Représentations alternatives du détail visuel pour le rendu en temps-réel

Lionel Baboud

▶ To cite this version:

Lionel Baboud. Représentations alternatives du détail visuel pour le rendu en temps-réel. Informatique [cs]. Université Joseph-Fourier - Grenoble I, 2009. Français. NNT : . tel-00452293

HAL Id: tel-00452293
https://theses.hal.science/tel-00452293

Submitted on 1 Feb 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER GRENOBLE 1

Spécialité : Mathématiques et Informatique

préparée au

Laboratoire Jean Kuntzmann

dans le cadre de

l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

Représentations alternatives du détail visuel

pour le rendu en temps-réel

préparée par

Lionel Baboud

soutenue publiquement le 12 novembre 2009

Composition du jury

Bruno Lévy	Rapporteur	Directeur de recherche à l'INRIA Nancy - Grand Est
Michael Wimmer	Rapporteur	Professeur associé à la Technische Universität Wien
Georges-Pierre Bonneau	Examinateur	Professeur à Grenoble Universités
Bernard Péroche	Examinateur	Professeur émérite à l'Univeristé Lyon 1
François X. Sillion	Directeur	Directeur de recherche à l'INRIA Rhône-Alpes

Résumé

Cette thèse se place dans le cadre de la synthèse d'images en temps réel. Le problème auquel elle s'attaque est celui du rendu efficace du détail visuel, principal élément du réalisme d'une image. Pour faire face à la complexité du détail visuel, il est nécessaire de disposer de représentations adaptées à la fois aux objets que l'on cherche à rendre ainsi qu'aux capacités des processeurs graphiques actuels.
Le premier axe de recherche porte sur l'utilisation du relief pour représenter et rendre efficacement du détail géométrique. La représentation compacte et structurée du relief par une carte hauteur permet la conception d'algorithmes de rendu exacts et efficaces. Nous en proposons deux : le premier permet de rendre des reliefs dynamiques, alors que le second s'adresse aux reliefs statiques en exploitant la possibilité d'effectuer un prétraitement sur la carte de hauteur. Nous developpons aussi une réflexion sur l'utilisation du relief pour la représentation de surfaces quelconques, et présentons une application au rendu réaliste et en temps réel de volumes d'eau.
Le deuxième axe de recherche se concentre sur les représentations non surfaciques, nécessaires lorsque les représentations géométriques sont inadaptées voire inexistantes. C'est le cas notamment des objets lointains ou des objets à géométrie dense, comme par exemple le feuillage d'un arbre. Le problème ici est d'être capable de représenter l'apparence d'un objet, sans recourir à un modèle géométrique. Nous proposons une méthode permettant, à partir de la seule donnée du light-field d'un objet, de déterminer les paramètres optimaux d'une représentation adaptée pour le rendu.

Mot clés

Rendu en temps-réel, représentations alternatives, rendu à base d'images, champs de hauteur, light-field, GPU.

Alternative representations of visual detail
for real-time rendering

Abstract

This thesis takes place in the context of real-time image synthesis. It addresses the problem of efficient rendering for visual detail, main component of an image's realism. To cope with the complexity of visual detail it is necessary to possess representations which are adapted to objects that are to be rendered, together with capabilities of modern graphics processors.
The first research effort concerns the use of relief for efficiently representing and rendering geometrical detail. The compact and structured representation of relief by an height map allows to design efficient and accurate rendering algorithms. We propose two of them: the first one can render animated reliefs, while the second one focuses on static ones by exploiting the possibility to preprocess the height map. We also develop a consideration on the use of relief for the representation of general surfaces, and present an application for real-time rendering of realistic water volumes.
The second research effort focuses on non-surfacic representations, necessary when geometrical representations are inadequate or even non-existent. It is notably the case of distant objects or objects possessing a dense geometry, like a tree's foliage for example. The problem here is to be able to represent the visual aspect of an object without resorting to any geometrical model. We propose a method taking the light-field of an object as only input data, to determine the optimal parameters for an efficiently renderable representation.

Keywords

Real-time rendering, alternative representations, image-based rendering, height-field, light-field, GPU.

