LIFO
Laboratoire d'Informatique Fondamentale d'Orléans

IUT Orléans

Rapport de stage — DUT Informatique

Cyril Crassin — Avril-Juin 2003

Réalisation d'une Démo de Réalité Virtuelle

IUT d'Orléans, Département Informatique

Remerciements

Je tiens tout d'abord à remercier Valérie Gouranton, Emmanuel Melin et Sébastien Limet qui m'ont accueillis au sein de leur équipe de recherche et qui m'ont suivi durant ces neuf semaines. Ils se sont tous les trois montrés extrêmement disponibles et ce fut un réel plaisir de travailler avec eux.

Je salut également Bertrand, Sylvain, Julien et Pierre les autres étudiants qui effectuaient également leur stage au LIFO ainsi que Sophie Robert et Souley Madougou membres eux aussi de l'équipe Réalité Virtuelle.

Sommaire

Introduction 4

Chapitre 1: La Réalité Virtuelle 5
1.1 Définition 5
1.2 Historique 6
1.3 Les domaines d'application 6
1.3.1 La visualisation scientifique 6
1.3.2 L'aide à la décision 7
1.3.3 La formation 7
1.3.4 La vulgarisation scientifique et l'éducation 7
1.3.5 Les loisirs, le divertissement, la vente 8
1.4 Le matériel 9
1.4.1 Les interfaces de sortie 9
1.4.2 Les interfaces d'entrée 12
1.4.3 Les interfaces d'entrée-sortie 13
1.4.4 Les calculateurs 14
1.5 Les problèmes de la Réalité Virtuelle 15

Chapitre 2: Le LIFO 16
2.1 Présentation générale 16
2.2 L'équipe Réalité Virtuelle 16

Chapitre 3: L'environnement logiciel 17
3.1 Description générale 17
3.2 L'API OpenGL 18
3.2.1 Historique 18
3.2.2 Les avantages d'OpenGL 18
3.2.3 Les bases 19
3.2.4 Les primitives de dessin 19
3.2.5 Le plaquage de texture 20
3.2.6 L'éclairage 21
3.2.7 Les buffers 22
3.2.8 Les display lists 23
3.2.9 Les matrices 23
3.2.10 Le blending 24
3.2.11 Les extensions 24
3.2.12 Le pipeline de rendu 25
3.2.13 Les évolutions matérielles 25
3.3 VR-Juggler 32
3.3.1 L'architecture 32
3.2.2 La boucle de dessin 33

3.4 Net-Juggler 34
3.5 MPI 34

Chapitre 4: Mon projet de stage 35
4.1 Description générale 35
4.2 L'installation d'une grappe de PC 36
4.2.1 L'architecture matérielle 36
4.2.2 La distribution Clic 36
4.3 La simulation d'écoulement de fluide 37
4.3.1 Le principe 37
4.3.2 Le passage en 3D 37
4.3.3 La visualisation de données scientifiques 38
4.3.4 Ma solution 41
4.3.5 Les perspectives 44
4.4 La création d'outils génériques de parallélisation de jeux cellulaires 45
4.4.1 Problématique 45
4.4.2 La grille 45
4.4.3 La fenêtre sur grille 45
4.4.4 La grille parallèle 46
4.4.5 Application au jeu de la vie 47
4.4.6 Le plateau de jeu 48
4.5 La réalisation d'une démo 49
4.5.1 Présentation de la démo 49
4.5.2 Ma problématique 49
4.5.3 La notion de Layer 49
4.5.4 La séparation des données brutes et des données de contexte 49
4.5.5 Les textures 50
4.5.6 Les matériaux 51
4.5.7 Les modèles 52
4.5.8 Les modèles animés 54
4.5.9 Le rendu de pelage 57

Conclusion 58

Introduction

Dans le cadre de l'obtention du DUT Informatique, les étudiants doivent effectuer un stage en entreprise d'une durée de 9 semaines afin de mettre en pratique les compétences acquises au cours des deux ans de formation à l'IUT.

Pour ma part, j'ai eu la chance d'effectuer mon stage au LIFO, le Laboratoire d'Informatique Fondamentale d'Orléans où j'ai intégré l'équipe de recherche sur la Réalité Virtuelle. Il s'agit en fait du laboratoire de recherche du département Informatique de l'université.

J'ai découvert le travaille qu'effectue cette équipe lors de la présentation au publique réalisée à l'occasion de la fête de la science et j'ai tout de suite été très intéressé par leurs recherches. Je suis passionné depuis très longtemps par tout ce qui touche à l'infographie et à la programmation 3D et un stage dans une équipe de recherche dans le domaine de la Réalité Virtuelle a vraiment été une formidable occasion d'en apprendre encore plus et d'alimenter ma passion.

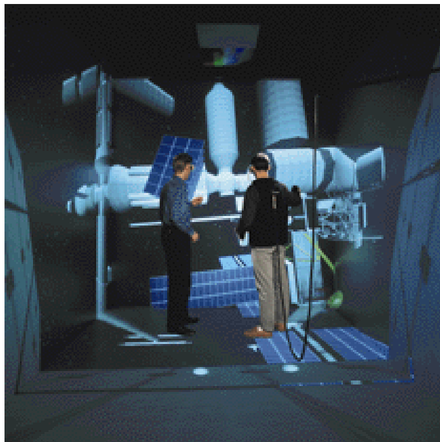
L'objectif premier de mon stage était de créer une démonstration de réalité virtuelle destinée à la présentation au public des différents travaux réalisés par l'équipe *Réalité Virtuelle*. Le cahier des charges de cette démo était d'être graphiquement évoluée et de mettre en oeuvre quelques techniques de parallélisation. Elle est basée sur un projet de simulation de vie artificielle réalisé en cours mettant en jeu une population de lapins mangeant de l'herbe.

Chapitre 1

La réalité virtuelle

1.1 Definition

"Réalité virtuelle: Interface homme/machine évoluée qui simule un environnement réaliste et autorise une interaction avec ce dernier"



Un environnement de réalité virtuelle et un ensemble visiocasque et gants de donnée

La réalité virtuelle, on parle souvent de VR de l'anglais Virtual Reality, rassemble l'ensemble des techniques et systèmes qui procurent à l'homme le sentiment de pénétrer dans des univers synthétiques créés sur ordinateur avec la possibilité d'y effectuer en temps réel un certain nombre d'actions définies par un ou plusieurs programmes informatiques. Ces techniques lui donnent la possibilité d'éprouver physiquement un certain nombre de sensations (visuelles, auditives, haptiques, olfactives, etc.) et celle de pouvoir opérer dans ces mondes par des moyens d'action "naturels" (mouvements du corps, gestes, voix,...).

L'élément de base à l'origine du développement de la réalité virtuelle est la visualisation tridimensionnelle. C'est sa combinaison avec des interfaces matérielles spécifiques qui permet une immersion totale dans un environnement entièrement créé sur ordinateur.

1.2 **Historique**

L'expression « réalité virtuelle » a été introduite par Jaron Lanier, fondateur avec Jean-Jacques Grimaud de VPL Research en 1985, compagnie qui a produit le dispositif du *DataGlove* (gant de données présenté plus haut).

Les premières expérimentations de réalité virtuelle datent cependant des années 1960, avec l'invention par Morton Heilig du *Sensorama*, sorte de machine cinématographique proposant une expérience d'immersion dans l'image, la fabrication par Ivan Sutherland au MIT (Massachusetts Institute of Technology) d'un dispositif d'affichage stéréoscopique monté sur tête avec capteurs de position, et la mise sur pied par Frederick Brooks, de l'Université de Caroline du Nord, d'un programme de recherche visant à créer un dispositif combinant la visualisation et le retour d'effort.

Le procédé *LEEP*, qui entre dans le dispositif d'affichage d'un bon nombre de casques de visualisation, a été introduit dans le milieu des années 1970. Les recherches de Myron Krueger dans le domaine de la réalité artificielle remontent à 1969. En fait, la recherche se poursuit de façon intensive depuis les années 1980.

Plusieurs dispositifs de visualisation ainsi que de retour tactile et d'effort ont ainsi été mis au point dans les centres et entreprises de recherche et développement, en même temps qu'on continuait de perfectionner les composantes matérielle et logicielle des systèmes de réalité virtuelle.

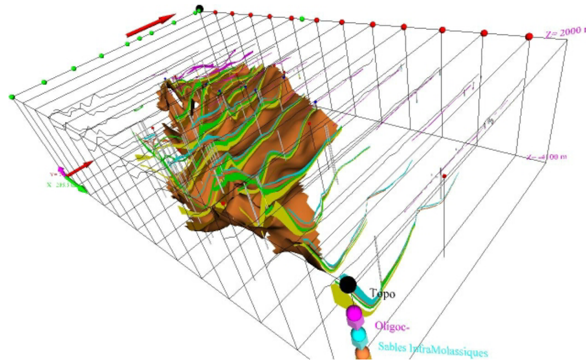
L'autre voie de développement de la réalité virtuelle est la partie logicielle qui elle ne se développe réellement que depuis à peine une quinzaine d'années. Immerger quelqu'un dans un environnement virtuel suppose en effet de lui proposer des images calculées sur ordinateur crédibles et donc réalistes, une chose qui est longtemps restée difficile en raison de la puissance de calcul nécessaire à la production en temps réel de ces images.

1.3 **Les domaines d'application**

Le champ d'application de la VR est extrêmement large du fait de toutes les facilitées qu'elle peut apporter dans beaucoup de domaines et voici une rapide description des principaux.

1.3.1 **La visualisation scientifique**

La visualisation scientifique est sans doute le premier domaine d'application de la VR. De part l'immersion qu'elle apporte, elle permet une meilleure perception et une facilité d'interprétation de résultats de simulation scientifiques. Il est ainsi possible d'explorer visuellement des modèles mathématiques complexes ce qui permet de mieux les comprendre. Cela apporte également une grande souplesse dans la manipulation des données très utile par exemple dans l'étude de phénomènes physiques. Un élément important de la réalité virtuelle est en effet la possibilité d'intervenir en temps réel sur l'environnement.

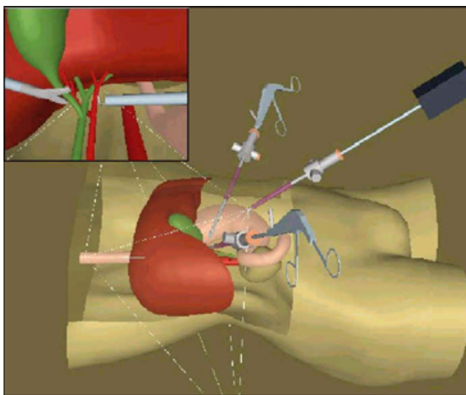


1.3.2 **L'aide à la décision**

La représentation tridimensionnelle est celle à laquelle on est habitué dans notre vie de tous les jours, c'est comme cela que l'on perçoit le monde qui nous entour. Dans certains domaines de pointe, la complexité des données à prendre en compte dans la prise de décisions peut nécessiter des compétences de spécialistes que ne possèdent pas forcément les décideurs. C'est dans ce contexte qu'un environnement de réalité virtuelle peut apporter une vision synthétique plus facilement assimilable.

1.3.3 **La formation**

Pour les mêmes raisons, la réalité virtuelle est un formidable outil de formation. Le fait de pouvoir représenter de manière synthétique des notions complexes ou abstraites ainsi que la mise en jeu de tous les sens dans la perception des données présentées apportent une très grande facilité d'assimilation. Cela peut être très utile dans la formation de spécialiste comme des pilotes ou des chirurgiens par exemple qui peuvent ainsi acquérir de l'expérience dans la réalisation de certaines opérations complexes. Cela peut donc être très utile pour éviter les risques réels liés à certaines formations.



Une opération chirurgicale virtuelle et un simulateur de vol

1.3.4 **La vulgarisation scientifique et l'éducation**

La VR peut également jouer un rôle très important dans l'explication de la science aux plus jeunes en la rendant ainsi plus attrayante et moins abstraite. Elle peut également être un puissant outil culturel en permettant par exemple la visite virtuelle de lieux historiques, de musées, de planètes lointaines... Le coté ludique de la réalité virtuelle peut ainsi susciter plus d'intérêt chez les enfants.

1.3.5 **Les loisirs, le divertissement, la vente ...**

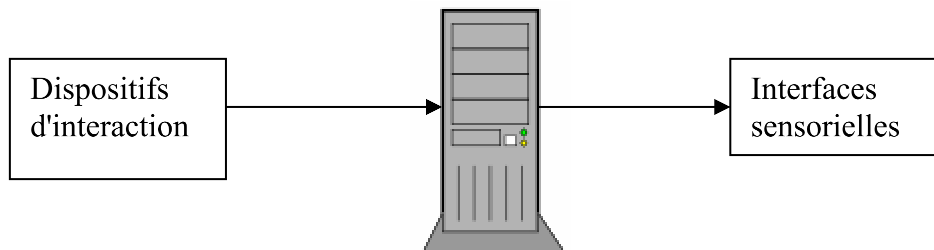
C'est sans doute dans les domaines grands publics que la VR trouvera ses applications les plus importants. Imaginez un jeu si réaliste que vous vous sentiriez comme dans un rêve où vous pourriez agir sur tout ce qui vous entoure avec la sensation d'en être le héro. On peut également imaginer des voyages virtuels dans des pays exotiques un peu à la manière de ce que l'on voit dans le film "Total Recall".

Une autre application qui va sans doute se développer rapidement concerne le domaine de la vente. La possibilité de pouvoir montrer à un client le produit qu'il va acquérir sans que celui ci n'existe encore ou ne soit disponible sur place. On peut imaginer proposer la visite de pavillons virtuels, la conduite d'une voiture etc...

Des clients découvrant *virtuellement* leur future voiture

1.4 **Le matériel**

Les composants matériels liés à un environnement de réalité virtuelle se divisent en trois catégories: Les interfaces d'entrée ou dispositifs d'interaction, les interfaces de sortie ou interfaces sensorielles et les calculateurs.



1.4.1 **Les interfaces de sortie**

Ce sont les éléments permettant de transmettre de l'information à l'utilisateur et ils participent grandement à l'immersion et la sensation de réalité. La plus importante est bien sur le dispositif d'affichage. Il existe des dispositifs audio, tactiles ou olfactifs mais ils restent assez peu répandus.

Le dispositif d'affichage de base est bien entendu le moniteur que nous connaissons tous, il permet tout comme le video-projecteur d'afficher sur une surface plane des images qui peuvent êtres tridimensionnels. Le problème de ce type d'affichage, est qu'il ne restitue pas correctement la sensation de profondeur. L'impression de volume est créée par l'utilisation de divers subterfuges visuels dans la modélisation de l'environnement (éclairages, ombres etc...) et l'utilisation d'une méthode de projection des données tridimensionnelles qui simule un effet de profondeur (les éléments éloignés du point de vue paraissent plus petit). Il manque donc un élément important qui nous permet de voir les objets en relief: la stéréographie. Il a donc été inventé divers dispositifs permettant une vision stéréographique dont voici une description rapide et non exhaustive des principaux.

- **Le visiocasque ou HMD (Head Mounted Display):**

Le principe est simple: deux écrans, un pour chaque oeil, affichants les informations avec le décalage nécessaire à une vision stéréoscopique.
Le problème de ce type d'équipement est qu'il est mono utilisateur et surtout assez troublant quand on n'y est pas habitué.

- **Les lunettes a obturation:**

Elles sont utilisées conjointement avec un système de visualisation "plan" comme un moniteur ou les interfaces présentées plus bas. Elles servent à masquer alternativement l'oeil gauche et droit afin que les images affichées parviennent au bon oeil pour créer l'effet stéréoscopique. Elles sont généralement sans fil et peuvent êtres traquées c'est a dire que leur position peut être envoyée au calculateur pour en tenir compte dans la simulation. Cela permet par exemple quand on tourne la tête, de faire tourner le point de vue affiché. Il existe d'autres systèmes dits passifs, basés sur la superposition d'images.

➢ Les Workbenchs ou plans de travail virtuels

Ce sont en fait de grands écrans rétro-projetés, généralement constitués de deux plans en L. Utilisées conjointement à des lunettes à obturation elles permettent a plusieurs personnes de visualiser en même temps le même environnement. J'ai eu la chance de pouvoir essayer ce type d'équipement au BRGM et l'effet est réellement saisissant. On a vraiment l'impression que les objets affichés flottent devant nous. Ils sont principalement utilisés pour la visualisation scientifique et leur principal défaut est leur manque d'immersion.

➢ Les murs d'images

Ce sont de grandes surfaces d'affichage planes ou cylindriques autorisant un plus grand nombre d'utilisateurs. Ils sont généralement formés de plusieurs vidéo-projecteurs

➤ Les Caves

Il s'agit d'énormes cubes constitués d'écrans un peu à la manière de *l'holodeck* dans la série Star Trek. L'utilisateur se place à l'intérieur et peut ainsi observer intégralement l'environnement qui l'entoure. Ce dispositif est sans doute le plus immersif et a l'avantage de pouvoir être utilisé par plusieurs personnes en même temps. Malheureusement, il y a très peu de laboratoires qui possèdent ce type d'installation dans le monde.

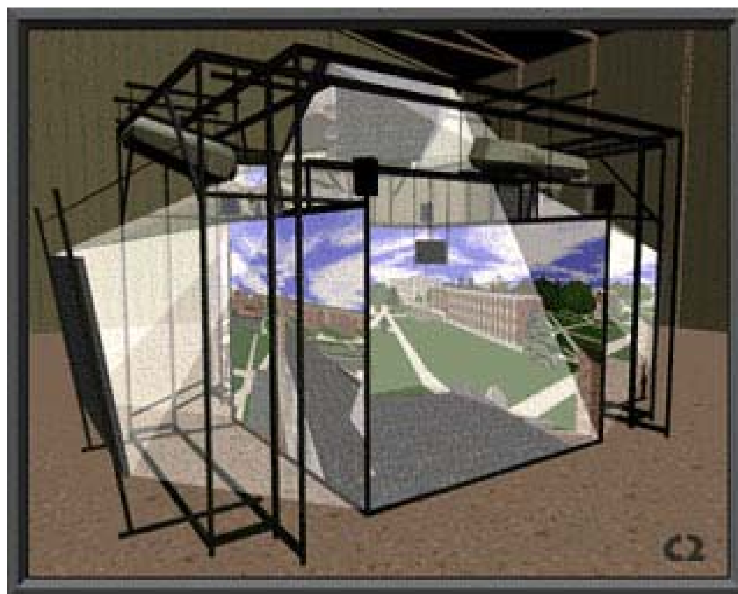


Schéma structurel d'un CAVE

1.4.2 Les interfaces d'entrée

Ce sont les dispositifs d'interaction avec l'environnement de VR. Ils permettent généralement de transmettre des informations de position et de mouvement au calculateur gérant la simulation. En voici quelques uns parmi les plus répandus:

➢ **Les souris 3D**

Il s'agit en fait d'un dispositif équipé d'une boule que l'on manipule à la manière d'un trackballs (une espèce de sourie inversée, c'est à dire avec la boule sur le dessus) mais dans les 3 dimensions. Ce genre de dispositif est en fait assez peu pratique de par sa manipulation peu intuitive.

