Titre: Title:	Calculs génériques sur les processeurs graphiques modernes : application à la résolution numérique de systèmes par la méthode des différences finies
Auteur: Author:	Quentin Bleton
Date:	2005
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Bleton, Q. (2005). Calculs génériques sur les processeurs graphiques modernes : application à la résolution numérique de systèmes par la méthode des différences finies [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7589/

UNIVERSITÉ DE MONTRÉAL

CALCULS GÉNÉRIQUES SUR LES PROCESSEURS GRAPHIQUES MODERNES : APPLICATION À LA RÉSOLUTION NUMÉRIQUE DE SYSTÈMES PAR LA MÉTHODE DES DIFFÉRENCES FINIES

QUENTIN BLETON
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JUIN 2005

Your file Votre référence
ISBN: 978-0-494-16758-8
Our file Notre référence
ISBN: 978-0-494-16758-8

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

CALCULS GÉNÉRIQUES SUR LES PROCESSEURS GRAPHIQUES MODERNES : APPLICATION À LA RÉSOLUTION NUMÉRIQUE DE SYSTÈMES PAR LA MÉTHODE DES DIFFÉRENCES FINIES

présenté par: BLETON Quentin

en vue de l'obtention du diplôme de: Maîtrise ès science appliquées

a été dûment accepté par le jury d'examen constitué de:

Mme. CHERIET Farida, Ph.D., présidente

M. OZELL Benoit, Ph.D., membre et directeur de recherche

M. GRANGER Louis, M.Sc., membre

REMERCIEMENTS

J'aimerais remercier mon directeur de recherche pour m'avoir fourni l'aide financière tout au long de ma maîtrise, et laissé une grande liberté dans mon travail. Je remercie également Jérome Bettinger pour la documentation qu'il m'a fourni et ses commentaires. Merci à Luc Lalonde qui a facilité l'installation de mon poste de travail et a toujours été efficace pour résoudre mes problèmes. Enfin, je salue Moncef, Haroun, et Sébastien avec qui j'ai passé de bons moments comme partenaires de laboratoire et partagé leurs connaissances de Latex et Linux.

RÉSUMÉ

Récemment, une innovation majeure dans l'industrie des cartes graphiques grand public a été accomplie avec l'introduction des cartes programmables, qui offrent au programmeur le contrôle sur les calculs effectués par l'unité de traitement graphique (appelée *GPU*). Cette programmabilité ouvre un champ de possibilités qui va au delà du cadre graphique, et rend les GPU capables d'effectuer des tâches calculatoires complexes.

Dans ce mémoire, un aperçu de la variété des problèmes abordés et de leur implémentation sur les GPU est présenté, ainsi que les stratégies communes qui en ressortent en termes de gestion de mémoire et d'algorithmique. Dans l'ensemble, on constate que les implémentations existantes sont spécifiques à un problème particuliers et ne peuvent s'appliquer dans un cadre différent sans une modification de leur code source. De plus ces modifications nécessiteraient des connaissances avancées en infographie que peu de scientifiques possèdent.

Partant de ce constat, notre objectif est de proposer un outil de calcul sur GPU *simple* à utiliser, *extensible* à plusieurs problèmes, et *efficace* par rapport à son équivalent sur le CPU. La motivation principale derrière notre travail est la volonté de *démocratiser* l'utilisation du GPU comme engin de calcul, qui est aujourd'hui encore réservé à un public très spécialisé.

Le résultat consiste en une bibliothèque de fonctions réutilisables et simples, dédiées au calcul algébrique linéaire sur GPU, à l'instar de la librarie BLAS (*Basic Linear Algrebra Subroutines*) pour le CPU. Ces calculs algébriques sont exécutés par un simple appel de fonction et sont facilement intégrables dans n'importe quelle application C++. De plus toute la partie technique d'infographie avancée intervenant dans le processus est masquée.

L'extensibilité de notre bibliothèque est démontrée au travers de l'implémentation, à partir des opérateurs de base, d'un nouvel opérateur dédié à la résolution d'un système linéaire. L'efficacité de ces opérateurs par rapport à leur équivalent BLAS sur CPU est ensuite présentée et analysée, avec gain pouvant atteindre un facteur 7 (GeForce 6600 / Athlon XP 2700). Enfin, la simplicité d'utilisation et d'intégration de ces outils est illustrée par l'implémentation de deux applications simulant des phénomènes physiques simples : une simulation de tissu et une simulation d'ondes 2D.

En conclusion, il est possible d'adapter les connaissances et les solutions déjà présentes dans le domaine du calcul générique sur GPU afin de proposer des outils plus souples et plus accessibles que les solutions existantes tout en restant aussi puissants. Afin d'utiliser ces outils, nous recommandons l'utilisation d'une carte de marque nVidia et de modèle Geforce 5 ou plus, ainsi que le système d'exploitation Linux et le compilateur Cg de nVidia.

ABSTRACT

Since early nineties, computer graphics hardware has evolved from expensive dedicated stations to powerful and affordable graphic cards that now come with every PC. Yet, the very recent introduction of programmable graphics hardware has been another huge improvement which, combined with a growing processing power, can turn these former *black boxes* into powerful and generic computing units capable of handling a wide panel of complex non-graphical problems.

First, we will review the panel of problems already solved with the GPU, and regroup their approaches in terms of data structures, memory management, and algorithms. What appears through that review is that most of the solutions proposed involve an advance knowledge in computer graphics and are impossible for a beginner to re-use and adapt for another purposes.

The goal of our work is to make a simple and expandable GPU-based solutions accessible for everyone skilled in C++ by providing a library containing a set of basic linear algebra tools on the GPU, just like BLAS (Basic Linear Algebra Subroutines) does for the CPU. After detailing the implementation of such tools, we will demonstrate their simplicity and efficiency through the implementation of two simple physical simulations.

The various uses of our library is demonstrated through the implementation of a conjugate gradients solver based on our basic operators. The efficiency of these operators can reach a gain factor of 7 over the CPU implementation (GeForce 6600 / Athlon XP 2700). The ease of use of our library is shown by two example of numerical simulations : a cloth simulation and a shallow 2D wave simulation.

Finally, we managed to use the current knowledge about GPGPU to make a set

of simple and efficient computational tools that only requires basic C++ skills. However, Linux and a nVidia graphic card (Geforce serie 5 and newer) as well as the Cg compiler are required to make these tools operational.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

CPU	Central Processing Unit
GPU	Graphical Processing Unit
SIMD	Single-Instruction stream Multiple-Data stream
AGP	Accelerated Graphics Port
SSE	Streaming SIMD Extensions
MRT	Multiple Render Targets
PS	Pixel Shader
VS	Vertex Shader
Pixel Shader	Nuanceur de pixels
Vertex Shader	Nuanceur de sommets
Rasterizer	Trameur
Stream	Flux
Kernel	Noyau
Frame Buffer	Mémoire d'image
Stencil Buffer	Tampon de masque
Depth Buffer	Tampon de profondeur
Vertex Buffer	Tampon de sommets
Pixel Buffer	Tampon de pixels

LISTE DES ANNEXES

INTRODUCTION

Depuis les années 90, les progrès dans l'accélération matérielle de l'affichage de scènes 3D ont progressivement permis de passer des exhorbitantes stations graphiques spécialisées aux cartes graphiques actuelles qui équipent aujourd'hui tous les ordinateurs personnels. Cette évolution a été jalonnée par plusieurs étapes importantes, qui permettent de distinguer quatre générations de cartes graphiques :

- Les premières stations spécialisées proposées au début des années 90 par des compagnies comme Silicon Graphics, qui prenaient en charge tout le processus de rendu, mais étaient encore réservées à un public restreint en raison de leur prix extrêmement élevé.

- L'apparition des premières cartes accélératrices 3D (jusqu'en 1998), beaucoup moins chères et libérant le CPU du tramage des pixels et de l'application des textures, mais en lui laissant toutefois la charge de la transformation et de l'éclairage des sommets 3D.

- Les cartes supportant en matériel à la fois les transformations et l'éclairage des sommets (technologie baptisée *Transform and Lighting*), ainsi que le tramage et l'application de texture (1999-2000). Le CPU n'intervient alors plus du tout dans le processus de rendu, désormais complètement pris en charge par le pipeline graphique qui est pris en charge matériellement par le GPU.

- Les GPU donnant pour la première fois le contrôle sur la transformation des sommets au sein du pipeline (2001), ainsi que quelques possibilités sur l'application des textures. Ce sont les premiers pas vers la *programmabilité* des GPU.

Tableau 1 Cartes de dernière génération

Fabricant	Modèle	Références
Nvidia	GeForce FX	5100 5200 5700 5900 5950
Nvidia	GeForce Série 6	6200 6600 6800
ATI	Radeon	9200 9550 9600 9700 9800
ATI	Série X	X300 X600 X700 X800

- La dernière génération pousse la flexibilité du pipeline beaucoup plus loin en offrant une programmabilité complète sur le rendu des sommets et des pixels (2002-2005).

Notre travail ne s'applique qu'aux cartes de la dernière génération, dont les modèles sont listés dans le tableau 1. Il faut cependant noter que les cartes ATI n'offrent pas le même degré de précision sur les calculs. Les interfaces de programmation et le système opérationnel choisis pour l'implémentation sont OpenGL 1.5, Cg (Mark *et al.* (2003)) et Linux, et sont requis pour faire fonctionner les outils que nous allons proposer.

La question à la base de cette recherche est la suivante : est-il possible, malgré l'aspect très technique et spécialisé du calcul générique sur GPU, de rendre cette technologie utilisable par des personnes ignorant la programmation sur GPU.

L'objectif global du projet est donc de proposer une interface, une librairie de fonctions, mettant le calcul sur GPU à la portée de n'importe qui ayant des connaissances de base en programmation C++. Cette interface devra être simple d'utilisation, extensible, et appliquable à une variété de problèmes, tout en offrant une rapidité de calcul supérieure au CPU.

Ce mémoire se décompose en quatre chapitres ; dans le premier on présente les particularités et les contraintes du calcul générique sur GPU. Dans le deuxième chapitre, au travers d'une revue de littérature, on montre comment ces contraintes

se répercutent sur la programmation du GPU en terme d'algorithmes et de structure de données. Le troisième chapitre détaille l'implémentation de notre bibliothèque de fonctions de calcul algébrique, à savoir les opérateurs élémentaires et un opérateur de résolution par gradients conjugués construit à partir des opérateurs élémentaires. Enfin, dans la dernière partie, on montre comment utiliser nos outils dans deux applications de simulation de tissu et de liquide, et on analyse les performances de nos opérateurs.

CHAPITRE 1

LE POTENTIEL DES GPU

Ce chapitre a pour but de souligner l'intérêt des cartes graphiques comme outil de calcul scientifique, d'en comprendre le fonctionnement, et d'en déduire les défis qu'implique une telle utilisation.

Si le lecteur n'a pas de connaissance particulière des nuanceurs (*shaders* en anglais), l'annexe I explique en détail leur mode de fonctionnement, leurs capacités ainsi que leurs limites.

1.1 Motivation et intérêt du calcul générique sur GPU

1.1.1 Puissance de calcul

Le calcul générique sur GPU présente-t-il un réel intérêt par rapport au calcul classique sur CPU ? La réponse est clairement oui, car en raison du fonctionnement hautement parallélisé des cartes graphiques, la puissance brute de calcul de certaines cartes dépasse de plusieurs fois les performances des plus récents processeurs. Comme critère de comparaison, nous considèrerons le nombre maximum théorique d'opérations sur des nombres en virgule flottante par seconde ou *GFlops*.

La figure 1.1 illustre le fonctionnement parallélisé d'une Geforce 6800 de Nvidia. Les sommets passés à la carte sont traités simultanément par 6 nuanceurs pour la mise en place des géométries destinées à être tramées. Les pixels ayant passé avec succès le test de profondeur sont ensuite distribués aux 16 nuanceurs disponibles

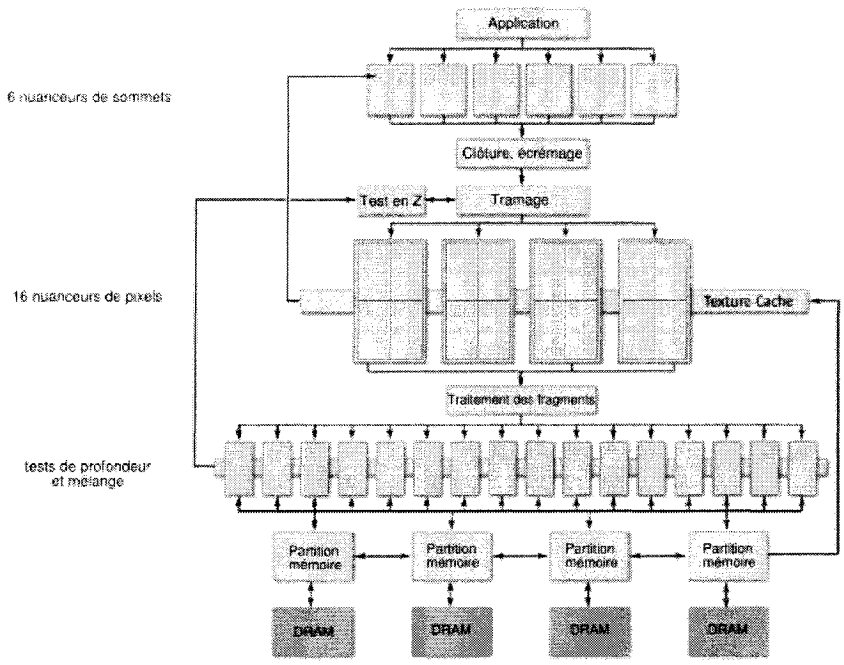

Figure 1.1 Architecture parallèle de la Geforce 6800 (site web de nVidia)

qui travaillent simultanément et de manière synchronisée, tout en ayant accès à la mémoire vidéo.

Sachant que la Geforce 6800 supporte, pour tout le pipeline, l'utilisation d'une précision de 32 bits par canal de couleur, la puissance maximale P_{max} s'exprime par la formule :

$$P_{max} = F_{mem} * Nb_{pipeline} * Nb_{canal}$$

où F_{mem} est la fréquence de fonctionnement de la puce graphique, $Nb_{pipeline}$ le nombre de pipelines disponibles, et Nb_{canal} le nombre de canaux utilisés.

La fréquence de la mémoire vidéo étant de 1.1 GHz, le nombre de pipelines de 16 et le nombre de canaux de 4 (RGBA), ceci donne une puissance théorique maximale de calcul de *70 GFlops*. Si l'on compare à un processeur classique fréquencé à 3 GHz,

capable d'effectuer au mieux *3 GFlops*. la différence est significative et démontre l'avantage de la parallélisation des calculs au sein des GPU.

Si les GPU apparaissent donc déjà comme de sérieux concurrents pour les processeurs, il est d'autant plus intéressant de miser dès maintenant sur leur puissance que le taux de progression des performances des GPU dépasse déjà de loin celui des CPU.

1.1.2 L'avenir des GPU

En raison de la nature parallèle de l'architecture des GPU, le gain de puissance d'une génération à l'autre se fait en augmentant la parallélisation et donc le nombre de pipelines. Les fabricants de processeurs quant à eux misent avant tout sur la fréquence de fonctionnement. Ceci est la raison sous-jacente à la loi qui régit l'évolution de la puissance des GPU, qui surpasse celle des CPU en doublant tous les 6 mois comparé aux 18 mois habituels pour les processeurs (loi de Moore).

La figure 1.2 montre les courbes d'évolution de puissance de calcul pour les puces graphiques ATI et Nvidia, ainsi que pour la série des processeurs Pentium IV. Dans le cas des puces ATI la précision est cependant de 24 bits seulement par rapport à 32 bits pour la série Nvidia et Pentium IV.

Ce graphique démontre donc l'intérêt de miser sur l'utilisation des GPU pour le calcul générique, car l'écart de performance entre GPU et CPU ne fait que se creuser avec le temps. La technologie SLI (*Scalable Link Interface*) est un argument de plus en faveur des GPU, qui permet de combiner la puissance de deux cartes graphiques en les reliant par un connecteur spécial (figure 1.3) et ainsi doubler la puissance disponible.

Dans les années qui viennent, il sera probablement possible de mettre plus que

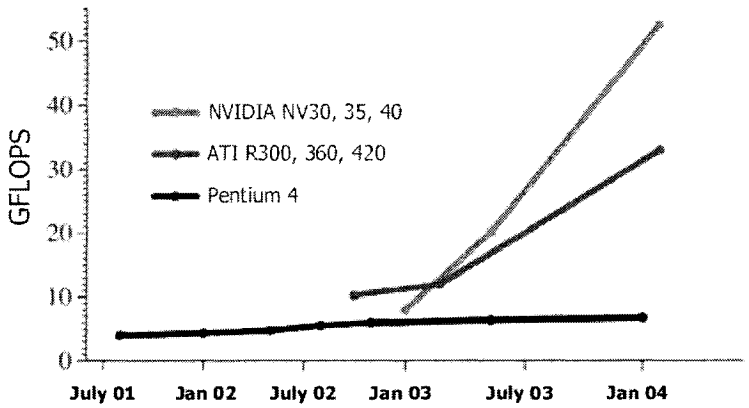

Figure 1.2 Évolution de la puissance des GPU / CPU (GPGPU (2004))

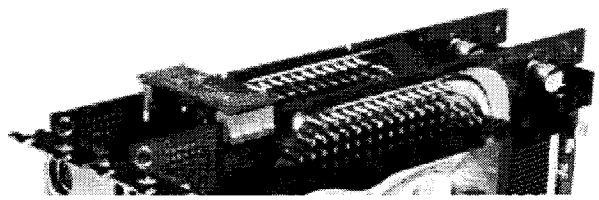

Figure 1.3 Mise en parallèle de deux GPU par la technologie SLI

deux cartes en parallèle, et d'accroître encore les performances. Tout ceci rend les solutions sur GPU extrêmement attirantes, c'est pouquoi certaines grosses sociétés de l'industrie graphique commencent déjà à proposer des produits comme par exemple le moteur de rendu de film Gelato (grappe de cartes graphiques Nvidia (2004)). Dernier atout, l'introduction récente du *PCI express* a quadruplé la bande passante disponible entre le GPU et le CPU (passant de 2Go à 8Go), et permet même d'utiliser la mémoire système comme mémoire vidéo (nouvelle série des Geforce *TurboCache*).

1.1.3 Coût

Si on prend comme critère de comparaison le coût du GFlops, les GPU sont beaucoup moins chers que les CPU. A puissance égale, les GPU sont aussi les moins chers. Si l'on compare les séries GeForce et Pentium IV, on obtient le tableau sui-

Tableau 1.1 Comparaison du prix du GFlops

Modèle	GFLops	prix	$/GFlops
GeForce5200	5.2	76	15
GeForce5900	14.4	270	19
GeForce6800	70	410	5
Pentium IV	2.4	195	81
Pentium IV	3	280	95
Pentium IV	3.2	340	106

vant (les prix sont tirés d'un revendeur informatique de Montréal). On voit que le GFlops peut coûter plus de vingt fois plus cher pour le processeur que pour le GPU !

De plus, lorsqu'il s'agit de changer de matériel il est beaucoup plus aisé de remplacer une carte graphique qu'un CPU qui nécessite une interface de connexion spécifique. Le changement d'un CPU pour une génération plus récente, implique donc le rachat d'une carte mère et éventuellement d'autres composants (mémoire par exemple). Les ports de connexion aux GPU changent quant à eux très rarement et on peut espérer que les PCI express ne soient pas remplacés avant des années.

Enfin, il est important de noter qu'un programme sur CPU est spécifique au jeu d'instructions du CPU (powerPC, Solaris, extensions SSE & 3dnow, amd64, etc), alors qu'un algorithme sur GPU peut être exécuté sur toutes les cartes graphiques supportant le modèle de nuanceur utilisé.

Les GPU semblent donc extrêmement prometteurs pour leur puissance de calcul, mais leur modèle de fonctionnement parallèle fait apparaître un certain nombre de contraintes et de compromis à prendre en compte pour pouvoir adapter un algorithme de calcul sur le GPU.

1.2 Particularités des GPU

L'architecture hautement parallélisée des cartes graphiques s'inspire en fait d'une architecture plus générale connue sous le nom de *processeurs de flux* (*Stream Processors*). La première machine développée selon ce principe baptisée *Imagine* fût présentée par Khailany *et al.* (2001).

1.2.1 Programmation par flux

Deux notions fondamentales en ressortent, la notion de **noyau** (*kernel*) et celle de **flux** (*stream*). Nous allons en résumer le principe, mais pour une explication plus détaillée, consulter Dally *et al.* (2004).

Un *flux* est par définition un ensemble de données destinées à être traitées de manière *indépendante* et *simultanée*. Dans le cas d'une carte graphique, le flux peut être issu de multiples endroits dans le pipeline :

- **Au début du pipeline**, où la liste des sommets est envoyée par l'application OpenGL à la carte graphique. Chacun de ces sommets subit alors indépendamment les mêmes transformations (transformations géométriques et de clôture).

- **À la sortie du trameur**, où chaque pixel subit le même calcul d'éclairage.

Ces deux sources de flux au sein d'un GPU ont des éléments d'entrée et de sortie présentés en annexe dans les tableaux I.1 - I.3 , et I.2 - I.4.

Quant au *noyau*, il est défini comme étant l'ensemble des instructions à appliquer à chacun des éléments d'un flux. Tous les blocs du pipeline présenté par la figure I.1 sont en fait des noyaux, mais les seuls qui nous intéressent sont les noyaux programmables par l'utilisateur à savoir les *nuanceurs* qui se divisent en deux

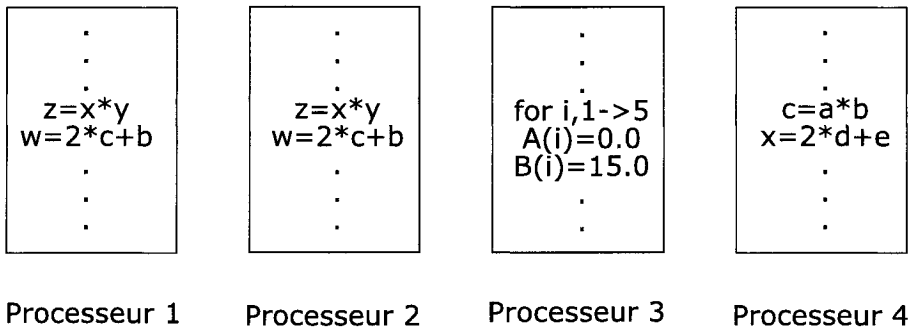

Figure 1.4 Illustration du fonctionnement d'une architecture MIMD

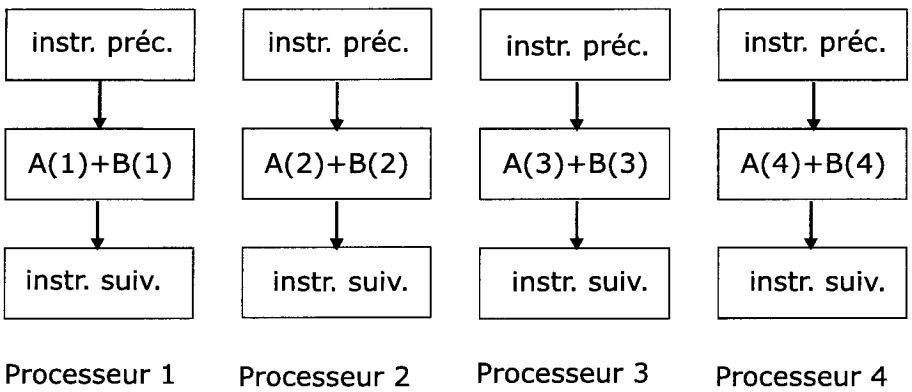

Figure 1.5 Illustration du fonctionnement d'une architecture SIMD

grandes familles, chacune traitant le flux d'une manière qui lui est propre.

- Les nuanceurs de sommets sont de la catégorie **MIMD** (*Multiple Instruction stream, Multiple Data stream*) : chaque unité traite le flux d'instructions de manière autonome et asynchrone. A tout moment, chacune des unités peut exécuter des instructions différentes sur des portions différentes de flux, comme illustré sur la figure 1.4.

- Les nuanceurs de pixels sont de type **SIMD** (*Single Instruction stream, Multiple Data stream*). Chaque unité de calcul exécute simultanément la même instruction mais sur des portions différentes de flux (voir la figure 1.5). Un GPU traite ainsi le flux de pixels en exécutant la même instruction pour un groupe de pixels voisins (*pixels lock steps*).

