

ÉCOLE POLYTECHNIQUE

DE MONTRÉAL

DÉPARTEMENT DE GÉNIE INFORMATIQUE

VISUALISATION SUR GPU D'UN SYSTÈME DE PARTICULES

RÉSULTANT D'UNE SIMULATION NUMÉRIQUE

Rapport de projet de fin d'études soumis
comme condition partielle à l'obtention du
diplôme de baccalauréat en ingénierie

Présenté par : Antoine Azar

Matricule : 1085253

Directeur de projet : Benoît Ozell, Ph.D.

Date : Le 16 Avril 2004

Sommaire

Nous présentons dans cette recherche une technique de calcul de déplacement de particules pour les simulations scientifiques, entièrement basée sur GPU. Plus particulièrement, nous développons un algorithme de haut-niveau pour calculer les déplacements de particules dans un maillage, avec une force appliquée à chaque nœud de ce maillage. Ce genre de problème se retrouve dans de nombreux domaines scientifiques, dont le génie mécanique, civil, ou biomédical.

Plusieurs voies possibles impliquant le vertex et le fragment shader sont explorées. Alors qu'une solution idéale utiliserait le vertex shader, en effectuant un calcul similaire à la technique du *displacement mapping*, les limitations du matériel actuellement disponibles et des APIs graphiques nous forcent à utiliser le fragment shader. Il sera alors nécessaire d'encoder les informations de position des particules dans une texture 2D, et les informations des forces du maillage dans une texture 3D. À partir de ces textures, le fragment shader sera en mesure de calculer les nouvelles positions. Nous aurons donc besoin de plusieurs passes avant de pouvoir afficher la nouvelle image. Quelques moyens existent pour transférer les données des nouvelles positions directement à un tableau de vertex (sans passage par le CPU), mais ces moyens ne sont pas universels et ne fonctionnent qu'avec quelques cartes graphiques. Nous optons donc pour une solution moins efficace mais de fonctionnement assuré, en repassant par le CPU pour la mise à jour des particules.

Nous arrivons ainsi à reproduire avec succès les résultats de simulation tels qu'ils auraient été obtenus via une solution CPU. Nous observons cependant une certaine perte de précision à la sortie du pixel shader. Il est possible que cela soit dû à l'architecture de la carte graphique utilisée, dont le pipeline utilise un format point-flottant à seulement 24 bits, ou à une conversion de format dans le code.

Plusieurs possibilités d'améliorations existent encore. En acceptant d'utiliser des extensions non-standard, ou d'utiliser Direct3D au lieu d'OpenGL, il aurait été possible d'optimiser notre solution. Notons également que dans le futur proche, les nouvelles cartes graphiques et les nouveaux APIs (DirectX 9.0c et OpenGL 2.0) permettront l'élaboration de techniques beaucoup plus élégantes et efficaces que celles utilisées ici.

Table des matières

1. PROBLÉMATIQUE.....	9
1.1 - SIMULATIONS NUMÉRIQUES.....	9
1.2 - ÉVOLUTION DES CARTES GRAPHIQUES VERS DES GPUS.....	10
1.3 - OPTIMISATION DE LA SIMULATION.....	11
1.3.1 - <i>Création et évolution des particules</i>	12
1.3.2 - <i>Localisation d'une particule dans le maillage (identification des nœuds adjacents)</i>	12
1.3.3 - <i>Interpolation des forces aux nœuds adjacents</i>	12
1.3.4 - <i>Calcul de la prochaine position de la particule</i>	13
1.3.5 - <i>Visualisation du système</i>	13
2. MÉTHODOLOGIE.....	14
2.1 - CALCULS VIA LE VERTEX SHADER.....	15
2.1.1 - <i>Modélisation du maillage par une texture 3D</i>	15
2.1.2 - <i>Modélisation du maillage via les constantes sur GPU</i>	16
2.1.3 - <i>Autres considérations</i>	17
2.2 - CALCULS VIA LE FRAGMENT SHADER.....	18
2.2.1 - <i>Modélisation des données par une texture 3D et 2 textures 1D/2D</i>	19
2.2.2 - <i>Utilisation de SuperBuffers</i>	19
2.2.3 - <i>Utilisation de pBuffers</i>	20
2.2.4 - <i>Readback par le CPU</i>	20
2.2.5 - <i>Extensions des APIs</i>	20
2.2.6 - <i>Autres considérations</i>	21
3. SOLUTION RETENUE.....	22
3.1 - ALGORITHME DE SOLUTION.....	22
3.2 - SIMPLIFICATIONS ACCEPTÉES.....	23
3.3 - LOGIQUE ET IMPLANTATION DE L'ALGORITHME ET DU SHADER.....	25
3.4 - CHOIX DES OUTILS LOGICIELS ET MATÉRIELS.....	29
4. RÉSULTATS.....	30
5. DISCUSSION.....	34
5.1 - DIFFICULTÉS RENCONTRÉES.....	34
5.1.1 - <i>Affichage de la texture 2D</i>	34
5.2.2 - <i>Encodage des positions en format RGB</i>	34
5.2.3 - <i>Lecture des valeurs dans la texture 3D</i>	35
5.2 - ÉVALUATION DE L'IMPORTANCE DE LA SOLUTION.....	35
5.3 - AMÉLIORATIONS ACTUELLES POSSIBLES.....	36
5.4 - AMÉLIORATIONS FUTURES POSSIBLES.....	37
6. BIBLIOGRAPHIE.....	38

Remerciements

J'aimerais tout d'abord remercier mon superviseur de projet, M. Benoît Ozell, pour son encadrement, son aide et ses précieux conseils. M. Ozell m'a introduit au monde de la simulation scientifique, et a su me guider vers mon projet en alliant ce domaine à mes intérêts en infographie.

J'aimerais également remercier de tout mon cœur mes parents que j'adore, Sam et Jacqueline Azar, et mes deux grands frères, Ramy et Fred. Mes parents, en plus de leur support et de leur amour, m'ont toujours encouragé à me dépasser et à rechercher l'excellence. Mes frères, tous deux brillants ingénieurs, sont pour moi des modèles extraordinaires tant au point de vue professionnel que personnel. Je les remercie infiniment de tout ce qu'ils ont fait pour moi.

Liste des tableaux

TABLEAU 1-1 - ÉVOLUTION DES GPUS DE NVIDIA DE 1999 À 2003 [1]	10
TABLEAU 3-1 - ALGORITHME DE SOLUTION.....	22
TABLEAU 3-2 - CODE DU FRAGMENT SHADER.....	25
TABLEAU 3-3 - FONCTION D'ITÉRATION.....	26
TABLEAU 3-4 - FONCTION DE RENDU DE LA TEXTURE	27

Liste des figures

FIGURE 1-1 - EXEMPLE SIMPLE DE MAILLAGE STRUCTURÉ	9
FIGURE 2-1 - ORDRE DES OPÉRATIONS D'UN GPU [1].....	11
FIGURE 2-2 - CHAÎNE DE TRAITEMENT D'UN VERTEX SHADER [1]	15
FIGURE 2-3 - CHAÎNE DE TRAITEMENT D'UN FRAGMENT SHADER [1]	18
FIGURE 4-1 - CHARGEMENT INITIAL DU LOGICIEL	30
FIGURE 4-2 - CHARGEMENT D'UN MAILLAGE PLOT3D.....	31
FIGURE 4-3 - CHARGEMENT DU FICHIER DE VECTEURS	31
FIGURE 4-4 - LES PARTICULES À L'INTÉRIEUR DE LA SOURCE.....	32
FIGURE 4-5 LES PARTICULES APRÈS QUELQUES ITÉRATIONS	33

Liste des symboles, abréviations et termes techniques

- CPU :** *Central Processing Unit*, le processeur central de l'ordinateur
- GPU :** *Graphics Processing Unit*, le processeur de la carte graphique
- API :** *Application Programming Interface*, une interface logicielle définissant l'interaction avec des fonctions du système. Des exemples sont OpenGL et Direct3D
- Vertex :** Un point géométrique
- Fragment :** Un élément d'une scène en cours de rendu qui pourrait occuper l'espace d'un pixel dans l'image finale
- Vertex Shader :** Un sous-processeur du GPU qui traite les vertex avec une architecture programmable
- Fragment Shader :** Un sous-processeur du GPU qui traite les fragments avec une architecture programmable

Introduction

Plusieurs expériences, simulations et mesures scientifiques utilisent comme modèle le calcul de déplacement d'une série d'éléments ponctuels (des particules), sous l'influence de certains paramètres comme des forces, des variations de températures ou de pression, etc. Afin de simuler ces modèles, on entrera typiquement ces paramètres dans un logiciel de simulation, qui ensuite générera un maillage avec une solution à chaque nœud. Il devient alors simple et pratique d'étudier les déplacements des particules dans le maillage. Des exemples d'applications incluent la simulation de modèles anatomiques déformables sous l'effet de forces, comme la compression du sein qui est utilisée en détection de cancer, l'écoulement d'eau dans une turbine, ou encore l'étude de déformation d'une poutre dans un pont.

