Université Joseph Fourier de Grenoble DEA IVR 2005

Représentations Alternatives pour l'Affichage d'Objets Lointains

Lionel BABOUD

Rapport de projet de DEA

Artis/GRAVIR/IMAG-INRIA. UMR CNRS C5527.

Composition du jury :

Gilles	DEBUNNE	Directeur de stage
Fabrice	NEYRET	Rapporteur externe
Nicolas	HOLZSCHUCH	Membre du jury
Augustin	LUX	Membre du jury
James	CROWLEY	Membre du jury

Table des matières

Chapitre 1

Introduction

1.1 Exposé du problème

L'objectif de ce stage est de proposer une méthode pour afficher les objets lointains d'une scène 3D. Nous appelons *objet lointain* un objet situé loin de la caméra, sur lequel l'attention n'est pas particulièrment portée et qui occupe une surface à l'écran de moins de 100×100 pixels.

Une telle méthode doit permettre d'afficher un grand nombre d'objets sans trop affecter le temps de rendu de l'image. Les principaux objectifs sont donc la rapidité d'affichage et le faible besoin en mémoire.

Malgré la faible surface occupée à l'écran par les objets lointains, leur présence permet d'augmenter de manière significative le degré de réalisme d'une scène. Ces objets gardent donc une importance relative assez élevée, ce qui motive l'intérêt d'avoir des méthodes efficaces pour les dessiner. Nous commençons par fixer un cadre à ce problème, ainsi que des objectifs à atteindre.

1.2 Cadre

1.2.1 Restrictions

Pour simplifier ce problème nous nous sommes fixé plusieurs restrictions. La plus importante est que l'objet ne peut être affiché que depuis un ensemble restreint de points de vue : on se fixe un axe (que nous appellerons par la suite l'*axe de rotation*) et un plan orthogonal à cet axe et on impose à la caméra de se trouver dans ce plan, dirigée vers un point du plan.

Cette simplification est raisonnable dans la mesure où les objets d'une scène ont souvent une liberté d'orientation restreinte. Par exemple dans le cadre de l'affichage d'un environnement urbain (bâtiments, véhicules, passants) la plupart des objets gardent une orientation verticale fixe. Le sol est alors le plan naturel où se déplace la caméra et sur lequel peuvent tourner les objets. En effet, un objet tournant sur lui-même (camion dans un virage) et une caméra tournant autour d'un objet sont deux

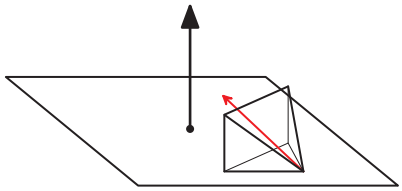


FIG. 1.1 – Restriction du point de vue de la caméra

manières duales de considérer un même mouvement de rotation.

La deuxième restriction est liée à la nature même du problème posé : la taille à l'écran des objets traités est limitée (de l'ordre de moins de 100×100 pixels), on peut se permettre une restitution visuellement approximative, en imposant quand même une transition acceptable (continue) entre la version simplifiée et la version originale. D'autre part le modèle d'éclairage peut être simplifié si le résultat n'en est pas trop affecté. On peut aussi se permettre de remplacer la projection perspective de l'objet à l'écran par une projection orthogonale en raison de la distance de l'objet à la caméra et de sa taille à l'écran.

1.2.2 But fixé

Le principal objectif visé est d'obtenir un qualité visuelle acceptable en utilisant le moins possible de temps de calcul (CPU) et d'affichage (GPU) et de mémoire pour laisser ces ressources libres pour les traitements plus importants comme l'affichage du reste de la scène. L'utilisation d'une certaine quantité de ces ressources est néanmoins justifiée dans la mesure où les objets lointains, malgré la faible surface occupée à l'écran ont une importance relative élevée dans le réalisme et la richesse de la scène affichée.

Pour obtenir une qualité visuelle suffisante on s'impose d'avoir un effet de parallaxe correct pour que la sensation de rotation autour de l'objet soir restituée de manière acceptable.

Le modèle d'éclairage minimal que nous nous fixons est un éclairage diffus qui permette de rééclairer l'objet à partir d'une position quelconque de la source lumineuse. On ne s'interdit pas de pouvoir éventuellement restituer des effets d'éclairage plus complexes liés aux matériaux qui composent l'objet comme l'éclairage spéculaire par exemple.

1.2.3 Méthode

Pour atteindre l'objectif fixé, nous avons choisi d'utiliser l'approche des représentations alternatives, et plus précisément de la modélisation à base d'images, aussi bien pour l'acquisition que pour la représentation.

L'acquisition de données à partir d'images de l'objet presente notamment les avantages suivants :

– on s'abstrait de la représentation initiale de l'objet modélisé (maillage polygonal, surfaces spline, équation mathématique rendue par raytracing, etc.).
– on peut capturer les éventuels effets d'éclairage complexes des matériaux qui composent l'objet sans connaissance à priori de sa représentation initiale.
– les images de l'objet peuvent être des images réelles (photographies, vidéo) et ainsi fournir une modélisation très réaliste de l'objet.

Les images ainsi obtenues servent également à représenter l'objet ce qui permet notamment d'utiliser les algorithmes de compression et de filtrage d'image pour les appliquer à l'objet. L'utilisation de coordonnées de textures normalisées entre 0 et 1 permet également de changer la taille des images sans modifier la méthode.

1.3 Idée de départ

1.3.1 Échantillonnage des vues autour de l'objet

L'idée dont nous sommes partis est la suivante : prendre N vues de l'objet en tournant autour de l'axe de rotation et mémoriser les images obtenues (voir figure 1.2). Nous avons cherché à voir comment combiner ces images (en utilisant éventuellement des informations supplémentaires si nécessaire) pour calculer une vue quelconque de l'objet. La cohérence des couleurs entre deux images successives permet à priori d'utiliser des méthodes de compression et d'interpolation de couleurs pour diminuer la place mémoire nécessaire.

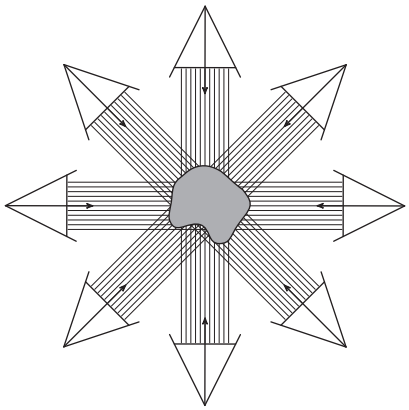


FIG. 1.2 – Capture de N images avec une caméra orthographique en tournant autour de l'objet

1.3.2 Version duale

En raison de la redondance d'information entre deux images successives nous avons également considéré l'image obtenue en ne retenant pour chaque image que sa colonne de pixels centrale et en collant à la suite ces colonnes de pixels (voir figure

1.3). Nous appelerons par la suite cette image une *texture cylindrique*. Pour des images de taille $N_x \times N_y$ le coût mémoire passe donc de $O(N \times N_x \times N_y)$ à $O(N \times N_y)$ et la texture cylindrique obtenue laisse envisager des possibilités de compression liées à la cohérence radiale des couleurs.

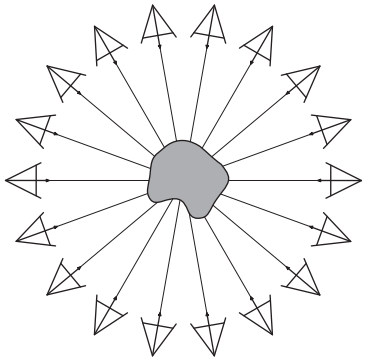
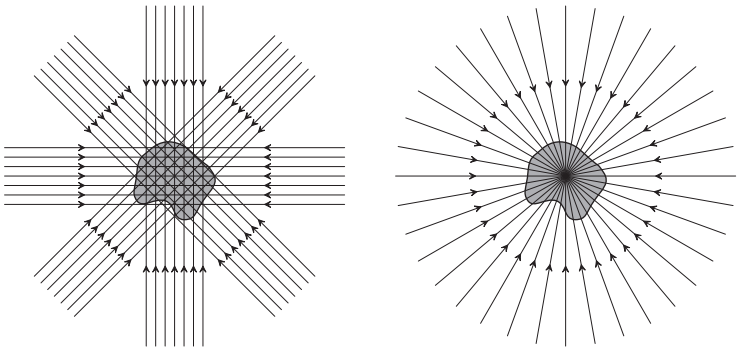


FIG. 1.3 – Capture d'une texture cylindrique

1.3.3 Comparaison des deux approches

Les deux approches peuvent être vues comme des cas particuliers de *Light Fields*[1] [LH96] avec pour la première approche beaucoup de rayons lumineux dans une même direction mais peu de directions, et dans la seconde un seul rayon lumineux par direction mais beaucoup de directions de vue.



(a) (b)

FIG. 1.4 – Comparaison des deux approches : (a) plusieurs images orthographiques, (b) une texture cylindrique

[1] nous décrivons cette techique dans l'état de l'art

1.4 Organisation du document

Nous présentons d'abord un état de l'art sur les méthodes existantes en rapport avec notre problème.

Les deux parties suivantes présentent notre contribution : la partie 3 expose la première piste de recherche que nous avons suivie, l'étude de la représentation d'objets par une texture cylindrique (figure 1.3) et la partie 4 traite de la seconde piste de recherche considérée, l'utilisation de plusieurs vues de l'objet obtenues en tournant autour pour redessiner l'objet d'un point de vue quelconque (figure 1.2).

Enfin nous concluons en expliquant notre méthode de travail, en commentant les résultats obtenus et en proposant des pistes de recherches futures possibles.

Chapitre 2

État de l'art

2.1 Approches existantes pour l'affichage d'objets lointains

Une première solution naïve pour traiter les objets lointains est simplement de ne pas les afficher. Le problème de cette approche est que le fond des scènes ainsi obtenues parraît vide et cela pose des problèmes de sauts à l'affichage lorsqu'un objet apparaît ou disparaît brutalement. Cette méthode n'est plus acceptable de nos jours mais reste utile lorsque les ressources sont très faibles.

Une approche plus évoluée consiste à utiliser une texture (*billboard*) calculée à partir d'un point de vue de l'objet et à l'afficher directement à l'écran. La texture est plaquée sur une face rectangulaire ayant une orientation fixe par rapport à la scène ou orientée pour toujours faire face à la caméra. La première solution est convenable quand l'objet à afficher se trouve sous un point de vue proche de celui qui a été utilisé pour la capture de la texture mais aucun effet de rotation ni de parallaxe n'est restitué. La deuxième solution donne des résultats convenables si l'objet a une symétrie de révolution, comme c'est la cas par exemple pour un arbre vu de loin.

La méthode des *Billboard Clouds* [DDSD03] généralise cette approche et est conçue pour la simplification extrême d'un objet. Elle permet d'afficher un objet à partir d'un nombre relativement faible de portions de *billboards* texturés, calculés pour minimiser l'erreur visuelle induite par l'approximation. Le rendu est très rapide mais les erreurs visuelles peuvent être génantes et le coût mémoire est relativement élevé.

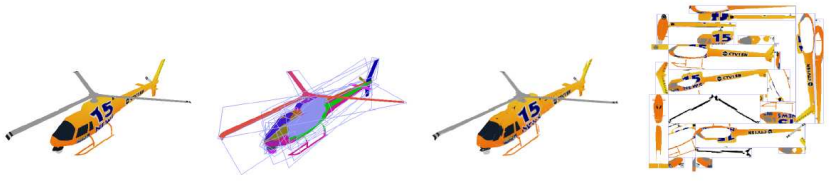


FIG. 2.1 – Illustration des *Billboard Clouds*

Une autre approche similaire est l'utilisation de plusieurs vues précalculées de l'objet, puis au moment de l'affichage le mélange des images correspondant aux points de vues les plus proches de celui de la caméra. L'inconvénient de cette technique est que pour obtenir des transitions entre les vues suffisamment fluides il faut stocker beaucoup de vues et donc utiliser une grosse quantité de mémoire. Cette méthode peut être vue comme un cas particulier de *Bidirectional Texture Function* (technique exposée dans [TZL+02] que nous décrivons plus en détail dans la partie suivante), qui consiste à échantillonner la fonction 6D qui à un point sur une surface, une direction de vue et une direction lumineuse associe la couleur correspondante. Des techniques de compression de données permettent de réduire considérablement le coût mémoire, mais il reste malgré tout très élevé.

Une autre méthode couramment appliquée est la simplification de maillage [GH97]. Des algorithmes permettent de simplifier un objet maillé jusqu'à un certain point sans trop le déformer, mais ils sont mal adaptés à la simplification extrême requise pour les objets lointains. Un inconvénient de ces méthodes est que le maillage est simplifié en minimisant une erreur géométrique mais ne tient pas forcément compte de l'aspect visuel de l'objet. De plus elle ne peut bien sûr être appliquée qu'aux objets représentés par un maillage polygonal. Cette technique s'étend à l'utilisation de niveaux de détails de l'objet, avec les problématiques de transitions fluides entre niveaux.

Nous voyons donc qu'aucune méthode existante ne permet de résoudre le problème de l'affichage des objets lointains de manière complètement satisfaisante.

2.2 Méthodes en rapport avec notre problème

Nous allons maintenant présenter plusieurs techniques qui n'ont pas d'application directe à l'affichage d'objets lointains mais qui nous ont inspiré dans nos recherches. Nous avons décidé de rassembler leurs présentations dans cette partie pour ne pas encombrer la suite de l'exposé des contributions. Le lecteur pourra se référer à ces descriptions durant la lecture de la suite de ce document.

Échantillonnage de la fonction plénoptique Comme nous l'avons vu en introduction, notre approche consiste à échantillonner des rayons lumineux selon plusieurs directions et chercher à rééchantillonner les rayons lumineux passant par l'objet sous un angle quelconque. C'est la démarche commune des techniques de rendu à base d'images, explicitée dans [MB95] : échantillonner la *fonction plénoptique* 5D qui à un point dans l'espace et une direction associe l'intensité (la couleur) du rayon lumineux correspondant.

