Recherches en rendu non réaliste temps réel

De nouveaux moyens d'utiliser la carte graphique

Antoine Zanuttini
Master en Arts et Technologies de l'Image

2008
Arts et Technologies de l'Image
Université Paris 8

Remerciements

Je tiens à remercier tous les professeurs de ces quatre dernières années au sein d'ATI, et principalement Marie-Hélène Tramus et Pascal Ruiz, pour leur aide tout au long de l'année, qui à permis au projet de garder un semblant de cohérence.

Je remercie également mes camarades d'ATI, pour le soutien moral, les conseils mais surtout les critiques constructives.

Je remercie toute l'équipe de Cyanide Studio, qui m'a offert de quoi m'occuper l'esprit toute l'année, mais aussi une vision très claire du développement de jeux, tant du coté logiciel que du coté humain.

Évidement, les remerciements les plus importants vont à ma famille, qui me soutient depuis la lointaine province de Strasbourg.

Sommaire

Introduction

Le travail tout au long de l'année à principalement été centré sur l'expérimentation de différentes techniques, au gré des nouvelles idées. Le but commun de ces expérimentations était de chercher de nouvelles méthodes de calculs d'effets, dans l'idée d'obtenir un style de rendu original.
A partir de certaines influences graphiques, des techniques ont été développées, qui ont donnée de nouvelles idées graphiques, et ainsi de suite, dans un va et viens s'approchant progressivement d'un style particulier.

Les logiciels et langages de programmation utilisé pour ce projet sont : Virtools, en temps que moteur de rendu temps réel, Maya pour la production de donnée 3D, le langage HLSL pour la quasi totalité de la programmation graphique et le langage C++ à travers le SDK de Virtools pour certains éléments nécessitant beaucoup de rapidité.

Le présent mémoire va tout d'abord présenter un bref état de l'art du rendu non réaliste dans le jeu vidéo, puis le projet en lui même, suivit des explications sur chacune des techniques développées pendant l'année.
Un glossaire situé à la fin du mémoire permet de définir les termes utilisées couramment dans ce mémoire, et une table des références donnera des pistes pour aller plus loin sur tel ou tel sujet. En annexe, les parties fondamentales des codes des principaux effets sont présenté.

1. Etat de l'art

1.1 Influences :

Au départ, il faut noter deux influences majeures, qui ont d'ailleurs déjà été adapté en partie en jeu vidéo.

Giorgio de Chirico / ICO & Shadow Of The Colossus

La première influence est celle de Giorgio de Chirico, peintre italien de la fin du 19ème. Le jeu Ico de Fumito Ueda est influencé par ce peintre, même si c'est surtout la pochette du jeu vidéo qui y fait référence. L'essentiel de ce qui nous intéresse est l'impression que laissent les paysages du peintre et les décors du jeu vidéo Ico. C'est une impression mélancolique, avec un soleil aveuglant mais en même temps froid, un certain traitement des couleurs, des ombres très fortes et très rasantes.

ICO

Giorgio de Chirico

Le jeu vidéo qui fait, en quelque sorte, suite à Ico, développé par la même équipe, se nomme Shadow Of The Colossus. Ce jeu reprend en grande partie l'atmosphère d'Ico, mais la ou dans Ico nous étions principalement dans des décors intérieurs et de petites cours extérieures, dans Shadow Of The Colossus, nous sommes confronté à un monde immense et totalement ouvert, qu'il va falloir explorer, principalement à cheval vu les distances. L'atmosphère mélancolique est renforcée par la solitude du personnage principal, isolé dans ce vaste univers.

Shadow Of The Colossus

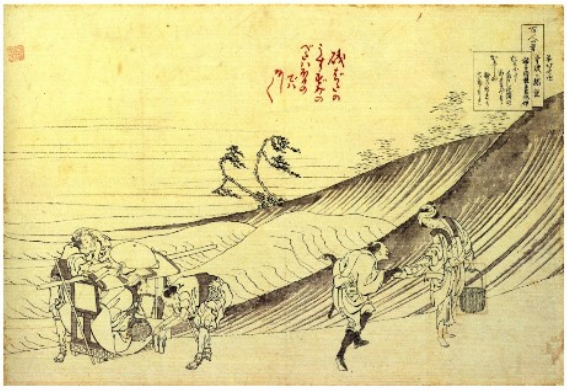
C'est cette impression qui va être le but de ce projet. L'ouverture du paysage, l'ambiance très marquée, ce sont les éléments qui devrons transparaitre dans le projet final.
Dans Shadow Of The Colossus, le personnage rencontre régulièrement d'énormes colosses, qu'il va s'agir de battre. La aussi, l'ambiance impressionne, avec une sensation de puissance, de fureur du colosse. Le colosse soulève la poussière, fait trembler la terre. Lors de la mort du colosse, celui-ci se désintègre en une fumée noire, qui fini par se précipiter pour s'introduire dans le corps du héros. Cette image de fumée est également une référence dans notre recherche sur les fluides.

Katsushika Hokusai / Okami

La seconde influence est celle de Katsushika Hokusai, un peintre japonais du début du 19ème. Le jeu vidéo Okami du studio Clover est inspirée des œuvres de ce peintre et du courant artistique qu'il représente, le ukiyo-e, un style d'estampes japonaises.
Deux aspects sont intéressants ici. Tout d'abord d'un simple point de vue technique, le jeu vidéo Okami est un des premiers jeux vidéo avec un rendu non réaliste aussi prononcé et particulier. Les techniques utilisées sont originales, le rendu est très particulier.
Et le titre se démarque du point de vue visuel, particulièrement au niveau de l'aspect encré du jeu vidéo, du style du ukiyo-e. L'univers est constamment en mouvements, les bords ondulent, on peux voir certains effets de coulures (par exemple lorsque l'on utilise le pinceau dans le jeu), et beaucoup d'effets de fumées lors des apparitions et disparitions des monstres du jeu.

Okami

Katsushika Hokusai

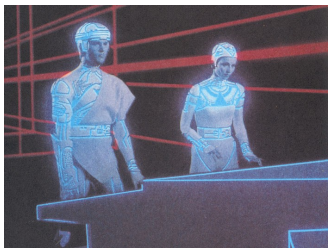


Bande dessinée

Une dernière influence est celle de la bande dessinée européenne et japonaise, particulièrement dans le traitement des surfaces et des valeurs de luminosité par l'utilisation de hachures et de traits plus ou moins épais.

1.2 Le rendu non-réaliste dans le jeu vidéo :

Le rendu non-réaliste à déjà une belle histoire, et bien évidement, elle s'est d'abord faite par le rendu pré-calculé.
Au début, le rendu non-réaliste était imposé par les limitations de la machine, qui empêchaient complètement d'obtenir un rendu réaliste et forçais donc les créateurs d'effets spéciaux à styliser le plus possible les graphismes, les simplifier afin qu'ils puissent être calculé par la machine.
Le film Tron, par exemple, le premier film utilisant des séquences entièrement réalisé numériquement, se décalait totalement de la réalité et proposait un design graphique très coloré, a base d'aplats de couleurs vives, afin de ne pas subir la comparaison avec l'image filmée.
Par la suite, la puissance des machines augmentant, l'intérêt des créateurs et du public se focalisa sur toujours plus de réalisme.
Les films d'animation qui sortent maintenant sont finalement assez réalistes et trouvent leur voie dans un léger décalage vers un rendu plus "patte à modeler".
Certains dessins animées, japonais notamment, montrent des designs novateurs et particuliers, mais n'utilisent pas forcement des techniques 3D pour cela, mais plutôt des assemblages d'éléments graphiques 2D, animés un peu comme des collages.



Tron, Star Wars et Asteroids

Le jeux vidéo a gardé bien plus longtemps ces contraintes de puissance de la machine.
On remarque un petit peu le même processus, les premiers jeux vidéos étant très abstrait, un rendu très éloigné de la réalité, puis avec le temps, les jeux sont devenus de plus en plus figuratifs et réalistes.
Ainsi de nombreux jeux du début de l'histoire du jeux vidéo peuvent être qualifiés d'abstraits. Asteroids, Pong, le premier Star Wars en 3D, bref tous les anciens jeux à base de graphismes vectoriels possèdent un aspect non réaliste qui les rends novateurs.
Malheureusement, le réalisme permet de vendre des consoles de jeux et de nouvelles machines, il est très soutenu par les constructeurs, et les arguments sont faciles a trouver pour séduire le public.
Mais, peut être parce que dans le jeu vidéo, tout est calculé en temps réel, de nombreux créateurs de jeux ont cherché a aller plus loin dans les techniques alternatives de rendu.
Le but n'est plus de trouver des techniques pour simuler et imiter la réalité un peu plus qu'elle ne l'était auparavant, mais de chercher des manières nouvelles de figurer les éléments graphiques, de faire passer des messages et des émotions sans faire appel aux éléments trop connus de la réalité.
Cela ne signifie pas que ces techniques ne se baserons pas sur la réalité, elles peuvent utiliser des techniques très réalistes d'éclairage ou de simulation physique, mais dans un but différent et en général novateur par rapport à l'état de l'art.

Nous ne parlerons pas beaucoup ici des jeux en deux dimensions, dont l'aspect non réaliste du rendu n'est en général lié qu'a la manière de dessiner les "sprites", c'est a dire les éléments qui constituerons décors et personnages. C'est alors un processus uniquement artistique, qui ne donne pas beaucoup de place à l'évolution technique.
Néanmoins, la plupart des jeux en deux dimensions du début du jeu vidéo utilisait des graphismes dessinés à la main, la synthèse d'image étant encore très balbutiante. Seul les jeux à base de graphismes vectoriels peuvent être qualifiés de réellement synthétiques. Ainsi la majorité des jeux dont les composants sont dessinés à la main pourraient être

qualifiés de non-réalistes. De plus les capacités limités des consoles de l'époque rendaient le choix des couleurs limités, ce qui déformait la réalité, et la faible résolution forçait une certaine caricature. Mais la majorité des jeux de cette époque cherchent le réalisme, sans y parvenir.
Les jeux de plateforme destinés aux enfants sont les premiers à se démarquer. Ainsi ils s'inspirent plus de l'atmosphère des dessins animées, et Mario ou Sonic en sont de bons exemples.

Les premiers jeux 3D suivent totalement cette idée. La troisième dimension n'est qu'un moyen de rendre les jeux plus réalistes. Ainsi Wolfenstein, Doom et finalement une apothéose de l'époque : Tomb Raider, poussent le graphisme le plus possible dans le réalisme. En parallèle de ce phénomène, nous retrouvons encore le genre du jeux de plateforme destinés aux enfants, avec les suites adaptés en 3D de Mario ou des jeux comme Spyro. Nous avons donc continuellement cette opposition : réalisme dans les jeux pour adultes et adolescents, dessin animé pour les jeux destinés aux enfants.

Ces deux styles sont alors pratiquement les seuls viables en terme de commercialisation.
C'est principalement lié au public qui était visé il y a encore seulement quelques années, c'est à dire les hard-core gamers (passionné de jeux vidéo, qui ne fait que cela), et les plus jeunes, les enfants.

Le monde du jeux vidéo à fini par s'ouvrir à un nouveau public, à s'adresser aussi spécifiquement au grand public dans toute sa diversité : femmes, personnes âgées.
L'esthétique habituelle du jeu vidéo ne fonctionnait plus vraiment avec ce public, peu sensible au changement apporté par une simple évolution graphique. Le gameplay des jeux à également changé, et donc l'objectif du jeu, ce qui influe sur l'esthétique qu'il doit dégager. Le jeu vidéo n'est plus simplement un processus immersif, cherchant à simuler au maximum la réalité, mais aussi un moyen de communiquer, un outil (Programme d'Entrainement Cérébral du Dr Kawashima), un compagnon (Nintendogs), ...

La Nintendo Wii est l'exemple le plus marquant de ce changement et surtout de la nécessité de ce changement. Cette console est à peine plus puissante que la précédente (la Nintendo Game Cube) mais concentre ses arguments sur de nouvelles manières de jouer, et vise un public très très diversifié. Le nombre de console Wii à rapidement dépassé ceux des autres constructeurs, Microsoft Xbox 360 et Sony PlayStation 3, qui conservent essentiellement le même discours centré sur l'évolution technique et la puissance de la machine.

1.3 Les jeux vidéo actuels :

On peu classer les jeux vidéo actuels utilisant un rendu non-réaliste dans quatre catégories :

Tout d'abord, les jeux qui s'adressent aux enfants ou présentent des atmosphères enfantines, et sont l'héritier direct de l'esthétique dessin animé. On pense à Mario (64, Sunshine et Galaxy) ou encore à Zelda Wind Walker. Les jeux issues de Manga tel que Naruto, Bleach et One Piece utilisent logiquement un type de rendu plus proche du dessin.

Une seconde catégorie est celle des jeux qui traditionnellement utilisaient une esthétique réaliste mais dont certains jeux sont maintenant passés à une esthétique non-réaliste.
Team fortress II est un exemple révélateur, suite d'un premier épisode réaliste, lui même tout d'abord présenté de manière réaliste, il a subit un changement d'orientation total et se rapproche maintenant visuellement beaucoup du film de Pixar, Les Indestructibles. Les personnages sont caricaturaux, le rendu est du pur cell shading. Le moteur utilisé est pourtant celui d'Half Life II, un moteur très réaliste. Il permet ainsi d'obtenir un rendu complexe, évolué, avec des calculs de lumière très poussés, tout en gardant un rendu non-réaliste.

Exactement dans le même style, le prochain Battlefield Heroes, fait la même évolution, prouvant que l'essai fonctionne.
Le nouveau Prince Of Persia créé la surprise. Même si les 3 récents épisodes adoptaient un rendu assez stylisé, au niveau des couleurs et des ambiances, ce nouvel opus change radicalement et utilise du cell shading, avec des traits autours des personnages et des textures très graphiques.
Des jeux comme Killer 7 et No More Heroes sont des jeux violent et pour les adultes mais qui utilisent du cell shading.

Prince Of Persia

No More Heroes

Madworld

La troisième catégorie est plus discrète, mais concerne beaucoup de jeux, dont l'aspect visuel va subtilement se styliser sans changer réellement. Ainsi les jeux Ico et Shadow Of The Colossus sont réalistes mais utilisent des couleurs, une saturation, certains effets qui ne sont pas réalistes et contribuent à donner une ambiance légèrement décalé par rapport à la réalité.

La dernière catégorie est celle des jeux qui touchent à l'abstrait, qui s'éloignent vraiment de la réalité et cherchent à atteindre une esthétique inhabituelle. Le jeu Okami en est un très bel exemple. Il reprend l'aspect des estampes japonaises (courant Ukiyo-e), un univers qui n'avait jamais été exploité en trois dimensions. Les jeux Locoroco et Patapon, bien qu'en deux dimensions, font appel à une nouvelle esthétique également.
Le prochain projet de l'équipe à l'origine d'Okami, le studio Clover, est cette fois pour les adultes. Il se nomme Madworld, et présente un rendu totalement en noir, blanc et rouge. L'atmosphère visuelle se rapproche du film Renaissance, avec un contraste noir/blanc énorme. La grande violence du titre est permise justement grâce au rendu non réaliste.

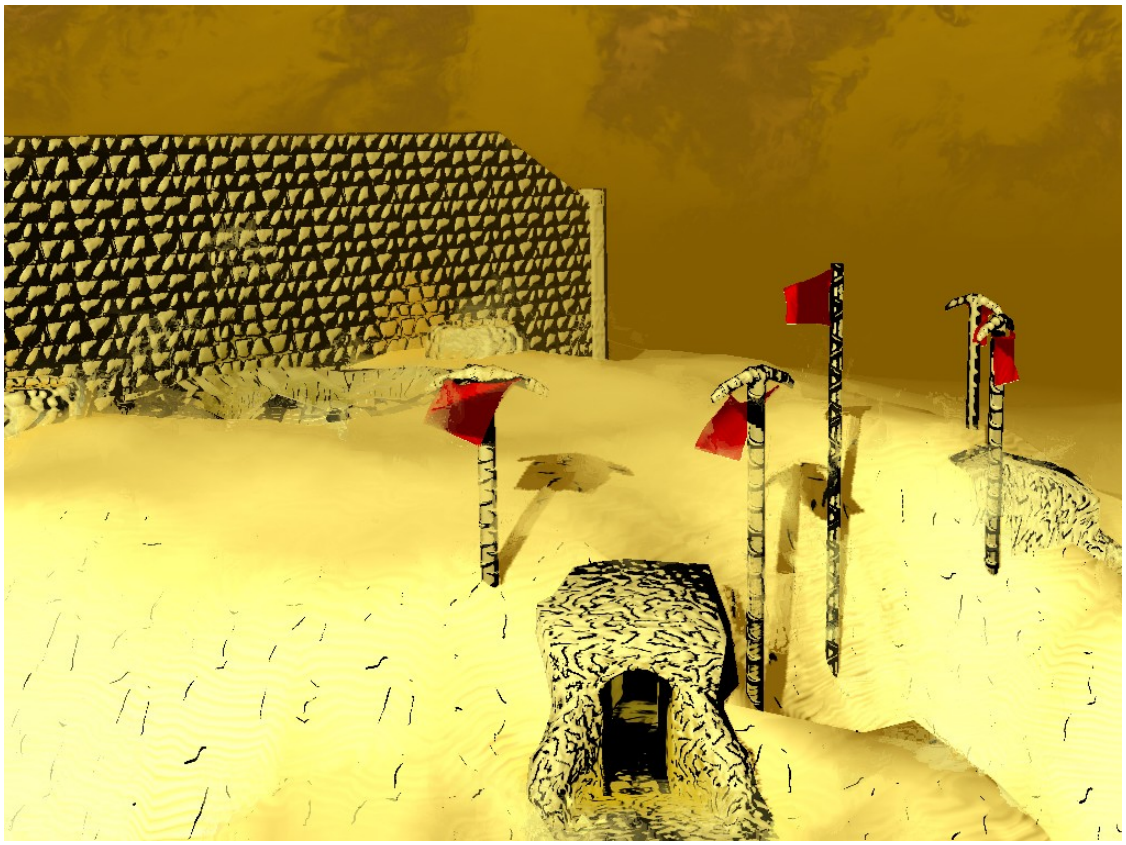
Naruto Narutimate Hero 3

et Team Fortress II

2. Le projet

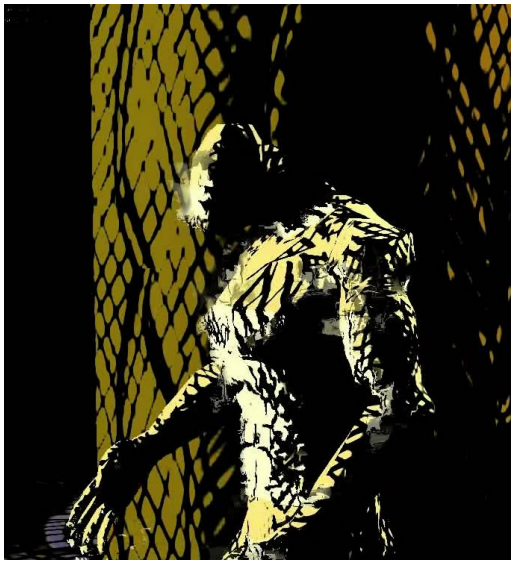
En dehors de la recherche technique opéré dans le cadre de ce Master, il s'agissait également d'aboutir à une réalisation concrète utilisant de manière adéquate les résultats des recherches.

Un univers à donc été mis en place et des graphismes ont été produit dans ce but.
La base de plusieurs des effets étudiés ici est le mouvement. Le type de rendu que nous souhaitons obtenir est un rendu toujours en mouvement, un mouvement provenant de fluides, de personnages, de décors.
L'univers imaginé possède une histoire qui transparait légèrement dans l'application finale. L'univers est dominé par le vent, mais cherche à le contrer. Le vent pousse le sable, qui recouvre la totalité du monde imaginé ici. Le sable forme des dunes qui recouvrent et écrasent toute tentative de construction. Mais certains êtres résistent à ce sable, il s'agit de sorte de démons éthérés qui repousse le sable par leur seul présence.
Afin de pouvoir entreprendre la construction d'une cité importante, le peuple vivant sur cette planète à enfermé un grand nombre de ces démons dans des sortes de temples-prisons qui concentrent la force des démons afin de faire rempart au sable. Mais cela remonte à de nombreuses années, depuis beaucoup de démons se sont échappés, les temples disparaissant inévitablement sur le sable. La cité protégé n'est plus, le peuple à disparu également. Le niveau représenté dans ce projet est l'un de ces temples, encore habité par certains démons et donc résistant tant bien que mal au sable qui règne tout autour. A la manière d'un Myst (de Cyan Worlds Inc), le joueur incarne un individu anonyme, on ne sait rien de son passé, celui-ci n'importe pas, le personnage est maintenant dans cet univers, ne peux pas en sortir, mais il est libre d'explorer ce qui l'entoure et de subir son destin.