La précision des résultats varie souvent avec la résolution du maillage ainsi que le nombre de particules dans la simulation, et la charge de calcul sur le CPU devient de plus en plus importante. Cependant, avec les progrès modernes très rapides des GPUs (Graphics Processing Units), il est possible de déplacer beaucoup des calculs sur la carte graphique, en laissant le CPU libre pour d'autres tâches.

L'objectif de cette recherche est de déplacer le plus de travail possible du CPU au GPU lors du calcul et de l'affichage des déplacements de chaque particule dans le maillage. On recherchera ainsi à développer un algorithme capable de remplacer l'approche de programmation CPU traditionnelle.

1. Problématique

1.1 - Simulations numériques

Dans beaucoup de domaines scientifiques, tel que le génie mécanique, on cherche à prévoir le comportement d'un élément tel que de l'air ou de l'eau dans un milieu donné. On pourrait chercher par exemple à modéliser l'écoulement d'un fluide dans une turbine. Des modèles mathématiques complexes sont alors élaborés selon les principes physiques en jeu, et on en tire ainsi un résultat de simulation. Ce résultat est souvent sous la forme d'un fichier de données qui représente les forces agissant à des nœuds définis à des positions précises. Un maillage tridimensionnel est ainsi bâti à partir de tous les nœuds, et peut prendre deux formes : structuré (régularité dans la forme géométrique), ou non-structuré (position des nœuds arbitraire). Dans le cadre de cette étude, je me limiterai à la simulation de maillages structurés et cartésiens, c'est-à-dire d'espacement régulier dans toutes les dimensions. Une transition vers des maillages non-cartésiens ne devrait cependant pas poser de défis majeurs additionnels.

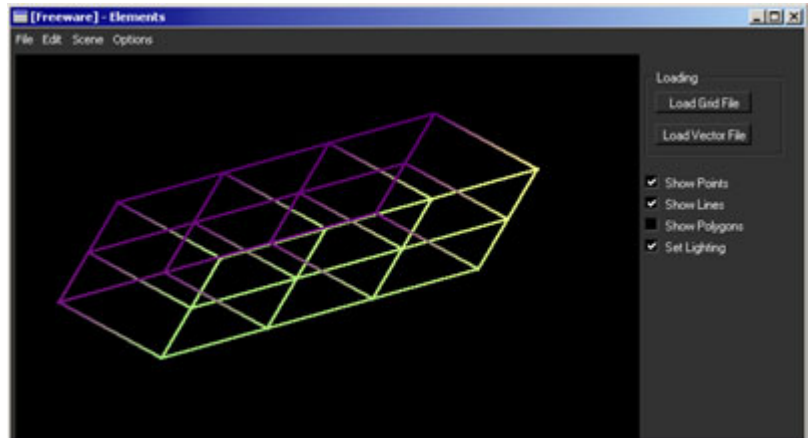


Figure 1-1 - Exemple simple de maillage structuré

Une fois que le maillage est bâti et que les forces appliquées à chaque nœud sont connues, on injecte des particules à un point d'entrée du volume, et on peut alors observer leur comportement alors qu'elles traversent le maillage. Ces simulations peuvent alors servir à optimiser le design d'une pièce, ou encore mieux localiser les points de stress maximaux d'un système.

Il est clair qu'un maillage avec une plus grande résolution (une densité de nœuds plus élevée) fournit des résultats plus précis. De plus, il est souvent utile d'injecter un grand nombre de particules à l'intérieur du système. Ces contraintes imposent une charge élevée de calcul sur le processeur de l'ordinateur de simulation, et limitent donc les

possibilités de calculs additionnels pouvant être effectués simultanément. Une optimisation des ressources de calcul serait donc nécessaire.

1.2 - Évolution des cartes graphiques vers des GPUs

Depuis quelques années, l'industrie des cartes graphiques est en pleine effervescence. Des nouvelles technologies sont maintenant disponibles qui augmentent les capacités des cartes graphique de manière phénoménale. Ainsi, ces cartes ont évolué de simples accélérateurs « fixes » (dont les fonctionnalités sont prédéfinies, comme l'accélération du *Gouraud shading*), vers des unités de traitement général nommées GPU (Graphics Processing Unit). Les GPUs visent à atteindre une flexibilité de calcul comparable à celle des CPUs. Les APIs OpenGL et Direct3D sont constamment mis à jour pour incorporer de nouvelles fonctionnalités, et les vendeurs de cartes graphiques (tels que nVidia et ATI) sont rapides à adapter leurs produits pour supporter les APIs au niveau hardware et software (via les drivers).

Cette augmentation de puissance est extrêmement rapide et surpasse la vitesse de croissance des CPUs. Plusieurs applications utilisent donc à présent les capacités des GPUs pour du traitement, tout en laissant le CPU libre à d'autres tâches. Ainsi, les jeux, par exemple, peuvent effectuer des calculs graphiques complexes sur GPU, et utiliser le CPU pour l'intelligence artificielle, les effets sonores, la communication réseau s'il y a lieu, etc.

Tableau 1-1 - Évolution des GPUs de nVidia de 1999 à 2003 [1]

Année	Nom	Procédé	Transistors	Antialiasing fill rate	Polygon Rate
Début 1999	RIVA TNT2	0.22 μ	9M	75M	9M
Fin 1999	GeForce 256	0.22 μ	23M	120M	15M
Début 2000	GeForce2	0.18 μ	25M	200M	25M
Début 2001	GeForce3	0.15 μ	57M	800M	30M
Début 2002	GeForce4	0.15 μ	63M	1200M	60M
Début 2003	GeForce FX	0.13 μ	125M	2000M	200M

Traditionnellement, la programmation des GPUs se faisait en langage assembleur pour chaque carte graphique. Cependant, des initiatives récentes de nVidia, Microsoft, et du groupe OpenGL, entre autres, ont fait naître plusieurs langages de programmation sur GPU de haut niveau. Un des plus populaires, développé par nVidia, se nomme Cg, pour « C for graphics ». Il porte en effet une forte ressemblance au langage C. Il peut être utilisé via les APIs OpenGL et Direct3D.

1.3 - Optimisation de la simulation

Les nouvelles possibilités des GPUs semblent idéales pour optimiser une simulation. Il faut d'ailleurs noter que les capacités des GPUs, quoique développées à l'origine pour les graphiques, peuvent servir également à bien d'autres tâches (de la détection de collision par exemple). Nous viserons donc à transférer une quantité maximale de traitement au GPU. Ce transfert est toutefois limité par la technologie actuelle des cartes graphiques, qui est encore relativement dans son enfance.

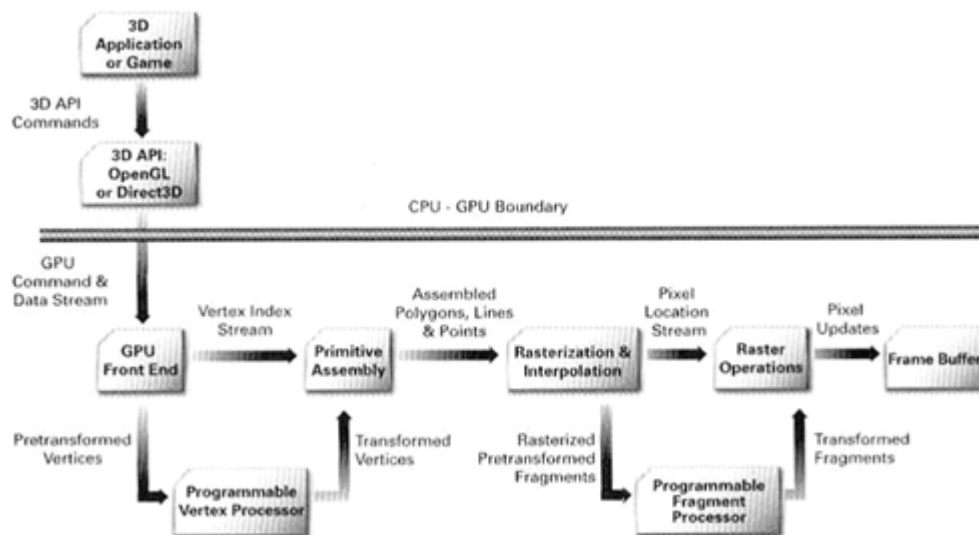


Figure 1-2 - Ordre des opérations d'un GPU [1]

Afin de mieux comprendre les tâches à optimiser, il est nécessaire d'identifier les calculs les plus importants effectués lors d'une simulation :

1. Création et évolution des particules
2. Localisation d'une particule dans le maillage (identification des nœuds adjacents)
3. Interpolation des forces aux nœuds adjacents
4. Calcul de la prochaine position de la particule
5. Visualisation du système

1.3.1 - Création et évolution des particules

Cette tâche peut facilement être développée sur un GPU. Cependant, la plupart des calculs d'un système de particules « standard » implique traditionnellement le calcul de la position d'une particule en fonction de sa position initiale, et d'un ensemble de forces via les lois de Newton. L'application que je cherche à développer ne possède pas un tel requis, et donc il n'y a pas vraiment de traitement important à effectuer pour l'évolution des particules.

1.3.2 - Localisation d'une particule dans le maillage (identification des nœuds adjacents)

Comme le cas étudié se limite aux maillages structurés et cartésiens, il est très simple de déterminer les nœuds entourant la position de chaque particule, et ne nécessite pas de calculs complexes. Dans le cas d'un maillage non-structuré, par contre, il faudrait effectuer une recherche à travers le maillage afin de trouver les nœuds, et cela pourrait être coûteux (proportionnellement à la taille du maillage).

