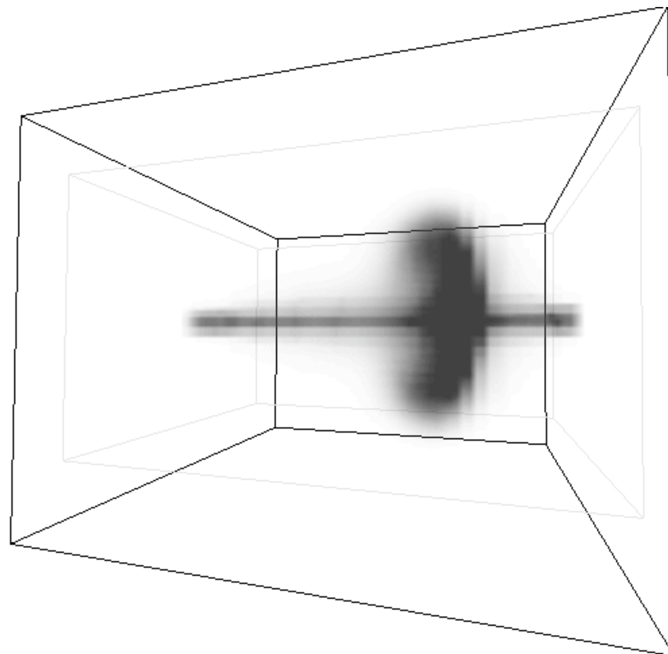
Simulation et visualisation d'un fluide 3D

Michael Ash

Septembre 2005



Remerciements

Je remercie mon maître de stage, Sophie Robert, et toute l'équipe de Réalité Virtuelle au LIFO. Je remercie aussi Jonathan Albers et North Dakota State University pour m'avoir donné accès à sa grappe de PC pour faire des tests de performances, Ed Wynne pour ses conseils sur l'optimisation de l'algorithme de lancer de rayons, et Eugene Ciurana pour son aide avec l'analyse de performances des réseaux.

Table des matières

1 Introduction **5**
1.1 Sujet du travail . . . 5

2 L'état de l'art du rendu volumique **6**
2.1 Applications . . . 6
2.2 Techniques actuelles . . . 6
2.2.1 Lancer de rayons . . . 6
2.2.2 Technique de recherche de surfaces . . . 7

3 Simulation de fluide **8**
3.1 La physique des fluides . . . 8
3.2 Simulation . . . 9
3.2.1 Diffusion de vélocité . . . 9
3.2.2 Advection de vélocité . . . 9
3.2.3 Diffusion et advection de densité . . . 9
3.3 Extension 3D . . . 9

4 Rendu volumique **10**
4.1 Techniques . . . 10
4.1.1 Voxels . . . 11
4.1.2 Tranches . . . 11
4.1.3 Lancer de rayons . . . 12

5 Optimisation de fluide **14**
5.1 Conversions int-float . . . 15
5.2 Les accès mémoire . . . 15
5.2.1 Ordre de boucles . . . 16
5.3 Résultats . . . 17
5.4 Autres possibilités . . . 17

6 Optimisation de lancer de rayons **19**
6.1 Calcul des adresses . . . 21
6.2 Accès mémoire . . . 22

7 Parallélisme **23**
7.1 Extension de fluide en parallèle . . . 23
7.2 Extension de lancer de rayons en parallèle . . . 26
7.2.1 Répartition du cube de données . . . 26
7.2.2 Partage de l'écran virtuel . . . 26
7.3 Réseau complet de l'application . . . 27
7.3.1 Schéma de communication . . . 27
7.3.2 Protocoles . . . 28

8 Analyse **30**
8.1 Analyse théorique de fluide . . . 30
8.2 Analyse théorique du rendu . . . 31

9 Données de performance **31**
9.1 Scalabilité . . . 31
9.1.1 Analyse de la scalabilité de fluide . . . 32
9.2 Analyse de la scalabilité de rendu . . . 33
9.3 Résultats des tests . . . 34
9.3.1 Fluide sur la grappe de NDSU . . . 34
9.3.2 Fluide sur la grappe du LIFO . . . 35
9.3.3 Données statiques sur la grappe de NDSU . . . 37
9.3.4 Données statiques sur la grappe du LIFO . . . 37
9.3.5 Données du CEA sur la grappe du LIFO . . . 38
9.3.6 Tests de LOD sur les données du CEA au LIFO . . . 38

10 Conclusion **39**
10.1 Directions futures . . . 40
10.1.1 Liens entre simulation et rendu . . . 40
10.1.2 Schémas de calcul dans la simulation . . . 40
10.1.3 Communication asynchrone . . . 40

1 Introduction

La visualisation scientifique devient de plus en plus important dans la physique, l'ingénierie, l'aviation, et beaucoup d'autres domaines. Enquêter sur les origines de l'univers, architecturer une centrale nucléaire, dessiner un l'aile d'un avion, toutes ces activités profitent de la visualisation scientifique sur ordinateur. Visualiser une simulation en 3D interactif permet à un scientifique d'avoir une meilleur compréhension de son sujet et ses résultats. Les humains ont beaucoup de facilité à comprendre une forme, une interaction, ou une évolution en forme d'image. La visualisation sur ordinateur permet d'exploiter cette facilité pour mieux communiquer de l'ordinateur à l'humain.

La progression de l'état de l'art de la simulation et instrumentation scientifique augmente toujours la quantité des données produites. La visualisation de ces données demande alors des nouvelles techniques pour traiter ces données.

Les grappes de PCs permettent de traiter des grandes quantités de données en utilisant le matériel informatique standard. Éviter le matériel spécialisé permet d'avoir une performance par prix supérieure pour les applications convenable.

1.1 Sujet du travail

Nous avons travaillé sur une simulation de fluide en 3D. La simulation de fluide est une application typique et utile dans la simulation scientifique qui produit des données difficiles à visualiser avec les techniques classiques de la visualisation 3D. Le but de ce travail a été de rechercher les techniques disponibles pour la visualisation de cette simulation, d'augmenter la taille des données qu'il est possible de traiter et de les visualiser en utilisant les grappes de PCs.

La simulation d'un fluide en 3D produit des données volumiques. Ce sont des données organisées en une grille en trois dimensions. Cette organisation de données est très différente de l'organisation classique qui ne comprend que les surfaces 2D dans l'espace 3D. La méthode pour afficher ces données s'appelle le rendu volumique.

Le travail s'est déroulé pendant une période de cinq mois pendant l'été 2005 au LIFO à l'Université d'Orléans. La première partie de la recherche a consisté en un état de l'art sur le rendu volumique et sur la simulation de fluide. La deuxième partie a consisté à concevoir et mettre en œuvre une simulation de fluide et sa visualisation à grande échelle. Il s'agissait donc d'optimiser les algorithmes retenus et de réaliser un déploiement de l'application sur grappes de PC pour augmenter la taille des données traitées.

Ce rapport est organisé de la manière suivante. Dans un premier temps l'état de l'art de la simulation du fluide 3D et le rendu volumique sont décrits et analysés. Ensuite nous détaillons le rendu volumique, les différentes techniques existantes d'après leurs avantages et les problèmes qu'elles induisent en vu d'une parallélisation sur grappe de PCs. Ensuite nous discutons de l'optimisation et de la parallélisation de l'application que nous avons choisi d'implanter. Finalement une dernière partie permet d'analyser nos travaux au regard des performances théoriques et des performances réelles obtenues pendant les tests sur deux grappes de PC.

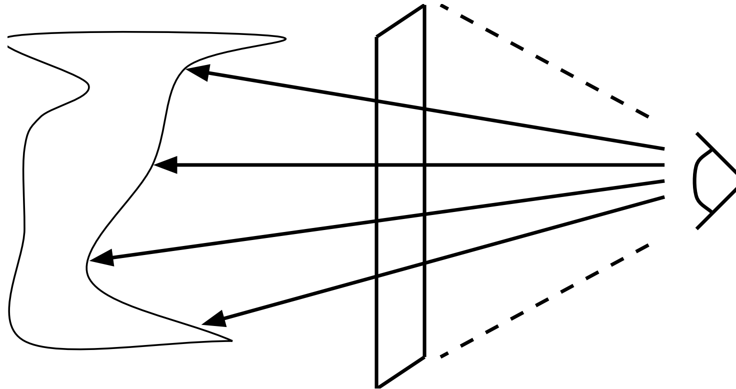


Figure 1: Le rendu avec lancer de rayons. À droite, la caméra, au milieu, l'écran virtuel, et à gauche, l'objet à afficher.

2 L'état de l'art du rendu volumique

2.1 Applications

Le rendu volumique est une technique très coûteuse en temps de calculs et qui nécessite également beaucoup de mémoire vive pour stocker ses données.

Une des premières applications importantes est la visualisation des données médicales. En effet, les scanners médicaux comme le CT ou MRI produisent une grande quantité de données 3D. Un scan CT génère beaucoup d'images de coupes dans le corps du sujet. Il est bien sûr possible de regarder directement ces images 2D, mais ces coupes peuvent être difficiles à comprendre. Rassembler ces coupes en une seule image 3D donne un meilleur survol des données. Initialement, les capacités informatiques limitées ne permettaient qu'une visualisation en seulement deux dimensions de ces données. Mais depuis les années 90 il est possible de visualiser ces données en trois dimensions avec le rendu volumique, qui donne une bien meilleure compréhension à l'utilisateur [4].

2.2 Techniques actuelles

Actuellement, on peut citer deux principales techniques de rendu volumique qui sont largement utilisées dans le domaine du graphisme avec quelques variantes. Une technique est l'algorithme de lancer de rayons qui simule la physique de la lumière dans les données volumiques, et l'autre technique consiste à chercher des surfaces dans les données, qui donne des objets 2D que l'ordinateur peut afficher avec les techniques de rendu 3D traditionnelles.

2.2.1 Lancer de rayons

La technique de lancer de rayons consiste à créer un écran virtuel, et ensuite à tracer le chemin des rayons de lumière pendant leur parcours dans les données volumiques pour déterminer la couleur de chaque pixel. Cette technique s'utilise souvent pour dessiner du brouillard dans les jeux, et pour les données médicales [3] [6].

Voxels Une technique plus simple est de dessiner chaque voxel directement sur l'écran en dessinant un ou plusieurs polygones ou points dans l'espace 3D où il se trouve. Cette technique est très facile à implanter cependant le rendu est très lent. Cette variante est en fait une simplification de la technique de lancer de rayons dans le sens mathématique, mais son implantation est extrêmement différente d'un lancer de rayons classique et a l'avantage d'être moins complexe.

Une équipe de Mitsubishi a développé une architecture pour une carte graphique qui peut rendre des données volumiques assez lourdes avec un matériel de bas coût. Cette architecture utilise un stockage en mémoire qui assure que beaucoup de processeurs peuvent travailler en même temps sur les mêmes données. Ce parallélisme permet un affichage rapide, jusqu'à 30 affichages/seconde pour un cube de 256^3. Cette architecture est basée sur la technique du voxel [15].

Coupes et textures 3D Il est possible d'utiliser une carte graphique traditionnelle qui prend en charge les textures 3D pour accélérer le rendu volumique. L'architecture de Mitsubishi promet un rendu rapide, mais le matériel est spécialisé et donc cher. Les cartes graphiques traditionnelles sont fabriquées à grande échelle ce qui les rend moins chères et plus rapides. Une telle carte est aussi d'usage universel qui nous permet d'utiliser le même matériel pour d'autres applications.

Les données volumiques sont chargées sur la carte graphique comme texture 3D. Le cube à afficher est coupé en tranches qui font face à la caméra. Cette procédure donne une suite de polygones. En appliquant la texture 3D, chaque polygone montre une tranche de la texture 3D sur l'écran. Quand l'ordinateur dessine ces tranches dans un ordre de l'arrière vers l'avant, l'utilisateur voit le cube volumique sur l'écran. Comme la technique de voxels, cette technique est une simplification mathématique de lancer de rayons, mais l'implantation est très différente. Cette technique utilise la carte graphique qui se trouve dans presque chaque ordinateur moderne, ce qui permet d'utiliser le processeur central pour d'autres tâches [14].

2.2.2 Technique de recherche de surfaces

L'autre technique actuelle est une technique de surfaces. Les régions sont définies dans le cube volumique, et ensuite l'ordinateur cherche des surfaces entre les régions. Par exemple, une région définie par une sphère dans le cube volumique met les valeurs des cellules intérieures de la sphère à 1 et les valeurs extérieures à 0. Cette technique reconstruit la surface de la sphère à partir des valeurs de toutes les cellules du cube.

Une simulation de fluide peut définir une région où la densité est supérieure à une certaine valeur, et cette technique cherche les isosurfaces où la densité est égale à cette valeur. Les données médicales pourraient par exemple définir les régions comme le cerveau, le crâne, les artères, etc., et chercher les surfaces entre elles. Après avoir défini les régions, l'algorithme de recherche de surfaces crée les surfaces.