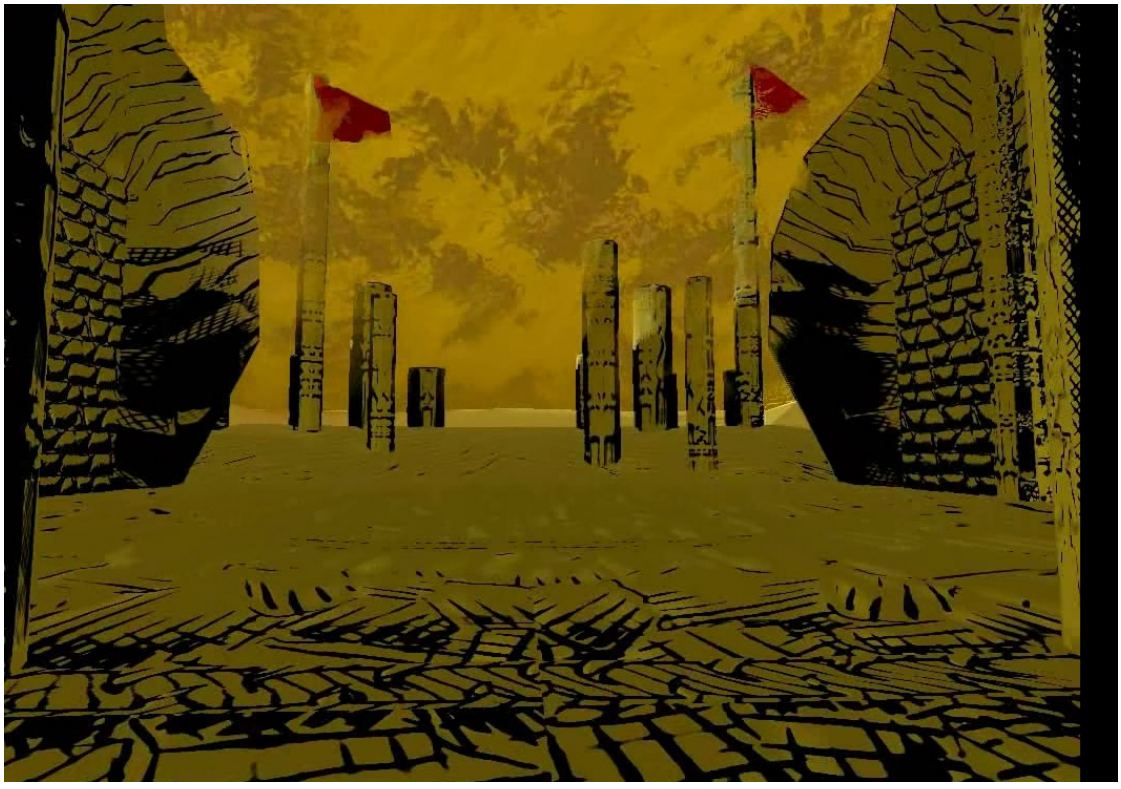


Une caractéristique intéressante à exploiter dans cet univers est le contraste entre les pierres, couloirs, colonnes du temple (petit détail : les colonnes sont générées aléatoirement, ainsi chacune est différente), qui sont très figés, solides, et le sable, le vent, les drapeaux, qui sont toujours en mouvement.

Dans un jeu vidéo complet, le niveau serait étendu à plusieurs temples, qu'il faudrait relier à pied à travers le sable, en évitant tornades et brusques tempêtes de sable, en luttant contre divers démons jaillissant du sable. Au final, le joueur aurais accès aux ruines de la ville centrale, à la recherche de survivants, et notamment d'une personne dont la légende raconte que son réveil sera la fin des sables ...



Au niveau des performances finales obtenues à travers ce projet, l'affichage de tous les effets en même temps est assez gourmand, principalement pour la carte graphique. Nous parvenons tout de même à maintenir une moyenne de 17-18 images par secondes, en ne descendant jamais en dessous de 13. La configuration de test est un ordinateur portable à 2,2 Ghz équipé d'une carte graphique Nvidia Geforce 8800M GTX et de 3 Go de mémoire vive. La carte graphique doit absolument supporter les shaders 3.0 pour faire tourner l'application. L'utilisation reste donc dans des normes acceptables pour un jeu sans action, où la réactivité n'est pas très importante.

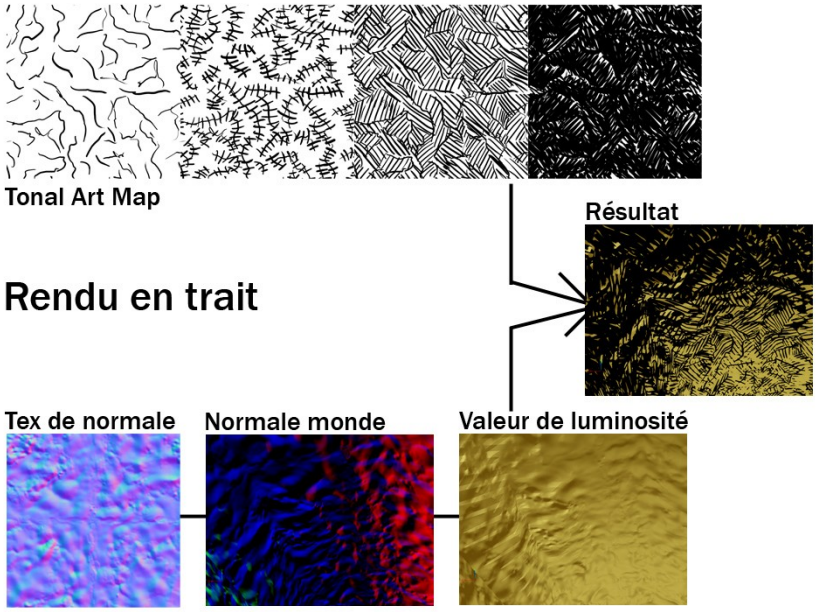


3. Le rendu au trait

Un des éléments très identifiable d'un rendu non-réaliste est l'utilisation de hachures et de traits pour représenter les ombres et les dégradés. Ce type de visuel est directement hérité du dessin traditionnel et du croquis. On le trouve également beaucoup dans la bande dessiné.

Le principe fondamental de la technique présenté ici est d'utiliser plusieurs versions d'une même texture, chacune correspondant à un degré différent de luminosité de la surface.
Ainsi, chaque version sera légèrement plus sombre que la précédente.
On appelle ce type de texture des Tonal Art Map [5].
Ensuite, le programme utilise pour chaque pixel, l'une ou l'autre version de la texture, selon la quantité de lumière reçue par le pixel.
C'est un principe très simple, et qui peux donner de nombreux effets différents selon le nombre et le type de texture utilisée.

Ainsi l'effet actuel est constitué de 4 versions de la texture, toutes en noir et blanc.
La première version (la plus claire) ne dessine que quelques contours, un trait pour chaque creux par exemple. Chaque version ultérieure va rajouter une couche de hachure, ce qui va assombrir au fur et a mesure la texture. Il est important de rajouter les hachures et non de recommencer d'autres hachures complètement différentes pour chaque version. Ainsi, on peux passer d'une texture à l'autre avec une transition douce et logique.
Il existe plusieurs manières de gérer ces transitions, ici nous utilisons simplement une texture et la suivante, modulé selon le pourcentage de luminosité. Nous pouvons aussi utiliser soit l'une soit l'autre complètement, ce qui donne une séparation très visible.



Le rendu obtenu est un peu trop flou, puisque entre deux textures, de nombreux pixels vont passer du blanc au noir en étant gris. Pour avoir un rendu bien délimité et contrasté, il suffit alors de multiplier et de limiter (clamp). L'aspect flou deviens alors très net et permet au traits de devenir plus ou moins fins, et d'avoir des courbes très douces.
Par contre, ce procédé de contraste ne fonctionne que pour l'objet présent, et les courbes ne pourrons pas déborder sur les autres objets, ni se fondre dans des contours d'objets calculés par une autre méthode. Pour obtenir cela, il faudrait ajouter du flou sur l'image finale calculé, et appliquer le contraste ensuite. Cette technique est par contre beaucoup plus lourde.

Les objets étant souvent assez faible en polygones, peu détaillé, ce rendu donnera des lignes de séparations assez franches entre un niveau de luminosité et le suivant. Pour remédier a cela, nous utilisons une texture de normale (normal map), qui va servir a modifier la normale habituelle de la surface lors du calcul de luminosité du pixel. Cela permet aussi d'avoir beaucoup de détails sur un simple plan, par exemple pour faire les pierres d'un mur.

En utilisant deux textures en couleurs et une séparation franche, on obtient un effet de cell-shading assez traditionnel mais très contrôlable.

Il existe plusieurs manières de stocker ces textures multiples.
La première manière est de poser les 4 textures les unes a la suite des autres dans une grande texture. On utilise alors un décalage des coordonnées de texture pour prendre la valeur des deux textures a utiliser pour le pixel courant. Le soucis tiens ici dans le sampling, c'est a dire l'échantillonnage du pixel dans la texture qui prend en compte les pixels adjacents pour fournir une valeur douce. Mais comme nous passons d'un bout à l'autre de la texture, ce sampling est totalement faux, il faut alors renoncer à celui ci, même si avec les shader 3.0, il est possible de fournir explicitement les dérivés utilisée dans le calcul de ce sampling.
Une technique plus simple est plus rapide est d'utiliser des textures 3D. Chaque section de la texture contiendra une des versions de la texture 2D. Le mélange entre une texture et la suivante est alors gratuit, il est calculé directement par la carte graphique.

Un soucis interviens, avec les Tonal Art Maps, lorsque l'on s'éloigne de l'objet texturé.
En effet, si l'on garde la même texture, nous aurons le même phénomène que pour n'importe quelle texture, c'est à dire un clignotement très désagréable, car pour chaque pixel affiché au final, plusieurs pixels de la texture pourront être choisi. Le choix étant plus ou moins aléatoire, le pixel clignote au moindre déplacement de la caméra.
Pour palier à cela, nous utilisons du mipmapping, c'est a dire des versions réduites en taille utilisé lorsque l'on s'éloigne. Cela fonctionne bien, mais le soucis tiens dans l'utilisation de textures 3D.
Le mipmapping est en effet appliqué sur les trois coordonnées, donc si nous avons 4 divisions sur l'axe des Z, afin de stocker nos 4 niveaux de luminosité, nous n'en aurons plus que 2 au prochain mipmapping, puis un seul.

Les textures sont ici composés de traits individuels. Lorsque l'on diminue la texture en taille à travers le mipmapping, les traits deviennent trop petits par rapport au nombre de pixel, et se mélangent. Au final la texture deviens grise ...
La solution acceptable ici est de définir nous même les textures utilisées lors du mipmapping, et ainsi nous n'utilisons que les versions les plus claire de la texture, qui sont donc composé de moins de traits que les plus sombres.
Lorsque la caméra s'éloigne, la texture deviens donc de plus en plus simple, jusqu'à disparaître. Pour conserver une

notion d'éclairage même à grande distance, nous noircissons artificiellement les parties ombrées de l'objet à partir d'une certaine distance, sans plus utiliser la texture.

La génération des Tonal Art Maps et des normals maps associés à été faite grâce à Photophop et le plugin Nvidia normal map, ainsi qu'une utilisation ponctuelle de zBrush.
L'insertion dans Virtools et le choix des mipmaps à été faite par l'intermédiaire du format .dds grâce à DxTex, disponible dans le SDK directX.
Les parties les plus intéressantes du code HLSL sont disponibles dans l'annexe 1.

4. La gestion des lumières

Il existe de nombreuses techniques pour gérer l'illumination d'une scène temps réelle, selon le type, le nombre, la position des lumières.
Le projet présentera deux types d'environnements distincts, des intérieurs et des extérieurs. Chaque type va faire appel à des techniques d'illumination différentes.

4.1 Les lumières en intérieur :

Les environnements intérieurs du projet se caractérisent par un nombre important de lumières à petite portée. Ces lumières sont indépendantes les unes des autres.
Parmi les lumières intérieures, plusieurs types vont être nécessaires : des lumières directes, omnidirectionnelles, sans ombres, des lumières « spot » avec des ombres portées, une illumination globale gérée par lightmap ...

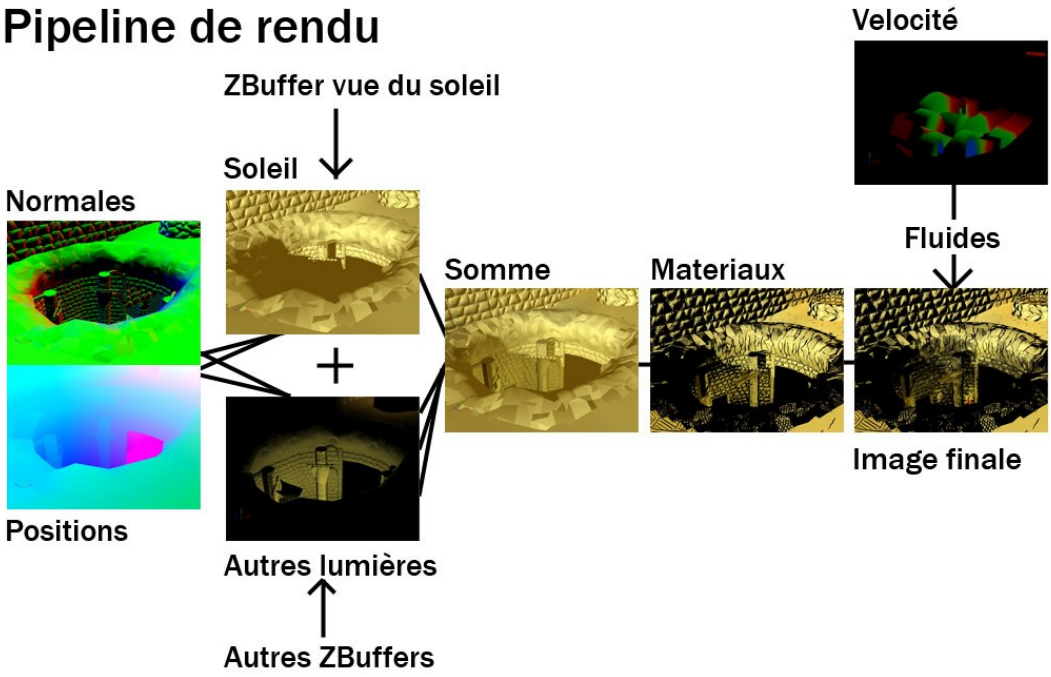
La méthode traditionnelle pour gérer les lumières est simplement d'avoir un nombre limite de lumières par objet, les « n » plus proches et plus lumineuses par rapport à l'objet seront gérée, et les autres n'auront aucun impact. Cela fonctionne très bien pour de petits objets indépendants, une faible concentration de lumière et des objets qui ne se déplacent pas trop à des endroits ou le nombre de lumières est limite et ou les lumières utilisée pourraient changer d'un coup.
Le shader de chaque objet recevra une liste des positions et intensités des lumières les plus proches. Cette liste est en général fourni automatiquement par le moteur 3D. Il est possible d'utiliser différents shader selon le nombre de lumières proches, permettant d'optimiser grandement les calculs.
Pour la gestion des ombres portées de ces lumières, la situation se complique. Les calculs sont bien sur plus importants, et il faut pouvoir récupérer (si l'on utilise le shadow mapping) une matrice de projection et une texture de profondeur par lumière. Il est alors assez difficile de gérer l'envoi de ces textures de manière automatique selon les lumières sélectionnées par objets (ce n'est pas fait automatiquement par le moteur 3D de Virtools).
Les calculs sont également assez redondant, puisqu'ils sont fait par objet, ce qui signifie que les pixels masquées par des objets située devant seront tout de même calculé. Une bonne gestion de l'occlusion par le moteur, ou bien un premier test de profondeur permet de réduire cette dépense inutile.
Dans notre cas, il s'agit surtout d'éclairer le décor, constitué d'éléments de grande taille. Cette méthode n'est donc pas très adéquate. De plus, nous souhaitons mélanger des lumières avec et sans ombres, choisir les objets ombrées par chaque lumière ...
Une solution est donc de chercher a gérer l'éclairage par lumière et non par objet.

Le processus est assez simple et repose sur la technique dite du « deferred shading » [11], c'est a dire éclairage différée, appliqué en plusieurs couches séparant le calcul.
La première étape est de rendre l'ensemble des objets à éclairer dans une texture en enregistrant, pour chaque pixel, la position dans le repère global, et dans une seconde texture, la normale de chaque pixel également dans le repère global. L'implémentation classique rend également la couleur et autres informations (spéculaires, ...) dans une troisième texture.
Ensuite, pour chaque lumière, nous allons faire le rendu d'un simple polygone recouvrant tout l'écran et permettant, par l'intermédiaire d'un shader, d'effectuer des calculs.
Les calculs effectués dépendront du type de lumière.
Dans le cas simple d'une lumière omnidirectionnelle, le calcul sera

simplement de prendre, pour chaque pixel, la position et la normale dans les textures calculées précédemment, et d'appliquer le calcul de lumière diffuse traditionnel :

intensité lumineuse = produit scalaire (direction de la lumière , normale)

Le calcul sera donc fait uniquement pour les pixels visibles par la caméra principale, un calcul par pixel de l'image finale et par lumière.
Divers moyens très simples permettent de réduire le nombre de pixel calculée par lumière. Chaque lumière va être représentée par un objet. Pour une lumière omnidirectionnelle, la forme sera une boule (par forcement très détaillé, un simple carrée peut déjà suffire), pour une lumière « spot » ce sera une pyramide. Les pixels calculées pour chaque lumière seront donc ceux couvert par cet objet. Si l'objet est complètement hors de la vue, l'objet n'est même pas calculé, et ceci de manière automatique par le moteur 3D. Il suffit donc de faire un simple rendu de la scène, en n'affichant que les objets correspondants aux lumières (en désactivant tout de même le test Z, afin de gérer le fait d'avoir plusieurs lumières se recouvrant), en utilisant de l'alpha blending, c'est a dire que chaque lumière va ajouter son impact sur une texture initialement totalement noire.
Cette technique, très efficace dans notre cas, n'est possible que depuis que les cartes graphiques gèrent les textures flottantes. Auparavant, même si tous les calculs sur la carte graphique était fait en virgule flottante, le résultat était remis dans le domaine du réel, chaque composant (rouge vert bleu et alpha) ne pouvant avoir pour valeur qu'un chiffre entre 0 et 255.
Avec les textures flottantes 128 bits, chaque composant est codée sur 32 bits flottants ce qui est très très précis, et permet de stocker une position par exemple.
Le calcul des ombres, avec notre technique, se fait par lumière et non par objet.
Il est possible de rendre les passes de positions et de normales à travers un seul appel, en utilisant la technique du Multiple Render Targets, se qui permet d'éviter de transformer deux fois les polygones. Des tests sous Virtools n'ont pas permis de gagner plus de 1 fps, ce qui relativise l'intérêt pour un petit projet.



Certains détails sont à prendre en compte, ainsi il faut rendre les objets représentant les lumières en « backface », c'est a dire uniquement les faces arrières des objets. Cela permet de gérer les cas ou la caméra est à l'intérieur d'une lumière.

Au final, nous avons donc une texture regroupant pour chaque pixel de l'écran, l'accumulation de la couleur et de l'intensité des lumières.

Chaque objet devant être éclairé va alors faire appel à cette texture pour connaître la couleur et l'intensité de la lumière. Ce qui peux être difficile c'est que l'on à pas directement disponible la direction de la lumière, mais simplement la couleur et l'intensité. Il est possible d'avoir, avec un processus plus complexe, la direction de la lumière la plus forte, pour chaque pixel, mais il est plus simple de s'en tenir à l'équation simple de la lumière, qui n'en à pas besoin.
Nous avons ici une différence par rapport à l'implémentation habituelle. Au lieu de simplement multiplier la passe de diffuse déjà rendu et celle de luminosité, nous avons besoin de rendre une seconde fois les objets qui utilisent le rendu en trait. En effet, la texture utilisée ne peut pas être sélectionné avant de connaître la luminosité.

4.2 Les ombres portées :

Les ombres portées en temps réel peuvent être globalement gérée de deux manières opposées :

Les stencils shadows :
L'idée est d'extruder les objets qui cachent la lumière, dans la direction de la lumière, ce qui permet au final de savoir pour chaque pixel, s'il est a l'intérieur d'un de ces objets extrudées, et donc dans l'ombre ou non. Le résultat est alors des ombres très dures, très lisses et carrées. Cette technique est bien adapté pour des lumières omnidirectionnelles.

Les shadows maps :
Ici, il s'agit d'effectuer un rendu des objets qui cachent la lumière, vue depuis la lumière, en stockant dans une texture la distance séparant la lumière du premier objet visible. C'est tout simplement un z-buffer. Ensuite, dans le calcul de la lumière vue par la caméra normale, pour chaque pixel, on calcule la position de ce pixel vue par la lumière, on récupère alors la distance du premier objet visible et on compare avec la distance actuelle. Si celle ci est plus grande, le pixel est dans l'ombre. Le principal soucis est la forte pixelisation de l'ombre, liée à la taille de la texture de z-buffer.
Les ombres peuvent être adoucis en utilisant plusieurs appels à la texture autour du point calculé et en utilisant la moyenne de ces résultats pour ombrer le pixel. Ce type d'ombrage fonctionne bien avec des lumières spots, le calcul de la texture de Z se faisant depuis une caméra représentant le spot. Pour des lumières omnidirectionnelles, il faut faire 6 rendus, afin de générer une cubemap de distance Z dans les six directions (avant, arrière, gauche, droite, haut et bas), ce qui est plutôt lourd [8].

Dans ce projet, les shadows maps vont être plus facile à utiliser, notamment car nous aurons beaucoup de lumières à faible portées.
Les ombres portées donnaient un aspect très pixelisé au rendu, mais une solution très simple, et directement adaptée à notre style graphique va permettre de masquer ces artefacts.
Il s'agit en fait de décaler légèrement l'endroit ou nous allons faire le test de profondeur pour savoir si l'on est dans l'ombre ou la lumière. Ce décalage aura une forme sinusoïdale, ce qui fonctionne parfaitement avec un rendu haché tel que le notre. Pour obtenir des hachures dans les deux sens, il suffit de faire deux appels à la texture, l'un utilisant une sinusoïde sur X et l'autre sur Y; le degré d'ombre sera la moyenne de ces deux résultats.
Tout d'abords, ce décalage était calculé dans le repère de la caméra, et lorsque celle-ci bougeait, les hachures changeaient continuellement de forme, ce qui n'était pas tout à fait dans le style voulu.
La solution est d'utiliser la seule valeur qui changera pour chaque pixel calculé tout en restant cohérent avec le déplacement de la caméra, c'est a dire la position dans le repère global, du pixel calculé.
C'est une position en trois dimensions, hors nous avons besoin d'un décalage en deux dimensions.
Un premier test avec le calcul suivant : $décalageXY = positionXY * positionZ$ donne déjà de bons résultats.