Voyons maintenant l'impact de cette architecture sur les contraintes de programmation des GPU.

1.2.2 Contraintes de programmation

Le premier type de contraintes intervient sur les opérations permises de lecture et d'écriture des flux. Etant donné qu'aucune communication entre éléments de flux ou entre noyaux n'est possible, cela implique que :

- Les noyaux ne peuvent pas écrire dans le flux d'entrée.
- Les noyaux ne peuvent communiquer entre eux.
- Les éléments du flux sont totalement indépendants.

Ces contraintes ont les implications suivantes sur la structure des noyaux eux-mêmes :

- **Pas d'opération de réduction** (par exemple min, max, produit scalaire). Cela est dû au fait que le flux d'entrée et de sortie sont de même taille.

- **Pas d'opération de diffusion** : impossibilité de communiquer un résultat vers des flux de sortie issus d'autres noyaux. Concrètement cela signifie qu'on ne contrôle pas l'endroit de la mémoire où le flux est écrit.

- **Pas d'instructions dépendant du contenu des éléments du flux d'entrée** : l'usage de structures de code de type *if,then,else* n'est pas supporté par les noyaux.

- **Nombre limité de flux de sortie** : la taille du flux est fixée par celle des registres de sortie.

- **Nombre limité d'instructions** : entre 30 et 512 selon les générations de GPU (celle du nuanceur 3.0 est à part avec 65000 instructions).

En somme, les GPU offrent un potentiel de puissance énorme mais au prix d'une utilisation laborieuse en raison de leur programmabilité limitée et de leur système de mémoire peu flexible. C'est la raison pour laquelle une véritable discipline en est née, appelée *GPGPU* : *General Purpose Graphical Processing Units*. La partie suivante fait la revue de cette nouvelle discipline au travers des différents travaux présentés dans la littérature, et montre comment elle s'est adaptée aux contraintes que nous avons mis en évidence.

CHAPITRE 2

REVUE DU CALCUL GÉNÉRIQUE SUR GPU

Cette partie est destinée à montrer, au travers d'une revue de la littérature, comment la théorie du traitement de flux s'applique à l'architecture des GPU. Nous verrons tout d'abord comment ces flux se matérialisent dans les GPUs en fonction du type de données traitées. Nous montrerons ensuite comment les noyaux sont implémentés, et quelles sont les méthodes utilisées pour contourner leurs limitations (évoquées dans 1.2.2). Cette étude nous donnera également les éléments techniques nécessaires pour la mise en place de nos opérateurs et nous permettra d'identifier les critères déterminants pour obtenir de bonnes performances.

2.1 Circulation des données sur le GPU

2.1.1 Système de mémoire des GPU

Avant d'aborder les différents formats de flux utilisés dans la littérature, il est important de présenter rapidement le fonctionnement de la mémoire présente au sein des GPU. Afin de garantir le critère de localité (c.f. 2.5.4) requis pour rentabiliser le calcul sur GPU, les données du calcul doivent être mémorisées sur la carte elle-même et y demeurer le plus longtemps possible.

La figure 2.1 montre où il est possible de stocker ces données sur le GPU :

- **La mémoire d'image**, qui sert à contenir l'image finale avant de l'afficher à l'écran. Cette mémoire est constituée de multiples couches, comme le tampon

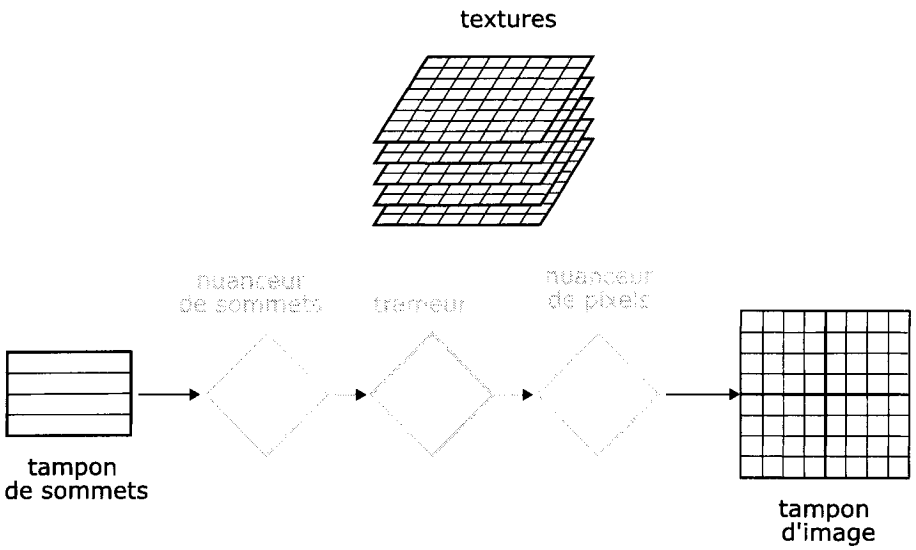

Figure 2.1 Système de mémoire d'un GPU

de profondeur, le tampon de masque, les tampons chromatiques avant/arrière (et gauche/droite pour les cartes supportant la stéréo). Cette mémoire a une grande capacité mais comme l'indique l'appellation *tampon*, elle est temporaire et écrasée à chaque affichage.

- **Le tampon de sommets** est lu à chaque invocation du pipeline, et y fait son entrée par le nuanceur de sommets. Contrairement à la mémoire d'image, cette mémoire-ci n'est pas écrasée à chaque passe et peut être contrôlée par le programmeur par l'intermédiaire de l'extention `GL_EXT_vertex_array` dans le cas d'OpenGL. Le format interne de ce tampon correspond aux registres disponibles pour les sommets (couleur, normale, coordonnées de texture, etc).

- **Les textures** sont le plus souvent utilisées, et offrent le choix du nombre de bits par canal de couleur, ainsi que le nombre de canaux. La possibilité de stocker un nombre en virgule flottante de 32 bits par canal en fait un outil idéal pour le calcul générique, en plus d'être accessible par le nuanceur de pixels.

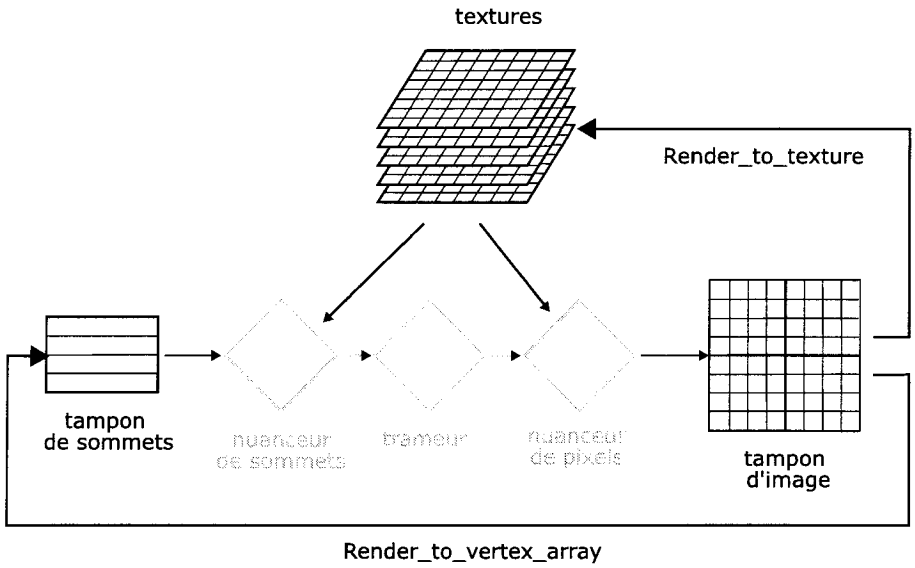

Figure 2.2 Système de retour de données dans un GPU

2.1.2 Retour de données au sein du GPU

Les zones mémoires évoquées précédemment peuvent communiquer entre elles au sein du pipeline, et maintenir ainsi le flux de données sur le GPU lui-même. La figure 2.2 indique les voies de communication disponibles pour le transfert des données en mémoire au sein du pipeline. L'objectif est de réutiliser les données sortant du pipeline pour ensuite les réinjecter plus en amont et continuer à travailler localement. De cette manière, on respecte la localité du calcul sur le GPU garantissant une bonne performance. Plusieurs techniques ont été développées dans la littérature pour réaliser cette réinjection.

2.1.2.1 Rendu/copie en texture

C'est la technique utilisée dans la majorité des travaux sur le calcul générique sur GPU. Elle consiste à utiliser les tampons de pixels (*Pixel Buffers*) pour faire un *rendu hors-écran* et transférer le contenu de la mémoire image dans une texture.

Harris *et al.* (2002) se servent par exemple des textures comme réseau régulier de couplage 2D ou 3D (*CML* ou *Coupled Map Lattice*) afin de simuler des phénomènes physiques simples comme l'ébullition ou la diffusion. Chaque pixel de la texture sert à stocker les propriétés d'un élément d'une grille 2D ou 3D, comme par exemple la température. Ces textures sont ensuites utilisées pour effectuer une simulation physique et mettre à jour le champ de température au pas de temps suivant dans une nouvelle texture, et ainsi de suite. La perte de performance induite par l'utilisation des textures 3D les oblige à utiliser à la place une pile de textures 2D afin de préserver l'interactivité de leur outil.

Purcell *et al.* (2002) utilisent les textures pour implémenter un algorithme de lancer de rayons. Les textures 2D sont employées pour stocker les listes de sommets, les listes de triangles de la scène, ainsi que la grille des rayons à lancer. Enfin, une texture 3D est utilisée pour faire un partionnement de l'espace et minimiser le nombre de tests d'intersection.

Moreland et Angel (2003), afin de réaliser le calcul d'une transformée de Fourier sur GPU, agencent les coefficients réels et imaginaires au sein d'une texture 2D d'une manière spécifique afin de minimiser le nombre d'opérations requises pour la transformée.

De nombreux autres exemples existent utilisant également des textures, comme Kim et Lin (2003) (simulation de croissance de cristaux), Strzodka *et al.* (2003) (transformée de Hough généralisée), M. et Strzodka (2001) (segmentation d'image)...

Suivant le matériel et le système d'exploitation, ce transfert se fait soit directement (DirectX & rendu en texture) soit indirectement (OpenGL & copie en texture). Buck *et al.* (2004) utilisent ces deux méthodes dans leur outil générique de calcul sur GPU multiplateforme, et reconnaissent la perte de performance d'OpenGL par rapport à DirectX en raison du processus de copie plutôt que de rendu direct.

2.1.2.2 Rendu/copie en sommets

De la même manière, il est possible de copier le contenu du tampon de pixels en tant que tableau de sommets et le réinjecter tout au début du pipeline pour un traitement ultérieur.

La seule illustration du rendu direct en sommets sont les travaux de Kipfer *et al.* (2004) qui utilisent le rendu en sommets pour implémenter un moteur de particules uniquement sur le GPU. Seules les cartes ATI supportent le rendu direct en sommets via une extension propriétaire, et du coté ARB OpenGL (*Architecture Review Board*), les *Superbuffers* qui devraient permettre cette manipulation sont encore en développement.

2.2 Stockage des données de calcul sur un GPU

Cette partie catalogue les différentes techniques de stockage des données utilisées pour le calcul générique sur GPU et les regroupe en plusieurs familles.

2.2.1 Tableaux

Ici nous nous intéressons au cas où les calculs se font sur des tableaux (pleins) de nombres réels de dimension quelconque.

Dans le cas des tableaux à une ou deux dimensions, il est possible d'utiliser soit des tableaux de sommets, soit des textures. Cependant ce sont les textures qui sont le plus souvent employées.

Dans le cas des tableaux à une dimension, même si les textures 1D semblent les plus adaptées, Kruger et Westermann (2003) notent des performances décevantes par

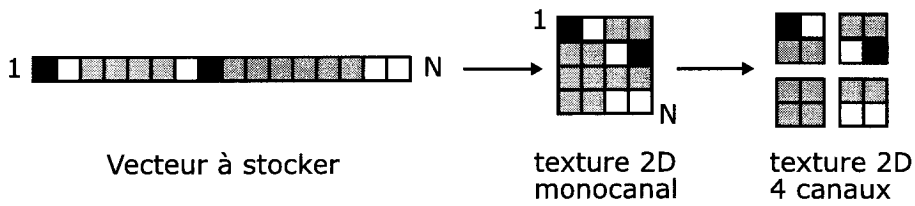

Figure 2.3 Compression de données dans une texture

rapport aux textures 2D. C'est pourquoi ils proposent de transférer les tableaux à une dimension dans un tableau à deux dimensions. La figure 2.3 illustre comment un vecteur de données peut être transféré dans une texture 2D avant d'être envoyé dans la mémoire vidéo. Les données sont agencées par quadruplets du haut vers le bas et de la gauche vers la droite, afin d'exploiter les quatres canaux RGBA de la texture. Cette méthode peut cependant engendrer une *perte de mémoire* lorsque les vecteurs n'occupent pas tout l'espace de la texture 2D. L'autre inconvénient est le coût associé à la transposition des adresses 1D/2D aussi bien pour l'écriture que pour la lecture des données. Purcell *et al.* (2002) estiment le coût associé à ces manipulations à environ 60% du total des opérations !

Les tableaux 2D ou *matrices* s'adaptent quant à eux plus naturellement en une texture à deux dimensions, chaque élément du tableau correspondant à un pixel de la texture. Harris *et al.* (2002) utilisent par exemple les textures 2D comme grille régulière de points dans un champ 2D ou 3D et y stockent des informations comme la vélocité ou la température. Il n'est cependant pas possible de stocker plus de 4 scalaires par élément de la grille. Il est important de noter qu'ils sont les premiers à avoir intégré leurs outils au sein d'une interface graphique permettant une expérimentation facile et rapide en y intégrant les opérations de diffusion/convection. Ces outils sont cependant limités aux opérations sur les grilles régulières de points et impossibles à intégrer dans un projet faisant intervenir d'autres structures. Hall *et al.* (2003) emploient les textures 2D pour stocker les valeurs d'une matrice, mais afin d'exploiter les opérations sur les quadruplets dans les nuanceurs, ils décom-

posent la matrice en quatre sous-matrices, chacune étant encodée dans un canal de la texture. Avec un tel agencement, la taille des matrices qu'il est possible de stocker est malheureusement directement limitée à la taille de la texture elle-même.

Dans le cas des tableaux à trois dimensions, plusieurs solutions sont proposées. La première passe par l'utilisation des textures 3D qui s'adaptent naturellement aux tableaux 3D, comme expérimenté par Harris *et al.* (2002). Cependant ils constatent que les performances médiocres de retour de données (voir 2.1.2) pour les textures 3D par rapport à l'équivalent pour les textures 2D ne permettent pas d'avoir de bons résultats. L'alternative qu'ils proposent est d'utiliser plusieurs couches de textures 2D pour représenter le domaine 3D. L'inconvénient majeur de cette méthode est le nombre élevé de passes nécessaire pour la mise à jour du domaine : une grille de 64x64x64 voxels nécessite par exemple 64 passes, ce qui nuit sérieusement aux performances car chaque passe impose un coup fixe supplémentaire.

Afin de résoudre ce problème, Harris *et al.* (2003) proposent une nouvelle technique permettant la mise à jour de l'ensemble du domaine en un seul rendu par l'intermédiaire d'une texture3D dite *plate* (*flat 3D texture*). Cette texture contient côte à côte dans une texture 2D les différentes couches du domaine 3D, selon un agencement illustré par la figure 2.4. Une série de quadrilatères et de lignes, correspondant aux couches 2D et aux domaines limites, sont rendus dans le tampon de pixels avant d'être copiés dans une texture. Cette technique a ensuite été reprise dans d'autres travaux (par ex. Youquan *et al.* (2004)). Cette technique n'est cependant applicable que pour des domaines 3D de petite taille, en raison de la taille maximale des textures 2D, au maximum de 64*64*64 voxels.

Le cas des tableaux à plus de trois dimensions n'a quant à lui pas encore été étudié, essentiellement parce qu'il s'adapte très mal à la mémoire des GPU. Même si cela est techniquement possible, les accès mémoires (retour de données, copie, lecture,

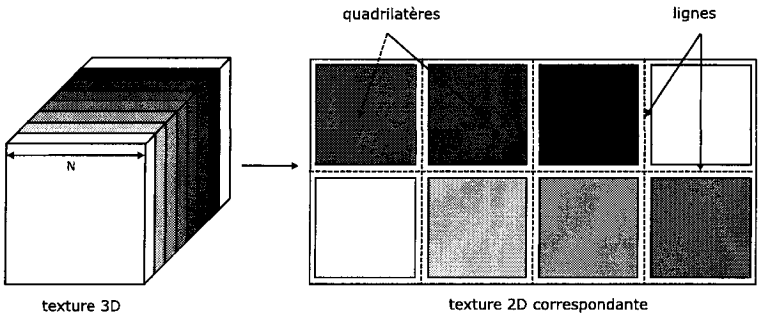

Figure 2.4 Transposition d'un domaine 3D dans une texture 2D

traduction de l'adresse nD->2D) seraient trop coûteux pour offrir des performances intéressantes. Pour ce genre de structures, il est préférable de laisser les calculs au CPU.

2.2.2 Structures creuses

Dans le cas où les tableaux à traiter comportent de nombreux éléments nuls, on adapte généralement d'autres stratégies de mise en mémoire pour se dispenser des informations inutiles ou redondantes.

Dans le cas des matrices, plusieurs techniques existent. Pour les matrices creuses par bande (*sparse band matrix*), Kruger et Westermann (2003) ne stockent par exemple que les vecteurs diagonaux non nuls selon la méthode illustrée à la figure 2.3. Dans le cas d'une matrice creuse aléatoire (*random sparse matrix*), ils utilisent la technique du *Copy-To-Vertex-Array* (c.f. 2.1.2), où chaque élément non-nul hors-diagonal d'une ligne est rendu dans une liste de sommets mise en mémoire sur le GPU. Les registres de coordonnées des sommets sont utilisés pour situer l'élément au sein de sa ligne et le registre de couleur pour enregistrer la valeur de l'élément non nul. Cette méthode permet une utilisation optimale de la mémoire de la carte et minimise les lectures de textures. Étant donné qu'ici c'est le nuanceur de sommets qui est mis à contribution, la puissance de calcul obtenue est cependant moindre

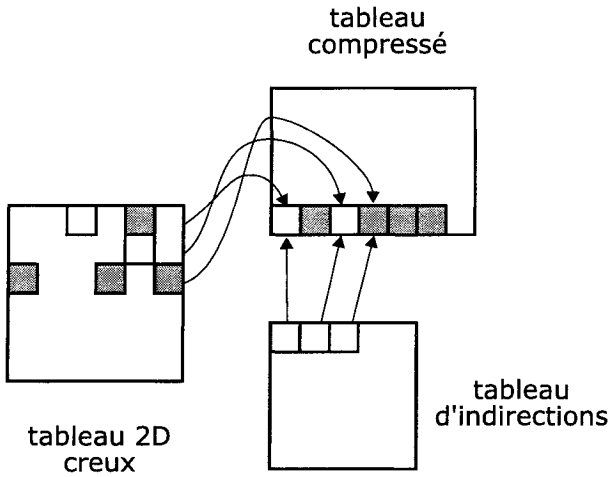

Figure 2.5 Utilisation de textures d'indirection pour le repérage d'éléments dans une texture

par rapport à une technique utilisant le nuanceur de pixels (voir la section 2.5.5).

Bolz *et al.* (2003) préfèrent l'utilisation des textures dépendantes, qui agissent comme *pointeurs* pour la lecture dans une autre texture. Ceci permet de compacter les valeurs non nulles dans une seule texture et de les retrouver facilement par la suite. La figure 2.5 illustre cette technique, où chaque élément i de la texture R pointe vers le premier élément non nul hors diagonale de la i^e ligne de la matrice dans la texture A.

Dans le cas d'un champ 3D creux , Lefohn *et al.* (2004) s'inspirent de la technique des textures adaptatives (Kraus et Ertl (2002)) pour compresser dans une texture 2D les portions d'intérêt du domaine 3D. La correspondance d'un élément du domaine compressé 2D dans le volume 3D se fait par l'intermédiaire de multiples coordonnées de texture, qui contiennent les références à la fois vers sa position dans le domaine 3D et vers celles de ses voisins dans la texture 2D compressée. Comme le montre la figure 2.6, chaque élément de volume est composé de 4 lignes 4 points ainsi qu'un quadrilatère, dont les sommets informent du positionnement dans la texture compressée des données adjacentes dans le domaine 3D. Cette méthode ne

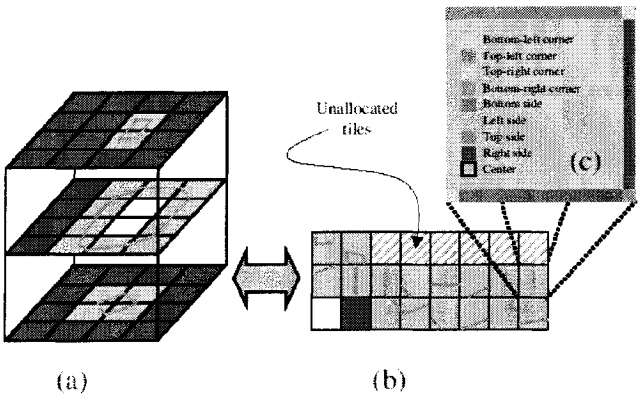

Figure 2.6 Compression d'un domaine 3D creux dans une texture 2D

s'applique cependant qu'aux champs 3D de données *statiques*, car si la connectivité des éléments du volume change, il faut reconstruire toutes les correspondances.

La prochaine section analyse les différentes manières de traiter les flux de données ainsi construits, et explique comment contourner les limitations propres à l'architecture des GPU.

2.3 Stratégies de calcul

2.3.1 Démarche commune

Tous les travaux cités se basent sur le même schéma d'invocation du noyau, illustré par la figure 2.8. Dans un premier temps, une fenêtre d'affichage est mise en place, de même dimension que la texture ou le tableau de sommets destiné à stocker le résultat. Un quadrilatère aligné parfaitement avec la vue orthogonale y est ensuite rendu, chaque pixel dessiné déclenchant l'éxecution d'un noyau. Une technique légèrement différente consiste à dessiner plutôt un triangle contenant la fenêtre d'affichage, ce qui peut améliorer les performances sur certaines cartes en économisant une opération de tramage (figure 2.7).

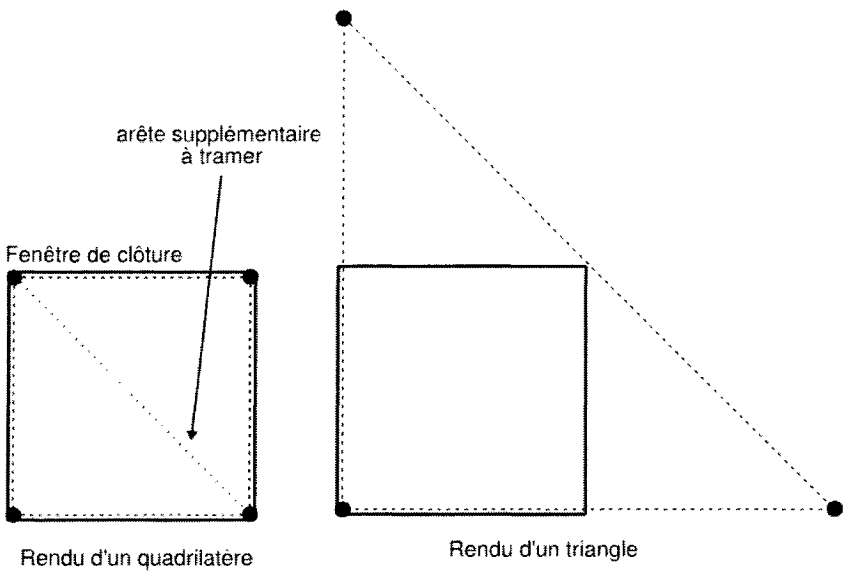

Figure 2.7 Alternatives pour l'invocation du noyau

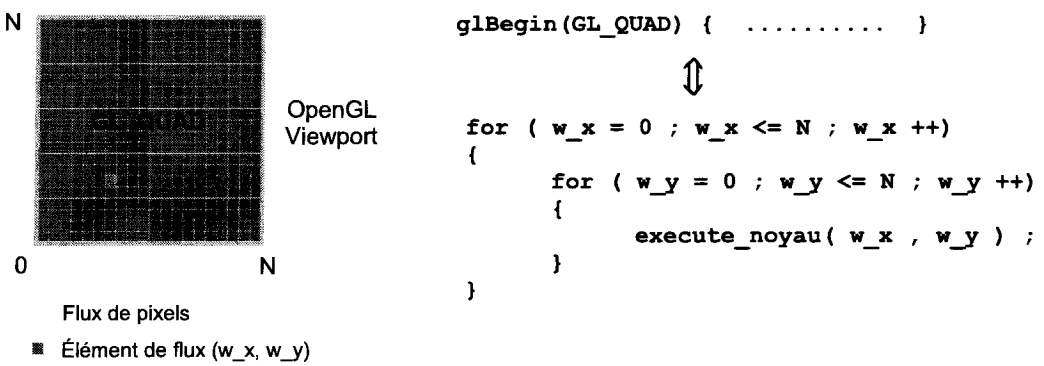

Figure 2.8 Invocation du noyau par le dessin d'une géométrie

Chaque noyau peut ensuite être alimenté en données externes par l'intermédiaire des registres de textures, ou des registres de constantes (voir la liste en annexe I.3). Le rendu du quadrilatère, une fois achevé, peut-être copié ou rendu directement par exemple dans une texture, pour un usage ultérieur dans un autre noyau. Une variante de cette technique est de rendre une grille de sommets à la place d'une grille de pixels, le noyau exécuté devient alors le nuanceur de sommets (voir plus haut les travaux de Kruger et Westermann (2003) pour les structures 2D creuses).