L'algorithme courant de recherche de surfaces est l'algorithme Marching Cubes (cubes marchant). Cet algorithme parcourt les données par groupes de huit voxels, chaque groupe formant cube. Sur chaque coin du cube, la définition des régions décide s'il est à l'extérieur ou à l'intérieur. En prenant en compte

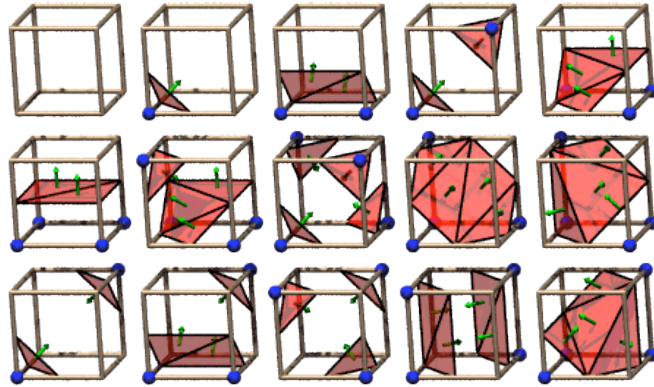


Figure 2: Les 15 combinaisons possibles pour chaque étape de Marching Cubes [13]

tous les rotations et réflexions possibles, il y a seulement 15 combinaisons possibles (voir Figure 2), chacune avec un ensemble unique de polygones qui sont ajoutés à la surface.

Les algorithmes de recherche de surfaces sont assez lents, et en général trop lents pour être utilisés en temps-réel. Cependant sur des données statiques pour lesquelles nous n'avons besoin que d'une application de l'algorithme, cette lenteur n'est pas très importante. Après avoir appliqué l'algorithme, l'ordinateur travaille directement avec les surfaces 2D [12].

3 Simulation de fluide

Les fluides sont récurrents en sciences physiques. Un avion, un bateau, une centrale nucléaire, tout dépend de la mécanique des fluides. Les mathématiques de la mécanique des fluides n'admettent pas de solutions exactes sauf dans les cas très simples. La simulation de fluide numérique qui génère une solution approximative est alors très utile.

Jos Stam [7] propose une simulation de fluide en deux dimensions basée sur la physique du fluide mais avec un coût de calcul qui convient à un logiciel sur un PC standard. Nous avons développé une simulation à partir de la proposition de Stam et on l'a généralisée en trois dimensions.

3.1 La physique des fluides

Un fluide est modelé comme un champ de vecteurs qui représentent la vélocité du fluide et un champ de scalaires qui représentent la densité du fluide. L'évolution du fluide peut être déterminée par les équations de Navier-Stokes.

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + v\nabla^2\mathbf{u} + \mathbf{f}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa\nabla^2\rho + S$$

La première équation détermine l'évolution des vitesses et la deuxième détermine l'évolution des densités.

3.2 Simulation

Pour une simulation sur l'ordinateur il faut discrétiser le fluide. Cette simulation met le fluide dans une grille où chaque cellule contient une vélocité et une densité. L'évolution du fluide se déroule en plusieurs étapes sur la vélocité et la densité.

3.2.1 Diffusion de vélocité

Le deuxième terme dans les équations de Navier-Stokes détermine la diffusion dans le fluide. La première étape de la simulation est la diffusion des valeurs de vélocité. L'étape de diffusion cherche une évolution de fluide qui vérifie le terme de diffusion dans les équations. Un solveur itératif de Gauss-Seidel dans la fonction `lin_solve` cherche une solution. Cette fonction est appliquée sur chaque composante de la vélocité.

Le solveur génère des solutions qui ne respectent pas la conservation de masse dans le fluide. Une autre fonction `project` utilisant aussi un solveur de Gauss-Seidel est appliquée pour transformer le champ de vélocité afin qu'il respecte la conservation de masse.

3.2.2 Advection de vélocité

Le champ de vélocité détermine les mouvements du fluide. L'étape d'advection applique le champ de vélocité au fluide pour calculer son évolution. Il faut alors appliquer la vélocité au champ de densité et aussi au champ de vélocité pour le mettre à jour. La fonction `advect` regarde la vélocité dans chaque cellule et détermine le bon changement des valeurs de la cellule pour cette vélocité.

3.2.3 Diffusion et advection de densité

Après avoir déterminer l'évolution du champ de vélocité, il faut faire évoluer le champ de densité. Les mêmes fonctions `diffuse` et `advect` détermine la diffusion et le mouvement des valeurs de densité dans notre fluide.

3.3 Extension 3D

La simulation de Stam est prasentée comme une simulation 2D, mais toutes ses techniques s'appliquent bien à une simulation 3D. Il faut ajouter une troisième dimension et changer les boucles et le stockage des données dans la simulation.

La transformation en 3D rend les résultats de la simulation difficiles à afficher. Une simulation 2D produit des données qui se transforment facilement en pixels et que la simulation peut afficher directement sur l'écran. Une simulation 3D produit des données volumiques qui ne sont pas faciles à afficher par l'utilisateur. Pour afficher les résultats de cette simulation il faut utiliser le rendu volumique.

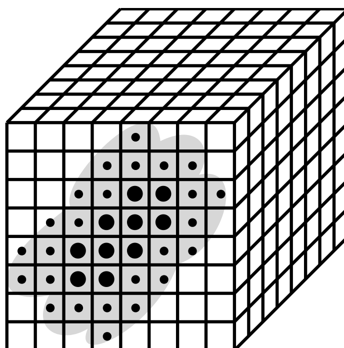


Figure 3: L'organisation des données volumiques d'un fluide

4 Rendu volumique

Pour représenter un fluide, généralement, l'espace (2D ou 3D) est discrétisé en autant de cellules que nécessaire pour une bonne précision de ce fluide. À chaque cellule sont associées des valeurs qui représentent ce fluide (la densité ou la vitesse essentiellement). Plus fine est la discrétisation plus fine est la représentation de ce fluide.

Une simulation de fluide produit des données volumiques. Ces données sont organisées comme les données d'une image 2D, mais l'idée est généralisée en trois dimensions. Nous avons choisi pour notre simulation de fluide 3D de partir d'un cube divisé en cellules que nous appellerons voxels. Pour une simulation de fluide, la valeur d'une cellule serait la densité du fluide à cet endroit dans l'espace (Voir figure 3). Cette représentation pourrait également s'appliquer aux données médicales, où à chaque cellule serait associée une valeur comme la densité ou le type du tissu, la quantité de sang, etc.

Bien que les données volumiques soient similaires aux données 2D habituelles, elles sont beaucoup plus difficiles à afficher car elles sont plus lourdes et moins convenables au matériel. La grande partie des techniques de rendu 3D et surtout les cartes graphiques 3D se concentrent sur le rendu des polygones 2D dans l'espace 3D. Les données volumiques 3D ne sont pas du tout pareilles aux polygones 2D, sauf si on traduit chaque voxel directement en polygones, ce qui génère une quantité énorme de polygones. Alors on est obligé d'utiliser la carte graphique pour rendre des données qui ne sont pas convenables aux capacités de la carte graphique, ou de ne pas utiliser la carte graphique et d'utiliser un algorithme plus pertinent directement sur le processeur.

4.1 Techniques

On a exploré les techniques actuelles en regardant leurs pertinences pour afficher les données dynamiques d'une simulation de fluide, et leurs possibilités de parallélisation.

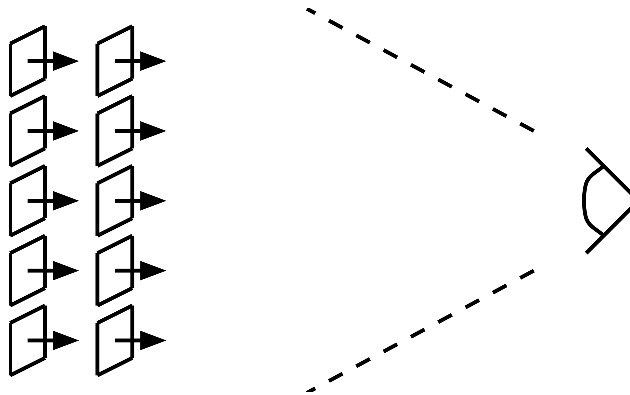


Figure 4: Le rendu volumique avec voxels dessinés directement sur l'écran

4.1.1 Voxels

La technique la plus simple est de dessiner les voxels directement sur l'écran avec un simple carré 2D. Pour chaque voxel dans le cube volumique, on positionne un carré qui est face à la caméra et avec la couleur du voxel.

Souvent on dessine des données creuses où la plupart des voxels sont vides. Alors, on peut optimiser cet algorithme et ne pas dessiner le carré qui correspond à un voxel si sa valeur est inférieure à un seuil donné.

Cette technique est très facile à implanter et assez rapide pour les petites données ou les données très creuses, mais elle peut devenir très lente. Sur les cartes graphiques typiques, dessiner un polygone est une opération relativement lente par rapport à dessiner un seul pixel texturé sur l'écran. Cette technique dessine beaucoup de polygones par rapport au nombre de pixels dans le résultat final. Par exemple, pour des données de taille 256x256x256, il y a presque 17 millione de polygones à rendre sur l'écran.

Paralléliser cette technique, pour utiliser une machine multiprocesseur ou une grappe de PC, est difficile. Il faut que chaque noeud ait une carte graphique puissante, même les noeuds qui n'affichent rien à l'utilisateur. Les noeuds doivent lire le buffer sur la carte graphique pour transmettre les résultats aux noeuds qui les affichent à l'utilisateur, mais l'opération de lecture des résultats n'est pas optimisée sur les cartes graphiques modernes. Alors cette technique ne convient pas à grande échelle.

4.1.2 Tranches

OpenGL et un grand nombre de cartes graphiques prennent en charge les textures 3D. Une texture 3D s'utilise pour texturer les polygones 2D exactement comme une texture 2D classique, et il n'est pas possible d'afficher la totalité de la texture en une seule opération.

Pourtant, s'il n'est pas possible d'afficher une texture 3D directement sur l'écran, il est possible d'utiliser ces textures pour afficher ses données volumiques. On calcule une suite de tranches dans le cube volumique qui sont toujours face à la caméra. Ces tranches sont texturées avec la texture 3D. Quand on dessine toutes les tranches, le résultat est une représentation du cube volumique sur l'écran. En variant le nombre de tranches dessinées, on peut très

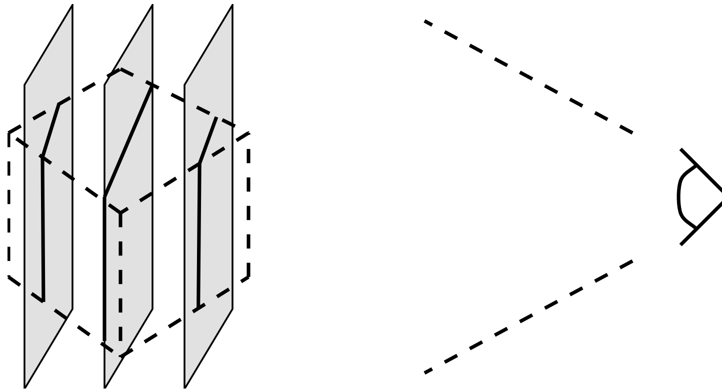


Figure 5: Le rendu volumique avec tranches dans une texture 3D

facilement changer le niveau de détail pour avoir des images plus détaillées ou plus lisses.

Cette technique utilise bien les capacités du matériel et elle peut être beaucoup plus rapide et moins coûteuse en calculs que la technique de voxels. Mais les capacités du matériel limitent les possibilités de cet algorithme. En effet, elle nécessite le chargement de toutes les données volumiques dans la mémoire de la carte graphique qui est généralement beaucoup plus petite que la mémoire principale de l'ordinateur.

Pour les données dynamiques, cela veut dire que chaque fois qu'on a de nouvelles données, il faut les charger dans la carte graphique. Comme la bande passante de la carte graphique est relativement faible, ce goulot d'étranglement peut poser problème. De plus, souvent les données volumiques ne sont pas dans un format qui est directement pris en charge par la carte graphique, ce qui nécessite une conversion coûteuse à chaque fois.

Il n'est pas possible de paralléliser cet algorithme, pour les mêmes raisons que pour l'algorithme de voxels. Il faudrait une bonne carte graphique pour chaque noeud, et ensuite reprendre les résultats sur la carte graphique qui est une opération lente.

4.1.3 Lancer de rayons

La physique de la lumière suggère qu'on utilise des rayons de lumière pour produire notre image. Dans le monde réel, un rayon passe dans un objet et puis il entre dans l'oeil. Dans notre monde virtuel, la réalité est reproduite à partir du même principe, par un rayon qui parcourt le même chemin dans l'autre sens. Ainsi à partir d'un écran virtuel devant notre cube volumique, des rayons sont lancés de la caméra vers cet écran. Le rayon traverse le cube, et un ensemble d'échantillons comme illustré par la figure 6 permet de calculer la couleur finale du point sur l'écran virtuel.

Par exemple, dans notre cas, les valeurs dans les données volumiques représentent une densité, et on peut tout simplement cumuler les valeurs de la manière suivante pour déterminer la transparence du cube :

$$pixel = transparence \times \sum_{rayon} voxel$$

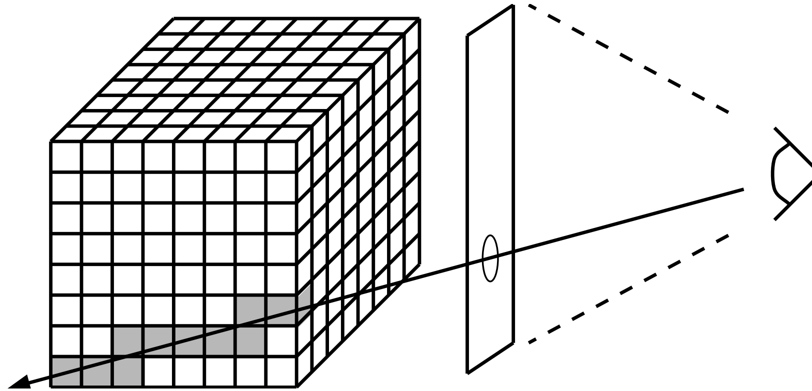


Figure 6: Le chemin d'un rayon lancé dans le cube volumique. Les voxels gris sont les échantillons.