➢ **Les wands**

Ce sont des périphériques plus évolués qui ressemblent à une télécommande et sont généralement trackés, c'est à dire localisés dans l'espace. Ils permettent de manipuler des objets dans l'environnement de VR d'une façon assez naturelle en donnant la sensation d'agir directement à la main. On peut ainsi saisir et déplacer des objets en manipulant de la même façon le *wand*. Ce type de périphérique est extrêmement répandu car relativement abordable par rapport à d'autres équipements, il est par exemple utilisé sur le Workbench du BRGM.

➢ **Les gants de données**

Ce sont des périphériques très évolués qui permettent de récupérer les mouvements de la main et des doigts de l'utilisateur. Il en existe plusieurs types utilisant des technologies de capteurs qui peuvent êtres optiques ou électromagnétiques. C'est l'un des périphériques les plus adaptés à la manipulation d'objets dans un environnement virtuel.

1.4.3 **Les interfaces d'entrée-sortie**

Appelées *interfaces sensori-motrices* elles permettent à l'utilisateur d'envoyer à la fois des informations de déplacement à l'ordinateur et de recevoir des informations tactiles sur les objets qu'il manipule. Ce sont généralement des périphériques dits à retour d'efforts un peu à la manière de ce que l'on connaît dans le domaine des jeux vidéos avec les joysticks à retour de force par exemple.

➢ Les interfaces haptiques

Les *interfaces haptiques* sont des périphériques capables de simuler la prise d'objet ou de reproduire la texture ou la résistance de surfaces. Elles sont de différents types et généralement basé sur des exosquelettes. Elles sont très utiles en médecine par exemple pour une simulation d'opération chirurgicale.

Plusieurs interfaces haptiques utilisées dans divers domaines d'application de la VR

➢ Les Spidars

Le Spidar (SPace Interface Device for Artificial Reality) est un dispositif constitué d'un mécanisme de fils et de poulies motorisées qui permettent d'appliquer des forces localisées spacialement sur l'utilisateur via une bague par exemple.

1.4.4 **Les calculateurs**

Le calculateur est l'élément central de tout environnement de réalité virtuelle. C'est lui qui gère le monde virtuel en exécutant les applications de simulations et qui effectue les taches de rendu graphique de l'environnement. Il reçoit les informations en provenance des périphériques d'entrée et envoi les données aux périphériques de sortie. Il s'agit généralement de stations graphiques haut de gamme équipées de plusieurs pipelines graphiques (spécialement dédiés à l'accélération OpenGL) et disposant d'une puissance de calcul très importante grâce à l'utilisation de nombreux processeurs associés autour d'une architecture mémoire commune. Elles sont également équipées d'une quantité très importante de mémoire vive (plusieurs giga-octets) et d'une grande capacité de stockage. Ce sont généralement des stations de type Cray ou SGI extrêmement performantes mais également très chères.

La gamme complète des stations SGI

Une station Cray

1.5 Les problèmes de la réalité virtuelle

1.5.1 Le coût du matériel

Le problème principal qui se pose lorsque l'on veut faire de la réalité virtuelle, c'est bien entendu le coût du matériel. Que ce soit les interfaces sensorielles, les environnements d'affichage ou les calculateurs, tous ces matériels sont extrêmement coûteux car très spécifiques à un domaine encore peu développé. Il n'y a que très peu d'entreprises qui produisent ce genre d'équipement et en général en petite quantité si ce n'est sur mesure. Cela vient surtout du fait que les principaux clients de ce genre de matériel restent encore les laboratoires de recherches, un marché tout de même assez restreint.

1.5.2 La puissance de calcul

L'autre problème qui se pose est le besoin extrêmement important (et virtuellement infini) en puissance de calcul que nécessitent les simulations de VR et particulièrement les simulations temps réel. Les environnements simulés demandent toujours plus de puissance de traitement que ce soit pour les simulations elles-mêmes ou pour les rendus graphiques. Même les stations haut de gamme dédiées à ce genre de calculs ne suffisent pas toujours pour faire tourner certains programmes.

Chapitre 2

Le LIFO

2.1 **Présentation générale**

Le LIFO (Laboratoire d'Informatique Fondamentale d'Orléans) est un laboratoire de l'Université d'Orléans reconnu par le Ministère de la Recherche et le CNRS. Les recherches menées au LIFO concernent la science et l'ingénierie du logiciel. Elles visent l'amélioration des théories de la programmation et des techniques qui en découlent. Les chercheurs du LIFO définissent, modifient et spécialisent de nombreux modèles mathématiques du calcul par ordinateur et du logiciel qui prescrit ces calculs. Le laboratoire est structuré en trois équipes:

- Contraintes et Apprentissage (CA)
- Graphes et Algorithmes (GA)
- Vérification, Parallélisme et Sécurité (VPS)

2.2 **L'équipe Réalité Virtuelle**

L'équipe Réalité Virtuelle dans laquelle j'ai effectué mon stage travaille en fait au sein de l'équipe VPS. Elle est constituée principalement de chercheurs issus du monde du parallélisme. Son principal axe de recherche est la mise en oeuvre de techniques de parallélisation dans le but de pallier aux problèmes de coût et de puissance de calcul que posent les environnements de réalité virtuelle en utilisant des grappes de PC. Le but est ainsi de permettre l'utilisation d'une grappe de PC de manière transparente en remplacement des coûteuses stations de travail.

Ces recherches ont permises de développer une première réponse logicielle appelée Net-Juggler qui permet la distribution de l'état de périphériques d'entrée ainsi que le paramétrage des affichages sur l'ensemble des machines d'une grappe. Net-Juggler est basé sur l'API de réalité virtuelle VR-Juggler. Une solution de synchronisation de l'affichage appelée *softgenlock* a également été développée afin d'assurer une parfaite synchronisation des machines indispensable dans une configuration multi écran.

L'équipe travaille principalement en collaboration avec le BRGM sur la réalité virtuelle appliquée aux géosciences avec un *workbench* sur le cite d'Orléans ainsi qu'avec l'INRIA de Grenoble et le CEA.

Chapitre 3

L'environnement logiciel

3.1 Description générale

La partie logicielle est l'élément fondamental d'un environnement de réalité virtuelle. Au LIFO, tous les développements se font dans un environnement PC Open Source donc sous Linux.

L'affichage 3D est la base de la réalité virtuelle, c'est grâce à lui que l'on peut créer des environnements et permettre à l'utilisateur de s'y plonger. Je vais donc commencer par vous présenter les bases de l'affichage 3D sous OpenGL, l'API que j'ai utilisée tout au long de mon stage. Cette partie est assez développée mais permettra de bien comprendre par la suite les développements que j'ai effectués. Je vous présenterai ensuite plus en détail VR-Juggler, la couche d'abstraction matérielle et Net-Juggler l'interface de parallélisation des entrées développée au LIFO. Pour finir je vais vous présenter MPI, l'API de communication que j'ai utilisée pour la parallélisation des calculs.

3.2 L'API OpenGL

OpenGL est donc une API (Application Programming Interface, une bibliothèque de programmation d'application) graphique développée par Silicon Graphics permettant l'affichage à l'écran d'environnements tridimensionnels. C'est une librairie de bas niveau qui offre tout de même des fonctions qui vont de l'affichage de points à l'éclairage de modèles en passant par le plaquage de textures.

3.2.1 Historique

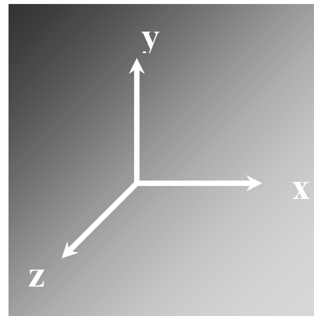
Le standard OpenGL est apparu en 1992 et est issu de l'API *IRIS GL* de Silicon Graphics. Ce standard est contrôlé par un organisme appelé ARB (Architecture Review Board) regroupant les principaux acteurs du monde de la visualisation 3D tels que 3DLabs, ATI, nVIDIA, IBM, Intel, Microsoft (qui l'a quittée récemment), SGI ou encore Sun. L'ARB est chargée de contrôler l'évolution de l'API en validant les nouvelles fonctions et en les intégrant au standard. Le standard OpenGL n'évolue que très rarement, nous en sommes actuellement à la version 1.4 et bientôt 2.0, mais chaque évolution marque l'intégration d'un grand nombre d'innovations. OpenGL a été développé à l'origine pour les stations graphiques SGI et s'est rapidement étendu au monde PC avec l'arrivée des cartes graphiques possédant une accélération 3D. Elle utilise donc pleinement les capacités matérielles de ces cartes pourvu que le constructeur fournisse des pilotes compatibles.

3.2.2 Les avantages d'OpenGL

Les principaux avantages de l'API OpenGL sont la portabilité tout d'abord puisqu'elle est totalement multi plateforme et indépendante du système de fenêtrage à l'inverse de Direct3D l'autre API 3D développée par Microsoft sous environnement Windows. L'autre intérêt d'OpenGL est son extensibilité assurée par un mécanisme d'extensions permettant aux constructeurs de matériel de fournir rapidement un support logiciel aux nouvelles fonctions qu'ils implémentent. OpenGL reste ainsi toujours à la pointe des évolutions matérielles et permet aux développeurs de profiter immédiatement de toutes les innovations sans avoir à attendre la sortie d'une nouvelle version comme c'est le cas avec Direct3D. De plus, la grande popularité de cette API fait qu'elle est extrêmement bien documentée et que l'on trouve facilement les informations nécessaires pour en exploiter toutes les fonctions.

3.2.3 **Les bases**

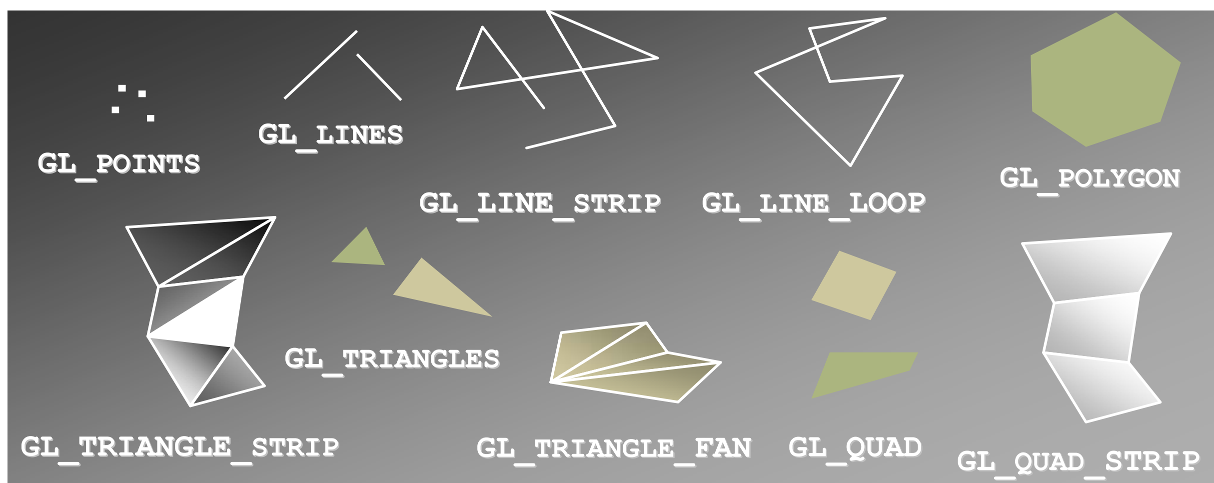
OpenGL est une API dite fonctionnelle, c'est à dire qu'elle se base uniquement sur un jeu de fonctions et non pas sur une architecture Objet. De plus, elle fonctionne sur le principe de la machine à état, à chaque appel d'une fonction OpenGL, on change l'état de la machine et toutes les actions de rendu effectuées ensuite en sont affectées. Le repère utilisé par OpenGL est un repère orthogonal *"right handed"*.



Le repère OpenGL

3.2.4 **Les primitives de dessin**

Quoique l'on veuille obtenir en OpenGL, on dessine toujours des points dans l'espace, on parle généralement de vertex. Ces vertex sont définis par leurs trois coordonnées spatiales et sont passées à OpenGL dans un ordre particulier afin de former des primitives de base. En général, ces vertex servent à définir des triangles (3 vertex) car c'est la figure basique définissant un plan et qui permet de former n'importe quel objet. OpenGL propose également d'autres primitives comme les QUADS (rectangles) définis par 4 vertex, les lignes ou les polygones, une figure fermée formées d'un nombre indéfini de vertex.

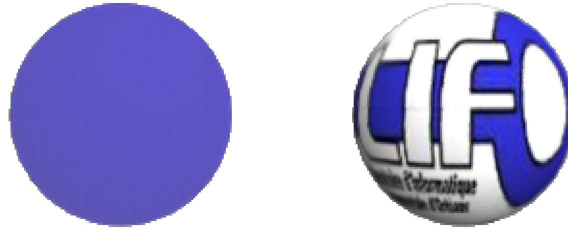


Les primitives de dessin OpenGL

Il existe des primitives un peu plus évoluées comme les *TRIANGLE_STRIP* ou les *TRIANGLE_FAN* qui permettent de former plusieurs triangles connectés sans redéfinir leurs vertex communs.

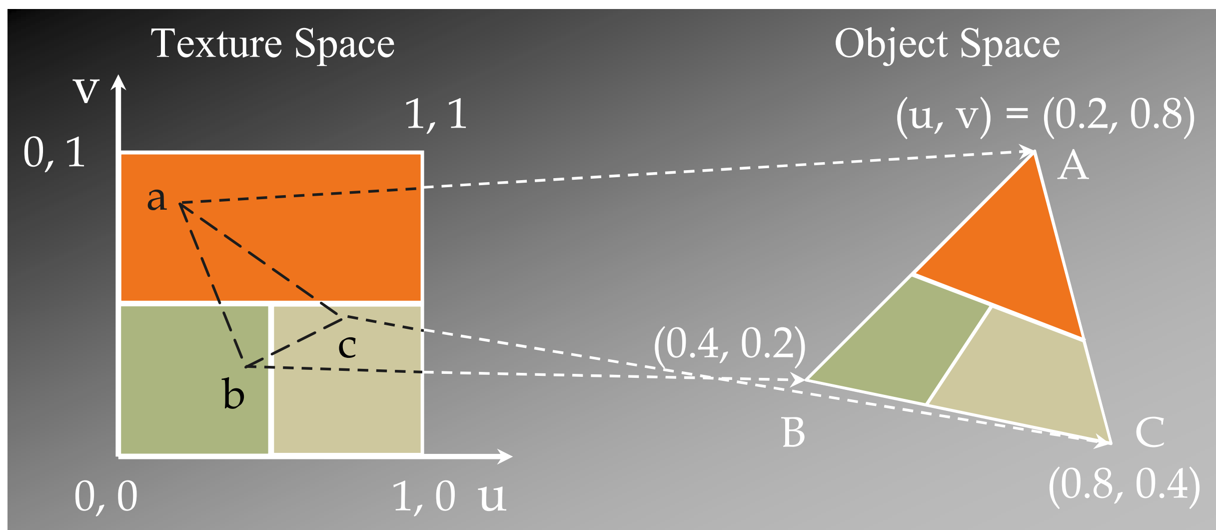
Les vertex possèdent en plus de leur position d'autres attributs comme une couleur, une coordonnée de texture ou une normale (et d'autres) dont je vous expose l'utilité un peu plus loin.

3.2.5 **Le plaquage de texture**



Un rendu de sphère sans, puis avec texture

L'un des éléments fondamental du rendu 3D sont bien entendu les textures, une sorte de peau recouvrant les objets. Elles permettent d'apporter un grand nombre de détails aux objets dessinés sans avoir à ajouter de points. C'est donc un élément très important pour le réalisme d'un rendu. Il peut s'agir d'images lues à partir d'un fichier ou générées par le programme lui même. Les textures sont généralement (nous verrons plus loin que ce n'est pas toujours le cas) des images en deux dimensions qu'il faut donc appliquer sur des objets définis eux dans l'espace. C'est là qu'interviennent les coordonnées de textures définis par vertex qui permettent d'associer à chaque point une coordonnées sur la texture. OpenGL peut ainsi plaquer la texture sur les primitives dessinées. Ces coordonnées de texture, désignées généralement par les lettres u et v (ou s et t), peuvent êtres définies à la main ou générées automatiquement par OpenGL.



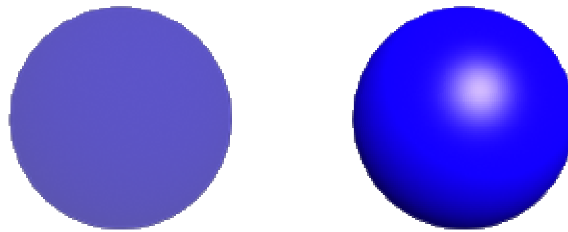
Exemple de *mapping* d'une texture sur un triangle

Une évolution du *texturing*, présente tout d'abord sous la forme d'une extension puis intégrée au standard OpenGL (a partir de la version 1.3) est le *multi-texturing*. Le *multi-texturing* permet d'appliquer plusieurs textures à une même primitive en choisissant la façon dont elles vont se combiner. Le *multi-texturing* nécessite la définition pour chaque vertex de plusieurs coordonnés de texture, une pour chaque texture appliquée.

Une autre évolution a été l'introduction de textures non plus carrées mais cubiques donc tridimensionnelles nécessitant la définition par vertex de 3 coordonnées de mapping (u, v, w ou s, t, r). Récemment sont également apparues un type de textures appelées *Cubes Map* qui définissent les 6 faces d'un cube.

3.2.6 **L'éclairage**

L'éclairage est le deuxième élément qui permet d'ajouter du réalisme à une scène 3D. OpenGL permet en effet de définir des sources lumineuses qui servent à éclairer les objets dessinés ajoutant ainsi des jeux d'ombres et lumières qui donnent une meilleure impression de relief.