Le dernier type de lumière est la radiosité, ou l'illumination globale, qui est efficacement géré par un lightmap, c'est a dire une texture contenant la lumière pour chaque point du décor, pré-calculé par un logiciel comme Maya par exemple. Ce procédé ne fonctionne que pour un décor fixe et des lumières fixes, puisque tout est pré-calculé.
Il peux être très aisément intégré dans notre architecture avec une passe globale affichant les objets avec leur texture de lightmap et l'ajoutant à la texture de luminosité.

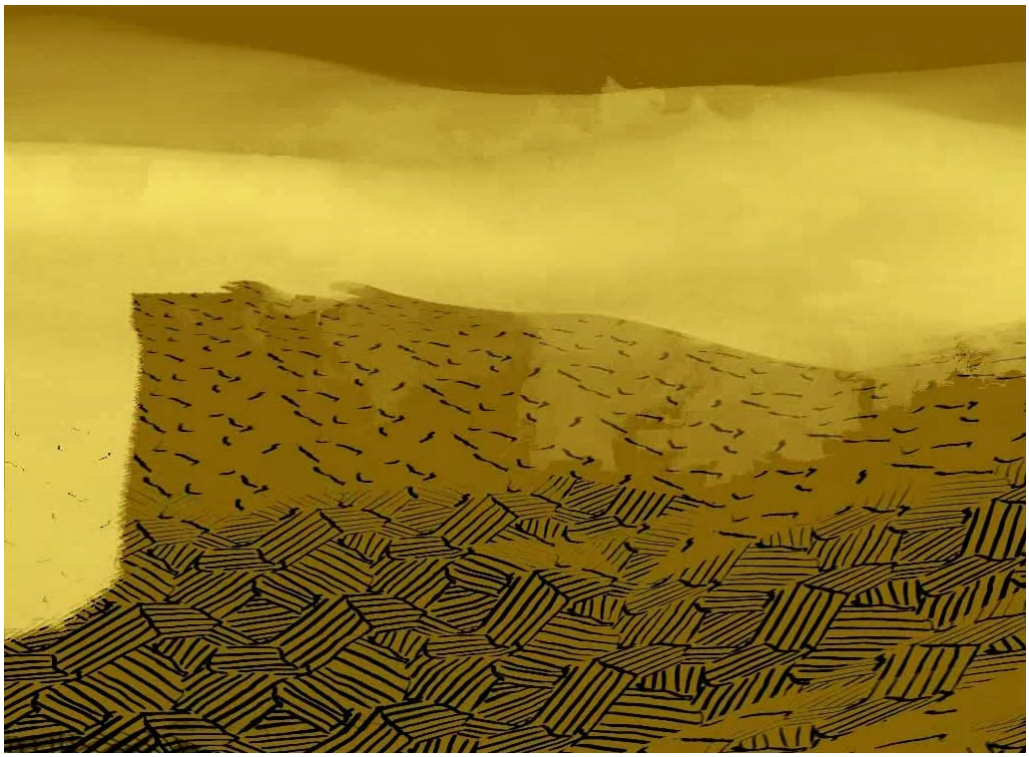
4.3 L'éclairage en extérieur :

Le problème est ici assez différent, puisqu'il n'est plus vraiment possible de gérer parfaitement toutes les ombres, car le paysage est trop ouvert et vaste. Le type de lumière principal est une directionnelle, c'est à dire la lumière du soleil.
L'avantage d'une lumière directionnelle, est que l'on peut modifier la position de la caméra représentant la lumière sans influer sur la lumière, puisque les rayons sont parallèles.
La solution la plus immédiate pour calculer la lumière du soleil est donc de positionner une caméra orthographique qui va se déplacer avec la caméra principale et pointer vers le sol. On peux alors utiliser la technique de la shadow map, en gérant par un dégradé les bords de la caméra.
Les objets proches de la caméra projetterons donc des ombres, et pas les objets plus lointains.
Il existe, dans ce domaine, de nombreuses recherches visant à mieux gérer le calcul de la shadow map. En effet, dans certains cas, par exemple celui où la caméra principale et la lumière sont dans des directions opposées, la shadow map sera très inefficace, puisque l'endroit où le plus de pixel serons à calculer, c'est a dire les endroits proches de la caméra, seront aussi ceux où la shadow map consacrera le moins de pixels (puisqu'ils sont loin de la lumière). Il est possible de déformer la matrice de projection de la caméra calculant le z-buffer de la lumière afin de privilégier les endroits les plus proches de la caméra principale. Ce type de technique peut être très efficace mais mène également à des problèmes de variations brutales de certains pixels lors du déplacement de la caméra (des « flicks »).
Comme nous calculons un terrain, nous pourrions également utiliser une caméra orthographique vue du dessus pour faire le calcul de l'ombre, et ensuite appliquer un flou afin d'avoir des ombres douces.

Il existe une autre technique, liée au fait que nous calculons la luminosité d'un terrain, c'est a dire en général d'un quadrillage régulier ou seul la hauteur de chaque sommet varie.
Souvent, cette hauteur est stockée dans une texture de hauteur (height map), particulièrement adaptée à un quadrillage régulier. Il est alors possible de calculer un lancé de rayon, depuis chaque pixel de la texture de hauteur, dans la direction de la lumière, et pour chaque pixel sur cette ligne, comparer la hauteur du terrain et celle de la lumière. Si le terrain est plus haut, cela signifie que le pixel calculé est dans l'ombre. Ce calcul peux être totalement fait sur le processeur graphique.
L'avantage de ce calcul, est de pouvoir appliquer ensuite un flou sur cette texture, ce qui donne des ombres douces.
Le calcul de l'éclairage direct de la scène par le soleil va utiliser une technique similaire a celle des éclairages intérieurs, mais l'effet sera appliqué directement à tous les pixels de l'écran.
Nous utilisons donc une camera orthographique positionné dans la direction du soleil pour effectuer un rendu de profondeur, calculé cette fois par rapport au plan de cette caméra, et non par rapport à une position, puisqu'il s'agit d'une camera orthographique, donc sans réel point d'application.
Pour chaque pixel de l'écran, on teste alors sa distance de la même manière que pour une shadow map classique.

L'utilisation du « deferred shading » permet d'utiliser autant de lumières que l'on souhaite, où presque. Seules les lumières avec ombres, et principalement les omnidirectionnelles, vont être lourdes à calculer. Une trentaine de lumières sans ombres pour une image sont facilement géré, accompagné de trois ou quatre lumières ombrés.
Un élément qu'il reste à intégrer avec cette architecture est le High Dynamic Range (HDR), c'est à dire calculer l'accumulation de lumières avec une très grande précision, puis ajuster le niveau de sortie selon la luminosité perçu par la caméra. Le seul soucis ici tiens dans l'incapacité, avec les cartes graphiques actuelles, de gérer l'alpha blending sur des textures flottantes, c'est à dire le cumul d'une lumière sur l'autre avec une certaine transparence.

Les parties les plus importantes des shaders HLSL dédiés aux calculs des ombres des lumières spots, omnidirectionnelles et directionnelles sont disponibles en annexe 2.



5. Gestion dynamique de terrains

Il existe un champ d'investigation à part dans le domaine du jeu vidéo, qui à été et sera encore extrêmement approfondi et réfléchi, c'est celui du rendu de terrains.

C'est aussi un point qui a beaucoup divisé les moteurs de rendu. Certains étaient prévu pour gérer des scènes en intérieurs, (typiquement HalfLife ou Quake), et d'autres au contraire, ne géraient que des environnements extérieurs (moteurs de jeux de stratégies, de project IGI, de FarCry)

Il faut savoir que les techniques qui vont permettre, dans les deux cas, d'accélérer le rendu sont totalement différentes, et souvent opposées.
Dans le cas de niveaux intérieurs, la meilleure manière d'optimiser est en général d'utiliser un système de portails (de type BSP tree par exemple) qui va permettre de n'afficher que la salle actuelle ainsi que celles qui sont directement visibles. Au contraire, dans un environnement extérieur, tout est potentiellement visible, il va falloir gérer la disparition des éléments à l'horizon ainsi qu'un LOD qui va permettre de réduire la complexité des éléments lointains. De plus, les niveaux intérieurs sont très découpés, nous pouvons facilement ne charger que les pièces proches, et charger le reste au fur et a mesure que le joueur avance (streamming). Au contraire, avec un terrain ouvert, il est difficile de séparer les objets, et donc difficile de les charger au besoin.

La gestion des lumières est également très différente, dans un cas, celui du niveau intérieur, ce sont les lumières ponctuelles et à faible distance qui sont majoritaires. Il y aura peu de lumières différentes mais celles-ci changeront de pièce en pièce. Avec un terrain, il y a en général une lumière principale (le soleil) et des lumières secondaires éventuelles. Les lumières secondaires seront difficile à gérer, puisqu'il sera couteux de savoir quels objets sont concernés par telle ou telle lumière.

Dans notre projet, nous avons les deux types de décors qui s'entremêlent, il va falloir gérer les transitions.

5.1 La gestion dynamique du terrain :

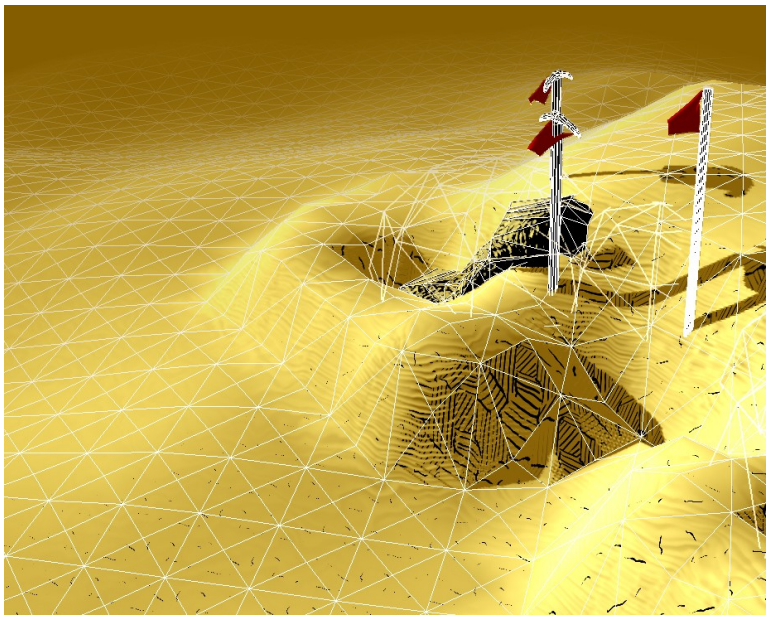
Un terrain est à une l'échelle gigantesque et, en général, c'est une seule surface, sans recouvrements d'une partie du terrain passant au dessus d'une autre.

Une technique qui est beaucoup utilisé pour la génération d'un terrain est celle de la texture de hauteur (heightmap). Il s'agit d'utiliser une image en niveau de gris, chaque pixel décrivant la hauteur d'un sommet de la grille qui formera le terrain.

Une spécificité du terrain que nous devons implémenter pour ce projet est qu'il doit être dynamique, c'est a dire que certaines parties du décors vont bouger et onduler, formant des dunes mouvantes.
La position de la lumière venant du soleil sera elle aussi changeante.
La taille du terrain a calculé devra être variable, notre terrain sera infini et sans bords.
La plupart du temps, les terrains sont fixes, le maximum de chose est pré-calculé : positions des sommets, textures du sol, ombres ...
Ici, nous devons mettre a jour le sol en temps réel.

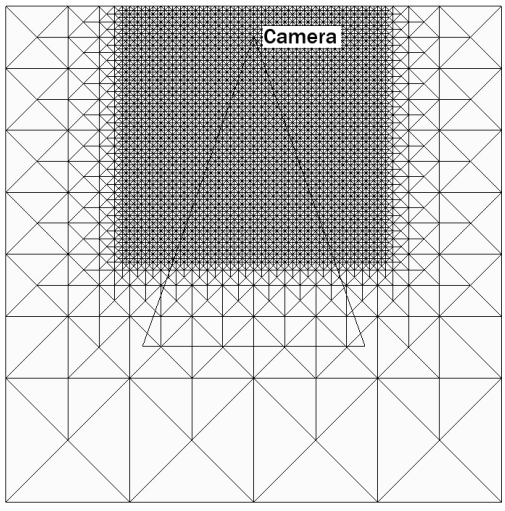
Nous avons testé de nombreuses techniques qui ont des avantages mais aussi des inconvénients.
La première manière de faire est de calculer sur le processeur principal les hauteurs des points d'un objet fixe, et d'envoyer ces positions à la carte graphique pour chaque image Le bon fonctionnement de cette technique va dépendre essentiellement de la manière dont on construit cet objet fixe.
La seconde technique est d'utiliser une des nouvelles fonctionnalités introduit dans les vertex shader 3.0 et qui

permet de faire appel à une texture. On peux donc envoyer les données concernant les hauteur via une texture, et la carte graphique se chargera toute seule de calculer les positions finales des sommets. Pour des raisons de simplicité, c'est la première technique qui à été retenu. Des essais avec la seconde technique n'ont pas été très concluants du point de vue des performances.



5.2 La construction d'un maillage adapté :

Avec notre technique, il est possible d'obtenir un terrain potentiellement infini.
Comme nous calculons les positions des sommets à chaque image, avancer dans le terrain peut aussi se traduire par changer les coordonnées de ces sommets afin qu'il apparaissent comme si nous avions avancé. [12]



L'idée est en fait d'avoir un maillage qui est plus ou moins fixé à la caméra et qui bougera avec elle.
C'est une idée qui est très utilisé dans le rendu d'étendues d'eau et qui permet d'avoir toujours l'étendue devant soi alors qu'elle ne fait en réalité que quelques unités de long.
La manière dont on fixe l'objet à la caméra peu varier. Il est possible de le fixer totalement (du moins en rotation sur l'axe Y c'est a dire la hauteur), ce qui fonctionne très bien, mais va poser problème si le maillage n'est pas assez défini (c'est a dire qu'il n'a pas assez de divisions), notamment dans le calcul de la lumière, les normales des faces changeants au moindre mouvement de caméra.
Nous avons donc choisi une manière plus souple de fixer l'objet à la caméra. En fait l'objet se « snappe » à la position, c'est à dire qu'il se décale de la taille d'une de ses divisions dés que la caméra le permet.
Visuellement, rien ne change lorsque le terrain avance d'un cran, à part le bord du fond du terrain qui augmente brusquement d'un cran.
Pour réduire encore la taille du terrain qu'il sera nécessaire de calculer, nous pouvons également « snapper » en rotation. A chaque fois que la caméra se tourne de plus de 90 degrés, le terrain se décale lui aussi de 90 degrés.

Avec cette technique, nous pouvons avoir un terrain infini, qui peux éventuellement boucler sur lui même (comme une planète)

Au départ, nous souhaitions avoir un maillage dont la complexité se réduise en fonction de la distance. Ainsi, les portions de terrain lointaines ne seront pas aussi bien définies que celle qui sont proches. Cela permet de faire moins de calculs. Malheureusement, cela ne fonctionne pas bien avec la technique de « snappe » du terrain, puisque les portions lointaines ne se répètent pas autant que les portions proches.
La typologie du maillage d'un terrain pourrait faire l'objet d'un chapitre entier. La problématique est d'obtenir un maillage qui rende le mieux possible et qui supporte en même temps différents niveaux de détails.

5.3 Détails d'implémentation :

Le calcul de la déformation du terrain va demander une grande rapidité d'exécution.
Il est pratiquement impossible de le faire directement dans le logiciel Virtools, avec l'aide des boites de constructions basiques. Il est possible de l'écrire en VSL, un langage propre à Virtools, mais limité en vitesse, fonctionnalités et fiabilité.
Le langage idéal à utiliser est donc le C++, à travers le SDK (Software Development Kit) de Virtools, qui va nous permettre d'accéder pratiquement à n'importe quel élément de la scène, de manière très rapide. Pour les calculs en eux même, le c++ est un langage très adapté, flexible et rapide.

Au niveau de l'organisation du code, nous avons fait très fortement appel à une organisation inspiré des classes de politique (policy classes).
L'idée derrière cette organisation est de regrouper les fonctions, concernant un algorithme par exemple, dans une même classe abstraite, c'est à dire qui ne fait que définir les entêtes des fonctions. L'implémentation concrète sera écrite dans des classes enfants qui héritent de la première et surchargent les fonctions nécessaires. Nous pourrons écrire autant de classes enfants qu'il y a de version de l'algorithme. L'avantage de cette organisation est que l'on peux faire appel indépendamment à tel ou tel algorithme de manière transparente, sans que le code ait besoin de savoir la version utilisé.
Habituellement, les classes de politique sont traités à la compilation. Le compilateur va utiliser les paramètres templates que le programmeur à choisi pour générer une version concrète de la classe à utiliser.
Nous avons transformé cet aspect afin que le traitement soit fait en runtime, c'est à dire pendant que l'application tourne. Ainsi, il est possible de changer l'algorithme utilisé à n'importe quel moment, ce qui était impossible auparavant.

Afin de rendre l'explication plus compréhensible, nous allons prendre l'exemple de la génération du maillage du terrain.
De nombreuses méthodes peuvent être utilisées pour générer ce maillage, et nous ne pouvons par avance savoir laquelle sera la meilleure. Au lieu de remplacer chaque méthode par une nouvelle en perdant l'ancienne, notre architecture permet de conserver toutes les versions, en gardant la compatibilité entres elles, et nous pouvons changer de méthode en modifiant une seule ligne de code.
Voici une version simplifié du code permettant de comprendre le principe :

```
class CFunClassMeshRender
{

public:

   virtual void   Create                  (unsigned int uiSizeX, unsigned int uiSizeY)  = 0;
   virtual void   Draw                    ()  = 0;

   virtual CK3dEntity*   Get3dEntity      () = 0;
};
```

Ceci est la classe parente, elle défini trois fonctions. La première sera appelé à la création du maillage (les paramètres concernent la résolution du maillage), la seconde pour l'afficher, et la dernière pour récupérer l'objet créée.
Ensuite nous avons écrit une classe enfant par technique de création du maillage.
Ainsi nous avons une classe qui va utiliser un maillage existant en se contentant de le déformer, une autre classe qui ne va pas créer de maillage mais au contraire envoyer directement à chaque image de nouveaux polygones à la carte graphique, et une dernière qui va créer se maillage par divisions successives d'un carré.
Ensuite, nous pouvons créer un conteneur pour un élément de ce type en utilisant le code suivant :

```
CFunClass<CFunClassMeshRender>      m_meshRender;
```

Puis, a tout moment, nous pouvons utiliser la ligne suivante pour indiquer la technique qui sera dorénavant utilisé :

```
m_meshRender.Set<CFunClassMeshRenderEntity>();
// pour utiliser la variante déplaçant directement une entité
```

L'appel à une fonction de l'algorithme se fera alors ainsi :

```
m_meshRender->Draw();
```

L'aspect générique de la classe utilisé est donc complètement masqué à l'utilisation, on ne change pas la syntaxe du code, comparée à un appel classique à une classe concrète.

Cette méthode va également être utilisé intensivement dans le calcul de la hauteur d'un sommet en fonction de sa position dans l'espace.
Voici quelques fonctionnalités dont nous avons besoin pour ce calcul : appels aux valeurs d'une texture, utilisation de fonctions mathématiques (sinusoïdes, multiplications ...) ou bien encore opérateurs de changement de coordonnées.
Toutes ces fonctions vont être regroupé sous une seule classe parente, appelé Input.
Essentiellement, cette classe possède des entrées et des sorties (c'est à dire des liens vers d'autres objets Input). Elle possède aussi une fonction GetValue, qui prend une position en entrée et nous renvoie en sortie la valeur de hauteur calculé. Une fonction nommée Update s'ajoute encore, qui va permettre de mettre à jour certaines valeurs qui changerais en fonction du temps par exemple.

Les différents inputs vont être branchées les unes aux autres.
Ainsi, au final, le calcul simplifié pourra être par exemple :

```
hauteur = lerp( texture1, texture2, sin(time))
```

Chaque fonction sera implémenté dans sa propre classe, que ce soit lerp, les appels à la texture ou la sinusoïde.
Cette manière d'architecturer les fonctions peux sembler très complexe, surtout pour un projet relativement simple. Néanmoins, et au dela du simple intérêt de l'apprentissage du langage de programmation, cette technique permet de vraiment bien séparer les parties du code dans des modules indépendants. Une fois une classe écrite, cette architecture garantie qu'elle sera toujours utilisable en restant compatible avec les modifications apportées aux autres endroits du code.
Il faut également comprendre que les compilateurs c++ modernes sont très performants et arrivent à optimiser grandement les appels de fonctions complexes, passant par diverses classes abstraites. Ainsi, le surcout de l'utilisation de cette architecture, en terme de vitesse d'exécution, est pratiquement imperceptible. Le gain en flexibilité est par contre très important.

6. La simulation de fluides

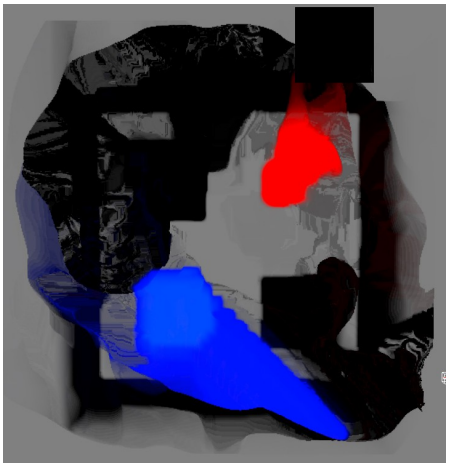
La simulation de fluide regroupe de nombreux effets.
L'eau sous toutes ses formes (cours d'eaux, surfaces océaniques, gouttes ...), les particules de très petites tailles (fumées, nuages ...) et bien d'autres effets.
Traditionnellement ces éléments sont imités par des techniques totalement différentes des lois physiques réelles, mais avec les nouvelles cartes graphiques, il devient possible de simuler des fluides efficacement et à grande échelle en utilisant des lois physiques réelles.