Le nombre d'échantillons est un paramètre de l'algorithme de lancer de rayons. Dans la réalité la couleur d'un rayon dans une matière transparente est proportionnelle à l'épaisseur de la matière et la transparence de la matière. Alors la couleur devient plus forte quand le rayon est plus long. Donc le rayon prend un nombre d'échantillons proportionnel à la longueur du rayon.

$$\acute{e}chantillons = k_1 \times dim \times longeur(rayon)$$

Il faut aussi décider le nombre de rayons à lancer dans le cube, c'est à dire la taille de l'écran virtuel. L'écran virtuel est un carré de taille $dim_{\acute{e}cran}$. Pour notre implantation de lancer de rayons, on a décidé de faire un calcul proportionnel à la taille du cube volumique :

$$dim_{\acute{e}cran} = k_2 \times dim_{cube}$$

Quand le cube est directement face à la caméra avec $k_2 = 1$, il y a exactement un rayon par voxel dans le plan qui est plus proche de la caméra. Le cube est exactement conforme à l'écran virtuel. Chaque rayon lancé de la caméra à l'écran virtuel a un point d'intersection avec le cube.

Par contre, si le cube tourne et si son bord ou son coin est face à la caméra, il y a beaucoup de rayons qui ne coupe pas le cube, car le cube n'est pas conforme à l'écran virtuel (Voir Figure 7). Il y a le même nombre de rayons lancé, mais moins de rayons qui passent dans le cube. Alors le rendu devient moins coûteux car il y a moins de rayons à tracer, mais la qualité du rendu diminue.

La valeur de k_1 n'est pas fixée. Quand k_1 est petit, chaque rayon prend moins d'échantillons qui donne un rendu plus rapide mais avec moins de qualité. Quand k_1 est grand, chaque rayon prend plus d'échantillons qui donne une qualité supérieure mais le rendu devient plus coûteux. Alors en variant ce paramètre, il est possible d'échanger entre le coût du rendu et la qualité du résultat, que nous appellerons le niveau de détail (LOD). Ce paramètre change le nombre d'échantillons qui sont pris sur la profondeur du cube, alors nous appellerons ce paramètre $LOD_{profondeur}$.

Avec la même idée, la valeur de k_2 est aussi un paramètre de LOD. Quand k_2 est petit, le nombre de rayons lancés à l'écran diminue, et le rendu devient

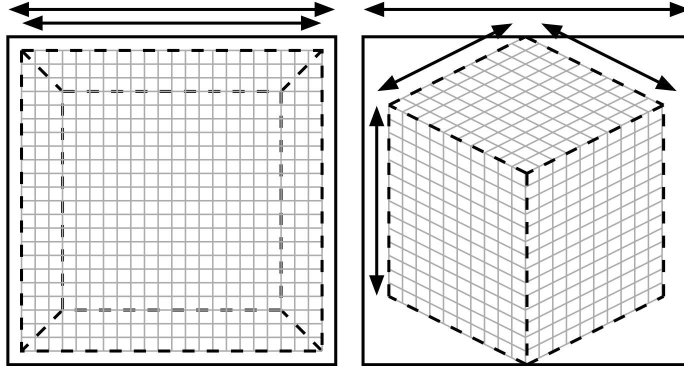


Figure 7: À gauche : le cube est face à la caméra. Le nombre de rayons égale le nombre de voxels dans le plan. À droite : le cube tourne. Le nombre de rayons qui coupent le cube diminue fortement, et on commence à passer des voxels.

moins coûteux. Quand k_2 est grand, il y a plus de rayons lancés et le rendu devient plus coûteux, mais la qualité de l'image final augmente. Ce paramètre change le nombre de rayons sur le plan de l'écran virtuel, alors nous appellerons ce paramètre LOD_{planar}

Pour avoir une qualité d'image et une vitesse constante, il faut varier le paramètre de LOD_{planar} en fonction de l'orientation du cube volumique. Quand le cube n'est pas face à la caméra, il faut lancer plus de rayons à l'écran virtuel pour que le même nombre de rayons coupe le cube volumique. Une possibilité serait d'analyser la relation entre l'orientation du cube et le proportion de rayons lancés qui coupe le cube. Cette analyse donnerait le LOD_{planar} pour une orientation du cube quelconque qui garde la qualité de l'image constante. Une autre possibilité serait de donner une vitesse désirée et créer un algorithme qui fait varier le LOD_{planar} en fonction de cette vitesse et la vitesse réalisée. Si la vitesse est inférieure au but, l'algorithme diminue le LOD_{planar} pour augmenter la vitesse du rendu, et si la vitesse est supérieure au but, l'algorithme augmente le LOD_{planar} pour donner une meilleur qualité d'image. Notre implantation garde un LOD_{planar} constant.

5 Optimisation de fluide

La simulation de fluide 3D proposée par Stam [7] est très coûteuse en calcul. Si N est la taille du cube, la quantité des données traitées est proportionnelle à N^3. La simulation demande des dizaines de parcours dans les données pour compléter une étape de simulations.

On souhaite toujours traiter plus de données en moins de temps. La simulation de base fait beaucoup de conversions entre les données flottantes et les données entières, et son accès mémoire est assez aléatoire. En optimisant ces deux faiblesses, la simulation gagne la capacité de traiter beaucoup plus de données.

5.1 Conversions int-float

Sur une grande partie des processeurs modernes, la conversion entre les données flottantes et les données entières est très lente. Par exemple, sur un processeur d'Intel or AMD de type x86, une conversion d'un *float* à un *int* nécessite de vider le pipeline du processeur, ce qui perturbe fortement les opérations du processeur. [16] Sur un PowerPC G4 de Motorola à 800MHz, une conversion d'un *int* de 16 bits prend plus de 43 nanosecondes, ce qui fait 35 cycles d'exécution [17]. Pour comparaison, la plupart des instructions dans l'unité flottant de ce même processeur s'exécutent sur un cycle dans le meilleur des cas, ou sur 5 cycles pour le pire cas. [18]

Souvent dans le code de la simulation il y a des variables de type `int` qui ne changent pas, ou qui changent d'une façon évidente, mais qui sont souvent utilisées dans les calculs avec des valeurs de type `float`. Ceci demande une conversion à chaque fois que la variable est utilisée, et cette conversion est redondante.

Par exemple, la variable `N` dans la fonction `advect()` est souvent utilisée dans les expressions avec des valeurs flottantes. En ajoutant une variable `Nfloat` et en faisant la conversion une seule fois avant la boucle, on économise beaucoup de conversions. Au lieu de convertir la valeur de `N` plusieurs fois pendant chaque exécution de la boucle, la valeur est convertie une fois au début.

Dans la même fonction, la variable `i` est dans le même cas, mais sa valeur change souvent. Mais ce changement est simple, alors il est possible de créer une variable `ifloat`. Au début de la boucle, `ifloat` commence avec la même valeur que `i`, et à chaque étape elle est incrémentée en même temps que `i`. Alors avec ce changement, il n'y a plus de conversion de la variable `i`.

Dans le même ordre d'idées, on préfère les multiplications que les divisions. Encore sur le PowerPC G4, une multiplication est dans la catégorie des instructions qui prennent entre 1 et 5 cycles, mais la division peut prendre plus que 14 cycles.

La fonction `lin_solve()` fait une division par la variable `c` à chaque itération de sa boucle. Comme `c` ne change jamais pendant cette boucle, on peut alors calculer sa réciproque, et multiplier par cette réciproque dans la boucle. Alors, on met au début de la fonction :

```
float cRecip = 1.0 / c;
```

Et puis dans la boucle, on remplace `... / c` par `... * cRecip`.

5.2 Les accès mémoire

L'ordre de chargement des données de la mémoire est très important pour optimiser le temps de calculs. Les systèmes de mémoire des ordinateurs modernes sont très optimisés pour certains ordres spécifiques.

Une grande partie des accès mémoire dans un programme est aux adresses proches ou adresses linéaires. C'est à dire qu'un programme accède à des données soit à des adresses aléatoires mais proches soit à des adresses X, $X+1$, $X+2$, $X+3$, $X+4$, $\ldots$. Ainsi, le matériel est très optimisé pour ces deux cas.

Un système de mémoire moderne possède plusieurs niveaux de *caches*. Un cache est un morceau de mémoire proche du CPU qui est petit mais très rapide. Quand une adresse de mémoire est chargée, le contenu de cette adresse et ses

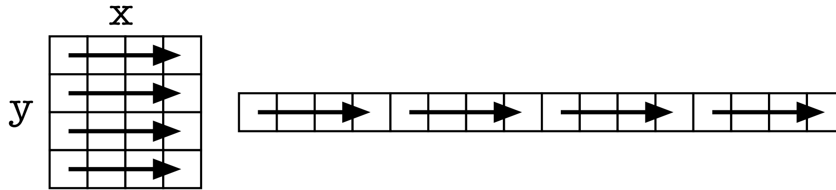


Figure 8: Quand x est à l'intérieur de la boucle, on charge le contenu de la mémoire dans l'ordre respectant le mode de stockage.

voisins sont chargés dans le cache. Donc, quand le processeur veut charger un voisin de cette adresse, il est déjà présent dans le cache, et son accès est beaucoup plus rapide.

Souvent les processeurs modernes ont aussi un module de *prefetching* (pré-chercher). Ce module, à partir de l'ordre d'accès à la mémoire du processeur, lorsque cet accès est séquentiel, est capable d'effectuer du préchargement. Alors, pendant que le processeur est en train de traiter les données à l'adresse $X + n$, le module de prefetching charge en même temps les données à l'adresse $X + n + 1$. Ce recouvrement chargement/calcul permet d'augmenter notablement les performances du processeur.

Pour une architecture qui utilise une hiérarchie de caches et un module de prefetching, architecture classique pour les ordinateurs modernes et hautes performances, il est très important d'organiser les accès mémoire dans un ordre qui convient au matériel. Et ceci afin d'exploiter au mieux les ressources CPU.

5.2.1 Ordre de boucles

Dans le cas d'une simulation de fluide ou de beaucoup d'autres algorithmes, on travaille avec des données multidimensionnelles, mais l'organisation des données en mémoire, pour la plupart des ordinateurs, est monodimensionnelle. Pour les données à deux dimensions, par exemple une matrice, les données sont traditionnellement rangées par ligne (on peut parfois trouver une organisation selon les colonnes comme en Fortran par exemple). Avec cet ordre de données, dans une grille de X par Y, X le nombre de colonnes et Y le nombre de lignes, l'indice de la cellule (x, y) est alors $y \times X + x$. Pour trois dimensions on garde cet ordre pour chaque plan, et les plans sont stockés consécutivement.

L'ordre d'accès mémoire dépend de quelle variable est incrémentée le plus souvent. Pour des données 2D, il y a deux façons pour itérer toutes les données en deux boucles. Si x est incrémenté par la boucle à l'extérieur et y est incrémenté par la boucle à l'intérieur, y changera le plus souvent. Les boucles parcouront les cellules $(0, 0)$, $(0, 1)$, $(0, 2)$, $(0, 3)$, ..., $(1, 0)$, $(1, 1)$, $(1, 2)$, $(1, 3)$, ... Ces cellules correspondent aux adresses 0, X, $2X$, $3X$, ..., 1, $X+1$, $2X+1$, $3X + 1$, ... (Voir Figure 9). Inverser les boucles et mettre y à l'intérieur touche les adresses 0, 1, 2, 3, ..., X, $X + 1$, $X + 2$, ..., une séquence qui est beaucoup plus conforme au matériel (Voir Figure 8).

Pour les données à trois dimensions, on doit faire la même chose. On préfère itérer selon x puis y, et finalement selon z.

Alors, il est très important de mettre les boucles dans le bon ordre.

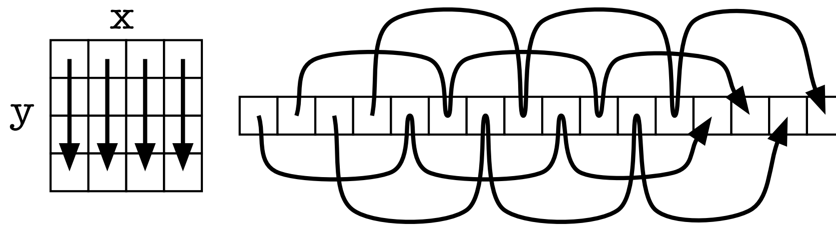


Figure 9: Quand `x` est à l'extérieur de la boucle, on charge le contenu de la mémoire d'une façon qui ne convient pas au système de mémoire.

```
for(x = 0; x < N; x++) {
    for(y = 0; y < N; y++) {
        for(z = 0; z < N; z++) {
```

Il faut changer l'ordre pour mettre x dans la boucle intérieur :

```
for(z = 0; z < N; z++) {
    for(y = 0; y < N; y++) {
        for(x = 0; x < N; x++) {
```

Cet ordre chargera les données d'une façon linéaire dans la mémoire ce qui optimisera les accès.