2.3.2 Communication CPU/GPU

Comme il est expliqué dans la section 2.5.4, le transfert de données entre le CPU et le GPU doit être réduit. Cependant, c'est parfois inévitable et il faut alors tâcher de réduire au minimum les données à transférer. Le problème principal est l'impossibilité de compresser les résultats avant de les envoyer au CPU. C'est pourquoi la communication se fait toujours en copiant directement la mémoire vidéo.

Il existe toutefois un exemple de compression de données sur GPU. Lefohn *et al.* (2004), dans leur outil de visualisation de courbes de niveau, utilisent les capacités de *mipmapping* (filtre bilinéaire) du GPU pour compresser un message dans une texture de petite taille (<64ko) afin d'informer le CPU des voxels à activer/désactiver lors du prochain rendu. Le CPU décode cette texture, et appelle le dessin des voxels appropriés. Les textures servent ici de *messagers* entre le CPU et le GPU.

2.4 Algorithmes avancés sur GPU

Même s'ils apparaissent comme des outils puissants et séduisants, les nuanceurs imposent en revanche un grand nombre de contraintes dans leur utilisation (cf. section 1.2.2). Cette partie montre les différentes solutions trouvées pour gérer ces contraintes.

2.4.1 Ramification de code

Le tableau I.5 montre que la grande majorité des GPU ne supporte pas la ramification de code (*branching*), même si celle-ci peut s'avérer indispensable dans certains cas. Certains GPU peuvent cependant simuler cette ramification en exécutant les deux branches, mais en n'autorisant qu'une seule à écrire dans les registres de sortie. Cette méthode est très pénalisante dans le cas où le nombre de branches est élevé car seules quelques instructions parmi les nombreuses exécutées sont finalement utiles. Il en résulte alors une mauvaise utilisation de la puissance de calcul.

Afin de contourner ce problème, une astuce a été proposée quasiment simultanément par Goodnight *et al.* (2003), Harris *et al.* (2003), et Lefohn *et al.* (2004). Cette astuce consiste à diviser le flux en sous-flux (*substreams*), chacun de ces sous-flux étant lié à un noyau qui lui est propre et représentant une des branches du code. Le travail de *tri* des différents cas (sous-flux) est réalisé en amont du pipeline par le CPU. Dans les travaux de Harris, la figure 2.9 illustre comment la ramification s'émule dans le cas d'un calcul sur un domaine 2D avec frontières. Les éléments (pixels) sur les bords sont regroupés en une primitive `GL_LINE` avant d'être envoyés dans le pipeline et liés à un nuanceur de pixels adapté. La même technique s'applique sur les éléments à l'intérieur du domaine regroupés au sein d'un `GL_QUAD`, avec un nuanceur de pixels différent. Lefohn applique le même principe mais sur

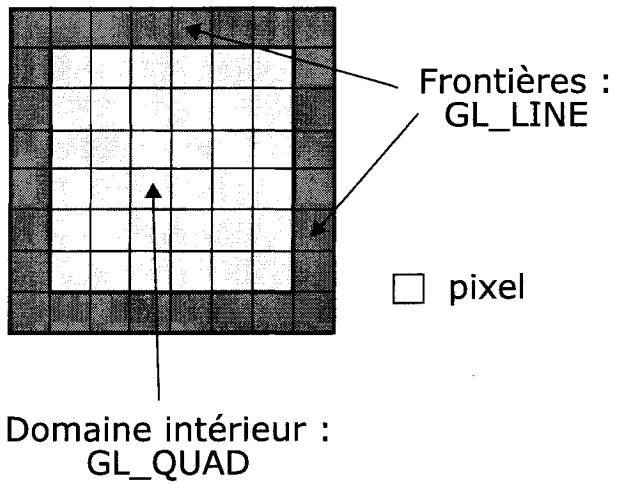

Figure 2.9 Emulation de la ramification de code par l'emploi de sous-flux

un ensemble volumétrique 3D. Cependant, il faut noter que comme le travail de tri des différents flux est effectué par le CPU, toute modification de la nature des flux entraine un nouveau tri et donc une chute de performance.

Purcell *et al.* (2002) proposent une autre technique d'émulation qui fait intervenir le tampon de masque pour sélectionner le noyau à exécuter. En pratique, cela consiste à utiliser le test du tampon de masque et à contrôler le contenu de ce tampon pour activer ou désactiver l'exécution des nuanceurs sur les zones désirées. L'avantage de cette technique est que contrairement à la précédente, les éléments du tampon de pixels ne sont traités par les nuanceurs que s'ils passent le test. Cependant cette méthode ne présente de bonnes performances qu'à condition que les zones d'intérêt constituent des blocs. En effet, étant donné que les nuanceurs de pixels sont éxécutés simultanément sur des blocs de pixels adjacents, une répartition aléatoire de fragments valides entraine une mauvaise utilisation des nuanceurs : le bloc traité risque de comporter beaucoup de fragments inactifs et de laisser inexploité une bonne partie des nuanceurs disponibles.

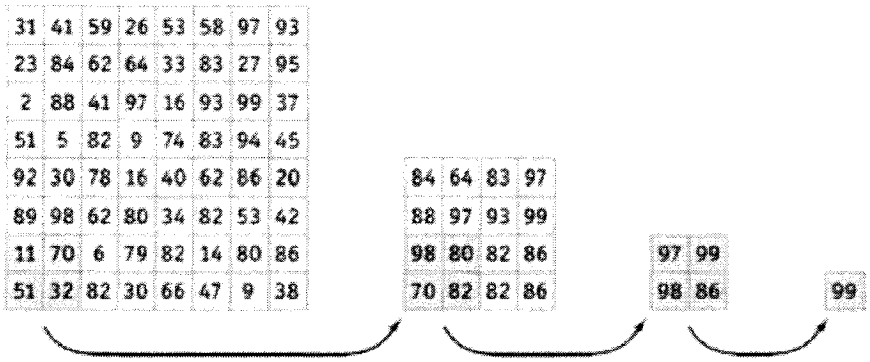

Figure 2.10 Calcul du max par la méthode de réduction

2.4.2 Opérateurs de réduction

Un opérateur de réduction permet d'analyser un flux et de retourner un flux de taille inférieure, comme par exemple l'opérateur *maximum* qui prend en entrée un vecteur de valeurs et retourne un scalaire. Comme le fonctionnement du pipeline comme processeur de flux impose un flux de sortie de taille fixe (à savoir un quadruplet correspondant à la couleur), il est nécessaire d'adapter le traitement du vecteur à ce mode de fonctionnement.

Kruger et Westermann (2003) proposent une technique illustrée à la figure 2.10. Le vecteur à réduire est d'abord transformé en une texture 2D carrée en puissance de deux. A chaque nouvelle passe, un quadrilatère 4 fois plus petit est rendu par l'intermédiaire d'un nuanceur qui consulte la texture du vecteur à réduire et stocke dans chaque pixel la valeur maximale de quatre pixels adjacents du vecteur. Ce rendu est ensuite stocké dans une texture, et subit le même traitement à la passe suivante, et ainsi de suite jusqu'à ce que l'on obtienne une texture de dimension 1x1, qui contient alors le résultat de la réduction. Cette méthode n'est cependant applicable que si les opérations utilisées pour la réduction sont associatives (min, max, somme, multiplication). Notons aussi que le nombre de passes à effectuer dépend de la dimension de la texture initiale à réduire, soit $n - 1$ passes pour une

texture de dimension 2^n.

Horn (2005) améliore ce procédé en l'étendant aux opérateurs non nécessairement associatifs basés sur un filtrage quelconque. Ce filtrage est l'adaptatation sur GPU des travaux de Hillis et Steele (1986) relatifs au scan de flux et ne nécessite pas plus de passes que la technique précédente.

A partir de ces techniques, nous présenterons dans le chapitre suivant une fonction de réduction simple à utiliser, capable de traiter un tableau de longueur quelconque.

2.4.3 Sorties multiples

En raison de la taille fixe du flux de sortie (voir le tableau I.4), les noyaux ne peuvent pas écrire plus qu'un quadruplet à chaque exécution.

Dans le but d'avoir une sortie plus complexe, Buck *et al.* (2004) utilisent une technique basée sur les capacités des compilateurs pour les GPU à éliminer le code inutilisé d'un nuanceur : si un registre temporaire utilisé dans le nuanceur pour stocker un calcul n'est finalement jamais utilisé pour contribuer au registre de sortie, toutes les instructions faisant intervenir ce registre sont automatiquement éliminées par le compilateur. Ils profitent de cette caractéristique des compilateurs pour générer plusieurs copies d'un nuanceur en modifiant juste la ligne responsable de l'écriture du flux de sortie, le compilateur se chargeant de ne garder que les instructions nécessaires. Avec un seul nuanceur, ils génèrent ainsi plusieurs sorties, donc un flux de plus grande taille, chaque élément étant généré dans une passe différente par un nuanceur différent.

Riffel *et al.* (2004) utilisent quant à eux les nouvelles possibilités de la norme 3.0 des nuanceurs, qui permettent aux nuanceurs de pixels d'écrire non plus une, mais quatre couleurs différentes directement en matériel. Cette caractéristique baptisée

MRT (*Multiple Render Targets*) est actuellement disponible uniquement sur les cartes Nvidia Geforce de la série 6. Cette méthode est très efficace car quatre fois moins de passes sont requises.

2.4.4 Partitionnement de code

Une autre contrainte des nuanceurs est le nombre limité d'instructions et de registres disponibles. Il arrive que le code d'un nuanceur soit matériellement irréalisable pour ces raisons.

Chan *et al.* (2002) pallient à ce problème en proposant un système capable de fragmenter automatiquement un nuanceur trop complexe en plusieurs sous-nuanceurs, chacun étant appelé sur des rendus successifs, en réutilisant éventuellement le résultat d'un rendu antérieur. Leur système décompose le nuanceur en un arbre de nuanceurs élémentaires puis essaye de le réagencer afin de minimiser le nombre de passes nécessaires. D'autres versions plus élaborées permettent aussi d'utiliser la technologie MRT (Foley *et al.* (2004)).

2.4.5 Opérateurs de diffusion

Comme évoqué dans la section 1.2.2, les noyaux ne permettent pas de contrôler où est écrit le résultat de l'exécution d'un noyau : il est par exemple impossible de demander au GPU de diffuser la couleur du pixel en cours de traitement traité aux pixels voisins, et cela en raison de la nature *SIMD* des nuanceurs de pixels.

Pour une implémentation sur GPU, Buck (2005) propose d'émuler la diffusion en la convertissant en une série d'opérations de *collecte* (*gather operations*), comme illustré sur la figure 2.12. Au lieu de diffuser chaque force de rappel sur les deux masses connectées au ressort, on effectue une deuxième passe pour l'ensemble des

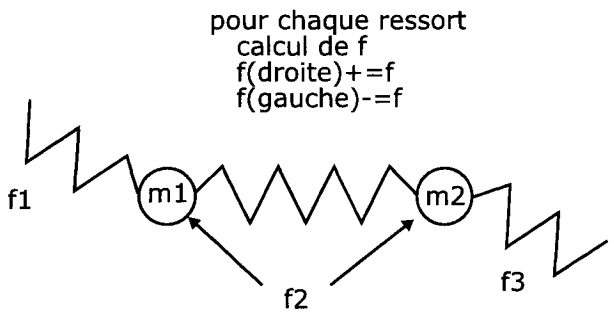

Figure 2.11 Diffusion au sein d'un flux de ressorts

masses à laquelle on rajoute les forces avoisinantes (figure 2.12). Cette technique n'est cependant valide que si la connectivité entre les masses est statique.

Coombe et Harris (2005) appliquent ce principe au calcul de la radiosité d'une scène 3D par la méthode du raffinage progressif originellement développé par Cohen *et al.* (1988), qui consiste à déterminer progressivement l'illumination de la scène en ne considérant à chaque itération que la surface émettrice de plus grande énergie, jusqu'à ce que l'énergie restante soit inférieure à un certain seuil. Les auteurs, afin d'adapter l'algorithme au fonctionnement du GPU, préfèrent lancer les calculs pour chacune des surfaces réceptrices et faire un test de visibilité de la surface émettrice, plutôt que de lancer un seul calcul pour la surface émettrice qui écrirait sur chacune des surfaces qu'elle voit. La première solution, même si elle nécessite en pratique plus de calculs, est parallélisée beaucoup plus efficacement et s'avère plus rapide que la deuxième, qui doit *diffuser* le résultat du calcul dans de multiples flux de sortie (un pour chaque surface réceptrice).

Purcell *et al.* (2002) proposent une autre alternative, qui consiste à utiliser non pas le nuanceur de pixels mais le nuanceur de sommets pour pouvoir contrôler l'endroit où sont écrits les résultats dans la mémoire. En effet, un nuanceur de pixels ne permet malheureusement pas au programmeur de décider où le pixel va être rendu à l'écran. La méthode proposée consiste à rendre une liste de points à l'intérieur d'une fenêtre 3D par l'intermédiaire d'un nuanceur de sommets. La position des

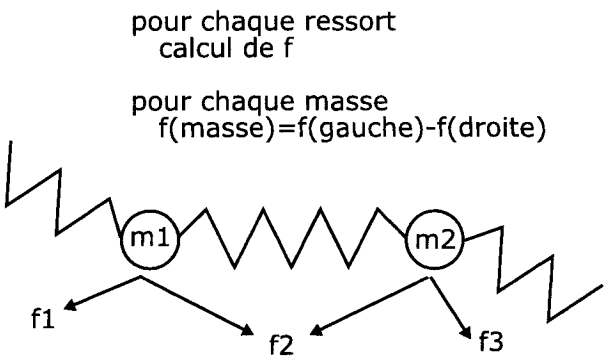

Figure 2.12 Emulation de la diffusion par la collecte

sommets dessinés à l'écran est déterminée à l'intérieur même du nuanceur, ce qui permet d'écrire potentiellement n'importe où à l'écran donc dans la mémoire vidéo (le contenu de la fenêtre 3D est ensuite copié dans une texture). Même si cette méthode est très pratique, elle sous-exploite en revanche la puissance du pipeline, car ni le trameur ni les nuanceurs de pixels ne sont mis à contribution au cours du processus. En outre, comme la quasi-totalité des nuanceurs de sommets n'ont pas accès en lecture aux textures, la quantité d'informations transmissibles au nuanceur est très limitée.

2.5 Analyse de performance

Cette section résume les principaux facteurs finalement identifiés comme étant au coeur de l'optimisation du calcul générique. Ils serviront également de critères dans notre implémentation et dans l'analyse de ses performances.

2.5.1 Intensité arithmétique

La figure 2.13 traçant l'évolution de la bande passante et de la puissance de calcul pour les dernières puces du fabricant ATI montre que la vitesse d'exécution des opérations arithmétiques excède de plus en plus vitesse de transfert depuis la

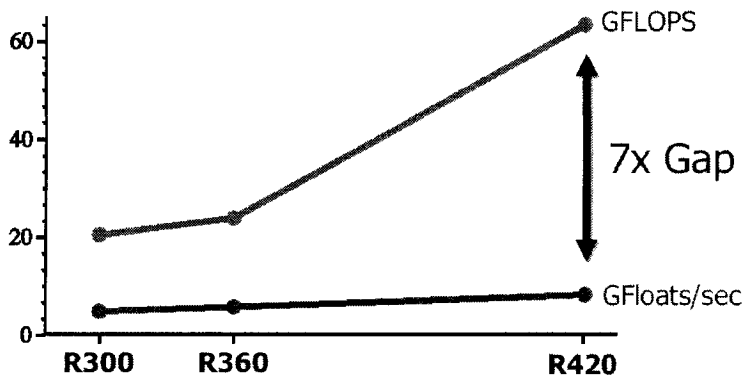

Figure 2.13 Evolution de la puissance de calcul et de la bande passante, ATI (GPGPU (2004))

mémoire vidéo des données subissant le calcul, jusqu'à un facteur 7!

Dally *et al.* (2003) notent qu'une telle différence a un gros impact sur la manière d'utiliser les flux d'entrée pour le calcul sur GPU, et expliquent qu'afin d'exploiter au maximum la puissance de calcul du GPU, il faut s'assurer que les flux à transmettre n'excèdent pas la bande passante maximale offerte par la mémoire vidéo. Partant de cette constatation, ils définissent un indice de *qualité* du noyau, appelé intensité arithmétique I, dont la valeur se calcule de la manière suivante :

$$I = \text{Durée des instructions arithmétiques} \,/\, \text{Durée des transferts de données}$$

Ils arrivent à la conclusion que pour qu'un algorithme soit intéressant à adapter sur le GPU, il est nécessaire que le gain de temps sur le CPU aquis par le GPU sur les instructions arithmétiques soit suffisant pour au moins couvrir le temps de transfert dû à l'envoi et à la réception des données sur le GPU.

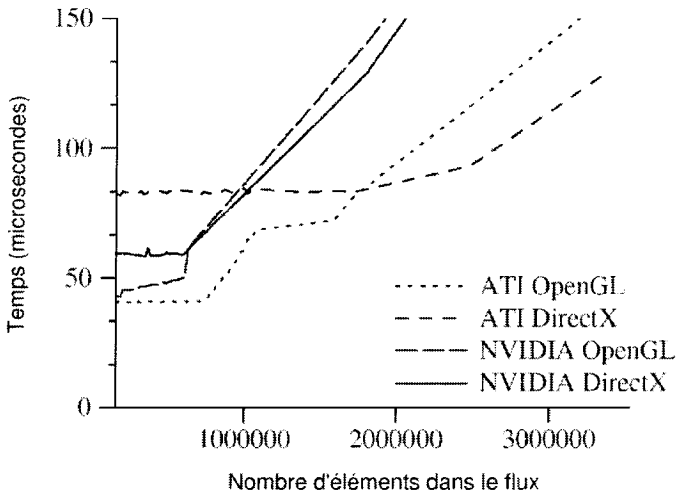

Figure 2.14 Influence de la taille des flux sur les performances, Buck *et al.* (2004)

2.5.2 Taille des flux

Un deuxième facteur important dans la performance des GPU est la taille des flux qui leur sont envoyés, car Buck *et al.* (2004) montrent qu'il existe un coup fixe associé à l'invocation d'un noyau par le CPU. Pour un grand nombre de petits flux, ils constatent que le CPU n'est pas assez rapide pour garder le GPU actif à plein régime. En revanche, pour des flux de grandes tailles, les noyaux sont invoqués moins souvent et libèrent donc le CPU, qui est alors capable d'alimenter en continu le GPU et de le maintenir à pleine puissance.

La figure 2.14 illustre ceci et montre l'évolution du temps moyen requis pour effectuer un noyau de 43 instructions en fonction de la taille des flux qui lui sont envoyés. Aux faibles tailles, ils constatent que le temps moyen est maintenu constant à cause du CPU qui limite la fréquence d'invocation des noyaux, d'où un nombre constant d'instructions exécutées et donc du temps d'exécution. En revanche, pour les tailles de flux plus grandes, ils mesurent une augmentation proportionnelle à la taille des flux dû au fait que le GPU est continuellement alimenté et exécute un nombre

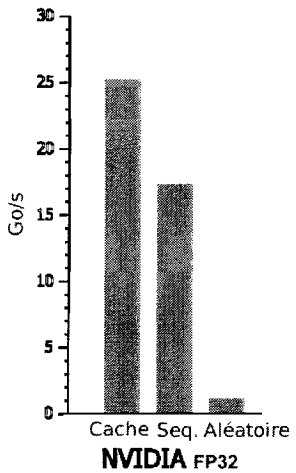

Figure 2.15 Influence de la localité sur la bande passante, GPGPU (2004)

toujours croissant d'instructions. Ils en arrivent à la conclusion qu'il est préférable de travailler sur des flux (textures ou tableaux de sommets) de grandes tailles pour rendre le calcul sur GPU réellement profitable.

2.5.3 Influence de la mémoire cache

Un autre facteur très important dans la comparaison CPU/GPU concerne la localité des données. La figure 2.15 en montre l'influence sur la bande passante des données sur le GPU de type *float* codées sur 32 bits (*Seq* fait référence à un accès mémoire séquentiel des données). Entre un accès du cache et un accès aléatoire de la mémoire vive, la différence est d'un facteur 25 !

Sur un Pentiun IV 3.0GHz, la bande passante maximum est de 44 Go/s pour la lecture du cache de niveau 1 et de 6 Go pour l'accès séquentiel de la mémoire. Quand à l'accès en mémoire vive, il dépend de la qualité de la mémoire RAM installée : < 0.4 Go pour la SDRam, < 0.7 Go pour la DDR2, et < 0.8 Go pour la RDRam.

Cette limitation du matériel graphique est abordée de manière plus détaillée par Fatahalian *et al.* (2004), qui analysent expérimentalement les vitesses de transfert du cache et arrivent à la conclusion que pour des calculs faisant intervenir de multiples accès à la mémoire vidéo, le facteur limitant est le plus souvent la faiblesse de la bande passante du cache de bas-niveau, qui entraîne une sous-exploitation des unités de calcul qu'à 20 % de leur capacité théorique maximum.

Ils déplorent qu'actuellement le GPU ne permette pas comme les CPU de contrôler le contenu du cache qui est automatiquement géré par le pilote et reste encore fermé au programmeur. Il est donc impossible d'utiliser intelligemment le cache afin d'optimiser les performances, contrairement à ce que peuvent faire sur le CPU certaines librairies de calcul algébrique comme Whaley et Dongarra (1998) qui manipule le cache de bas niveau L1 et L2 pour stocker des résultats intermédiaires.

2.5.4 Localité des données

Le coût du rapatriement des données depuis la mémoire du GPU sur la mémoire vive principale est inévitable si l'on veut récupérer les résultats de son calcul.

Sur les dernières cartes NVidia ce transfert est limité à 600 Mo/s et s'avère être un autre facteur limitant l'exploitation de la puissance de calcul. Les algorithmes nécessitant un rapatriement très fréquent du GPU vers le CPU (et réciproquement) souffrent beaucoup de cette limitation.

Une grande attention sera donc portée sur la fréquence et le volume des échanges entre le GPU et le CPU, afin de garantir de bonnes performances et maximiser le pourcentage de temps de calcul passé sur le GPU. Ceci servira d'indicateur de la performance de nos opérateurs.

2.5.5 Équilibrage de la charge de travail sur le pipeline

Afin de tirer profit au maximum de la puissance de calcul de chacun des noyaux, le calcul doit tenir compte des forces et faiblesses des deux familles de noyaux disponibles :

Concernant les nuanceurs de sommets, les points forts sont :

- **Capacité supérieure de registres** : chaque élément de flux peut avoir un nombre de registres supérieur par rapport au nuanceur de pixels, ce qui peut s'avérer très utile pour le calcul. Par exemple, dans le cas d'une GeforceFX, un sommet peut contenir 16 *attributs* (chaque attribut étant un quadruplet), alors qu'un pixel ne peut en contenir que 10.

- **Interpolation linéaire matérielle** : le flux de sortie du nuanceur de pixels est dirigé vers le trameur qui interpole automatiquement tous les attributs avant de les passer au nuanceur de pixels. C'est une puissance de calcul non négligeable, d'autant plus qu'elle peut alléger significativement la charge sur le nuanceur de pixels. Lefohn *et al.* (2004) allègent par exemple la charge du nuanceur de pixels en allouant toutes les interpolations linéaires au trameur par l'intermédiaire du nuanceur de sommets.

