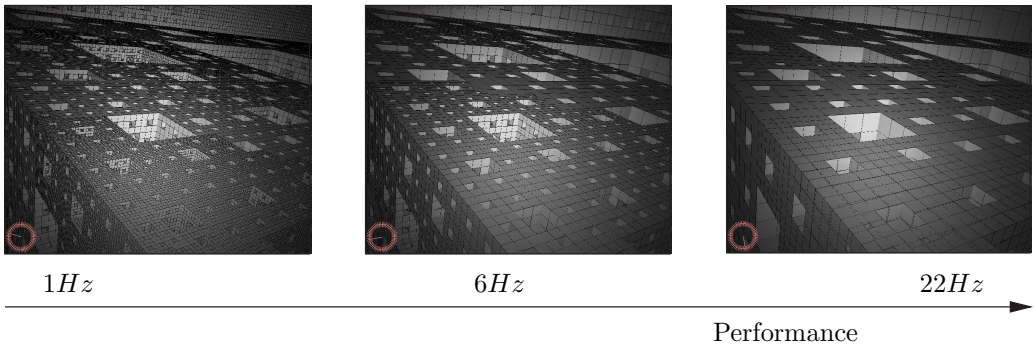


FIG. 7.9 – Différentes fréquences d'affichage : la qualité est inversement proportionnelle aux performances.

Boites englobantes

Remarquons qu'il appartient au créateur de spécifier la boîte englobante de chaque *amplifieur*. Un calcul automatique des boîtes englobantes est tout simplement impossible puisque il faudrait parcourir une infinité d'amplifieur pour s'assurer que rien ne "dépasse". Remarquons à cette occasion un travail récent proposant un calcul automatique de boîtes englobantes hiérarchiques [LH03]. Cette approche est possible grâce au formalisme et aux restrictions propres au langage des IFS. Dans le cadre de Dynamic Graph, l'expressivité du C++ rend complètement impossible ce genre de calcul.

Demander au créateur de spécifier les boîtes englobantes des amplifieurs qu'il modélise peut sembler très contraignant. Mais cela est en partie justifié. Remarquons cependant que les boîtes englobantes ne doivent pas être très précises. Par exemple, des boîtes englobantes deux fois plus grosses que ce qu'elles devraient influencent peu les performances. Ceci est du à l'utilisation de boîte englobante *hiérarchique* : à la fin de la décomposition, les boîtes englobantes sont de toute façon très petites et ne déborderont pas beaucoup du champ de vue.

Le créateur doit fournir différents types d'information sur l'occupation de l'espace :

- une boîte englobante sphérique (un centre est un rayon) ;
- un intervalle $[z_{min}, z_{max}]$;
- un maillage fermé englobant (c'est lui qui servira pour l'algorithme de détection d'occulta-

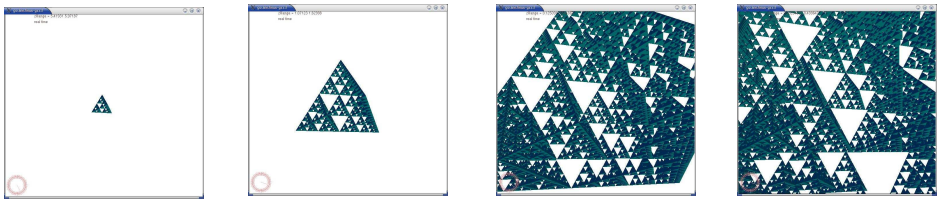


FIG. 7.10 – Un zoom sur la pyramide de Sirpienski.

tion.
Encore une fois, il peut sembler contraignant de demander autant d'information au créateur. Mais dans le cas où celui-ci est paresseux, il peut ne fournir que la boîte englobante sphérique dont seront déduits les deux autres type d'information.

Dans le cas du cube de Sirpienski, seul la boîte englobante sphérique est spécifiée.

Génération des fils

Lorsque la précision est insuffisante, les amplifieurs fils sont générés. Voici à quoi ressemble le corps de la fonction *childGeneration* :

```
void square::childGeneration()
{
    for(int i=0;i<20;i++)
    {
        float newSize = size/3;
        repere newRepere = localRepere*getRepere(i,size);
        square* newChild = new square(i,newSize,newRepere);
        childContainer.addChild(newChild);
    }
}
```

La fonction *getRepere* donne la transformation entre un repère et les 20 repères des amplifieurs fils (cf. figure 7.7).

Résultat

Les figures 7.8, 7.9 et 7.10 montrent quelques captures d'écran. Même si ces modèles ne sont pas animés, les résultats sont plus impressionnants lorsque la caméra est en mouvement. Des vidéos sont disponibles sur le site `http://www-evasion.imag.fr/~Frank.Perbet/these`. Ces modèles ont été réalisé par Sandrine Bard et Benjamin Rouveyrol, deux stagiaires qui débutaient en modélisation 3D, en une semaine. Ils ont aussi réalisé un modèle d'arbre qui sera décrit en 7.3.

7.2.3 Quelques points sensibles

L'étude de ce cas simple cache quelques subtilités dont il est important d'être conscient. Nous parlerons tout d'abord ici de la zone d'influence d'un amplifieur. Nous expliquerons ensuite pourquoi le choix des conteneurs des amplifieurs fils est laissé au créateur. Enfin, nous parlerons des différentes mémoires que Dynamic Graph propose et de leur utilisation dans le cadre de l'optimisation par cohérence temporelle.

Zone d'influence d'un amplifieur

Un amplifieur est une *transformation* de la forme tridimensionnelle qui consiste à l'amplifier (i.e. à ajouter de la précision). Pour chaque amplifieur, on peut déterminer la partie d'un objet qu'il amplifie. La fonction *draw* dessine précisément cette partie amplifiée. De même, la boîte englobante entoure cette partie (ainsi que toute la géométrie pouvant être amplifiée à partir de celle-ci).

On peut imaginer que la forme est marquée au feutre sur l'objet de différentes *zones d'influence* correspondant chacune à un amplifieur. Chaque zone est à son tour marquée par les zones des amplifieurs fils, et ainsi de suite. La figure 7.11 schématise ces zones d'influence en deux dimensions.

Remarquons que dans Dynamic Graph, ces zones d'influences ne sont représentées par aucune structure. Elles n'existent que par le simple fait que les amplifieurs modifient une certaine partie des informations représentant la forme. En fait, malgré le fait que la métaphore visuelle soit plus

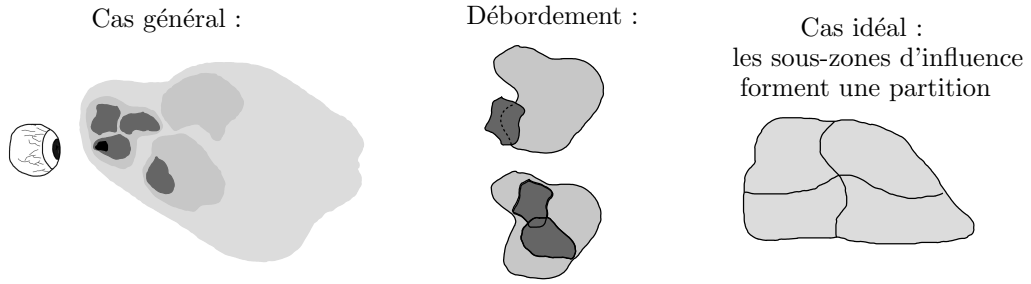


FIG. 7.11 – Lors d'une observation, les amplifieurs enrichissent différentes zones et sous-zones de la forme. Dans Dynamic Graph, rien n'assure le bon positionnement des zones les unes par rapport aux autres. En conséquence, certains comportements maladifs peuvent survenir.

intuitive, il est plus juste de définir une zone d'influence comme l'ensemble des données en mémoire modifiées par un amplifieur.

Lorsque plusieurs amplifieurs modifient les mêmes parties d'un objet, plusieurs zones d'influence peuvent s'intersecter : on parle alors de *zone sous influences multiples*. Ces dernières posent de sérieuses difficultés :

- puisque chaque amplifieur dessine sa zone, les zones sous influences multiples sont dessinées plusieurs fois ;
- rien n'assure que les opérations réalisées par les amplifieurs soient commutatives. En conséquence, la forme sera affichée de façon différente selon l'ordre que choisi l'organiseur (qui dépend, par exemple, du point de vue).

Dans le cas de structures très hiérarchiques tel que le sont tous les modèles présentés dans ce manuscrit, les zones d'influences sont parfaitement disjointes, évitant du coup cette difficulté. Dans le cas de structures continues, il est évident que le problème se pose. On peut par exemple imaginer que la figure 7.11 représente un terrain et que les amplifieurs ajoutent des montagnes.

Une solution consisterait à imposer un ordre d'évaluation, ou à s'assurer de la commutativité des amplifieurs. Mais comment savoir s'il faut afficher, à un instant donné, les parties de l'objet appartenant à la zone d'influence d'un amplifieur ? En effet, peut-être que celle-ci sera modifiée ultérieurement par un autre amplifieur, auquel cas il faut retarder l'affichage.

Toute cette problématique semble mettre en avant un lien étroit entre zone d'influence et boîte englobante (i.e. zone "influençant"). En effet, si une zone d'influence avait connaissance des boîtes englobantes, elle pourrait décider de ne s'afficher que lorsqu'aucune boîte englobante ne l'intersecte (et donc qu'aucun amplifieur ne l'influence). Ceci implique une représentation de la géométrie intelligente et capable de déterminer les zones qu'elle peut afficher sans risque.

Dynamic Graph n'apporte pas de solution à ces problèmes. Plus généralement, il est clair que cet outil est mieux adapté aux structures hiérarchiques. Ceci sera discuté plus en profondeur en 8.1 : nous y verrons comment il est possible de contourner ces limitations.

Conteneur des fils

Rappelons que le conteneur des fils est la structure de donnée qui recense tous les fils d'un amplifieur père. Un amplifieur, lorsque la forme nécessite plus de précision, engendre un certain nombre de sous-amplifieurs pouvant être de type différent. Les amplifieurs enfants sont référencés dans un conteneur de l'amplifieur parent. Ce conteneur est utilisé intensivement par l'*organiseur* lors des parcours de graphe.

Or, de façon générale, aucun conteneur n'est efficace pour stocker n'importe quel *type* de donnée : parfois, un tas sera efficace, d'autres fois une table de hachage, parfois un simple tableau... Dans Dynamic Graph, le problème vient du fait que les amplifieurs peuvent représenter de très nombreuses formes de nature très différente. Un choix générique de conteneur compromettrait donc les performances de l'*organiseur*.

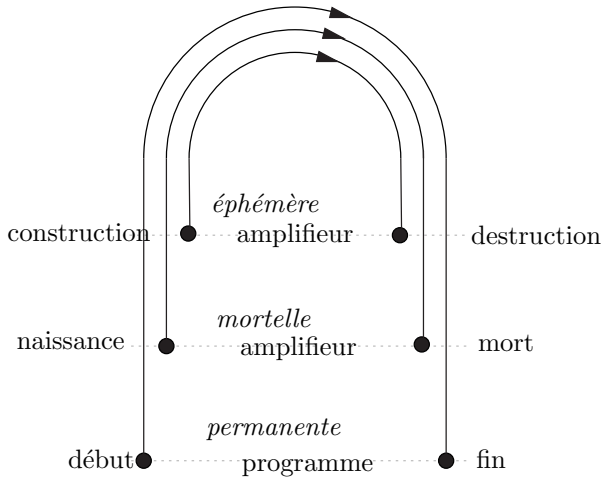


FIG. 7.12 – Plusieurs types de mémoire sont visibles depuis un amplifieur. Chacune d'elles a une durée de vie particulière. La mémoire *permanente* est allouée pour toute la durée de l'exécution de Dynamic Graph. La mémoire *mortelle* est allouée à la naissance d'un amplifieur et désallouée à sa mort. La mémoire *éphémère* est celle stockée dans le tampon d'arbres : elle dure 2 ou 3 pas de temps.