La technique des *Light Fields* [LH96, GGSC96] suit cette approche : pour un éclairage fixe, l'ensemble 4D des rayons lumineux intersectant l'objet est échantillonné (la fonction plénoptique 5D est ramenée à une fonction 4D en considérant que l'intensité lumineuse est constante le long d'un rayon). Un rayon lumineux est paramétré par les coordonnées de ses intersections avec deux plans parallèles fixes appelés *light slabs*. Les *Light Fields* capturent les effets d'éclairages complexes et permettent un rendu de haute qualité d'un objet, cependant ils demandent beaucoup de mémoire pour représenter l'objet.

Les *Bidirectional Texture Functions* [TZL+02] sont une généralisation de cette technique en faisant varier la direction de la source lumineuse : on échantillonne par des textures la fonction 6D qui correspond à un *Light Field* par direction de la source lumineuse. La quantité de données à stocker devient très grande et des méthodes de compression s'imposent. Comme pour les *Light Fields*, cette technique est plutôt destinée à un rendu de haute qualité que temps réel bien que certaines approches comme [MNP01] permettent d'obtenir des temps de rendu interactifs, le coût mémoire important étant amorti par l'utilisation de la *BTF* sur plusieurs objets identiques.

FIG. 2.2 – Calcul de BTF pour un arbre

Rendu à base de points Parmi les techniques d'affichage que nous présentons ensuite, certaines sont basées sur le rendu à base de points [GD98, KB04]. L'idée introduite par [LW85] est d'utiliser le point comme une méta-primitive permettant de modéliser une classe très large d'objets. La technique utilisée pour afficher un objet modélisé à base de points est le *splatting* (étalement des points à l'écran, comparable au pointillisme). Des techniques plus évoluées comme le *Surface Splatting* [PZvBG00, ZPvBG01] permettent de mieux résoudre le problème des trous à l'écran en affichant des ellipses dont l'orientation et les dimensions sont calculées en fonction de l'orientation de la surface échantillonnée et de la densité locale des échantillons. Les principaux intérêts du rendu à base de points sont leur généricité et les vitesses d'affichage qu'elles permettent d'obtenir.

Rendu de champs de hauteurs Nous allons par la suite être amenés à effectuer des calculs de rendu de champs de hauteurs. L'article présentant les *Layered Depth Images* [SGwHS98] donne deux représentations pour ajouter des informations de profondeur à une image : les *Sprites with depths* qui consistent simplement à rajouter à chaque pixel d'une image sa distance par rapport à la caméra au moment de son rendu, et les *Layered Depth Images* où chaque pixel contient plusieurs couches de profondeurs, permettant ainsi d'afficher des partie cachées lors du rendu de l'image. Dans les deux cas le rendu est effectué par un algorithme de *splatting*.

La représentation des *Relief Textures* [OBM00] permet de modéliser un objet à partir de six vues, une par face d'une boîte englobant l'objet, en stockant pour chaque vue une texture de couleurs et une carte de profondeurs. Lors de l'affichage de l'objet, pour chaque face visible de la boîte englobante la texture de couleur est prédéformée

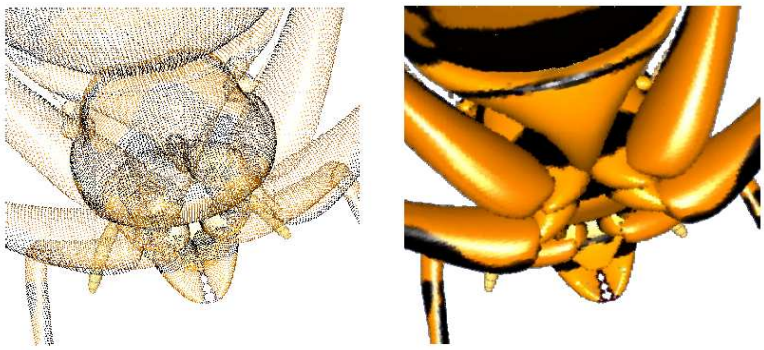


FIG. 2.3 – Dessin par splatting

en appliquant deux déformations 1D successivement, puis l'image obtenue est affichée à l'écran par un simple plaquage de texture sur la face de la boîte englobante correspondante. Cet algorithme permet une reconstruction exacte des champs de hauteurs composant l'objet au prix d'une déformation de texture relativement coûteuse et impossible à transposer en un programme executé par la carte graphique. De plus le fait de représenter l'objet à partir d'un nombre réduit de vues pose des problèmes d'occlusions pour certains objets, notamment les objets avec trous.

Pour résoudre le problème du coût de la recherche itérative de l'intersection d'un rayon lumineux avec un champ de hauteurs, la technique du *Parallax Mapping with Offset Limiting* [Wel04] approxime localement le champ de hauteurs par une fonction simple dont on sait calculer l'intersection avec le rayon lumineux facilement. L'algorithme d'affichage est donc très rapide mais l'approximation est trop grossière et ne permet que d'afficher des champs de hauteurs suffisamment lisses.

L'approche utilisée dans [Don05] consiste à plonger le champ de hauteurs dans une texture 3D et à précalculer pour chaque point de cette texture sa distance au champ de hauteurs. Ainsi une technique de *Sphere Tracing* permet de converger très vite vers l'intersection du rayon lumineux avec le champ de hauteurs. L'inconvénient de cette méthode est le coût mémoire important à cause du stockage d'une texture 3D en mémoire.

Chapitre 3

Textures cylindriques

3.1 Idée de départ

Les méthodes présentées dans ce chapitre sont basées sur l'idée suivante : prendre beaucoup de vues autour de l'objet mais ne garder horizontalement qu'un seul rayon par angle de vue pour limiter la redondance dans les couleurs mémorisées. On forme donc une texture cylindrique composée de la succession des colonnes centrales de pixels de chaque image.

FIG. 3.1 – La texture cylindrique d'un camion

3.2 Stockage des rayons

Il est clair qu'avec la simple donnée de la texture cylindrique il est impossible de reconstituer correctement l'objet : on ne garde notamment aucune information sur la silhouette de l'objet et il est impossible pour un rayon ne passant pas par l'axe de rotation de savoir à quelle abscisse chercher sa couleur dans la texture (l'idée naïve qui consisterait à plaquer la texture sur un cylindre ne restituerait évidemment pas la silhouette correcte ni le bon effet de parallaxe).

Il est donc nécessaire de garder une information supplémentaire : il nous a paru naturel de prendre pour chaque échantillon (c'est à dire pour chaque texel de la texture cylindrique) sa distance à l'axe de rotation. Nous appellerons par la suite cette distance le *rayon* de l'échantillon, par analogie avec la paramétrisation de l'espace en coordonnées cylindriques. Cette information permet de reconstituer complètement la position spatiale de l'échantillon comme nous le verrons.

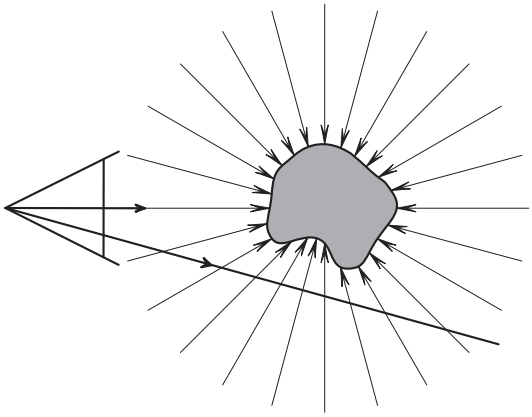


FIG. 3.2 – Manque d'information pour un rayon ne passant pas pas l'axe : on ne sait pas quel échantillon choisir.

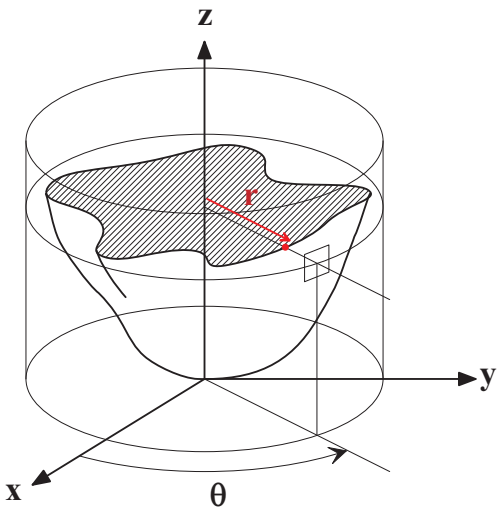


FIG. 3.3 – Reconstruction 3D d'un échantillon

Chaque échantillon correspond à un rayon lumineux orthogonal à l'axe de rotation et passant par et dirigé vers cet axe. Trois cas de figure sont possibles :

1. le rayon lumineux n'intersecte pas l'objet, on parlera de rayon *indéfini*
2. le rayon lumineux intersecte l'objet *avant* l'axe de rotation, le rayon de l'échantillon est *positif*
3. le rayon lumineux intersecte l'objet *après* l'axe de rotation, le rayon de l'échantillon est *négatif*

Nous allons d'abord étudier un cas simplifié de ce modèle où tous les rayons sont définis et positifs. Nous verrons ensuite les problèmes posés par l'ajout de rayons indéfinis, puis nous étudierons le cas général où un rayon peut appartenir à une des trois catégories.

3.2.1 Étude du modèle simplifié

Dans ce cas de modèle simplifié on impose à tous les rayons d'être positifs et définis. Nous appelons *tranche* de l'objet son intersection avec un plan orthogonal à l'axe de rotation. Les échantillons représentant une tranche correspondent à une ligne horizontale de pixels de l'image cylindrique.

Appelons T une de ces tranches et fixons un repère dont l'origine O est l'intersection du plan de la tranche avec l'axe d'échantillonnage. Soit d une droite passant par l'origine, d contient deux rayons lumineux d'échantillonnage l_1 et l_2 de sens opposés. Soient a_1 et a_2 les échantillons correspondants respectivement à l_1 et l_2 (ils existent forcément car tous les échantillons sont définis par hypothèse).

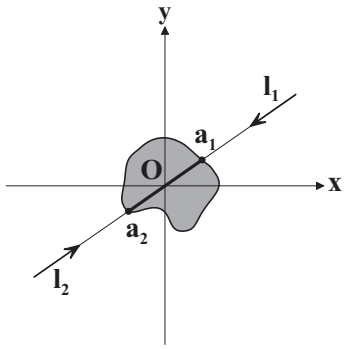


FIG. 3.4 – Analyse du cas simple

Pour que l'objet soit correctement représenté il faut donc que $T \cap d$ soit le segment $[a_1, a_2]$ complet car aucun rayon lumineux d'échantillonnage autre que l_1 ou l_2 ne nous donne d'information sur l'intérieur de ce segment. Comme le segment $[a_1, a_2]$ contient O, cette propriété implique que T doit être *étoilé* par rapport à O. Un objet correctement représenté par ce modèle doit donc être tel que chacune de ses tranches est étoilée par rapport à l'axe d'échantillonnage. Ce modèle est donc assez restrictif et la classe d'objets représentables est réduite.

3.2.2 Ajout de rayons indéfinis

Permettre à certains échantillons de ne pas être définis élargit la classe d'objets représentable au prix de l'apparition de trous dans l'objet aux endroits correspondants. Se pose alors le problème de l'affichage de l'intérieur de l'objet, qui peut maintenant être visible à travers un trou.

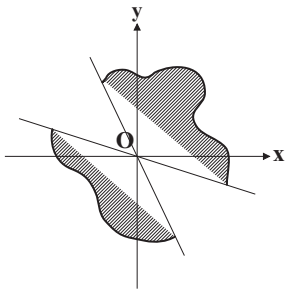


FIG. 3.5 – Problème de trous

3.2.3 Étude du modèle général

Gardons les mêmes notations que pour l'etude du cas simple. On a toujours $T \cap d = [a_1, a_2]$ mais comme les rayons ne sont plus forcément positifs, l'intervalle $[a_1, a_2]$ ne contient pas nécéssairement le point O, la tranche T n'est donc pas obligatoirement étoilée. Par contre si T a plusieurs composantes connexes, chacune est inclue dans un secteur angulaire et ces secteurs angulaires sont disjoints.

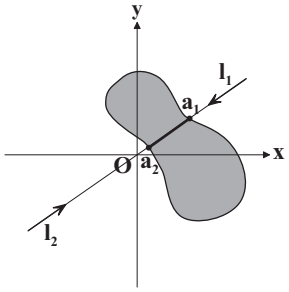


FIG. 3.6 – Analyse du cas général

Cette représentation est moins sensible au positionnement de l'axe d'échantillonnage que la représentation simplifiée et permet donc de modéliser plus d'objets.

La classe des objets représentables par ce modèle est donc relativement large mais ne contient pas n'importe quel objet. Notamment, les objets ayant des parties alignées par rapport à l'axe comme les pieds d'une chaise ou les roues d'un véhicule ne sont pas échantillonnés correctement.

Nous présentons maintenant la manière dont sont échantillonnés les objets.

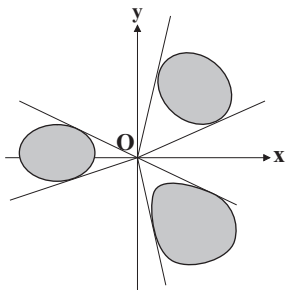


FIG. 3.7 – Objet avec plusieurs composantes connexes

3.3 Échantillonnage de l'objet

Pour échantillonner l'objet il faut d'abord choisir un axe. Une fois celui-ci fixé on effectue N rendus de l'objet en tournant autour de l'axe avec une caméra orthographique réglée de sorte que l'axe soit toujours vertical et centré à l'écran. De chacun de ces rendus on extrait la colonne centrale de fragments dont on garde la couleur et la profondeur. La profondeur d'un fragment correspond (à une conversion dans les bonne unités près) à la valeur du rayon de l'échantillon. Si ce rayon est négatif et qu'on se place dans le cas simplifié, on le considère comme indéfini. Les colonnes de couleurs (resp. de rayons) sont collées ensembles dans l'ordre dans lequel elles ont été capturées, on obtient ainsi l'image de couleurs et l'image de rayons servant à représenter l'objet.

L'objet est placé dans un cylindre englobant de rayon R ayant pour axe l'axe de rotation. Ceci permet d'un point de vue pratique de renormaliser les rayons sur l'intervalle $[-1,1]$, mais aussi à borner l'objet et accélérer les algorithmes d'affichage.