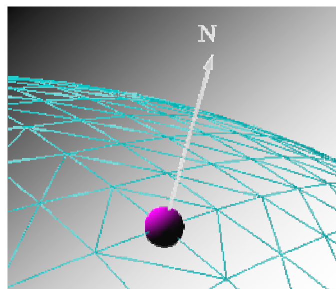


Un rendu de sphère sans puis avec éclairage

En standard, les calculs d'éclairage sont effectués par vertex en combinant à la couleur de ce dernier une couleur issue de l'application d'un modèle d'éclairage. Il existe de nombreux modèles permettant de simuler l'éclairage d'un objet et celui implémenté en standard en OpenGL (*Phong lighting*) utilise une combinaison de 3 composantes:

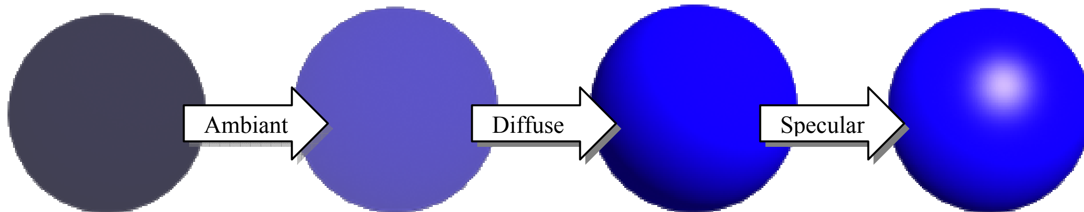
- **Ambiante**: Il s'agit simplement de multiplier la couleur de chaque vertex par une couleur fixe. Cela simule donc une lumière omnisciente sans source particulière d'émission.

- **Diffuse**: La composante diffuse est calculée grâce au vecteur normal stocké dans les vertex. Il s'agit d'un vecteur à 3 dimensions normalisé (c'est a dire de longueur 1) représentant la direction de la normale à la surface. Ces normales peuvent être calculées automatiquement par OpenGL dans certains cas mais doivent souvent êtres placées manuellement par le programmeur. La composante diffuse est donc calculée en fonction de la position de la source lumineuse, plus le vecteur reliant le vertex à la source lumineuse se rapproche de la normale et plus la valeur de cette composante est importante.



La normale propre a l'un des vertex d'une sphère

➢ **Spéculaire**: Cette composante ajoute la prise en compte de la position du point de vue à celle de la source lumineuse pour produire l'effet de halo lumineux d'un matériau plutôt réfléchissant.



OpenGL permet de définir une couleur pour ces trois composantes au niveau de la source lumineuse ainsi qu'au niveau de l'objet lui même (plusieurs objets éclairés par la même source peuvent avoir des reflets différents). La couleur de réaction de l'objet pour les trois composantes est définie par l'introduction d'une notion de *Materiaux* appliquée aux objets. Il existe également une quatrième composante d'*Emission* qui est très peu utilisée. OpenGL prévoit également un calcul d'atténuation qui change l'influence des sources lumineuse par rapport à leurs distances de l'objet éclairé.

3.2.7 **Les buffers**

Les buffers sont également un élément important en OpenGL, ce sont des zones mémoires de la carte vidéo qui servent à stocker des pixels. On peut les voir comme des tableaux à deux dimensions. Il y en a 2 principaux, le *frame buffer* et le *depth buffer*. Il en existe d'autres comme le *stencil buffer* ou *l'accumulation buffer* qui sont utilisés uniquement dans des cas très particuliers.

➢ **Le frame buffer**: C'est dans ce buffer qu'OpenGL place le résultat des rendus effectués, c'est donc lui qui contient l'image affichée à l'écran. Il est généralement décomposé en deux parties: le *front buffer* et le *back buffer* qui permettent d'utiliser une technique appelée *double buffering*. Le double buffering permet d'éviter l'apparition progressive des différents éléments dessinés à l'écran en n'affichant pas le buffer de dessin courant mais la dernière image complète formée. On dessine donc dans le *back buffer* pendant que le *front buffer* est affiché et on échange les deux buffers (swapping) une fois la nouvelle image terminée. Seules les images complètes sont ainsi affichées.

➢ **Le depth buffer**: Le *depth buffer* ou *Z-buffer* sert a faire ce que l'on appel des tests de profondeur (*depth test*), une fonction effectuée automatiquement par OpenGL. Le test de profondeur permet que les nouvelles primitives dessinées ne cachent pas les primitives précédentes si celles-ci se trouvent devant les nouvelles.

Plus précisément, lorsque l'on dessine des primitives en OpenGL, elles sont immédiatement rendues dans le *frame buffer*, il n'y a donc aucun moyen de savoir, au moment du dessin d'une primitive, si il n'y a pas déjà eu une autre primitive de dessinée plus près du point de vue et qui la masquerait. Le *depth buffer* a exactement la même taille que le *frame buffer* et stocke pour chaque pixel du *frame buffer* la distance à laquelle il se trouve par rapport au point de vue. Au moment de dessiner une nouvelle primitive,

OpenGL compare la distance du nouveau pixel avec celle stockée dans le *depth buffer* et si cette distance est plus faible alors il dessine réellement le pixel.

3.2.8 **Les display lists**

Comme je l'ai expliqué, lorsque l'on dessine des primitives, que l'on applique des textures ou que l'on effectue des changements d'états en *OpenGL*, les actions sont immédiatement effectués et on en voit le résultat à l'écran. Les *display lists* permettent d'englober et de stocker, dans la carte graphique elle-même, une série d'opérations identifiées par un numéro pour une utilisation future. On peut ainsi préparer le rendu d'un objet puis l'utiliser au moment voulu. Cette fonctionnalité offre une grande efficacité car les transferts de données, de vertex en particulier, qui sont énormément coûteux en temps processeur, (particulièrement pour de gros modèles) ne sont effectués qu'une fois. De plus, certains traitements sont effectués en amont et les *display lists* sont ainsi pré compilées. Il suffit ensuite de passer à OpenGL le numéro de la *display list* à afficher pour provoquer son rendu. Une fois créés les *display lists* ne peuvent pas être modifiées et la seule solution est de les détruire pour en recréer d'autres

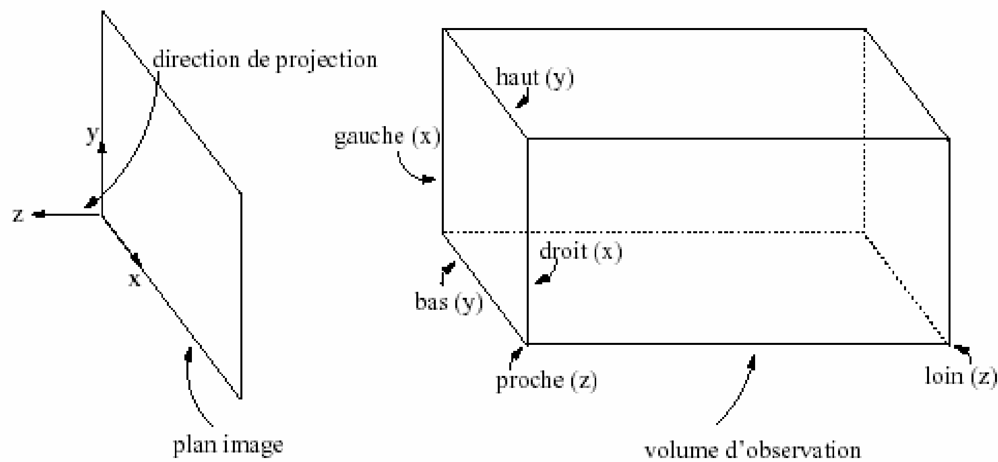
3.2.9 **Les matrices**

Pour toutes les transformations spatiales, OpenGL utilise un système de matrices très simples à manipuler. On peut leur appliquer des transformations simples tel que les rotations, translations et mise a l'échelle et également les empilées ce qui est très pratique lorsque l'on souhaite que les transformations effectues pour le dessin d'un objet n'affecte pas les rendus suivants. Il y en a deux principales:

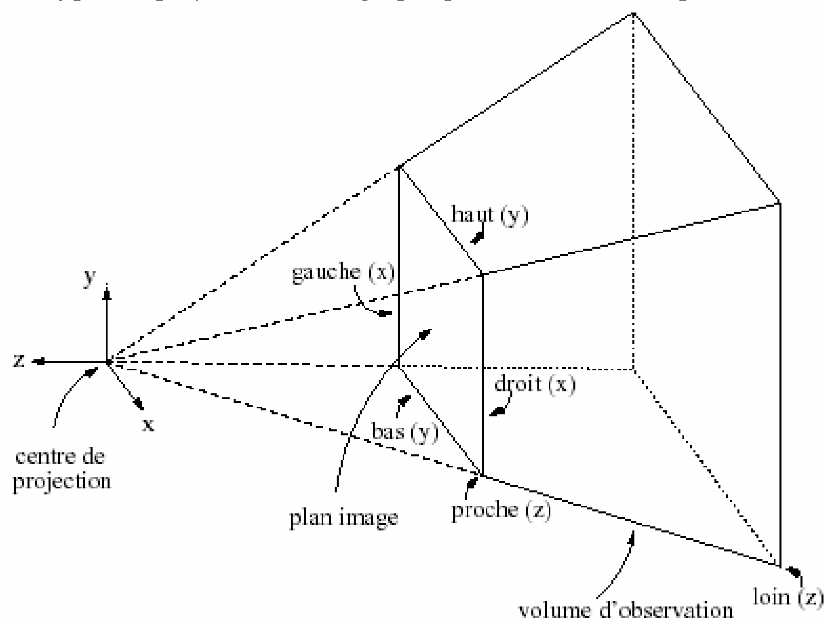
- **La matrice Modelview**: La matrice *Modelview* sert à définir la transformation à appliquer au repère dans lequel sont dessinés les objets avant leur rendu. A l'origine, ce repère se trouve au centre du point de vue. Pour effectuer une rotation du point de vue par exemple, il suffit d'appliquer une rotation dans le sens inverse à la matrice *Modelview* avant de commencer à rendre la scène.

- **La matrice Projection**: Elle sert à définir le type de projection à appliquer aux vertex lors du passage en deux dimensions. L'un des problèmes du dessin 3D est en effet qu'à un moment donné il faut transformer des coordonnés en trois dimensions en deux dimensions pour qu'un affichage sur un écran donne une impression de volume. Cette étape est appelée projection et la méthode de projection appliquée est donc définie dans cette matrice. Il y en a deux types: Orthographique et Perspective. La projection orthographique transforme tous les points orthogonalement au point de vue sans soucis de perspective alors que la projection Perspective est définie par une pyramide de vision (on appelle cela le *frustum*) avec une focale et un rapport horizontal/vertical.

Il existe une troisième matrice en OpenGL appelée *matrice de texture* (*Texture Matrix*) qui permet d'effectuer facilement des transformations sur les coordonnées de *mapping*. Cette matrice est très pratique pour faire tourner une texture sur un objet par exemple.



Les deux types de projection: Orthographique (en haut) et Perspective (en bas)



3.2.10 **Le blending**

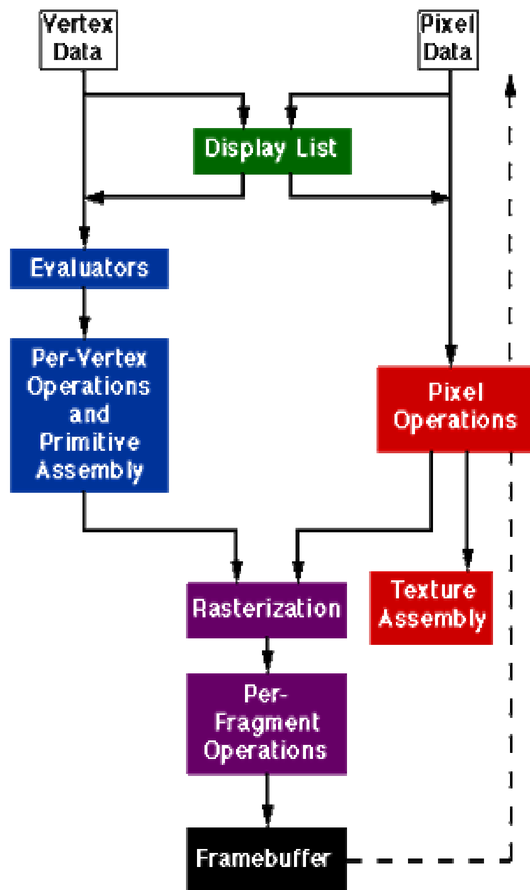
La fonction de *blending* (mélange en français) permet de choisir la façon dont les nouveaux pixels générés par un rendu de primitives seront combinés avec les pixels déjà présents dans le *framebuffer*. C'est cette technique qui permet par exemple d'obtenir des effets de transparence. Lorsque l'on définit une couleur en OpenGL, que ce soit dans une texture ou sur un vertex, on précise généralement quatre composantes: Rouge, Vert, Bleu et Alpha. On comprend parfaitement à quoi servent les trois premières et la quatrième trouve son intérêt justement lors de cette étape de *blending*. Elle indique l'importance de la couleur, en quelque sorte son niveau de transparence ou plus précisément d'opacité.

3.2.11 **Les extensions**

Comme je l'ai déjà indiqué, OpenGL possède un mécanisme d'extensions qui permet aux différents acteurs qui participent au développement de l'API (généralement des constructeurs de matériels ou de grosses entreprises de recherche) d'ajouter des fonctionnalités. L'ARB gère l'intégration de ces extensions au standard OpenGL mais chaque constructeur peut fournir ses propres extensions qu'il intègre à ses pilotes.

3.2.12 Le pipeline de rendu

Pour obtenir une image d'une scène 3D que l'on appelle donc un rendu, la suite des opérations effectuées par OpenGL est toujours la même. Cette suite d'opérations est ce que l'on appel le pipeline de rendu, l'image utilisée illustre le fait que l'on fasse entrer des vertex et des textures d'un coté du tuyau et qu'il en ressorte une image affichable de l'autre après application d'un nombre fini d'opérations. Quasiment toutes les fonctionnalités présentées sont réalisées à l'intérieur de ce pipeline qui est maintenant, avec l'évolution matérielle, entièrement réalisé par les cartes 3D.



1) En entrée, on trouve les vertex (coordonnées, normale, couleur etc...) et les textures transmises par le CPU et qui peuvent avoir été compilés dans une display list.

2) - Coté vertex, les Evaluators servent seulement à transformer des courbes paramétriques en segments, une fonction très peu utilisée et non accélérée sur les cartes grand publique. Les vertex passent donc tout d'abord par une étape de transformation où sont appliquées les matrices de transformation *Modelview* et *Projection* et où les vertex non visibles sont éliminés (clipping). C'est également ici que les calculs d'éclairage sont effectués.
- Coté pixels, les données de texture subissent une série de traitements avant d'être écrites en mémoire vidéo.

3) L'étape de *Rasterization* transforme une primitive (un triangle par exemple) en une série de *fragments*, c'est à dire en points prêts à recevoir une couleur d'une texture où à être combinés dans la *framebuffer*. Chaque fragment contient l'interpolation des données des vertex qui le forment comme une couleur ou une coordonnée de texture par exemple ainsi qu'une information de profondeur. Ce ne sont donc pas encore tout à fait des pixels.

4) L'étape de *Per-Fragment Operations* teste, pour chaque fragment, si celui-ci doit être affiché à partir des informations de profondeur (grâce au *depth buffer*) et de divers filtres comme le *stencil buffer* ou *l'alpha test*. Si le fragment est accepté, une couleur de texture lui est affectée (avec différents filtrages) et le fragment est copié ou combiné (*blending*) dans le *frame buffer*.

Les différentes étapes du pipeline de rendu d'OpenGL

3.2.13 Les évolutions matérielles

Dans les premières années des cartes 3D sur PC, les constructeurs ont tout d'abord cherché à intégrer un maximum de fonctions du pipeline de rendu à l'intérieur de leurs cartes afin d'en accélérer le traitement. Les cartes graphiques ont donc commencé à intégrer des processeurs dédiés au traitement de ces fonctions, la dernière en date étant l'étape de *transform'n lighting* (*Per-Vertex Operations*) qui a fait le succès de la geForce de nVidia. Le problème de cette approche, est qu'elle oblige le programmeur à utiliser les fonctions fixes intégrées aux cartes sans pouvoir intervenir dessus pour obtenir un résultat différent ou original.

Une carte 3D geForce4 de nVidia, véritable concentré de technologie

Depuis quelques années (trois ans environ) les constructeurs s'orientent donc vers une plus grande flexibilité du pipeline de rendu. Pour ce faire, ils proposent des moyens d'intervenir sur les principales étapes de ce pipeline en les programmant manuellement, c'est ce que l'on appel les *shaders*. Les *shaders* sont de deux types: les Vertex Shaders pour les opérations sur les vertex (c'est l'étape *Per-Vertex Operations*) et les Pixels Shaders pour les opérations sur les fragments (étape *Per-Fragment Operations*). En OpenGL, on parle plutôt de Vertex et de Fragment Programs. Dans un premier temps, ces *shaders* ont été programmés dans un langage assembleur très proche du matériel et depuis quelques mois commencent à apparaître des langages de plus haut niveau à la syntaxe proche de celle du C tels que le Cg de nVidia, le HLSL de Direct3D (qui est en fait une version du Cg spécifique à Direct3D) ou le GL2 Shading Language qui sera intégré a OpenGL 2.0.

Un exemple de scène tirée de 3D mark 2003 (Futuremark) qui montre le genre d'effet réalisable grâce aux *shaders*

- **Les Vertex Programs:**

Les *Vertex Programs* sont les premiers à être apparus et ils court-circuitent l'étape de *transform'n lighting* du pipeline OpenGL. Ces programmes opèrent par vertex et sont exécutés par la carte 3D si celle-ci les supportent ou par le CPU dans le cas contraire, le majorité des drivers acceptant d'effectuer cette émulation ce qui n'est pas le cas, nous le verrons plus loin pour les *Fragment Programs*. Ce qu'il faut bien comprendre, c'est que lorsque l'on active un *Vertex Program*, tous les vertex envoyés à OpenGL passent par ce programme et y sont traités.