De plus, une fonction de tri est appliquée aux conteneurs à chaque pas de temps afin de réaliser un rendu d'avant en arrière. Ce tri, s'il n'est pas fait avec précaution, peut coûter cher et ralentir notablement les performances. La répartition spatiales des sous-amplifieurs suit souvent des lois connues (cf. figure 4.5 du chapitre 4) qui peuvent être mises à profit pour accélérer le tri.

Ainsi, le créateur doit lui même choisir quel conteneur utiliser parmi ceux proposés par défaut. Éventuellement, il peut en recoder un pour un cas très particulier. C'est ce qui a été fait pour le cube de Sirpienski : la connaissance très particulière de la structure du cube (cf. figure 7.7) a permis de réaliser un conteneur adapté très efficace.

Mémoire et cohérence temporelle

Dynamic Graph propose un mécanisme d'évaluation à la volée. S'il était utilisé naïvement, *toute* la partie visible serait régénérée à chaque pas de temps. Afin de pouvoir réutiliser certaines informations d'un pas de temps sur l'autre, Dynamic Graph propose des mémoires à différente durée de vie (cf. figure 7.12).

Afin de fixer les idées, voici, pour chaque type de mémoire, un exemple d'utilisation :

mémoire permanente : supposons que l'on veuille dessiner l'un des éléments du cube de Sirpienski en rouge, et non en gris. Grâce à l'arbre permanent (cf. sous-section 6.1.3), ceci est possible en ajoutant dans la mémoire permanente associée à un amplifieur une fonction imposant cette caractéristique ;

mémoire mortelle : supposons que la géométrie dessinée par un amplifieur soit parfaitement statique. On peut alors la décrire grâce à un *vertex array*. Celui-ci sera alloué lors de la naissance de l'amplifieur et désalloué lors de sa mort (cf. sous-section 6.1.2) ;

mémoire éphémère : lors d'une animation, le créateur peut utiliser une fonction faisant intervenir des valeurs à des temps passé. C'est le cas, par exemple, d'un calcul d'accélération :

$$a(t) = \frac{x(t) - 2.x(t - \delta t) + x(t - 2.\delta t)}{\delta t^2}$$

La position étant stockée dans chaque amplifieur, le tampon d'arbres (cf. sous-section 6.1.2)

permet ce calcul par la reformulation :

$$a_{courant} = \frac{x - 2.x_{ancètre} + x_{ancètre^2}}{\delta t^2}$$

La bonne utilisation de ces mémoires est essentielle à de bonnes performances lors de l'évaluation et de l'affichage. Ces mémoires sont le support principal de l'utilisation de la cohérence temporelle.

7.3 Arbres animés

Dans cette section, nous présentons l'exemple de modèle le plus abouti réalisé avec Dynamic Graph : un modèle d'arbre animé. Nous présenterons ce modèle non pas comme un travail de recherche, mais comme une validation de Dynamic Graph comme outil d'aide à la modélisation multi-échelle. Nous décrirons donc rapidement un super-cylindre qui est quasiment la seule primitive graphique utilisée dans le modèle. Nous détaillerons ensuite la structure du modèle et les différents amplifieurs qui le constituent. Enfin, nous décrirons brièvement l'animation et montrerons les résultats.

7.3.1 Présentation

Nous dévoilons ici l'idée générale sur laquelle est bâti ce modèle. Nous expliquons ensuite le contexte de cette réalisation afin de mieux valider Dynamic Graph.

Idée générale

Le modèle le plus abouti réalisé avec Dynamic Graph est un arbre. L'arbre réalisé est extrêmement paramétrable. Cela ne signifie pas que toutes les espèces peuvent être modélisées mais nous pensons néanmoins que la structure mise en place peut être adaptée facilement à de nombreux cas particuliers.

Les niveaux de détail reposent sur la structure de l'arbre "biologique" : le tronc, les branches, les sous-branches (cf. figure 7.13). Cette structure a le mérite d'être un excellent codage de la répétitivité : deux générateurs d'amplifieurs sont suffisants : les branches et les feuilles. Les niveaux de détail grossiers sont supportés par les enveloppes des branches : à la place des feuilles contenues par une enveloppe, seule cette dernière est affichée avec une texture "feuillue". L'animation est réalisée de manière procédurale de façon très similaire aux prairies. Des primitives de vent (une brise légère et des bourrasques) dictent le mouvement de l'arbre.

Notons qu'un modèle similaire a été conçu par Lars Mündermann lors de sa thèse sous la direction de Przemyslaw Prusinkiewicz, suite à leurs travaux sur «l'information positionnelle» [PMKL01]. L'implémentation avait été réalisée avec L-Studio[PHM99]. Il est extrêmement intéressant que ce logiciel, initialement conçu pour simuler la *croissance* de plante, ait pu être "détourné" vers la *multi-échelle*. En fait, le lien entre la croissance et la multi-échelle est très fort : on peut imaginer qu'un jeune arbre est une version grossière (mais plus petite) de l'arbre mature. Pour vous en convaincre, je vous renvoie à l'excellent livre «The art of genes» de Enrico Coen [Coe00].

Contexte

Ce travail est présenté non pas comme un travail de recherche, mais comme un produit de Dynamic Graph. Il est donc important de préciser le cadre dans lequel il a été réalisé. Dynamic Graph, en proposant une modélisation très fondamentale (procédurale), n'est pas un outil facile à utiliser et il n'est guère possible de le faire tester en quelques heures par quelques victimes choisies dans l'entourage. J'ai eu la chance d'encadrer deux stagiaires, Sandrine Bard et Benjamin Rouveyrol, qui, durant sept semaines, ont chacun utilisé Dynamic Graph dans le but de modéliser ces arbres agités par le vent. Par ailleurs, ce sont aussi les auteurs des exemples de fractales utilisés en 7.1.

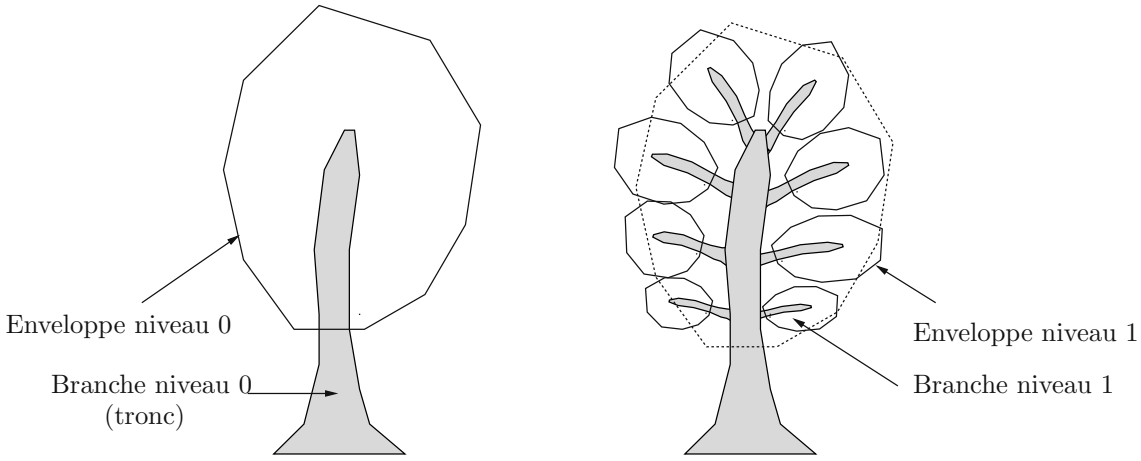


FIG. 7.13 – La structure générale, similaire à [BPF+03], est la suivante : de loin, un arbre est représenté par un tronc et une enveloppe. Lorsque l'observateur se rapproche, la grande enveloppe disparaît pour laisser la place aux branches et à leurs enveloppes.

Ces deux personnes ont réalisé une sorte de validation très informative. Afin de pouvoir mieux valider Dynamic Graph en tant qu'outil générique, voici en quelque sorte le "protocole expérimental" utilisé :

- les deux "créateurs" de cette expérience étaient tout deux en deuxième année d'école d'ingénieur en informatique et avaient donc une certaine expérience de la programmation, notamment en C++ ;
- ils ignoraient, au début du stage, la bibliothèque OpenGL et n'avaient pas suivi de cours d'infographie ;
- mon rôle principal, durant leur stage, était la maintenance de Dynamic Graph : j'ai corrigé les (nombreux) bogues qu'ils découvraient ;
- en conséquence, je ne connais pas une ligne du code qu'ils ont écrit. Bien sûr, nous discutions souvent des rouages du modèle, mais la réalisation, en dehors de l'idée générale décrite plus haut, leur appartient entièrement ;
- durant la première semaine, les modèles fractals ont été réalisés. Les rapports de stage ont été écrits au cours de la dernière semaine. Le temps passé à la modélisation des arbres est donc d'environ cinq semaines.

Vous savez tout, passons maintenant à la description du modèle. Les temps passés sur les différents stades du développement seront chiffrées en semaines cumulées, comme si une seule personne avait travaillé plus longtemps. Avec cette unité, la réalisation des arbres a durée 10 semaines au total (2 fois 5 semaines).

7.3.2 Super-cylindre

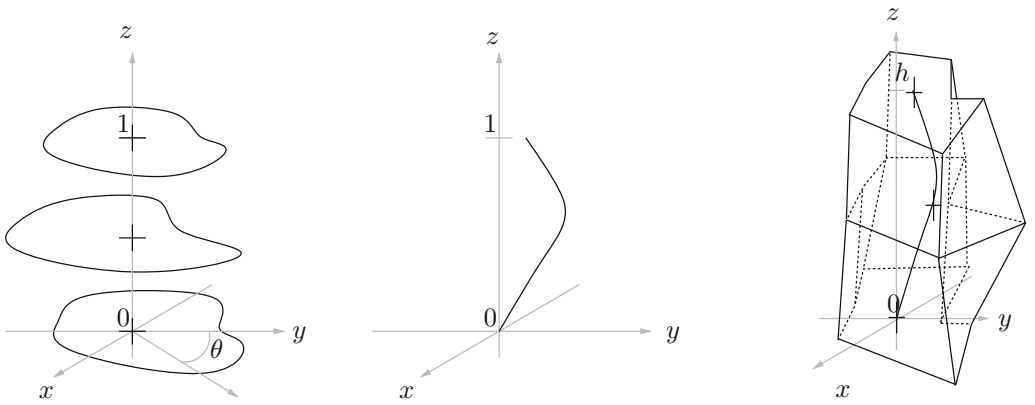
A l'exception des feuilles, les arbres sont affichés via un seul type de primitives géométriques : des cylindres généralisés normalisés, adaptatifs et procéduraux (générés à la volée). Par commodité, nous les appellerons *super-cylindre*. Leur implémentation a pris 6 semaines, soit plus de la moitié du temps.