Pour obtenir un bon échantillonnage de l'objet, la texture est d'abord acquise à une grande taille puis sous-échantillonnée à la taille voulue. De plus la représentation sous forme de textures permet de s'abstraire de la résolution réelle des textures utilisées, on peut donc sous-échantillonner les textures pour réduire la place mémoire occupée et ainsi obtenir un compromis réglable entre qualité à l'affichage et coût mémoire.

Pour pouvoir par la suite rééclairer l'objet on capture également une normale par échantillon. Lors de l'affichage de l'objet, la couleur de chaque pixel est modulée par un calcul d'éclairage diffus pour tenir compte de la position de la source lumineuse.

Pour chaque texel de la texture cylindrique il faut mémoriser une couleur (avec éventuellement une opacité), un rayon et une normale. Avec une précision standard (un octet par canal par pixel) on peut mémoriser un pixel sur 4 octets, donc il faut deux textures pour stocker un objet. Ces textures sont de taille $L_x \times L_y$, où L_x est le nombre d'échantillons radiaux et L_y le nombre d'échantillons verticalement.

Expérimentalement, une texture cylindrique de 256 échantillons radiaux par 64 en hauteur donne de bons résultats. La représentation d'un objet avec cette résolution occupe 128 Ko.

3.4 Algorithmes d'affichage simples

Nous présentons d'abord plusieurs algorithmes d'affichage simples basés sur une reconstruction 3D des échantillons, avec plusieurs manières de les afficher. La partie suivante traite d'une autre approche du problème, basée sur la déformation de textures.

(a) (b) (c) (d)

FIG. 3.8 – Les différentes méthodes d'affichage : (a) original, (b) triangles, (c) points, (d) lignes

3.4.1 Maillage avec triangles

La solution la plus directe et qui nous a servi de solution de référence pour les algorithmes d'affichage, est de dessiner le maillage à base de triangles (figure 3.8(b)) induit par la paramétrisation cylindrique de l'objet : pour un échantillon (i, j) on dessine les triangles $(p_{i,j}\ p_{i+1,j}\ p_{i,j+1})$ et $(p_{i+1,j}\ p_{i+1,j+1}\ p_{i,j+1})$ où $p_{u,v}$ désigne la position 3D correspondant à l'échantillon (u,v). Certains triangles n'ont pas de signification et ne doivent pas être dessinés :

- les triangles faisant intervenir un échantillon indéfini
- les triangles dont un point correspond à un échantillon de rayon positif et un autre à un échantillon de rayon négatif.

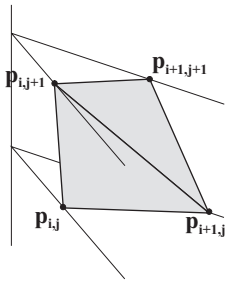


FIG. 3.9 – Deux triangles du maillage reconstruit

Cette méthode permet de reconstruire exactement l'objet représenté (notamment la silhouette et la parallaxe sont restitués correctement), au prix d'un nombre de triangles élevé à afficher (dans certains cas le maillage reconstruit est constitué de plus de

triangles que l'objet initial). Nous avons donc cherché à utiliser des primitives moins coûteuses que les triangles.

3.4.2 Splatting de points

Une solution plus simple et qui évite les problèmes de connectivité entre les échantillons rencontrés avec l'affichage par maillage est de considérer l'ensemble des échantillons comme un nuage de points et de l'afficher par *splatting* (figure 3.8(c)) . La technique courrament utilisée est d'afficher une ellipse pour chaque point en choisissant ses dimensions de manière à combler les trous entre les échantillons le mieux possible. L'échantillonnage sur une grille cubique utilisé par la méthode des *Surfels* [PZvBG00] permet de garantir une densité minimale d'échantillonnage. Le problème est que le caractère radial de l'échantillonnage empèche de déterminer une distance maximale entre échantillons sans imposer de condition trop stricte sur la continuité de la surface de l'objet.

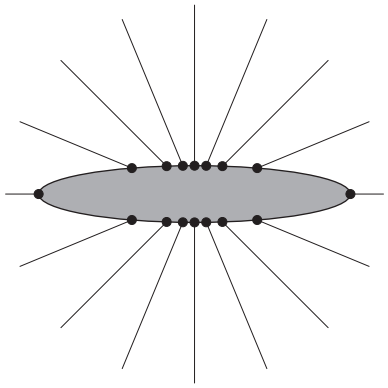


FIG. 3.10 – Irrégularité de l'échantillonnage radial pour un objet allongé

Ce problème n'est pas génant dans le cas d'un arbre ou les échantillons sont indépendants mais interdit l'affichage de surfaces continues.

La fréquence d'echantillonnage verticale étant constante, la hauteur des ellipses est facile à déterminer, mais l'irrégularité de l'échantillonnage horizontal empèche de trouver une largeur d'ellipse convenable. Pour résoudre ce problème on à recours à des lignes plutôt qu'à des ellipses, ce que nous décrivons dans la partie suivante.

3.4.3 Splatting de lignes

Nous avons cherché à allier les avantages de l'approche des triangles (pas de problèmes de trous) à ceux de l'approche des points (pas de problèmes de connectivité). Le bon compromis est d'utiliser des lignes entre les échantillons horizontaux : on n'a plus de problème de connectivité entre tranches et on remplit les trous horizontalement (figure 3.8(d)). L'espace entre deux tranches successives est rempli en adaptant l'épaisseur des lignes en fonction de la distance entre deux tranches successives à

l'écran, qui est constante. Comme on connecte deux échantillons entre eux il reste des cas particuliers à traiter mais beaucoup plus simples : il suffit de ne pas dessiner une ligne quand :

– un des deux échantillons est indéfini
– les deux rayons ne sont pas de même signe

On peut aussi sans trop compliquer l'algorithme d'affichage ajouter des lignes pour remplir certains trous dus au caractère discret de l'échantillonnage : l'« ouverture » ou la « fermeture » d'une composante connexe d'une tranche de l'objet.

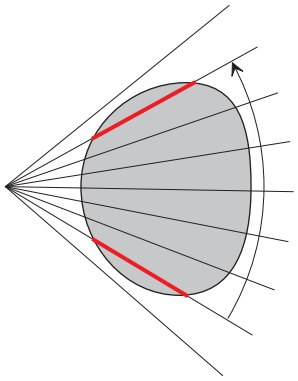


FIG. 3.11 – Ajout de segments supplémentaires pour fermer une composante connexe

D'un point de vue théorique la rasterisation d'une ligne horizontale demande très peu de calculs et nous nous attendions donc à ce que cet algorithme d'affichage soit très rapide. En fait le pipeline de rendu des cartes graphiques actuelles n'est pas optimisé pour la rasterisation des primitives 2D et ne tire pas parti du fait que les lignes à afficher soient horizontales, le coût d'affichage d'une ligne est donc du même ordre que celui de la rasterisation d'un triangle. D'autre part le nombre de lignes à afficher est élevé et la longueur d'une ligne à l'écran est parfois inférieure à un pixel. Cette technique d'affichage peut donc s'avérer très coûteuse et elle ne peut donc pas être utilisée dans le but que nous nous sommes fixé.

3.4.4 Maillage simplifié avec vertex shader

Le problème des approches précédentes est qu'elles nécessitent l'affichage d'une primitive graphique par échantillon, ce qui peut être très coûteux si l'on veut une qualité d'affichage correcte et donc suffisamment d'échantillons. Nous avons donc cherché à mieux tirer parti des capacités des cartes graphiques actuelles. L'idée est d'afficher une version simplifiée du maillage de la première méthode et de lui plaquer dessus la texture de couleurs avec un paramétrage en coordonnées cylindriques. La version simplifiée est calculée à partir d'une version sous-échantillonnée de l'image de rayons.

Cette approche, malgré ses particularités (géométrie stockée sous forme de texture, texture de couleurs cylindrique) ressemble beaucoup à une simplification de maillage. Nous avons donc pensé pour la rendre plus originale à exploiter les capacités des cartes

graphiques récentes : le maillage peut être crée par un *vertex shader* auquel on passe la texture de rayons simplifiée et qui la lit pour recréer les points du maillage en déformant un maillage cylindrique. Ensuite un *fragment shader* très simple se charge d'aller chercher la couleur dans la texture de couleurs aux bonnes coordonnées en transformant en coordonnées cylindriques la position cartésienne du point traité. Ainsi tout est mémorisé sur la carte graphique et on a juste à passer à la carte une grille de points permettant au shader de retrouver pour chaque point l'échantillon correspondant (par exemple un point de coordonnées $(i, j, 0)$ pour l'échantillon (i, j)). Le principal intérêt est que cette grille est la même pour tous les objets (elle est même implicite : on peut la construire sans l'avoir mémorisée).

Bien que les cartes graphiques actuelles supportent théoriquement les accès de textures dans un *vertex shader*, ces accès sont en fait encore très lents (vitesses d'affichage de l'ordre de quelques images par secondes) et leur utilisation pour cette application n'est pas envisageable. Nous n'avons donc pas poussé plus loin nos recherches dans cette direction.

Les méthodes d'affichage que nous venons de présenter correspondent à différentes approches tentant d'utiliser au mieux le pipeline graphique fourni par les cartes actuelles, pour l'affichage d'un objet représenté par une texture cylindrique. A cause des spécificités de ce pipeline (particulièrent optimisé pour l'affichage de triangles), aucune de ces méthodes ne donne de résultat probant. Nous avons donc adopté une autre approche, à base de déformation de textures.

3.5 Déformation de texture

Comme nos objets sont modélisés uniquement par des images, il parraissait naturel de chercher s'il est possible de déformer ces images en fonction du point de vue pour n'avoir ensuite qu'un affichage de textures à effecter. Nous allons d'abord voir comment la texture de couleurs peut être déformée par un algorithme executé par le CPU puis envoyée à la carte graphique pour être affichée. Nous verrons ensuite quels sont les problèmes qui se posent pour transposer cet algorithme au GPU et une solution utilisant un précalcul.

3.5.1 Warping direct (CPU)

Une idée importante de [OBM00] est de déformer une texture en fonction du point de vue puis d'afficher cette texture à l'écran. Cette déformation 2D peut être décomposée en une succession de deux déformations 1D ce qui rend cette technique praticable. Cette idée peut être transposée à notre modélisation car sous certaines hypothèses la reconstruction de l'objet correspond à une simple déformation 1D de la texture cylindrique de couleurs.

Les hypothèses à respecter découlent de la nature particulière de notre problème : comme l'objet est de petite taille à l'écran, le secteur angulaire qu'il occupe depuis le centre de projection de la caméra est très faible et sa projection à l'écran approche de

celle obtenue par une caméra orthographique de paramètres bien choisis. Une translation de la caméra orthographique revient à une translation de l'image à l'écran, on peut donc imposer sans perte de généralité que l'origine du repère de l'objet se projette au centre de l'ecran et que ce centre a pour coordonnées $(0,0)$.

En raison des restrictions sur la direction de vue de la caméra, les tranches de l'objet se projettent horizontalement à l'écran. On peut donc traiter les tranches de l'objet indépendamment et afficher à l'écran la succession de lignes obtenues apres transformation 1D de chaque ligne de la texture cylindrique. Comme le nombre de lignes projetées à l'écran n'est pas forcément égal à la hauteur en pixels de l'objet, il apparait un problème de sur-échantillonnage ou de sous-échantillonnage qui est résolu en appliquant une technique de mip-mapping sur la texture cylindrique, permettant de dessiner correctement toutes les lignes de l'objet à l'écran tout en économisant lorsque c'est possible.

On se ramène donc à l'étude de la transformation 1D nécessaire pour afficher une tranche horizontale de l'objet. Nous étudions d'abord le cas simplifié (rayons positifs ou indéfinis) puis le cas général.

Cas simple

Fixons d'abord quelques notations : L_x est la largeur de la texture cylindrique (comptées en nombre de texels donc d'échantillons). L'angle entre deux rayons lumineux d'échantillonnage successifs est $d\alpha = \frac{2\pi}{L_x}$. On appele $\alpha_i = i \times d\alpha$ l'angle du rayon lumineux correspondant à l'échantillon i. Ce rayon est dirigé par le vecteur $\vec{d_i} = (\cos\alpha_i, \sin\alpha_i)$. On applele α l'angle que fait la caméra par rapport au repère de l'objet. On suppose que α est un multiple de $d\alpha$, c'est à dire que le rayon lumineux central de la caméra est aligné avec un rayon d'échantillonnage (cela revient à discrétiser les angles sous lesquels peut être vu l'objet ce qui n'est pas trop restrictif si L_x est suffisamment élevé). On appelle i_0 l'indice de l'échantillon aligné avec la caméra, tel que $\alpha = \alpha_{i_0}$. Tous les calculs sur les indices radiaux se font implicitement modulo L_x. On pose $i_1 = i_0 + \frac{L_x}{2}$, l'indice de l'échantillon opposé à la caméra.

Remarquons que l'ensemble des échantillons radiaux de la tranche peut être séparé par la droite $d = (O, \vec{d_{i_1}})$ en deux parties[1] indépendantes : la partie à droite de $\vec{d_{i_1}}$ qui correspond à l'intervalle $[\![i_0, i_1]\!]$ et la partie à gauche de $\vec{d_{i_1}}$ qui correspond à l'intervalle $[\![i_1, i_1 + \frac{L_x}{2}]\!]$. Comme la droite d est orthogonale à l'écran de la caméra, aucune partie de la zone droite ne peut cacher de partie de la zone gauche et réciproquement. On peut donc dessiner ces deux parties indépendamment et dans un ordre quelconque. L'affichage des deux parties est symétrique, on se restreint donc à l'étude de la partie droite.

Un échantillon i se projette à l'écran en un point d'abscisse $x_i = r_i \sin\alpha_{i-i_0}$, où r_i est le rayon l'échantillon i. Comme deux échantillons successifs i et $i+1$ peuvent se projetter à des points x_i et x_{i+1} distants de plusieurs pixels à l'écran, il faut remplir les pixels entre eux en interpolant les couleurs des deux échantillons sur le segment horizontal $\overline{x_i x_{i+1}}$.