On peux distinguer deux grandes manières de simuler des fluides :
soit en prenant comme élément de base une particule, qui va pouvoir se déplacer selon certaines lois et en fonction de ses voisins, soit de manière discrète, c'est a dire en utilisant une grille de valeurs qui vont représenter le fluide comme un véritable champ de force.
La première manière se rapproche beaucoup de la simulation de tissu, abordé dans chapitre 7. La difficulté qui s'ajoute ici est de pouvoir trouver les voisins de chaque particule. Dans le cas d'un tissu, il suffit de prendre les voisins dans la texture, mais ici, chaque autre particule est potentiellement voisine et interagi avec la particule actuelle. Une manière de régler ce soucis pourrait être un algorithme de tri des particules par proximité, qui s' exécuterait de manière récursive, les particules ne changeant pas de voisine a chaque image. Chaque passe échangerais deux particules, par exemple. [14] [15]
Dans ce chapitre, nous allons nous intéresser uniquement à la seconde méthode.

6.1 Algorithme de base :

La grille utilisé peut être soit en deux dimensions, soit en trois dimensions.
La grille va représenter, en fait, un champ de valeur continue. On peux connaître l'intensité d'un champs en interpolant a partir des quatre valeurs de la grille les plus proche.
Comme nous utilisons une grille, il faut bien comprendre que l'espace couvert par celle ci va être limité, le fluide ne pourra pas se balader n'importe où, mais restera limité au domaine couvert par la grille.

La simulation de fluide est en général constitué d'un fluide de transport et d'un fluide transporté.
Pour de la fumé, le fluide transport sera l'air et le transporté sera une valeur représentant la densité de fumé. Pour de l'encre, le fluide transport sera l'eau et le transporté sera l'encre.



L'algorithme de base se constitue ainsi :
Une grille va contenir la vélocité du fluide transport, une autre va contenir l'intensité (la pression) du fluide transport et une dernière celle du fluide transporté. [16] [17]

Le procédé fondateur de la simulation est nommé : « l'advection »
C'est le fait de transporter une valeur selon la vélocité du fluide transport.

Ainsi, l'intensité du fluide transport va se modifier, se déplacer, en fonction de la vélocité, de même pour le fluide transporté.
La vélocité va elle aussi se déplacer en fonction d'elle même, c'est ce que l'on nomme « l'auto-advection ».

Le second procédé indispensable va nous permettre de modifier la vélocité et fonction de la pression du fluide transport.
Pour connaître la pression à un endroit, il suffit en fait d'utiliser la grille d'intensité du fluide transport, qui représente donc la quantité de fluide à un endroit. S'il y a une différence entre deux cases adjacentes, cela signifie qu'il existe une force entre ces deux cases, dont l'intensité dépend de la différence entre les deux valeurs, le sens allant toujours du plus haut vers le plus bas (de la surpression vers la dépression). Cette force est donc ajouté à la vélocité.

Il existe plusieurs autres étapes plus ou moins dispensables et plus ou moins justifiés physiquement que l'on peux ajouter ici.
Ainsi il existe un calcul nommé « vorticity confinement » qui va chercher à calculer la différence de vélocité entre deux cases adjacentes afin d'ajouter une sorte de friction, qui va créer de petites volutes agréables à l'œil.

Bien sur, il faut également calculer la friction, une déperdition, qui existe toujours quelques part (second principe de la thermodynamique) et qui va mener le fluide vers un état plus stable.

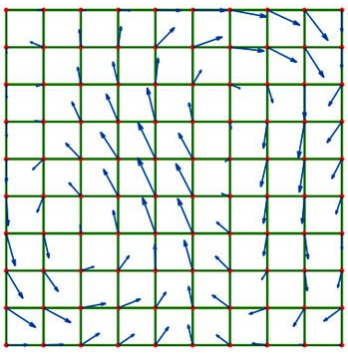
Il est possible de transporter d'autres valeurs que des informations de couleur ou de quantité de peinture. Ainsi, la création de certains effets de combustion vont nécessiter de représenter la température. La température, transporté au grée des courants d'airs, va dans certains cas influencer l'écoulement de celui-ci, avec des appels d'air des endroits les moins chauds. On peux également imaginer des explosions si la température est trop élevé en un point.
Si l'on souhaite simuler la formation de nuages, il est possible de prendre en compte une valeur d'humidité, qui va influencer la formation des nuages. [20]

6.2 Le calcul de l'advection :

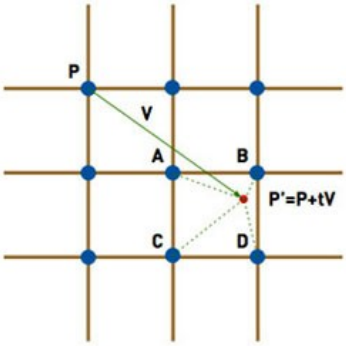
Pour calculer l'advection, il existe deux manières, l'advection avant (forward advection), et l'advection arrière (backward advection).

L'advection avant va, pour chaque point de la grille, calculer l'endroit où doit se déplacer ce point (c'est a dire l'endroit où va aboutir la quantité présente à ce point), et ajouter a cet endroit la quantité approprié.

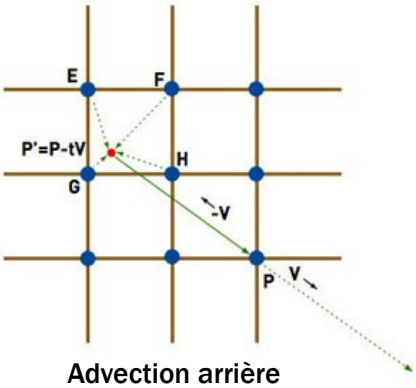
L'advection arrière va faire le contraire, c'est a dire que pour chaque point de la grille, nous allons calculer l'endroit d'où viens la quantité qui devrais s'ajouter ici, et l'ajouter.



Grille de vecteurs de vélocité



Advection avant (gamasutra.com)



Advection arrière

La seconde manière de faire est en fait idéale pour une carte graphique, alors que la première pose plus de problèmes.

En effet, avec la seconde, il suffit, dans le pixel shader, de regarder la valeur de la vélocité à cette endroit, de l'utiliser pour remonter (donc dans le sens inverse de la vélocité) au point source estimé, et faire un second appel à cet endroit dans la texture à calculer, pour connaître la nouvelle valeur. De plus l'interpolation entre les quatre pixels autour de la

valeur dans la texture à calculer est automatiquement fait par le sampling de la carte graphique.
Au contraire, la première technique demande à inscrire le résultat dans un autre pixel que celui utilisé au départ, ce qui n'est pas évident. Il faut même dans l'idéal inscrire le résultat interpolé dans quatre pixels, puisque nous souhaitons inscrire le résultat à un endroit intermédiaire entre ceux-ci.

Tout semble alors assez simple, il suffit d'utiliser l'advection inverse seule pour transporter n importe quelle valeur. Mais il se pose alors le soucis de l'incompressibilité du fluide.
En effet, la plupart des fluides de transport sont pratiquement incompressibles, comme l'eau et même l'air à une faible compressibilité.

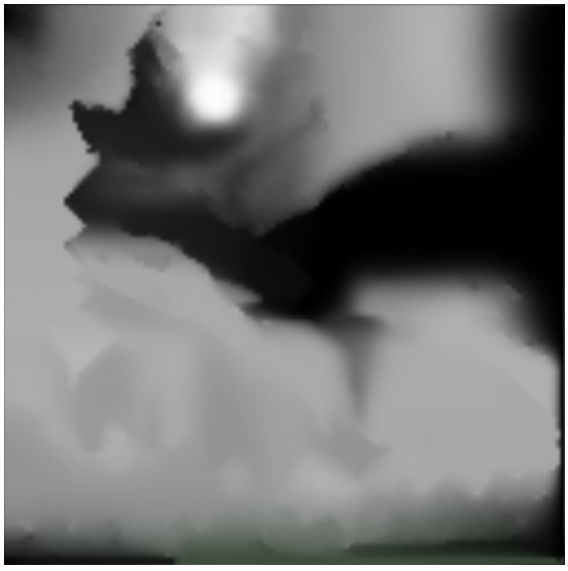
En utilisant l'advection inverse, plusieurs endroits vont en fait utiliser le même point de source pour connaître leur nouvelle valeur, ce qui logiquement peux doubler la quantité de fluide. Intuitivement, on comprend bien qu'il faudrait que chacun des deux points de destination utilise la moitié de la valeur de départ, mais les calculs sont indépendants et il est difficile de connaître le nombre de points de destinations.

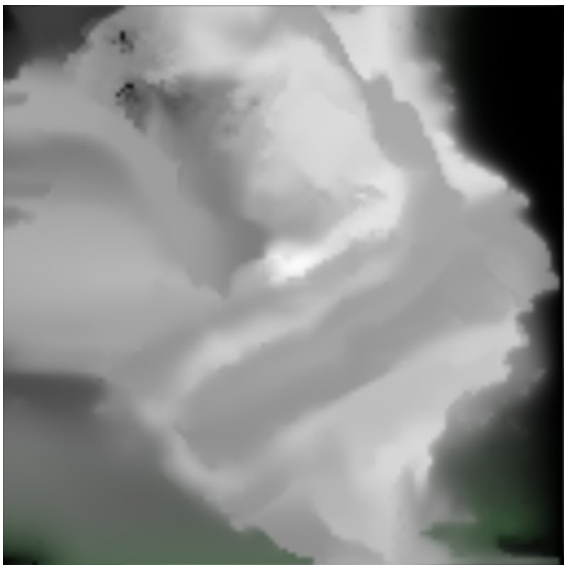
Il faut alors forcer l'incompressibilité du fluide par un autre moyen, en diluant en quelques sorte le fluide.
Dans l'algorithme que nous avons utilisé en première lieu, codé à partir de démonstrations de NVIDIA, cette incompressibilité était obtenue par un algorithme très complexe et gourmand nécessitant une intégration. Afin d'utiliser le parallélisme de la carte graphique au maximum, il existe un algorithme qui décompose cette intégration en plusieurs passes successives (intégration de Jacobi), plus le nombre de passes est important, plus le résultat sera précis et réaliste.
Pour avoir un bon résultat, le calcul nécessitait à peu prés 80 passes, ce qui est très couteux en ressources.

6.3 Un algorithme alternatif :

En cherchant du coté des algorithmes de simulation, nous avons trouvé un algorithme utilisé sur le processeur principal et qui ne nécessite pas d'intégration. [18]
Par contre il demande l'utilisation de l'advection inverse et de l'advection avant.
Nous avons donc tenté d'implémenter une version GPU (sur la carte graphique) de l'advection avant.

Depuis l'arrivée des shaders 3.0, il est possible de faire appel à une texture depuis le vertex shader.
En utilisant cette texture, on peux modifier l'endroit où nous allons positionner le vertice courant.
Si nous utilisons un rendu de type « point », nous pouvons avoir un pixel pour chaque vertex, et donc nous pouvons, dans le vertex shader, choisir l'endroit où nous allons écrire le pixel actuel. Cela rend donc bien possible l'advection avant.





Afin de rendre le résultat interpolé dans 4 pixels adjacents, il est même possible d'utiliser un rendu avec des points de tailles 2x2 pixels, avec une transparence pour l'interpolation, afin d'accumuler le résultat dans la texture finale.
L'algorithme fonctionne, mais un soucis interviens lorsque l'on augmente la résolution de la texture, avec par exemple un résolution de 512x512.
En effet, s'il est très rapide de calculer 512x512 pixels, il est beaucoup plus lent de calculer 512x512 sommets. Il faut effectivement stocker et calculer 512x512 sommets avec leur position mais aussi coordonnées de textures et normales, car il n'est pas évident de ne stoker que la position avec l'architecture d'un moteur 3D et de l'API graphique.
Bref, stocker, déplacer et afficher 512x512 sommets est très lourd, de plus comme nous écrivons 4 pixels pour chacun, au final il s'agit de calculer 512x512x4 pixels.

Avec directX 10, et l'ajout d'un geometry shader, il est peut être possible d'obtenir des temps de calculs raisonnables, puisqu'il n'est alors plus obligé de stocker les sommets, qui sont calculé par le geometry shader, et seul la position sera calculé, ni normale ni coordonnées de textures. Néanmoins cela nécessiterait plus de tests pour s'en assurer.

Au final, dans le cadre actuel, cette technique est viable pour de petites résolutions, mais explose de manière exponentielle avec de plus grandes résolutions.

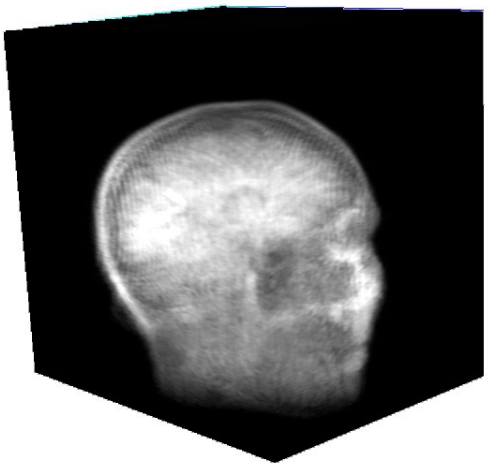
6.4 Les fluides en 3D :

Il est intéressant de noter que les techniques présentées ici marchent exactement de la même manière quelque soit le nombre de dimensions simulé. Les calculs s'appliquent juste à plus de voisins, au nombre de 4 en deux dimensions, nous passons à 6 en trois dimensions. La vélocité est stocké sous la forme d'un vecteur en deux ou trois dimensions, mais les algorithmes sont fondamentalement les mêmes.

Un des soucis, lorsqu'on utilise des fluides en trois dimensions, tiens dans la manière d'afficher la grille de valeur ainsi calculé.
S'il est très facile d'afficher une grille en deux dimensions, qui est tout simplement une texture, une grille en trois dimensions nécessite des techniques plus évolués.

La plupart des ces techniques consistent dans le calcul de « slices » c'est a dire de coupes de cette grille qui vont être affichés les unes devant les autres. Le nombre de coupes calculés va donner une qualité plus ou moins grande. La solution idéale est difficile à trouver, de nombreux artefacts graphiques désagréables apparaissant selon la manière d'afficher et de calculer ces slices. [19]

Ces plans de coupes (slices) peuvent être positionné de plusieurs manières et en nombres variables.
Dans certains algorithmes, ils sont alignés avec la camera, ce qui mène à des soucis lors des rotations de la caméra et l'on vois souvent des sortes de bandes sur les slices.
Les plans de coupes peuvent être tout simplement fixes, ici pas d'effet de bandes. Par contre, il faut alors croiser des slices dans chacun des trois axes, afin de couvrir tous les points de vues. Mais si l'on regarde depuis un axe intermédiaire, il est difficile de mélanger joliment les coupes des différents axes.



Rendu d'une texture 3D

Une solution à été testé, faisant appel uniquement au pixel shader, sans calculer de géométries intermédiaires. L'idée est qu'à partir d'un rendu stockant la position des faces avant, et d'un autre stockant celle des faces arrières, nous pouvons effectué une sorte de lancé de rayon dans la texture 3D en accumulant la valeur de chaque case rencontré.
L'algorithme fonctionne bien, mais le nombre d'appel à la texture peux devenir très important, et le dynamic branshing, c'est à dire pourvoir effectuer moins de calcul pour les pixels demandant très peu d'appels, est difficile à

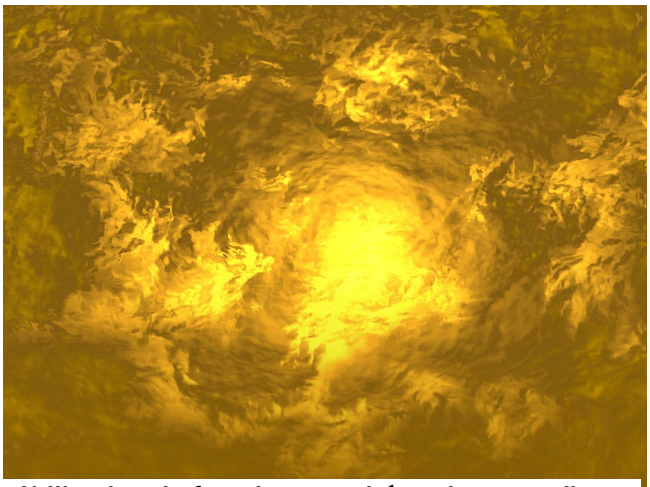
utiliser efficacement. Par contre le résultat est assez propre.

Encore une fois, l'utilisation de geometry shader, introduits part direct X 10 permet de créer les géométries intermédiaires (slices) très efficacement.

6.5 Les techniques non physiques :

Il existe bien évidement de nombreuses techniques utilisé traditionnellement dans le jeu vidéo afin de reproduire plus ou moins fidèlement les effets de fluides.
La plupart utilisent des vidéos de fluides ou d'explosions qui vont être affiché sur des « sprites » (ou bill-board) c'est a dire des plans qui font toujours face à la caméra. Si l'effet nécessite beaucoup de sprites, les développeurs utilisent généralement un système de particules pour les animer simplement. Le mouvement général sera donc donné par le système de particules et les petits détails (volutes, écoulements ...) qui nécessitent beaucoup de calcul seront issues de vidéos.
Certains jeux utilisent une technique mixte, qui consiste à calculer un effet de fluide avec des équations physiques en petite résolution, et utiliser ensuite l'image ainsi calculé sur des dizaines de particules à la fois. La répétition n'est pas forcement perceptible, notamment si les particules varient en taille et en orientation.

L'effet de vagues qui est présent à la surface des grandes étendues d'eau peut être simulé par des fonctions mathématiques simple, avec par exemple une simple accumulation de différentes sinusoïdes en décalage.



Utilisation de fonctions mathématiques et d'une texture de bruit pour un effet de nuage

La différence fondamentale entre les techniques physiques et celles-ci tiens dans le déterminisme des techniques non-physiques.
En effet, le processus d'advection lié à la simulation physique est itératif, c'est à dire que chaque résultat dépend du précédent, et donc de tous les précédents. Au contraire, la plupart des techniques utilisés habituellement dans les jeux sont déterministes, c'est à dire que l'on est capable de calculer directement le résultat qu'aura l'effet à l'image N, le résultat sera toujours le même. Bien sur, les développeurs utilisent également des fonctions aléatoires, mais ce n'est pas essentiel pour l'effet, cela sert simplement pour masquer le coté monotone de la technique utilisé.

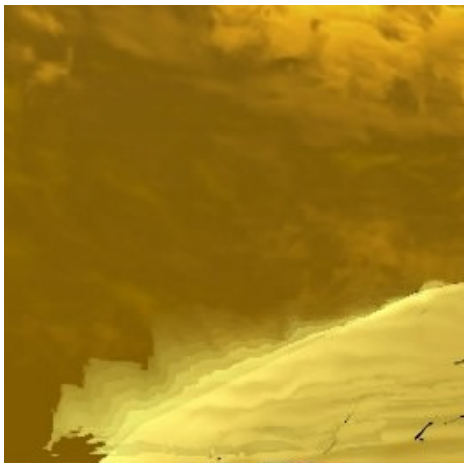
C'est aussi la raison qui impose ces techniques comme choix habituel, car les développeurs et les artistes peuvent contrôler très précisément le résultat, en jouant sur quelques paramètres, en modifiant les vidéos sources. On peux d'ailleurs obtenir des effets totalement différents, ce qui rend la technique utilisable de manière homogène pour la plupart des effets d'un jeu (explosions, rayons de lumières, fumées, chutes d'eau ....). Au contraire, avec une simulation physique, nous pouvons simplement essayer de respecter au maximum les valeurs réelles pour obtenir un effet réaliste mais sur lequel on à peu de contrôle et ou le moindre changement de paramètres peux modifier totalement le résultat, et ce de manière imprévue.

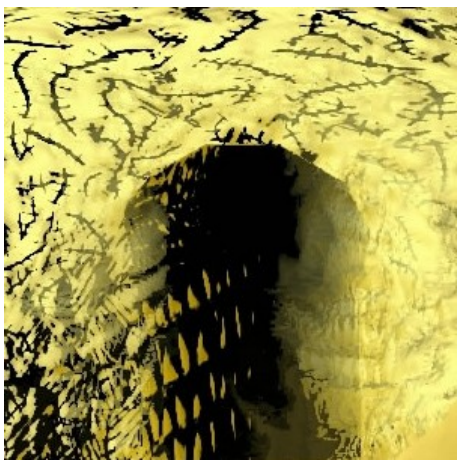
Après l'éventail de techniques qui prennent pour base la physique réelle et cherchent à résoudre partiellement les équations de la mécanique des fluides, et celui des techniques qui cherchent simplement à reproduire l'effet visuel de manière purement déterministe, nous introduisons ici une technique utilisant une partie physique et une autre partie purement déterministe.
L'idée est d'utiliser la technique de l'advection, qui permet de transporter de l'encre par exemple, mais au lieu de simuler le mouvement d'un liquide transport, nous allons utiliser directement une texture adéquate, qui va nous donner la direction dans laquelle va bouger l'encre.
Pour faire varier cette direction, nous utilisons simplement une série de sinusoïdes basé sur le temps et sur l'emplacement (par rapport à l'écran) du pixel à calculer.
L'effet fonctionne très bien sur les premiers tests, avec un personnage isolé et une texture simple.

Bien sur, le mouvement du fluide ne suis pas de vrai logique, mais l'association d'un mouvement cyclique fixé et d'un transport itératif de l'encre va créer à la fois des vagues de fluides, et de petites volutes très esthétiques.

C'est cette technique qui sera utilisé dans l'application finale, peu couteuse, elle est très contrôlable.