Un axe et une enveloppe

Un super-cylindre est principalement caractérisé par deux fonctions (cf. figure 7.14) :

axe : $[0,1] \stackrel{axe}{\longmapsto} [0,1] \times [0,1]$ tq $axe(0) = (0,0)$; cette fonction décrit l'axe du cylindre.

enveloppe : $[0,1] \times [0,2\pi] \stackrel{env}{\longmapsto} [0,1]$; cette fonction décrit l'enveloppe du cylindre.



Enveloppe Axe Forme affichée

FIG. 7.14 – Deux fonctions définissent un super-cylindre. La forme s'affiche avec une mise à l'échelle à la volée et un échantillonnage en z de l'axe et en (z, θ) de l'enveloppe.

La seule contrainte posée sur ces fonctions est d'être rapidement calculable (des splines font très bien l'affaire).

Lors d'un affichage, un super-cylindre est généralement mis à l'échelle. Stocker l'information sous une forme normalisée ne change rien conceptuellement. Cela offre néanmoins un moyen pratique pour créer une banque de fonctions facilement réutilisables .

Multi-échelle

Lors de l'affichage, un super-cylindre est échantillonné en triangles puis envoyé sur la carte graphique. Cet échantillonnage est adaptatif et permet de concentrer l'échantillonnage des triangles là où c'est le plus nécessaire (cf. figure 7.15).

Lors d'une mise à jour, la liste d'échantillon est parcourue :

- si deux points sont trop proches, on en supprime un ;
- si deux points sont trop lointains, on en rajoute un.

Des interpolations continues sont réalisées lors de l'ajout ou la suppression d'un nouveau point.

Texture

Lors de l'affichage à la volée d'un super-cylindre, il est possible de lui associer une texture grâce à un plaquage cylindrique. Des textures répétitives en hauteur et en largeur sont utilisées pour éviter les discontinuités.

Un effort particulier a été réalisé pour contrôler le canal Alpha. Celui-ci est représenté par une texture indépendante grâce au *multiTexturing* [WND99]. Nous verrons dans la sous-section suivante comment la transparence que permet la fonction *glAlphaTest* est utilisée pour donner un aspect feuillu.

7.3.3 Structure du modèle

Cette partie décrit principalement le générateur d'amplifieurs principal : la branche. Les générateurs de forêt et de feuille seront aussi décrits, mais leur temps de développement a été négligeable. Le travail présenté dans cette sous-section, décrivant les mécanismes essentiels du modèle, a été réalisé en 3 semaines.

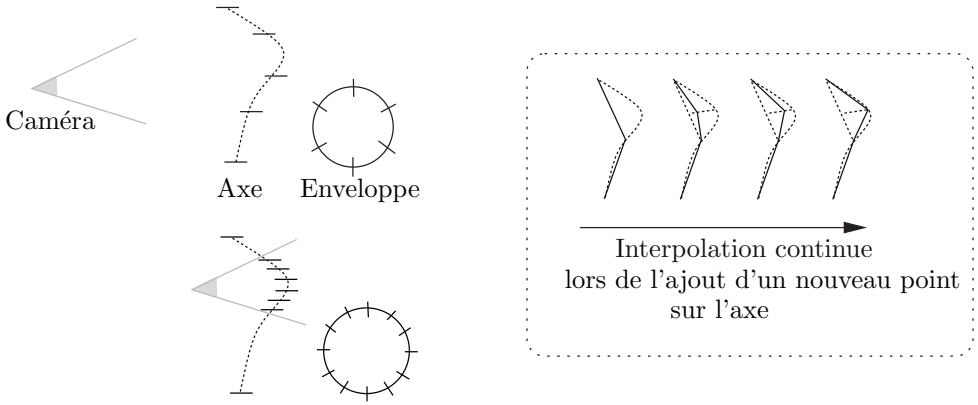


FIG. 7.15 – L'échantillonnage dynamique de l'axe et de l'enveloppe d'un super-cylindre lui permet d'adapter continûment sa précision à l'observation. Cette structure est mise à jour par *modification* : à chaque pas de temps, un parcours de liste permet d'enlever ou de supprimer certains points. Un super-cylindre est stocké dans la mémoire *mortelle*.

Enveloppe et tronc

Un amplifieur peut dessiner une enveloppe et un tronc. Le tronc est dessiné grâce à un super-cylindre texturé. Un amplifieur affiche le tronc quelque soit la précision, même s'il a engendré des amplifieurs fils.

L'enveloppe est dessinée avec un super-cylindre plus large, texturée de façon semi-transparente. La texture de couleur est une texture de bruit évoquant l'apparence d'un feuillage. En jouant sur le seuil Alpha de la fonction *glAlphaTest*, on peut changer la densité de feuilles (cf. figure 7.16) Cette représentation est, à une certaine distance, suffisante pour rendre l'aspect chaotique du feuillage de l'arbre.

Ajout d'un niveau supérieur

Pour passer de l'arbre représenté par un seul amplifieur (tronc et enveloppe) au niveau de détail supérieur, les enveloppes des branches principales sont ajoutées progressivement (cf. figure 7.17).

On peut décomposer la vie d'un amplifieur «branche» en différentes phases (cf. figure 7.18) :

non-existence : lorsque l'observateur est trop loin, l'amplifieur n'existe pas ;

première transition (apparition) : lorsque la précision augmente, l'amplifieur "pousse" verticalement (selon son repère local). Dans le cas du tronc, tout se passe comme si l'arbre sortait du sol. L'amplifieur est initialement constitué d'une enveloppe et d'une branche ;

premièr état stable : dans une certaine plage de précision, l'arbre s'adapte à la précision grâce aux possibilités offerte par les super-cylindres ;

seconde transition (disparition de l'enveloppe) : lorsque la précision augmente encore, l'enveloppe se rassemble sur son axe (homothétie axiale) et disparaît. Le tronc reste.

second état stable : ensuite, seul le tronc est dessiné. Il continue à adapter sa précision grâce aux super-cylindres.

La valse des maturités

La figure 7.17 met en évidence deux types de transitions guidés par deux maturités différentes $m_a(précision)$ et $m_b(précision)$:

- $m_a(précision)$: l'enveloppe principale disparaît en se rapprochant du tronc ;
- $m_b(précision)$: les sous-branches poussent à partir du tronc.

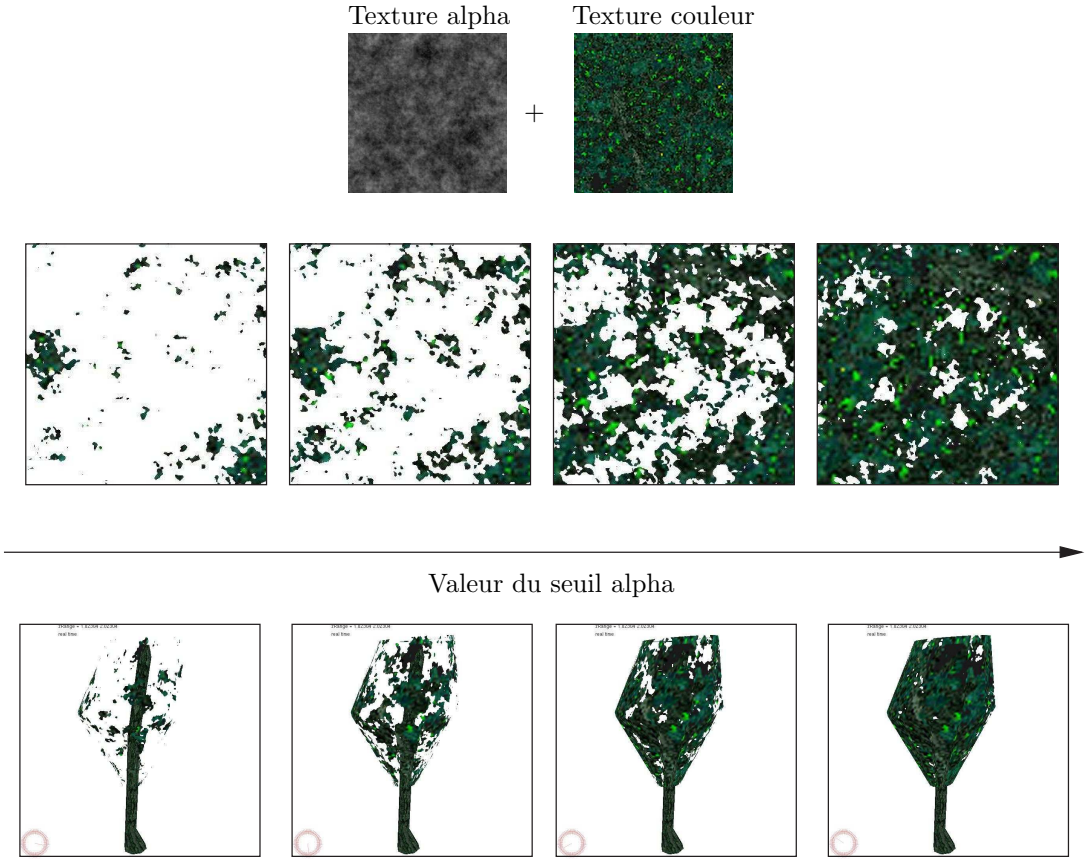


FIG. 7.16 – L'enveloppe est texturée par un image RGB et une image alpha. Le canal Alpha module la densité de feuille dans l'arbre.

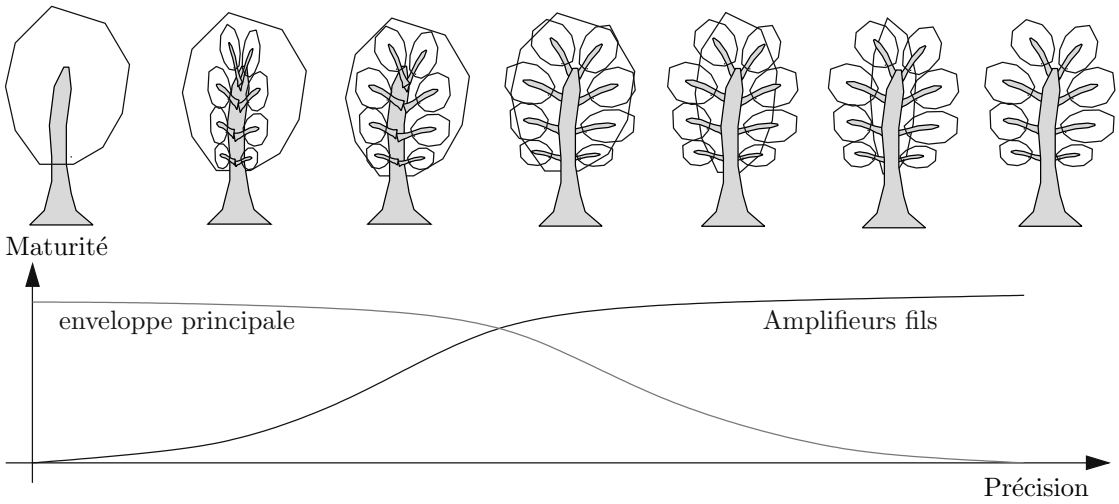


FIG. 7.17 – Transition entre deux niveaux de détail : les sous-amplifieurs apparaissent tandis que l'amplifieur père disparaît. Reqmarquez qu'il aurait été intéressant de réaliser un fondu continue de la transparence au lieu de la transparence "tout ou rien". Mais la fonction `glBlendFunc`, nécessaire pour avoir une transparence continue, est délicat à utiliser car elle nécessite un rendu parfaitement trié des polygones. De plus, même si la transparence utilisée ici est "tout ou rien", elle apparaît et disparaît de façon *spatiallement continue*, ce qui est rend le procédé très acceptable d'un point de vue perceptuel.