[1] Ce n'est pas une partition à proprement parler car les échantillons i_0 et i_1 sont communs aux deux intervalles.

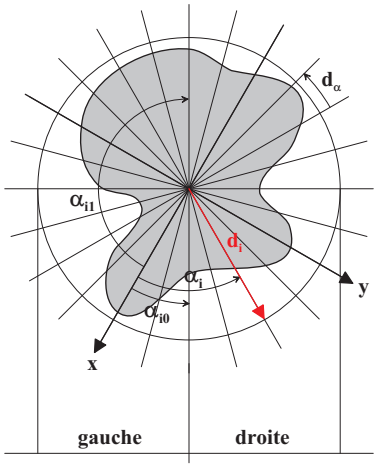


FIG. 3.12 – Notations pour l'algorithme de warping

Nous noterons s_i le segment $\overline{x_i x_{i+1}}$. Ce segment est orienté : l'intérieur de l'objet se trouve à gauche du vecteur $p_i\overrightarrow{p_{i+1}}$ (p_i étant la position 3D de l'échantillon i), donc sa projection s_i à l'écran n'est visible que si $x_i \leqslant x_{i+1}$. Par la suite nous dirons que le segment s_i est *caché* si $x_i > x_{i+1}$ et ce segment ne sera alors pas dessiné.

Une remarque importante concernant les occlusions et les parties cachées est que les segments entre échantillons peuvent être affichés séquentiellement en partant du fond (échantillon i_1) et en se rapprochant de la caméra jusqu'à i_0. En effet, supposons que l'on ait déja dessiné les segments $s_{i_1-1}, s_{i_1-2}, \ldots, s_i$. Ces segments se trouvent dans le demi-plan à gauche de d_i donc *derrière* la droite $(O, \vec{d_i})$ par rapport à la caméra. Ceci implique qu'aucun des segments déja dessinés ne peut cacher le segment s_{i-1}, on peut donc bien dessiner ce segment *après* les segments déja dessinés. Par un raisonnement direct de récurrence, ceci justifie l'ordre de parcours du fond vers l'avant.

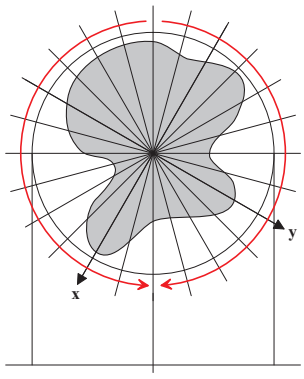


FIG. 3.13 – Warping direct : parcours de l'arrière vers l'avant

Algorithme de warping direct : de l'arrière vers l'avant

$x_{old} \leftarrow 0$ *# abscisse du dernier point*
$c_{old} \leftarrow$ couleur(i_1) *# couleur du dernier point*
$d_{old} \leftarrow$ defini(i_1) *# vrai $\Leftrightarrow$ dernier point défini*
pour i de 1 à $\frac{L_x}{2}$
    si (defini(i_1+i, j))
        $x_{new} \leftarrow$ rayon$(i_1-i) \times \sin(i.d\alpha)$
        $c_{new} \leftarrow$ couleur(i_1-i)
        si (d_{old} et $x_{new} > x_{old}$)
            DessinerSegment(x_{old}, x_{new}, c_{old}, c_{new})
        $x_{old} \leftarrow x_{new}$
        $c_{old} \leftarrow c_{new}$
        $d_{old} \leftarrow$ vrai
    sinon
        $d_{old} \leftarrow$ faux
procéder de même pour la partie gauche

On remarque qu'avec cet ordre de parcours il est possible de dessiner par dessus un segment déja dessiné dans le cas où une partie de l'objet en cache une autre. Il est possible d'éviter ces dessins inutiles en parcourant cette fois les échantillons de l'avant vers l'arrière, à condition que l'objet n'ait ni trous ni parties transparentes ou semi-transparentes. Supposons donc que l'on ait déja dessiné les segments $s_{i_0} \dots s_i$. Soit $x_{max} = \max_{k \in [\![i_0, i+1]\!]} x_k$ le pixel le plus à droite de l'écran que l'on ait dessiné. Chaque pixel de l'intervalle $[0, x_{max}]$ est recouvert par un des segments déja dessinés. Comme on l'a vu précédemment, aucun segment restant à dessiner ne peut cacher de segment déja dessiné, donc aucun segment à dessiner ne peut se projetter dans l'intervalle $[0, x_{max}]$. Il suffit donc de ne dessiner que les segments s'ils se projettent plus à droite que x_{max}, en maintenant x_{max} à jour à chaque segment dessiné.

Cet algorithme pourrait être étendu aux objets avec transparence ou avec trous en maintenant non plus l'information x_{max} mais une opacité pour chaque pixel de l'image, initialisée à 0 et actualisée à chaque fois qu'un nouveau segment est dessiné. La couleur d'un pixel qui devient complètement opaque ne peut plus être modifiée.

Algorithme de warping direct : de l'avant vers l'arrière

$x_{max} \leftarrow 0$ *# abscisse maximale dessinée*
$c_{old} \leftarrow$ couleur(i_0) *# couleur du dernier point*
pour i de 1 à $\frac{L_x}{2}$
    $x_{new} \leftarrow$ rayon$(i_0+i) \times \sin(i.d\alpha)$
    $c_{new} \leftarrow$ couleur(i_0+i)
    si ($x_{new} > x_{max}$)
        DessinerSegment(x_{max}, x_{new}, c_{old}, c_{new})
        $x_{max} \leftarrow x_{new}$
    $c_{old} \leftarrow c_{new}$
procéder de même pour la partie gauche

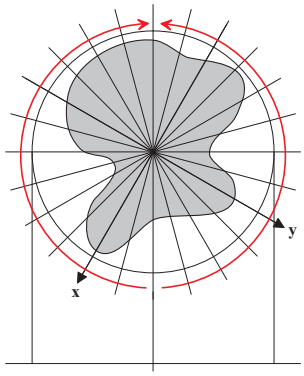


FIG. 3.14 – Warping direct : parcours de l'avant vers l'arrière

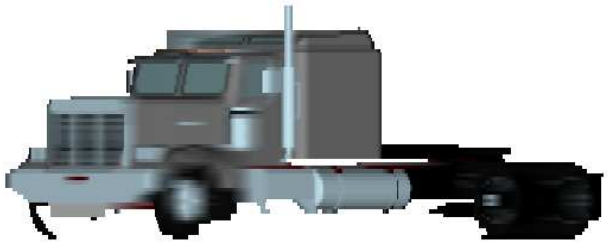


FIG. 3.15 – Image d'un camion obtenue après déformation de sa texture cylindrique

Cas général

La possibilité pour les rayons d'être négatifs ajoute quelques complications. Comme nous venons de le voir, la paramétrisation cylindrique de l'objet implique un ordre d'affichage sur les secteurs angulaires $S_i =< \alpha_i, \alpha_{i+1} >$. Le problème qui apparait avec les rayons négatifs est qu'un échantillon i de rayon négatif correspond à un point dans le secteur angulaire $S_{i+\frac{L_x}{2}}$. Pour palier à ce problème on adopte une approche différente : on ne parcourt plus les échantillons mais les secteurs angulaires et pour chaque secteur angulaire on dessine le ou les segments qu'il contient. Le secteur angulaire $S_{i+\frac{L_x}{2}}$ contient le segment défini par les échantillons i et $i+1$ mais peut aussi contenir un segment défini par les échantillons $i+\frac{L_x}{2}$ et $i+1+\frac{L_x}{2}$. Comme les échantillons peuvent aussi être indéfinis, chaque demi-droite d'échantillonnage peut contenir 0, 1 ou 2 échantillons et on a donc les 9 cas de figures suivants pour un secteur angulaire compris entre deux demis-droites successives :

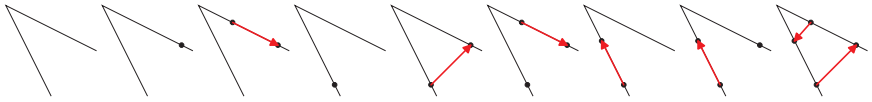


FIG. 3.16 – Neuf cas possibles pour un secteur angulaire

Il suffit donc de parcourir les secteurs angulaires de l'arrière vers l'avant en affichant le ou les segments correspondant au cas courant. Le seul cas qui peut poser problème est le cas avec deux segments (le cas à droite de la figure 3.16) : la question de l'ordre d'affichage entre ces deux segments peut à priori se poser. En fait on peut remarquer que jamais un des deux segments ne peut en cacher un autre quand on prend en compte l'orientation des segments et le fait qu'il est inutile d'afficher un segment orienté négativement (dans le cas où les deux segments sont visibles (troisième cas de la figure 3.17), ils ne peuvent pas se superposer).

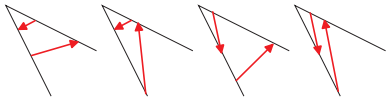


FIG. 3.17 – Quatre cas de visibilité pour deux segments (pour une caméra regardant de bas en haut)

On peut aussi parcourir les secteurs angulaires de l'avant vers l'arrière en utilisant la technique de l'opacité des pixels mentionnée pour le cas simple.

Comme on effectue un calcul exact de la déformation de la texture en fonction du point de vue, le résultat visuel est le même que pour la solution de référence du maillage à base de triangles. Cependant, la texture déformée étant calulée par le CPU puis transférée à la carte graphique, la fréquence d'affichage obtenue n'est pas suffisante pour notre problème. Nous cherchons donc maintenant un moyen de faire effectuer directement ces calculs par la carte graphique.

3.5.2 Warping inverse (GPU)

Le problème de l'approche du warping direct est qu'elle est difficilement transposable sur la carte graphique : on est obligé de faire les calculs en CPU et de passer l'image calculée à la carte graphique à chaque affichage ce qui est assez coûteux. La carte graphique est composée de plusieurs *pixel pipelines* qui executent un *fragment program* en parallèle sur les pixels à afficher. Il faut donc un programme qui permette d'afficher les pixels indépendamment et dans un ordre aléatoire, ce qui n'est pas le cas de notre algorithme qui affiche les pixels séquentiellement, dans un ordre bien défini.

Pour rendre l'algorithme parallélisable, il faut que chaque pixel puisse trouver dans la texture cylindrique à quelle abscisse chercher sa couleur. Pour ce faire l'idée la plus directe est de parcourir la texture de rayons de l'avant vers l'arrière jusqu'à trouver le segment qui correspond au pixel traité. Nous donnons l'algorithme dans le cas simple sans trous, qui se généralise sans difficulté :

Algorithme de warping inverse

$x_{pix} \leftarrow$ abscisse du pixel traité
$x \leftarrow 0$
$i \leftarrow \left\lfloor \frac{L_x}{2\pi} \arcsin(\frac{x_{pix}}{R}) \right\rfloor$ *# entrée dans le cylindre englobant*
tant que $(x < x_{pix}$ et $i < \frac{L_x}{2})$
    $i \leftarrow i+1$
    $x \leftarrow \text{rayon}(i_0+i) \times \sin(i.d\alpha)$
$c_{frag} \leftarrow \text{couleur}(i_0+i)$

Cet algorithme est donné pour le coté droit de l'objet, il se transpose facilement au coté gauche. Pour un meilleur affichage, la couleur c_{frag} du fragment traité peut être calculée comme une interpolation linéaire entre les échantillons (i_0+i-1) et (i_0+i) en fonction de la position de x à l'intérieur du segment correspondant.

Il est clair qu'en raison du nombre d'itérations à effectuer pour chaque pixel ($\frac{L_x}{2}$ au maximum), cette approche n'est pas praticable quand le nombre d'échantillons est élevé (on peut par exemple être amené à faire 128 itérations pour une texture cylindrique de 256 texels de largeur). C'est pourquoi nous avons cherché à utiliser un précalcul pour accélérer ces itérations.

Précalculs

Le problème est de retrouver pour un point de coordonnées (x,y) à l'écran, sous un angle de vue α, les coordonnées (i,j) où aller chercher la couleur dans la texture cylindrique. L'ordonnée j se déduit d'une simple transformation linéaire de l'ordonnée y à l'écran, la difficulté réside donc dans le fait de trouver l'abscisse i. La solution brutale consiste à mémoriser pour chaque triplet (x,y,α) la valeur du décalage i correspondant, c'est à dire une fonction à trois dimensions. Cette information est évidemment trop lourde à mémoriser et nous avons donc cherché à la compresser en en calculant une approximation linéaire basée sur peu d'échantillons.

L'analyse des décalages (voir figure 3.18) montre qu'ils évoluent relativement continuement quand x, y ou α varient. L'idée est donc d'approcher la fonction échantillonnée par une fonction affine par morceaux, optimisée pour qu'elle s'approche le plus

possible de la fonction initiale.

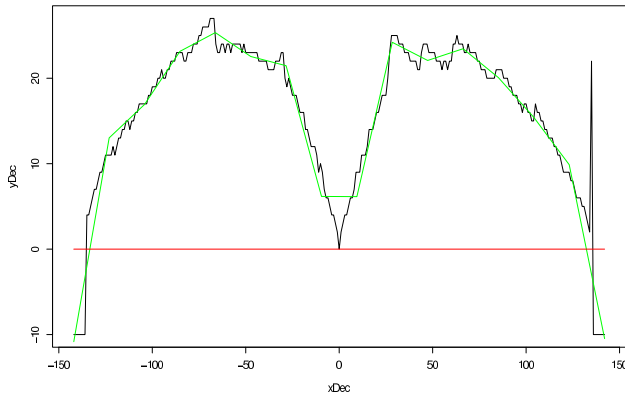


FIG. 3.18 – Décalages pour une hauteur donnée et un angle de vue donné (en noir) et approximation linéaire (en vert)

Comme nous le montrons en annexe (voir partie 6), ce problème peut être résolu par une méthode de moindres carrés et ainsi ramené à l'inversion d'une matrice creuse de dimension $A_x \times A_y \times A_\alpha$, ayant au plus 27 coefficients non nuls par ligne, où A_x, A_y et A_α sont les nombres d'échantillons en largeur, en hauteur et radiaux sur lesquels s'appuient les fonctions affines.