5.3 Résultats

On a testé ces optimisations sur un Apple PowerBook G4 avec un processeur à 1.5GHz et un cube de fluide à 64x64x64. Sans optimisations, la vitesse de la simulation est de 0,22 étapes de simulation par seconde. Avec toutes les optimisations proposées, la vitesse est de 1,52 étapes par seconde, soit une amélioration de 591%.

5.4 Autres possibilités

Si une donnée doit être utilisée plusieurs fois dans un programme, il vaut mieux utiliser cette donnée successivement avec le moins de temps possible entre chaque utilisation pour avoir plus de chance de la trouver toujours dans le cache. Par exemple, la boucle notée `2` dans le code suivant serait beaucoup plus rapide sur une grande quantité de données que la boucle notée `1` :

```
// 1
for(n = 0; n < 3; n++)
    for(i = 0; i < size; i++)
        A[i] = f(B[i]);

// 2
for(i = 0; i < size; i++)
    for(n = 0; n < 3; n++)
        A[i] = f(B[i]);
```

On peut imaginer le calcul dans un tableau où chaque calcul dépend de la valeur précédente dans le tableau, et on veut faire ce calcul trois fois. Alors, l'algorithme le plus simple serait de répéter cette boucle trois fois :

```
for(n = 0; n < 3; n++)
    for(i = 1; i < size; i++)
        A[i] = f(A[i-1], A[i], A[i+1]);
```

Cet algorithme parcourt le tableau linéairement trois fois. Si le tableau est plus grand que le cache, l'ordinateur doit charger le tableau de la mémoire trois fois.

On peut changer notre algorithme pour faire les calculs d'une manière qui optimise les accès à la mémoire. On va calculer seulement ce dont on a besoin pour faire le calcul complet pour chaque adresse dans le tableau :

```
A[1] = f(A[0], A[1], A[2]); // 1

A[2] = f(A[1], A[2], A[3]); // 2
A[1] = f(A[0], A[1], A[2]);

for(i = 1; i < size - 2; i++) // 3
    for(n = 2; n >= 0; n--)
        A[i+n] = f(A[i+n-1], A[i+n], A[i+n+1]);

A[size-2] = f(A[size-3], A[size-2], A[size-1]); // 4
A[size-1] = f(A[size-2], A[size-1], A[size]);

A[size-1] = f(A[size-2], A[size-1], A[size]); // 5
```

Cet algorithme devient plus compliqué à cause du début et de la fin du tableau. Pour démarrer, il faut d'abord faire le premier calcul de la première adresse dans le tableau (1). Après, il faut calculer les valeurs intermédiaires des deux premières adresses (2). Quand la boucle termine, il faut faire la même chose pour les deux dernières adresses (4 et 5).

La boucle même est la partie intéressante de cet exemple (3). On fait le calcul à l'envers pour avoir assez de valeurs intermédiaires pour calculer la valeur finale pour chaque adresse. On accède la mémoire dans cet ordre : $X + 2$, $X + 1$, X,

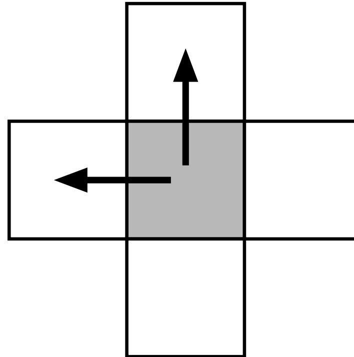


Figure 10: Les dépendances d'un solveur Gauss-Seidel. Le calcul de chaque cellule dépend du calcul des ses voisins dans un seul sens.

$X + 3$, $X + 2$, $X + 1$, $X + 4$, ... Les adresses sont proches et les accès suivants viennent rapidement, alors il est fortement probable que chaque donnée restera dans le cache jusqu'à la fin des trois calculs sur cette donnée.

La fonction `lin_solve` dans la simulation de fluide est similaire, sauf en trois dimensions. Pour le solveur Gauss-Seidel, le calcul de chaque cellule dans le fluide dépend du calcul des trois voisins (Voir figure 10).

Pour deux ou trois dimensions, on peut utiliser la même technique. Pour démarrer la boucle on fait le calcul dans un triangle ou une pyramide, ensuite on calcule les valeurs pour toute une ligne, et puis on peut continuer à faire le calcul sur un nombre de lignes égal au nombre d'itérations qu'on fait (Voir Figure 11).

Pour un solveur 3D, il faut faire le calcul sur plusieurs lignes en même temps. Si i est le nombre d'itérations, il faut faire le calcul sur $1 + 2 + \cdots + i - 1 + i = i \times (i + 1)/2$ lignes. Pour $i = 4$ et un cube de 64 cellules contenant chacun une valeur flottante de 4 octets, la quantité des données est :

$$\frac{4 \times 5}{2} \times 64 \times 4o = 2560o$$

Cette quantité de données est inférieure à la capacité du cache de niveau 1 d'un processeur typique, qui est souvent de 32ko. Par contre, pour faire le calcul sur tout le cube, on doit toucher 1Mo de données à chaque fois, ce qui est beaucoup plus grand pour le cache de niveau 1, et éventuellement trop grand pour le cache de niveau 2.

6 Optimisation de lancer de rayons

L'algorithme de base de lancer de rayons est simple mais coûteux. On travaille naturellement avec les coordonnées flottantes, mais toutes ces valeurs doivent être converties en entiers pour calculer l'adresse du voxel dans les données volumiques. L'algorithme nécessite aussi beaucoup d'accès mémoire aléatoires, ce qui peut limiter les performances.

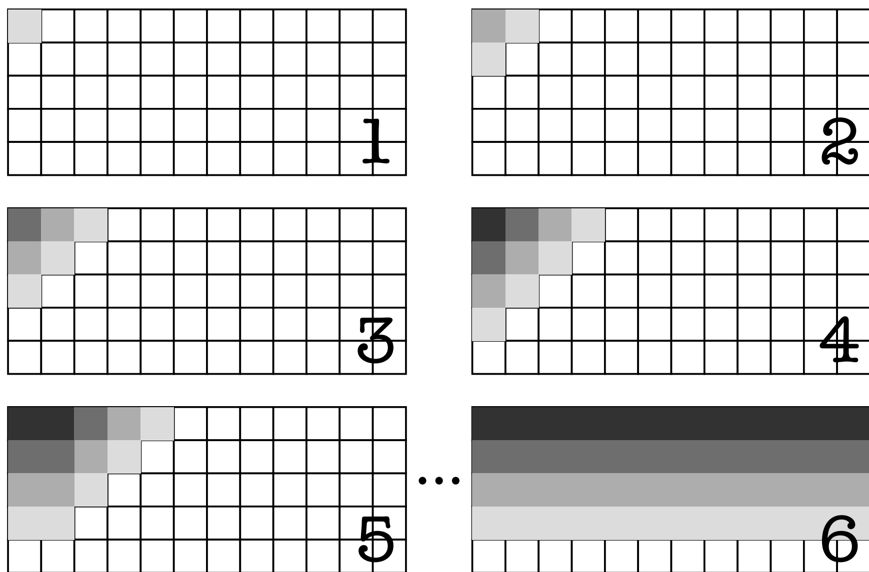


Figure 11: Schéma d'un solveur Gauss-Seidel utilisant plus fortement les caches avec quatre itérations. 1) On fait la première étape de calcul dans la cellule au coin. 2) On fait la première étape de calcul dans les voisins, et puis la deuxième dans le coin. 3) On fait la première étape dans les voisins des voisins, qui nous permet de faire la deuxième étape dans les voisins, et finalement la troisième étape au coin. 4) La région de calculs est maintenant dans un rayon de 4 cellules, ce qui permet de faire la quatrième étape au coin. 5) Ayant fait quatre étapes au coin, on continue à faire le calcul dans les quatre premières lignes. 6) On termine le calcul pour la première ligne, et on commence sur les lignes 2 à 5.

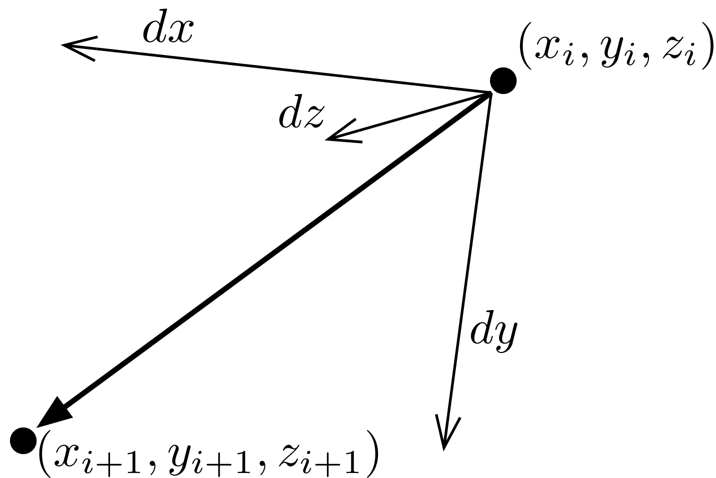


Figure 12: Une étape de la boucle intérieure de lancer de rayons.

6.1 Calcul des adresses

On a vu dans la partie 5.1 qu'il faut minimiser les conversions entre les valeurs `int` et `float` pour avoir la plus grande vitesse. La boucle principale de lancer de rayons fait parcourir un rayon dans les données volumiques. À partir de la taille du cube, les points d'entrée et de sortie, et les paramètres de LOD, on calcule un dx, dy, et dz, et un nombre d'échantillons à prendre. On commence avec le point d'entrée et on ajoute les deltas à chaque étape jusqu'à la réalisation de toutes les étapes ou jusqu'à la couleur maximale. Alors toutes ces coordonnées ne peuvent pas être des entiers, car on a souvent des deltas qui sont entre 0 et 1. On doit alors faire trois conversions `float-int` à chaque étape de cette boucle.

Hormis ces conversions, il faut ajouter le calcul de l'adresse du voxel courant à partir de ses coordonnées.

$$adresse = x_i + y_i \times size_x + z_i \times size_x \times size_y$$

Les multiplications dans ce calcul sont coûteuses. Il faut aussi borner les coordonnées pour éviter des problèmes d'arrondis, ce qui demande des comparaisons coûteuses.

Lorsque le cube est de taille $2^n \times 2^n \times 2^n$ il est plus facile de convertir une position dans le cube en une adresse en mémoire. Chaque dimension utilise un nombre entier de bits dans l'adresse. Par exemple, pour un cube de $32 \times 32 \times 32$, chaque coordonnée prend exactement 5 bits dans l'adresse du voxel.

$$\begin{aligned} adresse &= x + y \times 32 + z \times 32 \times 32 \\ &= x + y \times 2^5 + z \times 2^{10} \\ &= x|(y << 5)|(z << 10) \end{aligned}$$

Ce calcul n'utilise que les opérations bitwise de C, qui sont normalement très rapides (Voir Figure 13 pour une représentation graphique de l'adresse pour le cube de 32x32x32).

Pour éviter les conversions `float-int`, on utilise une représentation en virgule-fixe. Notre implantation utilise une mantisse de 20 bits et alors un

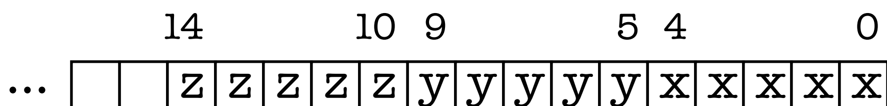


Figure 13: L'organisation des coordonnées dans l'adresse d'une valeur dans un cube de 32x32x32.

exposant fixé à 2^{-20}. Cette représentation permet une conversion en un entier pour les opérations peu coûteuses. [19]

On peut combiner ces deux opérations. Pour chaque coordonnée on a successivement un shift à droite pour changer en entier, et un shift à gauche pour le calcul de l'adresse. Ces deux opérations peuvent être simplifiées grâce à un masque de la manière suivante :

```
// 1
y_int = y >> 20
y_int = y_int % ysize
adresse = adresse | (y << log2(xsize))

// 2
xshift = log2(xsize)
yimask = (y - 1) << xshift
yishift = 20 - xshift

// 3
adresse = adresse | ((y >> yishift) & yimask)
```

Le calcul dans `1` est remplacé par le calcul dans `2`, qui est fait une seule fois, et le calcul dans `3` qui est fait à chaque calcul d'adresse.

Le nombre d'opérations est le même, mais cette version ajoute une fonctionnalité, elle borne les valeurs qui deviennent trop grandes. Alors, en deux opérations par coordonnée et deux opérations en plus pour combiner les résultats, on peut faire tout le calcul de l'adresse d'une valeur :

```
offset = ((sx >> xishift) & ximask) |
         ((sy >> yishift) & yimask) |
         ((sz >> zishift) & zimask);
```

6.2 Accès mémoire

Dans la partie 5.2, nous avons vu tout l'intérêt d'organiser correctement les accès mémoire. Malheureusement pour notre lancer de rayon, le parcours d'un rayon dans le cube touchant des valeurs d'une manière assez aléatoire il n'est pas possible d'optimiser les accès mémoire.

```
while (x, y, z) in cube
    calculer l'adresse de (x, y, z)
    charger valeur à l'adresse // 1
    totale = totale + valeur // 2
    if totale > max
        stop
    calculer les prochains (x, y, z)
```

Chaque étape dépend du résultat de l'étape précédente. La ligne la plus coûteuse est la ligne `1` à cause de l'accès mémoire. La seule ligne qui dépend directement de son résultat est la ligne `2`. Une boucle qui met toutes les autres lignes entre elles peut faire son calcul plus rapidement. Le processeur peut exécuter les autres lignes pendant qu'il attend l'accès mémoire.

```
calculer l'adresse de (x, y, z)
while (x, y, z) in cube
    charger valeur à l'adresse // 1
    calculer les prochains (x, y, z)
    if totale > max
        stop
    calculer l'adresse de (x, y, z)
    totale = totale + valeur
```

Cette boucle fait le même calcul mais elle est beaucoup moins coûteuse.