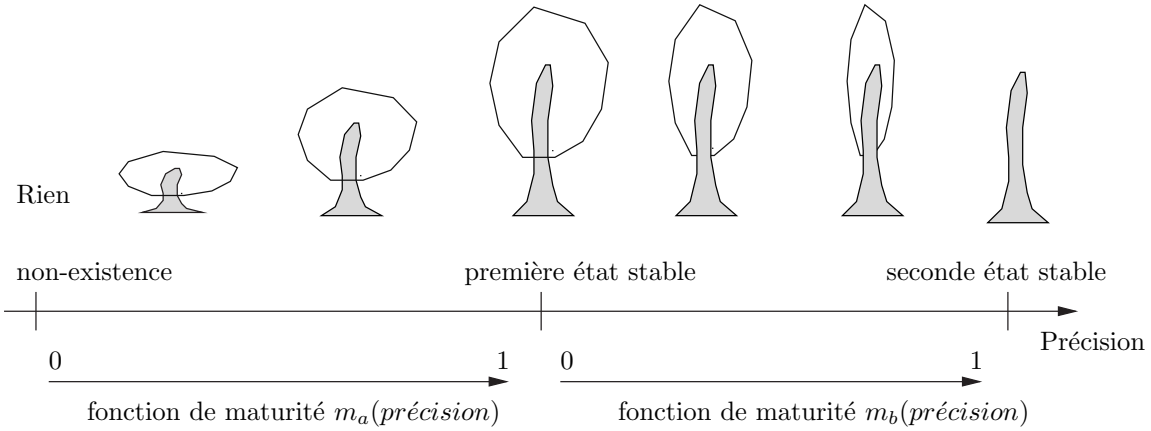


FIG. 7.18 – Différentes étapes d'un amplifieur en fonction de la précision. Deux fonctions de maturité assurent le passage d'un niveau de détail à l'autre.

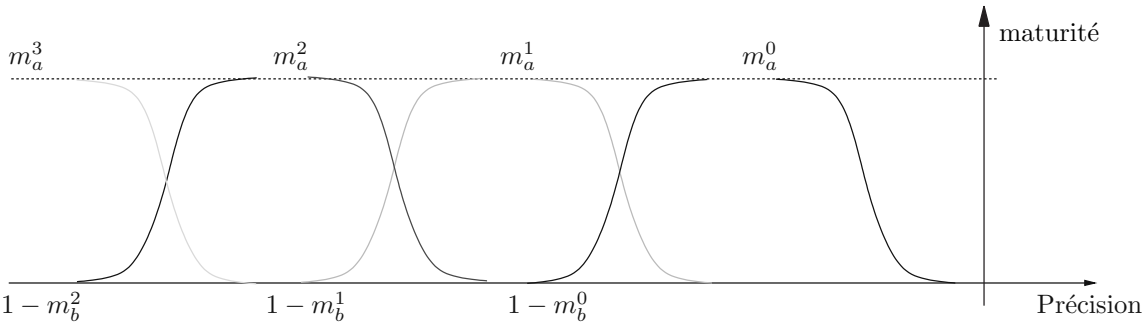


FIG. 7.19 – Les deux maturités sont déclenchées à des précisions bien particulières, assurant une bonne synchronisation entre les amplifieurs père et fils.

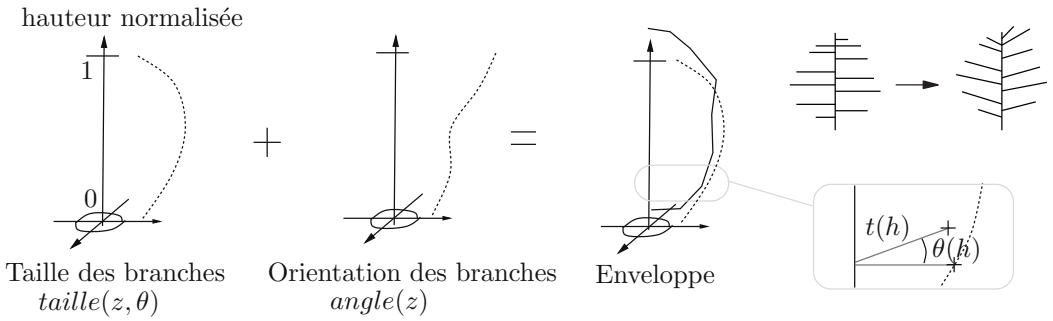


FIG. 7.20 – La fonction enveloppe n'est pas donnée explicitement. Elle est générée à la volée grâce à la connaissance des fonctions $taille(z, \theta)$ et $angle(z)$.

Comme le procédé est récursif sur 3 niveaux, ces deux transitions sont appliquées aux mêmes amplifieurs (cf. figure 7.18). Pour assurer la synchronisation de la disparition d'une enveloppe pendant l'apparition des sous-amplifieurs, on exprime la fonction m_a d'un sous amplifieur en fonction de la fonction m_b de l'amplifieur père : $m_a^{fils} = 1 - m_b^{père}$.

Soit i l'ordre de branchement d'un amplifieur, on nomme m_a^i et m_b^i les maturités des amplifieurs. La figure 7.19 illustre l'évolution de ces différentes maturités en fonction de la précision.

Conservation de l'espace

Les branches d'un arbre sont générées à partir de la fonction d'enveloppe. Lors de la création de nouveaux amplifieurs, de nouvelles fonctions d'enveloppe sont générées aléatoirement. Elles sont cependant déterminées de façon à remplir approximativement l'enveloppe mère (cf. figure 7.13).

Afin de ne pas effectuer de calcul complexe d'intersection entre une branche et l'enveloppe mère pour trouver la taille de la branche, cette dernière n'est pas directement décrite dans l'espace. Une enveloppe est décrite par deux fonctions : $taille(z, \theta)$ et $angle(z)$:

$taille(z, \theta)$ **:** cette fonction décrit la taille d'une branche dont l'origine de l'axe est à une hauteur z sur l'axe du père ;

$angle(z)$ **:** cette fonction décrit l'inclinaison d'une branche dont l'origine de l'axe est à une hauteur z sur l'axe du père ;

Ainsi, il est facile de reconstruire la fonction enveloppe lors d'un échantillonnage à la volée : il suffit de "remonter" le sommet de la branche (déterminer par $taille(z, \theta)$) d'une rotation d'angle $angle(z)$ (cf. figure 7.20).

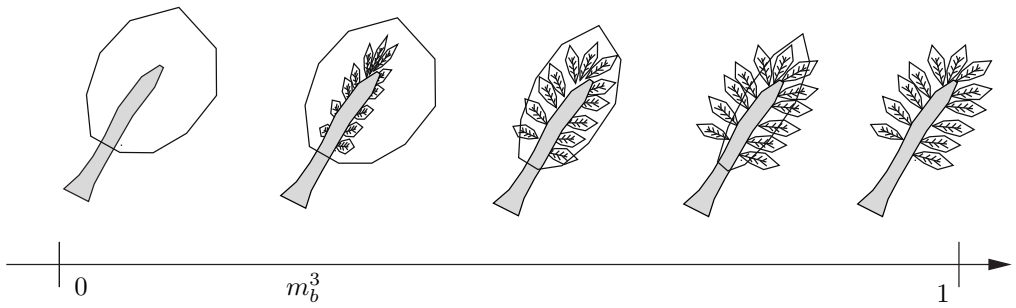


FIG. 7.21 – Les feuilles apparaissent lorsque l'enveloppe des branches d'ordre de branchement maximum disparaissent. D'autres fonctions d'ajout de précision plus sophistiquées aurait pu être utilisées. Néanmoins, les fonctions utilisées ici ont le mérite d'être très simple a implémenté et tout à fait continue. D'un point de vue perceptuel, combinées avec le mouvement de l'arbre, elles sont d'une qualité convenable.

Générateur d'amplifieurs : Feuilles

Les amplifieurs «feuille» reprennent toutes les fonctionnalités des amplifieurs «branche». Concrètement, la classe décrivant les feuilles hérite de celle décrivant les branches.

La seule différence est la suivante : lors de la seconde transition (lorsque l'enveloppe disparaît), les feuilles attachées aux troncs apparaissent (cf. figure 7.21). Les feuilles n'ont donc pas d'amplifieur individuel : elles sont simplement ajoutées aux branches les plus fines.

Une feuille est affichée par de simples polygones colorés. C'est ici que s'arrête l'amplification : il faudrait ensuite ajouter les nervures, affiner sa silhouette, et éventuellement faire apparaître les cellules...

Générateur d'amplifieurs : Forêt

Une forêt est un amplifieur qui appelle les amplifieurs «branche» plantés verticalement et répartis sur un terrain. Naturellement, chaque arbre est paramétré de façon subtilement différente de façon à briser toute impression de répétitivité. La taille, le nombre de branches, les fonctions d'enveloppe, la couleur : tous ces paramètres peuvent être bruités par des fonctions pseudo-aléatoires. On pourrait appeler ce procédé l'*amplification stochastique*.

Pour pouvoir reculer encore et voir la forêt de très loin, il faudrait imaginer une façon de l'afficher comme un seul "individu", un peu comme le niveau 2D de la prairie (cf. chapitre 4). Cela reste à faire.

7.3.4 Animation

Les fonctions d'animation ont été réalisées en une semaine. Elles reprennent le même concept que celui utilisé pour les prairies. Les récepteurs sont ici aussi la courbure et la direction de courbure de chaque branche.

Brise légère

Une petit mouvement aléatoire change la courbure et la direction de chaque branche à tous les niveaux. L'animation est conçue de façon complètement procédurale. Le but des deux utilisateurs de Dynamic Graph était de trouver une fonction mimant de façon acceptable le mouvement des arbres.

Le résultat est à mon goût très satisfaisant. Les résultats montrent encore une fois que l'œil est sensible à la complexité du mouvement mais qu'il nous est difficile de déterminer si une animation est réaliste ou non. Comme pour les prairies, c'est la *plausibilité* qui compte ici.

Bourrasque

De même que pour les prairies, une bourrasque impose une direction et une courbure à chaque branche de l'arbre. La seul différence vient du fait que l'orientation du repère tridimensionnel change entre chaque branche. Il est donc nécessaire d'appliquer une rotation pour déterminer la bonne courbure.

De même que pour les brins d'herbe, la taille de chaque branche est modulée *à la main* pour simuler une taille constante.

7.3.5 Résultats

Nous présenterons tout d'abord le modèle lui-même. Nous décrirons ensuite dans quelle mesure ces stages valident Dynamic Graph en tant qu'outil d'aide à la modélisation multi-échelle.

Le modèle

Le modèle prend tout son ampleur lorsqu'il est animé. Je vous convie à télécharger les vidéos sur le site de l'équipe Évasion `http://www-evasion.imag.fr/~Frank.Perbet/dg`. La figure 7.22 montre quelques captures d'écran d'un arbre à différentes précisions. La figure 7.23 est une forêt très simple.

Le coefficient *Qualité* × *Rapidité* du modèle est acceptable. Néanmoins, pour un affichage temps-réel, il faut vraiment sacrifier la qualité. Durant la création de ce modèle, le soucis de performance n'était pas prioritaire. Il me semble qu'un facteur d'approximativement 10 peut être gagné par une seconde implémentation plus soucieuse de l'efficacité.