Comme certains échantillons peuvent être indéfinis (rayons lumineux n'intersectant pas l'objet), le calcul des coefficients du système doit être adapté pour ne pas prendre en compte la contribution de ces échantillons sans signification. Le système peut alors être sous-déterminé (la matrice peut ne pas être inversible) mais il admet toujours au moins une solution car la problème de minimisation des erreurs d'approximation en a une. Il faut donc utiliser une méthode de résolution de système qui supporte les matrices non inversibles, comme la décomposition en valeurs singulières. En raison de la taille de notre système, cette méthode présente un temps de calcul prohibitif et nous avons utilisé un algorithme de *gradient biconjugué préconditionné* optimisé pour la résolution de système creux.

FIG. 3.19 – Déformation exacte et approximation

Expérimentalement, une résolution de $A_x = 15$, $A_y = 8$ et $A_\alpha = 16$ donne des

résultats acceptables. Un tel précalcul nécessite de mémoriser presque 2000 coefficients ce qui est tout à fait négligeable devant la place occuppée par la texture cylindrique.

Un problème d'ordre technique se pose lors de l'implémentation de cette méthode sur GPU : bien que les décalages soient bornés, les valeurs calculées par l'algorithme d'optimisation peuvent s'étaler sur un intervalle beaucoup plus large (une pente forte de la courbe peut générer un pic de grande hauteur dans la fonction approximée). Leur stockage dans une texture pose donc un problème lié à la faible dynamique utilisée pour les composantes de couleur (8 bits généralement). Pour surmonter ce problème il faudrait utiliser davantage de mémoire pour les stocker à une résolution plus élevée.

3.6 Conclusion

Nous avons vu la première approche du problème étudié. La modélisation d'objets par des textures cylindriques a plusieurs intérêts. Le premier est que le coût mémoire est faible : l'objet est représenté par deux images (couleurs et normales+rayons) de taille relativement petite, relativement indépendamment de sa complexité. Grâce à la paramétrisation en coordonnées cylindriques, cette représentation est adaptée à la reconstruction de vues obtenues en tournant autour de l'objet. De plus une classe assez large d'objets peut être représentée correctement par ce modèle. Enfin l'utilisation d'une représentation à base d'images permet de traiter le problème de l'aliasing en appliquant des algorithmes de filtrage et de mip-mapping aux textures représentant l'objet.

Cependant cette approche présente aussi plusieurs limites. Parmi tous les algorithmes proposés, aucun n'est entièrement satisfaisant du point de vue de la rapidité d'affichage. De nombreux objets se prêtent mal à la paramétrisation cylindrique, comme les objets à trous. L'échantillonnage radial implique des différences de densité d'échantillonnage à la surface de l'objet : les points près de l'axe sont plus sérrés que les points loins de l'axe. Le problème du filtrage horizontal est dur à traiter à cause de cet échantillonnage radial, la technique du mip-mapping ne suffit pas à résoudre les problèmes d'aliasing qui en résultent.

Nous avons donc une méthode qui demande très peu de mémoire pour représenter un objet mais demande trop de calculs pour être utilisable pour notre problème. Nous avons donc cherché une modélisation qui permette de diminuer les temps de calcul d'affichage, quitte à utiliser plus de mémoire. C'est l'objet de la partie suivante qui présente une nouvelle représentation et les méthodes d'affichage associées.

Chapitre 4

Images avec parallaxe

4.1 Idée de départ

Revenons à l'idée initiale qui était de prendre N vues en tournant autour de l'objet et de les combiner pour synthétiser une nouvelle vue quelconque. Nous noterons par la suite I_k l'image prise sous l'angle de vue $\alpha_k = \frac{2k\pi}{N}$. Connaissant l'angle de vue α sous lequel l'objet doit être affiché, on peut retrouver les deux images I_{k_1} et I_{k_2} dont les angles de vue sont les deux plus proches de α :

$$\alpha_{k_1} \leqslant \alpha < \alpha_{k_2} \quad \text{et} \quad k_2 = (k_1 + 1) \bmod N$$

Nous avons cherché comment combiner les images I_{k_1} et I_{k_2} pour synthétiser la nouvelle image I.

4.1.1 Mélange des images

L'idée la plus simple est d'utiliser ces images comme des *billboards*. Pour que la transition entre ces deux images soit fluide on en effectue un mélange (*blending*) en fonction de l'angle de vue :

$$I = (1-\beta)I_{k_1} + \beta I_{k_2} \quad \text{avec} \quad \beta = \frac{\alpha - \alpha_{k_1}}{\alpha_{k_2} - \alpha_{k_1}}$$

Après implémentation de cet algorithme, nous avons pu constater plusieurs effets visuellement génants :

- aux endroits où les deux images sont trop dissemblables se produisent des effets de fantômes, c'est à dire des parties d'images qui apparaissent ou qui disparaissent.
- lors d'un mouvement de rotation de l'objet, le mélange entre deux images successives ne suffit pas à restituer un effet de parallaxe correct.

Expérimentalement, il faut au moins 64 images de l'objet pour que ces effets s'estompent. Le coût mémoire d'une telle solution étant prohibitif (2 Mo pour des images 64×64 en tenant compte du stockage des images de normales), nous avons cherché à

ajouter un effet de parallaxe aux images en utilisant d'autres informations en plus des couleurs.

FIG. 4.1 – Effets de fantômes obtenus avec le mélange des images

4.1.2 Ajout de parallaxe

Lorsque l'on tourne autour d'un objet, l'impression de parallaxe est produite par la translation horizontale des points de l'objet à l'écran (on utilise toujours une caméra orthographique).

Supposons que l'on a pris une image de l'objet puis que l'on a tourné autour de l'axe de rotation d'un angle θ. Pour redessiner cette image en restituant l'effet de parallaxe correct il faut pour chaque pixel pouvoir estimer de combien le translater horizontalement. Soit un pixel de coordonnées (x_t, y_t) de l'image capturée, notons $x_e(\theta)$ la nouvelle abscisse de ce pixel à l'écran après la rotation d'angle θ, que l'on peut calculer analytiquement en connaissant la profondeur z_t du pixel au moment de la capture de l'image :

$$x_e = x_t \cos\theta - z_t \sin\theta \tag{4.1}$$

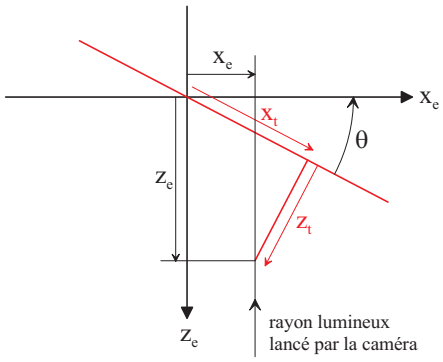


FIG. 4.2 – Calcul du décalage d'un pixel après rotation

En fait l'idée initiale était de mémoriser pour chaque pixel sa vitesse instantanée de translation quand l'objet tourne pour pouvoir approcher la translation $x_e(\theta)$ par une approximation au premier ordre en θ :

$$x_e(\theta) \simeq \tilde{x}_e(\theta) \quad \text{avec} \quad \tilde{x}_e(\theta) = x_e(0) + \frac{\partial x_e}{\partial \theta}(0) \times \theta = x_t - z_t \times \theta$$

On remarque donc qu'en fait la vitesse instantanée de translation $\frac{\partial x_e}{\partial \theta}(0)$ n'est rien d'autre que $-z_t$, qu'il faut donc mémoriser pour chaque pixel. Comme la connaissance de cette valeur permet de reconstituer la position dans l'espace de chaque pixel, on peut en fait effectuer un calcul exact de la translation $x_e(\theta)$.

Nous associons donc à chaque pixel de coordonnées (x_t, y_t) sa profondeur z_t, ce qui revient à stocker un champ de hauteurs ou *height field* de l'objet. Voyons maintenant comment afficher un objet en utilisant cette nouvelle information.

4.2 Modélisation

Dans toute la suite nous ne considérons qu'une tranche de l'objet de hauteur y_t, autrement dit on ne considère que les lignes de pixels d'ordonnée y_t des images capturées. On note θ_1 et θ_2 les angles entre le point de vue courant et les points de vues utilisés pour la capture des images I_{k_1} et I_{k_2} :

$$\theta_1 = \alpha - \alpha_{k_1} \geqslant 0 \quad \text{et} \quad \theta_2 = \alpha - \alpha_{k_2} < 0$$

Par la suite, quand ce ne sera pas précisé, on supposera que l'image considérée est I_{k_1} et donc que l'angle θ entre la vue et l'image est θ_1.

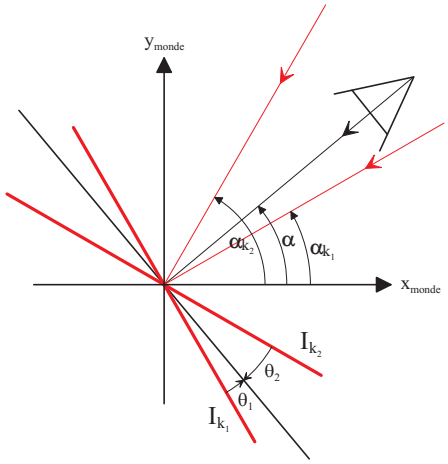


FIG. 4.3 – Vues utilisées pour l'affichage

Comme on utilise à chaque fois uniquement les deux images dont les points de vue sont les plus proches de celui de la caméra, l'angle de vue θ sous lequel est vue une image est restreint à l'intervalle

$$\Theta = [-\Delta\theta, \Delta\theta] \quad \text{avec} \quad \Delta\theta = \frac{2\pi}{N}$$

Nous introduisons deux repéres : le repère (O, x_e, y_e, z_e) qui tourne avec la caméra, les axes $\vec{Ox_e}$ et $\vec{Oy_e}$ étant alignés avec l'écran et l'axe $\vec{Oz_e}$ dirigé vers la caméra. Le

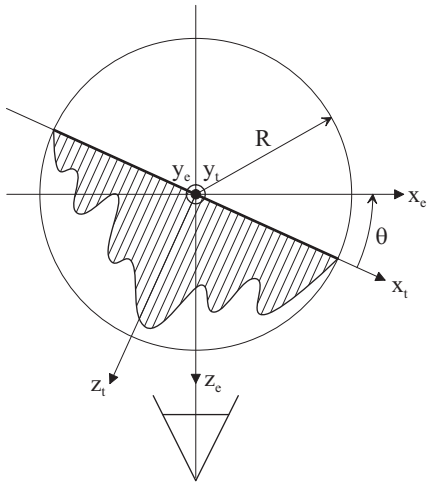


FIG. 4.4 – Repère caméra, champ de hauteurs et repère associé

repère (O, x_t, y_t, z_t) attaché à l'image traitée (quand ça ne sera pas spécifé on supposera que l'image traitée est I_{k_1}), confondu avec R_e quand $\theta = 0$.

Comme pour les textures cylindriques, nous plaçons l'objet dans un cylindre englobant de rayon R (ce qui permet d'un point de vue pratique de stocker les champs de hauteurs sous une forme renormalisée). On impose que les bords gauche et droit des images capturées soient sur le cylindre englobant (quitte à rajouter des pixels indéfinis aux bords des images). La distance selon x_t entre deux pixels voisins d'une image dans le repère (O, x_t, y_t, z_t) associé est donc

$$\delta x = \frac{2R}{L_x}$$

Le champ de hauteurs $z_t(x_t)$ peut être interprété soit comme une fonction en escaliers, correspondant à une interpolation de plus proche voisin, soit une fonction affine par morceaux, correspondant à une interpolation linéaire entre les hauteurs. Selon les algorithmes considérés par la suite, il sera plus simple de se placer dans un cas ou dans l'autre mais les calculs et les résultats visuels restent sensiblement les mêmes.

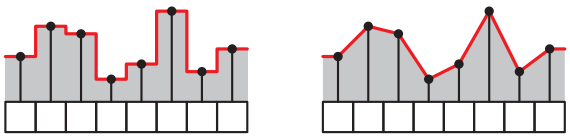


FIG. 4.5 – Interpolation des hauteurs : plus proche voisin et linéaire

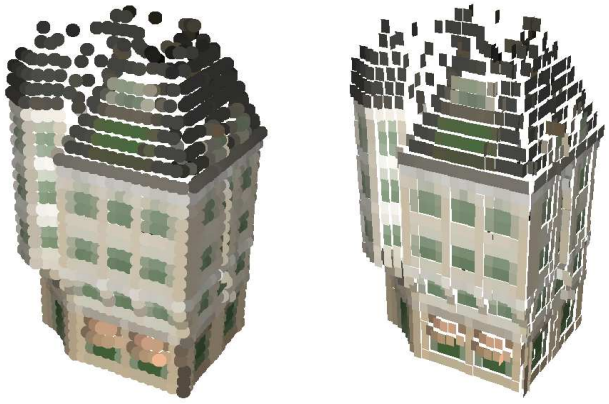
De la même manière que pour les textures cylindriques, on capture une normale par pixel pour chaque image, ce qui permet de rééclairer l'objet en fonction du point de vue et de la position de la lumière avec un calcul d'éclairage diffus.

Voyons maintenant la quantité de mémoire nécessaire pour stocker un objet avec cette représentation : la couleur et l'opacité peuvent être mémorisées sur les 4 canaux d'une image. Les trois composantes de la normale et la profondeur du pixel peuvent être stockés sur les 4 canaux d'une deuxième image. En utilisant une précison standard, le canal d'un pixel d'une image est stocké sur 1 octet, il faut donc $8NL_xL_y$ octets pour représenter l'objet. Dans la pratique nous avons généralement utilisé $N = 8$ images de taille 64×64 pixels, la quantité de mémoire utilisée dans ce cas est donc 256 Ko, ce qui est 8 fois moins élevé qu'avec l'approche consistant à mémoriser 64 vues de l'objet. Pour ce calcul de coût mémoire nous avons supposé que les textures de couleurs et de normales étaient de même taille, mais en fait selon les besoins ces tailles peuvent être différentes (en utilisant par exemple une texture de normales avec une résolution plus élevé si on veut accorder plus d'importance aux détails). De plus on a besoin d'un nombre N plus ou moins élevé de vues selon le type d'objet représenté (par exemple $N = 4$ textures sont adaptées pour représenter un bâtiment).