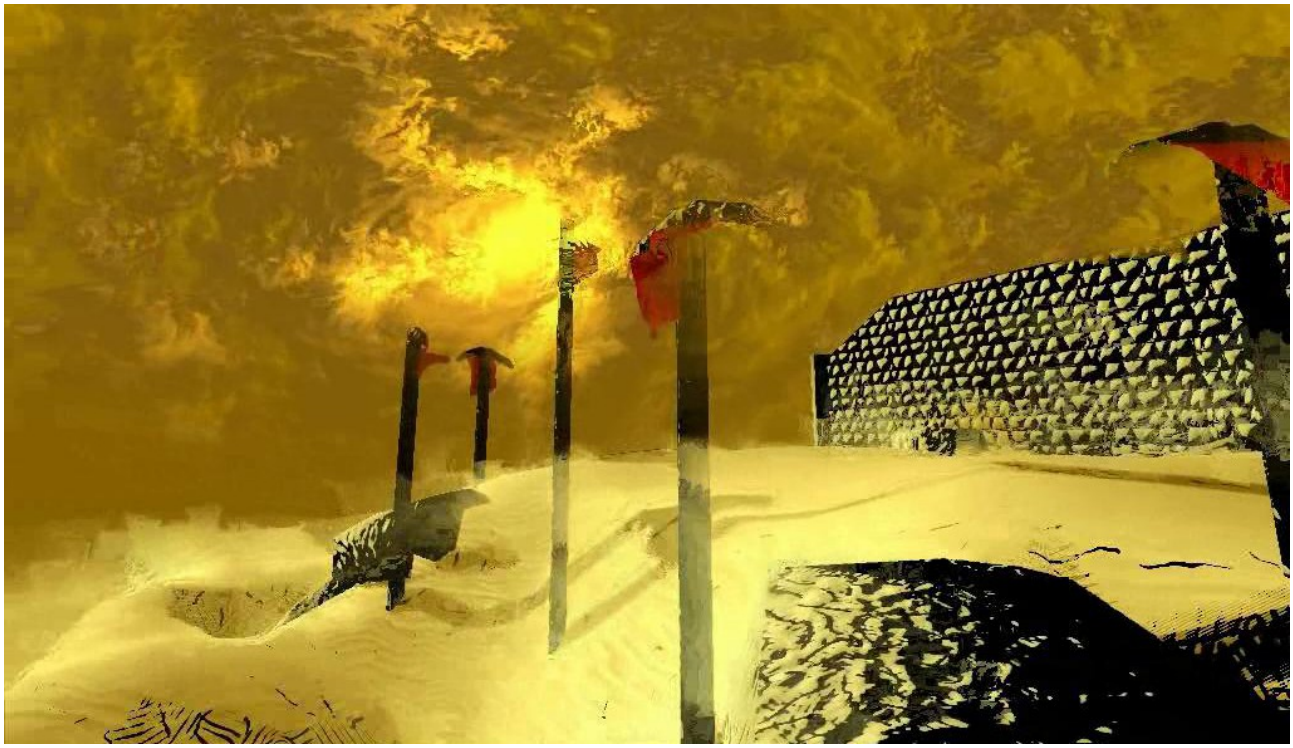
Dans l'application, le fluide est positionné sur une grille 2D placé devant l'écran.
L'encre y est déposé soit par des objets 3D spécifiques, soit par la scène 3D elle même. Ce sont soit les zones sombres (les traits noirs essentiellement), soit les parties les plus éclairées qui déposerons de l'encre.
Il est important que l'encre soit déposé de manière ponctuelle, à travers de petites surfaces, et non par blocs importants, car ces blocs bougeraient d'un seul tenant, l'encre ne pouvant se disperser autour, alors que c'est surtout cette dispersion qui donne des effets esthétiques.





Il est possible d'influencer le dépôt et le transport de l'encre au travers d'objets 3D.
L'idée est de disposer des objets 3D dans la scène, qui ne seront pas rendu, mais qui indiqueront au fluide 2D placé sur l'écran, le sens dans lequel le fluide doit se déplacer. Ces objets fixeront également la quantité d'encre déposé par le décor.
Cela permet alors de créer des zones avec beaucoup de fluides, d'autres avec moins, et de jouer sur l'orientation général du mouvement du fluide.

L'inconvénient majeur de la technique utilisé dans le projet, qui utilise un fluide en deux dimensions posé sur l'écran, se remarque dés que la caméra se déplace. En effet, comme le fluide est en 2D, si la caméra tourne, l'encre déjà déposé sur l'écran reste lui figé, et persiste quelques instants avant de disparaître.
Une solution partielle est de simuler le fluide, non pas dans un texture 2D mais dans un cubemap, c'est à dire un assemblage de 6 textures 2D qui forment un cube autour de la caméra. Lorsque l'on tourne la caméra, le fluide reste fixe.



Cela fonctionne idéalement, le seul effet indésirable arrive lorsque l'on avance ou recule, ou lors de déplacements latéraux. Néanmoins, les trainées sont moins perceptibles. Par contre, pour avoir une bonne résolution pour le fluide, il faut une cubemap de 1024x1024x6 pixels, ce qui est un peu trop lourd pour le projet actuel. Une solution intermédiaire est de simuler le fluide dans une texture légèrement plus grande que l'écran, et de calculer la différence lors des rotations de la caméra afin que les pixels du fluides suivent. Il est même possible d'envisager une opération d'agrandissement et de rétrécissement du fluide afin de compenser les avancées et reculs de la caméra.

Les portions de codes HLSL concernant cette technique ainsi que la technique de calcul des nuages sont disponibles en annexe 3 et 4.

7. Simulation de tissus en temps réels

La simulation de vêtement fait appel a beaucoup de calculs et peux devenir parfois incontrôlable. Pour ces raisons, elle était jusqu'à présent réservé à des applications pré-calculés, ou à des utilisations très simples.

Outre la quantité de calcul nécessaire, le problème posé par la simulation de tissu est le transfert des informations entre la carte graphique et le processeur. En effet, dans l'utilisation traditionnelle, le processeur calcule les positions des vertices du tissu et doit les envoyer à la carte graphique à chaque image. Ce transfert est très couteux en bande passante, et doit être limité au minimum. La plupart du temps, les objets 3D ne sont envoyé qu'une seule fois à la carte graphique, qui les stocke et les réutilise tels quel à chaque image. L' arrivé de nouvelles cartes graphiques à changé la donne, grâce aux nouvelles capacités de programmation du processeur de la carte graphique. On peux ainsi faire exécuter tous les calculs de la simulation de tissu directement dans la carte graphique et ainsi éviter tout transfert.
La fonctionnalité fondamentale qui à permit au cloth (simulation de vêtement) temps réel de voir le jour est la possibilité de faire appel à une texture dans le vertex shader, c'est à dire d'utiliser une texture pour modifier la position des sommets d'un objet. Cette possibilité n'est apparue qu'avec les shaders 3.0, auparavant nous ne pouvions faire appel à des textures que dans le pixel shader. La seconde innovation récente permettant ce calcul, est l'intégration de différents formats de textures, notamment le format flottant. Ainsi, nous pouvons avoir une texture de 32 bits float par composant, ce qui permet de stocker réellement un vecteur 3D quelconque. Le format traditionnel RGB 8 bits entier par composant ne permet que 256 valeur par direction ce qui rend impossible le stockage de positions. Le format flottant permet des étendues de valeurs énormes.

7.1 Principe général de fonctionnement :

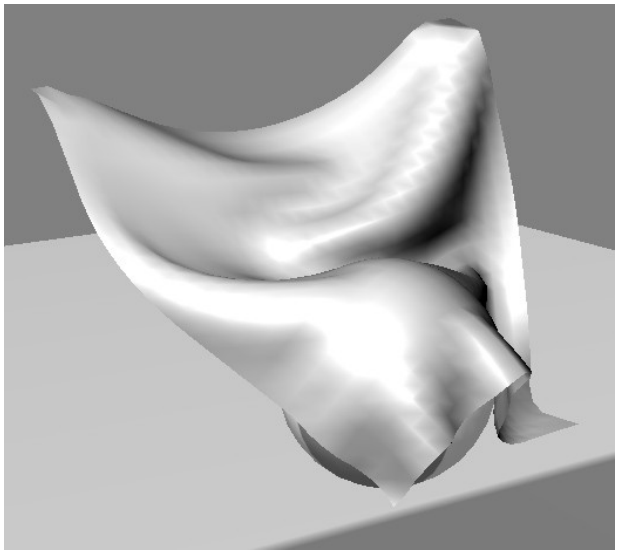
L'idée principale, déjà dévoilé partiellement dans le précédent paragraphe, est de stocker les positions des sommets dans une texture. On peux ensuite utiliser cette texture lors d'un rendu de la carte graphique, faire diverses opérations sur ces positions (gravité, collisions ...) et stocker le résultat dans une nouvelle texture. Lors du rendu de l'objet du tissu, la texture calculé sert a déplacer les sommets à l'endroit requis. Nous avons donc besoin de deux shader, l'un (dit shader de simulation) qui servira à calculer pour chaque sommet, sa prochaine position, et un autre (dit shader de rendu) qui a partir de la position des sommets, assurera le rendu de l'objet tissu.

Calcul principal de la simulation : l'intégration de Verlet [22]
Nous utilisons un algorithme itératif, c'est a dire qu'il se base sur la position actuelle pour calculer la position suivante. La première étape est donc de calculer la position suivante d'un vecteur en fonction de sa position et de sa vitesse précédente, ainsi que des divers forces qui peuvent influencer. Pour le moment nous ne tenons compte que de la gravité, mais nous allons facilement ajouter du vent, qui va utiliser la normale du sommet et la direction du vent pour calculer la force de chaque sommet. Pour éviter d'avoir à stocker la vitesse des sommets explicitement, nous allons utiliser les deux dernières positions d'un point afin d'en déduire sa vitesse. Nous aurons donc besoin de 3 textures de positions en tout, qui vont s'échanger à chaque frame. L'une est la position n-1, l'autre la position n, et la dernière la position n+1 (la texture dans laquelle nous écrivons)
La formule de l'intégration de Verlet est la suivante :

$$P(n+1) = (P(n) - P(n-1)) * dampling + dt^2 * Accélération;$$

L'accélération est la somme des forces, c'est a dire dans notre cas un simple vecteur représentant la gravité. dt est le temps passé entre deux calculs. Le dampling est une variable indiquant l'inertie du tissu, elle est donc lié au poids du tissu ainsi qu'a la densité de l'air. C'est en général une valeur empirique qui donne de bons résultats visuels. Cette formule à l'avantage d'être facile à calculer et donne un résultat qui à peu de chance de devenir complètement incohérent. Par contre, ce n'est pas une formule scientifiquement rigoureuse. [24] [25]

7.2 Les premières collisions :



Nous avons maintenant des sommets qui tombent, conformément à la gravité. Pour qu'ils ne tombent pas éternellement, nous allons ajouter des collisions, tout d'abord avec le sol. La collision avec un plan horizontal infinie est très simple à définir, il suffit de bloquer la coordonnée y des particules afin qu'elle ne descende pas en dessous d'une certaine valeur. Par contre, le tissu va avoir l'air de glisser sur le sol, on peux alors ajouter des frottements, sous la forme d'un ralentissement de sa vitesse s'il y a collision. Les collisions avec une sphère sont également très simple. On teste la distance entre le sommet et le centre de la sphère, si elle est inférieure à la taille de la sphère, on va calculer un vecteur qui va repousser ce sommet à l'extérieur de la sphère. On peux la encore ajouter des frottements. Nous utilisons également une texture de « blocage » qui va designer le degré d'influence du mouvement de l'objet sur le tissu. Cela permet essentiellement d'accrocher certains points afin qu'ils suivent le mouvement d'un personnage par exemple. Les parties les plus importantes du code HLSL de mise a jour des sommets sont disponible en annexe 5.

7.3 Garder la forme :

Pour le moment, nous n'avons aucune influence d'un sommet sur l'autre, et le tissu ne garde absolument pas sa forme de départ. Il faut pour cela ajouter des forces qui maintiennent ensemble chaque sommet avec ses voisins, comme s'ils étaient accroché par des cordes. La manière la plus simple est de relier chaque sommet à ses 4 voisins les plus proches par une sorte de ressort. On va donner une distance de repos, et si la distance entre le sommet et son voisin est inférieur à cette distance, il va s'éloigner de celui-ci, si elle est supérieur, il va s'en rapprocher. Il faut bien comprendre que ces forces vont être continuellement en désaccord, le sommet étant tiré entre ses quatre voisins. La force du ressort ne doit donc pas être trop importante afin que l'ensemble des forces puissent s'équilibrer. Si l'on veux un tissu plus solide, on peux soit augmenter cette valeur de force, soit faire plusieurs passes de ce calcul, ce qui est plus lourd mais va mieux équilibrer ces forces. Avoir simplement 4 ressorts ne va pas vraiment suffire pour garder une forme. Pour cela nous rajoutons les 4 voisins en diagonale. La forme est maintenant mieux conservé, mais nous observons de nombreux problèmes sur les plis, le tissu ayant tendance à s'aplatir et former des plis très dur. La solution est d'étendre ces ressorts aux voisins plus distants. Avoir les voisins de nos voisins va suffire à obtenir des plis harmonieux.

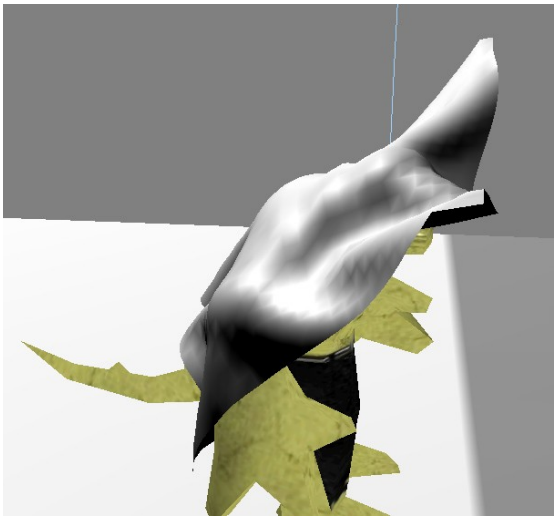
7.4 Le rendu du tissu :

L'inconvénient de déplacer les sommets à chaque image est que nous n'avons plus aucunes informations de normales. Nous allons donc devoir les calculer dans la carte graphique au lieu de les recevoir du processeur.
Plusieurs implémentations de cet algorithme calculent la normale dans le shader de simulation et la stockent ensuite dans une texture, comme pour la position. Au départ, le calcul de la normale dans notre application était effectué dans la passe de rendu, juste au moment d'afficher l'objet final.
D'un point de vue vitesse, cette technique à des avantages, puisque l'on ne va rendre le volume qu'une fois par image, alors que la simulation pourra être faite plusieurs fois, afin

d'avoir une meilleure précision. Par contre, le calcul de la force du vent va utiliser cette normale, il faut donc l'avoir a disposition dans le shader de simulation, et donc le stocker.
Une fois la normale et la position calculé, nous pouvons rendre le tissu et notamment l'influence de la lumière sur le tissu.

7.5 Vers des collisions plus avancées :

Avec des collisions plans et sphères (éventuellement sphères non orthonormées), les interactions entre le tissu et son environnement restent limitées.
Les collisions boites ou cylindres sont également facile à implémenter. Le soucis est de passer les informations relatives à ces objets. Passer directement la position d'un objet à un shader est très facile, mais le faire pour 1000 objets est impossible. On peut par contre passer ces informations dans une texture à une dimension. Nous aurons une texture par type d'objet de collision, et celle-ci contiendra par exemple la matrice de transformation d'un de ces objets par pixel.
L'interaction avec un personnage à été testé en positionnant des sphères déformées sur les os animés du personnage. Cela fonctionne, même si le passage des coordonnées des sphères est couteux.



Pour aller plus loin, et obtenir des collisions avec par exemple un sol vallonné, où un personnage, cela ne suffit pas vraiment. Plusieurs techniques peuvent être utilisée. La plus complexe, utilisée dans une démo de NVIDIA, fait appel à une texture 3D qui va contenir pour chaque pixel, s'il y a ou non une collision.

Le calcul de cette texture fait appel à un rendu par division en hauteur de la texture 3D, ce qui rend l'algorithme très lourd. Une technique plus simple et qui a été testé lors de ce projet, est celle d'un rendu de profondeur d'un personnage vu de dos.
Ce type de collision devrais être idéal pour la simulation d'une cape par exemple, qui va principalement entré en collision avec le dos du personnage.

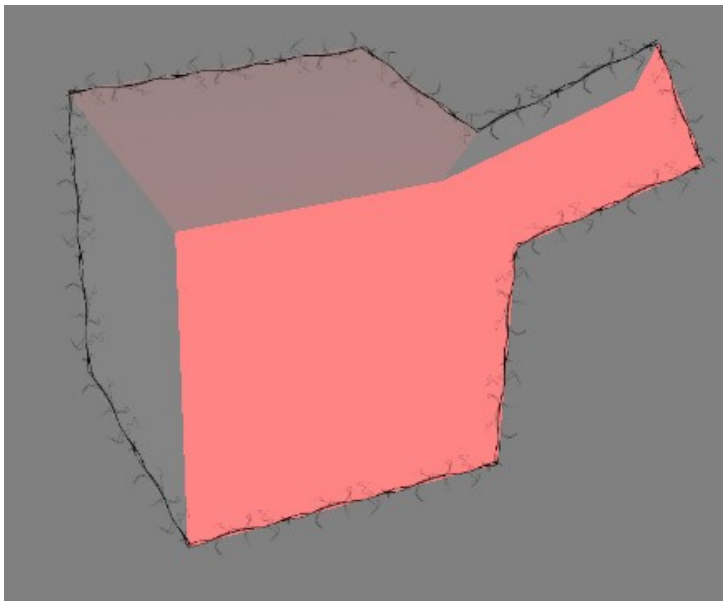
On place donc une camera orthographique derrière le personnage et nous pouvons comparer la distance entre chaque sommet du tissu et celle contenu dans la texture de distance. Si elle est inférieur, il y a collision. Pour que la cape puisse aller sur le devant du personnage, nous pouvons aussi utiliser une camera vu de devant, ce qui va donc nous donner la collision maximum du personnage.
Le même système peux être utilisé pour des collisions avec un terrain, en utilisant directement la texture de hauteur.

8. Effets divers

8.1 Les effets de contours :

Lorsque l'on parle de rendu non réaliste, on pense souvent au cell-shading, cette technique assez utilisé dans les jeux vidéo et qui consiste à tracer une ligne autour des objets, qui sont autrement composés uniquement d'aplats assez simple. L'effet donne un aspect dessin animé traditionnel.
La plupart du temps, la ligne entourant un objet est droite, elle suis en général le contour des polygones extérieurs de l'objet. Il existe des algorithmes qui permettent d'arrondir ce contour, par des filtres sur l'image finale (flou + contraste).
Une idée intéressante est de pouvoir modifier comme on le souhaite la forme de ce contour. [26]
Au lieu d'avoir une simple ligne, nous pourrions avoir des hachures, une ligne qui serpente doucement, ou bien tout autre image adéquate.

Pour obtenir cet effet, il va s'agir de construire une géométrie d'une certaine épaisseur qui va suivre les contours de l'objet dans l'espace de l'écran.
La première étape est de trouver les arrêtes de l'objet qui forment les bords. La technique la plus simple est de prendre toutes les arrêtes dont une face est dirigé vers la caméra et l'autre dans la direction opposée à la caméra.
Un premier soucis se pose déjà dans la manière dont sont représenté les maillages dans un moteur de rendu habituel. C'est simplement une liste de sommets (avec position, normale ...) et une liste de faces (chacune contient le numéro de 3 sommets). Il n'y a donc aucune notion d 'arrête. Si l'on prend simplement les sommets de chaque face deux par deux, nous aurons deux fois les mêmes arrêtes (une arrête fait appel à deux faces).
Il faut donc trier les arrêtes pour éliminer les doublons, il faut également garder une trace des faces liés afin d'avoir la normale de la face. La aussi, un soucis de rapidité se pose, puisque les faces n'ont pas vraiment de normale habituellement, c'est les sommets qui en ont une. La normale d'une face se calcule donc en fonction de la position des 3 sommets qui la compose (c'est en fait le produit vectoriel de deux arrêtes). Le calcul peux être long s'il doit être fait plusieurs fois pour chaque face (logiquement trois fois puisque nous avons trois arrêtes par face).
Il va donc falloir stocker un maximum d'éléments pour optimiser les calculs.
Si l'objet n'est pas sensé se déformer, c'est a dire que les sommets restent à la même place les uns par rapport aux autres (ce qui n'est pas vrai pour un personnage en mouvement par exemple), nous pouvons effectuer une grande partie des calculs une seule fois.
Le stockage se fera dans une liste d' arrêtes, référençant les deux sommets et les deux faces, plus la normale calculé de ces deux faces.



Ensuite, à chaque image, il va falloir sélectionner les arrêtes de bords. Avec notre structure, c'est assez simple, il faut transformer dans le repère de la caméra la normale de chacune des deux faces de chaque arrête. Si le signe de la coordonnée Z de ces deux normales est différent, l'arrête est un bord.
A partir de cette liste d' arrêtes, nous construisons des rectangles alignés sur la vue, afin de positionner une texture à l'endroit de l'arrête. La texture pourra soit être étiré sur toute la longueur de l'arrête, soit être répété.
On peux trier les arrêtes par la distance avec la caméra, afin d'afficher en dernier les plus proches. Par contre, il faudrait que ces textures ajoutés sur les arrêtes, qui ne sont pas de vrais objets géré par le moteur de rendu, prennent en compte les autres objets et surtout soient masqué par eux.
Pour géré cela, le plus simple est d'utiliser un rendu de profondeur vue par la caméra, qui va pour chaque pixel stocker la distance avec la caméra. Ensuite, une arrête ne sera affiché que si le ou les pixels correspondants dans la texture de profondeur ne sont pas plus proche que l'arrête.