1.3.3 - Interpolation des forces aux nœuds adjacents

Afin de déterminer plus précisément la force appliquée à une particule dans le maillage, il est nécessaire de trouver les nœuds qui l'entourent, et d'effectuer une interpolation (linéaire par exemple) entre les nœuds pour déterminer la force qui s'exerce

à sa position. Ces calculs s'effectuent pour chaque particule, et peuvent devenir coûteux. Il serait donc intéressant de les optimiser via le GPU.

1.3.4 - Calcul de la prochaine position de la particule

Une fois que la force est trouvée, il est simple de calculer la prochaine position de la particule selon un pas de temps donné. Cependant, cela implique tout de même des opérations de multiplication qui doivent être effectuées pour chaque particule, et il serait avantageux de les effectuer sur le GPU.

1.3.5 - Visualisation du système

Dans le cas actuel, la visualisation du système est très simple. Elle est constituée du maillage, ainsi que des particules. Pour ce qui est du maillage, il est simple de le placer dans une *display list* lors de son chargement, vu qu'il ne sera pas modifié subséquent. Pour les particules, il n'y a pas de traitement graphique complexe à effectuer. On peut simplement demander au GPU d'appliquer une texture simple (exemple : une sphère) sur un polygone pour chaque particule, ou encore de seulement faire un rendu d'un point. Pour l'application de la texture (une technique appelée « billboardage »), il existe des techniques pour effectuer les calculs nécessaires (rotation du polygone pour faire face au viewport) directement sur le GPU, dont l'extension OpenGL `ARB_point_sprite`. Les autres opérations peuvent se faire avec des appels OpenGL standards et ne nécessitent pas de programmation additionnelle en Cg.

Par cette analyse, on se rend donc compte que les principaux calculs à optimiser sont l'interpolation des forces aux nœuds adjacents, et le calcul de la prochaine position de la particule. Il faudrait idéalement que l'implantation GPU du système fournisse une performance au moins égale à celle CPU (une performance égale aurait déjà l'avantage de libérer le CPU pour d'autres tâches), et une précision équivalente. Le prochain chapitre concerne les solutions possibles au problème.

2. Méthodologie

Dans cette section, nous présentons un aperçu des solutions envisagées. Chaque solution est décrite du point de vue algorithmique, et analysée brièvement quant à sa faisabilité, sa simplicité et ses possibilités de performance. À partir de ces multiples solutions, nous choisirons celle qui présente le plus d'avantages, et une analyse plus poussée sera effectuée dans la section suivante.

Nous disposons de deux unités principales de traitement pour effectuer les calculs : le **vertex shader** et le **fragment shader**.

Comme on peut le voir à la figure 1-2, le vertex shader, identifié comme « Programmable Vertex Processor », est le premier dans la chaîne de traitement. Il prend en entrée les paramètres des vertex, effectue toutes les transformations requises, puis donne en sortie les primitives assemblées.

Le fragment shader, lui, identifié comme « Programmable Fragment Processor », reçoit en entrée des fragments (qui peuvent être considérés comme des « pixels potentiels » de l'image), effectue le traitement nécessaire, applique les textures, et donne en sortie les fragments transformés. La dernière étape du rendu est de déterminer les fragments qui composeront les pixels de l'image.

Chaque shader a accès à différentes informations et chacun possède différentes capacités de traitement et de stockage des données. Nous étudions ici les possibilités offertes par chacun.

2.1 - Calculs via le Vertex Shader

Comme chaque particule est représentée par un vertex (sur lequel est typiquement appliqué une texture afin de lui donner un rendu désiré), il semble naturel d'utiliser le vertex shader afin de calculer les déplacements et les vitesses des particules. Le problème devient alors de déterminer une façon optimale d'effectuer ces calculs, en donnant au vertex shader toute l'information dont il a besoin.

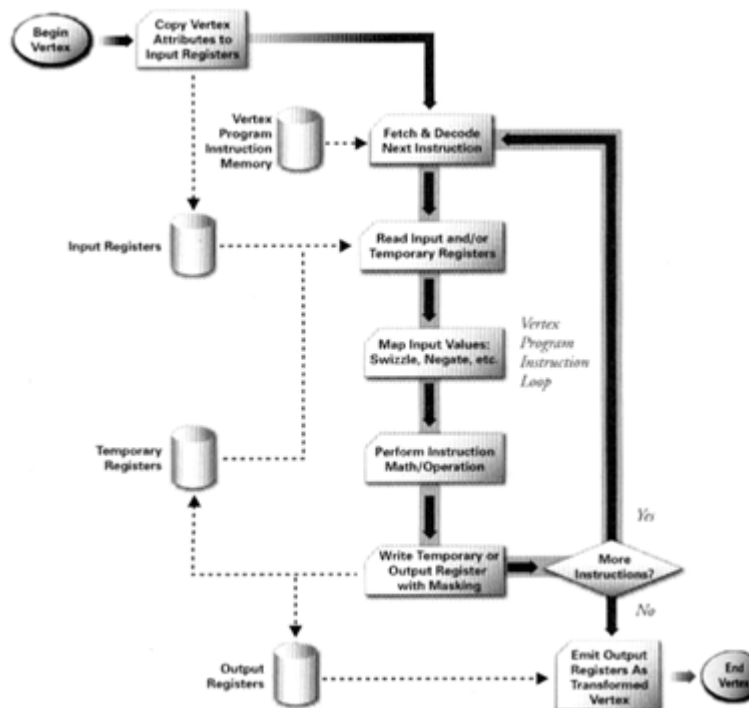


Figure 2-1 - Chaîne de traitement d'un vertex shader [1]

2.1.1 - Modélisation du maillage par une texture 3D

Le maillage est une série de nœuds avec des forces qui s'appliquent à chaque nœud. Chaque force contient des composantes en x , y , et z . Il serait alors possible de bâtir une texture 3D correspondant à notre maillage, en projetant les composantes spatiales en composantes de couleur. Ainsi, la force en x est représentée par une intensité de rouge, la force en y par une intensité de vert, et la force en z par une intensité de bleu.

Lorsque le vertex d'une particule est envoyé au vertex shader, il suffirait alors simplement d'effectuer un *texture lookup* dans notre texture 3D à la position du vertex

pour connaître précisément la force appliquée en ce point (le GPU s'occupe automatiquement de l'interpolation des couleurs dans la texture). Les GPUs étant fortement optimisés pour le traitement de textures, cette solution semble optimale tant au niveau de la simplicité que de la performance. Malheureusement, le vertex shader ne possède actuellement pas d'accès aux textures. Il serait donc impossible de déterminer la force appliquée à la particule. Cette limitation devrait disparaître avec la prochaine génération de cartes graphiques, lorsque le standard VS 3.0 sera défini et implanté. Le *OpenGL Shading Language* semble permettre actuellement l'accès aux textures dans le vertex shader via l'extension `ARB_VerTEX_Shader`¹, mais c'est une émulation software qui est actuellement utilisée.

2.1.2 - Modélisation du maillage via les constantes sur GPU

Une autre solution possible serait d'utiliser la mémoire réservée pour les constantes sur le GPU pour représenter les nœuds adjacents au vertex traité. Cependant, les GPUs actuels ont des tailles relativement petites de constantes, et ce nombre varie de carte graphique en carte graphique.

Il est donc clair qu'il est impensable de représenter le maillage complet dans cet espace mémoire. Il serait donc nécessaire d'implanter un algorithme de découpage multi-zone du maillage. La taille de chaque zone dépendrait du nombre de constantes permises. Ainsi, avant qu'un vertex ne soit traité, il faudrait déterminer la zone dans lequel il se trouve, envoyer les informations des nœuds aux constantes du GPU, puis laisser le vertex shader déterminer les nœuds exacts à interpoler et calculer la nouvelle position.

Cette approche, quoique théoriquement possible, ne semble pas présenter d'avantage. Le CPU serait tout de même sollicité pour déterminer la zone du vertex, et il y aurait une perte de performance puisqu'il serait nécessaire de constamment mettre à jour les valeurs des constantes.

¹ http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_shader.txt

2.1.3 - Autres considérations

Quoique la solution de la texture 3D semble optimale, une ombre se dessine au tableau : le résultat du vertex shader (dans notre cas, la nouvelle position de la particule) n'est envoyé qu'au fragment shader pour l'affichage. Ainsi, la valeur de la position de la particule dans notre programme (dans la mémoire centrale de l'ordinateur) n'est jamais mise à jour. Chaque itération serait donc semblable à la dernière, avec les particules évoluant constamment du temps 0 au temps 1 (en considérant un intervalle de temps unitaire). Des développements sont en cours pour permettre au vertex shader de modifier les données mêmes qui lui sont envoyées, mais les cartes graphiques et les APIs actuels ne permettent pas encore cette fonctionnalité.