Cependant, dans le cadre du calcul générique, ce sont les nuanceurs de pixels qui sont les plus adaptés pour les raisons suivantes :

- **Plus grande puissance de calcul** : le nombre maximum d'opérations par seconde est très supérieur à celui des nuanceurs de sommets. Cela est relié au fait que le débit de remplissage de sommets (*vertex fill rate*) est inférieur au taux de remplissage de pixels jusqu'à un facteur 10 sur les dernières cartes nVidia,

comme le montre la figure 1.1 où le nombre d'unités de calcul pour les sommets est de 6 unités comparé à 16 unités en parallèles pour les pixels. Ceci s'explique essentiellement par le fait que, dans un jeu, on a habituellement beaucoup plus de pixels que de sommets à dessiner.

- **Accès à la mémoire vidéo** : seuls les nuanceurs de pixels ont l'accès en écriture à la mémoire vidéo du GPU. Les calculs doivent donc nécessairement s'achever là si l'on désire les conserver.

2.5.6 Stratégie pour une bonne exploitation du GPU

Tous ces facteurs limitants abordés précédemment dans 2.5 permettent finalement de dresser le portrait type d'une application exploitant au mieux le GPU.

- **Traitement de gros volumes de données** : l'application doit travailler sur des données fournies par le CPU organisées en quelques gros volumes.

- **Minimisation du trafic entre le CPU et le GPU** : l'application doit envoyer et recevoir le moins souvent possible des données vers le GPU. Idéalement, on envoie les données seulement au début pour initialiser le calcul, et on ne récupère les données qu'à la fin, si besoin est. Le calcul doit nécessiter le moins d'informations supplémentaires possibles au cours de l'exécution (registres de constantes).

- **Intensité de calcul** : les noyaux employés doivent contenir plus de calculs arithmétiques que d'accès mémoire.

Ces facteurs conditionneront l'implémentation de nos opérateurs dans le chapitre suivant et serviront de critères pour évaluer leur performance.

CHAPITRE 3

ALGÈBRE LINÉAIRE SUR GPU : IMPLÉMENTATION

3.1 Objectifs

3.1.1 Hypothèse

La revue de littérature a montré que la grande variété des problèmes pouvant être traités par le GPU était très variées, tout comme les solutions trouvées pour contourner les limitations liées aux contraintes de la programmation par flux. Cependant, parmi ces travaux on constate que les solutions choisies et implémentées sont toujours *spécifiques* aux problèmes qu'elles traitent et ne peuvent pas s'appliquer dans un cadre différent sans une *modification* de leur code source. En plus de cela, même si ces modifications étaient envisageables, elles nécessiteraient des connaissances avancées en infographie que peu de scientifiques possèdent.

Partant de ce constat, notre hypothèse est qu'il est possible de masquer cet aspect technique et de rendre le calcul sur GPU simple, accessible et applicable à une grande variété de problèmes. L'objectif général sera ainsi d'implémenter, pour une certaine classe de problèmes, une solution *générale* qui puisse être utilisée *sans modification.*

Plus spécifiquement, les objectifs seront :

- définir un ensemble d'opérateurs algébriques d'utilisation fréquente qui peuvent être programmés sur GPU,

- développer une librairie de ces opérateurs et valider son implémentation,

- évaluer et comparer l'utilisation des versions GPU et CPU de ces opérateurs dans le cadre d'algorithmes de simulation numérique,

- et dégager les avantages et les limitations du calcul sur GPU.

3.1.2 Méthodologie

Afin d'être applicable à des problèmes différents, il est nécessaire de choisir des algorithmes qui interviennent eux-mêmes dans des domaines très différents. Le calcul algébrique linéaire est un choix approprié en raison de son utilisation dans des applications très variées comme par exemple le traitement d'images, la simulation numérique de phénomènes physiques ou encore les mathématiques pures.

Afin d'en faire des outils accessibles, nous encapsulerons l'implémentation de ces algorithmes de calcul algébrique sur GPU en une série d'opérateurs élémentaires s'intégrant dans une interface simple, dont l'utilisation ne nécessite que des connaissances de base en C++.

Pour démontrer que ces opérateurs sont applicables à des problèmes différents, deux simulations sur le GPU seront réalisées à l'aide de ces outils puis les résultats obtenus seront analysés. La première simulation fait intervenir un système de particules 3D soumises à des contraintes, et la deuxième utilise l'opérateur de gradients conjugués pour résoudre un système linéaire issu d'une modélisation par différences finies. On ne prétend pas pouvoir traiter tous les problèmes possibles, mais traiter des problèmes *différents* en utilisant les *mêmes* outils.

A l'instar des librairies connues d'algèbre linéaire comme *BLAS* ou *LAPACK* (An-

derson *et al.* (1999)), nous allons définir et décrire un ensemble d'opérateurs algébriques à émuler sur le GPU. Étant donnée la grande variété des opérateurs offerte par ces librairies, nous ne nous intéresserons qu'aux opérateurs élémentaires.

3.2 Opérandes

Il s'agit dans un premier temps de définir les opérandes du calcul, c'est-à-dire les structures sur lesquelles les opérateurs s'appliqueront. Dans le cadre d'une implémentation sur GPU, il faut donc déterminer comment mettre en mémoire ces opérandes du CPU vers le GPU.

3.2.1 Vecteurs

Le premier type d'opérande est le vecteur, autrement dit un tableau à une dimension et de taille quelconque. Nous sommes ici dans la situation décrite dans chapitre précédant à la section 2.2.1. La stratégie d'encodage du vecteur et le transfert sur le GPU reprend celle adoptée par Kruger et Westermann (2003).

En raison des limitations du matériel graphique, la précision des données du vecteur à transférer ne peut pas excéder 32 bits, à savoir l'équivalent d'un nombre en virgule flottante en précision simple (*simple precision floating point number*) selon la norme IEEE. Cette précision n'est actuellement supportée que par les plus récentes cartes graphiques nVidia, par l'intermédiaire de l'extension propriétaire `GL_NV_FLOAT_NV` qui permet d'utiliser des textures avec 32 bits de précision par canal. Il est aussi possible d'utiliser une précision de 16 bits par l'intermédiaire du type *half*, mais dans le cadre d'une simulation cela est généralement insuffisant et peut entraîner des instabilités.

L'implémentation proposée ne fonctionne pour le moment que sur le matériel Nvidia car elle repose grandement sur l'utilisation de ces extensions. Les textures employées pour stocker les valeurs du vecteur sont carrées en puissance de deux et de dimension adaptée au vecteur à stocker. Si jamais le vecteur ne remplit pas complètement la texture, cette dernière est complétée par des valeurs nulles. Ceci peut entraîner une *perte* dans l'utilisation de la mémoire vidéo, mais celle-ci reste négligeable pour un nombre total raisonnable de vecteurs stockés.

Dans la mesure où les opérateurs sur GPU sont initiés par le CPU, il faut que ce dernier puisse avoir une référence vers les vecteurs stockés sur la mémoire vidéo du GPU. Un vecteur est ainsi défini en mémoire vive comme une structure contenant des informations sur sa taille, la résolution de la texture correspondante, l'identificateur de la texture associée sur le GPU ainsi qu'un index, dont l'utilité sera expliquée plus tard.

```
structure gpuVec
{
int taille // nombre d'éléments dans le vecteur
int résolution // résolution de la texture correspondante sur le GPU
int texID // identificateur OpenGL de la texture
int index // position du vecteur au sein de la matrice, le cas échéant
}
```

La construction d'un vecteur se fait simplement par l'appel d'un constructeur, auquel on passe un pointeur vers le tableau des données à stocker ainsi que la taille de ce tableau :

```
float *data = new float[400];
.... remplissage des valeurs de data ....
gpuVec *V = new gpuVec(data,400); // mise en mémoire vidéo
```

```
delete[] data;
```

La mise en mémoire vidéo se fait automatiquement par un appel à `glTexImage2D` et en lui passant un pointeur sur le tableau de valeurs à stocker. Le type de la texture est une fois de plus spécifique au matériel nVidia (`GL_TEXTURE_RECTANGLE_NV`) ainsi que le format de 32 bits par canal (`GL_FLOAT_RGBA32_NV`). Les valeurs consécutives du vecteur sont ensuite regroupées par quadruplets au sein d'un texel afin de tirer profit des opérations vectorielles des GPU.

Une fois le transfert en mémoire vidéo effectué, l'espace occupé en mémoire vive par le tableau peut être libéré et remplacé par la structure décrite plus haut.

3.2.2 Matrices

Plutôt que de considérer une matrice directement comme une texture 2D (Larsen et McAllister (2001)), on considère ici une matrice comme un ensemble de vecteurs, donc un ensemble de textures 2D. Selon la nature de la matrice considérée, le choix de décomposition de la matrice en un groupe de vecteurs peut différer. Cependant, dans les deux cas, la manière dont la matrice est mise en mémoire RAM respecte la structure suivante :

```
structure gpuMat
{
int taille // dimension nxn de la matrice
bool bande // matrice pleine ou bande?
map<int,int> table // table des vecteurs composant la matrice
}
```

où `taille` est la dimension de la matrice, `bande` spécifie si la matrice est bande ou pleine, et `table` est une map STL faisant correspondre un index avec l'identificateur

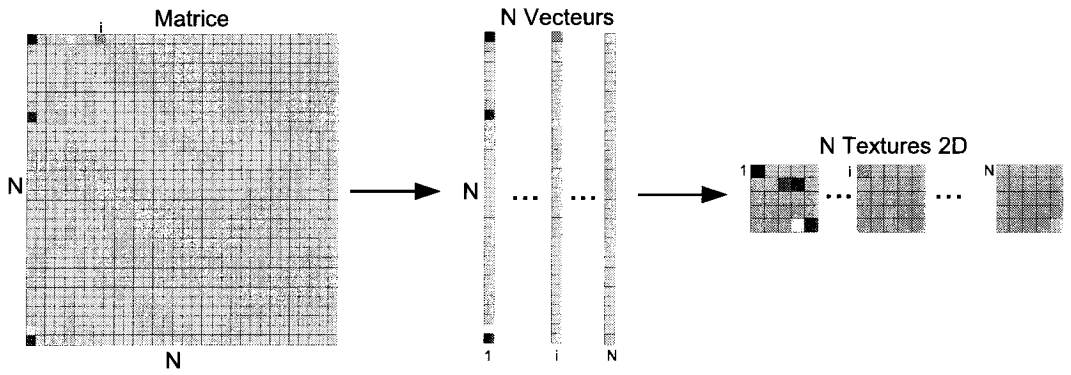

Figure 3.1 Décomposition d'une matrice pleine en un ensemble de vecteurs

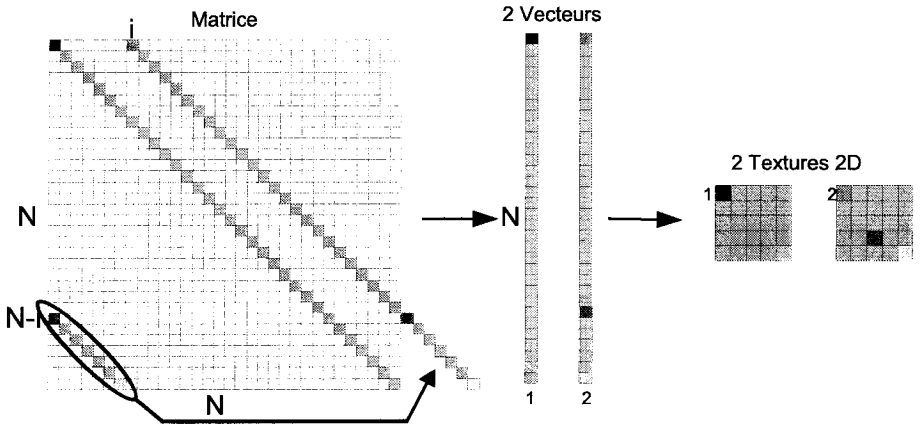

Figure 3.2 Décomposition d'une matrice bande en un ensemble de vecteurs

de texture du vecteur correspondant. Dans le cas d'une matrice pleine, les index font référence aux colonnes de la matrice, le nombre d'index étant donc égal à la taille de la matrice, comme l'illustre la figure 3.1. S'il s'agit d'une matrice bande, `table` fait le lien entre chaque diagonale non nulle et son vecteur correspondant (figure 3.2).

La construction de la matrice se fait par l'intermédiaire d'un constructeur, auquel il faut transmettre la taille de la matrice désirée, ainsi que sa nature (bande ou pleine). Ensuite le remplissage des valeurs est réalisé en appelant une des fonctions implémentées à cet effet dans la classe gpuMat. Une fois la matrice remplie, la construction de la table de vecteur doit être déclenchée par l'utilisateur. Voici un

exemple de la construction d'une matrice tribande 1024*1024 :

```
float data = new float[1024];
.... remplissage des valeurs de data ....
gpuMat *M = new gpuMat(1024,true);
M->setDiag(data,0);
M->setDiag(data,1);
M->setDiag(data,-1);
delete[] data;
```

Voyons maintenant comment les opérateurs exploitant ces structures ont été implémentés.

3.3 Opérateurs algébriques

3.3.1 Combinaison linéaire

C'est l'opérateur le plus simple, prenant en argument deux structures vecteurs et produisant un vecteur résultat. Cette opération peut être une combinaison linéaire, ou même une multiplication terme à terme, l'important étant que le résultat soit lui-même un vecteur.

Une des opérations implémentées a été la combinaison linéaire de deux vecteurs :

$$gpuVecAdd(float\ \alpha, Vec\ V_1, float\ \beta, Vec\ V_2, , Vec\ V_3)$$
$$V_3 \Leftarrow \alpha * V_1 + \beta * V_2$$

La figure 3.3 illustre comment s'effectue le processus sur le GPU. Tout d'abord un quadrilatère aligné avec la fenêtre de visualisation est rendu dans un tampon de

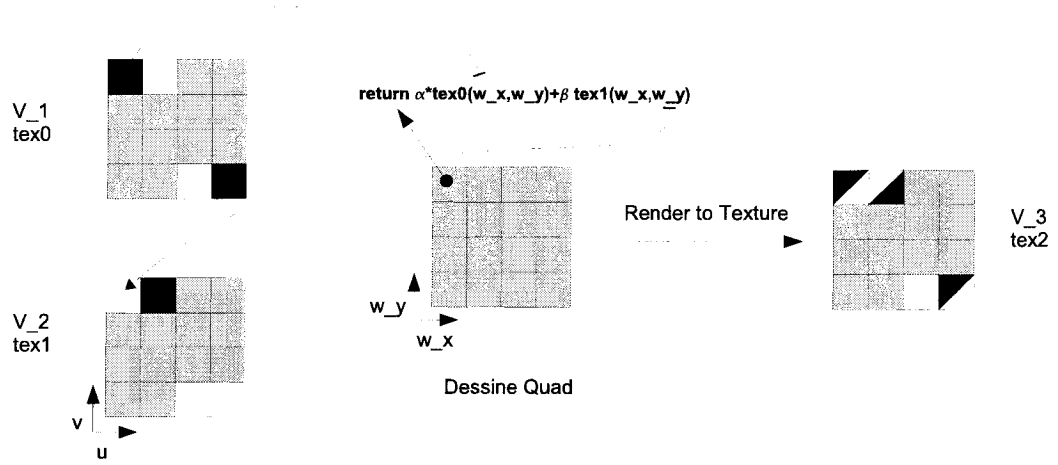

Figure 3.3 Combinaison linéaire de deux vecteurs sur GPU

pixels hors-écran, de même dimension en pixels que la texture destinée à stocker le résultat. Ceci reprend la méthode commune à tous les travaux de calcul générique tel que mentionnés à la section 2.3.1.

Chaque pixel du quadrilatère est ensuite traité par un nuanceur de pixels auquel on transmet les identificateurs de textures des deux opérandes, les coefficients multiplicatifs α et β, ainsi que la position interpolée du pixel P dans l'espace fenêtre $P(w_x, w_y)$. Tous ces paramètres sont spécifiés par le CPU à l'initialisation du nuanceur par l'intermédiaire des registres de constantes et des registres de textures tels que présentés à la section I.

Le pixel P est ensuite utilisé en tant que coordonnée de texture afin de lire dans les textures des deux vecteurs et d'en récupérer les valeurs avant d'y effectuer les opérations demandées. Le code Cg utilisé pour cette opération est disponible en annexe II. Ce procédé se répète ainsi pour chaque paire de quadruplets de valeurs des vecteurs V_1 et V_2.

Une fois le quadrilatère complètement rendu, le contenu du tampon de pixels est copié dans une texture grâce à un appel à `glCopyTexImage2D`. Une nouvelle struc-

ture vecteur V_3 est alors créée, de même taille et de même résolution que les deux vecteurs opérandes et se voit attribuer l'identificateur de la texture. Ce nouveau vecteur peut alors servir à son tour dans d'autres opérateurs.

3.3.2 Produit scalaire

L'opérateur de produit scalaire implémenté peut se décomposer en deux phases : une opération de transformation et une opération de réduction, suivant que le résultat de l'opérateur est un Vecteur de même dimension ou de dimension moindre. Dans le cas du calcul d'un produit scalaire de deux vecteurs V_1 et V_2, on appelle l'opérateur de la manière suivante :

$$gpuVecScal(Vec\ V_1, Vec\ V_2)$$

Voyons comment le calcul de ce produit scalaire se décompose en plusieurs passes. Dans une première passe, on applique à V_1 et V_2 un opérateur de transformation, dont le principe de fonctionnement est similaire à l'opérateur de combinaison linéaire. Cependant au lieu de faire calculer au noyau une combinaison linéaire des quadruplets, on calcule les 4 produits scalaires des quatre quadruplets adjacents de V_1 et V_2 et on les stocke dans un quadruplet du vecteur résultat V_3 de dimension quatre fois plus petite. La figure 3.4 montre comment les données sont agencées dans le vecteur du résultat intermédiaire. L'avantage de cette technique est de combiner en même temps un produit scalaire partiel et une étape de la réduction au sein d'une même passe.

À partir de ce résultat intermédiaire, on effectue plusieurs passes sucessives pour appliquer à V_3 une série de réductions et obtenir finalement le produit scalaire. La méthode utilisée est similaire à celle décrite dans le paragraphe précédent, mais est maintenant appliquée à un seul vecteur et répétée itérativement. A chaque passe,

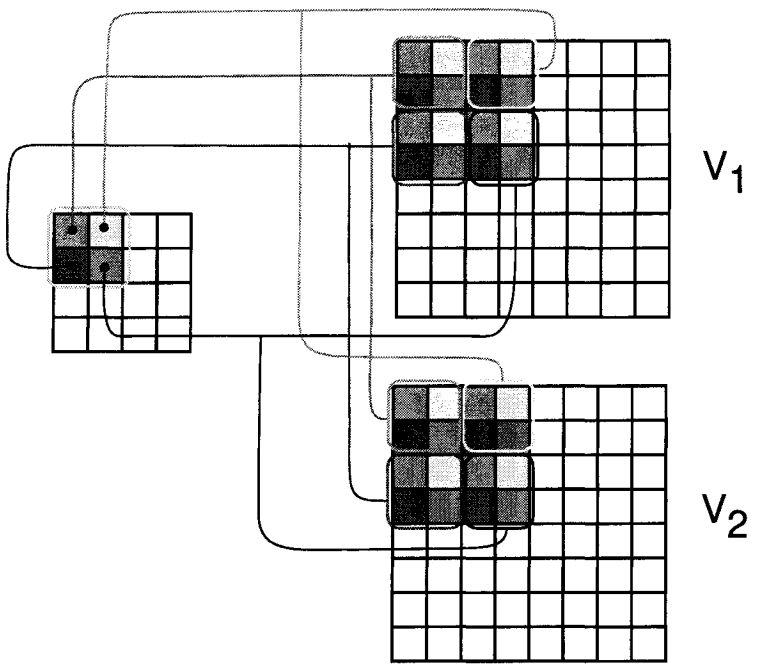

Figure 3.4 Produit scalaire partiel

on calcule les 4 sommes de 4 quadruplets adjacents puis on les transmet dans un quadruplet du vecteur destination de dimension 4 fois plus petite, comme illutré par la figure 3.5. À la passe suivante, le vecteur destination sert à son tour de vecteur source.

À chaque passe, on rend donc un quadrilatère de côté deux fois plus petit jusqu'à ce que le calcul se termine lorsque l'on on ne dessine plus qu'un seul pixel. On récupère ensuite ce pixel en mémoire, puis le CPU somme ses 4 composantes pour finalement obtenir la valeur du produit scalaire. Ainsi pour un vecteur de dimension 2^n, $n-1$ passes sont requises pour effectuer le produit scalaire.

3.3.3 Produit Matrice pleine / Vecteur

Etant donné que chaque matrice est considérée comme un ensemble de vecteurs, un opérateur Matrice/Vecteur est en réalité un opérateur Vecteur/Vecteur élaboré.

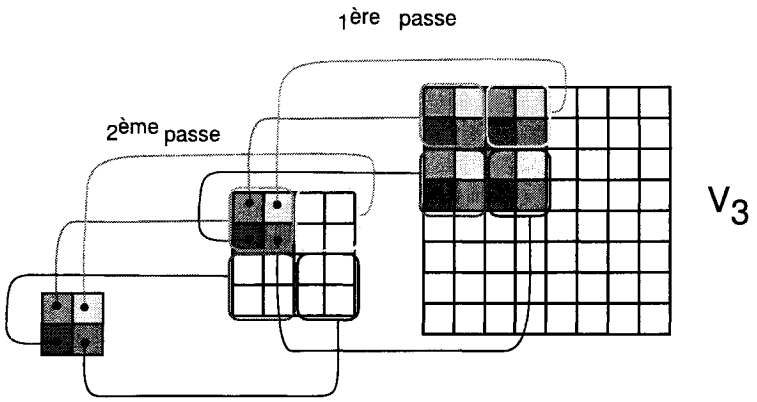

Figure 3.5 Réduction du résultat intermédiaire

L'opérateur qui a été implémenté est une multiplication matrice/vecteur additionné d'un second vecteur :

$$gpuMatOp(Mat\ M, Vec\ V, Vec\ V', Vec\ Res)$$
$$Res \Leftarrow M * V + V'$$

En raison des limitations matérielles du GPU, il est impossible de faire le rendu en une seule passe, car le nombre de textures utilisables par un nuanceur est limité à 16 dans notre cas. On doit donc décomposer cette opération en une série d'opérations de multiplication vecteurs/vecteurs dans des passes séparées. Dans un souci d'optimisation, le fonctionnement des opérateurs a cependant été conçu de manière à minimiser le nombre de passes nécessaires au calcul.

Dans le cas d'une matrice pleine, la multiplication se fait de manière assez naturelle, mais requiert un nombre de passes égal à la dimension de la matrice. La raison qui nous oblige à faire plusieurs passes est le fait que le noyau utilisé ne peut pas écrire plus de quatre valeurs à la fois. La figure 3.6 présente une explication graphique du processus.

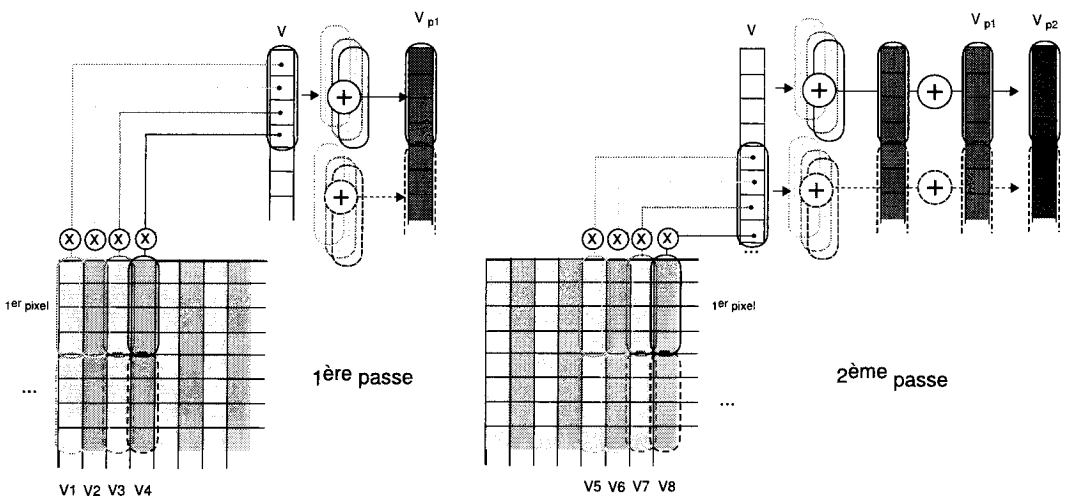

Figure 3.6 Multiplication d'une matrice pleine et d'un vecteur

Au cours de la première passe, on groupe les quatre premiers vecteurs V_1, V_2, V_3 et V_4 de la matrice et le noyau multiplie chacun de leur quadruplet respectivement par les valeurs R,G,B et A du premier quadruplet du vecteur V à multiplier. Chaque ensemble de quadruplets ainsi obtenu est ensuite sommé et placé dans un vecteur résultat intermédiaire V_{p1}. Les références aux vecteurs intervenant dans le calcul sont passées au noyau par le CPU par l'entremise des registres des textures correspondants aux V_i. Les éléments de V avec lesquels multiplier les quatres colonnes sont passés au noyau par l'intermédiaire des coordonnées de texture pointant à l'emplacement du quadruplet au sein de la texture stockant V.