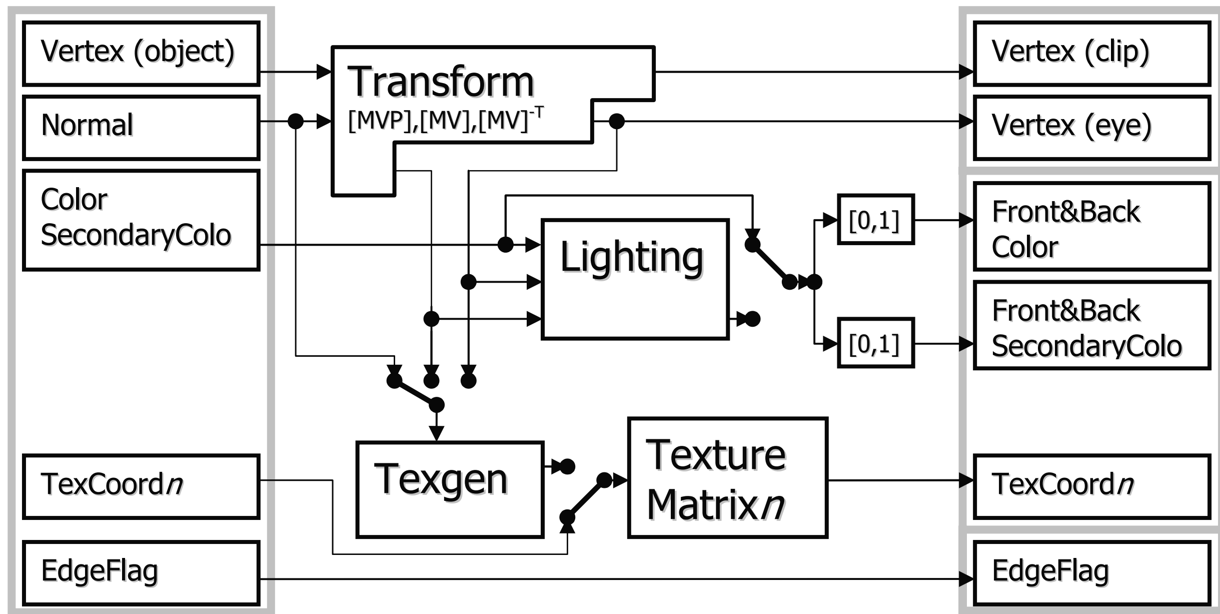


Schéma de principe des fonctions fixes de *transform'n lighting*. A gauche se trouvent les attributs stockés par vertex (position, normale, couleur etc...), au centre les différentes fonctions mises en jeu et à droite les attributs qui seront interpolés par fragment.

Les fonctions fixes sont donc remplacées par une unité de traitement programmable dont les entrées et les sorties sont généralisées. Les attributs de vertex n'ont plus de rôle défini, ce sont 4 floats qui peuvent servir à stocker n'importe quelle donnée utilisée dans le *Vertex Program*. Les attributs de sortie conservent tout de même, en plus des attributs génériques, des attributs standard qui seront passées aux fonctions fixes de *Rasterisation* puis, interpolées par *fragment*, à l'unité de traitement fixe des *fragments* ou à un *Fragment Program*. En plus viennent s'ajouter des paramètres dits uniformes communs à tous les vertex traités.

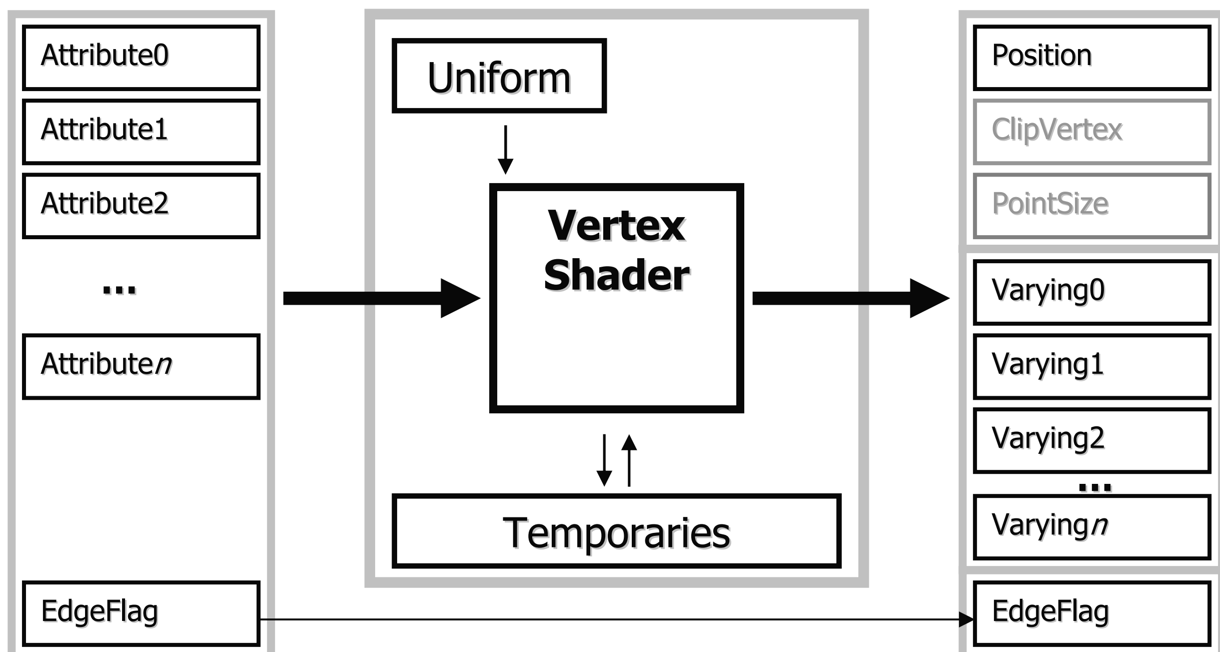


Schéma de principe de l'unité programmable de traitement des vertex, le *Vertex Shader*

Les *Vertex Programs* sont implémentés en OpenGL au travers d'extensions qui sont longtemps restés propriétaires et l'année dernière a été approuvée par l'ARB une extension appelée ARB_vertex_program qui fixe un standard pour la syntaxe et les fonctionnalités des *Vertex Programs*.

Les *Vertex Programs* permettent de manipuler principalement des vecteurs (4 floats) et la version actuelle fournit des opérations de base, tels que les additions, multiplications, produits scalaires et vectoriels. Par contre, ils ne supportent pas encore de structures de contrôle ni de sauts que seules les cartes de dernières génération peuvent gérer. Pour utiliser ces fonctionnalités sur des cartes telles que la geForce FX ou la radeon 9xxx de chez ATI, il faut donc passer par des extensions propriétaires telles que NV_vertex_program_2 (nVidia), en attendant OpenGL 2.0.

```
!!ARBvp1.0

ATTRIB pos = vertex.position;
PARAM  mat[4] = { state.matrix.mvp };

# Transform by concatenation of the
# MODELVIEW and PROJECTION matrices.
DP4    result.position.x, mat[0], pos;
DP4    result.position.y, mat[1], pos;
DP4    result.position.z, mat[2], pos;
DP4    result.position.w, mat[3], pos;

# Pass the primary color through w/o lighting.
MOV    result.color, vertex.color;

END
```

Un exemple simple de Vertex Program qui transforme les vertex avec les matrices *Modelview* et *Projection*. La première ligne indique le type de *shader* défini, les # désignent un commentaire, en haut, les déclarations désignent des registres dans lesquels sont copiés les valeurs affectées, l'opérateur *DP4* réalise un produit scalaire entre ses deux dernières opérandes et place le résultat dans la première. Le *MOV* effectue une copie de la deuxième opérande dans la première.

➢ Les Fragment Programs:

Les *Fragment Programs* sont donc l'équivalent des *Vertex Programs* mais par fragment cette fois et non plus par vertex. Ce type de *shader* est donc énormément plus coûteux en calcul puisque pour un triangle, il n'y a que 3 vertex mais énormément de fragment (leur nombre dépend de la résolution). De plus, cette fonction se trouve tout au bout du pipeline OpenGL et ne peut donc pas être émulée en software par le CPU si le matériel graphique ne la supporte pas. Le support des *Fragment Programs* via l'extension standard ARB_fragment_program est très récent et comme pour les *Vertex Program*, les constructeurs ont commencé par fournir des extensions propriétaires.

Chez nVidia l'ancêtre des *Fragment Programs* s'appelait les *Register Combiner* et existent depuis la riva TNT (sortie en 1998 chez nVidia). Les *Register Combiner* servaient à programmer la façon dont les textures étaient combinées lors de l'utilisation du *multi-texturing*, ils ont subis de multiples évolutions au fur et à mesure de l'évolution du matériel et sont devenus un outils très puissant pour le traitement et la combinaison de textures. Ils permettent de réaliser de nombreux effets et en particulier des calculs d'éclairage par fragment.

NVidia propose également des Texture Shader qui permettent de contrôler l'application des textures dans les différentes unités de texturing (les cartes 3D en comptent une par texture applicable via le *multi-texturing*) avant leur combinaison (donc en amont des *Register Combiners*).

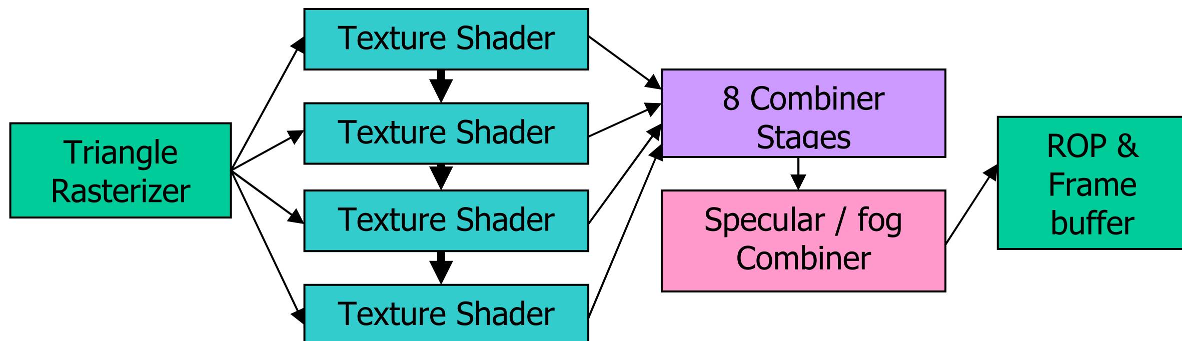


Schéma de principe des fonctions programmables d'application des textures proposées par nVidia sur les cartes à partir de la gamme des geForce 3.

Les *Fragment Programs* regroupent ces différentes fonctionnalités dans une architecture plus simple et beaucoup plus flexible puisqu'ils proposent de vrais traitements par fragment et non plus des traitements globaux appliquées aux textures. Ils permettent d'écrire, toujours dans un langage assembleur, des algorithmes d'éclairage, de plaquage ou de mélange de textures et bien d'autres avec comme seule limite, la puissance de traitement du matériel.

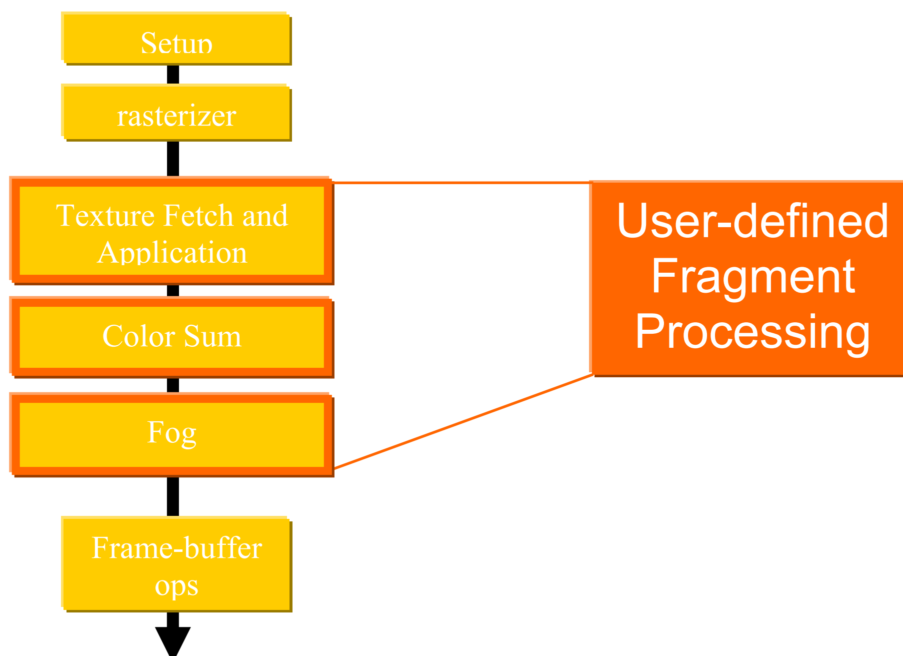


Schéma de principe détaillé de la parte *Per-Fragment Operations* du pipeline OpenGL et les étapes que remplacent les *Fragment Programs*. L'étape *Fog* sert à l'application d'une couleur simulant un effet de brouillard, je ne l'ai pas détaillée car elle n'est que peu utilisée.

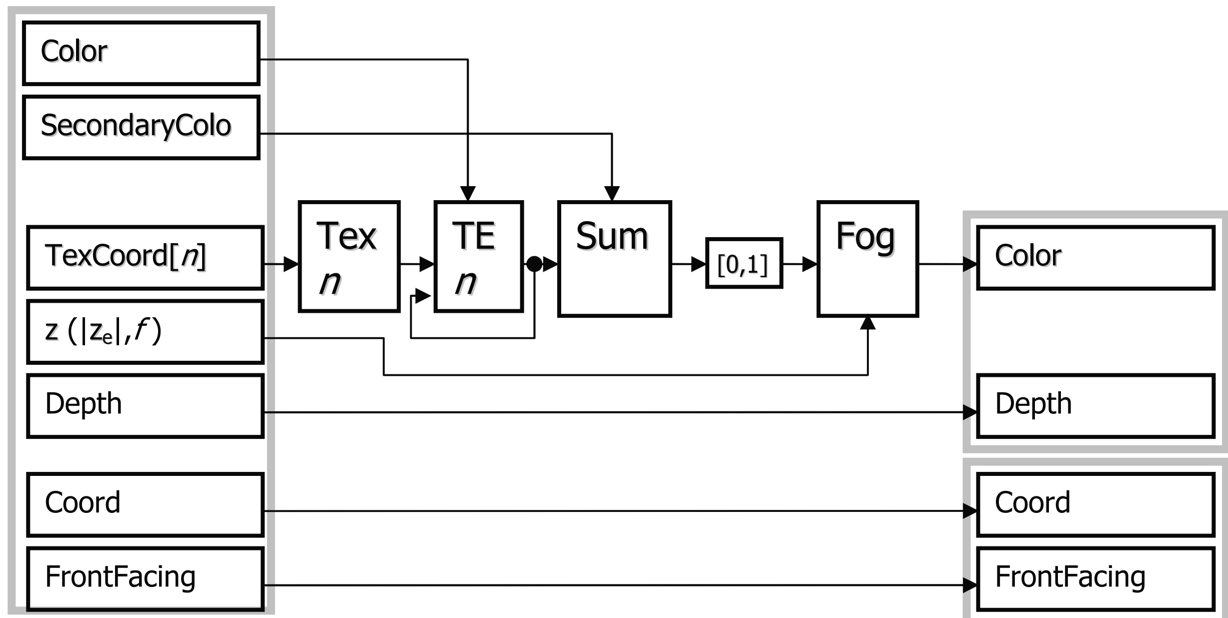


Schéma de principe du fonctionnement des fonctions fixes de traitement des fragments. Les entrées sont specifiques et les opérations sont effectuées a la suite avec très peu de possibilité d'intervenir.

Comme pour les *Vertex Programs*, les paramètres autrefois fixes par fragment sont généralisés. Les instructions de bases proposées par l'extension standard fournissent des opérateurs proches de ceux proposés par les *Vertex Programs* avec en plus les opérations d'accès aux textures et il faudra ici aussi attendre les prochaines versions et faire appel aux extensions propriétaires pour profiter des structures conditionnelles (des comparaisons simples sont déjà présentes) ou des sauts dans les *Fragment Programs*.

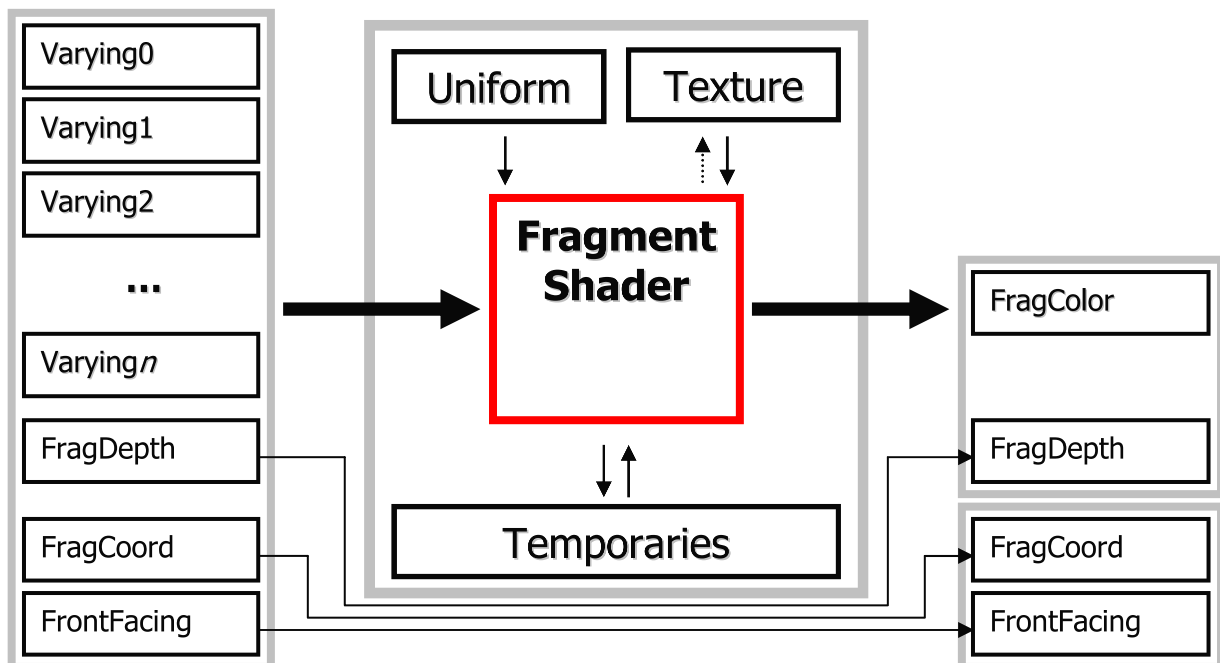
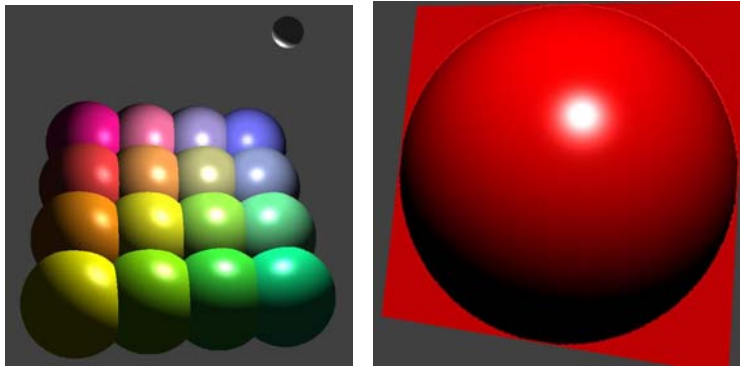


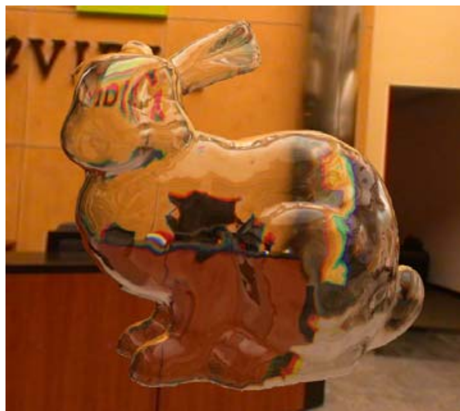
Schéma de principe des *Fragment Shaders*, les entrées sont généralisées et proviennent des sorties interpolées des *Vertex Programs* et les sorties par fragment sont standards et fixes puisque destinées à l'unité d'affichage dans le *framebuffer*. On trouve également en entrée des paramètres uniformes (constants) ainsi que des textures.