Dans les tests sur nos données, avec toutes les optimisations détaillées ici, on a amélioré la vitesse de l'algorithme de lancer de rayons d'un facteur de l'ordre de 3.

7 Parallélisme

Pour cela nous avons choisi une approche modulaire avec

- un module simulation de fluide qui utilise le solveur de fluide 3D et envoie les résultats au module de rendu
- un module de rendu qui prend les données de la simulation de fluide et utilise un algorithme de lancer de rayons pour générer des pixels.
- un module d'affichage qui présente une interface à l'utilisateur et affiche les pixels du module de rendu.

7.1 Extension de fluide en parallèle

Pour paralléliser la simulation de fluide, on coupe le cube en plusieurs morceaux et on fait le calcul de chaque morceau sur son propre noeud.

Pour minimiser les communications entre les noeuds, il faut que chaque morceau ait le moins de surface possible. L'optimale est alors de couper le cube en sous cubes qui sont répartis sur les noeuds de simulation. Cette distribution

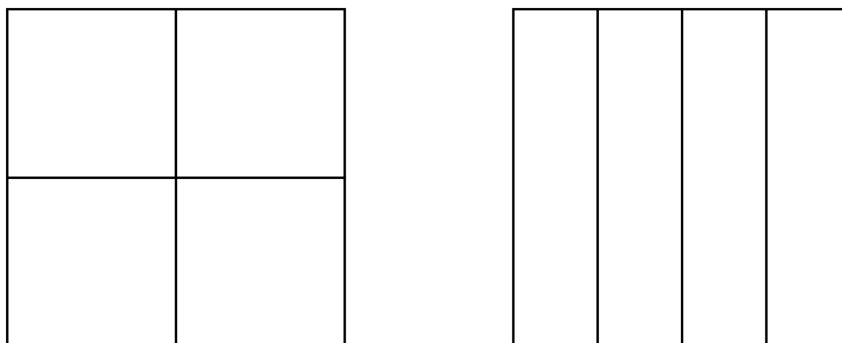


Figure 14: À gauche, le cube divisé en sous cubes. À droite, le cube divisé en tranches.

pose un problème sur le nombre de noeuds de simulation. Il y a toujours un nombre cube de sous cubes, alors cette distribution ne marche bien qu'avec un nombre cube de noeuds. Sur une grappe de PC, cela nous limite à 1, 8, et peut-être 27 noeuds de simulation. Il est possible de distribuer plusieurs sous cubes sur chaque noeud de simulation, mais il y a toujours des problèmes de répartition de charges. En plus, les communications de chaque noeud sont difficiles à gérer car il y a six voisins pour chaque noeud.

Une autre possibilité est de couper le cube en tranches et distribuer chaque tranche sur un noeud de simulation. Cette distribution n'est pas optimale au niveau de communications mais elle marche avec un nombre de noeuds quelconque et les communications sont beaucoup moins difficiles à gérer avec seulement deux voisins pour chaque noeud.

Ces deux approches sont illustrés par la figure 14.

On a décidé alors de couper le cube en tranches sur l'axe Z. On a choisi l'axe Z à cause de l'organisation de nos données volumiques : remonter les données d'un cube est une simple concaténation des données de chaque tranche.

Pour la communication, on crée une couche de *ghosts* (fantômes). Les ghosts sont des cellules du voisin qui sont copiées sur un noeud. Ces copies doivent être mises à jour après chaque sous étape de calcul qui les modifie. Les deux noeuds au bord ont des ghosts sur seulement une face, et les autres noeuds ont des ghosts sur deux faces. (Voir Figure 15.)

Il est à noter qu'il y a 4 couches de ghosts pour la densité et les trois composantes de la vitesse.

On échange les ghosts après chaque itération de `lin_solve`, après chaque `advect`, et au milieu et après chaque `project`. En total, on échange les ghosts 38 fois par étape de simulation.

La taille de la couche de ghosts est déterminante pour l'efficacité de notre parallélisation. Dans l'étape de `advect`, une cellule avec une grande vitesse peut modifier une cellule non voisine. Une cellule avec une vitesse de x peut modifier une autre cellule de distance x. Mais il n'est pas possible de modifier les cellules de l'autre côté des ghosts. Si une simulation contient des vitesses qui sont supérieures à l'épaisseur de sa couche de ghosts, elle risque de ne pas être fidèle (Voir Figure 16). Normalement les vélocités sont inférieures à 1, alors une épaisseur d'une cellule est suffisante.

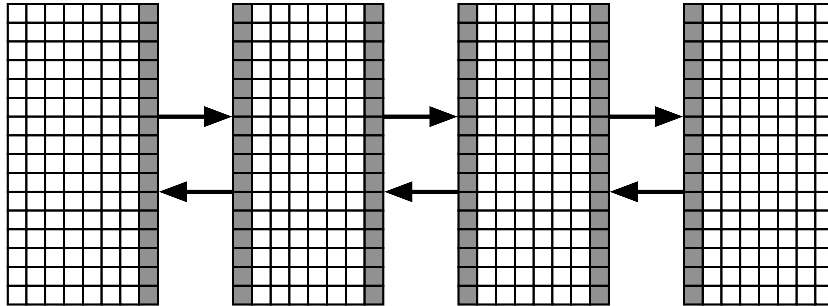


Figure 15: La simulation de fluide sur quatre noeuds. Les cellules en gris sont les ghosts qui sont communiqué après chaque sous-étape.

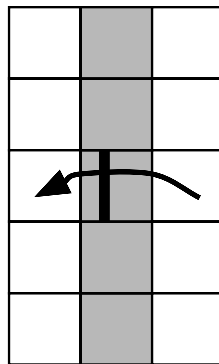


Figure 16: Une trop grande vitesse ne peut pas traverser la frontière entre deux noeuds.

Si N est la taille du cube, la couche de ghosts a alors N^2 cellules. Si chaque cellule contient une valeur de 4 octets, la quantité totale des données envoyées et reçues par un noeud interne est:

$$données = 2 \times 38N^2 \times 4o$$

Le facteur deux est dû au fait que chaque noeud interne à une couche de ghosts sur deux faces. Pour un cube de 32x32x32, il faut alors transférer 304ko par étape. Pour un cube de 64x64x64, il faut transférer plus de 1.18Mo, et pour un cube de 128x128x128, 4.75Mo. D'où l'importance des communications qui peuvent constituer un goulot d'étranglement pour notre simulation.

7.2 Extension de lancer de rayons en parallèle

On a vu dans la partie 4.1 que seule la technique de rendu volumique possédait les bonnes caractéristiques pour être parallélisé.

Nous avons conçu deux parallélisations qui peuvent être complémentaires, soit en divisant l'écran virtuel soit en partageant le cube de données.

Pour paralléliser l'algorithme de lancer de rayons, on a deux choix. Le premier choix est de diviser l'écran virtuel entre les noeuds en recopiant toutes les données volumiques sur chaque noeud. L'autre choix est de couper le cube en morceaux, et puis chaque noeud fait le rendu sur son morceau, sur la région de l'écran virtuel approprié. On pourrait aussi imaginer une combinaison des deux choix, où on divise le cube en morceaux qui sont données à un groupe de noeuds de rendu au lieu d'un seul noeud.

7.2.1 Répartition du cube de données

Si on divise le cube directement, on a un grand avantage au niveau des communications. Au lieu de communiquer tout le cube à tous les noeuds de rendu, on communique seulement une partie du cube à chaque noeud, ce qui diminue fortement la quantité de données communiquées. Par contre, la communication au noeud d'affichage est plus lourde, parce qu'il faut communiquer tous les écrans virtuels des noeuds de rendu. Le noeud d'affichage doit calculer un écran virtuel final à partir de tous les écrans intermédiaires. Il y a un problème de répartition de charges : il est possible qu'une région du cube demande plus de calcul pour l'algorithme de lancer de rayons qu'une autre région. Les régions avec plus de fluide prennent en général moins de calcul, parce qu'on atteint rapidement la limite de la couleur du rayon. Si le travail n'est pas équilibré, on peut perdre beaucoup de performance quand un noeud continue de travailler et que tous les autres sont inactifs. Cette répartition est illustrée par la figure 17.

7.2.2 Partage de l'écran virtuel

Si on divise l'écran virtuel, on a un schéma de communication beaucoup plus simple. On pourrait diviser l'écran en tranches ou carrés, mais on aurait le même problème de l'équilibre de travail. Pourtant, on a un grand avantage car il n'y a pas d'interaction entre les rayons. Alors, on peut diviser les rayons sur les noeuds d'une manière cyclique. C'est à dire, on met le rayon 1 sur le noeud 1, rayon 2 sur le noeud 2, et en général le rayon n est sur le noeud n mod k où k est le nombre de noeuds total. Cette division assure un travail équilibré parce

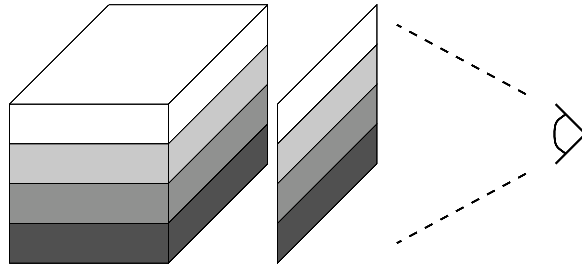


Figure 17: Parallélisation de lancer de rayon. On divise le cube en tranches, et chaque tranche est sur un processeur différent. Chaque niveau de gris est sur un processeur différent.

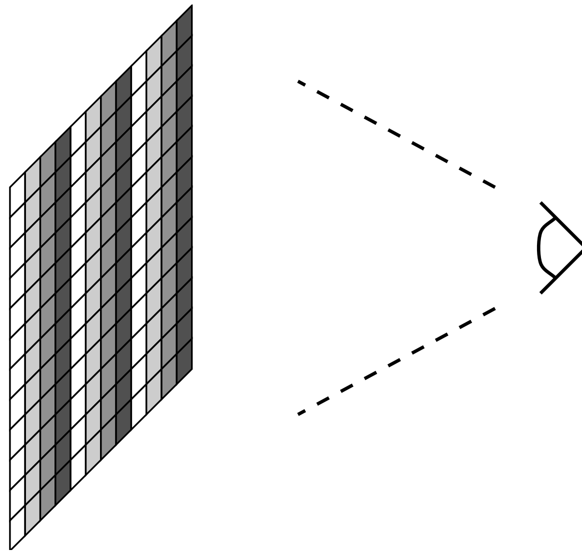


Figure 18: Parallélisation de lancer de rayon. On divise les pixels sur l'écran d'une manière cyclique. Chaque niveau de gris est sur un processeur différent.

que chaque noeud participe au calcul de chaque région. Cette répartition est illustrée par la figure 18.

À cause de la simplicité des communications et la répartition de charges, on a finalement implanté la division cyclique de l'écran virtuel pour notre application.

7.3 Réseau complet de l'application

On a trois types de modules qu'il faut déployer sur la grappe, les modules de fluide, de rendu, et d'affichage. Il faut trouver un protocole pour les communications, et créer un schéma pour le réseau entier.

7.3.1 Schéma de communication

On a vu dans la partie 7.1 qu'il faut que les modules de fluide communiquent entre eux. On a aussi vu dans la partie 7.2 que les modules de rendu n'ont pas

besoin de communications entre eux. Mais il faut que chaque noeud de fluide envoie ses données à tous les noeuds de rendu après chaque étape de simulation, et il faut que les noeuds de rendu communiquent avec le noeud d'affichage.

Chaque noeud de fluide doit communiquer ses ghosts à ses voisins, alors il y a une connexion aux voisins. Il doit aussi communiquer avec tous les noeuds de rendu, alors il y a une connexion de chaque noeud de fluide à chaque noeud de rendu.

Les noeuds de rendu ont besoin de deux ensembles de données. Ils ont besoin des données volumiques qui viennent de la simulation de fluide. La simulation est découplée du rendu, c'est à dire que la vitesse de simulation n'a pas de lien direct avec la vitesse de rendu. Le noeud de rendu peut recevoir plusieurs étapes de simulation pendant une étape de rendu, ou il peut faire le rendu plusieurs fois pendant une étape de simulation. On peut imaginer une simulation de fluide très lente : on voudrait toujours que le fluide sur l'écran tourne quand on bouge la souris, même que le fluide ne change que lentement. Alors on met un filtre "glouton" sur cette connexion. Ce filtre accepte toujours des nouvelles données du noeud de fluide, et il donne toujours les données les plus récentes au noeud de rendu.

Pour faire le rendu, il faut connaître la matrice actuelle qui décrit les paramètres de la caméra sur le noeud d'affichage pour pouvoir positionner la caméra et l'écran virtuel. On a alors une connexion du noeud d'affichage à tous les noeuds de rendu. Cette connexion n'a pas de filtre parce qu'il faut que les noeuds de rendu et le noeud d'affichage restent synchronisés. Les noeuds de rendu envoient ses données au noeud d'affichage, alors il y a aussi une connexion des noeuds de rendu au noeud d'affichage. Cette connexion n'a pas de filtre.