A la passe suivante, on répète le même processus, mais en considérant les quatres colonnes suivantes, à multiplier cette fois-ci avec le deuxième quadruplet de valeurs de V. Après cette multiplication, on additionne le vecteur résultat avec le résultat intérmédiaire V_{p1} obtenu à la passe précédente. On obtient alors un second résultat intermédiaire à considérer pour la passe suivante, et ainsi de suite, jusqu'à ce que toutes les colonnes de la matrice aient été traitées.

A la dernière passe, on additionne le résultat obtenu avec le vecteur V', pour obtenir le résultat final. Si on note N la dimension de la matrice, l'opérateur de multiplication nécessite au total $N/4$ passes.

Chaque passe est en fait une combinaison linéaire des quatres vecteurs colonnes, dont les coefficients multiplicatifs sont donnés par un quadruplet de V. La lecture progressive des quadruplets des quatres colonnes ainsi que l'écriture du quadruplet résultat correspondant se fait donc de la manière décrite à la figure 3.3.

La solution proposée ici améliore la proposition faite par Kruger et Westermann (2003) qui ne traite qu'une colonne à la fois, et nécessite un nombre de passes quatre fois plus élevé. Le code Cg du noyau utilisé figure en annexe II.

3.3.4 Produit Matrice bande / Vecteur

Dans le cas où l'on désire multiplier un vecteur avec une matrice bande, le calcul s'effectue d'une manière très différente. En effet, dans le cas d'une matrice bande où seuls quelques vecteurs diagonaux sont non nuls, la multiplication matrice/vecteur se réduit à quelques multiplications vecteur/vecteur et nécessite un nombre de passes très inférieur, surtout pour des matrices de grande taille.

La figure 3.7 illustre comment la multiplication d'une matrice bande M à deux vecteurs non nuls avec un autre vecteur V se traduit en opérations sur leurs textures respectives. Pour une meilleure compréhension, les vraies matrices M et vecteurs V ainsi que leurs décompositions en textures sont présentées à la figure 3.8. M y est représentée par deux vecteurs diagonaux non nuls chacun décomposé en une texture, dont un vecteur diagonal et un hors-diagonale.

La différence avec le cas des matrices pleines est que l'on ne peut plus multiplier directement les valeurs des vecteurs de M avec celles de V car il faut maintenant

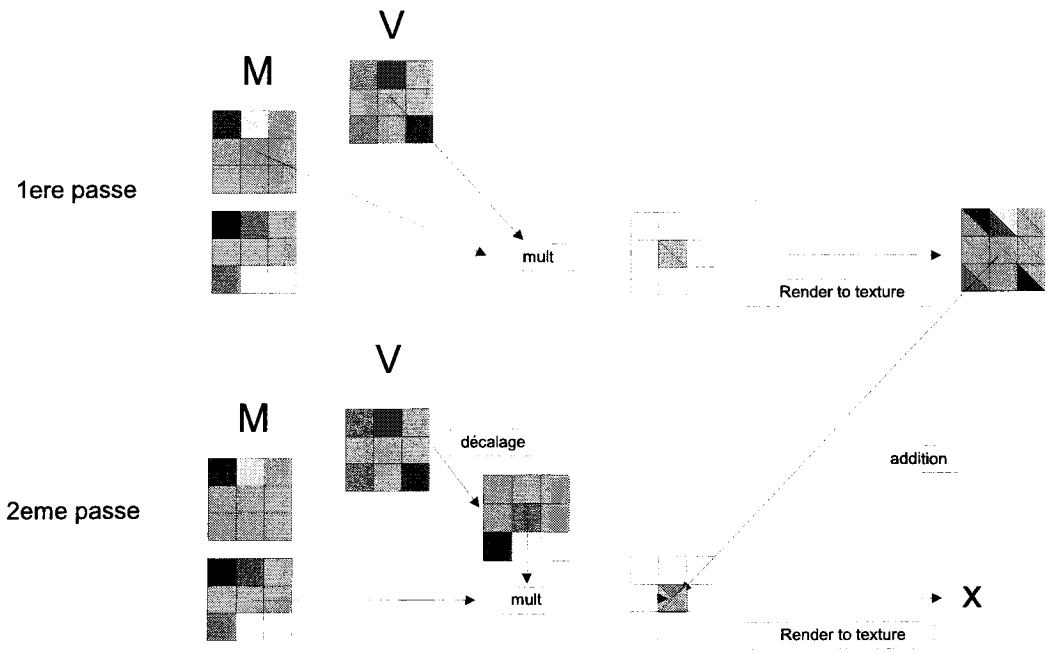

Figure 3.7 Multiplication d'une matrice bande et d'un vecteur

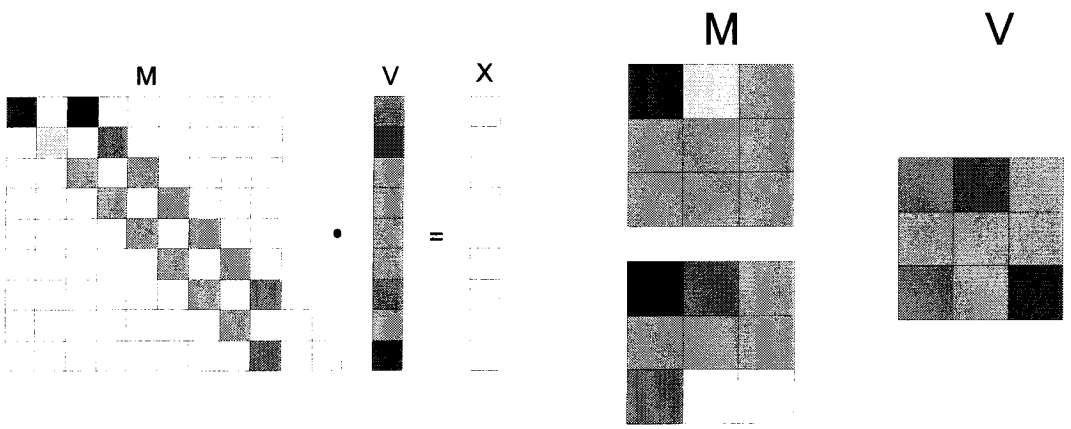

Figure 3.8 Décomposition d'une matrice bande et d'un vecteur en textures

introduire un décalage dans la lecture des valeurs de V en fonction de la position des vecteurs de M à multiplier. L'idée est de réduire au maximum le nombre de passes nécessaire, en groupant au sein du noyau plusieurs diagonales adjacentes. Dans notre cas, nous avons un maximum de 12 diagonales que nous noterons $[m_i]_{i=0..11}$.

Voyons comment notre noyau procède pour mener à bien ce calcul.

- En fonction des coordonnées de fenêtre $wpos(x, y)$ du pixel traité par le noyau et de la résolution N du vecteur, on en déduit l'index j_0 du quadruplet des m_i à multiplier avec V. Cela revient simplement à traduire une coordonnée fenêtre 2D ($wpos.x$,$wpos.y$) en une coordonnée 1D j_0, qui s'obtient simplement par la formule :

$$j_0 = (N - wpos.y) * N + wpos.x \tag{3.1}$$

- Grâce à j_0 et au décalage hors-diagonal de m_0 noté i, on peut en déduire l'index j_1 du quadruplet de V à multiplier, d'après la formule.

$$j_1 = (i + j)\%N \tag{3.2}$$

où % est l'opérateur modulo. Cependant comme cet opérateur n'est pas directement supporté par le GPU, on l'émule par l'opérateur :

$$a\%b \Leftrightarrow frac(a/b) * b \tag{3.3}$$

où $frac(r)$ est la partie fractionnelle de r.

- Une fois j_0 et j_1 déterminés, on en déduit les coordonnées de texture uv_0 et uv_1 correspondant à l'index j_0 et j_1 des quadruplets à multiplier. Cela revient à

traduire une adresse 1D en adresse 2D, par la formule :

$$uv_0 = \{j_0 \% N, N - floor(j_0/N)\} \tag{3.4}$$

où $floor(r)$ est la partie entière de r. On applique la même formule pour trouver uv_1.

- Enfin, étant donné que l'on multiplie 12 diagonales consécutives, il est nécessaire d'aller les multiplier par plusieurs quadruplets consécutifs de V. Si on utilisait pour chacun de ces quadruplets la conversion d'adresse 1D -> 2D, cela induirait inutilement une trop grande quantité d'opérations arithmétiques pour le noyau. C'est pourquoi on utilise une texture de décalages précalculés qui prend en entrée une adresse 2D de quadruplet et retourne l'adresse 2D du quadruplet suivant. On réduit ainsi significativement le nombre d'instructions nécessaires, car seulement 4 consultations successives de textures sont requises.

- Une fois que tous les quadruplets entrant en jeu sont obtenus, on peut multiplier toutes les valeurs pour obtenir le quadruplet résultat et l'écrire dans le pixel traité. Le détail de ces opérations est spécifié dans le code Cg figurant en annexe (II).

Contrairement au cas des matrices pleines, le nombre de passes nécessaire ici ne dépend pas de la taille de la matrice, mais du nombre de diagonales non nulles qu'elle contient. Des matrices contenant jusqu'à 12 diagonales non nulles pourront être multipliées par un vecteur en une seule passe quelle que soit sa dimension jusqu'à 1024*1024*4.

3.4 Gradients Conjugués

La méthode des gradients conjugués est une des méthodes itératives les plus utilisées pour la résolution de systèmes linéaires creux. Un système linéaire est défini par un ensemble de N équations linéaires à N inconnues. Une très bonne introduction à la méthode des gradients conjugués est proposée par Shewchuk (1994), mais nous allons en présenter ici rapidement le principe.

3.4.1 Description de l'algorithme

Considérons un système linéaire, représenté sous sa forme matricielle :

$$Ax = b \tag{3.5}$$

où A est la matrice de dimension $n \times n$ des coefficients du système, x est le vecteur des inconnues, et b est le vecteur connu du second membre. Le système s'écrit finalement :

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

A étant symétrique définie positive, elle vérifie, pour tout vecteur non nul x, la relation :

$$x^T Ax > 0$$

Condidérons maintenant la fonction suivante, appelée *fonction quadratique* d'un vecteur x

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{3.6}$$

où A est une matrice, b,b^T et x,x^T des vecteurs et leur transposé et enfin c un scalaire. Notons maintenant le gradient de cette fonction quadratique, défini par :

$$f'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \frac{\partial}{\partial x_2} f(x) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix}$$

En appliquant cette définition à l'équation 3.6, on obtient, après quelques opérations :

$$f'(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b$$

Si A est symétrique, $A = A^T$ et cette équation devient :

$$f'(x) = Ax - b$$

On constate alors que le vecteur x satisfaisant l'équation $f'(x) = 0$ est également solution de l'équation 3.5. Autrement dit, la solution du système linéaire 3.5 correspond à un extrêmum de la fonction quadratique f. On peut montrer qu'en fait, le x recherché minimise la fonction f. C'est là qu'intervient la méthode des gradients conjugués, qu'on notera MGC, qui permet de trouver, selon un algorithme itératif, le minimum de la fonction quadratique.

A chaque itération, la MGC génère une nouvelle approximation de la solution en établissant une direction de recherche, un déplacement suivant cette direction et le résidu associé à cette approximation. L'avantage de cette technique par rapport aux autres est, dans notre cas, qu'elle ne fait intervenir que des opérations algébriques simples, telles que nous les avons implémentées.

Voici le pseudo-code de l'algorithme des gradients conjugués non préconditionné :

GRADIENT CONJUGUÉ NON PRÉCONDITIONNÉ()

1 $d_{(0)} = r_0 = b - Ax_0$
2 $i = 0$
3 **while** $\|r_{(i)}\| \geq seuil$ **do**
4 $\quad \alpha_{(i)} = \frac{r_{(i)}^T r(i)}{d_{(i)}^T Ad(i)}$
5 $\quad x_{(i+1)} = x_{(i)} + \alpha_{(i+1)} d_{(i)}$
6 $\quad r_{(i+1)} = r_{(i)} - \alpha_{(i+1)} Ad_{(i)}$
7 $\quad \beta_{(i+1)} = \frac{r_{(i+1)}^T r(i+1)}{r_{(i)}^T r(i)}$
8 $\quad d_{(i+1)} = r_{(i+1)} + \beta_{(i+1)} d_{(i)}$
9 $\quad i = i + 1$
10 **endwhile**

où $d_{(i)}$ et $\alpha_{(i)}$ sont les directions de recherche et le déplacement suivant cette direction, $r_{(i)}$ sont les résidus, et $x_{(i)}$ les approximations de la solution recherchée. $\beta_{(i)}$ est un scalaire permettant d'améliorer la direction de recherche en fonction de

la direction de recherche précédante. L'algorithme se répète tant que l'écart à la solution est en dessus d'un seuil déterminé par l'utilisateur.

3.4.2 Implémentation

Voyons maintenant comment l'algorithme des gradients conjugués s'exprime en fonctions des opérateurs décrits dans la section 3.3. Afin d'économiser la mémoire vidéo, on choisit de jongler entre des paires de vecteurs plutôt que d'incrémenter les r_i, d_i, et x_i. L'argument $NULL$ dans les opérateurs signifie que cet argument est ignoré dans le nuanceur.

MGC SUR GPU ()

1 $gpuMatOp(A, b, NULL, temp)$
2 $gpuVecAdd(1.0, b, -1.0, temp, r_0)$
3 $gpuVecAdd(1.0, r_0, 0.0, NULL, d_0)$
4 $gpuVecAdd(1.0, b, 0.0, NULL, x_0)$
5 $i = 0$
6 **while** $\|r_{(i)}\| \geq seuil$ **do**
7 $\quad \rho = gpuVecScal(r_i, r_i)$
8 $\quad gpuMatOp(A, d_i, NULL, temp)$
9 $\quad \alpha = gpuVecScal(d_i, temp)$
10 $\quad \alpha = \rho/\alpha$
11 $\quad gpuVecAdd(1.0, x_i, \alpha, d_i, x_{i+1})$
12 $\quad gpuVecAdd(1.0, r_i, -\alpha, temp, r_{i+1})$
13 $\quad \beta = gpuVecScal(r_{i+1}, r_{i+1})$
14 $\quad \beta = \beta/\rho$
15 $\quad gpuVecAdd(1.0, r_{i+1}, \beta, d_i, d_{i+1})$
16 **endwhile**

Tous ces opérateurs sont encapsulés dans une classe à part entière, qui définit elle-même un opérateur de plus haut niveau :

$$gpuGC(A, b, res)$$
$$res \Leftarrow \text{solution du système } Ax = b$$

CHAPITRE 4

RÉSULTATS ET DISCUSSION

Ce chapitre présente une étude des performances des opérateurs sur GPU par rapport à leurs équivalents sur CPU, tout en analysant les points critiques et les limites associées à notre implémentation. Ensuite, on démontre la facilité d'utilisation de ces outils au sein de deux applications simulant des phénomènes physiques : le comportement d'un tissu et la propagation d'ondes 2D sur une surface liquide peu profonde. Ces deux applications ont été choisies car ce sont des simulations *autonomes* : une fois que l'initialisation du système sur le GPU est terminée la simulation peut s'effectuer en boucle sur le GPU sans nécessiter de transferts additionnels depuis le CPU. Dans les deux simulations, le système à résoudre est exactement le même à chaque pas de simulation.

4.1 Application 1 : Simulateur de tissu

Le premier exemple d'utilisation des opérateurs algébriques a été l'implémentation d'une simulation simple de tissu.

4.1.1 Théorie

Il existe une méthode utilisant une variante des gradients conjugués pour la simulation de tissu (Baraff et Witkin (1998)), néanmoins celle-ci fait intervenir un système linéaire dont les coefficients varient à chaque itération. Ceci impliquerait une mise à jour de la matrice donc un transfert de données continuel du CPU vers le

GPU. Ce transfert serait trop important pour le bus AGP, et nuirait sérieusement à l'efficacité de calcul sur GPU. C'est pourquoi une autre méthode plus simple a été choisie dans ce cas : la méthode d'intégration de Verlet (4.1).

La modélisation que l'on a choisi pour le tissu est celle de Jakobsen (2001), où le tissu est représenté par un système de particules de masses égales réparties sur une grille cartésienne. Chaque particule du réseau est reliée à ses 4 voisines par des ressorts de rigidité égale. La simulation consiste donc à mettre à jour la position de chacun des points en tenant compte de la force exercée par ces quatre ressorts, ainsi que d'un champ de force extérieur.

A chaque pas de la simulation, on peut mettre à jour la position x' ainsi que la vitesse v' des particules en fonction de leur valeur au pas de temps précédent, par l'intégration d'Euler :

$$x' = x + v * \delta t$$
$$v' = v + a * \delta t$$

où δt est l'intervalle de temps écoulé entre deux pas de simulation et a est l'accélération de la particule obtenue par le principe fondamental de la dynamique $f = m * a$.

C'est ici qu'intervient la méthode d'intégration de Verlet, pour laquelle seule l'information sur la position actuelle x et la position précédente x^* est nécessaire pour déterminer la nouvelle position. Notons que le facteur 2 dans l'équation peut être remplacé par une valeur inférieure, ce qui a pour effet d'introduire un amortissement du mouvement.

$$x' = 2 * x - x^* + a\delta * t^2 \tag{4.1}$$

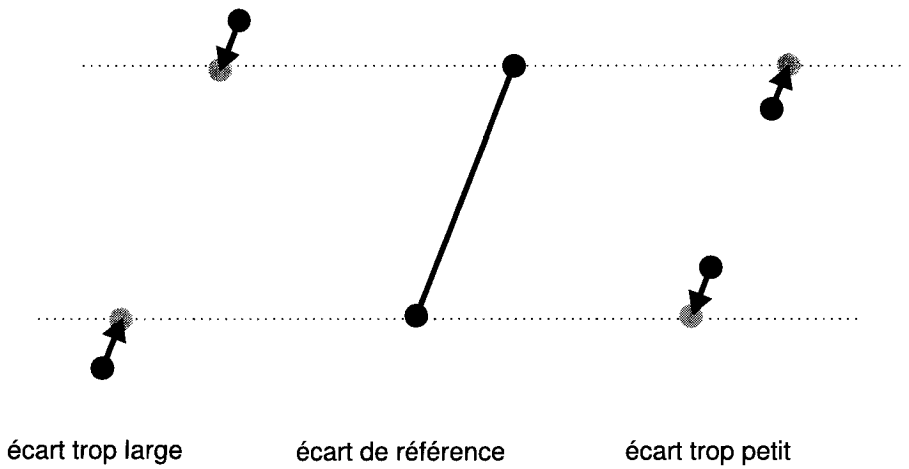

Figure 4.1 Contrainte de distance appliquée à un couple de points

En plus de ce schéma, on applique à chaque pas de simulation un schéma de relaxation basé sur certaines contraintes qui replacent les particules en fonction d'une règle simple, illustrée par la figure 4.1.

Sur un couple de points $[A, B]$, la contrainte s'effectue suivant la formule :

$$P_A = P_A + \frac{1}{2} * (\vec{AB}) * \frac{||\vec{AB}|| - l_{repos}}{||\vec{AB}||}$$
$$P_B = P_B - \frac{1}{2} * (\vec{AB}) * \frac{||\vec{AB}|| - l_{repos}}{||\vec{AB}||}$$

Cela revient à calculer l'écart entre la longueur du segment [AB] et la distance de référence l_{repos}, puis à déplacer les points AB de la moitié de cet écart tel que la distance qui les sépare soit à nouveau égale à la distance de référence.

Pour chaque particule, on applique cette contrainte avec ses 4 voisins. Ensuite, en la répétant un certain nombre de fois pour tout le système, on finit par converger vers une solution.

4.1.2 Implémentation

4.1.2.1 Vecteurs et matrices

Grâce au schéma de Verlet, les seules variables à stocker pour les particules sont les positions actuelles et précédentes. On initialise d'abord les positions initiales $p_{i,j}$ des particules en mémoire sous forme de tableau 1D avant de les transfèrer sur le GPU pour l'initialisation des calculs. Ce transfert se fait simplement par l'appel du constructeur de la classe vecteur décrite dans le chapitre précédent en lui passant un pointeur vers les données.

$$pos = new\ gpuVec(float*\ data,\ int\ size)$$

Ensuite, on définit 4 matrices $A_{0,1}$, $A_{0,-1}$, $A_{-1,0}$, $A_{1,0}$ qui, une fois multipliées par p, donnent les tableaux représentant pour chaque particule le vecteur le reliant avec son voisin de droite, de gauche, de haut et de bas (voir figure 4.2).

Les matrices $A_{i,j}$ sont des matrices creuses diagonales, avec seulement deux diagonales non nulles, $A_{1,0}$ a par exemple la forme :

$$A_{1,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ & & & & \vdots & & & \end{bmatrix}$$

Notons que ces matrices ne prennent délibérément pas en compte les cas particuliers des particules situées sur les bords du réseau. Nous verrons dans la section suivante pourquoi cela n'a pas été nécessaire.

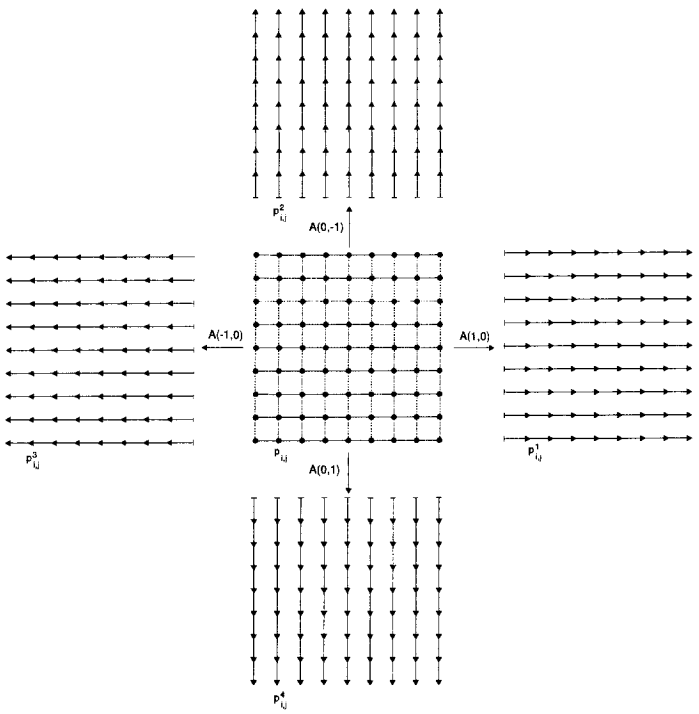

Figure 4.2 Matrices de décalages

Enfin, on initialise un vecteur a qui représente le champ d'accélération extérieur appliqué aux particules. On ne considère ici que l'effet de la gravité, ce qui fait de a un champ uniforme et constant.

Les initialisations des matrices sont réalisées par des fonctions d'aide *setDiag()* qui prennent en paramètre le décalage de la diagonale considérée et un pointeur vers le tableau de valeurs à donner à cette diagonale. Le vecteur associé à cette diagonale est alors automatiquement construit puis l'espace occupé en RAM par les valeurs passées à la fonction est libéré.

Une fois que les matrices $A_{i,j}$ et le vecteur p sont créés et envoyés sur le GPU, tout est en place pour commencer la simulation.