Validation de Dynamic Graph

Lors de l'utilisation de Dynamic Graph durant ce stage, certains points intéressants ont été soulevés par les deux étudiants. Tout d'abord, les différents outils d'aide à la modélisation se sont révélées globalement très utiles :

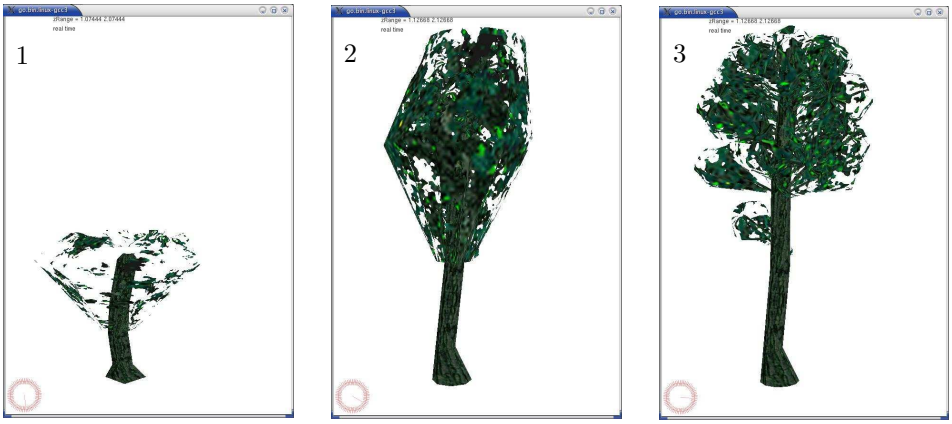
- notamment, le *debugPanel* a été utilisé de façon intensive ;
- la *seconde observation* a permis de voir facilement les éventuels problèmes de transitions ;
- les micro-interfaces ont été d'une aide précieuse pour déboguer ;
- le mode pas-à-pas a quelque fois servi à détecter l'origine d'erreurs intervenant au cours de la création de l'arbre ;
- l'arbre d'évaluation a permis de détecter la création de nouveaux amplifieurs (celle-ci étant indiscernable au niveau de l'affichage).

En revanche, les mesures d'occupation mémoire et de temps de calcul n'ont pas été utilisées. Il est probable que ces outils soient plus utiles dans le cas de scènes plus complexes comprenant plusieurs amplifieurs.

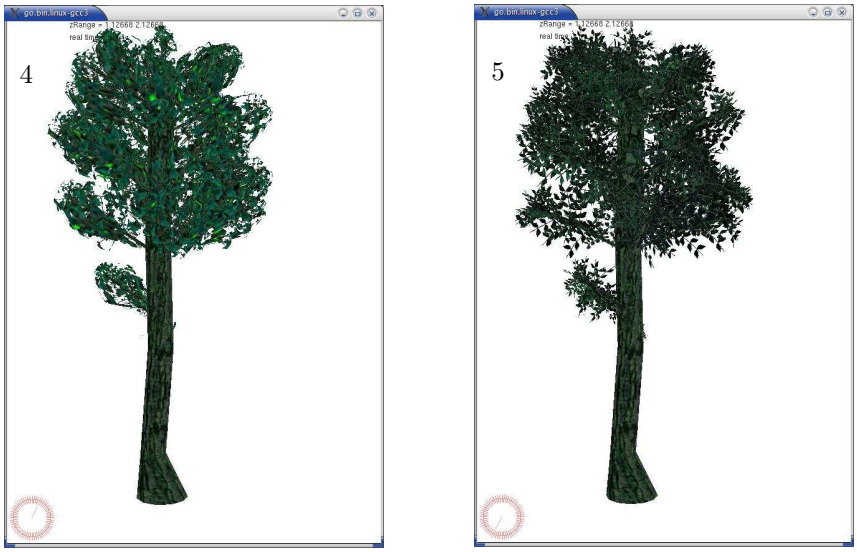
Ce stage a aussi été l'occasion de changer un peu Dynamic Graph lui-même. Mis à part les innombrables bogues, ces deux mois ont beaucoup enrichi le dialogue entre Dynamic Graph et le modèle. Avant le début du stage, la seule fonction d'échange (cf. sous-section 5.2.1) était la fonction de création de l'axiome. Très vite, de nouvelles fonctions ce sont révélées nécessaires. Par exemple, des fonctions d'initialisation ont permis de rajouter automatiquement de nouveaux curseurs dans le *debugPanel*. Ainsi, lorsque l'on charge un modèle, on charge automatiquement les curseurs qui sont associés à ces paramètres. Généralement, Dynamic Graph a beaucoup profité du retour utilisateur durant ces deux mois.

Dynamic Graph : un imposteur

Un phénomène curieux s'est produit lors de la conception du modèle d'arbre. Les utilisateurs étaient comme attirés par la précision la plus fine. Une fois les feuilles créées, les niveaux de détail intermédiaires ont été négligés : le plus gros du travail passé dans la modélisation du niveau le plus fin. Je vois différentes explications à ce phénomène :



123*polygones* 159*polygones* 412*polygones*



3257*polygones* 15498*polygones*

FIG. 7.22 – La précision de cet arbre varie, à titre illustratif, de façon uniforme et indépendamment du point de vue. On remarque ainsi les différents niveaux de détail : trois niveaux d'enveloppes imbriquées laissent ensuite place aux feuilles. Lorsque l'arbre est très loin, il s'enterre dans le sol : c'est le terrain qui doit alors être texturé de façon à représenter une forêt.

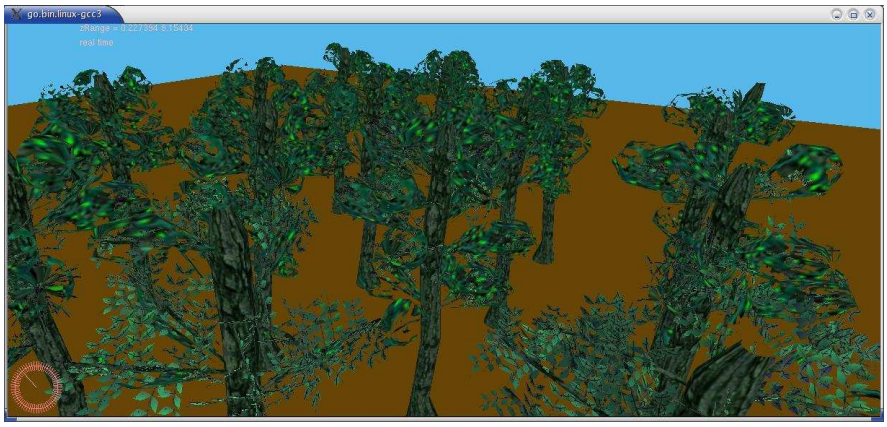


FIG. 7.23 – Plusieurs arbres à différentes précisions.

– tout d'abord, il est agréable de travailler avec le modèle plein écran. Mais les niveaux de détail grossiers sont laids vus de près. Il est quelque sorte plus valorisant d'afficher la précision maximum ;
– d'autre part, il semble difficile d'accorder beaucoup de crédibilité à des représentations grossières. Modéliser avec Dynamic Graph, c'est un peu l'art de trouver le bon camouflage. Ce travail d'impressionniste semble poser quelques difficultés.

Dynamic Graph est un outil permettant de *mimer* le réel. À plusieurs reprise, j'ai insisté pour ne pas utiliser de photo, mais des textures réalisées à la main. Lors de l'animation, il a fallu imposer une modélisation procédurale car les utilisateurs étaient séduits par des simulations physiques qu'il était impossible de réaliser en si peu de temps. Le réalisme, ainsi que "le niveau le plus fin", semble attirer notre concentration. Dynamic Graph est à ce titre assez dérangeant car il renvoie à la synthèse d'image une identité que, peut-être, elle ne s'avoue pas : celle d'un imposteur.

Chapitre 8

Conclusion

8.1 Discussion

8.1.1 Une modélisation par entité

Un langage descriptif “constructif”

Dynamic Graph propose un langage décrivant la *construction* d'un modèle. Ce type de description permet notamment une représentation très compacte. On peut prendre ici la métaphore du corps humain : représenter naïvement sa structure prend une place mémoire énorme. En revanche, on peut stocker une simple cellule souche et simuler la construction du corps. La molécule d'ADN est en quelque sorte une compression du corps humain.

Les générateurs d'amplifieurs jouent un rôle similaire aux règles de construction de la molécule ADN. Le modèle est le plus possible réduit à sa paramétrisation minimale. On peut voir la modélisation avec Dynamic Graph comme une compression manuelle dans une base de fonctions extrêmement variées.

Structure continue

Les règles de construction imposées par Dynamic Graph sont bien particulières : les amplifieurs sont stockés dans un *arbre* d'évaluation. Ceci impose des relations de dépendance très strictes : un amplifieur n'a qu'un seul parent. La précision est ajoutée de façon élémentaire : par exemple, les éléments «branches» sont ajoutés à l'élément «enveloppe». Dans le cas du modèle d'arbre présenté dans le chapitre précédent, cette décomposition est assez simple. Dans d'autre cas de nature plus continue, c'est moins évident.

Dans le modèle d'arbre présenté dans la section 7.3, le tronc est une structure continue unidimensionnelle. La gestion de la précision d'une telle structure n'est pas prise en charge par Dynamic Graph : ce sont les super-cylindres qui gèrent cela de façon autonome. Le problème serait le même pour un terrain : Dynamic Graph est incapable de faire mieux que les algorithmes existants.

Dans le cas de structures continues, Dynamic Graph n'est donc pas très adapté. En revanche, rien n'empêche d'intégrer des algorithmes dédiés (de simplification par exemple) au sein d'un amplifieur. Pour pouvoir utiliser l'algorithme de visibilité de Dynamic Graph, il faut cependant que ces algorithmes permettent d'afficher la structure en plusieurs étapes, de l'avant vers l'arrière.

Entité distincte

Même s'il est possible d'intégrer localement des structures continues, Dynamic Graph impose une sorte de décomposition stricte. Il est envisageable d'assouplir les relations de dépendance. Par exemple, on pourrait imaginer non plus un arbre d'évaluation, mais un graphe sans cycle : un amplifieur aurait plusieurs parents.

Néanmoins, le cœur de Dynamic Graph se base sur une entité très autonome : l'amplifieur. Cette approche va complètement à l'encontre d'un univers régi par des particules élémentaires et des lois simples. La modélisation avec Dynamic Graph consiste initialement à identifier des phénomènes et à les représenter individuellement. Par construction, les mondes de Dynamic Graph sont très peu interconnectés : un arbre d'une forêt ignorera tout de ses voisins.

Cette *localisation* des phénomènes me semble être le prix à payer pour un algorithme efficace.

8.1.2 Trop de choses à la fois

Comme tout outil, Dynamic Graph impose une certaine conception de la modélisation. C'est en quelque sorte le squelette d'un objet qu'il faut saisir, et non son apparence.

Deux sortes d'animations

Modéliser l'ajout de précision est finalement très similaire à la modélisation de l'animation. Le "mouvement" multi-échelle est induit par les mouvements de la caméra, alors que le "mouvement" de l'animation est induit par l'écoulement du temps. Mais conceptuellement, il s'agit dans les deux cas de paramétrer convenablement le modèle afin d'exhiber les degrés de liberté pertinents. Le créateur doit donc imaginer un mécanisme capable de deux mouvements simultanément :

- le mouvement induit par le temps ;
- le mouvement induit par l'échelle d'observation.

Est-ce que cette double quête est tout le temps possible ? Cela n'a rien d'évident au premier abord. Les mécanismes nécessaires à l'animation peuvent très bien aller à l'encontre de ceux nécessaires aux changements d'échelle. Heureusement, dans ces deux types de mouvement, une certaine marge de manoeuvre est permise.