Le noeud d'affichage a deux connexions à chaque noeud de rendu, une pour envoyer la matrice et une pour recevoir les résultats de rendu.

On peut voir le schéma de communication pour un réseau avec quatre noeuds de fluide et deux noeuds de rendu sur la Figure 19.

Pour le reste du réseau, le module de fluide est tout simplement un module qui produit des données volumiques quelconques. Alors il est possible de remplacer ce module avec un autre module qui produit ses données d'une autre manière. Un tel module est un module de données statiques. Souvent on a des fichiers qui contiennent des données volumiques. Le module de données statiques charge le fichier et l'envoie directement aux noeuds de rendu. On a implanté un module de données statiques qui nous permet de tester la performance du rendu sans regarder la simulation de fluide, et de visualiser ces fichiers.

7.3.2 Protocoles

Les protocoles traditionnels pour les applications distribuées comme MPI marchent bien pour les applications homogènes, mais ils ne conviennent pas bien aux applications hétérogènes comme celle-ci. Il existe des bibliothèques spécialisées aux applications hétérogènes, mais on a rencontré des difficultés pour les intégrer dans cette application. Comme la communication dans cette application n'est pas trop compliquée, on a décidé de créer une bibliothèque pour cette application.

Cette bibliothèque est un petit emballage de TCP et sockets standard. Elle permet de se connecter à un hôte, d'écouter des connexions, et d'envoyer ou recevoir des données avec un appel d'une fonction par opération. Elle s'occupe

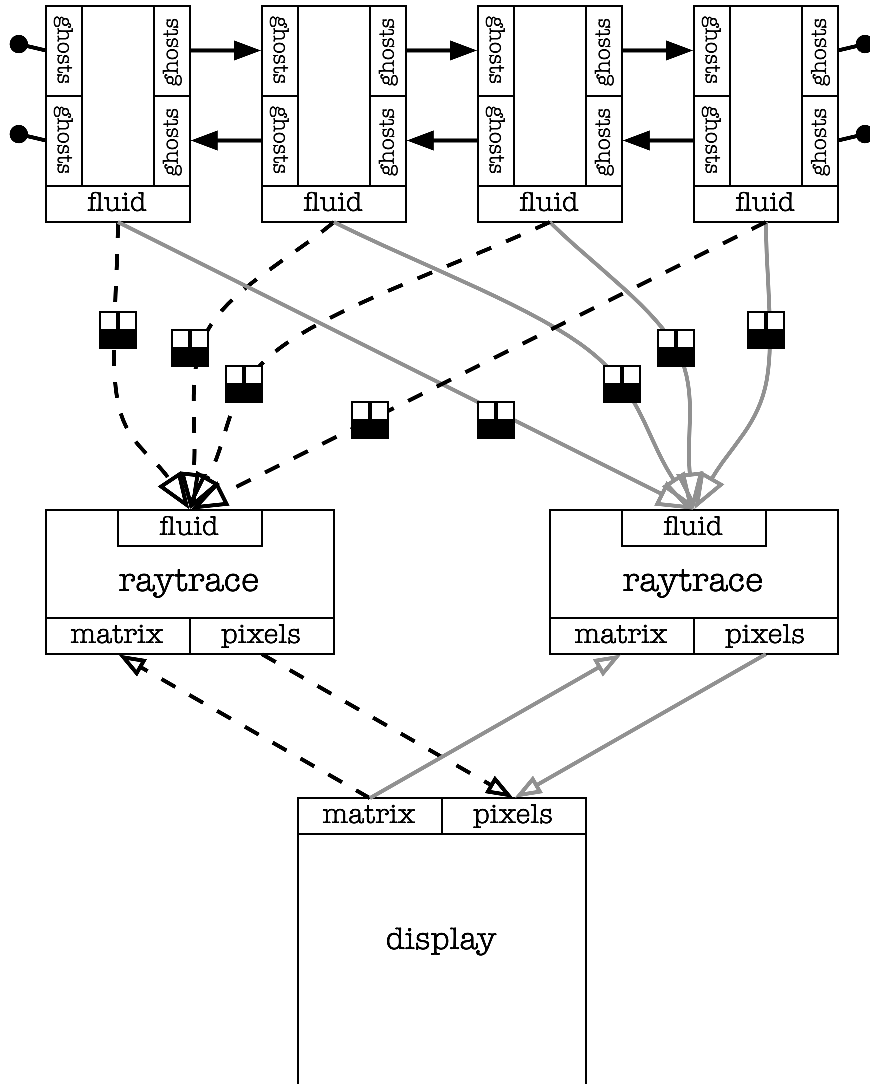


Figure 19: Schéma de communication pour un réseau avec quatre noeuds de fluide, deux noeuds de rendu, et un noeud d'affichage. Les boîtes sur les connexions entre les noeuds de fluide et les noeuds de rendu indique un filtre "glouton".

de la gestion des connexions, et permet d'envoyer ou recevoir des données sur plusieurs connexions en même temps.

On peut voir les modules de fluide comme un grand logiciel parallèle homogène. Alors on peut utiliser MPI pour les communications à l'intérieur de ce module. MPI est plus optimisé pour les transferts de données entre les noeuds homogènes, surtout pour deux noeuds sur le même ordinateur biprocesseur, alors MPI est plus efficace pour le module de fluide. Le module de fluide n'utilise pas beaucoup la fonctionnalité de MPI, alors on a gardé la possibilité d'utiliser notre bibliothèque même pour les communications internes.

La bibliothèque ne permet pas de construire le réseau de communication de l'application. Alors on a créé un script avec Python qui construit le réseau et lance tous les modules. Le script stocke une liste de tous les noeuds disponibles pour les modules de fluide et de rendu et construit le réseau à partir de cette liste et le nombre de chaque type de module à lancer.

8 Analyse

8.1 Analyse théorique de fluide

Pour l'analyse de notre simulateur, nous considérons que le programme est en deux étapes, une de calculs et une de communications. On a vu dans la partie 7.1 que la quantité de données transférées à chaque étape est $2 \times 38N^2 \times 4o$ pour tous les noeuds internes. Alors, si l est la latence du réseau et B la bande passante, on peut calculer le temps de communication de la manière suivante :

$$t_1 = 38l + \frac{2 \times 38N^2 \times 4o}{G}$$

Pour les machines biprocesseurs on peut enlever le facteur deux. Il est possible d'assurer que les deux noeuds de fluide sur le même ordinateur sont aussi des voisins dans le schéma de communications. Alors les communications entre ces deux noeuds ne passent pas par le réseau. Chaque noeud communique avec au plus un autre noeud sur le réseau.

Après cette étape, les noeuds de fluide envoient les données aux noeuds de rendu. Chaque noeud de fluide envoie ses données à tous les noeuds de rendu, et chaque noeud de rendu reçoit toutes les données volumiques du cube.

Si f est le nombre de noeuds de fluide, r le nombre de noeuds de rendu, et N la taille du cube, la quantité des données d_f envoyées par les noeuds de fluide est :

$$d_f = \frac{rN^3}{f} \times 4o$$

La quantité des données d_r reçues par les noeuds de rendu est :

$$d_r = N^3 \times 4o$$

La quantité des données d qui limite la communication est le maximum de ces deux valeurs. Alors, si on a plus de noeuds de rendu que de noeuds de fluide, $d = d_f$, sinon, $d = d_r$. Le calcul de fluide est beaucoup plus lourd, donc il est logique d'utiliser plus de noeuds de fluide et donc $d = d_r$. Le temps de cette étape de communication est alors :

$$t_2 = l + \frac{N^3}{G} \times 4o$$

Au total, le temps de communication est :

$$t = t_1 + t_2 = 39l + \frac{76N^2 + N^3}{G} \times 4o$$

8.2 Analyse théorique du rendu

Le module de rendu communique peu. Il reçoit bien les données volumiques, mais ces données sont gérées par le filtre "glouton" sur la connexion, et leur temps de communication est compris dans les communications du module de fluide. La seule communication qui compte pour ce module est la communication avec le noeud d'affichage. Pour chaque étape de rendu, on communique la matrice du noeud d'affichage. Suite au calcul du rendu tous les pixels résultants sont envoyés au noeud d'affichage. Alors on peut réunir les deux étapes de communication en comptant une latence deux fois plus grande.

La matrice a 16 éléments de 4 octets. Elle est envoyée à tous les noeuds de rendu par le noeud d'affichage, alors il faut compter la bande passante du noeud d'affichage. Chaque pixel est un niveau de gris d'un octet. Si N est la taille du cube, le nombre de pixels est $LOD_{planar} \times N^2$. Alors, la quantité de données communiquées est :

$$d = 64ro + LOD_{planar} \times N^2 o$$

Où r est le nombre de noeuds de rendu. Le temps de communication est alors :

$$t = 2l + \frac{64ro + LOD_{planar} \times N^2 o}{G}$$

9 Données de performance

9.1 Scalabilité

Les tests ont été réalisés sur deux grappes de PC, une grappe à North Dakota State University (NDSU) aux États-Unis, et la grappe du LIFO à l'Université d'Orléans.

La grappe à NDSU n'est pas très puissante, surtout au niveau du réseau, qui permet de voir les besoins du code. Il comprend 15 Power Mac G4s utilisant Mac OS X comme système d'exploitation à 533MHz avec 256MB de mémoire vive, et un réseau d'un 100Mbit switch.

La grappe au LIFO était en cours de maintenance et de mise à jour. Les tests ici se sont faits avec une grappe de 7 PCs biprocesseurs avec 1GB de mémoire vive et un réseau de Gigabit Ethernet et Linux comme système d'exploitation.

Les tests de fluide comprennent des tests sur une cube de 32x32x32, 64x64x64, et 128x128x128. Pour chaque taille, on a fait varier le nombre de noeuds de fluide entre 1 et le maximum possible sur la grappe sans mettre deux noeuds sur le même processeur. Les vitesses de fluide sont données en étapes de simulation par seconde.

Pour les tests des petites données statiques sur le fichier bonsai256x256x256.raw [1], on a lancé un noeud de données statiques sur un noeud quelconque, et puis on a varié le nombre de noeuds de fluide entre 1 et le maximum possible sur la grappe sans mettre deux noeuds sur le même processeur.

Pour les tests sur les données du CEA on a suivi le même plan, mais la taille des données oblige à des limitations fortes. Elles étaient vraiment trop grandes pour être testées sur la grappe de NDSU. Sur la grappe du LIFO on était limité à un noeud par ordinateur, car chaque noeud de rendu a besoin de plus de 512MB de mémoire. Alors même si chaque ordinateur a deux processeurs, mettre deux noeuds de rendu sur le même ordinateur oblige à utiliser le swap ce qui provoque une chute des performances.

On a fait les tests de niveau de détail (LOD) avec les données de CEA et sur la grappe du LIFO avec 7 noeuds de rendu, le maximum sur cette grappe. Les tests sont divisés en deux parties, une partie sur le LOD planar et une partie sur le LOD profondeur.

9.1.1 Analyse de la scalabilité de fluide

On calcule le temps de communication théorique pour la simulation de fluide pour les deux réseaux. Ce calcul nous a permis de valider l'analyse théorique en regardant les correspondances avec les tests de performance. On dit que la bande passante pour le réseau 100Mbit est à $10Mo/sec$, et pour le réseau 1Gbit elle est à $100Mo/sec$. La latence typique d'ethernet est de $350\mu sec$ [20]. On calcule alors le temps de communications avec un noeud de rendu et en fonction de différentes valeurs N de la taille du cube. On calcule aussi le temps qu'il faut pour seulement transmettre les données au noeud de rendu.

Bande passante	N	Latence	Temps → rendu	Temps totale
10	32	$350\mu sec$	$13.1ms$	$58ms$
10	64	$350\mu sec$	$105ms$	$243ms$
10	128	$350\mu sec$	$839ms$	$1351ms$
100	32	$350\mu sec$	$1.3ms$	$17ms$
100	64	$350\mu sec$	$10.4ms$	$30ms$
100	128	$350\mu sec$	$84ms$	$122ms$

Ici on a le temps de communication théorique. On a aussi le temps par étape de simulation qui a été mesuré dans les tests de performance sur un seul noeud. À partir de ces deux valeurs, il est possible de calculer le temps de calcul par étape avec un noeud de simulation qui est la différence du temps total et du temps de communication avec le noeud de rendu. Le temps de la phase de calculs sur un nombre de noeuds quelconque est inversement proportionnel au nombre de noeuds.

Bande passante	N	Temps total	Temps de comms	Temps de calcul
10	32	$103ms$	$13.1ms$	$90ms$
10	64	$1205ms$	$105ms$	$1100ms$
10	128	$11111ms$	$839ms$	$10272ms$
100	32	$71ms$	$1.3ms$	$69ms$
100	64	$1010ms$	$10.5ms$	$1000ms$
100	128	$20000ms$	$84ms$	$19916ms$

Dans ce tableau, `Temps total` est la performance mesurée avec un noeud de fluide, `Temps de comms` est le temps de communications théorique, et `Temps de calcul` est le temps de calcul sur un noeud qui est calculé à partir des autres valeurs.