Le changement majeur qu'apporte le travail par fragment et non plus par vertex se trouve au niveau de la précision des effets appliqués aux objets. Les calculs d'éclairage par exemple, réalisés par fragment plutôt qu'interpolés entre les vertex sont ainsi beaucoup plus précis et permettent même dans certain cas de réduire le nombre de vertex nécessaire à la définition d'un objet sans perdre pour autant en réalisme. Les textures prennent ainsi un nouveau rôle qui n'est plus seulement de définir des couleurs pour un objet mais aussi de transmettre des informations par fragment au *Fragment Program*. Elles peuvent par exemple servir à stocker des normales (*normalmap*) pour divers calculs d'éclairage qui donneront du relief à une surface qui autrement aurait été plate.



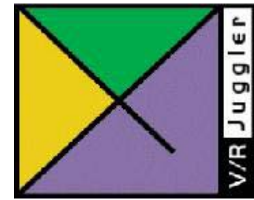
Un exemple de calcul d'éclairage spéculaire sur des plans (oui ce sont des plans !) grâce a l'utilisation d'une *normalmap* et des *Fragment Programs*. On voit bien ici le gain obtenu au niveau du nombre de vertex nécessaires.

Les *shaders* sont donc une véritable révolution dans la façon de programmer un rendu graphique et ils ouvrent énormément de perspectives tant les applications possibles de cette technologie sont vastes. La puissance de calcul des cartes 3D ne fait que croître d'année en année et la possibilité offerte de leur déléguer une grande partie du travail autrefois réservée au processeur central est extrêmement intéressante. En déchargent ainsi le processeur central, il est possible du lui affecter de nouvelles taches ou des calculs plus importants sans que le rendu graphique n'en pâtisse. C'est une possibilité très intéressante dans le domaine de la simulation scientifique en temps réelle par exemple qui demande une masse importante de calculs ainsi qu'un rendu graphique évolué. Il s'agit donc d'un apport très important pour la réalité virtuelle.



Un calcul de réfraction réalisé entièrement par la carte 3D grâce aux *shaders*.

3.3 VR-Juggler



VR-Juggler est une plateforme de développement d'applications de réalité virtuelle multi-plateformes conçue pour s'exécuter aussi bien sur des stations SGI que sur des PC par exemple. C'est un environnement logiciel Open Source créé par l'université de l'IOWA qui a l'avantage de proposer une abstraction totale du matériel utilisé aussi bien au niveau des périphériques d'entrée que de sortie. Cette abstraction assure une portabilité et une flexibilité totale des applications qui s'adaptent à une très grande quantité de configurations de VR qui vont d'un simple PC de bureau à un environnement CAVE en passant par tous types d'installations spécifiques. Une application écrite avec VR-Juggler est donc totalement indépendante des périphériques, de la plateforme matérielle et logicielle ainsi que de l'environnement de réalité virtuelle. VR-Juggler est également une véritable bibliothèque de programmation puisqu'il fournit toute une série d'objets de base indispensable à la programmation d'environnement de réalité virtuelle tels que les *Matrices*, les *Vecteurs* où les *Quaternions*.

3.2.1 **L'architecture**

L'architecture de VR-Juggler a été conçue pour être la plus flexible et extensible possible et s'articule autour d'une série d'objets abstraits spécialisés dans des domaines particuliers. Ces objets sont étendus par héritage pour s'adapter à tout type d'API graphique, de périphérique ou de système d'exploitation. Ce type d'architecture modulaire est appelé *Microkernel*.

Cette architecture se base principalement sur des objets appelés *Managers* qui sont de deux types: les *DrawManagers* et les *InputMangers* respectivement chargés de l'abstraction des environnements de visualisation et des périphériques d'entrées.

La flexibilité de l'architecture permet également une reconfiguration en temps réelle de l'environnement (sans redémarrage de l'application), une fonctionnalité implémentée grâce à un objet appelé *ConfigManager*.

Pour écrire une application VR-Juggler, on implémente les méthodes d'un objet spécialisé dans l'API graphique utilisée (OpenGL ou divers *scene graphs*, des surcouches d'OpenGL) héritier de l'objet de base *App*. Cet objet prépare automatiquement l'environnement graphique pour s'adapter au dispositif de visualisation utilisé.

La configuration de l'environnement est assurée par un mécanisme de fichiers de configuration qui permet le paramétrage de toutes les interfaces utilisées

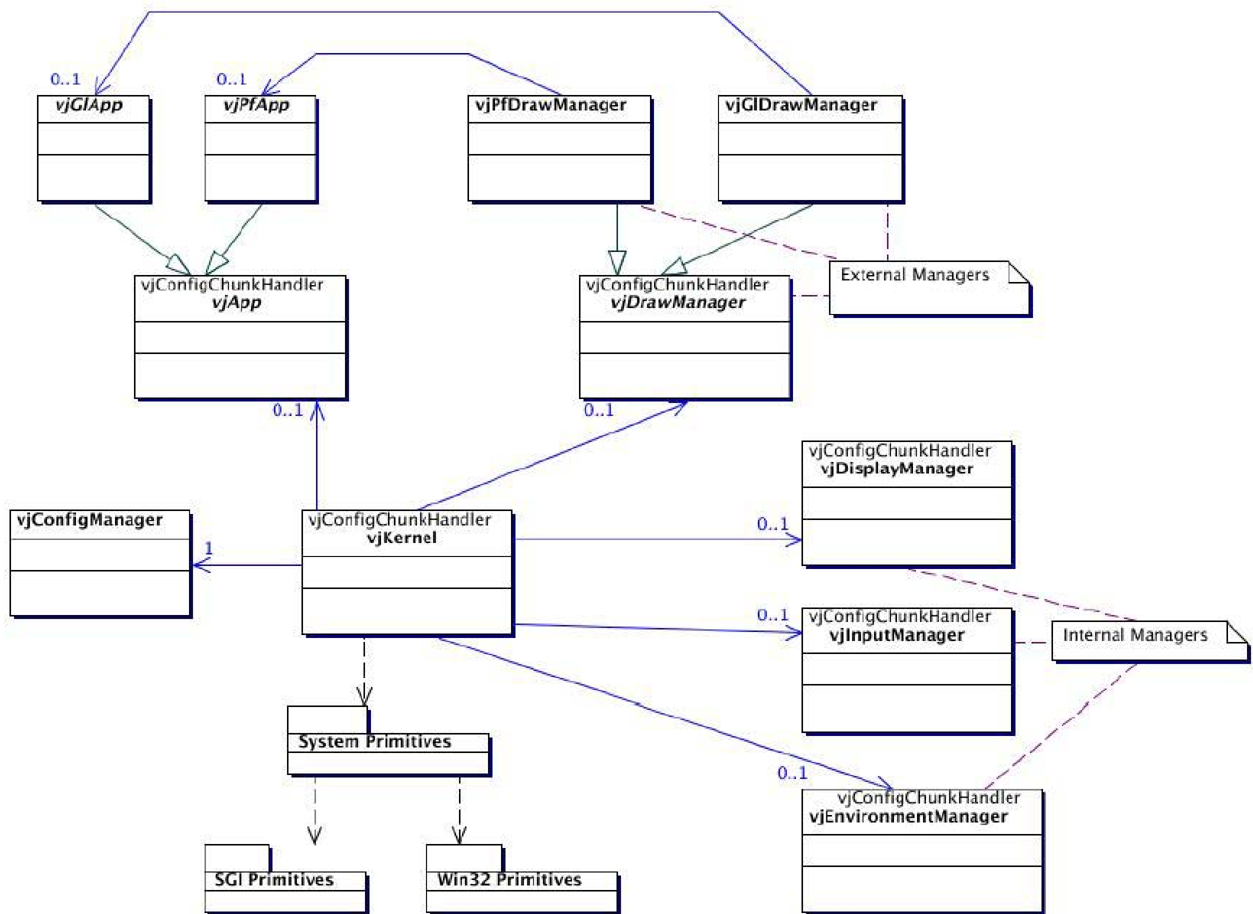


Diagramme structurel de l'architecture *Microkernel* de VR-Juggler

3.2.2 **La boucle de dessin**

VR-Juggler fixe dans la classe vjApp, une série de méthodes standard qu'il faut ré-implémenter pour effectuer chaque étape d'une simulation de VR. Ces méthodes sont toujours appelées dans le même ordre et permettent à VR-Juggler d'effectuer les paramétrages nécessaires à chaque étape.

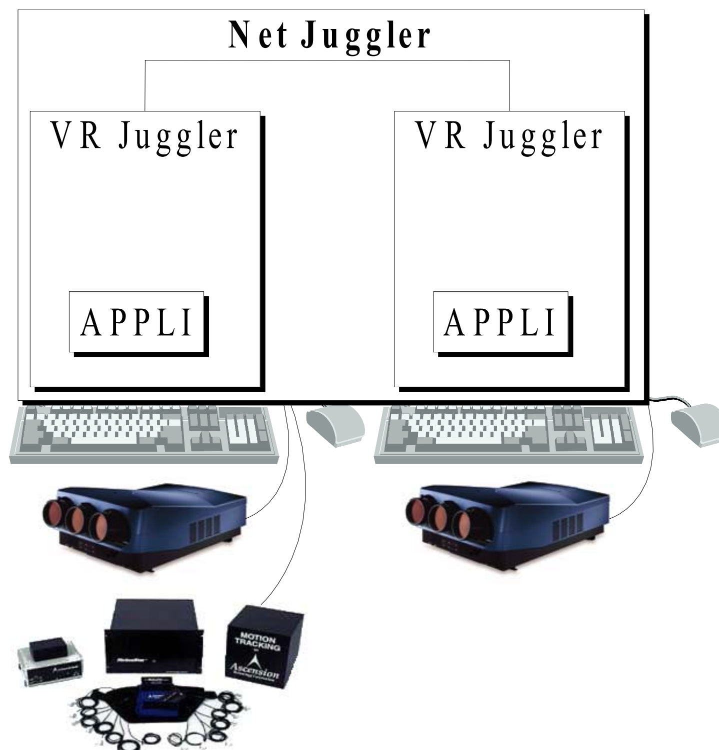
```
while (drawing) {
     preFrame();  //Calculs de simulation
    draw();        //Rendu
     intraFrame();
    sync();        //Synchronisation des affichages
     postFrame();

    UpdateTrackers();  //MAJ des périphériques d'entrée
}
```

La boucle de dessin de VR-Juggler

3.4 Net-Juggler

Net-Juggler est une surcouche de *VR-Juggler* développée au LIFO et permettant la distribution des périphériques d'entrée sur une grappe de machines. Elle se base sur le mécanisme de fichiers de configuration de *VR-Juggler* pour introduire une notion de machine qui permet de paramétrer totalement la localisation des périphériques. Cette distribution est totalement transparente dans l'utilisation de VR-Juggler et le passage d'une application *VR-Juggler* à *Net-Juggler* est extrêmement aisé.

Net-Juggler utilise en fait MPI pour renvoyer à chaque machine l'état des différentes *devices* connectées.



3.5 MPI

MPI (*Message Passing Interface*) est une bibliothèque de communication par passage de message. Elle permet d'effectuer des communications entre les différents noeuds d'un cluster et est destinée à la parallélisation de calculs. Son principe est de lancer l'exécution d'un même programme parallèle sur les déférents noeuds de façon centralisée grâce à l'utilisation de connexions distantes. Il en existe différentes implémentations et celles qui sont utilisées au LIFO s'appellent MPICH et lam-MPI.

Chapitre 4

Mon projet de stage

4.1 Description générale

L'objectif premier de mon stage était, comme je l'ai expliqué dans l'introduction, de réaliser une démonstration graphique destinée à montrer, de manière ludique, les capacités offertes par l'environnement de réalité virtuelle développé au LIFO. Cette démo, principalement destinée au grand public, se base sur un projet de simulation de vie artificielle que j'ai réalisé en début d'année dans le cadre du cours de *Base de la programmation*.

Une grande liberté m'a été laissée dans la mise en place de cette démo et ma volonté a tout de suite été d'en apprendre le plus possible sur le domaine très large qu'est la Réalité Virtuelle et d'en aborder un maximum d'aspects plutôt que de me focaliser sur un point particulier. J'ai ainsi effectué un parcours en largeur de ce domaine en en abordant de nombreux aspects qui vont du pré-rendu au rendu en passant par la parallélisation et l'interfaçage avec l'utilisateur. J'ai essayé à chaque fois d'analyser les solutions existantes et d'en apporter des originales avec toujours la volonté de réaliser un travail générique et réutilisable.

Cette approche m'a permise d'acquérir une bonne compréhension globale du domaine de la Réalité Virtuelle et d'en appréhender les problèmes.

J'ai effectué mon stage en parallèle avec Bertrand Greslier, un autre étudiant d'IUT qui lui a travaillé sur les périphériques d'entrée et la détection de collision. Nous avons collaboré sur de nombreux points communs à nos deux projets et plus particulièrement sur les structures de donnée.

J'ai également travaillé avec un étudiant de DEA qui s'occupait de la parallélisation d'une simulation de fluide et deux étudiants de Maîtrise qui travaillaient sur l'analyse de données médicales et leur rendu graphique ainsi que sur l'affichage de gros volumes de donnée.

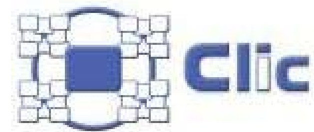
4.2 L'installation d'une grappe de PC

La première chose que nous aillons faite avec Bertrand lorsque le stage a débuté, a été d'installer nous même la grappe de PC (*cluster* en anglais) sur laquelle nous allions travailler. Ce premier travail nous a permis bien sûr d'acquérir une expérience dans l'installation d'une grappe mais également de nous familiariser avec les outils spécifiques à son administration.

4.2.1 L'architecture matérielle

Notre grappe était constituée de quatre PC standard équipés d'AMD Athlon XP 1800+, de 256Mo de ram, d'une carte graphique ATI rage 128 et reliés par un simple réseau Ethernet via un rooter. Il s'agit donc d'une installation qui n'a rien de spécifique à un environnement de réalité virtuelle mais qui est largement suffisante pour la mise en place d'un petit *cluster* destiné à faire des essais de parallélisation et à permettre à Bertrand de travailler avec ses périphériques.

4.2.2 La distribution *Clic*



La distribution Linux que nous avons déployée sur ce cluster s'appelle *Clic*. Il s'agit en fait d'une version particulière de la distribution Mandrake 9.0 (développée par la société française Mandrakesoft) spécialement conçue pour la mise en place de grappes de machines. L'avantage principal de *Clic* est la procédure d'installation spécifique aux *clusters* qu'elle propose. Cette procédure est quasiment totalement automatisée et permet (quand tout se passe bien !) d'installer très rapidement et avec un minimum d'effort une grappe de machines fonctionnelle (configuration réseau, serveurs, comptes *NIS*, etc.). L'autre avantage de *Clic* est qu'elle intègre tous les outils d'administration et de maintenance d'une grappe ainsi que les logiciels les plus couramment utilisés sur ce genre de plateforme avec entre autre une version de NetJuggler.

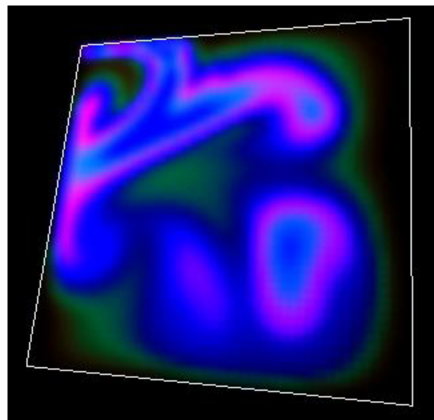
Dans la pratique, la mise en place d'un cluster, même sous Clic, pose toujours un certain nombre de problèmes et nous avons pu profiter pour notre déploiement de l'expérience acquise par Julien et Sylvain, les deux autres stagiaires avec qui nous avons travaillé pendant ces deux mois, qui avaient déjà expérimenté et documenté une installation.

Ayant à l'avance expérimenté chez moi, l'installation d'un cluster avec une distribution standard (RedHat), j'ai pu apprécier les grandes facilitées offertes par ce système et particulièrement en ce qui concerne la mise en place de comptes *NIS*(gestion centralisée de comptes utilisateurs) et d'un serveur DHCP.

4.3 **La simulation d'écoulement de fluide**

Il s'agit en fait du sujet de stage de Sylvain, le stagiaire de DEA. Les simulations d'écoulement de fluide demandent énormément de puissance de calcul et son but est de trouver une méthode pour les parallèliser afin de profiter de la puissance offerte par une grappe de PC. Si je me suis intéressé à cette simulation de fluide, c'est parce qu'elle est similaire en de nombreux points aux jeux cellulaires que je cherche à paralléliser. Cette collaboration nous a été mutuellement profitable puisque j'ai pu bénéficier des connaissances théoriques et mathématiques de Sylvain pour comprendre le fonctionnement même de la simulation et j'ai pu lui faire profiter de mon expérience en programmation pure (C++) et en OpenGL pour implémenter une version 3D et son rendu graphique.