4.1.2.2 Simulation

Ces matrices et vecteurs sont ensuite utilisés par les opérateurs pour la simulation. Le pseudo-code figurant plus bas indique quelles sont les opérations effectuées sur le GPU pour la simulation : dans un premier temps, on mémorise les positions courantes dans *temp* (ligne 1) puis on applique le schéma de l'équation 4.1 (l. 2-3), en introduisant en plus un facteur variable d'atténuation *damp*. Ensuite, à partir des matrices $A_{(i,j)}$, on obtient les quatres champs de vecteurs p_i (l. 5-8). On passe enfin ces champs à un nouvel opérateur *gpuVerlet*, qui se charge d'appliquer les contraintes de distance (cf fig. 4.1) pour mettre à jour le champ des positions des particules (représenté par le vecteur *pos*). A partir de ce résultat intermédiaire, on réitère le processus n_{iter} fois pour obtenir le résultat final. Enfin on attribue à pos^* les valeurs des anciennes positions (l. 11), afin de préparer l'itération au pas de temps suivant.

VERLET SUR GPU ()

1 $gpuVecAdd(1.0, pos, 0.0, NULL, temp)$
2 $gpuVecAdd(1.0 + damp, pos, -damp, pos^*, pos)$
3 $gpuVecAdd(1.0, pos, timestep, a, pos)$
4 **for** $k \leftarrow 0 \ldots n_{iter}$ **do**
5 $\quad gpuMatOp(A(1,0), pos, NULL, pos_1)$
6 $\quad gpuMatOp(A(0,-1), pos, NULL, pos_2)$
7 $\quad gpuMatOp(A(-1,0), pos, NULL, pos_3)$
8 $\quad gpuMatOp(A(0,1), pos, NULL, pos_4)$
9 $\quad gpuVerlet(pos, pos_1, pos_2, pos_3, pos_4)$
10 **endfor**
11 $gpuVecAdd(1.0, temp, 0.0, NULL, pos^*)$

On voit ici la simplicité du code nécessaire à l'implémentation de la simulation, qui

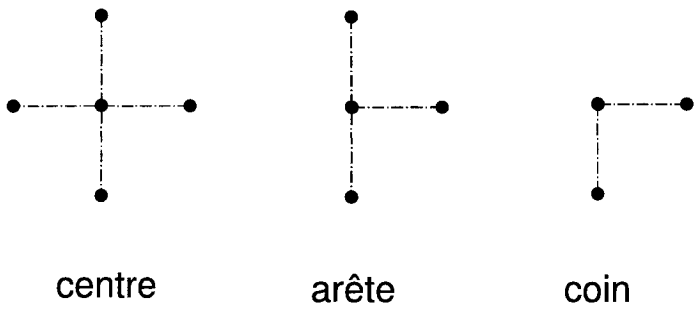

Figure 4.3 Différents cas pour l'application des contraintes

ne fait intervenir aucune connaissance en infographie, mais juste quelques appels de fonctions depuis un programme en C++.

Voyons maintenant comment fonctionne l'opérateur *gpuVerlet*. La figure 4.3 montre les différents cas que l'on peut rencontrer pour l'application des contraintes de distance. L'opérateur doit donc appliquer des opérations différentes selon la nature des points qu'il traite suivant qu'ils se trouvent au centre, sur une arête ou dans un coin du domaine. Afin d'optimiser le fonctionnement de cet opérateur, on a employé la technique évoquée dans la section 2.4, qui fait usage de sous-flux, chacun appliqué à un noyau différent. L'avantage de cet opérateur est qu'il gère automatiquement le partitionnement de la grille de simulation et laisse le choix des opérations à effectuer dans chacun des cas.

L'opérateur *gpuVerlet* met donc à jour la texture du vecteur *pos* après l'application des contraintes. Le dessin de cette texture est décomposé en trois types de régions, les coins (4 `GL_POINTS`) les arêtes (4 `GL_LINE`) et le centre (1 `GL_QUAD`), chacune étant associée à un nuanceur différent chargé d'appliquer les contraintes respectivement sur deux, trois, ou quatres couples de points.

Afin de rendre la simulation plus interactive et amusante, on impose au système des contraintes supplémentaires :

– Un domaine sphérique interdit : si un point se trouve à l'intérieur de ce domaine,

on déplace ce point à la limite de ce domaine le long du vecteur qui le relie au centre de la sphère.

- Une contrainte planaire : si un point passe en dessous d'une certaine hauteur, on le projette au niveau du sol.

- Contrôle d'un coin : l'utilisateur contrôle par la souris un des coins du tissu, qui échappe ainsi aux calculs de la simulation. La position à donner au coin est passée directement comme paramètre au nuanceur.

Le comportement du tissu dépend aussi d'un certain nombre de paramètres, notamment du facteur d'amortissement dans l'équation de 4.1, qui permet de simuler le frottement de l'air. On peut également contrôler la raideur des ressorts, ce qui s'avère être un facteur important dans la stabilité de la simulation. Enfin, le nombre d'itérations de la boucle des contraintes est paramétrable et utile pour garantir la stabilité de la simulation avec des raideurs élevées.

4.1.3 Résultats

4.1.3.1 Performances

Les résultats sont très variables en fonction de la taille de la grille de points. En effet pour afficher le tissu il est nécessaire de rapatrier la texture du résultat en mémoire principale puis renvoyer sur les GPU les points à afficher en 3D. C'est ce qui explique le délai supplémentaire important pour l'affichage qui apparaît dans le tableau 4.1.

La figure 4.4 montre un aperçu de la fenêtre de simulation interactive, pour un tissu de 64x64 points. L'utilisateur peut orbiter autour du tissu, contrôler un coin

Tableau 4.1 Performances de la simulation de tissu (3 relaxations)

Taille	Simulation (ms)	Simulation et affichage (ms)
16x16	7.9	10.7
32x32	8.8	17.9
64x64	11.0	45.5
128x128	18.1	166.3
256x256	69.0	846.0
512x512	336.0	2448.8

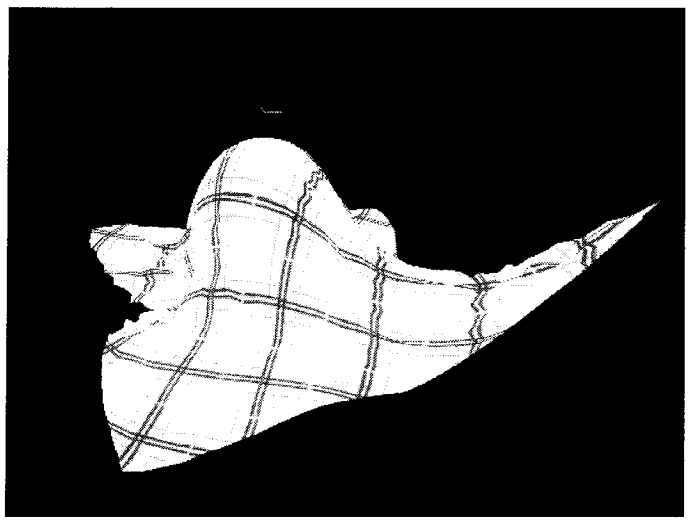

Figure 4.4 Capture d'écran de la simulation de tissu sur GPU

du tissu et voir l'effet en temps réél.

Le tableau 4.1 indique les performances obtenues pour différentes résolutions de grilles. On sépare bien les performances entre la simulation et l'affichage, car il s'avère que ce dernier est le facteur limitant pour des grilles de grande taille. Nous n'avons pas fait de comparaison avec le CPU car une telle simulation ne s'implémenterait pas du tout de la même manière : en effet les matrices de décalages sont un artifice que l'on utilise pour adapter le calcul de manière à ce qu'il puisse être traité par nos opérateurs. Une implémentation sur CPU serait plus simple et probablement plus rapide car on pourrait se passer de cet artifice.

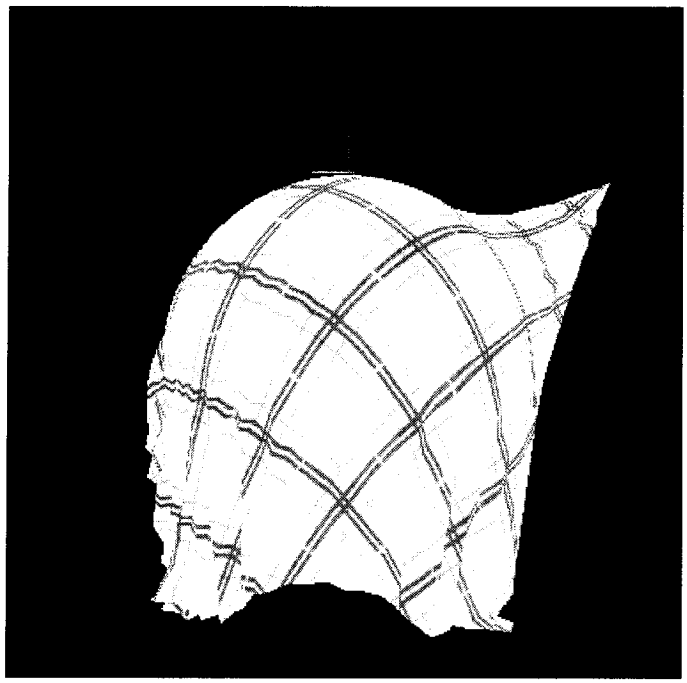

Figure 4.5 Autre capture

4.1.3.2 Limitations

L'analyse du code avec un outil de profilage (le projet Open Source *valgrind*), met en évidence un certain nombre de facteurs limitant l'exploitation du GPU, dont les principaux sont :

- **Activation/désactivation d'un contexte d'affichage OpenGL** : l'écriture dans un tampon de pixels hors-écran (*Offscreen PBuffer*) nécessite la création et l'activation par le serveur X d'un contexte d'affichage OpenGL. Cette opération doit être effectuée à chaque appel d'un opérateur algébrique avant l'exécution du noyau, qui coûte en moyenne 10% du temps passé, soit 5% pour l'activation du contexte du rendu hors-écran et 5% pour la restitution du contexte de rendu à l'écran.

- **Copie en texture** : étant donnée l'absence sous Linux d'une extension OpenGL

permettant le rendu direct en texture, on est contraint de copier le contenu du tampon d'image dans la texture par la commande `glCopyTexSubImage2D`. Ceci représente en moyenne 10% du temps de la simulation.

- **Passage des paramètres** : la grande quantité et surtout la fréquence du changement de la valeur des paramètres passés aux différents noyaux implique une certaine charge sur le CPU. Celle ci est d'autant plus grande que le taux de rafraîchissement est grand, pouvant nécessiter jusqu'à 10% du temps total.

- **Affichage en 3D** : tableau 4.1 montre que le rapatriement des résultats de la simulation pour l'affichage en 3D induit un transfert du GPU vers le CPU (mise en RAM de la texture résultat), puis du CPU vers le GPU (envoi des sommets à dessiner) qui est responsable d'un ralentissement d'autant plus gros que le nombre de points dans la grille (donc de données à faire circuler) est élevé.

Pour ces raisons, dans le cas d'une grille de 64x64 points, seulement 27% du temps est réellement passé sur le GPU.

4.2 Application 2 : Simulation d'ondes

Afin de démontrer l'efficacité du résoluteur par gradients conjugués, un simulateur d'ondes 2D a aussi été implémenté. Après une rapide présentation de la théorie, son implémentation sur GPU via les opérateurs algébriques sera présentée. Nous montrerons que les objectifs d'efficacité et de simplicité sont bien atteints.

4.2.1 Théorie

L'équation d'onde 2D décrit l'évolution d'une perturbation Δh se propageant à la vitesse c le long d'une surface planaire peu profonde. Cette équation s'écrit :

$$c^2 \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right) = \frac{\partial^2 h}{\partial t^2} \tag{4.2}$$

On se propose de résoudre numériquement cette équation sur une grille régulière de points en utilisant la méthode des différences finies (cf Kass et G. (1990)). Si on note $h_{i,j}$ le point de la grille se situant sur la i^e ligne et la j^e colonne et que l'on considère un espacement égal Δx des points dans les deux directions, on peut, en utilisant les différences centrées et le shéma de Crank-Nickolson, mettre l'équation 4.2 sous la forme discrétisée suivante :

$$\frac{h_{i,j}^{t+1} - 2h_{i,j}^{t} + h_{i,j}^{t-1}}{\Delta t^2} = \frac{1}{2} c^2 \left(\frac{h_{i+1,j}^{t} - 2h_{i,j}^{t} + h_{i-1,j}^{t}}{\Delta x^2} + \frac{h_{i,j+1}^{t} - 2h_{i,j}^{t} + h_{i,j-1}^{t}}{\Delta x^2} \right)$$

$$+ \frac{1}{2} c^2 \left(\frac{h_{i+1,j}^{t+1} - 2h_{i,j}^{t+1} + h_{i-1,j}^{t+1}}{\Delta x^2} + \frac{h_{i,j+1}^{t+1} - 2h_{i,j}^{t+1} + h_{i,j-1}^{t+1}}{\Delta x^2} \right)$$

En réagençant les termes selon les indices de temps, on obtient :

$$(4\alpha + 1) h_{i,j}^{t+1} - \alpha \left(h_{i+1,j}^{t+1} + h_{i-1,j}^{t+1} + h_{i,j+1}^{t+1} + h_{i,j-1}^{t+1} \right)$$

$$= \alpha \left(h_{i+1,j}^{t} + h_{i-1,j}^{t} + h_{i,j+1}^{t} + h_{i,j-1}^{t} \right) + (1 - 4\alpha) h_{i,j}^{t} - h_{i,j}^{t-1}$$

avec $\alpha = \frac{c^2 \Delta t^2}{2\Delta x^2}$. On peut maintenant écrire cette équation sous sa forme matricielle $Ax = b$, où x est le vecteur des $h_{i,j}^{t+1}$ à déterminer en fonction du second membre b connu dépendant des hauteurs aux temps t et $t-1$. La matrice A est symétrique définie positive, on peut donc appliquer la méthode des gradients conjugués pour résoudre ce système.

4.2.2 Implémentation

4.2.2.1 Vecteurs et Matrices

Il s'agit ici de construire, à l'aide des structures implémentées, la matrice A et le vecteur du second membre b, pour mettre en place le système à résoudre et le passer au résoluteur.

Afin de minimiser les transferts entre le CPU et le GPU, on se place dans le cas où le pas de temps Δt est constant, ce qui implique que la matrice A est également constante. Ceci nous permet de transférer la matrice une fois pour toute sur le GPU au tout début de la simulation et garantir ainsi le critère de localité mis en évidence lors de la revue bibliographique (sous-section 2.5.4). La définition de A est similaire à celle des matrices de décalages utilisées pour le simulateur de tissu (figure 4.2) et qui permettaient d'accéder aux valeurs de points voisins sur la grille. Si on reprend les même notations pour les matrices de décalages, on peut écrire A sous la forme :

$$A = (8\alpha + 1)I - \alpha(A_{0,1} + A_{0,-1} + A_{-1,0} + A_{1,0})$$

où I est la matrice identité. La matrice A contient aussi les informations sur les conditions limites, qui imposent une hauteur constante nulle à l'extérieur du do-

maine. Ceci donnera naissance à des réflexions lorsque le front d'onde atteindra un des bords de la grille.

Voyons maintenant quels vecteurs doivent être stockés. Comme ils sont stockés sur le GPU comme des textures 2D (cf 3.2.1), cela en fait une structure déjà adaptée pour contenir la grille des hauteurs. C'est pourquoi on considèrera ici les vecteurs sous leur représentation 2D, indexée par le couple d'indices (i, j). Comme chaque pas de simulation a besoin des valeurs du champ des hauteurs aux deux itérations précédentes, on construit trois vecteurs de hauteurs notés h_{t-1}, h_t, et h_{t+1}.

La construction du second membre b dépend des vecteurs h_{t+1} et h_t et s'écrit :

$$b_{i,j} = \alpha \left(h^t_{i+1,j} + h^t_{i-1,j} + h^t_{i,j+1} + h^t_{i,j-1} \right) + (1 - 4\alpha) h^t_{i,j} - h^{t-1}_{i,j}$$

On peut aussi l'écrire sous la forme :

$$b = M h_t - h_{t-1}$$

où l'expression de la matrice M est similaire à celle de A et s'exprime par :

$$M = 2I + \alpha(A_{0,1} + A_{0,-1} + A_{-1,0} + A_{1,0})$$

M est elle aussi une matrice à coefficients constants, initialisée et transférée au début de la simulation. La détermination du second membre b du système à résoudre ne nécessite alors plus qu'une multiplication matrice vecteur suivie d'une addition de deux vecteurs.

Enfin, on initialise les vecteurs h_t et h_{t-1} à leur création sur le GPU par une grille de valeurs uniformes nulles, excepté quelques points où l'on introduit une perturbation.

4.2.2.2 Simulation

Une fois les matrices A et M créées et envoyées dans la mémoire vidéo, on utilise les opérateurs pour le calcul du nouveau champ de hauteurs. L'algorithme se décompose de la manière suivante : on fait d'abord une sauvegarde des hauteurs courantes (ligne 1), puis on construit le second membre b du système matriciel à l'aide de la matrice M et des vecteurs h_t,h_{t-1} (1.2-3), avant de résoudre le système ainsi construit par l'opérateur de gradients conjugués (1.4). Finalement, on transfère les hauteurs courantes h_t dans les anciennes hauteurs h_{t-1} (1.5). On peut alors récupérer les nouvelles hauteurs dans la texture associée au vecteur h_t.

SIMULATION D'ONDES 2D ()

1 $gpuVecAdd(1.0, h_t, 0.0, NULL, temp)$;
2 $gpuMatOp(M, h_t, NULL, b)$;
3 $gpuVecAdd(1.0, b, -1.0, h_{t-1}, temp)$;
4 $gpuCGradient(A, b, h_{t+1})$;
5 $gpuVecAdd(1.0, temp, 0.0, NULL, h_{t-1})$;

On voit ici la simplicité de l'implémentation proposée qui ne contient que quelques lignes de code et ne fait intervenir que des connaissances en programmation de base. Enfin, l'initialisation des matrices et des vecteurs ne fait intervenir que de simples boucles sur les tableaux de valeurs à transmettre sur le GPU. On répond donc précisément aux objectifs que nous nous sommes fixés de simplicité d'utilisation et d'intégration. Une implémentation équivalente a été effectuée par Cai *et al.* (1998), cependant ces derniers utilisent un préconditionneur et un nombre variable

d'itérations pour garantir une précision donnée.

Si l'utilisateur désire visualiser à l'écran les résultats de la simulation, il n'a qu'à remplir la fonction d'affichage OpenGL avec un opérateur de visualisation développé pour l'affichage, et dont nous allons détailler le fonctionnement.

Afin de pouvoir visualiser le résultat, on transforme le champ de hauteurs en un champ de normales à l'aide d'un nuanceur simple. En effet si on gardait le champ de hauteurs tel quel, les valeurs seraient tronquées dans le domaine [0,1] avant d'être affichées et cela produirait des taches complètement blanches pour les hauteurs supérieures à 1, ou complètement noires pour les hauteurs négatives.

La conversion du champ de hauteur en champ de normales est très simple et se fait de la manière suivante : pour chaque pixel, on consulte la hauteur des pixels immédiatement au dessus et à droite pour en déduire le vecteur normal qui est le produit vectoriel des deux vecteurs différences $\vec{v_1}$ et $\vec{v_2}$. Ces vecteurs s'écrivent :

$$\vec{v_1} = \begin{pmatrix} 1 \\ 0 \\ h_{i,j+1} - h_{i,j} \end{pmatrix} \text{ et } \vec{v_2} = \begin{pmatrix} 0 \\ 1 \\ h_{i+1,j} - h_{i,j} \end{pmatrix}$$

Finalement, la normale $\vec{n}$ s'obtient en chaque pixel par la formule :

$$\vec{n} = \frac{\vec{v_1} \times \vec{v_2}}{\|\vec{v_1} \times \vec{v_2}\|} = \frac{1}{\sqrt{(h_{i,j+1} - h_{i,j})^2 + (h_{i+1,j} - h_{i,j})^2 + 1}} \begin{pmatrix} h_{i,j} - h_{i,j+1} \\ h_{i,j} - h_{i+1,j} \\ 1 \end{pmatrix}$$

Cette normale est ensuite utilisée pour simuler la réfraction du liquide et calculer la direction suivant laquelle décaler les coordonnées de texture du pixel. On rajoute aussi un terme d'atténuation pour simuler un éclairage par l'intermédiaire d'un

produit scalaire entre la normale et l'orientation de la lumière directionnelle. Le détail du nuanceur figure en annexe II. Afin de faciliter la réutilisation de cette visualisation, un nouvel opérateur *gpuHeightField* a été implémenté, capable de rendre à l'écran n'importe quel vecteur en tant que champ de hauteurs. Une seule ligne est donc nécessaire pour l'affichage à l'écran, en appelant :

$$gpuHF(vec\ H)$$

$\Rightarrow$ affichage du vecteur H en tant que champ de hauteur

4.2.3 Résultats

La figure 4.6 montre une capture d'écran du simulateur d'ondes à différents instants, où l'on simule également la réfraction et l'éclairage de la surface liquide.

Le tableau 4.2 indique les temps obtenus sur le GPU et sur le CPU pour différentes tailles de grilles. Il faut noter que pour une grille de points de 256*256, la résolution des textures utilisées est de 128*128 car chaque pixel contient les valeurs de 4 points successifs de la grille, ce qui permet de tirer profit de la vectorisation des calculs dans les nuanceurs. Pour la simulation, on obtient un gain moyen satisfaisant par rapport au CPU.

On note que la simulation seule obtient des temps permettant l'interactivité. Cependant, l'affichage induit malheureusement plusieurs calculs qui limitent cette interactivité pour les grandes grilles. En effet, afin d'afficher, on doit d'abord décompresser la texture du vecteur résultat (opérations qui demande une vingtaine d'instructions), puis calculer le champ de normales correspondant et enfin appliquer les effets de lumières et de réfraction.

En termes de simulation, l'avantage principal de cette méthode de résolution utilisant les gradients conjugués est sa stabilité, même pour de grands pas de temps

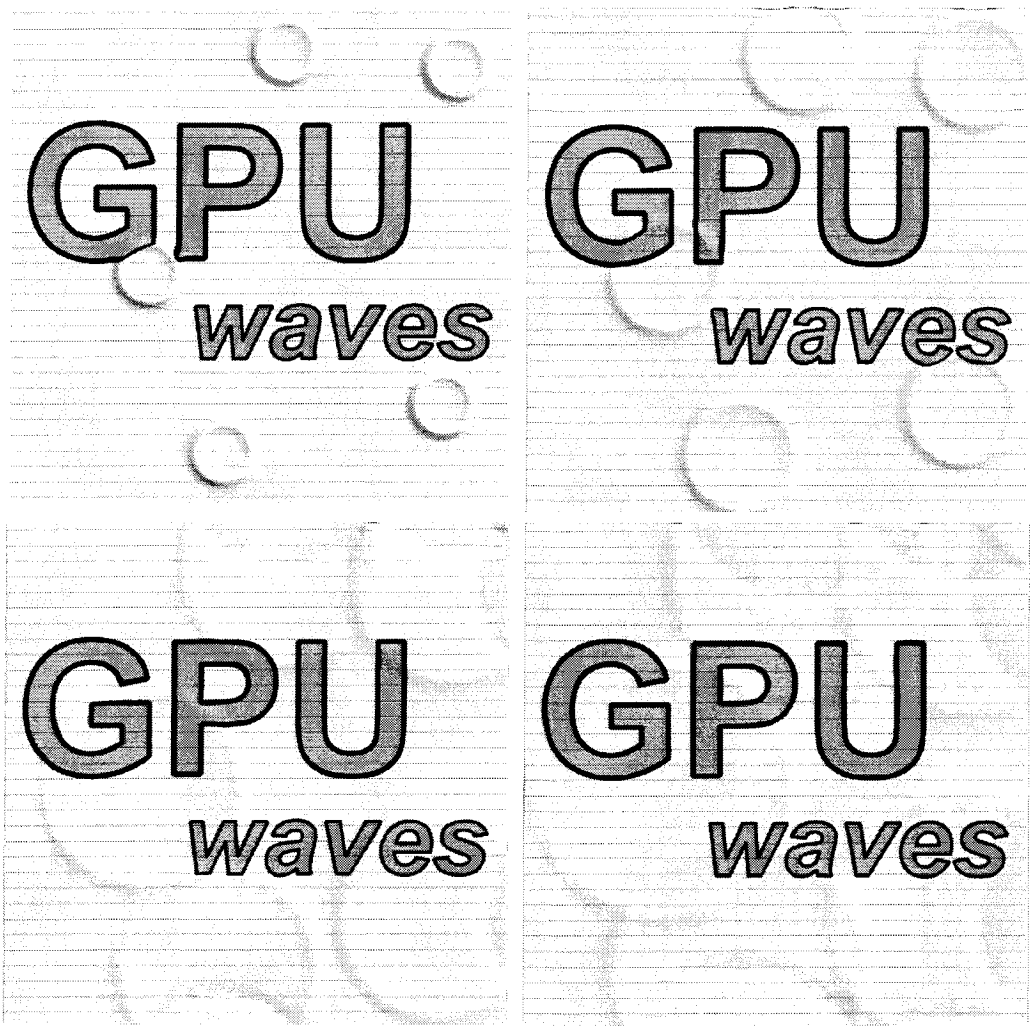

Figure 4.6 Propagation et réflection de plusieurs ondes 2D

Tableau 4.2 Performances de la simulation d'ondes

Taille	Simu. GPU (ms)	Simu. + aff. GPU (ms)	Simu. CPU (ms)	Gain
64x64	4.2	4.9	3.98	0.95
128x128	4.35	8.7	8.93	2.05
256x256	9.34	23.7	36.5	3.91
512x512	31.4	113.9	110.9	3.53

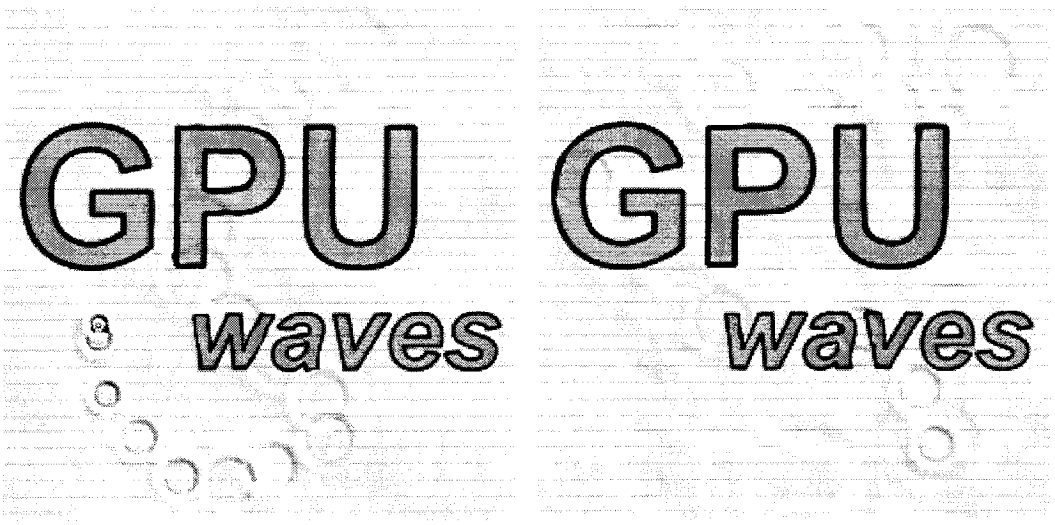

Figure 4.7 Interaction avec la simulation d'ondes

ou une vitesse élevée de propagation de l'onde.