4.3 Affichage à base de points

4.3.1 Affichage de points

La solution la plus simple pour l'affichage des images en tenant compte de l'information de profondeur est d'afficher un point par pixel par *splatting* en tenant compte du décalage induit par sa profondeur. Comme le montre le schéma ci-dessous, le fait d'utiliser deux vues proches de l'objet encadrant le point de vue de la caméra permet pour la plupart des objets d'éviter le problème des trous entre échantillons rencontrés avec une seule vue.



(a) (b)

FIG. 4.6 – Affichage (a) à base de points (b) à base de quads

Pour mieux prendre en compte l'orientation des pixels décalés et leur taille il est possible d'utiliser des faces rectangulaires (*quads*) à la place des points (cette technique n'a d'intérêt que si chaque pixel de l'image se projette sur un nombre de pixels à l'écran suffisamment élevé).

L'intérêt de cette approche est sa simplicité et la rapidité d'affichage des points ou des quads. L'incnvénient est que le nombre de primitives à afficher est égal au nombre de pixels de l'image source, ce qui peut s'avérer coûteux.

Nous avons donc cherché une approche demandant l'affichage de moins de primitives géométriques.

4.3.2 Maillages simplifiés (imposteurs)

L'idée pour éviter de dessiner une primitive par pixel est de calculer pour chaque image source un imposteur basé sur un maillage simplifié (à partir d'un grille de l'ordre de 16×16 sommets) approchant son height field. Il suffit ensuite d'y plaquer la texture de couleur de l'image et de faire un calcul d'éclairage par pixel en se servant de la carte de normales capturée.

Cette solution est à priori très efficace car complètement adaptée au pipeline de rendu des cartes graphiques actuelles.

Le problème est que les champs de hauteurs capturés sont généralement assez discontinus et sont mal approchés par les maillages calculés, même en utilisant une méthode d'optimisation comme celle décrite en annexe (voir partie 6).

4.4 Déformation de texture

Les algorithmes d'affichage présentés ci-dessus ne sont pas pleinement satisfaisant : le premier resitue la parallaxe exacte mais demande l'affichage de beaucoup de points, le second est efficace mais ne restitue qu'une approximation de la parallaxe. Nous avons donc cherché une méthode d'affichage efficace utilisant l'information de parallaxe mémorisée dans la profondeur de chaque pixel.

Remarquons d'abord l'analogie de notre représentation avec celle utilisée dans les *Relief Texture* [OBM00] dans le cas où $N = 4$: mémoriser 4 vues en tournant autour de l'axe revient à capturer un rendu par face d'un cube englobant l'objet (nous ne capturons pas les vues de dessus et de dessous car on restreint le point de vue à un plan orthogonal à l'axe de rotation de l'objet).

Comme dans [OBM00], pour chaque vue on mémorise une image de couleurs et une carte de profondeurs. Cette similitude nous a fait penser à utiliser un algorithme de déformation de texture pour l'affichage de l'objet. Nous sommes ici dans un cas plus restreint :

- on utilise une projection orthogonale et les images sont alignées avec l'écran, donc les pixels d'une image se déplacent uniquement horizontalement et les lignes se déforment indépendament les unes des autres.
- on utilise les deux images dont les angles de vue encadrent celui de la caméra, donc une image ne peut être vue que sous un ensemble restreint d'angles de vue ($\theta \in \Theta$).

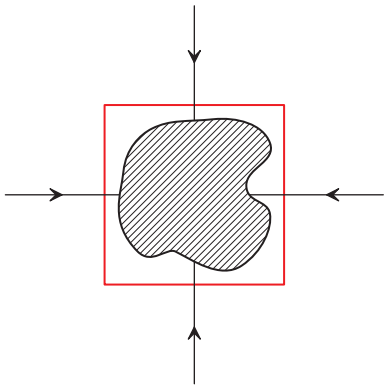


FIG. 4.7 – Analogie avec les *Relief Texture*

– les objets sont destinés à être affichés de loin, on peut donc se permettre des imprécisions d'affichage tant qu'elles ne sont pas génantes visuellement.

De plus on s'autorise à utiliser plus de 4 images pour modéliser l'objet. On peut donc espérer trouver un algorithme spécifique plus efficace que celui proposé dans [OBM00].

4.4.1 Warping direct (CPU)

Pour disposer d'une solution de référence nous avons d'abord étudié la solution la plus directe, qui consiste à créer un nouvelle image en déformant les textures de couleurs sans utiliser les spécificités de la carte graphique. Les deux images I_{k_1} et I_{k_2} sont d'abord déformées séparément puis combinées en tenant compte de certains problèmes qui peuvent se poser.

Déformation d'une image

Nous étudions d'abord la déformation d'une image I_k vue sous un angle θ positif. Comme noté précédemment, les lignes de l'image se déforment de manière indépendante, on se restreint donc à l'étude de la déformation 1D d'une ligne d'ordonnée y_t.

L'orientation du champ de hauteurs implique un ordre de parcours de gauche à droite (c'est à dire avec x_t croissant) pour qu'aucune erreur d'occlusion n'apparaisse. En effet si on prend deux pixels d'abscisses x_1 et x_2 dans l'image originale telles que $x_1 < x_2$, le pixel d'abscisse x_1 ne peut pas cacher celui d'abscisse x_2.

En utilisant la formule de décalage d'un pixel (equation 4.1), l'algorithme pour déformer une ligne de l'image s'écrit très simplement :

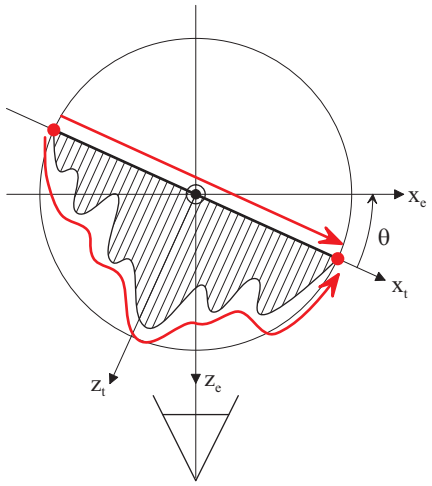


FIG. 4.8 – Warping direct : parcours du champ de hauteurs

Algorithme de warping direct de l'arrière vers l'avant

$x_{old} \leftarrow \infty$
pour i de 0 à $L_x - 1$
    $x_t \leftarrow -R + (i + \frac{1}{2})\delta x$
    $x_e \leftarrow x_t \cos\theta - z(x_t)\sin\theta$
    si $(x_e > x_{old})$
        DessinerSegment($[x_{old}, x_e]$, couleur(x_t))
    $x_{old} \leftarrow x_e$

Le fait d'afficher des segments pour relier deux pixels adjacents permet de résoudre les problèmes de trous qui peuvent apparaitre entre les pixels. Pour des raisons de simplicité d'écriture nous n'avons pas tenu compte des pixels indéfinis : ils peuvent être traités comme cas particuliers en n'affichant pas le segment correspondant ou en affichant des pixels avec une opacité nulle.

Un problème peut se poser lors de l'affichage d'un segment entre deux pixels dont la différence des profondeurs est élevée : dès que θ augmente, ce segment devient très étiré à l'écran ce qui est génant visuellement. Deux pixels ayant une différence de profondeurs au dessus d'un certain seuil doivent donc être considérés comme disjoints et le segment qui les relie ne doit pas être dessiné. Ce seuil de continuité peut être fixé en fonction de la géométrie de l'objet.

Nous n'avons traité pour l'instant que le cas où θ est positif. Le cas $\theta < 0$ est symétrique : l'ordre de visibilité est inversé (les pixels de gauche peuvent cacher ceux de droite), il faut donc parcourir la texture de la droite vers la gauche pour afficher les pixels de l'arrière vers l'avant. Le calcul du déplacement $x_e(\theta)$ reste le même.

Remarquons que comme pour la déformation de textures cylindriques, il est également possible d'appliquer le même algorithme en parcourant les segments dans l'ordre inverse, c'est à dire de l'avant vers l'arrière. Avec cet ordre de parcours, le nouveau segment à dessiner ne doit pas cacher une partie d'un segment déja affiché, il

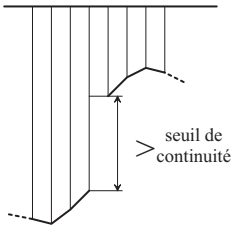
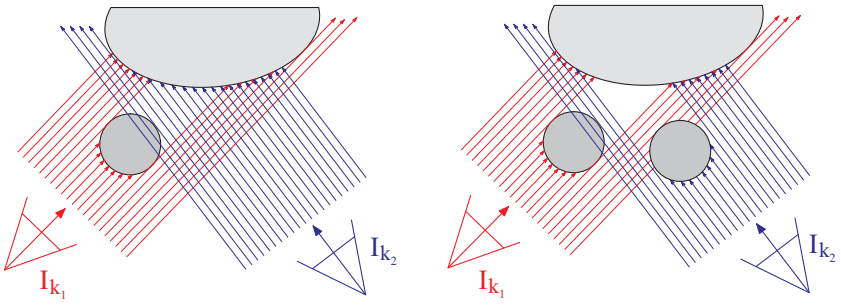


FIG. 4.9 – Illustration du seuil de continuité

faut donc utiliser l'information d'opacité des pixels à l'écran pour gérer correctement les occulsions. Si l'objet n'a pas de pixel transparent et n'a pas de trou[1], on peut se passer du canal d'opacité et mémoriser uniquement la plus petite abscisse atteinte (la plus grande quand $\theta < 0$), actualisée à chaque itération. Cet ordre de parcours permet d'eviter de dessiner des segments qui seront finalement cachés.

Combinaison des images

L'utilisation des deux images dont les angles de vue encadrent celui de la caméra permet de résoudre les problèmes de visibilité liés à l'utilisation d'un seul point de vue : une partie de l'objet cachée pour une des deux vues est généralement visible pour l'autre vue. Il existe des cas où les deux vues ne permettent pas de reconstruire totalement la nouvelle vue mais ils sont relativement rares si on choisit correctement les points de vues capturés et si on prend un nombre d'images N adapté à la topologie de l'objet.



(a) (b)

FIG. 4.10 – Problème de vues incomplètes : (a) la vue de droite complète celle de gauche, (b) une partie de l'objet est cachée pour les deux vues

Se pose alors le problème de la combinaison des deux images déformées pour les

[1]On entend ici par *trou* une suite horizontale de pixels non définis compris entre deux pixels définis. Les pixels non définis du bord de l'image ne posent pas problème.

afficher à l'écran :

- un pixel qui n'est défini dans aucune des deux images correspond soit à un point hors de l'objet, soit à un point de l'objet visible pour aucune des deux vues (on n'a dans ce cas aucune information sur la couleur du point), on n'affiche donc rien à ce pixel.
- un pixel qui est défini uniquement dans une des deux images correspond à une partie visible uniquement par une image, il faut donc lui associer la couleur prise dans cette image.
- un pixel défini dans les deux images est visible par les deux points de vue, les deux couleurs doivent donc théoriquement concorder. Pour rendre la transition la plus fluide possible on fait un mélange de ces deux couleurs en fonction de la proportion d'angle de vue entre la caméra et les images.

(a) (b) (c)

FIG. 4.11 – Combinaison des deux images déformées : (a) image de gauche déformée, (b) image de droite déformée, (c) image combinée

Les résultats obtenus donnent une impression visuelle très satisfaisante avec une nombre d'images relativement faible ($N = 8$ suffit pour la plupart des objets). Maintenant que nous avons validé cette méthode d'affichage par une implémentation software, cherchons à la transposer sur GPU.

4.4.2 Warping inverse (GPU)

L'algorithme que nous venons de présenter effectue une reconstruction exacte de l'objet mais n'est pas efficace car à chaque affichage il faut précalculer l'image déformée puis la passer à la carte graphique, ce qui est relativement coûteux pour notre problème. Pour éviter ces transferts de textures à chaque affichage, il faut passer toutes les textures à la carte graphique au début puis laisser le soin à un *fragment shader* de déformer ces textures en fonction du point de vue. Se présente alors le même problème que pour les textures cylindriques : l'algorithme de warping est itératif et traite les pixels les uns après les autres. Il faut donc trouver un moyen de paralléliser ce calcul.

On procède comme pour les textures cylindriques : pour chaque pixel à l'écran, le fragment shader doit effectuer le calcul de warping *inverse*, c'est à dire parcourir itérativement la texture pour y trouver le pixel original et utiliser sa couleur. L'algorithme s'écrit comme suit :

Algorithme de warping inverse

$x_{pix} \leftarrow$ abscisse du pixel traité
$x_e \leftarrow \infty$
$x_t \leftarrow x_1 + \delta x$
tant que $(x_e > x_{pix}$ et $x_t > x_2)$
    $x_t \leftarrow x_t - \delta x$
    $x_e \leftarrow x_t \cos\theta - z(x_t)\sin\theta$
$c_{frag} \leftarrow$ couleur(x_t)

Le problème est de déterminer les abscisses x_1 et x_2 de début et de fin d'itération. Rappelons que l'objet se trouve dans un cylindre englobant de rayon R. Appelons x_1 et x_2 les abscisses des projections orthogonales sur l'image des intersections avec le cylindre englobant du rayon lumineux que l'on est en train de traiter (x_1 correspondant à la première intersection en partant de la caméra, x_2 à la deuxième). De par la définition du cylindre englobant, aucun pixel d'abscisse hors de l'intervalle $[x_2, x_1]$ ne peut se projeter sur le rayon lumineux. On peut donc restreindre la recherche à cet intervalle.