Une telle simulation permet de rendre de nombreux phénomènes naturels comme un déplacement de liquide, de la fumée ou des nuages avec énormément de réalisme.



Notre rendu de fluide en deux dimensions

4.3.1 **Le principe**

Nous sommes en fait partie d'un algorithme fournit par Sylvain fonctionnant uniquement en deux dimensions. Il s'agit en fait de l'application des équations de *Navier-Stokes*, à la base de la mécanique des fluides, à partir des travaux de *Jos Stam* sur leur application dans les jeux vidéo.

Le principe de l'algorithme est en fait assez simple, il travaille sur deux grilles à deux dimensions contenant l'une un champ de densités (scalaires) et l'autre un champ de vecteurs forces. Deux actions sont possibles sur la simulation: L'ajout de densité et l'ajout de force. On visionne les densités et lorsque l'on ajoute des forces, les densités sont affectées en conséquence et se trouvent déplacées. Pour simuler un écoulement de fluide, avec ses tourbillons et ses trajectoires particulières, deux *solveurs* sont mis en jeux: Un *solveur* de densité et un *solveur* de vitesse qui servent, à chaque étape de simulation, à calculer le nouvel état des deux grilles en fonction de leur état précédent. Pour paramétrer la simulation, il est possible d'agir sur 3 variables: La vitesse, la diffusion et la viscosité du fluide.

4.3.2 **Le passage en 3D**

Le passage en 3D de l'algorithme n'a pas posé de difficulté majeure, il nous a juste fallu ajouter une troisième dimension aux deux grilles ainsi qu'une composante supplémentaire aux vecteurs force. Il a donc également fallu modifier les différentes étapes de l'algorithme en conséquence ce qui a occasionné un surplus de code d'environ 50%.

4.3.3 La visualisation de données scientifiques

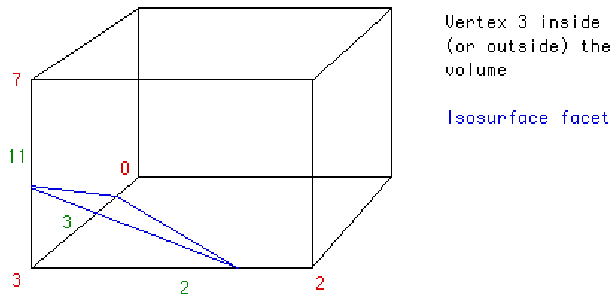
Pour le rendu graphique, j'ai commencé par rechercher les méthodes couramment utilisées pour la visualisation scientifique de champs de scalaires comme c'est le cas ici. Le problème qui se pose est comment passer d'un tableau tridimensionnel contenant les valeurs de densité en chaque point à une représentation graphique permettant de l'observer. J'ai ainsi découvert deux grandes techniques comportant chacune leurs avantages et leurs inconvénients.

- **La détermination de surface**:

Cette approche, basée sur l'analyse des données, consiste à déterminer une surface appelée *isosurface*, à partir du champ de scalaire tridimensionnel. On fixe pour cela une valeur de seuil (une densité critique) qui déterminera la limite entre le volume rendu (ici le fluide) et le vide. C'est donc cette seule surface qui sera représentée après polygonisation lors de la visualisation. Il existe plusieurs techniques pour créer cette surface polygonisée et la plus répandue s'appelle les *Marching Cubes*. Cette technique, dans sa version la plus simple, consiste à partitionner l'espace en un nombre fini de cubes contigus. Les angles de ces cubes sont pondérés avec les valeurs des cases correspondantes du champ de scalaire et l'algorithme de polygonisation est le suivant:

Pour chaque cube:

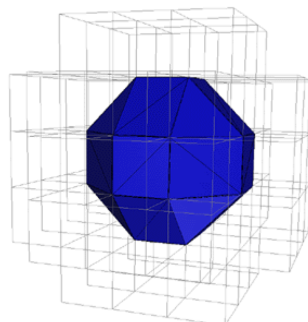
1) On commence par tester si les angles se trouvent à l'intérieur où a l'extérieur de la surface. Pour cela on compare leur pondération avec la valeur de seuil, si elle est inférieure c'est que l'angle se trouve à l'extérieur sinon c'est qu'il est à l'intérieur.



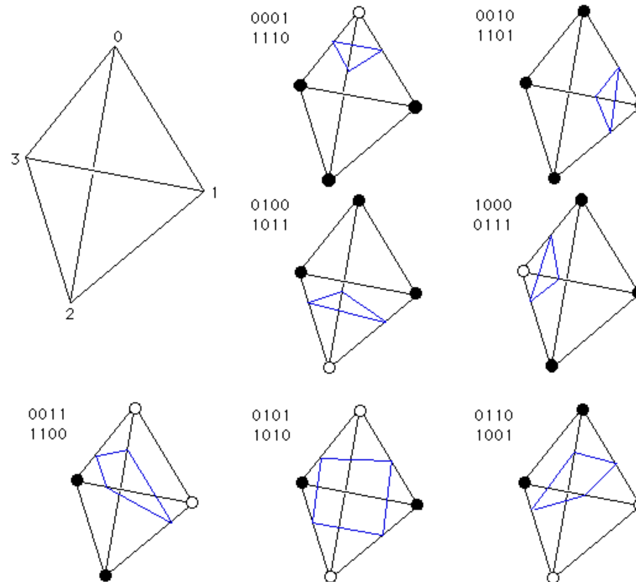
2) On compare ensuite chaque angle avec ses trois voisins et si ils ont des états différents (à l'intérieur où a l'extérieur de la surface) c'est que la surface coupe l'arête formée par les deux angles.

3) Pour chaque arête coupée, on détermine simplement la position du point d'intersection par interpolation linéaire par rapport aux pondérations des deux angles.

4) Pour finir, on forme des triangles à partir points d'intersection créés.



Un cube peut contenir au maximum 5 triangles et il existe des méthodes basées sur des tables d'indices permettant d'optimiser grandement cet algorithme mais l'idée générale est celle ci. Il existe également des techniques dérivées basées non plus sur des cubes mais sur des tétraèdres par exemple qui offrent d'avantage de précision et des cas d'intersection plus simples.



Les 8 cas d'intersection d'une *isosurface* avec un tétraèdre

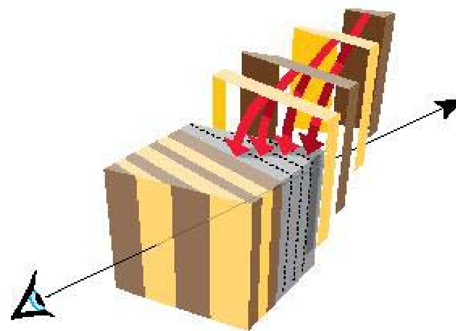
L'avantage de cette méthode est qu'elle permet de créer des surfaces nettes extrêmement précises, s'intégrant facilement dans tout environnement de VR et pouvant êtres éclairées facilement. Le problème est qu'elle demande un nombre très important de calculs et est de ce fait difficilement applicable à un champ de scalaire animé. Il faut en effet recalculer la surface à chaque image ce qui s'avère beaucoup trop coûteux pour être réalisé en temps réel. De plus, elle est relativement peu adaptée à la visualisation de fluides qui ne sont généralement pas uniquement caractérisées par une surface, mais plutôt par des zones plus ou moins denses.

Pierre, l'étudiant de maîtrise qui effectuait également son stage au LIFO a développé une autre technique de polygonisation d'un champ de scalaires par détermination de surface. Son but était l'affichage de la structure d'un os issue d'un scan tridimensionnel réalisé par un appareil d'imagerie médicale. Son approche a été différente puisqu'il crée des surfaces en se basant sur une forme cubique, il obtient ainsi un rendu ressemblent un peu à un *légo* quand on y regarde de trop près mais nécessitant moins de calculs que les *marching cubes*. Combiné à un calcul d'éclairage, ce rendu semble donné également de très bons résultats.

➢ **Le Volume Rendering**:

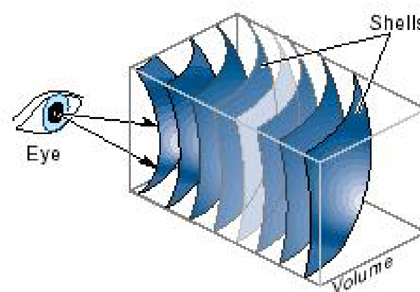
L'autre approche appelée *volume rendering* est une technique de visualisation utilisée pour afficher des données tridimensionnelles directement sans avoir à déterminer de surface ni à former de polygone. Elle se base sur l'utilisation intensive des textures pour afficher directement les informations issues d'un tableau tridimensionnel sans analyse préalable des données. Encore une fois, il existe plusieurs approches pour mettre en place cette technique mais le principe général reste globalement le même.

Cette technique consiste à former, à partir du champ de scalaires 3D, une texture 3D dont on détermine simplement les couleurs (RGBA, la composante alpha ayant son importance) à partir des scalaires correspondants dans le tableau de données. L'affichage de cette texture 3D est assuré par une série de plans disposés parallèlement au point de vue et dont on calcul les coordonnées de mapping de façon à répartir correctement les données dans l'espace. Ces plans sont rendus de l'arrière vers l'avant et combinés par une méthode de *blending* pour laisser apparaître de la transparence. Ce sont en quelques sortes des tranches découpées dans la texture 3D (appelée champ de *voxel*) que l'on affiche les une derrière les autre pour en recréer le volume. Les plans d'affichage restent toujours fixes et pour faire tourner un volume par exemple, ce sont les coordonnées de mappping de chaque plan que l'on modifie (ce qui est relativement simple avec l'utilisation de la matrice de texture), on fait ainsi en quelque sorte tourner la texture à l'intérieur de l'empilement de plans. Plus le nombre de plans affichés est important et plus le rendu sera précis.



Le découpage en tranches d'une texture 3D pour rendre un volume

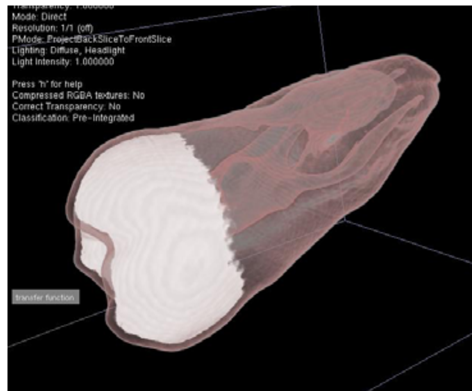
Une technique dérivée consiste à afficher non plus des plans mais plutôt des paraboles parallèles au point de vue afin de compenser la déformation due à la projection. Cette technique donne donc de meilleurs résultats mais est un peu plus complexe à mettre en oeuvre.



Le découpage d'une texture 3D en "coquilles" paraboliques

L'avantage de cette technique est qu'elle nécessite le rendu de très peu de polygones en comparaison des techniques de détermination de surfaces. De plus, elle exploite les capacités de *mapping* et surtout de *blending* des cartes 3D et ne demande aucun calcul sur les données issues du champ de scalaires qui est directement transformé en texture. Elle permet également de rendre de façon correcte des zones de densités différentes et est donc très bien adaptée à la visualisation de fluide. Le principal problème de cette technique est la difficulté d'intégration dans un environnement tridimensionnel dû à la

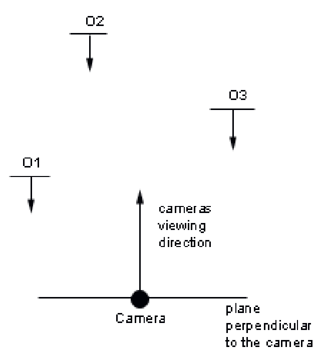
nécessiter de rendre toujours une série de plans parallèle au point de vue. Le deuxième problème est que lorsque l'on ne détermine pas de surface, la compréhension des données visualisées peu parfois être compliquée car non synthétiques et confuses. Il est de plus très difficile d'effectuer un calcul d'éclairage sur le rendu à cause, encore une fois, de l'absence de surface et également parce qu'il doit se faire par *fragment*. Il est possible de combiner un rendu volumique à une détermination de surface mais cela se révèle extrêmement compliqué à mettre en oeuvre.

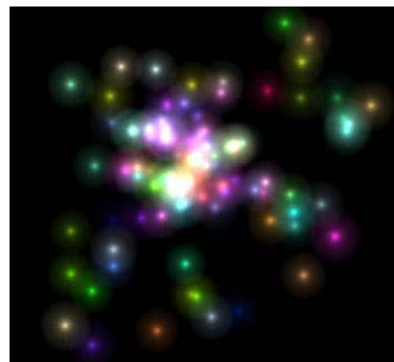


Un rendu de dent utilisant une combinaison de *volume rendering* avec une détermination de surface

4.3.4 **Ma solution**

J'ai en fait cherché à mettre en oeuvre une technique la plus simple possible, donnant un résultat visuel satisfaisant et s'intégrant facilement dans tout environnement de réalité virtuelle. La solution que j'ai adoptée est relativement différente des deux que je viens de présenter et est basée sur la technique dite des *sprites* ou du *billboarding*. Un *sprite* est en fait un plan faisant toujours face au point de vue, cette technique a été longtemps employée dans les jeux vidéo pour rendre des personnages ou des éléments de décors en utilisant simplement une image afin d'économiser des polygones. La technique des *sprites* est maintenant principalement employée pour le rendu de particules qui permettent de simuler des traînés de fumée, des flammes ou divers effets de lumière.

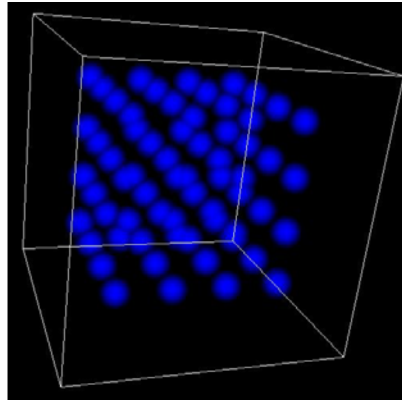




Le principe du *billboard* ou *sprite* et un rendu de particules multicolores

Je me suis aperçu que cette technique s'adaptait très bien à la visualisation de données scientifiques car elle permet de dessiner directement un champ de scalaires dans l'espace, à chaque case du tableau tridimensionnel correspondant une particule. L'avantage est que cette technique permet d'orienter l'ensemble des points comme tout autre objet tridimensionnel, les *sprites*, qui font toujours face à l'utilisateur, restant visibles. Il est ainsi

possible de ne rendre un *sprite* seulement si le scalaire correspondant est supérieur ou inférieur a une valeur de seuil, ce qui revient un peu à effectuer une détermination de surface. On peut également rendre tous les *sprites* avec des couleurs et surtout des composantes alpha (déterminant leur visibilité) proportionnelles au scalaire affiché. On utilise alors une technique de *blending* pour combiner correctement les *sprites* superposés.



Mon rendu d'un tableau de 4x4x4 scalaires par des *sprites* texturés

J'ai implémenté cette technique à l'aide d'une extension propriétaire développée par nVidia et appelée NV_POINT_SPRITE. Cette extension, uniquement disponible sur les cartes graphiques nVidia à partir de la geForce 4, permet de faire rendre des *sprites* directement par la carte graphique. Cela évite d'avoir à orienter chaque *sprite* pour qu'il face toujours face au point de vue. Il suffit d'envoyer un vertex représentant la position du *sprite* à OpenGL et la carte se charge de dessiner le plan, de l'orienter et de lui appliquer une texture. Cette extension permet donc, en plus d'économiser en calcul pour orienter les particules, d'économiser en vertex passés à la carte puisque leur nombre se trouve ainsi divisé par 6 (un vertex à la place de 2 triangles formés de 3 vertex).

Pour l'obtention du rendu final, plusieurs étapes sont nécessaires:

1) L'activation de l'extension NV_POINT_SPRITE, du *texturing* et désactivation du test de profondeur:

```
glEnable( GL_POINT_SPRITE_NV );
glEnable(GL_TEXTURE_2D);
glDisable(GL_DEPTH_TEST);
```

2) La l'activation et la détermination de la méthode de *blending*:

```
glEnable( GL_BLEND );

glBlendFunc( GL_SRC_ALPHA , GL_ONE);  //On multiplie les composantes RGB des
nouveaux fragments par leur composante alpha et on les ajoutes aux composantes
RGBA multipliées par 1 du fragment déjà présent.
```

3) Le paramétrage des sprites (taille et technique de *mapping*):

```
float maxSize = 0.0f;
glGetFloatv( GL_POINT_SIZE_MAX_EXT, &maxSize );
glPointParameterfEXT( GL_POINT_SIZE_MIN_EXT, 1.0f );
glPointParameterfEXT( GL_POINT_SIZE_MAX_EXT, maxSize );

glTexEnvf( GL_POINT_SPRITE_NV, GL_COORD_REPLACE_NV, GL_TRUE );
glPointSize( maxSize-(((Nx+Ny+Nz)/3.0)*maxSize/maxi) );
```

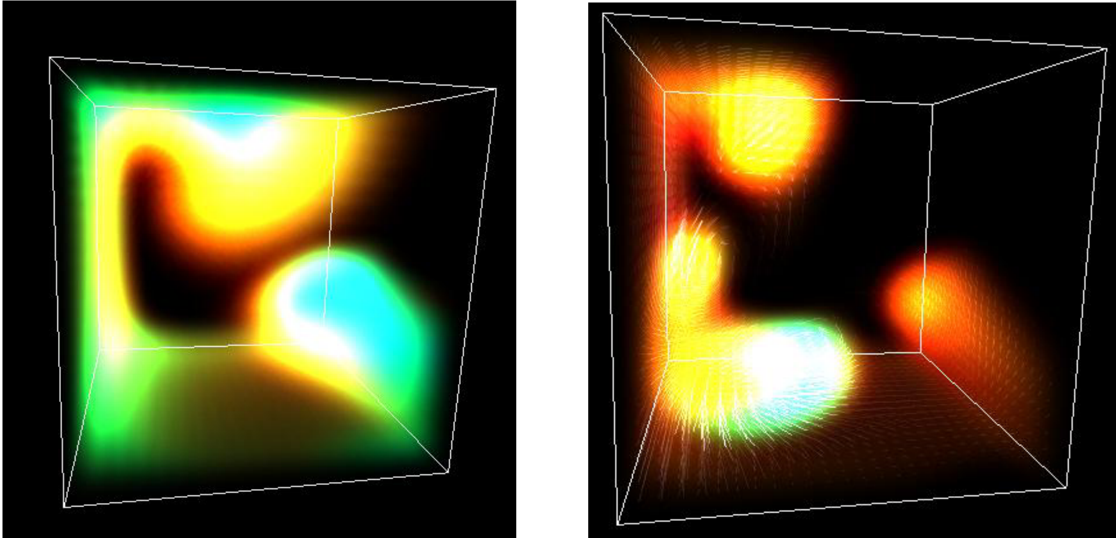
4) Le chargement de la texture et son activation:

J'ai utilisé pour cela des classes de gestion de textures et de matériaux que j'ai créée et intégrée a l'architecture globale dont je parle plus loin.

```
/*    TextureData texh;
      Material matSprite;
*/
texh.load("common/particle.tga");  //Chargement des données de texture
matSprite.bindTex(0, texh);        //Envoi de la texture a OpenGL
```

5) A chaque rafraîchissement de l'affichage, après calcul de la simulation, le dessin des sprites eux mêmes:

```
/*    int Nx, Ny, Nz : Dimensions des tableaux contenant les données de
simulation (densités et vecteurs force)
      float hx, hy, hz : Permet de transformer les positions entières
dans le tableau en positions réelles pour le dessin.
      dens : tableau des densités
*/
//Transformation des coordonnées x, y, z en décalage dans le tableau C
#define IX(i,j,k) ( (i)+(Nx+2)*(j)+(Nx+2)*(Ny+2)*(k) )

glBegin( GL_POINTS );    //Début du rendu de points

  for(k=1; k<Nz; k++){  //Parcours du tableau de densités
    z = (k)*hz-1.0;
    for ( i=1 ; i<Nx ; i++ ) {
      x = (i)*hx-1.0;
      for ( j=1 ; j<Ny ; j++ ) {
        y = (j)*hy-1.0;
        //Récupération de la densité du point courant
        float d00 = dens[IX(i,j, k)];
        //Activation de la couleur désirée, les 4 composantes RGBA sont
passées à OpenGL
        glColor4f ( cos(d00/0.5), sin(d00/0.5)*0.5,
(d00*d00)/8.0,d00*d00);
        //Dessin du point lui même, un sprite texturé et orienté est
rendu par la carte 3D
        glVertex3f( x, y, z);
      }
    }
  }
glEnd();    //Fin du rendu de points
```



Le résultat final, deux rendus de fluide sans puis avec l'affichage des vecteurs force

4.3.5 **Les perspectives**

Cette solution se révèle très pratique car elle est simple et rapide à mettre en oeuvre. Seulement elle ne s'avère pas toujours pleinement satisfaisante en particulier lorsque l'on a besoin d'un rendu précis. De plus, elle n'intègre pas de prise en compte de l'éclairage un élément qui pourrait permettre une meilleure appréhension de la simulation.