Avec le temps de calcul, on peut alors faire un calcul théorique de la scalabilité de la simulation de fluide. Si t_c est le temps de calcul par processeur et f est le nombre de noeuds de fluide, on peut calculer le temps qu'il faut pour le calcul et le temps pour la communication :

$$t = \frac{t_c}{f} + 39l + \frac{76N^2 + N^3}{G} \times 4o$$

Cette fonction donne des résultats qui sont très proches de la réalité. Une comparaison entre les résultats théoriques et les données de tests est données dans les parties 9.3.1 et 9.3.2.

9.2 Analyse de la scalabilité de rendu

On calcule le temps de communication théorique pour le rendu de la même façon que pour la simulation de fluide. On utilise les mêmes valeurs que pour l'analyse de la simulation de fluide : bandes passantes de $10Mo/sec$ et $100Mo/sec$, et une latence de $350\mu sec$. Le LOD_{planar} est à 1.

Bande passante	N	Latence	Temps total	Communication	Calcul
10	256	$350\mu sec$	$2439ms$	$7ms$	$2432ms$
100	256	$350\mu sec$	$403ms$	$1ms$	$402ms$
100	512	$350\mu sec$	$4000ms$	$3ms$	$3997ms$

On peut tout de suite voir que le temps de calcul est énorme par rapport au temps de communication. Alors les projections théoriques de la vitesse sont presque linéaires en fonction du nombre de noeuds de rendu.

Les données de performance des tests de module de rendu sont dans les parties 9.3.3, 9.3.4, et 9.3.5. On peut voir que la performance du module de rendu est très loin d'être linéaire par rapport au nombre de processeurs. Sur le nombre maximal de noeuds, la différence est entre $40 - 100\%$.

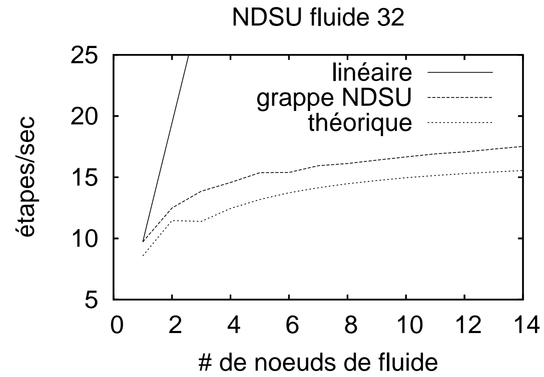
La raison de cette différence n'est pas très claire. L'algorithme de lancer de rayons est très parallèle et il n'y a pas de communications entre les noeuds de rendu.

9.3 Résultats des tests

9.3.1 Fluide sur la grappe de NDSU

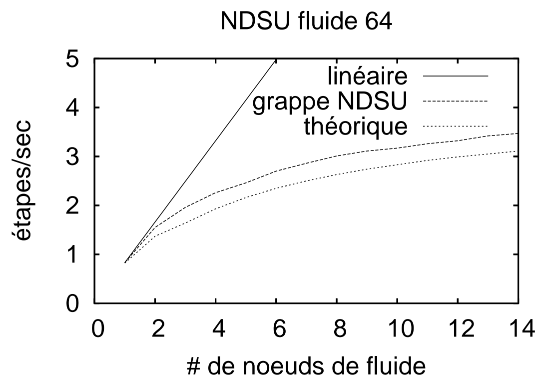
Ici les tests sur le fluide de taille 32 sur la grappe de NDSU :

Noeuds	Vrai	Théorique
1	9.72	8.58
2	12.49	11.46
3	13.86	11.39
4	14.57	12.45
5	15.37	13.18
6	15.39	13.73
7	15.94	14.14
8	16.12	14.47
9	16.40	14.73
10	16.66	14.96
11	16.91	15.14
12	17.07	15.30
13	17.30	15.43
14	17.52	15.55



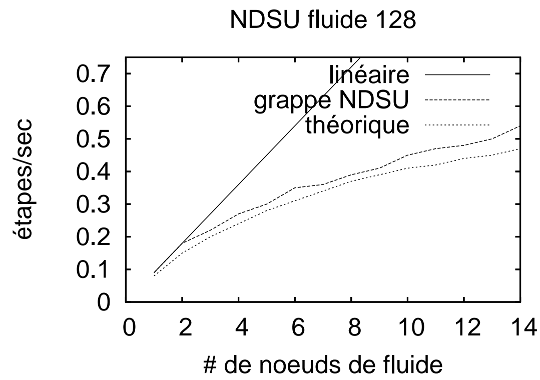
Ici les tests sur le fluide de taille 64 sur la grappe de NDSU :

Noeuds	Vrai	Théorique
1	0.83	0.82
2	1.55	1.37
3	1.96	1.64
4	2.26	1.93
5	2.46	2.16
6	2.70	2.35
7	2.86	2.50
8	3.01	2.63
9	3.11	2.74
10	3.17	2.83
11	3.26	2.92
12	3.32	2.99
13	3.42	3.05
14	3.47	3.11



Ici les tests sur le fluide de taille 128 sur la grappe de NDSU :

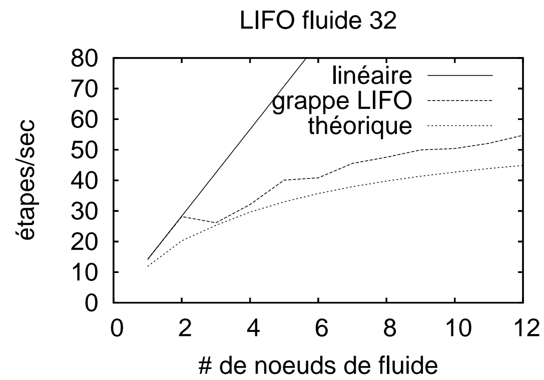
Noeuds	Vrai	Théorique
1	0.09	0.08
2	0.18	0.15
3	0.22	0.20
4	0.27	0.24
5	0.30	0.28
6	0.35	0.31
7	0.36	0.34
8	0.39	0.37
9	0.41	0.39
10	0.45	0.41
11	0.47	0.42
12	0.48	0.44
13	0.50	0.45
14	0.54	0.47



9.3.2 Fluide sur la grappe du LIFO

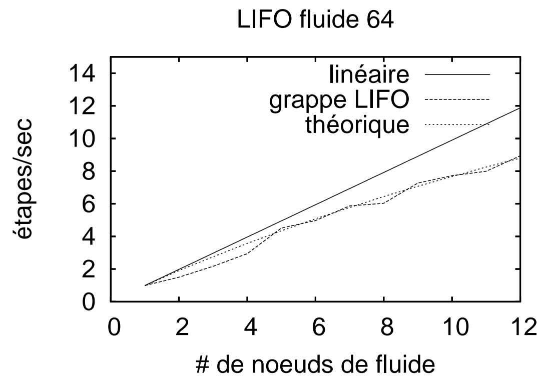
Ici les tests sur le fluide de taille 32 sur la grappe du LIFO :

Noeuds	Vrai	Théorique
1	14.16	11.91
2	28.17	20.22
3	26.14	25.31
4	32.11	29.61
5	40.08	32.98
6	40.81	35.69
7	45.55	37.92
8	47.57	39.77
9	49.94	41.35
10	50.40	42.70
11	52.11	43.88
12	54.74	44.91



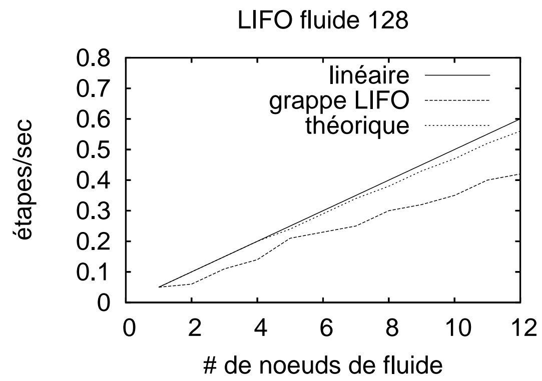
Ici les tests sur le fluide de taille 64 sur la grappe du LIFO :

Noeuds	Vrai	Théorique
1	0.99	0.98
2	1.50	1.91
3	2.17	2.75
4	2.94	3.57
5	4.51	4.34
6	4.97	5.08
7	5.87	5.77
8	6.03	6.44
9	7.27	7.07
10	7.72	7.67
11	7.99	8.25
12	8.94	8.80



Ici les tests sur le fluide de taille 128 sur la grappe du LIFO :

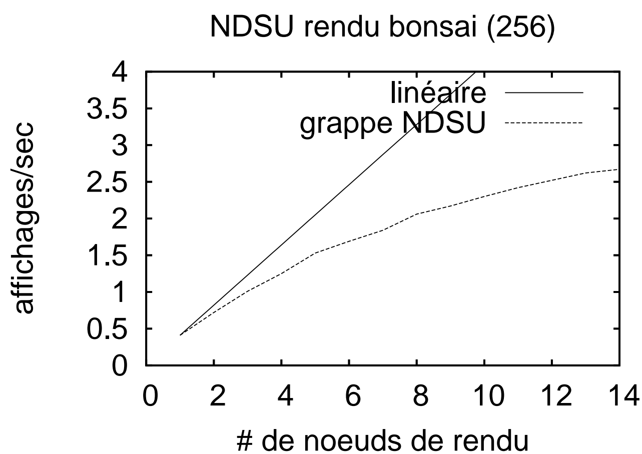
Noeuds	Vrai	Théorique
1	0.05	0.05
2	0.06	0.10
3	0.11	0.15
4	0.14	0.20
5	0.21	0.24
6	0.23	0.29
7	0.25	0.34
8	0.30	0.38
9	0.32	0.43
10	0.35	0.47
11	0.40	0.52
12	0.42	0.56



9.3.3 Données statiques sur la grappe de NDSU

Ici les tests sur les données statiques avec le fichier bonsai256x256x256.raw de taille 256 sur la grappe de NDSU :

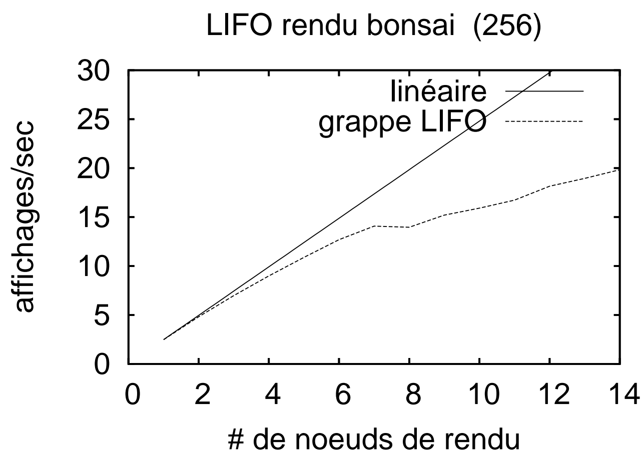
noeuds	FPS
1	0.41
2	0.72
3	1.01
4	1.25
5	1.53
6	1.69
7	1.84
8	2.06
9	2.17
10	2.30
11	2.42
12	2.52
13	2.62
14	2.67



9.3.4 Données statiques sur la grappe du LIFO

Ici les tests sur les données statiques avec le fichier bonsai256x256x256.raw de taille 256 sur la grappe du LIFO :

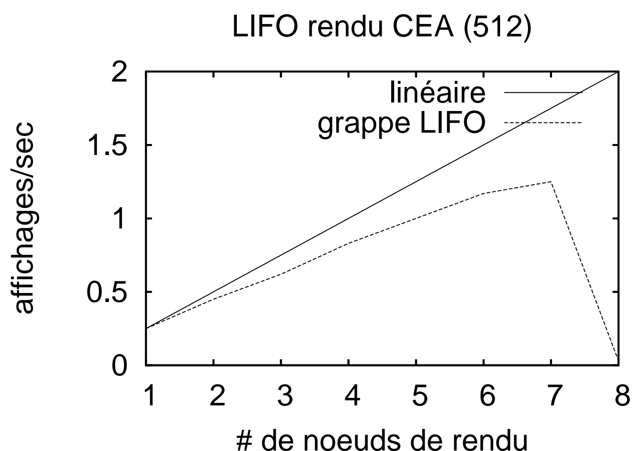
noeuds	FPS
1	2.48
2	4.82
3	6.98
4	8.99
5	10.88
6	12.69
7	14.08
8	13.96
9	15.20
10	15.91
11	16.73
12	18.15
13	18.95
14	19.84



9.3.5 Données du CEA sur la grappe du LIFO

Ici les tests sur les données statiques avec les données du CEA de taille 512 sur la grappe du LIFO. On peut voir que la vitesse diminue très fortement à 8 noeuds à cause d'un manque de mémoire vive :

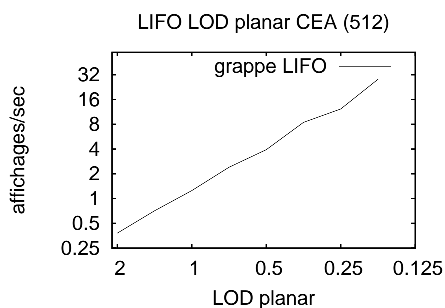
noeuds	FPS
1	0.25
2	0.45
3	0.62
4	0.83
5	1.00
6	1.17
7	1.25
8	0.03



9.3.6 Tests de LOD sur les données du CEA au LIFO

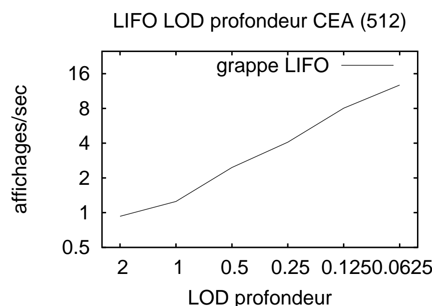
Ici les tests de LOD planar sur les données statiques avec les données du CEA de taille 512 sur la grappe du LIFO. Les données sont affichées sur une échelle logarithmique, et l'augmentation optimale théorique est une ligne droite :

noeuds	FPS	LOD_{planar}
7	0.38	2.0
7	0.71	1.414
7	1.25	1.0
7	2.40	0.707
7	3.93	0.5
7	8.44	0.3536
7	12.28	0.25
7	28.30	0.1765



Ici les tests de LOD profondeur sur les données statiques avec les donnés du CEA de taille 512 sur la grappe du LIFO. Les données sont aussi affichées sur une échelle logarithmique :

noeuds	FPS	$LOD_{profondeur}$
7	0.93	2.0
7	1.25	1.0
7	2.46	0.5
7	4.08	0.25
7	8.04	0.125
7	12.75	0.0625



10 Conclusion

L'objectif de ce travail était de créer une application de simulation de fluide intéractive en 3D et d'explorer les façons de traiter des grandes quantités de données en utilisant plus efficacement le matériel de l'ordinateur, et en déployant l'application sur une grappe de PCs.