L'animation influence la multi-échelle

Il n'existe pas qu'une seule façon d'ajouter de la précision à un objet. Par exemple, pour les arbres, on aurait pu choisir de représenter le feuillage dans son ensemble et indépendamment des branches. Mais il aurait été alors difficile de rendre compte du mouvement induit par les branches. En revanche, il aurait été peut-être plus facile de réaliser des opérations globales telles que des feuilles qui tombent en automne.

Il est clair que le choix des niveaux de détail influence les possibilités d'animation. Selon les besoins, on peut préférer une représentation particulière. On regrettera éventuellement qu'un modèle ne soit pas plus polyvalent. Il me semble que c'est le prix à payer de toute modélisation par complexification : le créateur, avant de commencer un modèle, doit déjà plus ou moins savoir les traitements qu'il veut lui faire subir. En conséquence, il choisira les niveaux de détail adéquats aux besoins de l'animation.

La multi-échelle influence l'animation

Les mécanismes d'animation modélisés avec Dynamic Graph ne cherchent pas le réalisme à tout prix. Une approche phénoménologique permet de mimer un certain nombre de comportements *dans la mesure du possible*. Pour éviter lorsqu'on le peut une description au niveau le plus fin, des phénomènes émergents sont identifiés et décrits globalement (comme les bourrasques de vent).

Cette approche simpliste permet d'animer des modèles complexes. Par exemple, les bourrasques de vent de la prairie animent de la même façon les niveaux 2.5D et 3D. Après la conception du modèle inanimé, on récupère en quelque sorte les degrés de liberté : il est facile "d'en faire quelque chose". En conséquence, le créateur choisira les méthodes d'animation en fonction des niveaux de détail du modèle.

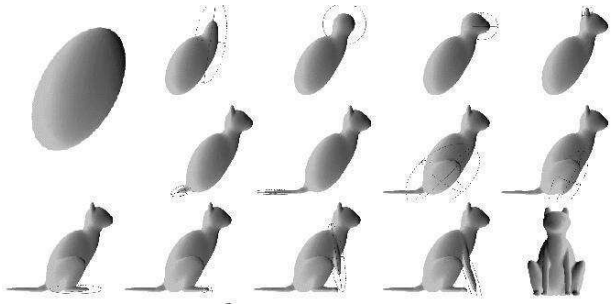


FIG. 8.1 – Exemple de modélisation dont les opérations pourraient très bien être enregistrées puis rejouées selon la précision. Remarquez que ce modèle n'a pas été réalisé spécialement pour l'occasion : c'est la preuve que dans certain cas, une modélisation multi-échelle est naturelle.

Le juste milieu

Le rôle du créateur est donc de jouer entre les fonction d'animation et de transition pour arriver à ses fins. Il n'y a pas de règle précise ici, et c'est certainement là le point commun à tout acte de création. Néanmoins, la marge de manœuvre est peut-être finalement assez étroite. Essayez de trouver un autre modèle de prairie animé sous le vent... La démarche, au bout du compte, est très naturelle.

8.1.3 Une création délicate

Trop difficile ?

En décrivant un modèle par des générateurs d'amplifieurs, celui-ci est décrit par une méthode de construction bien particulière : la précision est progessivement ajoutée au modèle. N'est-ce pas trop demander au créateur que de lui imposer une telle conception de l'objet ?

Tout d'abord, il est évident que Dynamic Graph peut encore être énormément amélioré. Notamment, une interface graphique augmenterait beaucoup l'ergonomie du programme. Malheureusement, je vois mal à quoi peut ressembler une interface graphique assurant une expressivité aussi grande que celle requise par Dynamic Graph. Mais mettons de côté cette implémentation particulière et abordons la question du point de vue général de la complexification.

Quelques exemples

Enrico Coen [Coe00] prend l'exemple d'un peintre qui réalisait ses oeuvres en commençant par une version très grossière et en ajoutant progressivement de la précision. La figure 8.1 montre les différentes étapes de la modélisation d'un chat réalisé dans le cadre du travail de recherche de Philippe Decaudin [Dec96]. Ces exemples sont bien entendu en faveur de l'hypothèse d'une conception par complexification facile.

Néanmoins, soyons honnêtes, ces exemples ne sont pas la majorité. J'ai un jour demandé à un utilisateur de Maya expérimenté s'il modélisait habituellement du plus grossier au plus fin. Par chance, il était en train de créer un personnage et avait gardé la liste de toutes les opérations depuis le début du modèle. Quand il les a rejouées devant moi, j'ai beaucoup douté de ma thèse... Le modèle semblait commencer par le pied droit et remonter la jambe, puis le corps et les bras et enfin la tête.

Ici, de nombreux arguments peuvent étayer le débat. Tout d'abord, un utilisateur, dans une certaine mesure, s'adapte à un outil. Ensuite, il est tout simplement possible d'imposer une certaine "discipline" au créateur en forçant une modélisation du plus grossier au plus fin. En effet, pourquoi ne pas limiter les mouvements de caméra : le créateur ne s'autorise à zoomer que s'il est satisfait de lui à l'échelle courante.

Ordre des opérations

Enfin, on pourrait envisager des méthodes de "remise en ordre" : les opérations que réalise le créateur seraient triées de la plus grossière à la plus précise automatiquement. Mais il semble très difficile de briser des relations de dépendance pour les relier de manières différentes.

La seule façon de faire cela à mon avis serait d'utiliser des opérations plus simple et d'imposer un certaine certaine représentation visuelle, tel que le maillage. Mais cela va à l'encontre de la grande force de la modélisation procédurale : sa grande expressivité. On est encore une fois confronté au dilemne entre expressivité et automatisme. Ces deux caractéristiques semblent être quasi-antinomique.

8.2 Perspectives

Modélisation par complexification

La modélisation par complexification porte bien son nom : d'une part elle propose une description de forme visible par ajout de détails (la forme se complique), d'autre part elle utilise des langages descriptifs procéduraux, c'est-à-dire complexes. L'utilisation de langages très expressifs pour décrire des scènes tridimensionnelles permet des variations d'échelles inégalables. De plus, et c'est peut-être le plus important, l'animation est permise.

Mais il ne faut pas se voiler la face : la modélisation par complexification, et la modélisation procédurale en général, demandent beaucoup d'efforts de la part de l'utilisateur. Dynamic Graph propose de simplifier cette tâche et dans l'ensemble, il y parvient. Les performances, les modèles obtenus et les retours utilisateurs de deux intervenants extérieurs amènent un bilan dans l'ensemble très positif.

Vers une interface graphique

Néanmoins, l'interface reste essentiellement textuelle. La question se pose : est-il possible de créer une interface graphique intuitive pour assister la modélisation procédurale ? J'en suis convaincu, mais le travail est colossal. En fait, compte tenu de l'expressivité nécessaire à une telle interface, sa réalisation est équivalente à celle d'un environnement graphique pour le développement. Ceci entraîne une nouvelle interrogation : est-il possible de supprimer (ou en tous cas de réduire) le texte des environnements de programmation ?

Dans bien des cas, cette question revient à se demander s'il est possible de supprimer le texte de votre éditeur de texte favori... Pourtant, le langage textuel n'est pas inhérent à la programmation, il n'est qu'un moyen de permettre à notre intelligence de s'exprimer et de la pérenniser au sein d'un circuit électronique. D'autres moyens sont possibles, ils restent à inventer.

Trop ambitieux ?

L'avenir de la modélisation par complexification est incertaine : son utilisation prend à revers la plupart des méthodes traditionnelles. Notamment, ne pas connaître la scène a priori peut décourager de nombreux utilisateurs potentiels. Pourtant, j'ai la conviction que Dynamic Graph montre la bonne direction : la modélisation et le rendu de scènes tridimensionnelles animées avec de grandes variations d'échelle doivent forcément passer par une modélisation procédurale multi-échelle.

Néanmoins, même s'il montre la bonne direction, Dynamic Graph me semble au bout du compte assez prétentieux. En effet, pourquoi s'attaquer au problème posé par la multi-échelle animée tridimensionnelle alors que le cas 2D est largement suffisant pour dégager les principales difficultés ? J'irai encore plus loin : la réalisation de Dynamic Graph et sa mise en pratique m'ont convaincu que beaucoup de travail reste à faire dans le cas unidimensionnel !

Remarquons ici qu'un programme est finalement représenté par un tableau $1D$ de caractères[1]. Une application évidente d'une étude de la multi-échelle procédurale dans le cas $1D$ est une

[1] Un programme est généralement visualisé, il est vrai, en deux dimensions.

interface graphique pour un environnement de développement. Par deux chemins différents, nous retombons sur la même conclusion.

Je pense qu'une telle réalisation compte parmi les défis scientifiques majeurs actuels et qu'elle sera la motivation d'un pont essentiel entre la synthèse d'image et l'interaction homme-machine.

Annexe A

Exemple du cube de Sirpienski

Afin d'avoir une idée précise de ce à quoi peut ressembler un modèle réalisé avec Dynamic Graph, voici le code du cube de Sirpienski. Pour plus de renseignement sur cet outil, je vous renvoie à la documentation en ligne sur `http://www-evasion.imag.fr/~Frank.Perbet/these`.

A.1 Le générateur d'amplifieurs

Voici le générateur d'amplifieurs que le créateur doit écrire. Une fois la phase de création ou de modification terminé, ce bout de code est compilé et chargé à la volée durant l'éxécution de Dynamic Graph.

```
class ty_square:
  public ty_drawableNodeBase
{
public:
  // square size
  ty_float size;
  // branching order
  ty_uint branchingOrder;
  // color
  vec3 color;

public:

  // constructor without ancestor
  ty_square::ty_square(
    ty_pt_nodeBase pt_father,
    ty_localID li,
    const ty_uint& _branchingOrder,
    const ty_float& _size,
    const vec3& _color,
    ty_repere initialRepere):
      ty_drawableNodeBase(pt_father,li,createDefaultNodeContainer(),
          initialRepere,ty_sphericalBB(_size)),
      branchingOrder(_branchingOrder),
      color(_color),
      size(_size)
  {
    // Nothing
```

```
  }

  // constructor with ancestor
  ty_square::ty_square(
    ty_pt_nodeBase pt_father,
    ty_localID li,
    ty_nodeBase& _ancestor,
    const ty_uint& _branchingOrder,
    const ty_float& _size,
    const vec3& _color,
    ty_repere initialRepere):
      ty_drawableNodeBase(pt_father,li,createDefaultNodeContainer(),_ancestor,
          initialRepere,ty_sphericalBB(_size)),
      branchingOrder(_branchingOrder),
      color(_color),
      size(_size)
  {
    // Nothing
  }


  // visitor: generation
  virtual void upDown(parserFunction::ty_generation& v)
  {
    setMaturity(v.getNodeInfo());

    if(branchingOrder>0)
    {
      ty_float reductionCoef = 1.0;
      ty_float coef = reductionCoef+(1.0-reductionCoef)*(1.0-maturity());
      size*=coef;
    }

    if(precision()>0)
    {
      // create child node
      childGeneration(v);
    }
    else
    {
      draw();
    }
  }

protected:

  // creation of child node
  void childGeneration(parserFunction::ty_generation& v)
  {
    ty_ncRedirection& childContainer =
        static_cast<ty_ncRedirection&>(container());

    static boost::array<vec3,8> dec = {{
      vec3(1.0,1.0,1.0),
```

```
    vec3(1.0,1.0,-1.0),
    vec3(1.0,-1.0,1.0),
    vec3(1.0,-1.0,-1.0),
    vec3(-1.0,1.0,1.0),
    vec3(-1.0,1.0,-1.0),
    vec3(-1.0,-1.0,1.0),
    vec3(-1.0,-1.0,-1.0) }};

  ty_tmpChildList tmpList = childContainer.startGeneration();
  for(int i=0;i<8;i++)
  {
    vec3 newPosition = dec[i]*(size/4.0);
    ty_float newSize = size/2.0;
    vec3 newColor = color;

    ty_pt_nodeBaseBase it = childAncestor(i);
    if(it.isValid())
    {
      ty_nodeBase& a = static_cast<ty_nodeBase&>(it.ref());
      ty_nodeBase* newChild;
      NEW(ty_square,newChild)
          (this,i,a,branchingOrder+1,newSize,
          newColor,globalRepere*ty_repere(newPosition));
      childContainer.addChild(tmpList,newChild);
    }
    else
    {
      ty_nodeBase* newChild;
      NEW(ty_square,newChild)
        (this,i,branchingOrder+1,newSize,
        newColor,globalRepere*ty_repere(newPosition));
      childContainer.addChild(tmpList,newChild);
    }
  }
  childContainer.endGeneration(tmpList);
}


// opengl draw function
void drawCore()
{
  glPushMatrix();
  glMultMatrix(globalRepere);
  glColor(color);
  draw_cube(1,0xFF,get_rapport(),size);
  glPopMatrix();
}


/// opengl draw function
void drawDebug(ty_glViewer& v)
{
  pixel vpSize = v.mainViewportSize();
```

```
    ty_float pos=-30;
    v.strBuf()<<"size = "<<size<<std::endl;
    v.drawText(10,vpSize[1]-(pos+=30));
    v.strBuf()<<"branchingOrder = "<<branchingOrder;
    v.drawText(10,vpSize[1]-(pos+=30));
    v.strBuf()<<"precision = "<<ty_maturity::precision();
    v.drawText(10,vpSize[1]-(pos+=30));
    v.strBuf()<<"maturity = "<<ty_maturity::maturity();
    v.drawText(10,vpSize[1]-(pos+=30));
  }


}; // end of ty_square
```