2.2 - Calculs via le Fragment Shader

À cause des limitations inhérentes au vertex shader, nous devons nous tourner au fragment shader pour une solution au problème. L'approche n'est pas évidente : le fragment shader n'est conçu que pour traiter des fragments et ne prend pas en compte les vertex. Cependant, en acceptant des solutions multi-passes (qui requièrent plusieurs rendus afin d'obtenir le résultat désiré) et en faisant une utilisation originale des ressources données, de nouvelles possibilités s'ouvrent à nous.

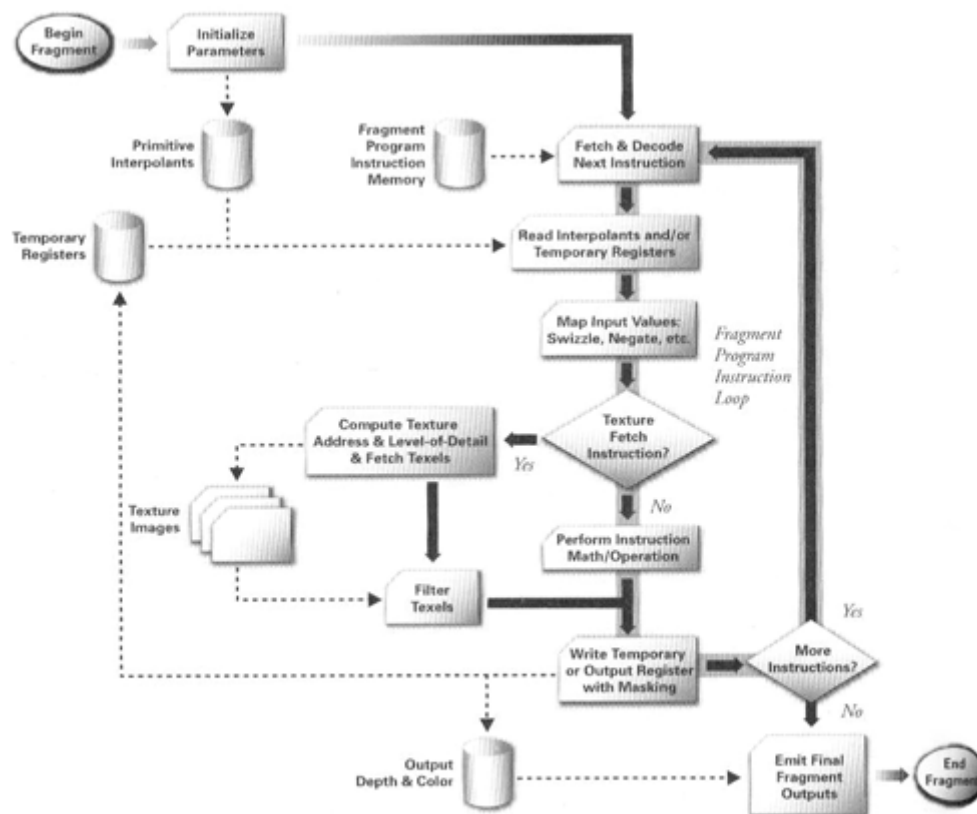


Figure 2-2 - Chaîne de traitement d'un fragment shader [1]

2.2.1 - Modélisation des données par une texture 3D et 2 textures 1D/2D

Tout comme la première solution du vertex shader, nous modélisons ici le maillage dans une texture 3D avec une projection des composantes des forces sur les composantes de couleur. Cependant, comme aucun traitement n'est effectué sur les vertex des particules dans le vertex shader, nous devons trouver un moyen d'effectuer toutes nos modifications des données des particules dans le fragment shader.

Le fragment shader possède un accès aux textures. Ainsi, nous n'avons plus de problème pour accéder à la texture 3D et calculer la force à appliquer à la particule. Afin de pouvoir effectuer le calcul, il faut d'abord encoder les positions et les vitesses des particules dans des textures 1D (ou 2D). Ainsi, pour la *n*ème particule, le fragment shader pourrait prendre le *n*ème texel (*texture element*) de la texture de position, effectuer un *texture lookup* dans la texture 3D à cette position, prendre le *n*ème texel de la texture de vitesse, et à partir de ces informations calculer la nouvelle position et la nouvelle vitesse de la particule et les restocker dans leurs positions respectives dans les textures 1D/2D.

Un nouveau problème se pose. Il faut à présent effectuer une deuxième passe afin d'afficher les particules à leurs nouvelles positions, ainsi que de mettre à jour les textures. Quelques solutions se présentent à nous : les SuperBuffers, les pBuffers, le *readback* par CPU, et les extensions d'API.

2.2.2 - Utilisation de SuperBuffers

Les SuperBuffers sont des espaces de mémoire alloués sur la carte graphique qui permettent une utilisation variée. Typiquement, un espace mémoire est réservé pour une tâche bien spécifique : un framebuffer, une texture, un tableau de vertex, etc. Les SuperBuffers éliminent cette limitation. Ainsi, il devient entre autres possible de faire un rendu dans un tableau de vertex (*render to vertex array*). La deuxième passe ne ferait donc qu'afficher le tableau de vertex, et les particules apparaîtraient à leurs nouvelles positions.

2.2.3 - Utilisation de pBuffers

Les pBuffers sont moins évolués que les SuperBuffers, mais permettent de faire un rendu directement à une texture, et sont actuellement disponibles sur les cartes graphiques modernes. C'est une fonctionnalité qui est requise pour notre application, vu que les textures doivent être mises à jour d'itération en itération, mais il n'y a toujours pas de moyen de convertir l'information de la texture en position de particules. Pour cela, nous aurons besoin de la prochaine technique, le *readback* par le CPU.

Un autre avantage des pBuffers est qu'ils permettent un format graphique totalement différent de celui utilisé pour le buffer d'affichage, ainsi qu'une taille non-limitée à la fenêtre.

2.2.4 - Readback par le CPU

Une autre possibilité, très simple, est de faire un rendu du fragment shader au *back framebuffer*, puis ensuite de faire un appel de lecture des pixels (en OpenGL, `glReadPixels` pour lire le *framebuffer* en mémoire principale, et `glCopyTexImage2D` pour transférer le contenu du *framebuffer* à une texture). Les valeurs peuvent alors être lues individuellement et interprétées comme des positions pour les particules. Cela implique cependant du travail pour le CPU et le bus AGP, puisque les données doivent faire un aller-retour de la carte graphique jusqu'à la mémoire centrale.

2.2.5 - Extensions des APIs

Une solution à mi-chemin entre les deux précédentes serait d'utiliser des extensions des APIs qui permettent de transférer les données de texture à un tableau de vertex directement sur la carte graphique, sans passer par la mémoire centrale. Nvidia, par exemple, possède des extensions qui permettent cette fonctionnalité, comme `NV_Pixel_Data_Range` et `NV_Vertex_Array_Range`. Cette solution n'offre pas la convivialité des SuperBuffers, mais devrait offrir une bien meilleure performance que la solution de *readback* par le CPU.

2.2.6 - Autres considérations

Si nous désirons tenir compte de la vitesse de la particule lors de son évolution, il faut utiliser tel que décrit en 2.2.1 deux textures 1D/2D (une pour la position des particules, et une pour la vitesse). Si la vitesse n'a pas besoin d'être prise en compte, une seule texture peut être utilisée.

Ce point est important, car la mise à jour de deux textures ne se fait pas facilement. Soit, on peut effectuer deux passes de rendu, la première mettant à jour la texture de positions et la deuxième la texture de vitesses, puis effectuer encore une passe de rendu pour l'affichage, mais cela n'est pas efficace. Il existe une autre méthode, nommée *Multiple Render Targets* (MRT), qui est actuellement disponible via l'API Direct3D. Cette technique n'est cependant pas encore implantée en OpenGL standard, mais le sera probablement dans le futur proche. ATI a déjà pris les devants avec son extension OpenGL `ATI_draw_buffers` qui permet l'écriture à plusieurs tampons de mémoire, mais cette extension n'est compatible qu'avec les cartes ATI modernes. Il faudra sans doute attendre une extension du groupe ARB pour que les MRTs soient standards.

3. Solution retenue

3.1 - Algorithme de solution

Après examen de toutes les solutions possibles au chapitre 2, il est clair qu'avec la génération actuelle de cartes graphiques, nous devons utiliser le fragment shader. Nous retenons alors la méthode de modélisation des données par une texture 3D et 2 textures 1D/2D. De plus, afin de mettre à jour les positions de particules et l'information de texture d'itération en itération, nous optons pour le *readback* par CPU, étant donné sa simplicité et son support universel. En effet, certaines cartes graphiques ne supportent pas les pBuffers dans leurs drivers, et les extensions d'API ne sont pas disponibles par tous les vendeurs. La solution optimale serait les SuperBuffers, mais ceux-ci ne sont pas encore disponibles.

De plus, pour ce qui est du rendu simultané des deux textures, nous ferons deux passes de rendu pour la mise à jour des textures, puis une nouvelle passe pour l'affichage.