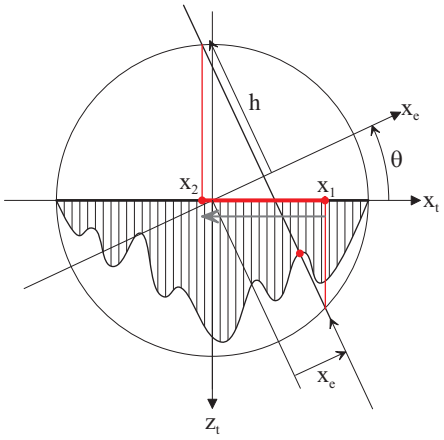


FIG. 4.12 – Calcul des bornes x_1 et x_2 de l'itération

Étude de la complexité

Pour calculer x_1 et x_2 on calcule d'abord la longueur h définie sur le schéma :

$$h = \sqrt{R^2 - {x_e}^2}$$

et on a alors par un raisonnement géométrique direct :

$$\begin{aligned} x_1 &= x_e \cos\theta + h \sin\theta \\ x_2 &= x_e \cos\theta - h \sin\theta \end{aligned}$$

Remarquons que le nombre d'itérations en pire cas vaut :

$$n_{iter}(x_e,\theta) = \frac{|x1-x2|}{\delta x} = \frac{2h|\sin\theta|}{\delta x}$$

Ce nombre est maximal pour $\theta = \Delta\theta$ et $x_e = 0$:

$$n_{iter}^{max} = n_{iter}(0,\Delta\theta) = \frac{2R\sin\Delta\theta}{\delta x} = L_x \sin\Delta\theta$$

On peut aussi calculer le nombre d'itération moyen en pire cas :

$$\begin{aligned}
n_{iter}^{moy} &= \frac{1}{2R}\int_{x_e=-R}^{R} \frac{1}{2\Delta\theta}\int_{\theta=-\Delta\theta}^{\Delta\theta} n_{iter}(x_e,\theta).d\theta.dx_e \\
&= \frac{1}{4R.\Delta\theta}\int_{-R}^{R}\int_{-\Delta\theta}^{\Delta\theta} \frac{2}{\delta x}\sqrt{R^2-{x_e}^2}\,|\sin\theta|\,.d\theta.dx_e \\
&= \frac{2}{R.\Delta\theta.\delta x}\int_0^{\Delta\theta}\sin\theta.d\theta \times \int_0^R \sqrt{R^2-{x_e}^2}.dx_e \\
&= \frac{2}{R.\Delta\theta.\delta x}\times(1-\cos\Delta\theta)\times\frac{\pi R^2}{4} \\
&= \frac{\pi(1-\cos\Delta\theta)}{4\Delta\theta}L_x
\end{aligned}$$

Pour $N = 8$, on a $\Delta\theta = \frac{\pi}{4}$, donc

$$n_{iter}^{max} \simeq 0.7L_x \quad \text{et} \quad n_{iter}^{moy} \simeq 0.3L_x$$

ce qui fait un nombre moyen de presque 19 itérations pour une image de 64 pixels de large. Ce nombre d'itérations est à multiplier par deux car il correspond à la déformation d'une seule image. Rappelons que c'est un nombre d'itérations en pire cas, le nombre d'itérations réel est moins élevé. Nous avons mesuré une valeur moyenne de 12.4 itérations sur un modèle de camion et 10.9 sur la théiere. Ceci permet d'obtenir des fréquences d'affichage de plus de 1000 Hz pour un objet échantillonné avec 8 vues affiché sur une surface de 100 pixels de coté (GeForce 6800).

Une itération consiste en deux additions, deux multiplications, un accès texture et deux comparaisons. Nous allons d'abord voir une manière différente d'écrire l'algorithme permettant de réduire une itération à une addition, un accès texture et deux comparaisons. Nous verrons ensuite une méthode pour réduire de manière significative le nombre d'itérations en utilsant un précalcul.

Optimisation des itérations de l'algorithme

La partie principale de chaque itération de l'algorithme est le calcul de x_e, qui consiste à transformer les coordonnées (x_t, z_t) d'un pixel de l'image dans le repère de l'ecran. On effectue donc à chaque itération un changement de repère de l'image vers l'écran. Pour éviter ce changement de repère il suffit de prendre le problème dans l'autre sens : on considére que l'image est fixe et que le rayon lumineux traité est incliné

d'un angle θ. Il suffit de paramétrer le rayon lumineux dans le repère de l'image puis d'effectuer les itérations dans ce repère.

Appelons $\vec{d}$ le vecteur directeur de la droite correspondant au rayon lumineux traité, orienté de la caméra vers l'objet, exprimé dans le repère de l'image, ayant pour abscisse la largeur δx de l'intervalle entre deux pixels de l'image :

$$\vec{d} = (-\delta x, \frac{\delta x}{\tan\theta})$$

Le rayon lumineux entre dans le cylindre englobant au point p_1 d'abscisse x_1 et en ressort au point d'abscisse p_2 :

$$\begin{aligned} p_1 &= (x_e\cos\theta + h\sin\theta, -x_e\sin\theta + h\,cos\theta) \\ p_2 &= (x_e\cos\theta - h\sin\theta, -x_e\sin\theta - h\,cos\theta) \end{aligned}$$

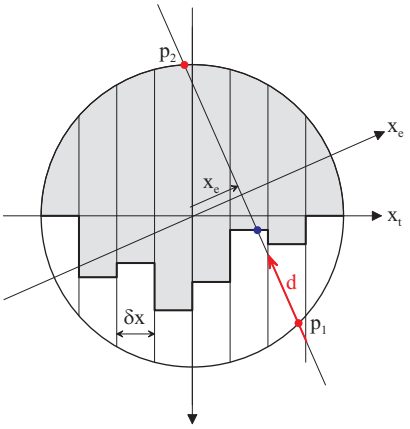


FIG. 4.13 – Optimisation du parcours du champ de hauteurs

L'algorithme consiste donc simplement à prendre un point p ayant pour valeur initiale p_1, qu'on incrémente de $\vec{d}$ à chaque itération. On compare à chaque itération la composante p_z du point p avec la valeur du champ de hauteurs à l'abscisse p_x du point p. On s'arrête dès que le point p passe sous le champ de hauteurs ou quand on dépasse le point x_2.

Algorithme de warping indirect par lancer de rayons

$z \leftarrow -\infty$
$p \leftarrow p_1$
tant que $(p_z > z$ et $p_x > x_2)$
    $p \leftarrow p + d$
    $z \leftarrow z(p_x)$
$c_{frag} \leftarrow$ couleur(p_x)

Par raison de simplicité l'algorithme exposé ne traite pas des cas particuliers comme la présence de trous ou de discontinuités. De même la couleur finale ne doit pas être

prise directement en p_x mais à une abscisse calculée en fonction du point d'intersection exact du rayon avec le champ de hauteurs.

Une itération a donc ainsi été ramenée à une addition, un accès texture et deux comparaisons. Voyons maintenant comment réduire le nombre d'itérations en utilisant un précalcul.

Précalculs de x_1 et x_2

Le nombre d'itérations effectuées dépend des bornes x_1 et x_2 de début et de fin. Ces bornes sont calculées pour chaque pixel en fonction de l'angle de vue. Pour diminuer le nombre d'itérations il faut donc disposer pour chaque pixel d'une information permettant de calculer des bornes les plus proches possibles de l'abscisse x_t corresponant à l'intersection du champ de hauteurs pour un angle de vue donné.

Pour simplifier les notations, nous noterons par la suite Θ^+ l'intervalle $[0,\Delta\theta]$ et Θ^- l'intervalle $[-\Delta\theta,0]$.

La définition exacte des bornes dont nous avons besoin est la suivante :

$$\begin{aligned} &\forall\theta\in\Theta^+, \quad x_1 \geqslant x_t \geqslant x_2 \\ \text{et} \quad &\forall\theta\in\Theta^-, \quad x_1 \leqslant x_t \leqslant x_2 \end{aligned}$$

(on parcourt toujours le champ de hauteurs de x_1 vers x_2 mais le sens de parcours est différent selon le signe de θ).

La technique utilisée dans [Don05] pour accélérer le calcul d'intersection avec un champ de hauteurs est de précalculer dans une texture 3D la distance de chaque point du volume englobant au height field. Cette information est indépendante de l'angle de vue et permet d'accélérer considérablement la recherche d'intersection.

Pour notre application on ne veut mémoriser qu'une information par pixel de l'image. La transposition directe de l'idée donnée dans [Don05] est de mémoriser pour chaque pixel d'abscisse x_e de l'écran la distance minimale de la caméra au champ de hauteurs quand l'angle de vue varie entre $-\Delta\theta$ et $\Delta\theta$, c'est à dire la profondeur minimum des pixels de l'image qui se projettent à l'écran à l'abscisse x_e quand l'image tourne entre $-\Delta\theta$ et $\Delta\theta$. Ceci revient à mémoriser le maximum de la composante selon z_e des pixels balayés par l'ensemble des rayons lumineux ℓ_θ d'abscisse x_e pour chaque angle de vue θ :

$$z_{max} = \max_{\theta\in\Theta} z_e(\theta)$$

Géométriquement, ceci revient à trouver un arc de cercle de centre O et de rayon minimal $r = \sqrt{{x_e}^2 + {z_{max}}^2}$ entourant la partie du champ de hauteurs balayée par l'ensemble des rayons lumineux $\{\ell_\theta \ / \ \theta\in\Theta\}$.

On peut alors calculer pour un angle de vue θ la valeur x_1 correspondant à l'intersection du rayon lumineux ℓ_θ avec l'arc de cercle de rayon r :

$$x_1 = x_e\cos\theta + z_{max}\sin\theta$$

On procède de même pour x_2 : on calcule

$$z_{min} = \min_{\theta\in\Theta} z_e(\theta)$$

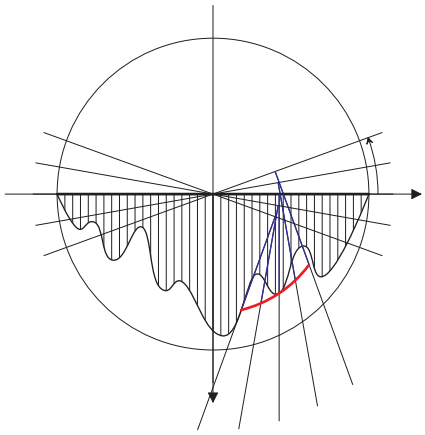


FIG. 4.14 – Arc de cercle approchant localement le champ de hauteurs à partir du calcul de z_{max}

et on obtient la borne

$$x_2 = x_e \cos\theta + z_{min} \sin\theta$$

On peut vérifier que le raisonnement géométrique que venons de faire permet bien de calculer des bornes correctes qui encadrent x_t :

$$\begin{aligned}
\forall \theta \in \Theta^+ \qquad & z_{min} \leqslant z_e(\theta) \leqslant z_{max} \\
\iff \quad & z_{min} \sin\theta \leqslant z_e(\theta)\sin\theta \leqslant z_{max} \sin\theta \\
\iff \quad & x_e \cos\theta + z_{min}\sin\theta \leqslant x_e\cos\theta + z_e(\theta)\sin\theta \leqslant x_e\cos\theta + z_{max}\sin\theta \\
\iff \quad & x_1 \leqslant x_t \leqslant x_2
\end{aligned}$$

et par une démonstration similaire on a aussi

$$\forall \theta \in \Theta^- \quad z_{min} \leqslant z_e(\theta) \leqslant z_{max} \quad \iff \quad x_2 \leqslant x_t \leqslant x_1$$

ce qui correspond à la définiton des bornes x_1 et x_2. Remarquons que le nombre d'itérations en pire cas a été ramené à :

$$n_{iter}(x_e, \theta) = \frac{(z_{max} - z_{min})\,|\sin\theta|}{\delta x}$$

Cette méthode de précalcul nécessite le stockage supplémentaire de z_{min} et z_{max} pour chaque pixel de chaque image, ce qui rajoute 2 canaux aux 8 déja utilisés. Dans la pratique ce précalcul divise en moyenne le nombre d'itérations par 5, le surcoût mémoire nécessaire au stockage de z_{min} et z_{max} est donc justifié.

Généralisation Le fait de prendre les maximum et minimum de $z_e(\theta)$ pour calculer des bornes pour x_t nous a été inspiré par un raisonnement géométrique mais nous allons maintenant voir qu'on peut généraliser cette approche.

Nous avons utilisé le fait que si l'on connait la valeur z_e d'un point d'abscisse x_e on peut calculer x_t, donc par certaines propriétés de la fonction qui à z_e associe x_t, si on possède un encadrement de z_e on peut en calculer un pour x_t.

Supposons maintenant de manière plus générale que l'on dispose d'une fonction f_θ qui permet de calculer $x_t(\theta)$ à partir d'une valeur $v(\theta)$ calculable en x_e pour un angle de vue θ donné :

$$f_\theta(v) = x_t(\theta)$$

On veut alors garantir que les bornes x_1 et x_2 calculées par :

$$\begin{aligned} x_1(\theta) &= f_\theta(v_{max}) \quad \text{avec} \quad v_{max} = \max_{\theta' \in \Theta} v(\theta') \\ x_2(\theta) &= f_\theta(v_{min}) \quad \text{avec} \quad v_{min} = \min_{\theta' \in \Theta} v(\theta') \end{aligned}$$

vérifient la définition donnée ci-dessus. Pour ceci, il faut et il suffit que

$$\begin{aligned} \forall \theta \in \Theta^+, \quad & v_{min} \leqslant v \leqslant v_{max} \Longrightarrow f_\theta(v_{min}) \leqslant f_\theta(v) \leqslant f_\theta(v_{max}) \\ \text{et} \quad \forall \theta \in \Theta^-, \quad & v_{min} \leqslant v \leqslant v_{max} \Longrightarrow f_\theta(v_{min}) \geqslant f_\theta(v) \geqslant f_\theta(v_{max}) \end{aligned}$$

ce qui est équivalent à

$$\begin{aligned} \forall \theta \in \Theta^+, \quad & f_\theta \text{ est croissante} \\ \text{et} \quad \forall \theta \in \Theta^-, \quad & f_\theta \text{ est décroissante} \end{aligned}$$

On peut facilement vérifier que cette propriété est vérifiée pour $v = z_e$ par la fonction

$$f_\theta(z_e) = x_e \cos\theta + z_e \sin\theta$$

utilisée pour dans première approche car c'est une fonction affine en z_e dont le coefficient de la partie linéaire est du signe de θ.

Expérimentalement et géométriquement cette fonction donne de bons résultats pour les objets se rapprochant du cylindre mais on obtient de meilleurs résultats pour les objets ayant des faces planes avec la fonction

$$f_\theta(v) = (2 - \cos\theta)x_e + v \sin\theta$$

qui vérifie aussi la propriété. Nous avons obtenu cette fonction expérimentalement mais nous ne lui avons pas trouvé de signification géométrique directe.