A.2 Fonctions d'échange

Les fonctions d'échanges sont les fonctions que Dynamic Graph va explicitement chercher dans la librairie avec la fonction *dlsym* (dans le fichier d'entête standard *dlfcn.h*). Parmis ces fonctions, ont remarquera celle qui se charge de la création du premier amplifieur, racine de l'arbre d'évaluation.

```
extern "C"
{
  typedef boostEXT::te_array<GLfloat,4> glvec4;
  typedef boostEXT::te_array<GLfloat,3> glvec3;


  // called one time when loading the model
  void init()
  {
    dpGV::initSlider(7,1,"Rapport",333,1000);
    dpGV::initSlider(7,2,"GL_SPOT_CUTOFF",200,1000);
    dpGV::initSlider(7,3,"GL_SPOT_EXPONENT",145,1000);
    dpGV::initSlider(7,4,"GL_AMBIENT",380,1000);
    dpGV::initSlider(7,5,"GL_DIFFUSE",500,1000);

    glPushMatrix();
    glLoadIdentity(); // on est dans le repere de la camera

    glvec4 light_ambiant = glvec4(
       4*dpGV::sliderF(7,4),
       4*dpGV::sliderF(7,4),
       4*dpGV::sliderF(7,4),1);
    glvec4 light_diffuse = glvec4(
      4*dpGV::sliderF(7,5),
      4*dpGV::sliderF(7,5),
      4*dpGV::sliderF(7,5),1);

    glLightfv(GL_LIGHT1, GL_POSITION, glvec4(0,0,0,1).c_array());
    glLight(GL_LIGHT1, GL_SPOT_DIRECTION, glvec4(0,0,-1,1).c_array());
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambiant.c_array());
```

```
  glLightfv(GL_LIGHT1, GL_DIFFUSE, light_diffuse.c_array());

  glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 180*dpGV::sliderF(7,2));
  glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 128*dpGV::sliderF(7,3));

  glPopMatrix();
} // init


// called one time when unloading the model
void kill()
{
  // Nothing
} // kill


// called one time before each tree generation
void initFrame()
{
  glEnable(GL_LIGHT1);
} // initFrame


// call one time after each tree generation
void killFrame()
{
  // nothing
} // killFrame


// create the root node (the axiom)
void getStartInfo(ty_tree& tree)
{
  // color
  vec3 color =
    vec3(dpGV::sliderF(3,1),dpGV::sliderF(3,2),dpGV::sliderF(3,3));

  // size and position of the first square
  ty_float size = 1;
  vec3 position = vec3(0,0,0);

  // initial repere:
  ty_repere ri = tree.initialRepere;

  // construct the axion
  if(tree.isAncestorAxiomAlive())
  {
    ty_nodeBase& ancestor = static_cast<ty_nodeBase&>(tree.ancestorAxiom());
    NEW(ty_square,tree.ptAxiom())
      (ty_pt_nodeBase(),0,ancestor,0,size,color,ri*position);
  }
  else
  {
    NEW(ty_square,tree.ptAxiom())
```

```
        (ty_pt_nodeBase(),0,0,size,color,ri*position);
    }
  }
}
```

Bibliographie

[ACW+99] Daniel Aliaga, Jonathan Cohen, Andrew Wilson, Hansong Zhang, Carl Erikson, Kenneth Hoff, T. Hudson, Wolfgang Stürzlinger, Eric Baker, Rui Bastos, Mary Whitton, Frederick Brooks, and Dinesh Manocha. MMR : An interactive massive model rendering system using geometric and image-based acceleration. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 199–206, April 1999.

[Ali03] Alias|Wavefront. *Maya API White Paper*, May 2003. `http://www.alias.com/eng/products-services/maya/file/maya5_apiwhitepaper.pdf`

[Bae03] Xavier Baele. *Génération et rendu 3D temps réel d'arbres botaniques*. PhD thesis, Université libre de Bruxelles, 2003.

[BCL+99] Pierre Berlioux, Marie-Paule Cani, Augustin Lux, Roger Mohr, Denis Naddef, and Jean-Louis Roch. Algorithmique et recherche opérationnelle, 1999. Cours ENSIMAG 2ème année.

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. The GPU as numerical simulation engine. In *Proceedings of SIGGRAPH 2003",*. ACM Press, 2003.

[BKCN03] Florence Bertails, Tae-Yong Kim, Marie-Paule Cani, and Ulrich Neumann. Adaptive wisp tree - a multiresolution control structure for simulating dynamic clustering in hair motion. *Symposium on Computer Animation'03*, July 2003. `http://www-imagis.imag.fr/Publications/2003/BKCN03`

[Ble03] Blender Foundation. *The Blender Python API Reference*, 2003. `http://www.blender3d.org/`

[BLH02] Brook Bakay, Paul Lalonde, and Wolfgang Heidrich. Real-time animated grass. In *Proceedings of Eurographics (short paper)*, 2002.

[BPF+03] Frederic Boudon, Przemyslaw Prusinkiewicz, Pavol Federl, Christophe Godin, and Radoslaw Karwowski. Interactive design of bonsai tree models. In *Eurographics Rendering Workshop 2003*, 2003.

[BS04] Jeff Bolz and Peter Schröder. Evaluation of subdivision surfaces on programmable graphics harware (to appear), 2004.

[Car84] Loren Carpenter. The A-buffer, an antialiased hidden surface method. In Hank Christiansen, editor, *Proceedings of SIGGRAPH 84*, volume 18, pages 103–108, July 1984.

[CBL99] Chun-Fa Chang, Gary Bishop, and Anselmo Lastra. Ldi tree : A hierarchical representation for image-based rendering. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 291–298, August 1999.

[cel] Celestia. `http://www.shatters.net/celestia`

[Cha98] Jacquelin Charbonnel. *LE LANGAGE C++ : Le standard ANSI/ISO expliqué*. DUNOD, 1998.

[COCSD03] Daniel Cohen-Or, Yiorgos L. Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3) :412–431, 2003.

[Coe00] Enrico Coen. *The Art of Genes - How Organisms make themselves.* Oxford University Press, 2000.

[CXGS02] Yanyun Chen, Yingqing Xu, Baining Guo, and Heung-Yeung Shum. Modeling and rendering of realistic feathers. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 630–636. ACM Press, 2002.

[D'A02] Lawrence D'Antonio. Generic programming using STL. *J. Comput. Small Coll.*, 17(3) :5–6, 2002.

[DCSD02] Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis. Interactive visualization of complex plant ecosystems. In *Proceedings of the IEEE Visualization Conference*. IEEE, October 2002. `http://www-sop.inria.fr/reves/publications/data/2002/DCSD02`

[DDSD03] Xavier Décoret, Frédo Durand, François Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. In *Proceedings of Siggraph 2003*. ACM Press, 2003. `http://www-imagis.imag.fr/Publications/2003/DDSD03`

[Dec96] Philippe Decaudin. Geometric deformation by merging a 3D object with a simple shape. In *Graphics Interface*, pages 55–60, May 1996.

[des] Descensor. `http://www.binaryworlds.com/index.html`

[DHN85] Stephen Demko, Laurie Hodges, and Bruce Naylor. Construction of fractal objects with iterated function systems. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 271–278. ACM Press, 1985.

[Dis03] Discreet. *3DS Max new features guide*, 2003. `http://www.discreet.com/docs/products/3dsmax`

[Dur99] Frédo Durand. *Visibilité tridimensionnelle : Etude analytique et applications.* PhD thesis, Université Joseph Fourier (Grenoble), Jul 1999. `http://www-imagis.imag.fr/Publications/1999/Dur99`

[DVS03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. In *Proceedings of SIGGRAPH 2003*, pages 657–662. ACM Press, 2003.

[eas] Easyfractal. `http://www.gnu.org/software/xaos/xaos.html`

[EDD+95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of SIGGRAPH 95*, pages 173–182. ACM, August 1995. `http://www.cs.washington.edu/research/projects/grail2/www/pub/pub-author.html`

[EMP+98] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing & Modeling, A Procedural Approach.* AP Professional, third edition, 1998.

[ESSS01] J. El-Sana, N. Sokolovsky, and C. T. Silva. Integrating occlusion culling with view-dependent rendering. In *IEEE Visualization 2001*, pages 371–378, October 2001.

[Fau99] François Faure. Fast iterative refinement of articulated solid dynamics. *IEEE Transactions on Visualization and Computer Graphics*, 5(3) :268–276, jul 1999. `http://www-imagis.imag.fr/Publications/1999/Fau99`

[GC98] Christophe Godin and Yves Caraglio. A multiscale model of plant topological structure. In *Journal of theoretical biology*, pages 1–46, 1998.

[GCF01] Thomas Di Giacomo, Stéphane Capo, and François Faure. An interactive forest. In Marie-Paule Cani, Nadia Magnenat-Thalmann, and Daniel Thalmann, editors, *Eurographics Workshop on Computer Animation and Simulation*, pages 65–74. Springer, sept. 2001. `http://www-imagis.imag.fr/Publications/2001/DCF01`

[GGSC96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. *Proceedings of SIGGRAPH 96*, pages 43–54, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[Giz] Gizmo 3D. http://www.gizmo3d.com/

[GLDH97] M. H. Gross, L. Lippert, R. Dittrich, and S. Häring. Two methods for wavelet-based volume rendering. *Computers & Graphics*, 21(2) :237–252, March 1997. ISSN 0097-8493.