Avec ce choix de solutions, nous sommes en mesure de présenter l'algorithme qui sera utilisé :

Tableau 3.1 - Algorithme de solution

<ul style="list-style-type: none"> - Lire le fichier de simulation (ex: Plot3D) - Encoder le maillage dans une texture 3D en projetant les composantes vectorielles XYZ sur les composantes de couleur RGB - Lire les attributs des particules (position initiale, vitesse, etc) - Encoder chaque attribut dans une texture 1D ou 2D - Pour chaque nouvelle image: <ul style="list-style-type: none"> o Pour chaque attribut: <ul style="list-style-type: none"> ▪ Charger le fragment shader dont la sortie correspond à l'attribut courant ▪ Indiquer au fragment shader les textures à utiliser ▪ Lier la sortie du fragment shader à la texture de l'attribut courant ▪ Afficher dans le back-buffer n'importe laquelle des textures 1D/2D sur un polygone aligné à la fenêtre d'affichage * ▪ Dans le fragment shader, pour la particule 0 à n :
--

- Lire les attributs de la particule en effectuant des *lookups* dans les textures 1D/2D à la position exacte de la particule (position dans la texture même, et non dans le monde réel)
- Lire la force interpolée appliquée à cette particule en faisant un *lookup* dans la texture 3D à la position dans le monde réel
- Calculer les nouveaux attributs de la particule
- Faire le rendu
 - Décharger le fragment shader
 - Convertir la texture de position à un tableau de vertex.
 - Afficher le tableau de vertex dans le back-buffer
 - Inverser le back et le front buffer.

*Afin d'effectuer le calcul sur chaque particule, nous devons nous assurer que le polygone affiché respecte un alignement parfait de texel à pixel. De cette façon, le fragment shader sera appelé une fois pour chaque texel.

3.2 - Simplifications acceptées

Nous avons accepté les simplifications suivantes du système afin de rendre la tâche réalisable en l'espace de temps donné et de pouvoir nous concentrer sur les points majeurs du système :

- Les vitesses des particules ne sont pas conservées en mémoire. Ainsi, nous n'avons besoin que d'une texture bidimensionnelle qui conserve la position des particules. Les nouvelles positions se calculent simplement en fonction des valeurs aux nœuds adjacents à la particule. Ceci n'est pas conforme à la réalité, mais il est simple d'ajouter cette fonctionnalité une fois que la mise à jour des positions fonctionne. En OpenGL, cela requiert actuellement une passe de rendu additionnelle, ainsi qu'une ligne de code en plus dans le shader pour tenir compte de la vitesse.

- La simulation à l'intérieur du logiciel se fait image par image. Nous n'avons pas implanté de mécanisme d'animation. Quoique ceci rajouterait à l'esthétique du logiciel, ce n'est pas une fonctionnalité majeure pour notre recherche.
- La source de particules est très limitée en fonctions. Pour l'utilisateur du programme, il peut simplement modifier sa position. Les particules sont ensuite générées aléatoirement autour de la position de départ (dans un rayon donné). De plus, le nombre de particules est fixe : elles n'ont pas de temps limité de vie, et la source ne régénère pas de nouvelles particules. Il est cependant possible pour l'utilisateur de réinitialiser le système et de replacer les particules à la source. Notons que la génération continue de particules entraîne un nouveau problème: le ré-encodage constant de la texture de position. Comme il n'y aurait plus moyen de s'assurer que l'on conserve en tout temps un nombre de particules égal à une puissance de deux, il faudrait utiliser du « padding » sur cette texture. Au fur et à mesure que des nouvelles particules seraient créées, on pourrait utiliser les texels du padding pour stocker leurs données de position. Dans notre cas, ceci serait assez simple, vu que nous utilisons la technique de *readback* par CPU pour lire les nouvelles positions. Nous pouvons alors facilement filtrer les positions des particules courantes des positions « bidon » des particules de padding. Si l'on utilisait une autre technique n'impliquant pas le CPU, cependant, il faudrait trouver une autre solution. On pourrait alors encoder la texture en format RGBA, et se servir du canal alpha pour indiquer si la particule est valide (1) ou non (0).
- En OpenGL, les textures doivent typiquement avoir des dimensions qui sont des puissances de 2. Il existe des extensions pour utiliser des textures d'autres dimensions, mais nous ne les avons pas utilisées. De plus, afin de ne pas compliquer excessivement le calcul des positions, nous utilisons dans nos tests des maillages dont les dimensions sont conformes (chaque dimension est une puissance de deux), ainsi que 64 particules (qui constitue une texture 2D de 8x8). Nous pourrions utiliser du « padding »

afin de pouvoir simuler d'autres maillages, mais cela pose un nouveau problème : comme les informations de couleur des textures sont limitées entre 0 et 1 en format point flottant, il faut porter une attention particulière à la position des particules dans un maillage ayant subi du « padding », pour s'assurer que leurs positions sont correctes et qu'elles sont bien à l'intérieur de nœuds valides. Ceci ajoute donc des contraintes et des efforts au niveau de l'implantation qui ne font pas partie du cadre de cette recherche.

3.3 – Logique et implantation de l'algorithme et du shader

Nous présentons ici le code du fragment shader, qui a la responsabilité de mettre à jour les particules :

Tableau 3-2 – Code du fragment shader

```

struct PS_OUTPUT
{
    float3 pos : COLOR;
};

PS_OUTPUT main(float2 point : TEXCOORD,
               uniform sampler2D PosTexture,
               /*uniform sampler3D VelocityTexture,*/
               uniform sampler3D ForceTexture,
               uniform float deltaTime,
               uniform float3 velocity)
{
    PS_OUTPUT output;

    // point représente le texel courant, pas la position
    //de la particule
    float3 position = tex2D(PosTexture, point);
    //float3 velocity = tex2D(VelocityTexture, point);
    float3 force     = tex3D(ForceTexture,  position);

    float3 distance = velocity*deltaTime + 0.5*force*deltaTime*deltaTime;
    output.pos = position + distance;

    return output;
}

```

Nous avons originellement implanté le support pour la texture de vitesses, mais l'avons subséquemment retiré (d'où les commentaires) et remplacé par une vitesse

constante attribuée à toutes les particules, *velocity*, que l'on peut facilement régler à partir du code C++.

Le calcul de la nouvelle position est simple : il utilise l'équation de base de la cinématique, soit :

$$\Delta x = v_i t + \frac{1}{2} a t^2$$

En supposant que les forces sont normalisées (ou en considérant simplement des particules de masse unitaire), on peut alors remplacer l'accélération par la force et on obtient donc :

$$\Delta x = v_i t + \frac{1}{2} F t^2$$

C'est bien cette équation qui a été implantée dans le shader.

Tous les paramètres de la fonction à part la coordonnée de texture sont réglées à partir du code C++. La coordonnée de texture, elle, est envoyée automatiquement à partir du vertex shader.

Nous présentons également ici les deux fonctions C++ responsables de la réalisation de l'algorithme : `iterateParticleTextures()`, qui est responsable de chaque itération, et `renderTexQuad()`, qui fait le rendu de la texture avec un alignement parfait entre texels et pixels dans le backbuffer.

Tableau 3-3 – Fonction d'itération

```
void Viewer3D::iterateParticleTextures()
{
    int partSize1=0;
    int partSize2=0;
    ePoint3D *p = m_partGen->getParticlePos(partSize1, partSize2);
    //ePoint3D *s = m_partGen->getParticleSpeed();

    if (cg_enable)
    {
        cgGLEnableProfile(cgVertexProfile);

        // Bind Our Vertex Program To The Current State
        cgGLBindProgram(cgProgram);
    }

    //load in Cg the position texture
    CGparameter PosTexture = cgGetNamedParameter(cgProgram,
```

```

                                                                    "PosTexture");
cgGLSetTextureParameter(PosTexture, m_texName[0]);
cgGLEnableTextureParameter(PosTexture);
//load in Cg the force texture
CGparameter ForceTexture = cgGetNamedParameter(cgProgram,
                                                                    "ForceTexture");

cgGLSetTextureParameter(ForceTexture,
                        m_vectorMesh->get3DTextureID());
cgGLEnableTextureParameter(ForceTexture);

renderTexQuad(partSize1, partSize2, true);

cgGLDisableTextureParameter(PosTexture);
cgGLDisableTextureParameter(ForceTexture);

//Unload fragment shader
if (cg_enable)
    cgGLDisableProfile(cgVertexProfile);

glReadBuffer(GL_BACK);
GLfloat *buffer = (GLfloat *)malloc( partSize1 * partSize2 * 3 *
                                    sizeof(GLfloat));
glReadPixels(0,0,partSize1,partSize2,GL_RGB,GL_FLOAT,buffer);
m_partGen->setNewParticlePositions(buffer, partSize1*partSize2);
free(buffer);

//restore original viewport
glViewport( 0, 0, width(), height());
if(m_bOrtho)
    glOrtho( m_virtX0, m_virtX1, m_virtY0, m_virtY1, -100, 100 );
else
    glFrustum(m_virtX0, m_virtX1, m_virtY0, m_virtY1, 0.1, 100 );
}

```

Tableau 3-4 – Fonction de rendu de la texture

```

void Viewer3D::renderTexQuad(int partSize1, int partSize2, bool
changeViewport)
{
    makeCurrent();
    glColor3f(1.0, 1.0, 1.0);
    glClearColor( 0.0, 0.0, 0.0, 0.0 ); // Let OpenGL clear to black
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    GLboolean val = false;
    glGetBooleanv(GL_LIGHTING, &val);
    glDisable( GL_LIGHTING );
    glDisable(GL_COLOR_MATERIAL);

    glDisable(GL_TEXTURE_3D_EXT);
    glEnable(GL_TEXTURE_2D);

    //display texture with texels/pixels aligned
    if(m_texName[0] != 0)
    {