La simulation de fluide est basée sur une simulation 2D proposée par Jos Stam. Le but de cette simulation est d'étendre ce fluide en 3D en conservant le temps-réel et la qualité de rendu.

Une simulation de fluide en 3D produit des données volumiques. Pour les afficher à l'utilisateur, on a étudié les techniques actuelles de rendu volumique, et on a implanté trois techniques pour les comparer. L'algorithme de lancer de rayons a été choisi pour sa souplesse, mais surtout pour ses caractéristiques parallèles.

L'optimisation des composants de l'application a permis d'augmenter la vitesse du rendu volumique d'un facteur de 3 et la simulation de fluide d'un facteur de presque 7, toujours sur un seul processeur.

Pour gagner encore plus de vitesse, on a parallélisé les algorithmes retenus pour déployer l'application sur grappes de PC. Pour gérer la communication entre les modules de simulation de fluide et de rendu, on a créé une bibliothèque pour les communications intermodule. Pour la communication dans le module de fluide, la bibliothèque standard MPI a donné de bons résultats.

On a fait une analyse théorique de performances des modules dans l'application, et l'application a été testée sur deux grappes de PC pour vérifier cette analyse. Les performances réelles de la simulation de fluide ont confirmé l'analyse théorique, avec un décalage de 20% au maximum. Le gain maximal de la simulation de fluide a atteint un facteur de 9 sur 12 noeuds sur la grappe du LIFO. Les performances réelles du rendu volumique ont été beaucoup moins élevées que l'analyse théorique ne le laissait supposer, avec un facteur maximal de 8 sur 14 noeuds, ou l'analyse indique un facteur de presque 14. Les raisons de cette différence n'ont pu être éclaircies.

Finalement la taille de la simulation a pu être doublée de 32 sur un processeur à 64 sur la grappe du LIFO avec une vitesse très similaire. Un facteur 2 augmente la taille des données d'un facteur de 8, alors le passage à la version

parallèle a vraiment permis d'obtenir des gains très intéressants. On a aussi réussi à afficher des données statiques à une vitesse de presque 20 affichages par seconde, ce qui est supérieur à la vitesse considérée comme acceptable pour la visualisation scientifique, 15 affichages par seconde, et 8 fois plus rapide que l'affichage non parallèle. On a aussi démontré l'efficacité d'une variable de niveau de détail (LOD) qui permet d'échanger la priorité de la vitesse et la qualité de l'affichage. Le résultat est une application de simulation de fluide et d'affichage des données volumiques statiques qui devraient permettre de faciliter l'exploration des données aux physiciens.

10.1 Directions futures

L'application finale a encore des possibilités d'amélioration. Avec le temps limité et le désir de créer quelque chose qui marche, il y a des raffinements qui n'ont pas été implantés.

10.1.1 Liens entre simulation et rendu

Souvent dans la simulation de fluide, il y a un grand espace qui reste vide. En localisant cet espace vide, le module de rendu devrait voir sa quantité de calculs fortement diminuer et gagner ainsi en vitesse.

La façon la plus simple est de chercher des coordonnées maximales et minimales sur chaque axe qui décrivent un cadre qui borne tout le fluide. Elle est facile à implanter mais son efficacité est liée aux conditions de simulation. Par exemple dans notre cas, le fluide rempli très rapidement le cube de la simulation, limitant ainsi les espaces vides.

Une autre possibilité serait de créer un octtree (arbre avec 8 fils pour chaque noeud) pour le cube. L'arbre commence avec un noeud qui décrit tout le cube. Pour chaque noeud, on peut le diviser en 8 sous-cubes, ou on peut le laisser. Avec une simple valeur de *vide* ou *non-vide* sur chaque feuille, on peut décrire l'état du cube avec peu de données. Il y aurait aussi la possibilité de limiter la résolution de l'arbre pour passer moins de temps à le construire et communiquer. Ce même arbre pourrait aussi servir pour limiter la quantité de données qu'il faut envoyer aux noeuds de rendu. Un exemple d'un octtree est illustré par la figure 20.

10.1.2 Schémas de calcul dans la simulation

Dans la partie 5.4, on a vu une possibilité pour améliorer l'accès à mémoire de la simulation. Il faudrait complètement changer le solveur de la simulation d'une façon difficile à gérer, mais on peut beaucoup gagner en utilisant mieux les caches du système de mémoire.

10.1.3 Communication asynchrone

Dans l'analyse traditionnelle des applications parallèles, on considère que la communication est une étape séparée de la phase de calcul, cependant le processeur ne fait rien sauf gérer la communication. Mais il est normalement possible de communiquer et calculer au même temps quand on peut faire le calcul sur les données qui ont été déjà communiquées avant.

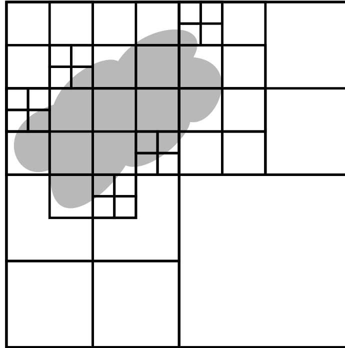


Figure 20: Un octtree sur un cube de fluide.

Dans la simulation de fluide, il y a beaucoup d'opportunités pour ce genre d'optimisation. Par exemple, dans l'étape de simulation il y a trois appels successifs à la fonction de `advect` sur trois ensembles de données indépendants. On pourrait alors communiquer les résultats d'un appel a cette fonction pendant que l'appel suivant fait son calcul. Si la communication est plus rapide que le calcul, on a effectivement enlevé deux phases de communication par étape de simulation. Sinon, on a effectivement enlevé deux étapes de calcul. Les autres possibilités sont la dernière communication dans la fonction de `lin_solve` et la dernière communication dans `project`. La gestion de ces communications deviendrait beaucoup plus difficile. Il faudrait analyser toutes les dépendances entre les calculs et insérer du code qui bloque l'exécution jusqu'à la fin de la communication, à chaque endroit où la communication asynchrone est utilisée. La vitesse de la simulation est très limitée par la communication, alors la possibilité de la réduire est prometteuse.

L'envoi des données au noeuds de rendu est aussi prometteuse. Les noeuds de fluide envoient les données de densité qui sont calculées à la fin de l'étape de simulation, et qui ne sont pas touchées ailleurs. Alors changer cette communication en communication asynchrone laisserait presque tout le temps d'une étape de simulation, pour transmettre ces données, et il est fort probable que cette communication deviendrait effectivement gratuite.

Le module de rendu est moins limité par le calcul. Il serait possible de décaler le rendu en introduisant une latence d'une étape d'affichage. Alors quand le rendu termine, il y aurait déjà une autre matrice prête pour commencer la prochaine étape de rendu, qui pourrait se dérouler pendant l'envoi des résultats de l'étape précédent.

Bibliographie

[1] Dirk Bartz, Stefan Gumhold Universität Tübingen http://www.gris.uni-tuebingen.de/areas/scivis/volren/datasets/datasets.html

[2] Scientific Visualization Laboratory, Georgia Tech "Scientific Visualization Tutorial" http://www.cc.gatech.edu/scivis/tutorial/tutorial.html

[3] Dan Baker, Charles Boyd "Volumetric Rendering in Realtime" *Proceedings of the 2001 Game Developers Conference*, 2001 http://www.gamasutra.com/features/20011003/boyd_pfv.htm

[4] Paul S. Calhoun, BFA, Brian S. Kuszyk, MD, David G. Heath, PhD, Jennifer C. Carley, BS, Elliot K. Fishman, MD "Three-dimensional Volume Rendering of Spiral CT Data: Theory and Method" *Radiographics*, 1999;19:745-764. http://radiographics.rsnajnls.org/cgi/content/full/19/3/745

[5] Chuck Hansen, Michael Krogh, James Painter, Guillaume Colin de Verdière, Roy Troutman "Binary-Swap Volumetric Rendering on the T3D" *Proceedings of the Cray Users Group 1995 Spring Conference*, 1995 http://www.ccs.lanl.gov/ccs1/projects/Viz/pdfs/95-cug95.pdf

[6] David S. Ebert, Roni Yagel, Jim Scott, Yair Kurzion "Volume Rendering Methods for Computational Fluid Dynamics Visualization" *IEEE Visualization*, 1994 http://dynamo.ecn.purdue.edu/~ebertd/papers/vis94.ps.Z

[7] Jos Stam "Real-time Fluid Dynamics for Games" *Proceedings of the Game Developers' Conference*, 2003 http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf

[8] Wei Hong, Feng Qiu, Arie Kaufman Stony Brook University "Hybrid Volumetric Ray-Casting" http://www.cs.sunysb.edu/~vislab/projects/gpgpu/hybrid.pdf

[9] Jens Krüger, Rüdiger Westermann TU-München "GPU Simulation and Rendering of Volumetric Effects for Computer Games and Virtual Environments" http://wwwcg.in.tum.de/Research/data/Publications/eg05.pdf

[10] Timothy J. Purcell, Ian Buck, William R. Mark, Pat Hanrahan Stanford University "Ray Tracing on Programmable Graphics Hardware" http://graphics.stanford.edu/papers/rtongfx/rtongfx.pdf

[11] Peter Shirley, Allan Tuchman University of Illinois "Polygonal Approximation to Direct Scalar Volume Rendering" http://www.cs.utah.edu/~shirley/papers/polygonal.pdf

[12] William E. Lorensen, Harvey E. Cline "Marching Cubes: A High Resolution 3D Surface Construction Algorithm" *Computer Graphics (Proceedings of SIGGRAPH '87)*, Vol. 21, No. 4, pp. 163-169

[13] James Sharman "The Marching Cubes Algorithm" http://www.exaflop.org/docs/marchcubes/ind.html

[14] Robert Fraser SGI Computer Systems Advanced Systems Division "Interactive Volume Rendering Using Advanced Graphics Architectures" http://web.archive.org/web/20001022092548/http://www.sgi.com/Technology/volume/VolumeRendering.html

[15] Rändy Osborne, Hanspeter Pfister, Hugh Lauer, Neil McKenzie, Sarah Gibson, Wally Hiatt, TakaHide Ohkami Mitsubishi Electric Research Lab "EM-Cube: An Architecture for Low-Cost Real-Time Volume Rendering" *Proceedings of the 1998 IEEE symposium on Volume visualization*, pp. 31-38 http://online.cs.nps.navy.mil/DistanceEducation/online.siggraph.org/2001/Courses/cd1/courses/13/Ws97.pdf

[16] Erik de Castro Lopo "Faster Floating Point to Integer Conversions" http://mega-nerd.com/FPcast/

[17] Apple Computer, Inc. "Software Pipelining" http://developer.apple.com/hardware/ve/software_pipelining.html

[18] Motorola, Inc., Freescale Semiconductor, Inc. "MPC7450 RISC Microprocessor Family User's Manual" http://www.freescale.com/files/32bit/doc/ref_manual/MPC7450UM_ZIP.zip

[19] Alexei Lebedev "Fixed Point Math For Speed Freaks" *MacTech*, Vol. 10, No. 3. http://www.mactech.com/articles/mactech/Vol.10/10.03/FixedPointMath/

[20] "Performance and Capacity Management Planning Guide for Solaris OS" *Sun Microsystems Blueprints*, CD 3, 2005-08-07

[21] Sylvain Jubertie "Techniques de parallélisation de pré-rendu pour la réalité virtuelle sur grappe de PC" *Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans*, Septembre 2003

[22] Jérémie Allard, Valérie Gouranton, Loïck Lecointre, Sébastien Limet, Emmaneul Melin, Bruno Raffin, Sophie Robert "FlowVR: a Middleware for Large Scale Virtual Reality Applications" *Euro-Par 2004 Parallel Processing. Proceedings.*, 2004 http://www-id.imag.fr/~raffin/papers/ID/europar04.pdf

[23] Jérémie Allard, Valérie Gouranton, Loïck Lecointre, Sébastien Limet, Emmaneul Melin, Bruno Raffin, Sophie Robert "FlowVR : Coupling Distributed Codes for High Performance Interactive Applications" http://flowvr.sourceforge.net/doc/flowvr/flowvr-manual.pdf