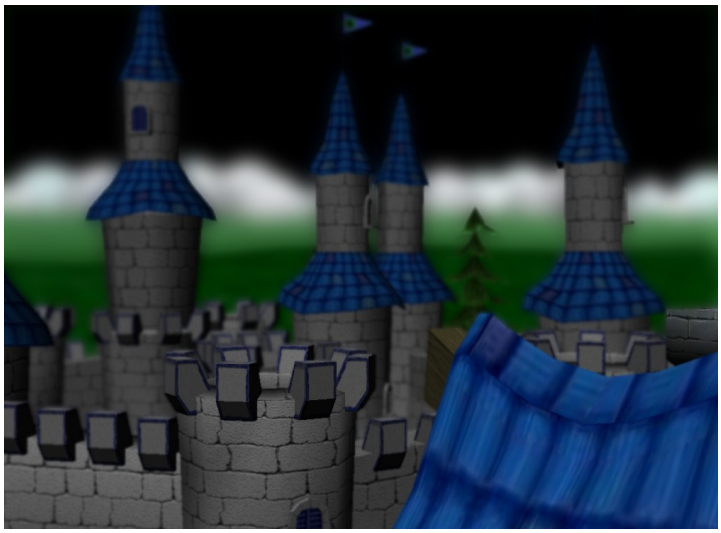
L'effet fonctionne bien, en tout cas avec des textures simples de traits, même s'il est assez lourd, notamment s'il est appliqué à un personnage animé.
Le dernier soucis rencontré est le raccord entre les arrêtes. Pour le moment les textures ajoutés sont toutes indépendantes, ce qui fonctionne avec des textures simples, mais avec des images plus complexes, le raccord entre unr arrête et la suivante est très dérangeant visuellement.
Si la texture est étiré, il n'y a pas vraiment de soucis, il suffit d'avoir une texture qui se raccorde bien aux bords, mais le rendu n'est pas très jolie.
Pour gérer une texture qui se répète, il faut réunir deux éléments.
Le maillage en lui même doit se raccorder d'une arrête à la suivante, et les coordonnées de texture doivent se suivre, afin qu'une image que l'on à commencé à afficher sur une arrête se continue sur l'autre.
Pour cela, il faut trier les arrêtes pour quelle soient affiché dans le bon ordre. Pour chaque arrête sélectionné pour être affiché (arrête de bord), nous allons regarder si un des deux sommets est le même que l'un d'une autre arrête. Si c'est le cas, il faut changer l'ordre d'affichage pour qu'elles soient affiché l'une après l'autre. Le processus est assez lent à calculer, même si l'on peux l'optimiser en utilisant l'information de distance dans l'espace écran afin de ne tester que les arrêtes dont les sommets sont proches.
Pour connecter les maillages d'une arrête à la suivante, l'idéal est de rajouter un coude qui permet une rotation douce de la texture vers la direction de la prochaine arrête.

Au final, même avec toutes ces optimisations, l'algorithme se révèle très lent et l'effet peu esthétique, même si cela dépend beaucoup de la texture utilisé. Cette technique ne sera donc pas utilisé dans le projet final.

8.2 Profondeur de champ :

La profondeur de champ vise à reproduire la focalisation de l'œil humain, qui module grâce au cristallin la distance à laquelle les objets seront nets. Ainsi, les objets plus proche ou plus éloigné de la caméra qu'une certaine valeur seront floutés.
Il existe plusieurs techniques pour cela, qui tiennent dans la manière de faire un flou tout d'abord, et dans la manière de moduler ce flou selon la distance.
Toutes ces techniques sont réalisés sur la carte graphique. S'il est possible de le faire sur le processeur principal, la carte graphique sera bien plus efficace, car elle à déjà les textures à flouter sous la main, et son architecture massivement parallèle est ici pleinement exploitée.
Pour obtenir un flou de haute qualité, on peux utiliser un flou en deux étapes. La première étape va effectuer un flou horizontal uniquement, par exemple sur 8 pixels consécutifs, puis la seconde étapes applique sur ce rendu un flou vertical uniquement, de 8 pixels également. Le résultat est alors équivalent au flou qui serait calculé avec 8x8 soit 64 échantillons de la texture.
Pour obtenir des flous très prononcés, une technique très efficace est d'utiliser le mipmapping automatique calculé par la carte graphique. [27]
Pour éviter certains effets visuel désagréables de clignotements quand une texture est vue de loin, la carte graphique à besoin de stocker des versions réduite de la texture, elle utilisera alors la plus adapté.

Nous utilisons donc une texture de résolution plus faible que la normale, et la carte graphique interpolera automatiquement entre les pixels. Nous avons donc un flou pratiquement gratuit.



Une autre idée, valable pour de très gros flou, est d'accumuler des flous de différentes tailles.
Le second point sur lequel il faut travailler est la manière dont on module le flou avec la distance. Il va falloir d'abord avoir une valeur comprise entre 0 et 1 indiquant le degré de flou à obtenir. On l'obtient par exemple en calculant la distance avec un point (qui sera le point focal où tout sera net). [28]
La manière la plus simple de moduler ensuite l'image finale, est de simplement afficher soit l'image nette, soit la version flouté, selon la valeur de flou. Si cela fonctionne, on note de nombreux artefacts dérangeant. Ainsi les objets lointains peuvent déborder sur les objets proche, ce qui est complètement absurde dans le cadre d'une profondeur de champ. Les valeurs de flou intermédiaires sont également assez mal géré.
Une manière plus évolué est d'utiliser directement les valeurs de flou dans l'algorithme de flou en lui même et l'on peux alors moduler la distance des échantillons selon la valeur de flou. Le résultat est bien meilleur visuellement, même si l'algorithme est légèrement plus lourd, on ne peux pas simplement translater les coordonnées de texture, se qui permettait d'utiliser cette technique sans aucun shader, avec d'anciennes cartes graphiques par exemple. Avec les nouvelles cartes, les deux techniques sont pratiquement équivalente en terme de rapidité.
Pour régler le problème des débordements des objets lointains, la solution est de demander, dans l'algorithme de flou, la distance de chaque échantillon. Si cette distance est supérieure à celle du pixel à calculer, l'échantillon n'est pas pris en compte dans le calcul. Cela fonctionne bien, même si la différence n'est visible que pour des flous importants. Par contre l'algorithme est sensiblement plus lourd, il faut faire, pour 8 échantillons par exemple, 8 appels supplémentaires à la texture de profondeur.

8.3 Flou de mouvement :

Une fois que nous avons un moyen d'obtenir du flou, il est très tentant de vouloir reproduire la rémanence de l'œil, qui va rendre flou les objets qui se déplacent vite, le flou se présentant dans le sens du mouvement. C'est un effet qui est aussi très perceptible dans la photographie, ou la vitesse des éléments est souvent représenté par leur degré de flou. L'effet compte beaucoup pour rendre les images crédibles et également plus compréhensibles.
La manière physiquement correcte serait tout simplement de garder une certaine partie des images précédente. Cette idée triviale ne fonctionne pas en pratique, car si l'œil reçoit des milliers de photons par seconde et va mélanger tout cela, l'ordinateur calcule en général moins d'une centaine d'image par seconde. Si l'on utilise les rendus précédents nous aurons une dizaine de fois le même objet qui s'étire, et non un seul flou continue.
En résumé, pour obtenir un beau flou il va falloir le calculer nous même.
Le flou est directionnel, c'est a dire qu'il s'applique uniquement dans le sens du mouvement. Il faut donc connaître le sens du mouvement, mais aussi le stocker.
Pour cela, le plus simple est d'utiliser une texture, qui stockera dans ses valeurs de rouge et de vert, un vecteur donnant

la direction et l'intensité du mouvement pour chaque pixel de l'image finale.
Une première passe va donc calculer cette texture, appelé ici texture de vélocité, et une seconde utilisera cette texture pour flouter l'image finale.
Parlons d'abord de la seconde étape, qui est en fait la plus simple, puisque nous avons déjà parlé du flou pour la profondeur de champ. La différence est qu'ici il n'y a pas besoin d'une passe horizontale puis une passe verticale, puisque le flou n'est que dans une direction. Le shader, au lieu de prendre les pixels contigu horizontalement, va regarder la texture et prendre les pixels contigu dans la direction indiqué par la texture. Pour obtenir un effet vraiment propre, il faudra, pour chaque échantillon, regarder la nouvelle direction du mouvement à cet endroit, et l'utiliser pour trouver le prochain échantillon. C'est un exemple typique de l'interdépendance très couteuse entre différents appels aux textures, qui est traité dans le chapitre 9.

La première étape de l'algorithme va donc devoir calculer la vélocité de chaque pixel de l'image. [29]
Pour cela il faut simplement afficher chaque objet avec un shader qui va utiliser, pour chaque sommet, la distance précédente pour connaître la direction du mouvement (dans le repère de l'écran). Connaître la position précédente du sommet pose problème. C'est une information qui n'est pas directement accessible. Si l'objet est indéformable, nous pouvons tout simplement récupérer la matrice de transformation utilisé à l'image précédent, l'appliquer sur les sommets de l'objet (aux mêmes positions puisque l'objet est indéformable). Par contre, pour un objet déformable, il faut connaître réellement la position de chaque sommet à l'image précédente. Pour cela, nous pouvons utiliser les extra data, c'est à dire des données passé à la carte graphique de la même manière que les coordonnées de textures. Nous utilisons concrètement un script VSL qui va copier à chaque image la position des sommets de l'image précédente. Le soucis ici est que ce script est exécuté sur le processeur principal, ce qui oblige donc à récupérer les positions des sommets (et donc empêche tout hardware skinning, c'est a dire d'appliquer les transformations du squelette sur la carte graphique), de les traiter, et de les renvoyer à la carter graphique (ce qui est assez lourd), tout cela pour un calcul qui n'est utilisé que sur la carte graphique. Avec les shaders 3.0, il ne semble pas possible de faire autrement (sauf peu être de manière complexe en utilisant le render to vertex buffer).

Malgré cela, nous avons tout de même notre vecteur de direction dans le shader. Mais si l'on se contente d'afficher l'objet normalement avec la direction en sortie, nous n'aurons de flou qu'aux endroits où se trouve l'objet, sans que ce flou puisse déborder. C'est très gênant, notamment pour des objets fins, qui conservent des bords très dur, le flou n'étant appliqué qu'a l'intérieur.
Il faut donc que l'objet affichant la vélocité déborde de l'objet initial. En fait, il suffit d'utiliser la direction du mouvement que nous avons déjà pour modifier les coordonnées des sommets. En utilisant la normale du sommet, nous pouvons étirer le sommet dans la direction du mouvement, ou au contraire dans la direction inverse. Le tout fonctionne très bien, même si c'est le type d'effet qui doit être utilisé avec subtilité, s'il est remarquable c'est qu'il est trop fort. Il doit simplement améliorer l'image, pas la transformer.

Si ces deux techniques, essentiellement réalistes, ne sont pas directement utilisés dans le projet, elles ont permis de comprendre et d'intégrer différentes manières de penser, en ce qui concerne le rendu en plusieurs passes et l'utilisation d'une texture de vélocité, réutilisé pour le shader de fluide.

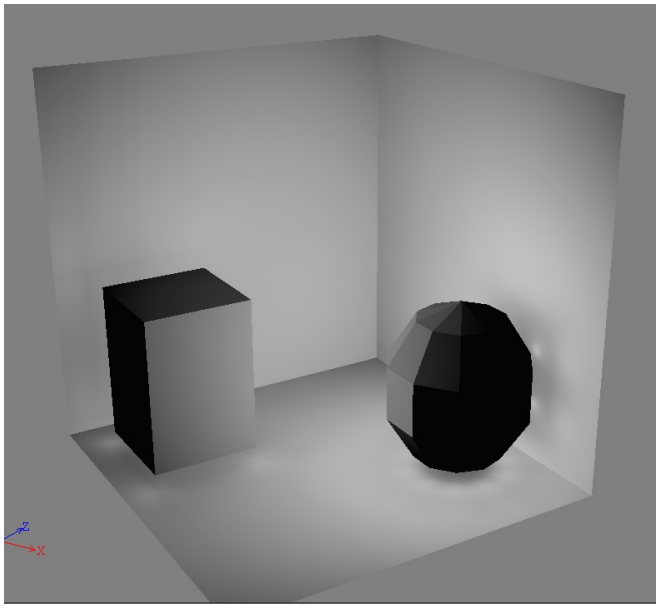
8.4 Ambient occlusion :

La technique de l'ambient occlusion est très connue dans le domaine du film d'animation. C'est une technique visant à simuler une sorte d'illumination globale, c'est à dire venant de toutes les directions, en gérant les rebonds et occlusions des autres objets, et ce de manière simple et peu couteuse. Néanmoins, c'est une technique encore très rarement utilisée dans les jeux, même si certains commencent à l'inclure, mais toujours en option pour les machines les plus puissantes, souvent réservé aux cartes graphiques directX10 (par exemple dans la version PC de Burnout Paradise et la version directX10 d'Age Of Conan).
Le principe est en fait d'estimer le pourcentage d'occlusion de chaque point par les autres points de la scène. L'algorithme de base utilise les sommets, et va pour chacun calculer la contribution de chacun des autres sommet à son occlusion. En général, le calcul pour connaître l'occlusion est de simuler deux disques, l'un sur le sommet à calculer (récepteur) et l'autre sur celui dont on doit calculer la contribution à l'occlusion (émetteur). Ces deux disques sont positionnés sur leur sommet, alignés selon la normale de ce sommet et d'une taille qui varie selon l'aire des faces utilisant ce sommet. [35] L'occlusion est alors calculé ainsi :

$$occlusion = A * cos(aR) * cos(aE) / (Pi * r^2 + A)$$

A : Aire du disque émetteur
aR : angle entre la normale du disque récepteur et le vecteur reliant les deux disques
aE : angle entre la normale du disque émetteur et le vecteur reliant les deux disques
r : distance entre les deux disques

Certains calculs peuvent être fait une fois pour toute. C'est le cas du calcul de la moyenne de l'aire des faces partageant un sommet. Pour des objets indéformable, c'est évident, mais même pour des objets déformables, la plupart du temps cette aire ne variera pas beaucoup , les mouvements étant souvent des rotations plus que de réels changement de l'aire des faces. Supposer cette aire fixe semble en tout cas une optimisation acceptable.



Le but ici est d'implémenter cet algorithme entièrement sur la carte graphique, en cherchant à profiter au maximum des optimisations que celle ci offre par rapport à un processeur.
La difficulté qui apparais immédiatement tiens dans le fait que dans un vertex shader, nous n'avons pas accès directement aux autres sommets, mais seulement à celui en cours de traitement. Contrairement au pixel shader, ou nous pouvons utiliser des textures pour accéder à des informations calculés sur d'autres pixels, il n'existe pas vraiment de structure de sauvegarde des données sur les vertex auquel le shader puisse accéder. Il va donc falloir utiliser également des textures. Cette technique n'est possible que depuis les shaders 3.0 qui permettent d'accéder à une

texture depuis le vertex shader. Si celle ci contient des informations relatives à d'autres sommets, nous pourrons donc y accéder. Le soucis est donc de pouvoir écrire ces informations.
La solution trouvé est d'utiliser le rendu en point, d'une manière un peu similaire à celle utilisé dans le chapitre 7 sur les fluides. Nous rendons donc l'objet en mode point, chaque sommet va donc écrire un seul et même point. Avec le vertex shader, nous allons nous arranger pour positionner ces points successivement dans une texture à une dimension, dans l'ordre des sommets.

Dans une première étape, nous allons ainsi stocker dans différentes textures la position, la normale et l'aire correspondant à chaque sommet.

La bonne qualité du résultat va dépendre du nombre de sommet, qui doit être assez important.
Dans l'optique de l'appliquer au temps réel, il semble assez handicapant de se limiter aux sommets.
Il existe maintenant différentes manière de calculer le résultat final.
Nous pouvons le calculer dans l'espace écran, c'est à dire que pour chaque pixel de l'objet final, nous allons boucler sur tous les sommets (c'est à dire sur tous les pixels des textures représentant ces sommets) et additionner ainsi les occlusions de chaque autre sommet.

Une optimisation possible est de rendre encore une fois l'objet en mode point.
Dans le vertex shader, il est possible de fixer la taille de ce point. Cette taille va dépendre en fait de l'aire du vertex émetteur. En effet, après une certaine distance (qui va dépendre en fait de l'aire) la contribution d'un vertex deviens très faible et peu être ignorée. Donc au final, pour chaque sommet, nous affichons un carré qui va calculer, pour tous ces points, la contribution du sommet courant au pixel concerné. Les pixels au-dela d'une certaine distance ne seront donc même pas calculé.
Cette manière de rendre l'ambient occlusion va être très précise, puisque calculé pour chaque pixel de l'image finale. Par contre les effets obtenus lors des tests sont assez moyens, avec beaucoup d'artefacts étranges, des zones trop ou pas assez éclairés. C'est effets sont due à la forme de cercle utilisé pour simuler les calculs d'occlusion.

Une autre idée est de calculer tout cela dans l'espace des coordonnées UV. Nous calculons en fait une texture de lightmap. L'avantage, outre que l'occlusion peut n'être rendu que de temps en temps par exemple, est que la résolution peux être plus faible, la texture se chargeant d'interpoler entre les pixels calculés. Pour calculer dans l'espace des coordonnées UV, il faut bien sur des coordonnée de texture qui ne dépassent pas le domaine 0-1 et qui ne se recouvrent pas. Il suffit alors d'afficher l'objet, et au lieu de la position du sommet transformé dans le repère de l'écran, nous utilisons la coordonnée de texture du sommet.
Par contre, cette technique ne permet pas d'utiliser l'optimisation décrite précédemment, avec le rendu en point. De plus, il faut alors calculer une texture pour chaque objet, et notamment calculer de nombreux pixels qui ne sont pas du tout affiché actuellement. C'est l'intérêt de l'autre technique, qui ne calcule que les pixels réellement affichés.
Au niveau des résultats techniques, l'algorithme reste très lourd à utiliser, et même sur des objets simples, le nombre d'image par seconde tombe très vite sous les 20 images secondes nécessaires pour que ce soit fluide.
De nombreuses idées pourraient être exploitées si l'effet tournait en temps réel. Par exemple, on peux imaginer utiliser la couleur des sommets afin d'avoir une véritable notion de couleur, chaque surface colorerait alors un peu les surfaces alentour.

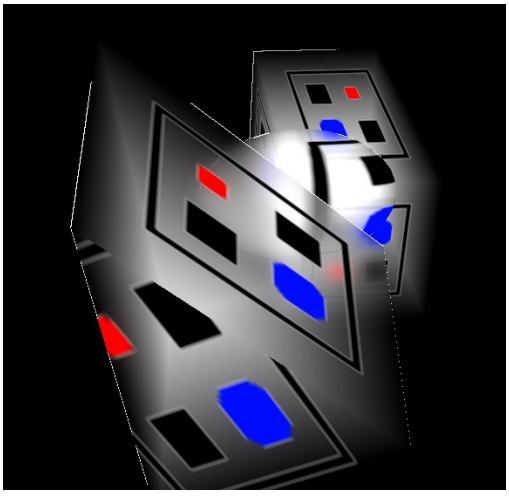
Dans les jeux vidéo actuels, une technique semble se développer. Elle ne travaille plus du tout sur les sommets, mais directement sur les pixels de l'image finale. Il s'agit en fait d'échantillonner pour chaque pixel autour des pixels voisins, et de regarder la position et la normale. En fonction de cela, il est possible de déterminer des surfaces d'occlusion, et donc un éclairage. Souvent l'algorithme utilise également un second rendu affichant les faces dans le sens opposées, et permettant donc d'avoir une bonne partie des faces cachées. Cet effet peut en fait se comparer aux techniques qui cherchent à augmenter le contraste en assombrissant ou éclaircissant les bords des objets dont le fond est très éloigné. Cela permet de faire se détacher les objets les uns des autres. [36]

La technique de l'ambient occlusion en général semble faite pour obtenir un éclairage hyper réaliste, et donc en contradiction avec la recherche esthétique du présent mémoire, mais cette technique est pourtant très souvent utilisé dans le domaine du dessin animé en 3D, car elle donne des dégradés très subtiles, des ombres très douces, ce qui se marie très bien avec un rendu en aplat, er donne des couleurs unies variant uniquement en luminosité. Le jeu vidéo kingdom heart montre particulièrement bien l'effet recherché, même si la tout est précalculé.

8.5 Subsurface Scattering :

La transparence est un élément encore difficile à prendre en compte dans les applications temps réelles. En général, la transparence est générées en affichant les faces de la plus lointaine à la plus proche, chaque nouvelle face modulant sa couleur en fonction de la couleur déjà présente.
Le problème avec le trie des faces, c'est tout d'abord qu'il est assez couteux, mais surtout qu'il ne gère pas correctement les intersections entre différentes faces, ce qui est normalement géré par le zbuffer, permettant de garder le pixel le plus proche de la caméra. Avec la transparence, le pixel final est fonction de plusieurs faces à des distances différentes, impossible à gérer avec un zbuffer.
La solution pratique est souvent, en plus du trie des faces, d'afficher en premier lieu les objets transparent avec une transparence binaire, c'est a dire soit un pixel est totalement transparent, soit totalement opaque. Le z buffer peux donc servir ici. Ensuite, une seconde passe rajoute les objets avec de la vrai transparence, mais en gardant le zbuffer calculé précédemment.
Mais tout cela ne traite que de la transparence brute, mais ne permet absolument pas de simuler des transparences plus subtiles telle que celle de la peau, ou bien à travers des verres déformants.
Certains exemples de nvidia, le constructeur de cartes graphiques, montrent qu'il est possible d'obtenir un effet de peau intéressant en stockant l'éclairage d'un maillage dans une texture, en la floutant, et en la réappliquant. La lumière va donc se diffuser légèrement au dela des limites du maillage. [38]
La manière dont la lumière traverse les surfaces comme la peau, ou bien la cire d'une bougie est communément appelé : subsurface scattering, c'est à dire diffusion de lumière sous la surface visible. En effet, la lumière va se diffuser progressivement dans le volume, et selon son épaisseur et sa densité, va devenir de plus en plus opaque.