```

```

        if(changeViewport) //allows for rendering the texture to
            //debug
            glViewport(0,0, partSize1, partSize2);
        glMatrixMode(GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
        gluOrtho2D(0.0, 1.0, 0.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glLoadIdentity();

        glBindTexture(GL_TEXTURE_2D, m_texName[0]);

        glBegin(GL_QUADS);

            glTexCoord2f(0.0f, 0.0f);
            glVertex3f(0.0f, 0.0f, 0.5f);

            glTexCoord2f(1.0f, 0.0f);
            glVertex3f(1.0f,0.0f, 0.5f);

            glTexCoord2f(1.0f, 1.0f);
            glVertex3f(1.0f,1.0f, 0.5f);

            glTexCoord2f(0.0f, 1.0f);
            glVertex3f(0.0f,1.0f, 0.5f);

        glEnd();

        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        glPopMatrix();
    }

    glFlush();
    glDisable(GL_TEXTURE_2D);
    glEnable(GL_TEXTURE_3D_EXT);

    if(val)
        glEnable(GL_LIGHTING);
}

```

Nous avons ajouté à cette fonction une variable booléenne, `changeViewport`, qui permettait d'afficher la texture sur toute la largeur du viewport. Ainsi, l'appel à `renderTexQuad` à partir de `iterateParticleTextures` était inchangé, mais on remplaçait le code de la fonction de rendu (`paintGL()`) par un appel à `renderTexQuad` en forçant la variable booléenne à `false`. Cela était fort utile lors du débogage, pour

examiner les variations de couleur de la texture (qui indiquaient donc le mouvement des particules).

3.4 - Choix des outils logiciels et matériels

Afin de réaliser la simulation, le langage de programmation C++ a été choisi, avec la librairie OpenGL pour la visualisation. La programmation sur GPU est faite avec le langage Cg développé par nVidia, à l'aide du Cg Toolkit, et l'interface graphique du programme est écrite avec la librairie QT de Trolltech. L'environnement de développement est Visual C++ 6.0 sous Windows XP.

L'ordinateur utilisé est un Dell Dimension 8300 équipé d'un processeur Pentium IV à 2.66Ghz, d'1GB RAM, et d'une carte graphique ATI Radeon 9800 PRO avec les drivers *Catalyst 3.7*

4. Résultats

Un logiciel a été écrit spécialement pour ce projet. Le logiciel lit en entrée des fichiers de format Plot3D, génère le maillage, et l’affiche en 3D dans la fenêtre de visualisation. Il est possible de demander un rendu sous différentes formes du logiciel (points, lignes, polygones, avec ou sans éclairage, etc), et toutes les opérations standard ont été implantées, tel que :

- Translation dans le monde
- Agrandissement / réduction (scaling)
- Rotation du monde, selon le modèle « trackball »
- Transition de mode perspective à orthogonal et vice-versa

Il est difficile de montrer sur papier l’évolution d’un système de particules. Ainsi, cette section de résultats présentera simplement quelques captures d’écran représentatives du logiciel et de son utilisation.

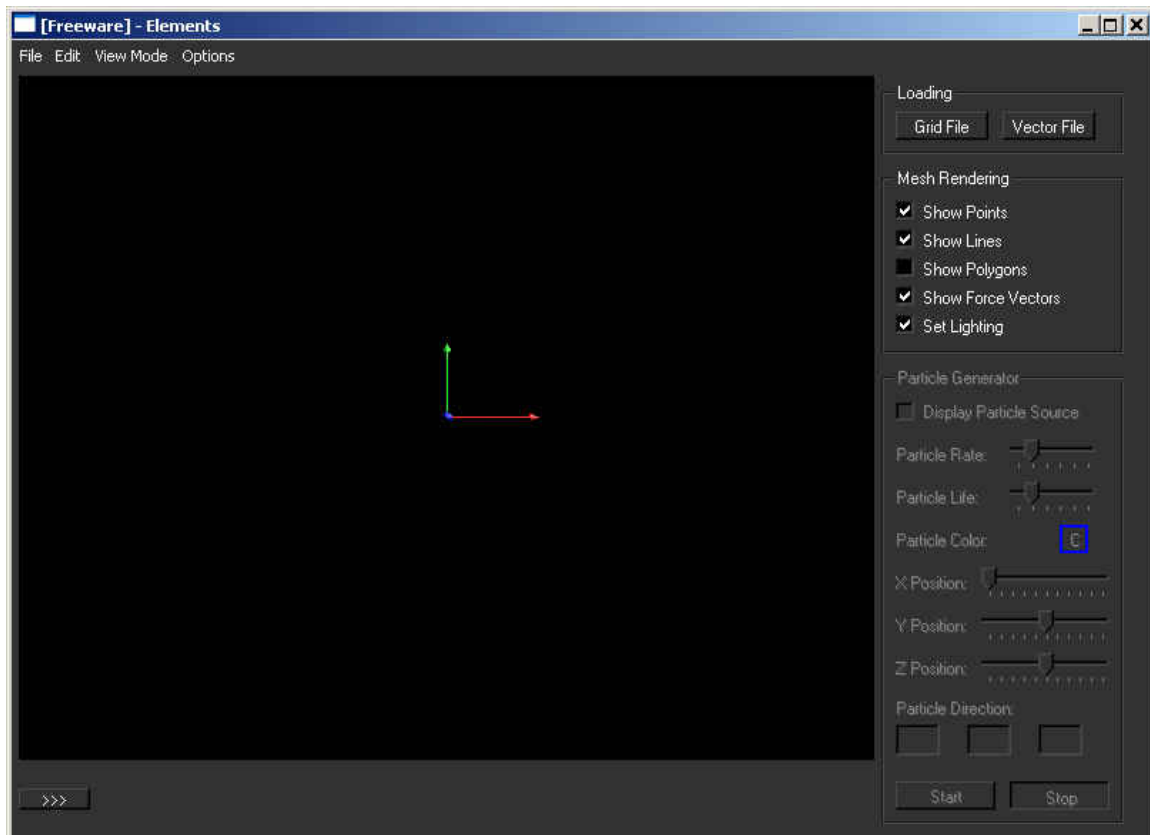


Figure 4-1 - Chargement initial du logiciel

Au chargement du logiciel, l'utilisateur peut cliquer sur « Grid File » pour charger le maillage Plot3D désiré. Voici le résultat d'un chargement (et de quelques rotations) :