En pratique nous remarquons que la fonction d'approximation f_θ donnant les meilleurs résultats dépend de la topologie de l'objet et n'est pas la même pour tous les pixels. Il est donc intéressant de mémoriser deux couples de valeurs précalculées et de choisir au moment de l'affichage celui qui permet de faire le moins d'itérations. On utilise donc cette fois 4 canaux en plus des 8 déja utilisés. Cette approche permet de diviser en moyenne par 12 le nombre d'itérations effectuées, ce qui implique une fréquence d'affichage de plus de 10 kHz (autrement dit d'afficher plus d'une centaine d'objets à 100 Hz).

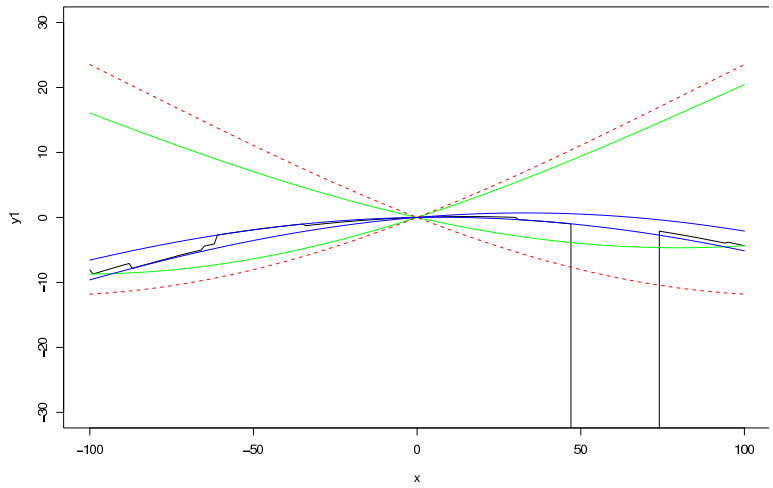


FIG. 4.15 – Différentes bornes calculées : en rouge les bornes correspondant au calcul de l'intersection avec le cylindre englobant, en vert les bornes correspondant au calcul de z_{min} et z_{max} et en bleu les bornes obtenues avec la fonction f_θ déterminée expérimentalement.

4.5 Conclusion

Nous venons d'étudier différents algorithmes permettant d'afficher un objet représenté par un ensemble d'images avec parallaxe. L'affichage à base de points ou de quads malgré sa simplicité donne des résultats intéressants mais est encore trop coûteux pour le but que nous nous sommes fixés. Nous avons vu que l'utilisation d'imposteurs n'est pas adaptée à ce problème.

L'approche la plus satisfaisante est la méthode de déformation de textures par un *fragment shader*, qui en utilisant un précalcul relativement peu cher à stocker permet d'atteindre des fréquences d'affichage très élevées. Cette technique peut même être utilisée pour des objets relativement gros à l'écran en gardant un bon aspect visuel et un affichage rapide. On pourrait aussi envisager de l'étendre à l'affichage d'un objet avec une orientation quelconque en capturant plus de vues de l'objet.

D'autre part la représentation d'objets par des images avec parallaxe se prête bien au problème du filtrage et une technique de mip-mapping peut être utilisée pour obtenir un bon antialiasing.

Chapitre 5

Conclusion

5.1 Méthode de travail

Nous présentons rapidement la démarche que nous avons suivi pour mener à bien ce stage.

Pour chacune des deux contributions (textures cylindriques et images avec parallaxe) nous avons développé deux applications : une application permettant de générer la représentation d'un objet sous forme de fichiers et une application permettant de charger cette représentation et de l'afficher en choisissant une des méthodes implémentées. Pour pouvoir comparer visuellement deux techniques différentes, nous avons séparé l'afficheur en deux fenêtres permettant chacune de choisir indépendamment un algorithme d'affichage.

Pour l'implémentation des algorithmes d'affichage, nous avons toujours suivi la même démarche : implémentation et validation de l'algorithme sur CPU, puis passage éventuel sur GPU.

5.2 Bilan des résultats obtenus

Remarquons tout d'abord que durant cette étude nous avons été confronté à une problématique récurrente, spécifique à l'utilisation de matériel dédié au graphisme : il est toujours difficile d'essayer de contourner le pipeline de rendu utilisé par la carte graphique, notamment certains algorithmes optimaux d'un point de vue théorique peuvent s'avérer être moins efficaces qu'une methode plus simple utilisant les primitives d'affichage fournies par la carte graphique. C'est le cas de nos premiers algorithmes de déformation de textures.

Comparons maintenant les différentes méthodes d'affichage proposées dans ce rapport. Trois aspects sont à considérér : l'aspect visuel, le coût mémoire et la vitesse d'affichage.

Pour les deux représentations, les meilleurs résultats visuels sont obtenus avec les méthodes de déformation de textures. Les tableaux suivants récapitulent les vitesses d'affichage atteintes pour chaque méthode :

méthode	fréquence d'affichage en Hz
original	40
TC triangles	240
TC lignes	400
TC warping	140
TC GPU précalculs	2600
IP mélange	140
IP points	400
IP warping	100
IP GPU	1000

FIG. 5.1 – Fréquences d'affichage obtenues pour l'affichage du camion sur 300 pixels de large

Récapitulons les coûts mémoires de chaque méthode : la représentation en textures cylindriques de dimensions 256×64 occupe 128 Ko. Pour les images avec parallaxe une résolution comparable est obtenue en prenant 8 images de tailles 64×64, ce qui demande 256 Ko de mémoire. Rappelons aussi que la solution consitant à stocker suffisamment de vues pour en faire un simple mélange à l'affichage correspond à 64 images de tailles 64×64, ce qui occupe 2 Mo de mémoire. Les deux représentations proposées sont donc relativement compactes.

La méthode des images avec parallaxe utilisant la déformation de texture en GPU avec précalculs peut donc être considérée comme une solution au problème que nous nous étions fixés : elle est suffisamment rapide pour permettre l'affichage de beaucoup d'objets simultanément, donne de bons résultats visuellement et occupe peu de place en mémoire.

Dans l'optique de l'utilisation de cette technique dans un environnement 3D complet, plusieurs problèmes sont à considérer. Le problème du rééclairage est partiellement pris en compte par notre représentation. Le problème du calcul d'ombres n'a pas été étudié mais on peut au moins générer simplement des ombres dures en effectuant un rendu depuis la source lumineuse. Pour le calcul d'ombres douces, l'objet peut être considéré comme plat en raison de sa distance[1], on génère ainsi simplement une ombre à partir de la silhouette de l'objet. Se pose ensuite le problème des transitions entre version originale et version simplifiée de l'objet. Ces transitions devraient être fluides car les images avec parallaxe restituent l'objet avec une très bonne fidélité.

5.3 Pistes de recherche possibles

L'utilisation de textures ouvre la voie à des techniques de compression. La diminution des tailles des textures permet de s'adapter simplement à un coût de mémoire im-

[1]Ce n'est pas tout à fait vrai car on a supposé que l'objet était loin du point de vue, mais pas forcément de la source lumineuse.

posé, au détriment de la qualité. La cohérence des textures devrait permettre d'utiliser les techniques de compression hardware et ainsi de diviser par 2 ou 4 leurs empreintes mémoire.

Une autre piste de recherche possible serait de voir si la technique du précalcul stockant une information par pixel pour les images avec parallaxes pourrait être transposée aux textures cylindriques. En utilisant des fonctions d'approximations appropriées il doit être possible d'obtenir une optimisation du même ordre que pour les images avec parallaxe.

Pour les images avec parallaxes il reste à comprendre quelles fonctions f_θ permettent d'obtenir les meilleures approximations pour un modèle donné. On peut essayer de créer une base de fonctions, dans laquelle on choisira un optimum lors de la phase de précalculs pour pouvoir s'adapter à des formes d'objets variées.

Toujours pour les images avec parallaxe, on remarque que les informations apportées par les deux images sont fortement corrélées. Il n'est peut être pas nécessaire de déformer les deux textures : la plupart des parties de l'objet peuvent être restituées à partir d'une seule image. On peut envisager de passer d'une image à l'autre dans le *shader* pour en minimiser le coût.

Enfin la levée de la restriction sur le positionnement du point de vue semble difficile pour les textures cylindriques, mais est envisageable pour les images avec parallaxe. On ajouterait des vues prises de dessus et dessous, il faudrait alors utiliser les trois vues les plus proches du point de vue courant.

Chapitre 6

Annexe : approximation affine

Nous exposons ici le calcul basé sur la méthode des moindres carrés permettant de calculer une approximation affine par morceaux d'une fonction réelle à trois variables réelles. Le problème se formule ainsi : on cherche à approcher une fonction

$$f : [0,1]^3 \to \mathbb{R}$$

par une fonction g affine par morceaux définie par interpolation trilinéaire entre les valeurs

$$a_{i,j,k} = g(x_i, y_j, z_k) \quad \text{avec} \begin{cases} x_i = \frac{i}{A_x}, & i \in [\![0, A_x]\!] \\ y_j = \frac{j}{A_y}, & j \in [\![0, A_y]\!] \\ z_k = \frac{k}{A_z}, & k \in [\![0, A_z]\!] \end{cases}$$

La valeur de g en un point $(x,y,z) \in D_{i,j,k} = [x_i, x_{i+1}] \times [y_i, y_{i+1}] \times [z_i, z_{i+1}]$ est

$$g(x,y,z) = \sum_{a=0}^{1} \sum_{b=0}^{1} \sum_{c=0}^{1} \alpha_a.\beta_b.\gamma_c.a_{i+a,j+b,k+c} \quad \text{avec} \begin{cases} \alpha_a &= A_x.|x_{i+1-a} - x| \\ \beta_b &= A_y.|y_{j+1-b} - y| \\ \gamma_c &= A_z.|z_{k+1-c} - z| \end{cases}$$

On dispose d'un ensemble d'échantillons

$$d_{i,j,k} = f(u_i, v_j, w_k) \quad \text{avec} \begin{cases} u_i = \frac{i}{N_x}, & i \in [\![0, N_x - 1]\!], & N_x = n_x \times A_x \\ v_j = \frac{j}{N_y}, & j \in [\![0, N_y - 1]\!], & N_y = n_y \times A_y \\ w_k = \frac{k}{N_z}, & k \in [\![0, N_z - 1]\!], & N_z = n_z \times A_z \end{cases}$$

On appelle $e_{i,j,k}$ la somme des erreurs quadratiques faites sur le domaine $D_{i,j,k}$ en approximant f par g :

$$e_{i,j,k} = \sum_{u=0}^{n_x-1} \sum_{v=0}^{n_y-1} \sum_{w=0}^{n_z-1} \left(d_{i.n_x+u, j.n_y+v, k.n_z+w} - g(u_{i.n_x+u}, v_{j.n_y+v}, w_{k.n_z+w}) \right)^2$$

On cherche à minimiser l'erreur totale

$$e = \sum_{i=0}^{A_x} \sum_{j=0}^{A_y} \sum_{k=0}^{A_z} e_{i,j,k}$$

Pour ce faire on cherche les valeurs des $(a_{i,j,k})$ qui annulent les dérivées partielles

$$e'_{i,j,k} = \frac{\partial e}{\partial a_{i,j,k}} = \sum_{i',j',k'} \frac{\partial e_{i',j',k'}}{\partial a_{i,j,k}}$$

On obtient $A_x \times A_y \times A_z$ équations linéaires faisant intervenir chacune au plus 27 inconnues : l'équation $e'_{i,j,k} = 0$ fait intervenir les inconnues $a_{i+a,j+b,k+c}$ avec $(a,b,c) \in \{-1,0,1\}^3$ (les cas de bords ne font intervenir qu'un sous-ensemble de ces valeurs, correspondant aux indices bien définis). Pour calculer les valeurs des $(a_{i,j,k})$ il suffit donc de résoudre un système linéaire creux de dimension $A_x \times A_y \times A_z$.

Bibliographie

[DDSD03] Xavier Décoret, Frédo Durand, François Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. In *Proceedings of the ACM Siggraph*. ACM Press, 2003.

[Don05] William Donnelly. *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 8 Per-Pixel Displacement Mapping with Distance Functions, pages 123–136. Addison-Wesley, 2005.

[GD98] J. P. Grossman and William J. Dally. Point sample rendering. In *Rendering Techniques*, pages 181–192, 1998.

[GGSC96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *SIGGRAPH '96 : Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 43–54, New York, NY, USA, 1996. ACM Press.

[GH97] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97 : Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[KB04] Leif Kobbelt and Mario Botsch. A survey of point-based techniques in computer graphics. In *Computers & Graphics 28*, pages 801–814, 2004.

[LH96] Marc Levoy and Pat Hanrahan. Light field rendering. In *SIGGRAPH '96 : Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42, New York, NY, USA, 1996. ACM Press.

[LW85] Marc Levoy and Turner Whitted. The use of points as display primitives. In *Technical Report 85-022*, University of North Carolina at Chapel Hill, 1985.

[MB95] Leonard McMillan and Gary Bishop. Plenoptic modeling : an image-based rendering system. In *SIGGRAPH '95 : Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 39–46, New York, NY, USA, 1995. ACM Press.

[MNP01] Alexandre Meyer, Fabrice Neyret, and Pierre Poulin. Interactive rendering of trees with shading and shadows. In *Eurographics Workshop on Rendering*, Jul 2001.

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *SIGGRAPH '00 : Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 359–368, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels : surface elements as rendering primitives. In *SIGGRAPH '00 : Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[SGwHS98] Jonathan Shade, Steven Gortler, Li wei He, and Richard Szeliski. Layered depth images. In *SIGGRAPH '98 : Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242, New York, NY, USA, 1998. ACM Press.

[TZL+02] Xin Tong, Jingdan Zhang, Ligang Liu, Xi Wang, Baining Guo, and Heung-Yeung Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. In *SIGGRAPH '02 : Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 665–672, New York, NY, USA, 2002. ACM Press.

[Wel04] Terry Welsh. Parallax mapping with offset limiting : A per-pixel approximation of uneven surfaces. Technical report, Infiscape Corporation, 2004.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *SIGGRAPH '01 : Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, New York, NY, USA, 2001. ACM Press.