Un élément important de l'effet est le coté flou de la lumière qui traverse une telle surface. L'intensité de la lumière va également dépendre de la densité et de l'épaisseur de l'objet à chaque endroit.
Pour connaître la distance entre le point d'entré d'un objet et le point de sortie par rapport au pixel calculé actuellement, une méthode simple est de tout d'abord rendre les faces arrières des objets (les faces qui sont tournée vers l'arrière) en stockant la position, puis les faces avant. La différence de position des deux donnera l 'épaisseur. Bien sur, cette méthode ne marche réellement qu'avec les objets convexes, sans creux qui pourrais décomposer le trajet d'un rayon à travers l'objet en plusieurs parties séparées. Mais dans la plupart des cas, l'épaisseur que nous souhaitons gérer est suffisamment faible pour que seul le premier point de sortie soit requis.
Cette technique permet déjà d'avoir des objets plus ou moins transparents selon l'épaisseur.
L'ajout de flou est par contre plus problématique.
Une idée qui fonctionne est d'afficher les objets du plus loin au plus proche. A chaque fois que l'on cherche à afficher un objet, on va d'abord utiliser le rendu déjà calculé pour cela. Il va donc falloir flouter l'image à chaque fois qu'un objet est transparent. Cela fonctionne bien, mais est très lourd si de nombreux objets sont transparents. Il est possible bien évidement de regrouper des objets présents à des distances à peu près égales de la caméra mais qui ne se recouvrent pas les uns les autres, et de faire un seul rendu pour tous.



Pour éviter de devoir floutter l'image à chaque nouvel objet, il est possible d'accumuler dans une autre texture une valeur de flou, puis d'appliquer un flou sur l'image finale par pixel en fonction de cette valeur.
L'effet n'étant pas directement indispensable pour le projet, il n'a pas été poussé plus loin.

9. Les optimisations des shaders programmables

Lorsqu'on s'intéresse à la programmation de la carte graphique, il faut se demander pourquoi les calculs effectués sur la carte graphique seraient plus rapides que s'ils étaient calculé par le processeur principal.

Tout d'abord, la carte graphique possède des circuits câblés qui permettent de calculer très rapidement certaines opérations (multiplications de matrices par exemple). Ainsi, au lieu de devoir convertir ce calcul de multiplication en dizaines d'opérations intermédiaires, qui pourront être interprété par le processeur principal, le calcul est effectué en une seule passe et le résultat sort à la vitesse du circuit. Néanmoins, ce type d'opérations câblé tend à disparaître, ou tout au moins à être moins importante car si elles étaient essentielles avec les premières versions des shaders, qui ne faisaient en fait que câbler quelques opérations entre elles, les nouvelles cartes graphiques supportent des algorithmes bien plus évolué et il deviens difficile d'utiliser efficacement les fonctions pré-câblés.

Ainsi, la version 4 des shaders supprime beaucoup de fonctions standard du langage qui n'étaient plus supportées mais émulées (c'est a dire simulées, en décomposant en plusieurs opérations) par les cartes graphiques modernes. De même les fonctions deviennent universelles et peuvent être utilisées uniformément dans les pixels et vertex shaders, le tout fonctionnant de manière beaucoup plus malléable qu'avec la structure très figé des anciennes versions.
Un élément assez évident d'optimisation est d'effectuer le plus possible de calculs sur les sommets au lieu des pixels. Ainsi il est souvent possible de calculer certaines transformations au niveau du sommet, puis passer le résultat aux pixels (ce résultat sera interpolé entre les sommets) au lieu de faire ce calcul pour chaque pixel.
Un second élément de rapidité pour les cartes graphiques, et la disponibilité de la mémoire vidéo. Dans l'idéal, toutes les textures et les objets sont chargé au préalable dans cette mémoire ultra rapide, et le processeur envoie simplement la position ou serons affiché ces objets.

Bien sur, il faut alors bien gérer cette mémoire afin qu'elle ne soit pas saturée, ce qui engendrerait des swapp, c'est a dire des transferts assez long entre la mémoire principale et la mémoire vidéo, voir même depuis le disque dur. Le but est de stocker les textures les plus utilisées afin d'avoir le moins de transferts possibles.
La rapidité d'un algorithme GPU va également dépendre de son adéquation avec la gestion de la mémoire cache.
Lorsque l'on demande la valeur d'un pixel d'une texture par exemple, la carte graphique va en fait demander toute la région autour du pixel et va la copier dans un cache. Ainsi, si un autre appel à la texture est fait et que la position du pixel demandé est proche du premier, les données serons déjà dans la mémoire cache, et l'appel très rapide. Par contre, si les appels aux pixels sont totalement aléatoires, l'utilisation d'un cache est alors inutile, voir même contre productive, puisque la copie inutile de pixels dans le cache à tout de même un cout.

9.1 Une architecture massivement parallèle :

Le point principal qui permet à une carte graphique d'obtenir de grandes vitesses, c'est que de nombreux pixels vont utiliser les mêmes calculs. C'est l'élément important qui va permettre d'optimiser les calculs.
Les cartes graphiques modernes possèdent de nombreux registres mémoires et les processeurs sont séparé afin d'obtenir l'équivalent de 256 processeurs différents. La manière de les utiliser est donc très différente de celle d'un unique processeur.

L'architecture est appelé SIMD (Single Instruction, Multiple Data), c'est a dire que la carte graphique va préparer un programme, c'est a dire une suite d'opérations, et va ensuite l'appliquer sans délais ni synchronisation sur plusieurs « data », c'est a dire souvent des pixels ou des sommets.
Pour les sommets, ils sont en général tous calculé ensemble, par objets et par shader.
La manière dont sont regroupé les pixels est par contre beaucoup plus mystérieuse. Les constructeurs souhaitent garder ces informations en internes car le programmeur n'est pas sensé s'en occuper, puisque au niveau du code source, il n'y a pas de limitations et que ces détails d'implémentations varient d'une carte graphique à l'autre. Mais en terme

d'optimisation, il est essentiel de connaître un minimum les contraintes habituelles.
Ainsi, sur la carte graphique Nvidia Geforce 8800, les pixels sont apparemment regroupés par blocs de 32x32.
L'idéal est donc que ces pixels soient calculés avec le même shader, en parallèle, chaque processeur s'occupant d'un pixel inutilisé, profitant de mémoires embarquées pour ses calculs intermédiaires.
Quand tous les processeurs ont terminés, ils passerons en bloc à un autre shader, ou portion de l'écran.
Dans la réalité, il est bien sur rare que tous ces pixels utilisent le même shader. Mais il n'y aura peu être que 4 ou 5 shaders différents sur cette portion de 32x32.

Les cartes graphiques évoluant, il est maintenant possible d'utiliser une technique appelé « dynamic branching ».
Un shader va alors pouvoir appliquer un calcul différent sur les pixels qui l'utilise.
Auparavant, si un pixel avaient besoin d'une texture, tous les pixels utilisant ce shader effectuaient cet appel à la texture, même si le résultat n'étaient pas utilisé.
Avec le dynamic branshing, ce n'est plus obligatoire.
Par contre, si chaque pixel nécessite des calculs différents, il n'y a alors plus d'optimisation lié au calcul en parallèle, il faut donc trouver un équilibre entre ces deux éléments la.
L'élément essentiel du dynamic brashing est la fonction « clip » qui va tester une variable, et si celle ci est négative, va stopper complètement le calcul du pixel courant (pas d'écriture dans le buffer de sortie, écran ou texture). Si tous les processeurs travaillant sur la même zone avec le même programme font de même, l'ensemble du bloc va immédiatement aller travailler sur le bloc suivant, mais si un seul des pixels concerné continue son calcul, les autres vont devoir patienter.
Comme l'utilisation du clip ajoute un calcul, il faut donc s'assurer que des pixels adjacents vont profiter de ce clip pour stopper ensemble les calculs. Sinon, il vaux mieux laisser tous les pixels faire les calculs jusqu'au bout même si ceux ci sont inutiles (un résultat transparent par exemple).

9.2 Optimisations :

Il est parfois difficile de comprendre ce qui va faire ralentir un programme de carte graphique.
Par exemple, les appels aux textures vont prendre un certain temps (il faut calculer les coordonnées finales du pixel dans la texture, demander l'information à la mémoire, souvent interpoler avec les pixels adjacents). Pendant ce temps, les processeurs vont faire tous les calculs qu'ils peuvent sur le reste du programme, ces calculs sont donc « gratuits » puisqu'ils profitent du temps d'exécution de l'appel à la texture. Tous les appels de textures sont lancé dés le début du programme, même si dans celui-ci ils n'interviennent que plus tard, afin d'obtenir les résultats le plus vite possible.
Par contre, si le résultat de cette appel de texture est utilisé, par exemple, pour calculer les coordonnées d'un autre appel à une texture, il n'est alors plus possible de faire ces calculs en parallèle. Il faut attendre le résultat du premier appel pour pouvoir lancer le second.
Le temps est alors doublé, même si le programme à l'air d'avoir à peine changé.

Ex :
float2 *appel1 =* ***tex2D****(texture1, coord1);*
float2 *appel2 =* ***tex2D****(texture2, coord2);*

float2 *appel1 =* ***tex2D****(texture1, coord1);*
float2 *appel2 =* ***tex2D****(texture2, appel1);*

Ainsi, par exemple dans les algorithmes de displace dans le pixel shader, il existe différentes techniques, certaines utilisant une série d'appels indépendants à la texture de hauteur, et d'autres ou chaque appel utilisera les résultats des appels précédents dans ses calculs. La première sera donc bien plus rapide, même si le nombre d'appel est le même.
Pour plus d'informations sur les algorithmes de displace, voir [13].

Il faut également se méfier des optimisations cachées de la carte graphique, qui peuvent tromper lors des tests.

Par exemple, le calcul suivant sera effectué normalement :

```
float4 appel1 = tex2D( texture1, coord1) *0.00000001;
```

Par contre le calcul :

```
float4 appel1 = tex2D( texture1, coord1) *0.00000000;
```

sera en fait remplacé par :

```
float4 appel1 = 0.0;
```

C'est a dire complètement éliminé, et donc ne jouera plus sur les performances, ce qui peux tromper facilement.

Certains calculs vont être plus rapide que d'autres, par exemple, les conditions sont assez lentes à calculer.
Ainsi :

```
test = min ( test, 1);
```

sera probablement plus rapide que :

```
if(test<1) test = 1;
```

et pourtant strictement équivalent en terme de résultats.

Pourquoi le mot « probablement » ? Car cela peux changer selon la carte graphique et selon le compilateur qui optimisera ou non automatiquement ce type d'appels.

Les boucles sont également un sujet épineux, notamment les boucles avec un nombre d'itérations variables.
Souvent, le compilateur va automatiquement vouloir « dérouler » la boucle, ce qui est plus rapide pour de petites boucles que de vouloir faire les tests et sauts inhérent à la boucle.
Ainsi, il transformera :

```
for(int i=0; i<3; ++i)
{
        resultat += i*3;
}
```

en :

```
resultat += 0; // probablement même éliminé automatiquement
resultat += 3;
resultat += 6;
```

Si le nombre d'itérations est variable, il tentera parfois d'estimer le nombre maximum et les calculera tous afin d'avoir le même programme pour tous les pixels.

Il faut donc souvent optimiser soi même les calculs et organiser son code efficacement pour éviter les boucles inutiles.

Conclusion

Le travail sur ce projet à été principalement un travail de recherche qui apporte à la fois de nouvelles connaissances et un réel plaisir de la découverte de nouveaux effets. Du point de vue de la réalisation concrète, le résultat est suffisamment abouti pour présenter de manière adéquate une bonne partie des techniques développées. Bien sur, dans le cadre de la création d'un vrai jeu vidéo, l'intégration d'un vrai gameplay, avec un vrai but, aurait été idéal. Cependant, c'est la recherche graphique qui a été privilégié, sans que le temps suffise pour implémenter concrètement des algorithmes d'animations comportementales ou d'utilisation de la manette Wii pour se diriger et se battre, ce qui était prévu au départ.
Au final, ce projet, et surtout le mémoire associé, à apporté une base solide de connaissances dans le domaine, dans un but personnel mais aussi professionnel.

Glossaire

Classe : Objet en programmation qui comporte à la fois des données internes (variables membres) et des fonctions propres (fonctions membres). L'on peu créer une instance de cette classe, puis changer les valeurs de ses variables, appeler ses fonctions .... C'est un élément de base de la programmation orienté objet.

Couleur ambiante ou émissive : Couleur minimum d'un objet, c'est a dire la couleur qu'il aura en l'absence de lumière.

Couleur diffuse : Couleur de base d'un objet en pleine lumière. En général, elle ne pourra que s'assombrir en fonction de la position de la lumière par rapport a la normale de la face. Cette couleur ne dépend pas de la position de l'observateur, mais uniquement de l'objet et de la lumiére.

Couleur spéculaire : couleur des reflets de la lumière sur un objet. En général s'additionne à la couleur diffuse. Elle est souvent accompagnée d'un facteur désignant l'ouverture maximum du reflet. Cette couleur dépend de la position de l'observateur, de celle de l'objet et de la lumière.

CPU : Central Processing Unit, c'est à dire processeur principal de la machine. Il s'occupe principalement, dans un jeu vidéo, de l' Intelligence Artificielle, de la gestion du moteur physique (collisions), des calculs pour les animations des personnages ...

GPU : Graphic Processing Unit, c'est à dire processeur graphique. Situé sur la carte graphique, une carte qui est relativement optionnelle, elle ne s'occupe traditionnellement que des calculs visant à afficher des éléments évolués à l'écran.

HLSL : Langage de programmation pour la carte graphique. C'est en langage de haut niveau avec une syntaxe similaire au C. Il est par contre dédié à l'utilisation de DirectX. Il est opposé au GLSL pour la programmation OpenGL. Les deux peuvent être géré de manière commune par l'utilisation du langage CG.

Lumière ambiante : lumière émettant de partout et dans toute les directions, c'est donc une constante fixe précisant le niveau minimum de lumière dans la scène.

Lumière omnidirectionnelle : lumière émettant à partir d'un point uniformément dans toutes les directions, également appelée lumière point (point light).

Lumière « spot » : lumière émettant à partir d'un point mais uniquement dans une direction, comme un spot. La forme de la lumière peut alors être soit un cône (projetant une lumière ronde) ou une pyramide (projetant une lumière carrée)

Lumière directionnelle : lumière émettant de partout dans une seule direction. Les rayons sont donc parallèles. C'est typiquement la lumière provenant du soleil

Normale : Vecteur de direction normée désignant l'orientation d'un sommet ou d'une face. On l'utilise notamment dans le calcul de l'intensité lumineuse d'un point.

Pixel : Information de couleur d'un point. En général la couleur est divisé en 4 composantes : rouge, vert, bleu et alpha (transparence). Le nombre de bits dédié a chaque composant peux varier (8 bits, 32 bits ...) , ainsi que la manière de l'interpréter (entier ou float).

Pixel shader : Partie d'un programme de shader dédié aux calculs sur les pixels d'un objet. Il est exécuté pour chaque pixel. Il récupère les informations données par le vertex shader (en général interpolé selon les 3 sommets du triangle contenant le pixel) et renvoi en sortie une couleur (rgba).

Shader : Partie programmable de la carte graphique, dédié au calcul de rendu d'un objet. Les langages utilisées sont : assembleur, GLSL, HLSL, cg ...

Template : Modèle de conception. L'idée est en fait de pouvoir écrire des classes génériques, dont les types de certaines variables ne sont pas mentionnée explicitement. A la compilation, le compilateur va écrire lui même une classe pour chaque type différent qui sera utilisé.

Texture : Matrice de pixels. Le nombre de dimensions d'une texture est en général de 2, mais peux également être de 1 ou de 3 (texture volumétrique) voir plus.

Vertex shader : Partie d'un programme de shader dédié aux calculs sur les sommets d'un objet. Il est exécuté pour chaque sommet. Il prend divers paramètres envoyé par le CPU et renvoi, au minimum, la position finale de chaque sommet. Il envoi généralement aussi des informations de normales et de coordonnées de textures.

Vertex, vertice : Sommet. En général sommet d'une face (souvent triangle). Il posséde au minimum une position, mais très souvent contient également une normale, des coordonnées de textures, une couleur.

Virgule flottante : opposée aux nombres réels qui ont toujours une valeur très précise, le nombre en virgule flottante sépare le stockage du chiffre en deux partie, l'une stocke un nombre à virgule inférieur à 1 (la mantisse), et la seconde une puissance de dix (l'exposant). L'un multiplié par l'autre donnera notre nombre. La grande particularité de ce type de codage est d'être aussi précis sur de très grand nombre que sur de très petits, proportionnellement.

Références

Les références citées ici ne sont bien évidement pas les seules du domaine, et chacune cite de nombreux autres documents qu'il ne faut pas hésiter à consulter pour aller plus loin.

3.

[1] Simplification et abstraction de dessins au trait
P. Barla, J. Thollot, F. Sillion - ARTIS / GRAVIR-IMAG-INRIA
Méthodes de génération de dessins au trait à partir d'images réelles.

[2] Survol de techniques de rendu non-photoréaliste (NPR)
Victor Ostromoukhov - Université de Montréal

[3] Real-Time Pencil Rendering
Hyunjun, Lee Sungtae, Kwon Seungyong Lee - POSTECH
Technique de rendu de hachures très efficace est esthétique.

[4] Conception et mise en oeuvre d'une plateforme de rendus non-photoréalistes
Romain Vergne, INRIA Futurs
Différentes techniques de rendus non réaliste, dont l'une très proche de notre rendu au trait

[5] Real-time hatching - SIGGRAPH 2001
Emil Praun, Hugues Hoppe, Matthew Webb, Adam Finkelstein
Document fondateur du rendu au trait, introduisant notamment une texture de hachure (Tonal Art Map) devenu classique, décomposé en 6 niveaux de luminosité et 4 niveaux de mipmapping (taille de l'image). Voir figure

[6] Rendering Methods for Augmented Reality
Jan T. Fischer - Eberhard-Karls-Universität, Tübingen
Présentations de diverses méthodes de rendu, la plupart non réalistes.

[7] Non-Invasive, Interactive, Stylized Rendering
Alex Mohr, Michael Gleicher - University of Wisconsin, Madison
Document très original, présentant diverses méthodes de rendus non réalistes, et également les méthodes pour les utiliser directement dans n'importe quel jeu vidéo existant, en détournant les appels à la librairie OpenGL.

4.

[8] Omnidirectional Shadow Mapping, GPU Gems - Chapitre 12
Philipp S. Gerasimov - iXBT.com

[9] Percentage-Closer Soft Shadows
Randima Fernando - NVIDIA Corporation

[10] Real-Time Shadowing Techniques - Siggraph 2004
Marc Stamminger - University of Erlangen-Nuremberg
Cours complet sur le rendu des ombres, principale via des techniques liés au shadow mapping.

[11] Deferred Shading in S.T.A.L.K.E.R., GPU Gems 2 – Chapitre 9
Oles Shishkovtsov, GSC Game World

5.

[12]Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail
Bent Dalgaard, Larsen Niels, Jørgen Christensen - Technical University of Denmark
Traite notamment de la topologie adéquate pour traiter un paysage avec un niveau de détail variable (LOD).

[13]Displacement Mapping on the GPU - State of the Art
László Szirmay-Kalos, Tamás Umenhoffer - Department of Control Engineering and Information Technology, Budapest University of Technology, Hungary
Document indispensable, qui présente toutes les techniques temps réelles pour simuler le relief sur une surface plane.

6.

[14]Dynamic Particle Coupling for GPU-based Fluid Simulation
Andreas Kolb, Nicolas Cuntz - Computer Graphics Group - University of Siegen, Germany
Simulation de fluides basé sur des particules, et non d'une grille.

[15]Building a Million Particle System
Lutz Latta - Massive Development GmbH
Utilisation du GPU pour calculer et afficher un nombre important de particules.

[16]Real-Time Simulation and Rendering of 3D Fluids, GPU Gems 3 - Chapitre 30
Keenan Crane, Ignacio Llamas, Sarah Tariq - University of Illinois at Urbana-Champaign - NVIDIA Corporation
Présentation de la méthode de simulation de fluide utilisé dans le fameux exemple NVIDIA montrant une gargouille battant des ailes dans un flux de fumée. Certaines méthodes utilisé dans notre projet viennent directement de la, bien que appliqué en deux dimensions.

[17]Fast Fluid Dynamics Simulation on the GPU, GPU Gems - Chapitre 38
Mark J. Harris - University of North Carolina at Chapel Hill
Simulation de fluides en deux dimensions.

[18]Practical Fluid Dynamics
Mick West - gamasutra.com
Document de référence pour notre seconde méthode d'implémentation des fluides. L'article est très clair, même si le système est basé sur le CPU.

[19]Volume Rendering Techniques, GPU Gems – Chapitre 39
Milan Ikits, Joe Kniss, Aaron Lefohn, Charles Hansen - University of Utah and University of California, Davis
Techniques de rendu de volumes, principalement à partir de textures 3D, donc idéales pour le rendu de fluides 3D.

[20]Simulation of Cloud Dynamics on Graphics Hardware
Mark J. Harris, William V. Baxter, Thorsten Scheuermann, Anselmo Lastra - Department of Computer Science, University of North Carolina at Chapel Hill, North Carolina, USA
Utilisation de fluides dans la simulation de formation et déplacement de nuages.

[21]Simulation de l'écoulement d'un fluide
Humbert Florent (2008)
Description très scientifique de l'implémentation d'un solveur de fluides sur le CPU.