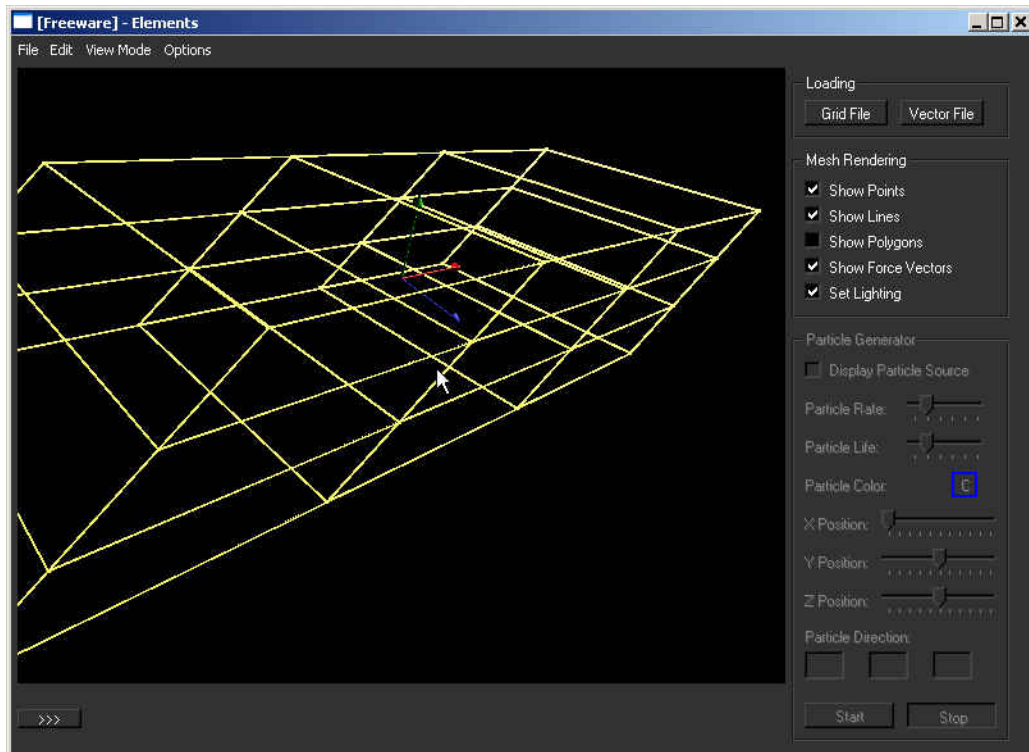


Figure 4-2 - Chargement d'un maillage Plot3D 4x4x2

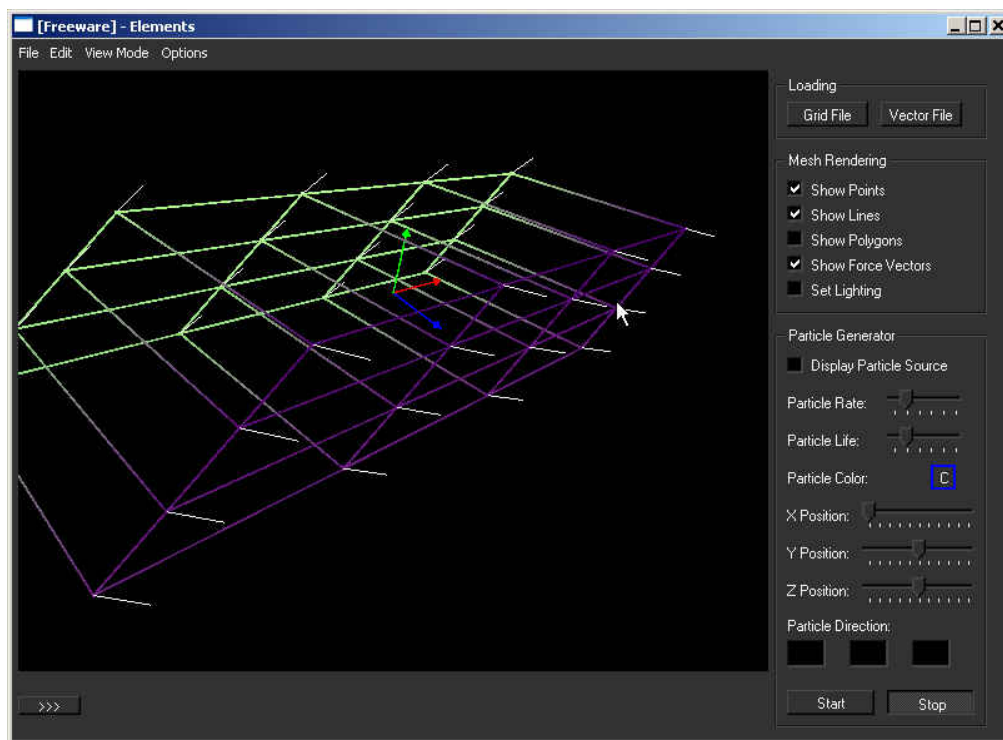


Figure 4-3 - Chargement du fichier de vecteurs

Une fois le maillage et son fichier de vecteurs chargés, l'utilisateur est prêt pour débiter la simulation. Il peut cliquer sur « Display Particle Source », puis sur « Start » :

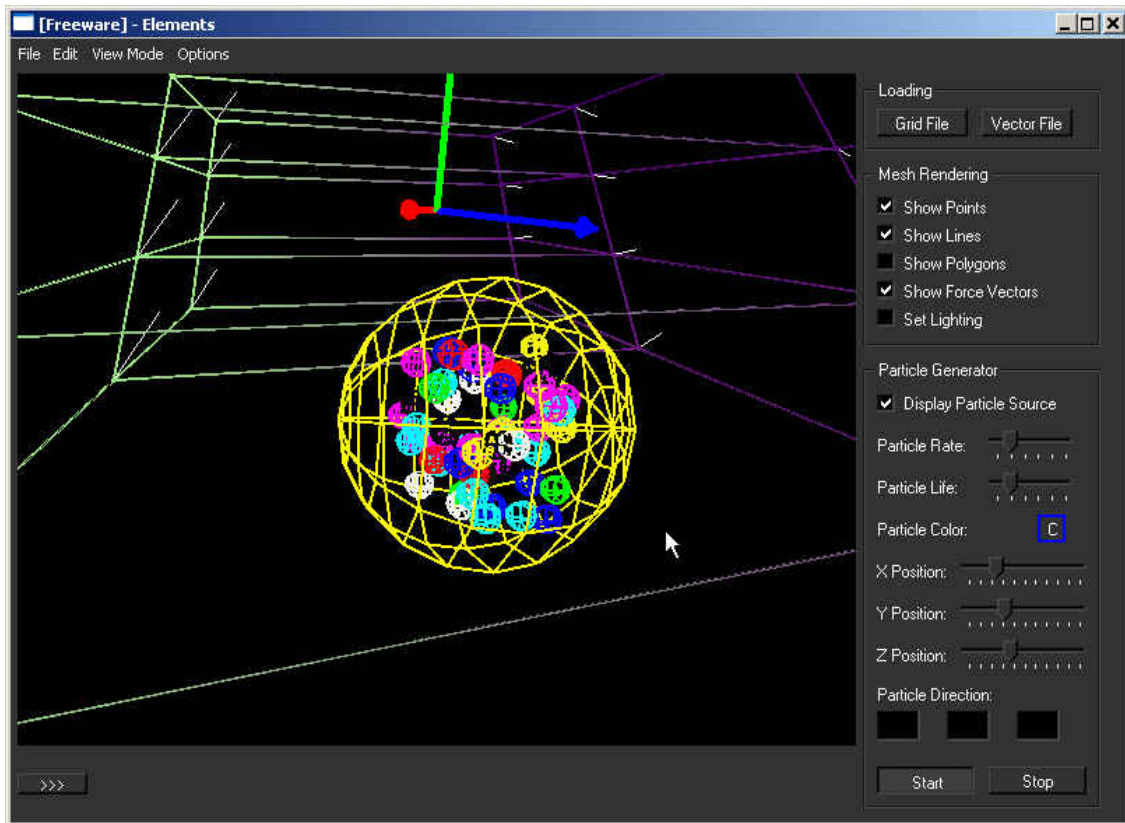


Figure 4-4 - Les particules à l'intérieur de la source

Soixante-quatre particules sont générées à des positions aléatoires à l'intérieur du volume de la source. Le volume de cette source peut être facilement changé dans le code. Les couleurs sont également aléatoires. Il aurait été possible (et probablement plus esthétique) de représenter les particules par des « billboards », mais la solution plus simple de sphères a été retenue.

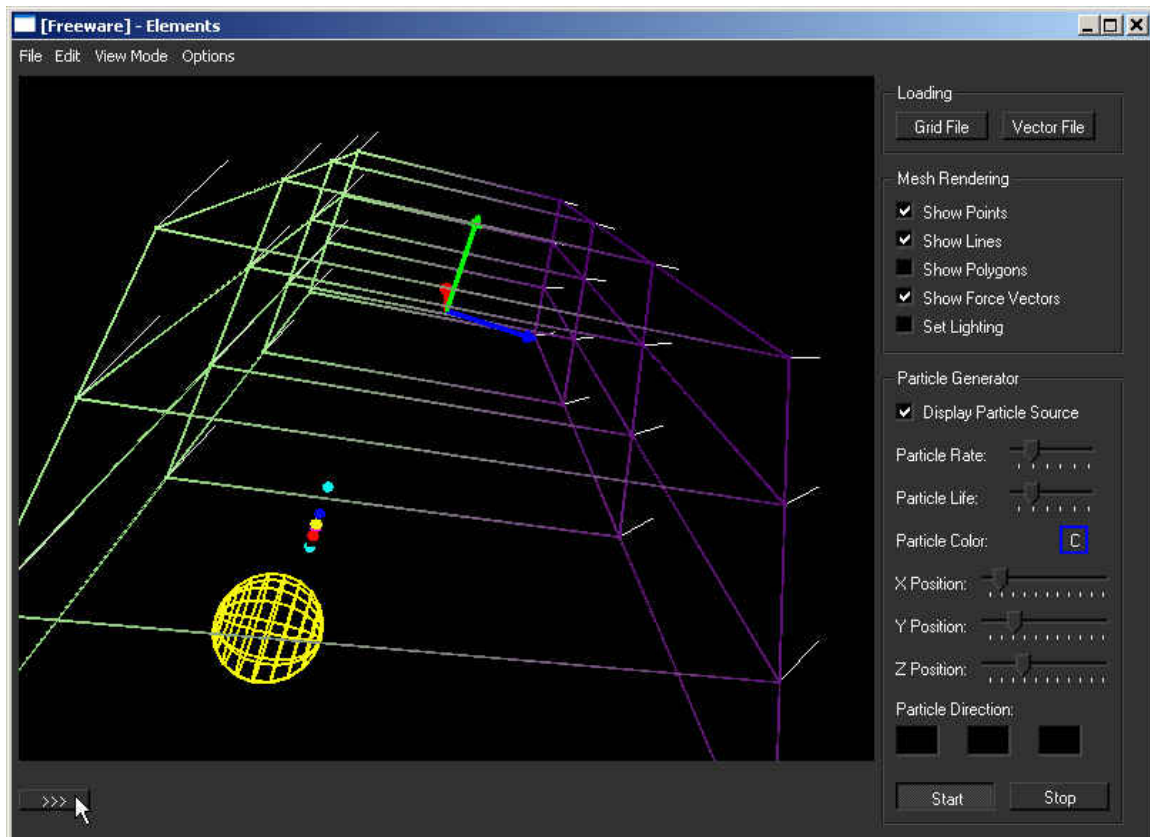


Figure 4-5 - Les particules après quelques itérations

On voit ici les particules en mouvement selon le champ de forces qui les entoure. On remarque également que les particules ont une tendance à s'aligner, d'où un certain soupçon de perte de précision dans le processus. Cette perte de précision se produit peut-être lors de la lecture de la texture dans le backbuffer, lors de conversions de formats double à float, ou encore dans le pipeline de la Radeon (qui utilise des formats point-flottant à 24 bits au lieu de 32).