Pour des développements futurs, je pense qu'il faudrait plutôt s'orienter vers une implémentation du *volume rendering* profitant de l'énorme apport que sont les *shaders* et plus particulièrement les *fragment programs* pour effectuer des calculs d'éclairages par fragment sur les couches rendues. Je pense qu'il y a ainsi moyen d'obtenir un rendu extrêmement réaliste et précis. De plus, le problème lié au fait qu'il faille toujours rendre des plans parallèles au point de vue semble, à priori, soluble en rendent non plus une mais trois séries de plans, disposée orthogonalement les une par rapport aux autres. Cela permettrait ainsi de disposer toujours d'au moins une série de plan visible (non rendu sur la coupe). Cette voie me parait donc très intéressante à explorer.

Il y aurait peut être également des choses à expérimenter du coté des *marching cubes* et de leur implémentation avec les *vertex programs*. Le matériel actuel ne permet pas de créer de nouveaux vertex à l'intérieur d'un *vertex program* mais cette capacité ne devrait pas tarder à faire son apparition et pourrait s'avérer très utile dans ce cas particulier. On pourrait imaginer par exemple d'envoyer seulement à la carte graphique le quadrillage de cubes avec leurs angles pondérés. Elle se chargerait ensuite dans un *vertex program* de calculer les intersections de *l'isosurface* avec les arrêtes des cubes et de créer les vertex correspondants afin de former la surface.

4.4 La création d'outils génériques de parallélisation de jeux cellulaires

4.4.1 Problématique

J'ai pu découvrir le fonctionnement de plusieurs jeux cellulaires et je suis en fait parti d'une observation simple: Que ce soit le jeu de la vie, les fourmis, les termites, la simulation de lapins/renards ou même la simulation de fluide, tous ces problèmes fonctionnent sur le même schéma. Ils sont tous basés sur une grille discrète à deux ou trois dimensions et chaque case de cette grille a besoin de connaître un certain nombre de cases qui l'entourent pour évoluer.

L'idée est de généraliser la parallélisation de ce genre de problème en fournissant des outils génériques capables de réaliser cette paraléllisation tout en la masquant le plus possible à l'utilisateur.

4.4.2 La grille

Avant de songer à la parallélisation, il a d'abord fallu inventer une structure de base pour représenter une grille. J'ai donc commencé par créer un objet générique appelé *Grid* qui servira d'interface à toutes les implémentations possibles d'une grille. Cette interface fixe les méthodes d'accès et de manipulations de base.

template<class T> Grid

```
Grid(IVector t);

virtual void getSize(IVector &t));

T &getCell(FVector &pos);

virtual T &getCell(IVector &pos)=0;
virtual T &getCellToric(IVector &pos)=0;

void setSize(FVector t);
void getSize(FVector &t));


virtual T &operator[](int i)=0;
virtual int getNbElem()=0;
```

La classe *Grid* et ses méthodes publiques

J'ai réalisé une implémentation de cette interface dans une classe appelée *ArrayGrid* qui stocke les données sous la forme d'un tableau C.

4.4.3 La fenêtre sur grille

Cette classe que j'ai appelée *GridWindow* permet de manipuler seulement une portion de grille comme si il s'agissait en fait d'une grille complète. Elle sera utile par la suite pour permettre à chaque noeud de travailler de façon transparente sur la partie de la grille parallèle dont ils ont la charge.

```
template<class T> GridWindow : public Grid<T>

IVector vpos;
Grid<T> *grille;

GridWindow(IVector t, IVector p, Grid<T> *gr);
virtual T& getCellToric(IVector &pos);
virtual T& getCell(IVector &pos);
Grid<T> *getGrid();
IVector &getPos();
T &operator[](int i);
int getNbElem();
```

La classe *GridWindow*, la partie grise contient les champs privés

4.4.4 La grille parallèle

Je l'ai appelée *ParallelGrid* et il s'agit donc de la classe parallèle elle même. L'implémentation que j'ai faite se base sur une *ArrayGrid* et parallélise donc un tableau C grâce a l'utilisation de la bibliothèque MPI. Cette architecture n'est pas définitive car il manque ici une interface générique utilisée par tous les objets parallèle et définissant leurs méthodes de base. La parallélisation se fait principalement grâce à 3 méthodes : update(), updateA() et waitA().

La méthode *update* est une méthode de transfert synchrone qui se contente de transférer les données nécessaires à l'étape suivante de calcul. Dans le cas présent, elle transfert ses bordures aux voisins concernés et récupèrent donc celles de tous ses voisins. J'ai implémenté ici un découpage horizontal de la grille par soucis de simplicité, chaque noeud ayant ainsi uniquement deux voisins.

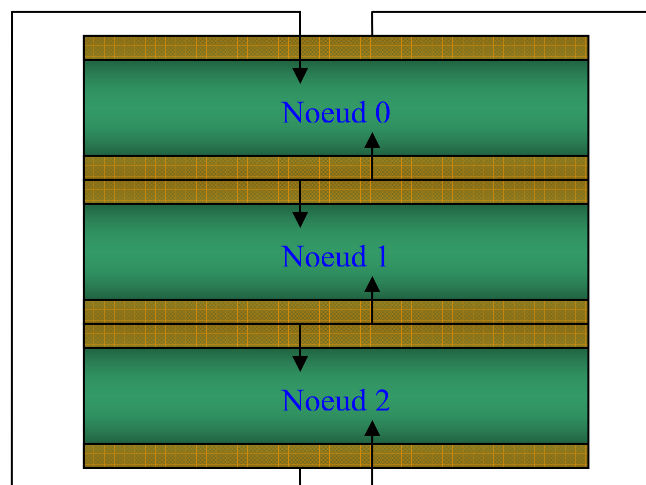


Schéma de principe des transferts de bordures effectuées par la méthode *update* entre 3 noeuds

Les méthodes *updateA* et *waitA* permettent de faire des transferts asynchrones pendant que des calculs ont lieux, *updateA* initie un transfert asynchrone et *waitA* le termine. Ces méthodes sont destinées à envoyer l'ensemble des données propres à chaque noeud au noeud d'affichage.

```
template <class T> ParallelGrid:public ArrayGrid<T>

int nbhost;
int myrank;
int blockSize;

ParallelGrid(IVector t, T*c);
ParallelGrid(IVector t);

void update();

void updateA();

void waitA();
```

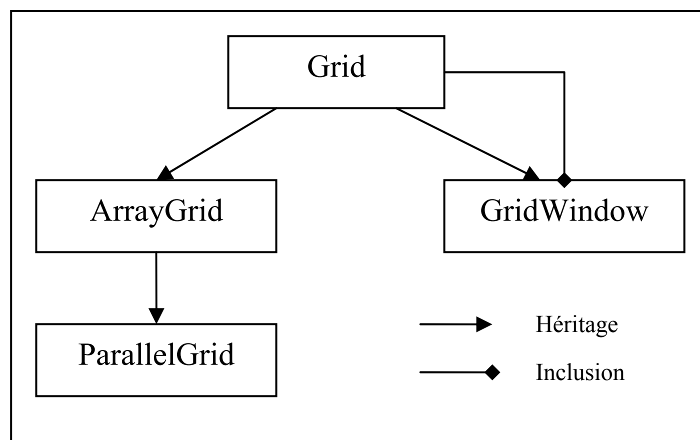


Schéma du graphe d'héritage et d'inclusion mis en jeu

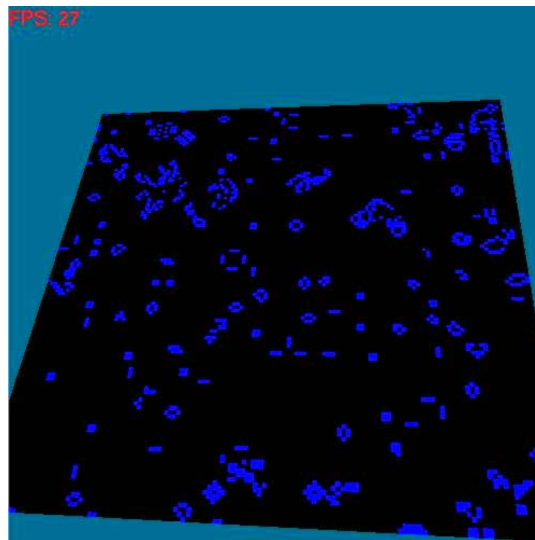
Les méthodes de parallélisation ont été implémentées grâce à l'utilisation de l'API MPI. Les transferts synchrones sont réalisés par de simples appels aux méthodes de communication point à point *Send* et *Receve* alors que les transferts asynchrones utilisent les fonctions *ISend* et *IReceve* pour l'initialisation des transferts ainsi que *Waitall* pour tester leur terminaison effective.

4.4.5 **Application au jeu de la vie**

J'ai testé cette architecture sur une implémentation du jeu de la vie. Cela m'a permis d'en tester tous les composants et surtout d'effectuer des tests de performance. J'ai pu observer un *speed-up* (multiplication des performances) au niveau du *framerate* (nombre d'images affichées par seconde) de pratiquement 2 sur une grappe constituée de 4 machines avec un noeud d'affichage par rapport à une exécution monoposte. Il s'agit d'une bonne performance mais elle aurait du être encore supérieure vu que les seuls transferts effectués de manière synchrone étaient les bordure qui ne représentent que très peu de données. Je me suis aperçu que ce qui nuisait aux performances était en fait le transfert asynchrone vers le

noeud d'affichage. En le désactivant, le *speed-up* atteignait quasiment 4, une performance beaucoup plus normale. Après de nombreux tests, je suis arrivé à la conclusion que les transferts que MPI était sensé effectuer de manière asynchrone, pendant le calcul de la simulation, ne se faisait en fait qu'après les calculs au moment de l'attente (MPI_Wait) de l'achèvement des transferts. La documentation MPI indique que les transferts asynchrones peuvent êtres utilisés pour effectuer un recouvrement des calculs sur des plateformes matérielles qui supportent ce genre d'opération, sans plus de précision. Il semblerait donc que les cartes Ethernet sur les quelles j'ai effectué ces tests, que ce soit a l'IUT ou au LIFO ne soient pas capables d'aller chercher directement les données à transférer en mémoire centrale sans intervention du processeur. Il se pourrait également que ce soit simplement l'implémentation MPI que nous utilisons (LAM et MPICH) qui ne supporte pas ce genre d'opération, le problème est que cela n'est documenté nulle part.

Pour pouvoir aller plus loin, il faudrait maintenant effectuer des tests sur une autre architecture réseau comme *Myrinet* pour voir si les transferts se font de manière asynchrone ou non. Je n'ai pas pu effectuer ces tests pendant mon stage car le réseau *Myrinet* qui équipe la grappe du LIFO ne fonctionnait pas. En tout cas, l'impossibilité de pouvoir procéder à un recouvrement des transferts destinés aux noeuds graphiques par les calculs de simulation est un problème très pénalisant pour les performances globales d'une simulation parallèle et ce sujet mériterait vraiment d'être approfondit.



L'implémentation parallèle du jeu de la vie

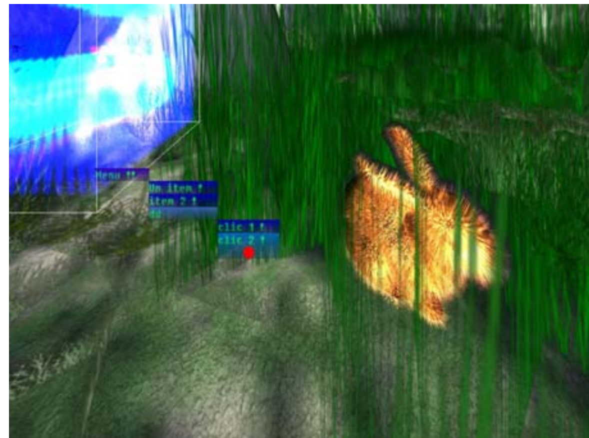
4.4.6 Le plateau de jeu

La notion de plateau de jeu permet de généraliser le fonctionnement de la grille à un monde continu. Je l'ai implémenté sous la forme d'une classe appelée *Board* qui est en fait basée sur un objet *Grid*. Le but de cet objet est de fournir des méthodes génériques de manipulation d'occupants dans ce monde. Cette classe sert également de base à une classe dérivée appelée *ParallelBoard* qui est en fait la version parallèle du plateau de jeu. Il offre des méthodes génériques pour paralléliser le calcul d'une simulation de type vie artificielle dans un monde continu. Elle utilise une interface *Pawn* que doivent implémentés les occupants pour êtres parallélisés.

4.5 La réalisation d'une démo

4.5.1 Présentation de la démo

La démo que j'ai réalisée utilise en fait une grande partie des concepts sur lesquels j'ai travaillé durant mon stage. Il s'agit d'une simulation de vie artificielle extrêmement simple mettant en jeu des lapins qui mangent de l'herbe. Les lapins se déplacent donc sur un terrain vallonné recouvert d'herbe. Un menu permet d'activer ou de désactiver l'affichage de fourrure sur les lapins et les déplacements de caméra s'effectuent à l'aide d'un pad sans fil grâce à l'intégration du travail effectué par Bertrand sur les périphériques d'entrée.



Cette démo réalisée en OpenGL est basée sur l'API VR-Juggler et utilise bien sur Net-Juggler pour la distribution des périphériques d'entrée. De plus, la simulation de lapins est parallélisée et exploite les possibilités offertes par une grappe de PC.

4.5.2 Ma problématique

J'ai essayé, avant d'implémenter cette démo, de concevoir une architecture propre et réutilisable qui me permette de disposer de tous les outils nécessaires tant au niveau de la simulation parallèle que du rendu. J'ai ainsi créé toute une série d'objets s'inscrivant dans une architecture commune.

4.5.3 La notion de Layer

La notion de *Layer* que j'ai imaginée permet de regrouper dans une même entité tous les éléments permettant le calcul et le rendu d'une simulation. Cela permet ainsi de regrouper plusieurs simulations dans un même environnement de réalité virtuelle et d'autoriser leur interaction.

4.5.4 La séparation des données brutes et des données de contexte

En OpenGL, lorsque l'on travaille dans un environnement avec plusieurs surfaces d'affichage comme c'est souvent le cas avec VR-Juggler lorsque l'on paramètre un affichage stéréoscopique par exemple, il faut prendre en compte un élément important lié a l'architecture même de l'API OpenGL : La notion de contextes. Chaque surface d'affichage appartient à ce que l'on appelle un contexte, un environnement de travail stockant toutes les données qui lui sont propres et totalement indépendant des autres. Ce mécanisme permet à

plusieurs applications d'utiliser OpenGL au même moment sans risque de conflits. Tout se passe donc comme si chaque surface appartenait à une application différente et il est impossible d'accéder aux données d'un contexte à partir d'un autre. Certains environnements permettent le partage de contexte d'affichage mais cela n'est pas possible sous VR-Juggler.

Les contextes stockent bien sur les surfaces d'affichage mais également les données de texture, les *display lists* et même les programmes de *shaders*. C'est pour cette raison qu'il convient toujours de bien séparer les données brutes, comme les couleurs d'une textures ou la liste de vertex d'un objet, que l'on charge une fois dans une structure appropriée et les identifiants fournis par OpenGL dans chaque contexte pour identifier les données chargées dans la carte 3D. Lorsque l'on travail avec plusieurs contextes différents il faut donc créer les *display lists*, charger les textures et les *shaders* dans chacun des environnements qui vont les utilisés.

Dans la structure d'une application VR-Juggler, les données brutes peuvent donc êtres stockés dans la classe application elle même alors que les données de contexte doivent êtres utilisées à travers un objet spécialement conçu pour leur gestion appelé `vjGlContextData`.

4.5.5 Les textures

Compte tenu des contraintes qu'imposent les contextes sous VR-Juggler, j'ai décomposé la gestion des textures en deux classes. La première que j'ai appelée *TextureData* a pour rôle le chargement et le stockage des données de couleur d'une texture. La deuxième appelée simplement *Texture*, stocke l'identifiant de texture fournis par OpenGL.