7.

[22]A Simple Time-Corrected Verlet Integration Method
Jonathan "lonesock" Dummer - GameDev.net
Méthode et équation d'intégration de mouvements adapté au temps réel et au framerate inconstant.

[23]Stable but Responsive Cloth
Kwang-Jin Choi Hyeong-Seok Ko - Graphics and Media Lab - Seoul National University

[24]Cloth Simulation on the GPU - SIGGRAPH 2005
Cyril Zeller - NVIDIA Corporation

[25]Stupid OpenGL Shader Tricks – Game Developers Conference
Simon Green - NVIDIA
Traite notamment de la simulation de tissu, c'est le document de base utilisé dans notre implémentation

8.

[26]WYSIWYG NPR: Drawing Strokes Directly on 3D Models
Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Phillip L. Davidson Matthew Webb, John F. Hughes Adam Finkelstein - Princeton University - Brown University
Document très intéressant présentant une applications cumulant plusieurs principes afin de permettre aux utilisateurs de peindre sur un objet 3D (des bords, des hachures, des dessin), le style de peinture étant conservé lors des rotations de la caméra. La gestion des formes de traits est extraordinaire. On trouve également des recherches sur le grain du papier et son interaction avec le dépôt de pigments par les traits.

[27]Practical Implementation of High Dynamic Range Rendering - Game Developers Conference
Masaki Kawase - BUNKASHA GAMES
Présentation des méthodes de gestions de rendu HDR. Nous y trouvons également des algorithmes de calcul de flou.

[28]Depth of Field: A Survey of Techniques, GPU Gems - Chapitre 23
Joe Demers - NVIDIA

[29]Stupid OpenGL Shader Tricks – Game Developers Conference
Simon Green - NVIDIA
Traite notamment du motion blur, c'est le document de base de notre implémentation de cet effet.

[30]Advanced Image Processing
Robert Strzodka
Présentations de différents filtres dans l'espace de l'image sur le GPU : diffusions, distorsions, détections de contours.

[31]Hardware Accelerated Ambient Occlusion Techniques on GPUs
Perumaal Shanmugam, Okan Arikan - University of Texas at Austin

[32]GPUGI: Global Illumination Effects on the GPU
László Szirmay-Kalos, László Szécsi, Mateu Sbert - TU Budapest – U of Girona
Série de présentations sur de nombreuses méthodes d'illumination globale : radiositée, final gather ... La plupart de ces méthodes sont encore assez statiques, pré-calculant encore une grande partie des données afin de pouvoir être temps réel.

[33]Ambient Occlusion, GPU Gems - chapitre 17
Matt Pharr, Simon Green - NVIDIA

[34]Dynamic Ambient Occlusion and Indirect Lighting, GPU Gems 2 - chapitre 14
Michael Bunnell - NVIDIA Corporation

[35]L'ambient occlusion
Laurent Gomila - Developpez.com
Une présentation claire d'une technique de calcul de l'ambient occlusion sur le CPU.

[36]Image Enhancement by Unsharp Masking the Depth Buffer
Thomas Luft∗ Carsten Colditz∗ Oliver Deussen∗
University of Konstanz, Germany
Technique utilisant le depth buffer pour contraster les bords des objets.

[37]Caustics Mapping: An Image-space Technique for Real-time Caustics
Musawir Shah, Sumanta Pattanaik
Méthode de génération de caustics sur le GPU. La gestion de la lumière et de son transport n'a pas été très approfondi à travers ce projet, mais c'est un élément très intéressant pour des recherches futures.

[38]Real-Time Approximations to Subsurface Scattering, GPU Gems - Chapitre 16
Simon Green - NVIDIA

Divers :

[39]GPGPU: Beyond Graphics
Mark Harris - NVIDIA
Présentation des méthodes utilisant la carte graphique pour effectuer des calculs non graphiques. On y parle de pratiquement tout les sujets traités ici : fluides, cloths, calculs de luminosité ...

[40]ShaderX2 Introductions and Tutorials with DirectX 9
Livre accessible, introduisant de nombreux concepts sur les nouveaux concepts liés à l' arrivé des shaders 3.0f.

[41]ShaderX2 Shader Programming Tips & Tricks With DirectX 9
De nombreux effets y sont décrits : cloth, displacement, rendu non réaliste ...

[42]Shader X3 Advanced Rendering with DirectX and OpenGL
Techniques avancées : deferred shading, Softs Shadows ...

[43]GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation

[44]GPU Gems 3

[45]Modern C++ Design: Generic Programming and Design Patterns Applied
Andrei Alexandrescu
Livre de référence sur l'utilisation moderne des templates.

Annexes

Annexe 1 : Shader de rendu en traits

```
float4x4 Wvp   : WORLDVIEWPROJECTION;
float4x4 World         : WORLD;

float3 eyePos  : EYEPOS <string Space="Local";>;

sampler texSampler = sampler_state
{
        texture = <tex>;
        MipFilter = LINEAR;
        Minfilter = LINEAR;
        Magfilter = LINEAR;
        AddressU  = WRAP;
        AddressV  = WRAP;
        AddressW  = CLAMP;
};

//--- Vertex Shader
VS_OUTPUT TransformBasic( float3 Pos : POSITION,
                                                        float3 Norm : NORMAL,
                                                        float3 UV : TEXCOORD0)
{
        float3 wPos = Pos;
        VS_OUTPUT Out;
        Out.Pos         = mul(float4(Pos, 1), Wvp);
        Out.UV          = UV;
        Out.UVView      = float3(Out.Pos.x*0.5f, -Out.Pos.y*0.5f, Out.Pos.w);
        Out.CamDir      = eyePos-wPos;
        return Out;
}

//--- Pixel Shader
float4 BumpPS( VS_OUTPUT p ) : COLOR
{
        float2 uv = p.UVView.xy / p.UVView.z + float2(0.5f,0.5f);
                // calcul de la coordonnée uv dans l'espace écran

        uv.y *= 0.75f; // compense le format 4/3 de l'image

        float4 lightDir = tex2D( lightMap, uv); // récupère la couleur et intensité de la lumière
        float Intensity = lightDir.w;
        float3 uv3D = float3(p.UV, Intensity);

        float4 finalCol = tex3D(texSampler, uv3D);
                // récupère les traits de la texture 3D en utilisant l'intensité lumineuse

        float dist = length(p.CamDir) / 9.0;
        finalCol *= saturate(saturate(Intensity*3)+saturate(1-dist));
                // gestion de la disparition avec la distance

        finalCol = saturate((finalCol-0.5f)*10.f+0.5f);     // augmentation du contraste des traits

        finalCol.xyz *= (lightDir.xyz); // multiplie par la couleur le la lumière

        return finalCol;
}
```

Annexe 2 : Calcul des ombres

```
float4x4 Wvp   : WORLDVIEWPROJECTION;
float4x4 Vp    : VIEWPROJECTION;
float4x4 World: WORLD;
float4x4 Wi    : IWorld;
float4x4 WIT   : ITWorld;
float4x4 WT    : TWorld;



//--- Vertex Shader commun
VS_OUTPUT_RENDER VSLightRender( float3 Pos            : POSITION  )
{
      VS_OUTPUT_RENDER Out;
      Out.Pos      = mul( float4(Pos , 1.0f), Wvp);
      Out.UVView   = float3(Out.Pos.x*0.5f, -Out.Pos.y*0.5f, Out.Pos.w);
      return Out;
}




//--- Lumière spot avec des ombres :

float4 CompConeValue( float3 objPos, float3 objNorm, VS_OUTPUT_RENDER p)
{           // calcule l'intensité lumineuse pour un spot
      float3 lightDir         = mul(float4(objPos , 1), Wi);
      // couleur et intensité de la lumière
      float curLight          = length(lightDir);
      clip(1.0f-curLight); // stoppe le shader si noir

      float3 normDir          = mul(float4(objNorm, 1), WT);
      normDir                 = normalize(normDir);
      lightDir = -lightDir/curLight;

      curLight = pow(curLight,2); // divers calcul de la luminosité
      curLight = saturate(1.0f-curLight);
      curLight *= dot(lightDir, normDir);
      curLight *= saturate( ( dot(lightDir, float3(0,0,-1))-0.82f ) * 10 );

      clip(curLight-0.01f); // stoppe le shader si noir

      return float4(colDif, saturate(curLight));
}



float ProcessShadowTest(float2 uvShadow, float pxlDist)
{ //  effectue un test d'ombrage dans le Zbuffer de la lumière
      return ((1/tex2D(Shadow, uvShadow.xy)) > pxlDist);
}
```

```
float4 PSLightRender( VS_OUTPUT_RENDER p) : COLOR
{
      float2 uv = p.UVView.xy / p.UVView.z + float2(0.5f,0.5f);
      uv.y *= 0.75f; // calcul des uv de la vue

      float3 objNorm = tex2D( ObjNormals   , uv); // world normal
      float3 objPos  = tex2D( ObjPositions , uv); // world position

      float4 coneValue = CompConeValue(objPos, objNorm, p);

      float4 uvShadow = mul( float4(objPos,1) , mul(CamInvMatr,CamMatr ));
      float2 uvShad = uvShadow.xy * 0.5f;
      uvShad .y = -uvShad .y;
      uvShad /= uvShadow.w;
      uvShad .xy += float2(0.5f,0.5f); // calcul des uv vue par la lumière

      float pxlDist = length(objPos - CamPosition) - 0.05f;
      float shad = 0.0f;
      float2 jitle = objPos.xy*objPos.z * 1.0f; // decalage des samples

      shad += ProcessShadowTest(uvShad + float2(ShadowFilterSize *
            cos(jitle.x), 0), pxlDist );
      shad += ProcessShadowTest(uvShad + float2(0, ShadowFilterSize *
            sin(jitle.y)), pxlDist );

      return coneValue*shad*0.5f;
}
```

```
//--- Lumière omnidirectionnelle avec des ombres :

float4 CompSphereValue( float3 objPos, float3 objNorm, float3 lightDir)
{     // calcule la luminosité dans une sphere
      float  curLight         = length(lightDir);
      clip(1.0f-curLight); // stoppe le shader si noir

      float3 normDir          = mul(float4(objNorm, 1), WT);
      normDir                 = normalize(normDir  );

      lightDir = -lightDir/curLight; // calculs de luminosité
      curLight = pow(curLight,2);
      curLight = saturate(1.0f-curLight);
      curLight *= dot(lightDir, normDir);

      return float4(colDif, saturate(curLight));
}
```

```
float ProcessShadowTest(float3 uvShadow, float pxlDist)
{ // test de la texture de Z dans une cubemap
      return ((1/texCUBE(CubeMapZ, uvShadow)) > pxlDist);
}
```

```
float4 PSLightRender( VS_OUTPUT_RENDER p) : COLOR
{
	float2 uv = p.UVView.xy / p.UVView.z + float2(0.5f,0.5f);
	uv.y *= 0.75f; // calcul des uv de la vue

	float3 objPos  = tex2D( ObjPositions , uv); // world normal
	float3 objNorm = tex2D( ObjNormals   , uv); // world position

	float3 lightDir         = mul(float4(objPos , 1), Wi);
	float4 sphereValue      = CompSphereValue(objPos, objNorm, lightDir);
	lightDir                = normalize(lightDir);
	float3 normShad         = mul(float4(lightDir,1), WIT);
	float pxlDist           = length(objPos - CamPosition) - 0.05;
		// diminue la taille du filtre au dela d'une certaine distance
	float3 jitle = objPos * 10;
	float shad = 0;

	shad += ProcessShadowTest(normShad + float3(cos(jitle.x) *
		ShadowFilterSize, 0, 0), pxlDist);
	shad += ProcessShadowTest(normShad + float3(0, sin(jitle.y) *
		ShadowFilterSize, 0), pxlDist);
	shad += ProcessShadowTest(normShad + float3(0, 0,-cos(jitle.z) *
		ShadowFilterSize), pxlDist);

	return saturate(sphereValue*shad*0.33334f);
}

//--- Définition commune de la technique

technique LightRender
{
	pass p
	{
		AlphaBlendEnable   = true;
		AlphaFunc          = NEVER;
		BlendOp            = ADD;
		DestBlend          = INVSRCALPHA;
		SrcBlend           = SRCALPHA;

		ZWriteEnable       = false;
		ZFunc              = GREATER;
		// le zBuffer doit encore contenir le z du décor
		CullMode           = CW;

		VertexShader = compile vs_2_0 VSLightRender();
		PixelShader  = compile ps_3_0 PSLightRender();
	}
}
```

Annexe 3 : Simulation simple de fluides

```
float2 GetVelocity(float2 coords, float2 decalCoord, float2 velCoord)
{
	float2 baseVel  = tex2D(velBase , float2(decalCoord.x, decalCoord.y));
	 // texture de vélocité fixe
	float4 modifVel = tex2D(velocity, float2(decalCoord.x, decalCoord.y));
	 // texture d'ajout de vélocité (rendu à par)
	float2 coordsModif = baseVel + modifVel.xy * modifVel.w;
	 // calcul de l'origine supposée de l'encre
	return velCoord - TimeStep * SizeGrid * coordsModif;
}

float4 PSReverseAdvect( VS_OUTPUT_REVERSE p) : COLOR
{
	float time2 = time  * 1.5f;
	float2 uv  = p.UV + float2(-sin(time2 * 2)*sin(p.UV.x*22)*0.01,
		sin(time2*2.21)*sin(p.UV.y*31)*0.011) * 0.3;
	float2 uv2 = p.UV + float2(-sin(time2 * 2)*sin(p.UV.x*22)*0.01,
		sin(time2*2.21)*sin(p.UV.y*31)*0.011) * 0.3;
	// le décalage oblige le fluide à être toujours en mouvement

	float2 pos = GetVelocity(p.UV, uv, uv2);

	return tex2D(inkToAdvect, pos);
}
```

Annexe 4 : Simulation simple de nuages

```
float4 PSCloud( VS_OUTPUT_RENDER p) : COLOR
{
	float time2 = time  * 0.5f; float time1 = time2 * 0.1f;

	float3 nSunPos = normalize(SunPos);
	float3 SunDir  = (float3(p.Posi);  float  SunDist = length(SunDir);
	SunDir /= SunDist; // recuperation de la position du soleil

	float2 decalUV = float2(-sin(time2 * 2)*sin(p.UV.x*22)*0.01,
		sin(time2*2.21)*sin(p.UV.y*31)*0.011) );
		// calcul du mouvement à partir de sinusoides

	float3 bumpNorm = tex2D( bumpSampler, p.UV + decalUV);
	bumpNorm = TransformNormal( bumpNorm, p );
	 // transformation de la normale prise dans la texture en tangent space

	float Intensity = dot(SunDir, bumpNorm)*(1/SunDist)*0.2 + 0.5;

	float2 decalCloud = bumpNorm.xz * 2 - float2(1.0, 1.0);
	float2 uvCloud = p.UV + decalCloud * 0.1 + float2(time1, time1*0.7);
	float cloudInt = tex2D(noiseMap, uvCloud);
	 // recuperation de l'intensité

	cloudInt = saturate((cloudInt-0.5)*2) + 0.5;
	cloudInt *= Intensity;
	float alpha = saturate((cloudInt-cloudLimit)*cloudColor.w*10);
	 // saturation appropriée

	return float4(cloudColor.xyz*cloudInt * alpha +
		backColor.xyz * (1-alpha), saturate(p.Posi.y*5));
		// calcul des couleurs
}
```

Annexe 5 : Calcul principal de la simulation de tissu

```
// verlet integration : P(t+ Ät) = P(t) + k (P(t) -P(t-Ät)) + Ät2F(t)
float3 Integrate(float3 x, float3 oldx, float3 a, float timestep2, float damping)
{
	return x + damping*(x - oldx) + a*timestep2;
}

// Constraints :

float3 DistanceConstraint(float3 x, float3 x2, float restlength, float stiffness)
{
	float3 delta = x2 - x;
	float deltalength = length(delta);
	float diff = (deltalength - restlength) / deltalength;
	return delta *stiffness*diff;
}

float3 SphereConstraint(float3 oldx, float3 x, float3 center, float r)
{
	float3 delta = x - center;
	float dist = length(delta);
	if (dist < r)
	{
		x = center + delta*(r / dist);
		x = lerp(x,oldx, floorResist);
	}
	return x;
}

float3 CylinderConstraint(float3 oldx, float3 x, float4x4 mat, float4x4 invMat)
{
	float3 pos = mul(float4(x,1.0f),invMat);
	float dist = length(pos.xy);
	float distZ = length(pos.z);
	if(dist<1.0f && distZ<1.0f)
	{
		pos                 = float3(normalize(pos.xy),pos.z);
		pos                 = mul(float4(pos,1.0f), mat);
		x = pos;
		x = lerp(x,oldx, floorResist);
	}
	return x;
}

float3 FloorConstraint(float3 oldx, float3 x, float level)
{
	if (x.y<level) // level est la hauteur du sol
	{
		x.y = level;
		x = lerp(x,oldx, floorResist);   // résistance aux frottements
	}
	return x;
}

float3 WindComp(float3 normal)
{
	return -normal * dot(normal,windForce); // vecteur direction et force du vent
}

//--- Pixel Shader (le vertex shader est tout simple))
float4 PSClothSimulation( VS_OUTPUT p) : COLOR // fait pour chaque sommets du tissu
{
	float3 prevPosition = tex2D(prevSampler , p.UV);   // récupération de la position n-2
	float3 nextPosition = tex2D(nextSampler , p.UV);   // récupération de la position n-1
	float  blockVertex  = tex2D(blockSampler, p.UV).r; // texture des sommets fixes
	float3 normal       = tex2D(normSampler , p.UV);   // normale (calculé séparément)
```

```
	float3 acceleration = float3(0.0f,-1.0f,0.0f)*gravity; // gestion de la gravité
	acceleration        += WindComp(normal);  // gestion de la force du vent

	float3 newPosition = Integrate(nextPosition, prevPosition, acceleration,
		timeStep*timeStep , damping); // calcul de la position suivante

	newPosition = ClothConstraint(p, newPosition); // contraintes entre les sommets
	newPosition = CylinderConstraint(nextPosition, newPosition, cylColideMat,
		cylColideInvMat); // collision cylindrique

	newPosition = lerp(newPosition,objPosDecal+nextPosition,blockVertex);
		// gestion des sommets fixes

	return float4( newPosition, 1.0f);
}

float3 StringConstraint(float x, float y, float3 nextPosition, float2 baseUV, float restDist)
{
	float3 posNeigh = tex2D(nextSampler , baseUV  + float2(x*pxlSize, y*pxlSize) );
	return DistanceConstraint(nextPosition , posNeigh, restDist, stiffness);
}

float3 ClothConstraint(VS_OUTPUT p, float3 nextPosition)
{
	float2 uv = p.UV; float iX = uv.x * nbVertex; float iY = uv.y * nbVertex;

	float diagDist  = constraintDist*rac2; // précalcul des distances entre les sommets
	float diag2Dist = constraintDist*rac5;

	float a = nbVertex-1;
	float b = nbVertex-2;

	// les 8 voisins    directs
	if(iX>1 &&   iY>1) nextPosition += StringConstraint(-1,-1,nextPosition,uv,diagDist);
	if(iX<a &&   iY<a) nextPosition += StringConstraint( 1, 1,nextPosition,uv,diagDist);
	if(iX>1 &&   iY<a) nextPosition += StringConstraint(-1, 1,nextPosition,uv,diagDist);
	if(iX<a &&   iY>1) nextPosition += StringConstraint( 1,-1,nextPosition,uv,diagDist);

	if(iX>1) nextPosition += StringConstraint(-1, 0,nextPosition,uv,constraintDist);
	if(iX<a) nextPosition += StringConstraint( 1, 0,nextPosition,uv,constraintDist);
	if(iY>1) nextPosition += StringConstraint( 0,-1,nextPosition,uv,constraintDist);
	if(iY<a) nextPosition += StringConstraint( 0, 1,nextPosition,uv,constraintDist);

	// les 16 suivants
	if(iX>2 && iY>2) nextPosition += StringConstraint(-2,-2,nextPosition,uv,diagDist * 2);
	if(iX<b && iY<b) nextPosition += StringConstraint( 2, 2,nextPosition,uv,diagDist * 2);
	if(iX>2 && iY<b) nextPosition += StringConstraint(-2, 2,nextPosition,uv,diagDist * 2);
	if(iX<b && iY>2) nextPosition += StringConstraint( 2,-2,nextPosition,uv,diagDist * 2);

	if(iX>2) nextPosition += StringConstraint(-2, 0,nextPosition,uv,constraintDist * 2);
	if(iX<b) nextPosition += StringConstraint( 2, 0,nextPosition,uv,constraintDist * 2);
	if(iY<b) nextPosition += StringConstraint( 0, 2,nextPosition,uv,constraintDist * 2);
	if(iY>2) nextPosition += StringConstraint( 0,-2,nextPosition,uv,constraintDist * 2);

	if(iX>2 &&   iY>1) nextPosition += StringConstraint(-2,-1,nextPosition,uv,diag2Dist);
	if(iX<b &&   iY<a) nextPosition += StringConstraint( 2, 1,nextPosition,uv,diag2Dist);
	if(iX>1 &&   iY<b) nextPosition += StringConstraint(-1, 2,nextPosition,uv,diag2Dist);
	if(iX<a &&   iY>2) nextPosition += StringConstraint( 1,-2,nextPosition,uv,diag2Dist);

	if(iX>2 &&   iY<a) nextPosition += StringConstraint(-2, 1,nextPosition,uv,diag2Dist);
	if(iX<b &&   iY>1) nextPosition += StringConstraint( 2,-1,nextPosition,uv,diag2Dist);
	if(iX<a &&   iY<b) nextPosition += StringConstraint( 1, 2,nextPosition,uv,diag2Dist);
	if(iX>1 &&   iY>2) nextPosition += StringConstraint(-1,-2,nextPosition,uv,diag2Dist);

	return nextPosition;
}
```