Le listage du code de la simulation figure en annexe III. Afin d'alléger le listage, le contenu des fonctions *initialisermatrice* n'a pas été détaillé : il est codé par l'intermédiaire de nombreuses boucles que nous avons choisi d'omettre. Il serait également possible de charger ces matrices depuis un fichier texte.

4.3 Résultats : Performance des opérateurs

4.3.1 Produit matrice / vecteur

Voyons maintenant quel est le gain apporté par le GPU sur le CPU pour les opérateurs algébriques implémentés.

Le tableau 4.3 indique les temps de calculs moyens nécessaires pour la multiplication d'une matrice bande avec un vecteur (moyenne sur 1000 itérations). Dans ce cas, on considère une matrice bande à 12 diagonales non nulles que l'opérateur sur GPU

Tableau 4.3 Performances pour la multiplication Matrice creuse / vecteur (GeForce 6600 / Athlon XP 2800)

Taille	GPU (ms)	Bande Pass. (Go/s)	GFLOPS	CPU (ms)	fact. de gain
64x64x4	0.33	16.7	8.5	1.26	3.8
128x128x4	0.82	26.9	13.7	5.08	6.2
256x256x4	3.0	29.3	15.0	20.1	6.7
512x512x4	14.5	24.3	12.4	80.3	5.54
1024x1024x4	71.0	19.9	10.1	322.9	4.55

est capable de calculer en une passe. Les temps moyens du CPU ont quant à eux été mesurés d'après les performances de la librairie BLAS, et de l'opérateur *sgbmv* optimisé pour les matrices creuses par bande.

Dans l'ensemble on constate un facteur de gain appréciable (c'est à dire le rapport des deux temps moyens), allant de 4 à 7 fois plus vite sur le GPU. Les deuxième et troisième colonnes donnent une idée de la quantité de données et d'opérations traitées par seconde par le GPU : ces valeurs ont été déduites du nombre d'accès mémoire (21) et d'opérations arithmétiques (43) effectuées par le noyau. On constate qu'elles sont très élevées et bien supérieures à ce que peuvent fournir un CPU et de la mémoire vive classique.

4.3.2 Combinaison linéaire de vecteurs

Le tableau 4.4 compare les performances de l'opérateur *gpuVecAdd* par rapport à la fonction BLAS *saxpby* pour différentes tailles de vecteurs. D'une manière surprenante, le gain de performance est inférieur à celui obtenu précédemment, et peut même être fortement inférieur à 1 pour les petites tailles. L'analyse du code avec l'outil de profilage *Valgrind* a permis de mettre en évidence plusieurs facteurs limitants :

Tableau 4.4 Performances pour la Combinaison linéaire de deux vecteurs (GeForce 6600 / Athlon XP 2800)

Taille	GPU (ms)	Bande Pass. (Go/s)	GFLOPS	CPU (ms)	fact. de gain
64x64x4	0.25	2.1	0.52	0.02	0.08
128x128x4	0.26	8.0	2.0	0.15	0.58
256x256x4	0.7	11.9	3.0	2.29	3.27
512x512x4	2.9	11.5	2.9	9.2	3.17
1024x1024x4	15.1	8.8	2.2	36.8	2.44

- Mauvaise intensité arithmétique : en effet, le nuanceur contient autant d'accès mémoire que d'opérations arithmétiques (à savoir 2), il est probable que ce soit la latence de la mémoire qui limite la performance du noyau.

- Coût du changement de contexte de rendu OpenGL : à chaque exécution de l'opérateur, le changement très fréquent de contexte de rendu OpenGL entraîne un coût significatif par cycle, allant jusqu'à 55% du temps total pour les vecteurs de plus petites tailles. C'est ce qui explique le palier de 0.25 ms.

- Coût de la copie en texture : plus les vecteurs traités sont gros, plus ce coût augmente, s'échelonnant de 4% pour les petites tailles, jusqu'à 15% pour les vecteurs de 1024x1024x4 éléments.

4.3.3 Produit scalaire de vecteurs

C'est l'opérateur le moins performant par rapport au CPU : dans le meilleur des cas, le GPU ne calcule que moitié moins vite. Ceci s'explique essentiellement par le fait que par rapport aux autres opérateurs, un grand nombre de passes est nécessaire pour mener le calcul à terme, soit n passes pour des vecteurs de taille 2^n x 2^n. Chaque passe induit une invocation du noyau ainsi que le réglage de leurs

Tableau 4.5 Performances pour la norme d'un vecteur (GeForce 6600 / Athlon XP 2800)

Taille	GPU (ms)	CPU (ms)	fact. de gain
64x64x4	0.58	0.05	0.05
128x128x4	1.01	0.13	0.13
256x256x4	2.68	1.22	0.46
512x512x4	10.81	4.82	0.45
1024x1024x4	43.83	19.22	0.44

Tableau 4.6 Performances pour l'algorithme des gradients conjugés (GeForce 6600 / Athlon XP 2800)

Taille	GPU (ms)	CPU (ms)	fact. de gain
64x64x4	4.9	3.45	0.7
128x128x4	6.5	15.1	2.32
256x256x4	19.6	59.3	3.03
512x512x4	90.2	236	2.62

paramètres, qui représente jusqu'à 20% du temps passé au calcul. La copie en texture, appelée à chaque passe, accapare à elle seule jusqu'à 20% du temps.

4.3.4 Gradients conjugués

Le tableau 4.6 résume les résultats obtenus pour une itération de l'algorithme des gradients conjugués non préconditionné. Une version sur CPU de l'algorithme a été implémentée à partir des opérateurs BLAS appropriés pour comparer les performances avec l'opérateur *gpuCGradient*.

Le gain sur le CPU varie de l'ordre de 1 à 3, en fonction de la taille des vecteurs considérés. On voit que pour les petites tailles, la version sur GPU est moins efficace que son équivalent sur CPU, surtout en raison de la faiblesse des opérateurs d'addition et de produit scalaires. Quant aux grandes tailles, c'est l'opérateur norme qui est responsable de la baisse de gain. Cependant, on a en moyenne un gain de

temps de calcul confortable sur le CPU.

4.4 Discussion

4.4.1 Limites

Les limites dans l'utilisation des opérateurs proposés dans ce mémoire reposent en fait sur les limites matérielles du GPU.

En effet, dans le cas de l'opérateur de résolution par la méthode des gradients conjugués, le fonctionnement n'est vraiment optimal que lorsque la matrice du système à résoudre est identique d'un pas de temps à l'autre. En effet, si cette dernière devait être changée à chaque itération, cela impliquerait *pour chaque itération* le transfert depuis le CPU vers le GPU des nouveaux coefficients de la matrice. Comme la matrice est un ensemble de vecteurs, il serait alors nécessaire d'écraser le contenu de plusieurs de ses vecteurs, même pour le changement de quelques coefficients. Dans ce cas, la bande passante du port *AGP* montre vite ses limites et la simulation est beaucoup moins rapide.

La deuxième limite est illustrée dans la simulation de tissu, où l'affichage 3D des résultats pendant la simulation ralentit beaucoup le processus. Initialement il était prévu d'utiliser la capacité des nuanceurs de sommets à pouvoir lire la texture du résultat et l'afficher (GeForce 6600). Cependant, après de multiples essais et après avoir consulté la communauté GPGPU, il semble que personne n'ait encore utilisé cette fonctionnalité sous Linux et OpenGL. Il est probable que les pilotes, qui supportent que depuis très peu le modèle de nuanceur 3.0, ne soient pas encore très au point.

L'autre limitation de taille est l'absence très regrettable sous Linux du support du

rendu en texture, alors qu'il est présent sous Windows et DirectX. Ceci a de gros impacts sur les performances et constitue le point faible d'OpenGL par rapport à DirectX pour le calcul générique sur GPU, c'est pourquoi DirectX est souvent l'alternative la plus souvent choisie. Dans notre cas, ceci peut entraîner une perte de performances de 20% (tissu) à 40% selon les opérateurs. Cependant, l'introduction de l'extension `GL_EXT_FRAME_BUFFER_OBJECT` comble cette lacune, mais celle-ci n'est malheureusement implémentée que sous Windows par le pilote expérimental 76.10 beta.

4.4.2 Perspectives

Pour l'instant, lorsqu'un vecteur est construit et envoyé sur le GPU, il n'est possible de changer ses valeurs que par l'intermédiaire des opérateurs algébriques. Ainsi, si l'utilisateur désire changer seulement une ou deux valeurs dans ce vecteur, il ne peut le faire qu'en écrasant toutes les valeurs de ce vecteur en lui passant le tableau mis à jour. Il serait possible de proposer un opérateur [ ] qui par l'intermédiaire de la commande `gl_TexSubImage2D` écrirait à l'endroit désiré dans la texture du vecteur. Cependant, ceci nécessiterait tout de même de rapatrier auparavant la texture du vecteur afin de connaître les valeurs des canaux de couleur entourant le canal à mettre à jour et de les réutiliser pour l'écriture du texel.

Avec le système d'opérateurs que l'on a mis en place, il est assez facile d'étendre leur famille. Prenons par exemple le cas de l'opérateur Laplacien, défini par :

$$\nabla^2 T(x,y) = c^2 \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \tag{4.3}$$

La forme discrétisée correspondante est (Reggio (2004)) :

$$\nabla^2 T_{i,j} = T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1} - 4T_{i,j}$$

A l'aide des matrices $A_{i,j}$ définies dans la section 4.1.2.1, l'équation matricielle correspondante est :

$$\nabla^2 T = -4 * (A_{1,0} + A_{1,0} + A_{1,0} + A_{1,0}) * T = M * T$$

Si on initialise la matrice M, le Laplacien s'exprime comme un simple produit matrice vecteur. Si on note $\nabla^2 T$ le Laplacien de la grille T, il s'obtient simplement grâce à l'opérateur de multiplication matrice vecteur que l'on peut définir à son tour comme un nouvel opérateur :

$$gpuLaplacien(T, \nabla^2 T) \Longleftrightarrow gpuMatOp(M, T, NULL, \nabla^2 T)$$

Avec ce nouvel opérateur, on peut encore définir un nouvel opérateur de diffusion qui s'exprimerait par :

$$T^{diff} = T + \frac{C_d}{4} \nabla^2 T$$

Si on le traduit à l'aide de nos opérateurs on a :

$$gpuDiffusion(T, T^{diff}) \Longleftrightarrow \begin{array}{c} gpuLaplacien(T, \nabla^2 T) \\ gpuVecAdd(1, T, \frac{C_d}{4}, \nabla^2 T, T^{diff}) \end{array}$$

Grâce à ce nouvel opérateur, il serait possible de simuler d'autres phénomènes physiques faisant intervenir la diffusion.

Ces deux opérateurs montrent ainsi qu'il est aisé d'utiliser les opérateurs de base pour en construire de nouveaux. En fait tous les opérateurs se résumant à des opérations sur des grilles de points s'adaptent très bien en suivant la même démarche, en passant par les matrices de décalage $A_{i,j}$ qui forment en général la base de l'opérateur. On peut également envisager de nouveaux opérateurs dans le domaine du traitement d'images, comme la détection d'arêtes par la méthode de Canny. Les possibilités sont vastes !

CONCLUSION

Ce travail de maîtrise nous a permis de confirmer notre hypothèse qu'il était possible, à partir de l'état des connaissances du calcul sur GPU, de développer un outil simple ne nécessitant que des connaissances de base en programmation C++. Nous avons aussi démontré que les opérateurs proposés pouvaient être facilement réutilisés pour élaborer des fonctions plus complexes et s'appliquer à une grande variété de problèmes.

Plusieurs améliorations sont cependant envisageables :

- Une meilleure granularité du contrôle sur le contenu des matrices et des vecteurs afin de réduire les transferts en cas de modification partielle.

- Porter le code pour la plate-forme Windows + DirectX, beaucoup plus répandue que Linux. Cela permettrait de mieux diffuser notre librarie de fonctions.

- Optimiser les performances en utilisant les dernières extensions OpenGL lorsqu'elles seront officiellement supportées.

- Expérimenter un système hybride, où les tâches inadaptée au fonctionnement du GPU seraient réalisées par le CPU.

Dans la mesure où le calcul sur GPU devient de plus en plus populaire, on peut également espérer que les pilotes permettront dans l'avenir un contrôle de la mémoire cache afin d'atteindre des performances encore supérieures. L'industrie elle-même commence à prendre conscience de l'énorme potentiel des GPU et on voit apparaître de plus en plus de logiciels reposant sur leur utilisation : par exemple pour le traitement de son (BionicFX), d'images ou de vidéos (moteur de rendu *Gelato* de

Nvidia ou *CoreImage* de Apple computers. Pixar...), de visualisation et traitement de données (nVidia, ATI research lab ...).

L'avenir des GPU est très prometteur. Leur modèle de calcul parallèle en fait un concurrent de plus en plus sérieux pour les CPU qui reposent uniquement sur la fréquence de fonctionnement et semblent aujourd'hui avoir atteint une sorte de palier. L'avenir nous dira qui sortira vainqueur...

RÉFÉRENCES

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEK, J., DONGARRA, J., DU CROZ, J., GREENBAIM, A., HAMMARLING, S., MCKENNEY, A. ET SORENSEN, D. (1999). *Lapack User's guide.* Society for industrial and applied mathematics.

BARAFF, D. ET WITKIN, A. (1998). Large steps in cloth simulation. *Proceedings of the 25th annual conference on Computer graphics and interactive techniques.* 43–54.

BOLZ, J., FARMER, I., GRINSPUN, E. ET SCHRODER, P. (2003). Sparse matrix solvers on the gpu : Conjugate gradients and multigrid. *Proceedings of ACM SIGGRAPH 2003.* vol. 22, 917–924.

BUCK, I. (2005). *GPU gems2*, Addison wesley, chapitre Taking the Plunge into GPU Computing. 509–519.

BUCK, I., FOLEY, T., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M. ET HANRAHAN, P. (2004). Brook for gpus : Stream computing on graphics hardware. *Proceedings of ACM SIGGRAPH 2004.* vol. 23, 777–786.

CAI, X., LANGTANGEN, H. P., NIELSEN, B. F. ET TVEITO, A. (1998). A finite element method for fully nonlinear water waves. *J. Comput. Physics*, 143, 544–568.

CHAN, E., NG, R., SEN, P., PROUDFOOT, K. ET HANRAHAN, P. (2002). Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. *HWWS '02 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* 69–78.

COHEN, M. F., CHEN, S. E., WALLACE, J. R. ET GREENBERG, D. P. (1988). A progressive refinement approach to fast radiosity image generation. *Proceedings of ACM SIGGRAPH 1988.* ACM Press, New York, NY, USA, 75–84.

COOMBE, G. ET HARRIS, M. (2005). *GPU gems2*, Addison wesley, chapitre Global illumination using progressive refinement radiosity. 635–647.

DALLY, W., HANRAHON, J., EREZ, P., KNIGHT, M., LABONT, F., AHN, J., JAYASENA, N., KAPASI, U., DAS, A., GUMMARAJU, J. ET BUCK, I. (2003). Merrimac : Supercomputing with streams. *Proceedings of the 2003 ACM/IEEE conference on Supercomputing.* IEEE Computer Society, 35.

DALLY, W., KAPASI, U., KHAILANY, B., AHN, J. ET DAS (2004). Stream processors : Programmability with efficiency. *ACM Queue*, 2, 52–62.

FATAHALIAN, K., SUGERMAN, J. ET HANRAHAN, P. (2004). Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. *HWWS '04 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* 133–137.

FOLEY, T., HOUSTON, M. ET HANRAHAN, P. (2004). Efficient partitioning of fragment shaders for multiple-output hardware. *HWWS '04 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* 45–53.

GOODNIGHT, N., WOOLEY, C., LEWIN, G., LUEBKE, D. ET HUMPHREYS, G. (2003). A multigrid solver for boudary value problems using programmable graphics hardware. *HWWS '03 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* 102–111.

GPGPU, V. (2004). Gpgpu : General-purpose computing on graphics processors. *http ://www.gpgpu.org/vis2004.* Visité le 16 mai 2005.

HALL, J., CARR, N. ET HART, J. (2003). Cache and bandwidth aware matrix multiplication on the gpu. Rapport technique, UIUC Technical Report UIUCDCS-R-2003-2328.

HARRIS, M., BAXTER, I., SCHEUERMANN, T. ET LASTRA, A. (2003). Simulation of cloud dynamics on graphics hardware. *HWWS '03 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* 92–101.

HARRIS, M. J., COOMBE, G., SCHEUERMANN, T. ET LASTRA, A. (2002). Physically-based visual simulation on graphics hardware. *HWWS '02 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* 109–118.

HILLIS, W. ET STEELE, G. (1986). Data parallel algorithms. *Communications of the ACM 29(12).*

HORN, D. (2005). *GPU gems2*, Addison Wesley, chapitre Stream reduction operations for GPGPU applications. 573–588.

JAKOBSEN, T. (2001). Advances character physics. *Game Developer Conference.*

KASS, M. ET G., M. (1990). Rapid, stable fluid dynamics for computer graphics. *SIGGRAPH '90 : Proceedings of the 17th annual conference on Computer graphics and interactive techniques.* 49–57.

KHAILANY, B., DALLY, W., RIXNER, S., KAPASI, U., MATTSON, P., NAMKOONG, P., OWENS, J., TOWLES, B. ET CHANG, A. (2001). Imagine : Media processing with streams. *IEEE Micro 21.*

KIM, T. ET LIN, M. (2003). Visual simulation of ice crystal growth. *SCA '03 : Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer*

animation. 86–97.

KIPFER, P., SEGAL, M. ET R., W. (2004). Uberflow : A gpu-based particle engine. *HWWS '04 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* 115–122.

KRAUS, M. ET ERTL, T. (2002). Adaptive texture maps. *HWWS '02 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* 7–15.

KRUGER, J. ET WESTERMANN, R. (2003). Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22, 908–916.

LARSEN, S. ET MCALLISTER, D. (2001). Fast matrix multiplies using graphics hardware. *Supercomputing '01 : Proceedings of the 2001 ACM/IEEE conference on Supercomputing.* 55–55.

LEFOHN, A., KNISS, M., HANSEN, C. ET WHITAKER, R. (2004). A streaming narrow-band algorithm : interactive computation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 10, 422–433.

M., R. ET STRZODKA (2001). Level set segmentation in graphics hardware. *Proceedings of IEEE International Conference on Image Analysis and Processing (ICIP'01).* 1103–1106.

MARK, W., GLANVILLE, R., AKELEY, K. ET KILGARD, M. (2003). Cg : a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22.

MORELAND, K. ET ANGEL, E. (2003). The fft on a gpu. *HWWS '03 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hard-*

ware. 112–119.

NVIDIA (2004). Gelato : Gpu-accelerated final-frame renderer. *http ://film.nvidia.com/page/gelato.html.*

PURCELL, T., BUCK, I., MARK, W. ET HANRAHAN, P. (2002). Ray tracing on programmable graphics hardware. *Proceedings of ACM SIGGRAPH 2002.* vol. 21, 703–712.

REGGIO, M. (2004). *Mth6201 : Systèmes différentiels.* Ecole polytechnique de Montréal.

RIFFEL, A., LEFOHN, A., VIDIMCE, K., LEONE, M. ET OWENS, J. (2004). Mio : fast multipass partitioning via priority-based instruction scheduling. *HWWS '04 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* 35–44.

SHEWCHUK, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain. Rapport technique, Carnegie Mellon University, Pittsburgh, PA, USA.

STRZODKA, R., IHRKE, I. ET MAGNOR, M. (2003). A graphics hardware implementation of the Generalized Hough Transform for fast object recognition, scale, and 3d pose detection. *Proceedings of IEEE International Conference on Image Analysis and Processing (ICIAP'03).* 188–193.

WHALEY, R. ET DONGARRA, J. (1998). Automatically tuned linear algebra software. *Supercomputing '98 : Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM).* 1–27.

WOO, M. ET SHREINER, D. (1999). *OpenGL Programming Guide.* OpenGL

Architecture Review Board, troisième édition.

YOUQUAN, L., XUEHI, L. ET ENHUA, W. (2004). *Real-Time 3D Fluid Simulation on GPU with Complex Obstacles.* Thèse de doctorat, University of Macau.

ANNEXE I

GPU programmables et nuanceurs : mise à niveau technique

En 2001, avec la mise sur le marché des cartes nvidia Geforce3 et ATI Radeon 8000, un changement majeur dans le fonctionnement des cartes graphiques s'est produit avec l'introduction du *pipeline graphique programmable*. Nous laisserons le soin au lecteur de s'informer au besoin du pipeline graphique classique en se référant à l'ouvrage de référence de Woo et Shreiner (1999). Voyons maintenant le fonctionnement du pipeline graphique programmable.

I.1 Le pipeline graphique programmable

Afin de passer d'un pipeline graphique fixe à un pipeline programmable, les constructeurs ont ouvert l'accès à certains des modules de ce pipeline, qui peuvent désormais effectuer des opérations définies par l'utilisateur. La figure I.1 montre comment ces opérations viennent se greffer sur le pipeline par l'intermédiaire des *nuanceurs de pixels et nuanceurs de sommets*. A l'origine, les nuanceurs ont été introduits pour satisfaire la demande de l'industrie du jeu qui désirait effectuer des effets avancés que ne permettait pas le pipeline fixe, tels que le placage de relief (*bump mapping*) ou le placage d'ombre (*shadow mapping*).

Les nuanceurs sont de petits programmes qui indiquent comment traiter les éléments circulant dans le pipeline et se substituent ainsi aux anciens algorithmes de calcul des transformations géométriques, d'éclairage et d'application des textures auparavant utilisés dans le pipeline fixe. Voyons brièvement le fonctionnement des deux familles de nuanceurs.