➢ La classe TextureData

Cette classe stocke les données de couleur des textures sous la forme d'un tableau d'octets (unsigned char) ainsi que leur taille et leur nombre de composantes de couleur (généralement 4: RGBA). Elle fournit une méthode de chargement appelée *load* qui prend en paramètre le nom du fichier à charger. Les seuls formats acceptés pour le moment sont les BMP et le TGA. Elle fournit également une méthode pour charger ses données a partir d'une *heightmap*, une classe que je décris un peu plus loin. Les données sont publiques pour permettre un chargement de l'extérieur.

TextureData

```
int sizeX, sizeY;
unsigned char *data;
int channels;


TextureData(char *name=0);


int load(char *name);

int build(HeightMap *hm);

unsigned char &getPixel(int px, int py, int chan);
```

➢ **La classe Texture**

La classe Texture se charge de toutes les étapes nécessaire au passage d'une nouvelle texture à OpenGL, elle stocke son identifiant et fournit une méthode pour activer la texture. Elle permet également de choisir la méthode de filtrage à appliquer à la texture.

Texture

```
TextureData *data;

GLenum texFormat;
GLuint texID;  //Identifiant de texture
//Modes de filtrage d'agrandissement et de
réduction
GLint magFilter;
GLint minFilter;

int filterMode; //Identifiant utilisateur de la
méthode de filtrage
```

```
Texture(TextureData *td=0);
//Construction de la texture
int build(TextureData *texdata);
//Activation
void activate();
//Specifie la methode de filtrage
void setFilterMode(int m);
//Renvoi l'identifiant de texture
GLuint getTexID();
```

La méthode de filtrage détermine la façon dont OpenGL transforme les *texels* (pixels sur la texture) en fragments. Il y a deux méthodes de filtrage couramment utilisées en OpenGL:

- NEAREST : OpenGL prend seulement les points de la texture les plus près des points affichés.

- LINEAR : OpenGL fait une moyenne des points les plus près sur la texture ce qui provoque un effet d'anti-aliasing.

Il existe d'autres méthodes comme le filtrage *anisotropic* qui permet de tenir compte de l'inclinaison par rapport au point de vue des objets texturés.

En plus de ces méthodes de filtrage vient s'ajouter la technique de *Mip-Mapping* qui permet d'avoir un jeu de plusieurs résolutions de la même texture afin d'améliorer le rendu des objets lointains.

Toutes ces techniques sont implémentées à l'intérieur de l'objet Texture.

4.5.6 **Les matériaux**

La notion de matériau, implémentée dans une classe que j'ai appelée *Material*, permet de stocker tout ce qui caractérise la surface d'un objet. C'est une classe de contexte et doit donc être construite pour chaque surface de visualisation. L'implémentation actuelle se contente de gérer un ensemble de textures appliquées par *multi-texturing* à un objet.

Material

```
int nbTex;          //Le nombre de texture
Texture tex[4];     //Les textures elle mêmes

FVector texScale[4]; //La mise a l'échelle de
chaque texture
int rgbScale[4];     //L'amplification des couleurs
pour chaque texture
```

```
Material();
//Renvoi la texture demandée
Texture &getTex(int nb);
//Ajoute une texture au rang nb
void bindTex(int nb, Texture t);
//Ajoute une texture a partir d'une TextureData
void bindTex(int nb, TextureData &t);
//Active le matériel
void activate();
//Spécifie l'amplification de couleur pour une
texture
void setRGBScale(int nb, int sc);
//Spécifie l'échelle d'une texture
void setScale(int nb, FVector sc);
//Renvoi le nombre de textures placées
int getNbTex();
```

4.5.7 **Les modèles**

Comme pour les textures, j'ai décomposé la gestion de modèle en une classe de données brutes que j'ai appelée t3DObject et une classe d'affichage qui s'appelle Model3D.

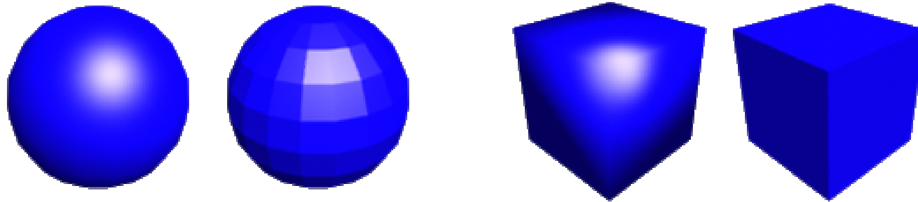
➢ La classe t3DObject

Il existe un grand nombre de façon de stocker des données de modèle et j'ai tenté de mettre en place une structure capable de s'adapter au chargement d'un maximum de formats de fichier. Je ne décrirai pas dans le détail les différents champs présents dans la classe t3DObject, ils permettent de stocker les informations de vertex (position), de normales et de coordonnées de mapping. Il stocke également des informations sur les triangles qui forment le modèle ce qui permet de partager des vertex entre triangles. Les triangles stockent en effet les indices des vertex qui les composent ainsi que ceux de leurs normales et coordonnées de mapping. C'est en fait cette liste de triangle que l'on parcourt pour rendre l'objet.

Cette classe ne comporte qu'une seule méthode que j'ai appelée builNormals() qui permet de calculer des normales à l'objet. Suivant le paramètre passé, ces normales sont calculées par triangles ou par vertex (normales lissées ou *smooth*). Les normales par vertex sont issues de l'interpolation entre les normales des différents triangles auxquels ils appartiennent. Dans la plupart des cas, ce sont les normales calculées par vertex qui sont le mieux adaptées mais il est utile de pouvoir disposer de normales par triangles dans certains cas particuliers comme pour l'éclairage d'un cube par exemple ou des normales calculées par vertex ne sont pas appropriées.

Exemple sur un cube (vue de face) de l'application d'un calcul de normale par triangle (à gauche) et par vertex

Le calcul de normale est une opération très simple puisqu'il s'agit simplement, pour chaque triangle, d'effectuer le produit vectoriel de deux de ses cotés. La difficulté a simplement été dans l'optimisation de ce calcul et surtout de celui qui permet de calculer pour chaque vertex, la moyenne des normales des triangles auxquels il appartient. La difficulté viens du fait qu'il faut, pour chaque vertex, parcourir la liste des triangles pour vérifier si il y appartient et si c'est le cas récupérer la normale a ce triangle. J'ai donc mis en place une table stockant pour chaque vertex la liste (indices) des triangles auxquels il appartient. Un seul parcours préalable de l'ensemble des triangles suffit donc maintenant pour remplir ce tableau d'indices. Un seul parcours du tableau d'indices suffisant ensuite pour calculer les normales moyennes.



Exemples d'un éclairage basé sur des normales lissées puis non lissées. On voit parfaitement que les normales lissées ne donnent pas un résultat satisfaisant sur le cube qui semble *s'arrondir*.

➢ La classe 3DModel

Cette classe gère l'affichage d'un modèle 3D. Elle se construit à partir d'un *t3DObject* et d'un *Material* et fournit une méthode *draw* qui en effectue le rendu. Elle se charge de la création d'une *display list* et du stockage de son identifiant ainsi que de la mise en place de tous les états nécessaire pour assurer un rendu correct.

➢ La classe SkyBox

Il s'agit d'un descendent de la classe 3DModel qui se construit à partir d'un vecteur taille et de 6 textures. Cette classe permet en fait d'afficher une boite englobante servant a simuler un paysage qui entoure une scène. La taille du cube combinée à l'utilisation de textures adaptées produit un effet très réaliste.

Un exemple de *cubemap* (6 textures) représentant un paysage montagneux et utilisée sur une SkyBox

4.5.8 Les modèles animés

Cette classe appelée Model3D_Cg étend les capacités de l'objet Modele3D en fournissant le moyen d'afficher des modèles qui peuvent êtres animés. Le nom de cette classe vient du fait que j'ai implémenté le rendu de l'animation en utilisant le langage de programmation de *shaders* Cg (d'où le nom de la classe) développé par nVidia. L'utilisation des *shaders* m'a permis de rendre et d'animer de très gros modèles avec énormément de triangles sans surcharger le processeur central, en déléguant un maximum de calcul à la carte graphique.

- **Le Cg**

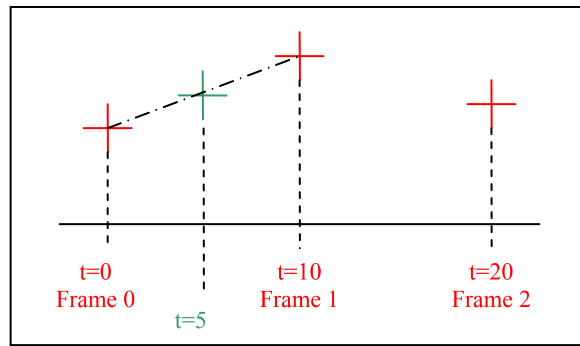
Le Cg est un langage de programmation développé par nVidia dans le but de faciliter la programmation des *shaders* (cf. section 3.2.13) en fournissant un langage à la syntaxe proche de celle du C. Il permet de programmer aussi bien les *Vertex Programs* que les *Fragment Programs* de façon transparente par rapport aux extensions utilisées. Il dispose en effet d'un mécanisme de *profile* qui offre au programmeur la possibilité de choisir l'extension cible pour laquelle le programme Cg sera compilé. Ce système permet donc d'utiliser les extensions ARB_vertex_program et ARB_fragment_program sur les configurations les supportant ce qui fournit une portabilité totale des programmes développées sur toutes les cartes 3D supportant ces extensions. Il est également possible d'utiliser les extensions propriétaires de nVidia pour, par exemple, pouvoir bénéficier des *fragment shaders* sur une geForce4 (situation à laquelle j'ai été confronté) qui ne supporte pas l'extension ARB_fragment_program. Au moment de la compilation d'un programme Cg, les limites propres a chaque profiles son prises en compte pour signaler l'utilisation de fonctionnalités non supportées.

Bien que fournissant une syntaxe de haut niveau, le Cg reste extrêmement proche du matériel et une connaissance de la programmation des *shaders* en assembleur s'avère très utile pour bien en appréhender le fonctionnement.

Le Cg est également indépendant de l'API graphique utilisé et fonctionne aussi bien sous OpenGL que sous Direct3D.

- **L'animation d'un modèle**

Les modèles animés sont chargés à partir de fichiers enregistrés au format MD3 et stockés dans l'objet *t3DModel*. Le format MD3 est le format de fichier utilisé dans le jeu Quake 3 et il a l'avantage de contenir à la fois un modèle et son animation. Ces animations sont encodées sous la forme de *keyframes*, c'est à dire que l'on enregistre la position de chaque vertex formant le modèle pour chaque étape de l'animation. Lors du rendu, pour obtenir une animation fluide, il faut pouvoir déterminer la position de chaque vertex à un temps t, qui peut se trouver entre deux étapes (*frames*). On effectue donc pour chaque vertex, une interpolation linéaire entre les positions stockées dans les deux *frames*.



Schématisation de l'animation d'un vertex comportant 3 keyframes. Un rendu a t=5 nécessite une interpolation linéaire entre les positions du vertex dans les *frames* 0 et 1.

Il faut, en plus des positions, interpoler les normales pour chaque vertex et c'est pour effectuer ces deux interpolations linéaire dans un *vertex program* que j'ai utilisé le Cg. Je stocke donc, dans chaque vertex, un deuxième paramètre de position et de normale. Les deux positions et normales représentent donc les positions et normales du vertex dans les deux *frames* qui entourent sa position réelle.

```
struct appin {
      float4 position : POSITION;
      float4 pos2     : TEXCOORD4;
      float3 normal   : NORMAL;
      float4 normal2    : TEXCOORD5;

      float4 color0   : COLOR0;
      float4 texCoord0 : TEXCOORD0;
      float4 texCoord1 : TEXCOORD1;
};
```

La structure Cg définissant les paramètres passés au *vertex program* par vertex. Les noms indiqués en vert précisent au compilateur Cg les paramètres fixes dans lesquels il doit placer ses paramètres personnalisés. Cela permet d'utiliser les fonctions standard d'OpenGL pour passer des paramètres au *vertex program*.

En plus des paramètres par vertex, le *vertex program* a besoin d'un paramètre supplémentaire, commun à tous les vertex rendus pour un modèle, qui indique le temps courant entre les deux *keyframes* passées par vertex. C'est un *float* compris entre 0 et 1 transmis au *vertex program* en tant que paramètre uniforme.

Le rendu de chaque étape est encapsulée dans une *display list*. Pour un modèle animé, il y a autant de *display list* qu'il y a de *frame*, la dernière permettant de lier la dernière étape avec la première afin de boucler l'animation.

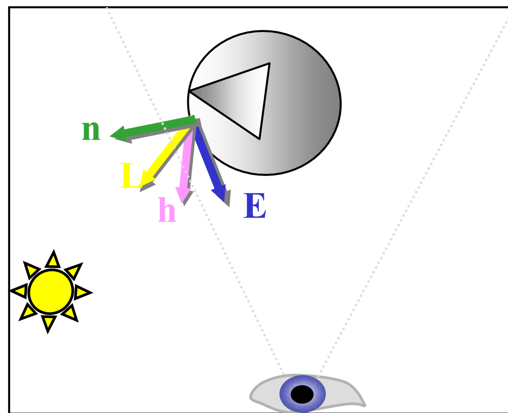
```
vertout main( appin IN, uniform float time ) {
        vertout OUT;

        //...
        //interpolation position
        float4 temp = float4( IN.position.x, IN.position.y, IN.position.z, 1.0f );
        temp = temp + time * (IN.pos2 - IN.position);

        OUT.position = mul( modelViewProjection, tempPos );

        //interpolation normales
        temp=float4( IN.normal.x, IN.normal.y, IN.normal.z, 1.0f );
        temp = temp + time * (IN.normal2 - temp);
        //...
}
```

La partie de mon programme Cg qui effectue les interpolations de position et de normale. La fonction main est, comme en C, le point d'entrée par défaut d'un programme Cg. Le type *vertout* renvoyé par la fonction est définit dans une structure qui contient les paramètres destinés aux fragments. La position finale du vertex est calculée en multipliant la position interpolée par la concaténation des matrices *ModelView* et *Projection*.

➢ L'éclairage du modèle

A partir du moment où l'on utilise les *shaders*, un certain nombre de fonctions fixes autrefois assurées par OpenGL doivent êtres implémentés dans les programmes qui les remplacent. C'est le cas, dans les *vertex programs*, pour les transformations matricielles (*ModelView*, *Projection* et *Texture matrix*) et également les calculs d'éclairage.

Il m'a donc fallu réimplémenter manuellement le calcul des trois composantes du modèle d'éclairage standard (cf. section 3.2.6). J'effectue ces calculs d'éclairage par vertex puis je les combine aux textures dans un *fragment program*.



Les 4 vecteurs nécessaires au calcul d'éclairage. n: normale, L: light, h: *half-angle* (Vecteur médian entre l'éclairage et le point de vue) E: eye

Diffuse
max{L dot n, 0}* $diffuse_{light}$

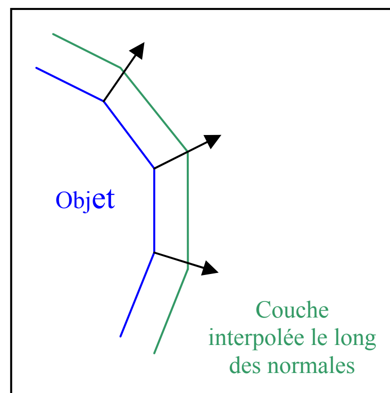
Specular
max{h dot n, 0}^{SHININESS} * $specular_{light}$ Si {L dot n} < 0, la composante égale 0.

Le calcul des composantes Diffuse et Spéculaire. SHININESS détermine le coefficient de réflexion du matériau. Le mot dot représente le calcul d'un produit scalaire.

4.5.9 **Le rendu de pelage**

Le rendu de pelage est un effet qui a été implémenté pour la première fois à l'aide de *shaders* par un étudiant qui a travaillé sur le projet NetJuggler (Jérémie Allard). Le principal problème de cette version était qu'elle utilisait des extensions propriétaires uniquement disponibles sur les cartes 3D nVidia (En particulier les *register combiners*, cf. section 3.2.13). Je suis donc parti du fonctionnement global de cette première version pour réaliser ma propre implémentation en Cg de cet algorithme de rendu de pelage.

L'idée globale est de rendre une succession de couches englobant l'objet sur lequel on veut appliquer un pelage, en leurs appliquant une texture conçue pour en simuler l'aspect. En dessinant un nombre suffisant de couches et en appliquant un éclairage approprié, l'illusion de fourrure est parfaite.



Pour créer les couches formant la fourrure, on interpole la position de chaque vertex le long de sa normale. Pour obtenir un résultat correct, les normales doivent donc êtres lissées.

J'ai incorporé ce rendu de pelage dans mon architecture globale en l'Implémentant dans une classe dérivée de *Model3D_Cg* appelée *Model3D_CgFur*. J'ai également créé une classe de génération de textures de poils utilisée par *Model3D_CgFur*. C'est un héritier de *Texture* que j'ai nommé *FurTexture*.

Deux exemples de rendu de pelage sur des modèles chargés à partir d'un fichier MD3

Conclusion

Ce stage a réellement été très positif pour moi. Il m'a permis tout d'abord de découvrir le monde de la recherche fondamentale en Informatique, un domaine qui m'est apparu extrêmement intéressant. Il m'a également permit de travailler en collaboration avec un grand nombre de personnes très compétentes dans différents domaines de la Réalité Virtuelle grâce à quoi j'ai pu acquérir énormément de connaissances et de compétences techniques dans ce domaine.

La collaboration avec Bertrand a également été très positive pour moi car il s'est avéré que nous n'avions pas du tout la même approche des problèmes rencontrés. Il abordait les problèmes de façon très mathématique alors que j'avais tendance à les appréhender plutôt de façon intuitive. En fait, je pense que les deux approches ont leurs avantages et que la bonne, si elle existe, doit se trouver entre les deux.

Mon approche d'un grand nombre de points clef de la VR m'a permise de découvrir l'étendu des problèmes posés et la largeur de son champ d'application.

Je regrette seulement de ne pas avoir eu d'avantage de temps pour développer plus en profondeur un certain nombre de points que j'ai abordés durant ce stage. Je n'ai en effet pas pu implémenter un certain nombre d'idées en particulier sur la désynchronisation rendu/calcul et dans la mise en place d'une architecture complète de création de démos de *réalité virtuelle*.