5. Discussion

5.1 - Difficultés rencontrées

Même après avoir longuement recherché les différentes solutions possibles et avoir identifié la seule méthode réalisable, plusieurs difficultés se sont présentées lors de l'implantation du logiciel. Certaines de ces difficultés furent éliminées simplement en ne les prenant pas en compte, tel que décrit à la section 3.2 « Simplifications acceptées ». D'autres, par contre, devaient être absolument résolues afin d'assurer le bon fonctionnement de la simulation.

5.1.1 – Affichage de la texture 2D

Le premier problème survint lors de l'affichage dans le back buffer de la texture de positions, afin d'activer le fragment shader pour chaque texel. Des résultats insensés apparaissaient. Nous avons alors désactivé le fragment shader et observé les nouvelles positions ; Celles-ci étaient encore différentes des positions originales. Enfin, nous avons modifié notre logiciel afin de permettre le rendu directement de la texture, au lieu de la conserver uniquement dans le back buffer. Nous avons alors remarqué que le polygone s'affichait sans aucune trace de la texture, l'affichage d'une texture 2D sur un polygone rectangulaire étant pourtant une opération triviale. Nous avons finalement constaté qu'il était nécessaire de désactiver la fonction OpenGL de textures 3D en plus d'activer les textures 2D. Au meilleur de nos connaissances, cette précaution n'est cependant indiquée nul part dans la littérature.

5.2.2 – Encodage des positions en format RGB

Un deuxième problème fit surface alors que les positions des particules évoluaient. En effet, comme les positions XYZ sont encodées en format RGB point-flottant, la plage de valeurs possibles ne varie qu'entre 0 et 1. Or, notre maillage avait été initialement construit dans son repère d'origine, qui ne souffrait d'aucune contrainte. Nous avons donc dû insérer les mécanismes nécessaires pour tout remettre à l'échelle, en nous assurant qu'aucune dimension ne dépasse la valeur de 1. Pour ce faire, nous avons

simplement soustrait à toutes les positions des nœuds du maillage la position minimum (afin de déplacer le minimum à 0), puis divisé chaque position par la taille de la plus grande dimension. Ainsi, si la dimension en X varie de -5 à 15 , celle en Y de -10 à -5 , et celle en Z de 0 à 2 , elles sont redimensionnées en X de 0 à 1 , en Y de 0 à 0.25 , et en Z de 0 à 0.1 .

5.2.3 - Lecture des valeurs dans la texture 3D

Le logiciel a été testé principalement sur la machine décrite en 3.3, mais il a également été essayé sur une machine additionnelle, équipée d'une GeforceFX 5200. Sur la machine principale, les lectures dans la texture 3D s'effectuent sans problème. Cependant, nous avons observé sur la seconde machine que ces lectures retournaient constamment la valeur de 0. Malheureusement, nous n'avons pas été en mesure de corriger ce problème dû à un accès limité à cet ordinateur. Le problème peut donc venir soit du code, soit des drivers de la carte, de la version du cg toolkit, ou encore de la version OpenGL installée sur la machine.

5.2 - Évaluation de l'importance de la solution

La mise en œuvre de cette solution a demandé beaucoup de réflexion, de débogage, et de travail. Il est donc pertinent de se questionner de l'utilité et de l'importance d'une solution GPU au problème des simulations scientifiques, par rapport à une simple implantation CPU qui aurait nécessité seulement une fraction du travail. Il est clair que l'utilisation du GPU libère le CPU pour d'autres tâches ; Cependant, tel que mentionné dans la section 2.2 « Calculs via le Fragment Shader », il ne semble pas exister actuellement de solution universelle pour conserver tout le traitement sur le GPU. La conversion des données de texture en données de positions doit encore passer par le CPU. De plus, il est difficile d'ajouter de nouvelles fonctionnalités au logiciel à cause de toutes les contraintes qu'il doit respecter pour la solution GPU.

Du côté de la performance, nous n'avons pas effectué de tests de comparaison entre l'implantation GPU et une implantation CPU. Il aurait été délicat d'effectuer une

telle comparaison d'ailleurs : à défaut d'avoir accès à une grande gamme de processeurs et de cartes graphiques, il faudrait se résoudre à comparer un processeur et une carte graphique qui ne sont peut-être pas de même génération. De plus, il aurait été discutable d'implanter sur GPU une solution optimale mais non universelle, comme des extensions disponibles exclusivement sur des cartes nVidia.

On peut se baser cependant sur une autre étude portant sur un sujet similaire pour obtenir une idée des performances d'une solution GPU. Dans [4], GoodNight et al. implantent une technique GPU pour la résolution de problèmes de valeur limite. Les auteurs ne semblent cependant pas effectuer de visualisation du système, et donc éliminent tout le problème de la conversion des données de texture en données de position. Le rendu de leur fragment shader est dirigé immédiatement à la texture, pour l'itération suivante, probablement en utilisant des pBuffers, tel que décrit ici en 2.2.3. Les auteurs trouvent ainsi un facteur d'accélération d'environ 15 entre la solution GPU (utilisant une Radeon 9700 PRO) et la solution CPU (utilisant un AMD Athlon 1600 avec 1GB de RAM).

En se fiant sur ces résultats, nous pouvons alors affirmer que pour certains problèmes typiques, une solution GPU, quoique plus complexe de réalisation, peut être très avantageuse. Il est donc important de continuer à explorer de telles issues, et nous prévoyons qu'avec les avancées imminentes des technologies graphiques, de tels problèmes seront beaucoup plus facilement abordables.

5.3 - Améliorations actuelles possibles

Il existe plusieurs améliorations que l'on pourrait apporter au logiciel actuel. On pourrait tout d'abord facilement activer la deuxième texture de vitesse des particules, afin d'obtenir des simulations plus réalistes. Ensuite, il serait intéressant de modifier la source de particules afin de permettre à l'utilisateur de modifier plus de paramètres, incluant le temps de vie des particules, le taux de génération, les vitesses initiales, etc. La lecture en entrée du fichier Plot3D pourrait être ajustée pour permettre des dimensions différentes de puissances de 2.

Enfin, il est possible d'optimiser la solution en adoptant les techniques décrites en 2.2.3 et 2.2.5.

5.4 - Améliorations futures possibles

Les améliorations prévues pour le futur sont très encourageantes. Il y a à peine deux jours, la nouvelle carte *GeForce 6800 Ultra* de nVidia fut annoncée. Cette carte supporte la technique de « displacement mapping », et devrait donc permettre le déplacement des vertex beaucoup plus facilement via le vertex shader. Il n'est cependant pas clair si le nouveau vertex shader supporte l'export des données à un emplacement mémoire (idéalement directement sur la carte graphique), chose essentielle à la conservation des données des particules. De plus, avec l'arrivée d'OpenGL 2.0 et des drivers en conséquence des vendeurs de cartes graphiques, nous devrions bénéficier de nombreuses nouvelles fonctionnalités, tel que les SuperBuffers, ou les MRTs. La *GeForce 6800 Ultra* supporte déjà 4 MRTs, mais cette fonction n'est pas encore implantée dans l'API OpenGL.

La conversion d'algorithmes CPU en algorithmes GPU devrait donc devenir de plus en plus populaire, vu les possibilités d'amélioration de performance possibles ainsi que les fonctionnalités croissantes des cartes graphiques.

6. Bibliographie

Livres

- [1] Fernando R., Kilgard M., *The CG Tutorial : The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley, USA, February 2003, 336 pages.
- [2] Woo M., Neider J., Davis T., Shreiner D., *OpenGL Programming Guide*, 3rd edition, Addison-Wesley, USA, Octobre 1999, 730 pages.
- [3] Watt A., Policarpo F., *The Computer Image*, Addison-Wesley, USA, 1999, 751 pages.

Articles

- [4] Goodnight N., Woolley C., Lewin G., Luebke D., Humphreys G., *A Multigrid Solver for Boundary Value Problems using Graphics Hardware*, Proceedings of Graphics Hardware 2003, 2003, 4 pages.
- [5] Percy James, *OpenGL Extensions*, ATI Research, Siggraph 2003, 42 pages.
- [6] Green Simon, *Stupid OpenGL Shader Tricks*, nVidia Corporation, Game Developers Conference 2003, 29 pages.
- [7] Moule Kevin, *Making Good On The Real-Time Promise*, University of Waterloo, 2001, 29 pages.
- [8] Harris M., Coombe G., Scheuermann T., Lastra A., *Physically-Based Visual Simulation on Graphics Hardware*, University of North Carolina, USA, 2002, 11 pages.

Références Internet

- [9] Harris Mark, *Dynamic Texturing*, nVidia Corporation.
<http://developer.nvidia.com/attach/1250>
- [10] CodeSampler.com, *OpenGL (1.2-1.5) Code Samples*
<http://www.codesampler.com/oglsrsrc.htm>
- [11] Qt Online Reference Documentation, Trolltech, 2001
<http://doc.trolltech.com/2.3/index.html>