[GLY+03] Naga K. Govindaraju, Brandon Lloyd, Sung-Eui Yoon, Avneesh Sud, and Dinesh Manocha. Interactive shadow generation in complex environments. 2003.

[GPR+03] Sylvain Guerraz, Frank Perbet, David Raulo, François Faure, and Marie-Paule Cani. A procedural approach to animate interactive natural sceneries. In *CASA03*, 2003. http://www-imagis.imag.fr/Publications/2003/GPRFC03

[Gra95] Paul Graham. *Advanced techniques for common Lisp*. Prentice Hall, 1995.

[gsc] Gscape. http://www.gscape.com

[HA90] Paul E. Haeberli and Kurt Akeley. The accumulation buffer : Hardware support for high-quality rendering. In Forest Baskett, editor, *Proceedings of SIGGRAPH 90*, volume 24, pages 309–318, August 1990.

[HHK+95] Taosong He, L. Hong, A. Kaufman, A. Varshney, and S. Wang. Voxel-based object simplification. In *Proc. Visualization '95*. IEEE Comput. Soc. Press, 1995. http://http ://www.cs.sunysb.edu/∼taosong/taosong-papers.html

[HNC02] Damien Hinsinger, Fabrice Neyret, and Marie-Paule Cani. Interactive animation of ocean waves. In *Symposium on Computer Animation*, july 2002. http://http ://www-imagis.imag.fr/Publications/2002/HNC02.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH 97*, pages 189–198, 1997.

[HRSV01] Hannes Hartenstein, Matthias Ruhl, Dietmar Saupe, and Edward R. Vrscay. On the inverse problem of fractal compression. In Bernold Fiedler, editor, *Ergodic Theory, Analysis, and Efficient Simulation of Dynamical Systems 2001*, 2001.

[Inv] Open inventor. http://oss.sgi.com/projects/inventor/

[ISGM02] William V. Baxter III, Avneesh Sud, Naga K. Govindaraju, and Dinesh Manocha. Gigawalk : Interactive walkthrough of complex environments. In *Rendering Techniques 2002 : 13th Eurographics Workshop on Rendering*, pages 203–214, June 2002.

[Jav] Java 3D. http://java.sun.com/products/java-media/3D/

[KK89] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. In *Proceedings of SIGGRAPH 89*, pages 271–280, July 1989.

[KN02] Tae-Yong Kim and Ulrich Neumann. Interactive multiresolution hair modeling and editing. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 620–629. ACM Press, 2002.

[KP03] Radoslaw Karwowski and Przemyslaw Prusinkiewicz. Design and implementation of the l+c modeling language. In Jean-Louis Giavitto and Pierre-Etienne Moreau, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.

[LCV03] Javier Lluch, Emilio Camahort, and Roberto Vivó. Procedural multiresolution for plant and tree rendering. In *Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, pages 31–38. ACM Press, 2003.

[LH96] Marc Levoy and Pat Hanrahan. Light field rendering. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Annual Conference Series, pages 31–42. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[LH03] O.S. Lawlor and J.C. Hart. Bounding recursive procedural models using convex optimization. In *Proc. Pacific Graphics 2003*, 2003. http://charm.cs.uiuc.edu/users/olawlor/projects/2002/ifsbound/

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 451–458, July 1994.

[Loo94] C. Loop. Smooth spline surfaces over irregular meshes. In *Proceedings of SIGGRAPH 94*, pages 303–310, July 1994.

[LP02] P. Lindstrom and V. Pascucci. Terrain simplification simplified : A general framework for view-dependent out-of-core visualization, 2002.

[LPFH01] Jerome Edward Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Real-time fur over arbitrary surfaces. In *Symposium on Interactive 3D Graphics*, pages 227–232, 2001.

[LR76] A. Lindenmayer and G. Rozenberg, editors. *Automata, Languages, Development*, Amsterdam, 1976. North-Holland.

[LRC+02] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann / Elsevier Science, 2002.

[LS97] J. Lengyel and J. Snyder. Rendering with coherent layers. In *Proceedings of SIGGRAPH 97*, pages 233–242, August 1997.

[Man75] Benoit Mandelbrot. *Les objets fractals*. Flammarion, Paris, 1975.

[Mar02] David Margery. *OpenMASK programming guide*, 2002.

[Mec98] Radomir Mech. *CPFG Version 3.4 User's Manual*, 1998.

[MKM89] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. The synthesis and rendering of eroded fractal terrains. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 41–50, July 1989.

[MN98] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. In George Drettakis and Nelson Max, editors, *Eurographics Rendering Workshop 1998*, pages 157–168, New York City, NY, Jul 1998. Eurographics, Springer Wein. `http://www-imagis.imag.fr/Publications/1998/MN98b`

[MN00] Alexandre Meyer and Fabrice Neyret. Multiscale shaders for the efficient realistic rendering of pine-trees. In *Graphics Interface*, pages 137–144. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 2000.

[MNP01] Alexandre Meyer, Fabrice Neyret, and Pierre Poulin. Interactive rendering of trees with shading and shadows. In *Eurographics Workshop on Rendering*, Jul 2001. `http://www-imagis.imag.fr/Publications/2001/MNP01`

[moj] MojoWord. `http://www.pandromeda.com/`

[MS95] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *1995 Symposium on Interactive 3D Graphics*, pages 95–102, April 1995.

[Ney95] Fabrice Neyret. Animated texels. In Dimitri Terzopoulos and Daniel Thalmann, editors, *Computer Animation and Simulation '95*, pages 97–103. Eurographics, Springer-Verlag, September 1995. ISBN 3-211-82738-2.

[Ney98] Fabrice Neyret. Modeling animating and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), Jan–Mar 1998. `http://www-imagis.imag.fr/Publications/1998/Ney98`

[Ney01] Fabrice Neyret. *Complexite Naturelle et Synthese d'Images*. PhD thesis, UJF (Habilitation), Oct 2001. `http://www-imagis.imag.fr/Publications/2001/Ney01`

[NF87] Tom Nadas and Alain Fournier. GRAPE : An environment to build display processes. In Maureen C. Stone, editor, *Proceedings of SIGGRAPH 87*, volume 21, pages 75–84, July 1987.

[Nvi02] Nvidia. *CG Toolkit Reference Manual v1.5*, 2002. http:// www.cgshaders.org

[Opea] Open RM. http://www.openrm.org/

[Opeb] Open scene graph. http://openscenegraph.sourceforge.net/

[Opec] Open SG. http://www.opensg.org/

[PC01] Frank Perbet and Marie-Paule Cani. Animating prairies in real-time. In Stephen N. Spencer, editor, *Proceedings of the Conference on the 2001 Symposium on interactive 3D Graphics*. Eurographics, ACM Press, 2001.

[Per] Performer. http://www.sgi.com/software/performer/

[Per02] Ken Perlin. Improving noise. In *Proceedings of SIGGRAPH 2002*, volume 21, pages 681–682, July 2002.

[PGF01] Przemyslaw Prusinkiewicz, Christophe Gaudin, and Pascal Ferraro. Platonic world, 2001. Discussion orale avec Przemyslaw Prusinkiewicz.

[PHM99] Przemyslaw Prusinkiewicz, Jim Hanan, and Radomir Mech. An l-system-based plant modeling language. In *Proceedings of the International workshop AGTIVE'99*, 1999.

[PMKL01] Przemyslaw Prusinkiewicz, Lars Mündermann, Radoslaw Karwowski, and Brendan Lane. The use of positional information in the modeling of plants. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 289–300. ACM Press, 2001.

[PV95] Ken Perlin and Luiz Velho. Live paint : Painting with procedural multiscale textures. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 153–160, August 1995.

[RB93] J. Rossignac and P. Borrel. Multi-resolution 3D approximation for rendering complex scenes. In *Second Conference on Geometric Modelling in Computer Graphics*, pages 453–465, June 1993. Genova, Italy.

[Red97] M. Reddy. *Perceptually Modulated Level of Detail for Virtual Environments*. PhD thesis, University of Edinburgh, 1997.

[Ren] Renderware. http://www.renderware.com/

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat : A multiresolution point rendering system for large meshes. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 343–352, July 2000.

[SD01] Marc Stamminger and George Drettakis. Interactive sampling and rendering for complex and procedural geometry. In K. Myskowski and S. Gortler, editors, *Rendering Techniques 2001 (Proceedings of the Eurographics Workshop on Rendering 01)*, 12th Eurographics workshop on Rendering. Eurographics, Springer Verlag, 2001. http://www-sop.inria.fr/reves/publications/data/2001/SD01

[SDS96] E. Stollnitz, T. Derose, and D. Salesin. *Wavelets for Computer Graphics*. Morgan Kaufmann, San Francisco, California, 1996.

[Smi84] Alvy Ray Smith. Plants, fractals and formal languages. In *Proceedings of SIGGRAPH 1984*, 1984.

[spe] Speedtree. http://www.idvinc.com/html/product_browser.htm

[Sta97] Jos Stam. Stochastic dynamics : Simulating the effects of turbulence on flexible structures. *Computer Graphics Forum*, 16(3) :159–164, August 1997.

[Stu01] Lionheads Studios. Black and white, 2001. http://blackandwhite.ea.com/

[TLC02] Franco Tecchia, Céline Loscos, and Yiorgos Chrysanthou. Image-based crowd rendering. *IEEE Computer Graphics & Applications*, 22(2) :36–43, 2002.

[TT95] J. Thollot and E. Tosan. Constructive fractal geometry : Constructive approach to fractal modeling using language operations. In *Graphics Interface '95*, pages 196–203, May 1995.

[vsk] Virtual skipper 3. http://http ://www.virtualskipper3.com/.

[vte] Virtual terrain project. http://www.vterrain.org/

[WH91] Jakub Wejchert and David Haumann. Animation aerodynamics. In *Proceedings of SIGGRAPH 91*, pages 19–22, July 1991.

[Whi81] Turner Whitted. The causes of aliasing in computer generated images. In *SIGGRAPH '81 Advanced Image Synthesis seminar notes*. August 1981.

[Wid] Mattias Widmark. Procedural geometry for real-time terrain visualisation in blueberry 3d. http://www.blueberry3d.com/

[Wil83] Lance Williams. Pyramidal parametrics. In *Proceedings of SIGGRAPH 83*, volume 17(3), pages 1–11, July 1983.

[WND99] Mason Woo, Jackie Neider, and Tom David. *OpenGL 1.2 Programming Guide, 3rd Edition : The Official Guide to learning OpenGL, Version 1.2*. Addison Wesley, 1999. WOO m 99 :1 1.Ex.

[Wor] Worldtoolkit. http://www.sense8.com/products/wtk.html

[WP95] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 119–128, August 1995.

[WS02] M. Wand and W. StraBer. Multi-resolution rendering of complex animated scenes. In *Computer Graphics Forum*, 2002.

[WW03] Michael Wimmer and Peter Wonka. Rendering time estimation for real-time rendering. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 118–129. Eurographics Association, 2003.

[xao] Real-time fractal zoomer. http://www.berkhan.com/easy/easyfrac.htm

[YSM03] Sung-Eui Yoon, Brian Salomon, and Dinesh Manocha. Interactive view-dependent rendering with conservative occlusion culling in complex environments. In *IEEE Visualization, 2003*, 2003.

[ZS03] Douglas E. Zongker and David H. Salesin. On creating animated presentations. In *Proceedings of SIGGRAPH 2003*, pages 298–308. Eurographics Association, 2003.