I.2 Les nuanceurs de sommets

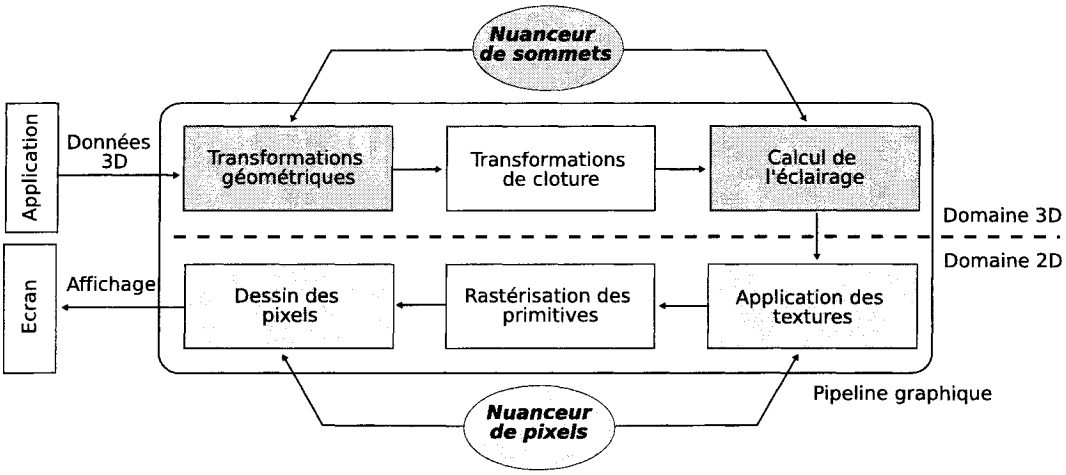

Figure I.1 Intégration des nuanceurs de pixels et nuanceurs de sommets au sein du pipeline graphique

Ils contrôlent la manière dont les sommets 3D sont transformés et éclairés. Plus généralement, ils calculent des fonctions qui sont exécutées sur chacun des sommets parcourant le pipeline et modifient leurs attributs tels que la position, les coordonnées de texture, la couleur, etc. La figure I.2 montre comment fonctionne l'unité de calcul du nuanceur de sommets.

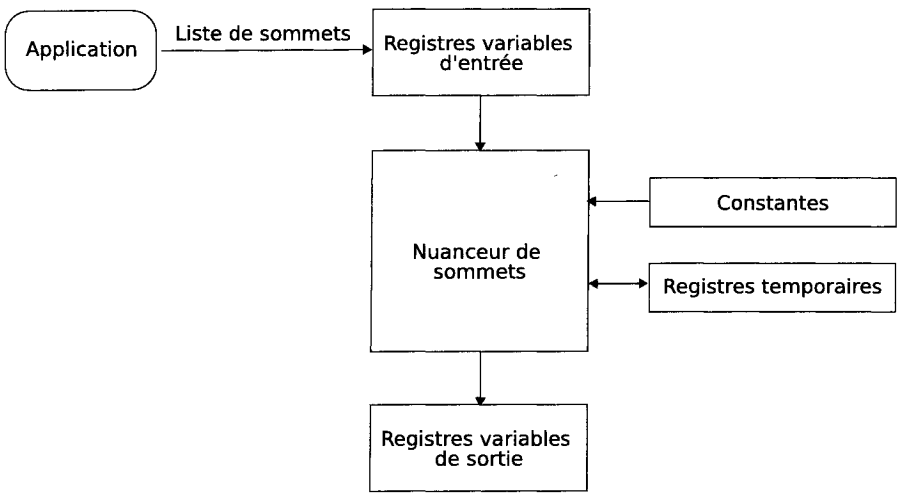

Figure I.2 Modèle de fonctionnement d'un nuanceur de sommets

Tableau I.1 Liste des registres d'entrée du nuanceur de sommets (GeForceFX)

Nom	Type de registre	Nombre	Type	Dimension	Permissions
v	entrée	16	float	4-D	lecture
r	temporaire	12	float	4-D	lecture-écriture
c	constant	256	float	4-D	lecture
i	constant	16	int	4-D	lecture
b	constant	16	booléen	scalaire	lecture
a0	adresse	1	int	4-D	lecture
aL	boucle	1	int	scalaire	lecture
p0	prédicat	1	booléen	4-D	lecture

L'accès aux attributs se fait par l'intermédiaire de registres mémoires, généralement des quadruplets de réels, la variété des registres dépendant de la génération de la carte graphique. Les opérations disponibles sur les registres sont regroupées en un ensemble d'instructions, qui peuvent chacune manipuler un maximum de 4 registres (par exemple un accès en lecture ou en écriture compte pour une instruction).

Tous les registres sont en lecture seule (flèche unidirectionnelle), mis à part quelques registres temporaires locaux utilisés en interne pour le calcul. Les seuls registres n'étant pas issus du pipeline sont les registres de constantes, qui sont contrôlées par le programmeur via l'application OpenGL et passées au nuanceur. Le tableau I.1 montre la liste des registres d'entrée et sortie disponible pour une GeForceFX.

Tous les résultats que l'on veut transmettre plus loin dans le pipeline sont écrits dans les registres de sortie, qui représentent les attributs du sommets, comme l'illustre le tableau I.2.

I.3 Les nuanceurs de pixels

Le principe de fonctionnement des nuanceurs de pixels est le même, les différences concernant seulement le nombre et la nature des registres (figure I.3). Les registres d'entrée sont issus des registres de sortie du nuanceur de pixels préalablement

Tableau I.2 Liste des registres de sortie du nuanceur de sommets (GeForceFX)

Nom	Type de registre	Nombre	Dimension
oD	Couleurs diffuse-spéculaire	2	4-D
oFog	Brouillard	1	scalaire
oPos	Position	1	4-D
oPts	Taille du point	1	scalaire
oT	Coordonnées de texture	8	4-D

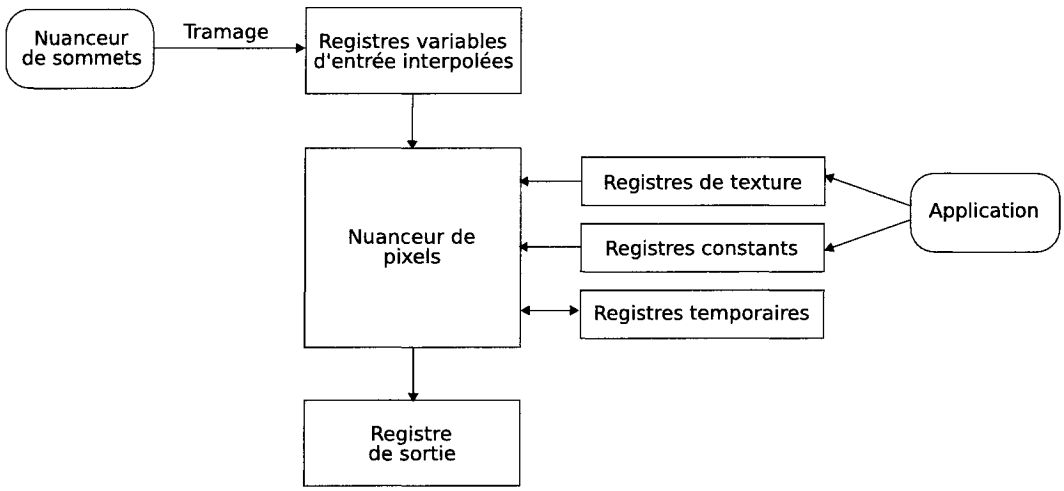

Figure I.3 Modèle de fonctionnement d'un nuanceur de pixels

interpolés pour chaque pixel par le trameur (*Rasterizer*). Quant aux registres de sortie, il n'y en a qu'un seul, à savoir la couleur finale du pixel traité.

Bien que très proches en apparence d'après la figure I.3, les nuanceurs de pixels se distinguent des nuanceurs de sommets par leur capacité à utiliser des registres de texture, donc à lire la mémoire vidéo de la carte. Nous verrons que c'est ce qui justifie l'emploi des nuanceurs de pixels comme outil central pour le calcul générique dans la grande majorité des cas.

Voyons maintenant comment ces nuanceurs s'utilisent et quelle évolution ils ont suivi.

I.4 Programmabilité

Tableau I.3 Liste des registres d'entrée du nuanceur de pixels (GeForceFX)

Nom	Type de registre	type	Nombre	Dimension
v	Couleurs interpolées	float	2	4-D
c	Constantes	float	32	4-D
t	Coord. texture interpolées	float	8	4-D
s	Textures	float	16	4-D
r	Temporaires	float	12	4-D

Tableau I.4 Liste des registres de sortie du nuanceur de pixels (GeForceFX)

Nom	Type de registre	type	Nombre	Dimension
c	Couleur	float	1	4-D

Tableau I.5 Support de structures de code élémentaire des cartes programmables

Nuanceur	nb. instructions max.	if()..then	for()..do	y=f(g(x))	Cartes concernées	Année
1.x	8-14	non	non	non	Geforce3 Radeon8000	2001
2.0	96	non	non	non	Geforce4 Radeon9600	2002
2.0 ét.	512	simulé	partiel	oui	GeforceFX Radeon9800	2003
3.0	illimité	oui	oui	oui	Geforce6800	2004

Il existe plusieurs générations de nuanceurs, chacune apportant des nouveautés plus ou moins importantes. Il est important de connaître leurs différences car elles conditionnent l'implémentation. Nous nous intéressons ici plus particulièrement aux nuanceurs de pixels, puisque ce sont eux qui interviennent principalement dans le calcul générique.

Le tableau I.5 montre l'évolution du support de structures de code élémentaires par différentes générations de cartes et de nuanceurs de pixels. Portons ici une attention particulière à la génération 2.0 étendue, puisque c'est celle qui a été utilisée dans le cadre de la recherche.

La gestion des conditions est considérée comme simulée car les deux codes possibles sont toujours exécutés, mais un seul avec les droits d'écriture dans les registres. De la même manière, les boucles sont partiellement supportées car automatiquement déroulées par le compilateur et ont donc une longueur fixée une fois pour toute. Ceci a un impact très important sur la manière d'aborder le calcul générique sur GPU.

ANNEXE II

Listage du code des principaux shaders

II.1 Combinaison linéaire de deux vecteurs

```
/* Nuanceur de sommet/pixel pour effectuer
la combinaison linéaire de deux vecteurs */

struct app2vert {
    float4 position;
};
struct vert2frag {
    float4 position : POSITION;
    float4 wposition : WPOS;
};
struct frag2frame {
    float4 value : COLOR;
};
vert2frag ScaleVert(app2vert input) {
    vert2frag output;
    output.position = input.position;
    return output;
}
frag2frame ScaleFrag(
    vert2frag input,
    uniform float scale1,
    uniform float scale2,
    uniform samplerRECT source1,
    uniform samplerRECT source2
)
{
    frag2frame output;
    output.value = scale1*f4texRECT(source1, input.wposition)
     + scale2*f4texRECT(source2, input.wposition);

    return output;
    }
```

II.2 Multiplication d'une matrice pleine et d'un vecteur

```
/* Programme Cg pour multiplier une matrice pleine
 par bande bande par un vecteur v1, le resultat est ajoute a v2 */

struct app2vert {
float4 position;
};
struct vert2frag {
float4 position : POSITION;
float4 wposition : WPOS;
};
struct frag2frame {
float4 value : COLOR;
};
vert2frag MatVert(app2vert input) {
vert2frag output;
output.position = input.position;
return output;
}
frag2frame MatFrag
(
vert2frag input,
uniform float2 uv,

// Sampler pour parcourir la texture du vecteur v1 a multiplier
uniform samplerRECT v1,

// Sampler pour parcourir la texture du vecteur v2
uniform samplerRECT v2,

// Samplers pour parcourir les textures des vecteurs extraits de la matrice
uniform samplerRECT m0,
uniform samplerRECT m1,
uniform samplerRECT m2,
uniform samplerRECT m3,

// Sampler pour parcourir la texture du vecteur resultat (en lecture)
uniform samplerRECT res
)
{
frag2frame output;
```

```
float4 scal4 = f4texRECT(v1,uv)  ;
output.value =    scal4.x*f4texRECT(m0, input.wposition)
                + scal4.y*f4texRECT(m1, input.wposition)
                + scal4.z*f4texRECT(m2, input.wposition)
                + scal4.w*f4texRECT(m3, input.wposition)
                + f4texRECT(v2, input.wposition)
                + f4texRECT(res, input.wposition);
return output;
}
```

II.3 Multiplication d'une matrice creuse et d'un vecteur

```
/* Programme Cg pour multiplier une matrice diagonale
par bande bande m1 par un vecteur v1, le resultat est ajoute a v2 */

struct app2vert {
float4 position;
};
struct vert2frag {
float4 position : POSITION;
float4 wposition : WPOS;
};
struct frag2frame {
float4 value : COLOR;
};
vert2frag MatVert(app2vert input) {
vert2frag output;
output.position = input.position;
return output;
}

frag2frame MatFrag
(
vert2frag input,
// Nombre de texel non nuls dans les vecteurs
uniform float N,
// Taille des textures representant les vecteurs
uniform float texsize,
// Offset positif du vecteur m[0] extrait de la matrice
uniform float i,
// Sampler pour lire la texture du vecteur v1 a multiplier
uniform samplerRECT v1,
```

```
// Sampler pour lire la texture du vecteur v2
uniform samplerRECT v2,
uniform samplerRECT m0,// ****************************************
uniform samplerRECT m1,// *
uniform samplerRECT m2,// *
uniform samplerRECT m3,// *
uniform samplerRECT m4,// *
uniform samplerRECT m5,// *
uniform samplerRECT m6,// *   Samplers pour parcourir les textures
uniform samplerRECT m7,// *   des vecteurs extraits de la matrice
uniform samplerRECT m8,// *
uniform samplerRECT m9,// *
uniform samplerRECT m10,// *
uniform samplerRECT m11,// ****************************************
// Sampler utilise pour le precalcul des modulos
uniform samplerRECT offset,
// Sampler pour parcourir la texture
// du vecteur resultat (en lecture)
uniform samplerRECT res)  {

frag2frame output;

// Calcul de l index j de l element a multiplier
// du vecteur m0 extrait de la matrice grace a wpos
float j_m0 = floor((texsize-input.wposition.y-0.5)*texsize
+ input.wposition.x-0.5) ;

// Calcul de l index j2 de l element a multiplier
// du vecteur v1 avec m0 grace a j_m0 et N
float j2 = frac((i+j_m0)/N)*N ;

// Calcul des coordonnees de texture a multiplier pour m0
float2 uv_m0 =
float2(frac(j_m0/texsize)*texsize,texsize-floor(j_m0/texsize)-1.0)
+ float2(0.5,0.5) ;

// Calcul des coordonnees de texture a multiplier avec v1 pour m0
float2 uv_v1_m0 =
float2(frac(j2/texsize)*texsize,texsize-floor(j2/texsize)-1.0)
+ float2(0.5,0.5);

// Calcul des coordonnees uv a multiplier pour les m[i],
// offset sert a calculer recursivement
```

```
// le 0.5 est rajoute pour eviter les pb d'arrondis
// pour les acces aux textures
float2 uv[4];
uv[0]=uv_v1_m0 ;
float2 uvtemp = uv_v1_m0 ;
int k ;
for(k=1;k<4;k++)
{
uvtemp = f2texRECT(offset, uvtemp) ;
uv[k] = uvtemp  ;
}

float4 v1_1 = f4texRECT(v1, uv[0]);
float4 v1_2 = f4texRECT(v1, uv[1]);
float4 v1_3 = f4texRECT(v1, uv[2]);
float4 v1_4 = f4texRECT(v1, uv[3]);

output.value =
  f4texRECT(m0, uv_m0)*v1_1
+ f4texRECT(m1,uv_m0)*float4( v1_1.yzw , v1_2.x)
+ f4texRECT(m2,uv_m0)*float4( v1_1.zw  , v1_2.xy)
+ f4texRECT(m3,uv_m0)*float4( v1_1.w   , v1_2.xyz)
+ f4texRECT(m4,uv_m0)*v1_2
+ f4texRECT(m5,uv_m0)*float4( v1_2.yzw , v1_3.x)
+ f4texRECT(m6,uv_m0)*float4( v1_2.zw  , v1_3.xy)
+ f4texRECT(m7,uv_m0)*float4( v1_2.w   , v1_3.xyz)
+ f4texRECT(m8,uv_m0)*v1_3
+ f4texRECT(m9,uv_m0)*float4( v1_3.yzw , v1_4.x)
+ f4texRECT(m10,uv_m0)*float4(v1_3.zw  , v1_4.xy)
+ f4texRECT(m11,uv_m0)*float4(v1_3.w   , v1_4.xyz)
  + f4texRECT(res, input.wposition)
+ f4texRECT(v2, input.wposition) ;
return output;}
```

II.4 Produit scalaire partiel de deux vecteurs

```
/* Programme Cg pour reduire un vecteur (max,min,norme) par 4 (une passe)*/

struct app2vert
{
    float4 position;
};
```

```
struct vert2frag
{
    float4 position : POSITION;
    float4 wposition : WPOS;
};
struct frag2frame
{
    float4 value : COLOR;
};
vert2frag DotVert(app2vert input)
{
    vert2frag output;
    output.position = input.position;
    return output;
}

frag2frame DotFrag(
vert2frag input,
uniform samplerRECT v1,// vecteur 1 a reduire
uniform samplerRECT v2) // vecteur 2 a reduire
{
    frag2frame output;

    // calcul des coordonnees de texture de la texture a reduire
    // on fait un "scaling" des coordonnees fenetre
    float2 wpos_scale = 2.0*input.wposition ;

     float2 uv1 = wpos_scale + float2(-1,0) ;
     float2 uv2 = wpos_scale ;
     float2 uv3 = wpos_scale + float2(0,-1) ;
     float2 uv4 = wpos_scale + float2(-1,-1) ;

     output.value =    float4(  dot(f4texRECT(v1, uv1),f4texRECT(v2, uv1)) ,
dot(f4texRECT(v1, uv2),f4texRECT(v2, uv2)) ,
dot(f4texRECT(v1, uv3),f4texRECT(v2, uv3)) ,
dot(f4texRECT(v1, uv4),f4texRECT(v2, uv4)) ) ;

    return output;
}
```

II.5 Réduction d'un vecteur

```
 /* Programme Cg pour reduire un vecteur */

struct app2vert
{
    float4 position;
};
struct vert2frag
{
    float4 position : POSITION;
    float4 wposition : WPOS;
};
struct frag2frame
{
    float4 value : COLOR;
};
vert2frag RedVert(app2vert input)
{
    vert2frag output;
    output.position = input.position;
    return output;
}

frag2frame RedFrag(
vert2frag input,
uniform samplerRECT v1) // vecteur 1 a reduire

{
    frag2frame output;

    // calcul des coordonnees de texture de la texture a reduire
    // on fait un "scaling" des coordonnees fenetre
    float2 wpos_scale = 2.0*input.wposition ;

     float2 uv1 = wpos_scale + float2(-1,0) ;
     float2 uv2 = wpos_scale ;
     float2 uv3 = wpos_scale + float2(0,-1) ;
     float2 uv4 = wpos_scale + float2(-1,-1) ;

     output.value = f4texRECT(v1, uv1)+f4texRECT(v1, uv2)+f4texRECT(v1, uv3)+f4texRECT(v1, uv4) ;

    return output;
}
```

II.6 Contraintes de distance (cas du centre)

```
/* integrateur de verlet appliqué sur les éléments du centre du domaine */

struct app2vert
{
    float4 position;
};

struct vert2frag
{
    float4 position : POSITION;
    float4 wposition : WPOS;
};

struct frag2frame
{
    float4 value : COLOR;
};

vert2frag VerletVert(app2vert input)
{
    vert2frag output;
    output.position = input.position;
    return output;
}

frag2frame VerletFrag(
vert2frag input,
        uniform samplerRECT im,
uniform samplerRECT ip,
uniform samplerRECT jm,
uniform samplerRECT jp,
uniform samplerRECT pos,
uniform float3 spherepos,
uniform float sphereradius,
uniform float gridspan,
uniform float stiffness
)
{
float2 uv = input.wposition;
frag2frame output;
```

```
float3 dx_im = f3texRECT(im,uv);
float3 dx_ip = f3texRECT(ip,uv);
float3 dx_jm = f3texRECT(jm,uv);
float3 dx_jp = f3texRECT(jp,uv);

float3 dx = (1.0 - gridspan/length(dx_im))*dx_im +
(1.0 - gridspan/length(dx_ip))*dx_ip +
(1.0 - gridspan/length(dx_jm))*dx_jm +
(1.0 - gridspan/length(dx_jp))*dx_jp ;

float3 x = f3texRECT(pos,uv) + stiffness*dx ;
float3 ds = x-spherepos;
float dist = length(ds);

float3 dummy = 3.0*dx ;
if(dist<sphereradius)
{
//  x = spherepos + ds*(sphereradius);
x = spherepos + ds*(sphereradius / dist);
}

float4 res = float4(x,1.0);

output.value = res ;

    return output;}
```

II.7 Affichage d'un champ de hauteur

```
/* Programme Cg pour transformer un champ de hauteur en champ de normes et simuler la réfraction */

struct app2vert
{
float4 position;
};

struct vert2frag
{
float4 position : POSITION;
float4 wposition : WPOS;
};
```

```
struct frag2frame
{
float4 value : COLOR;
};

vert2frag HeightVert(app2vert input)
{
vert2frag output;
output.position = input.position;
return output;
}

frag2frame HeightFrag
(
vert2frag input,
uniform samplerRECT source,
uniform sampler2D background,
uniform float scale
)
{
frag2frame output;
float2 winpos = input.wposition.xy ;

float4 sample = f4texRECT(source, winpos);
float4 samplehaut = f4texRECT(source, winpos + float2(1.0,0.0) );
float4 sampledroit = f4texRECT(source, winpos + float2(0.0,1.0) );

float3 norm = float3((sample.xx - float2(samplehaut.x,sampledroit.x)) , 0.6 );
norm = normalize(norm) ;
float2 uvdecal = 0.02*norm.xy ;
float atten = dot(norm,float3(1.0,1.0,1.0));
output.value = atten*tex2D(background,scale*winpos+uvdecal);

return output;
}
```

ANNEXE III

Code de la simulation d'ondes

```
#include "gpuSolver.h"

const int gridsize = 256;
const float ce = 0.4 ;
const float timestep = 0.05;
const float alpha = (ce*ce*timestep*timestep)/(2.0*gridspan*gridspan);

void simulationStep()
{
gpuVO->execute(1.0,0.0,pos,NULL,temp);
gpuMO->execute(A_p,pos,NULL,c);
gpuVO->execute(1.0,-1.0,c,oldpos,c);
gpuCG->execute(A,c,pos,gpuMO,gpuVO,gpuVM);
gpuVO->execute(1.0,0.0,temp,NULL,oldpos);

gpuDecomp->execute(pos,posDecomp);
gpuHF->execute(bmp_text,posDecomp,normals);
}

void display()
{
simulationStep();

glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho(-1.0,1.0,-1.0,1.0,-1.0,100.0);
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();

normals->display();

glutSwapBuffers();
}
```

```
void Reshape( int width, int height )
{
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective( 60.0, (double)width / height, 0.1, 100.0 );
glMatrixMode( GL_MODELVIEW );
glViewport( 0, 0, width, height );
}

void Init()
{
initialiservaleurs(p);
initialiservaleurs(p2);

gpuVec *pos = new gpuVec(gridsize*gridsize,p);
gpuVec *oldpos = new gpuVec(gridsize*gridsize,p);
gpuVec *posDecomp = new gpuVec(gridsize*gridsize*4,p2);
gpuVec *normals = new gpuVec(gridsize*gridsize*4,p2);

gpuMat *A_p = new gpuMat(gridsize*gridsize,true);
gpuMat *A = new gpuMat(gridsize*gridsize,true);

initialisermatrice(A_p);
initialisermatrice(A);

gpuProgramVecAdd *gpuVO = new gpuProgramVecAdd();
gpuProgramVecMult *gpuVM = new gpuProgramVecMult();
gpuProgramMatOp *gpuMO = new gpuProgramMatOp();
gpuCGradient *gpuCG = new gpuCGradient();
gpuProgramDecomp *gpuDecomp = new gpuProgramDecomp();
gpuProgramHeightField *gpuHF = new gpuProgramHeightField();
}

int main( int argc, char** argv )
{
glutInit( &argc, argv );
glutInitWindowSize( gridsize, gridsize );
glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
glutCreateWindow( "Water Simulation" );
Init();
glutDisplayFunc( display );
glutMouseFunc( OnMouseButton );
glutIdleFunc( display );
```

```
glutReshapeFunc( Reshape );
glutMainLoop();

return 0;
}
```