Remerciements

Il me tient à cœur ici de remercier les personnes qui à différents niveaux et plus ou moins directement ont contribué à l'aboutissement de cette thèse.

Je tiens en premier lieu à remercier François Sillion pour avoir été mon directeur de thèse, ainsi que Xavier Décoret, dont la co-direction durant mes deux premières années de thèse a été déterminante.

C'est au sein de l'équipe Artis que j'ai eu la chance de faire progresser mes recherches. Difficile d'imaginer meilleur association d'excellence scientifique et de franche convivialité!

Je suis particulièrement reconnaissant aux relecteurs des différentes parties de cette thèse, Franck, Adrien, Thierry et David, dont les corrections et conseils ont été d'une grande aide pour finaliser le manuscrit.

Un remerciement particulier à Karim Kochen dont le travail de DEA a constitué un point de départ pour ma thèse.

Cette thèse doit grandement à la qualité des pauses cafés qui l'ont rythmée. Une pensée déja nostalgique donc pour la fine équipe des David, Juh, Francky et Yo!

Enfin, il va sans dire que mon soutien de plus longue date, je le dois à ma famille, pour son affection et sa confiance sans borne.

Table des matières

1 Introduction **15**
Motivation et contributions 16
Organisation de la thèse 17

I Raycasting et rastérisation **19**

2 Détermination de la visibilité directe **21**
2.1 Hypothèses et définitions 21
2.1.1 Scène 22
2.1.2 Vue 22
2.2 Définition du problème 24
2.3 Deux approches possibles 24
2.4 Forward rendering 26
2.4.1 Rastérisation 27
2.4.2 Autres primitives 27
2.4.3 Avantages 28
2.4.4 Désavantages 28
2.4.5 Gestion de la complexité 29
La complexité interobjets 29
La complexité intraobjets 29
2.5 Backward rendering 29
2.5.1 Avantages 30
2.5.2 Désavantages 30
2.5.3 Gestion de la complexité 31
Complexité interobjets 31
Complexité intraobjets 31
2.6 Comparaison 31
2.7 Conclusion 33

3 Raycasting sur GPU : une méthode hybride **35**
3.1 État de l'art 36
Lancer de rayons sur GPU 36
Structures d'accéleration 37
Rendu volumique 38
Surfaces algébriques 38

3.2 Le raycasting dans une scène rastérisée 38
3.2.1 La technique 39
Génération des rayons de vue 39
Repère normalisé 40
Isométrie 40
Transformation affine 41
Transformation projective 42
Ajustement de l'enveloppe 43
Intégration avec la carte de profondeur 44
Implémentation 45
Dessin de l'enveloppe polygonale 45
Vertex shader : génération des rayons 46
Fragment shader : intersection rayon / surface 46
3.2.2 Utilité 47
Surfaces courbes 47
Effets volumiques 48
Projections non-linéaires 49
Rayons secondaires 51
3.2.3 Exemple : quadriques 52
Résultats 53
3.3 Conclusion 54

II Le relief comme primitive de rendu 55

4 Rendu efficace de relief 57
4.1 Définition 58
4.1.1 Représentation continue 58
4.1.2 Représentation échantillonnée 58
4.1.3 Reconstruction 59
Maillage polygonal 60
Interpolation 61
Plus proche voisin 61
Bilinéaire 62
Bicubique et plus 62
Barycentrique 63
4.2 Rendu efficace sur GPU 63
4.2.1 État de l'art 63
Bilan 65
4.2.2 Approche par raycasting 65
4.2.3 Intersection d'une droite avec le graphe d'une fonction 67
Particularités de notre problème 69
Deux étapes nécessaires 70
Recherche de l'intervalle englobant 71
Aucune recherche 71
Parcours à pas constants 71
Utilisation de données précalculées 72
Calcul précis de la racine 73

Pas de calcul précis 74
Approximation linéaire 74
Recherche dichotomique 74
Conclusion 75
4.2.4 Algorithme sans précalcul 75
Interpolation bilinéaire 76
Autres interpolations 81
4.2.5 Algorithme avec précalcul 81
Propriétés à garantir 82
Le rayon de sûreté 86
Préparation pour l'algorithme de rendu 87
Algorithme de rendu 90
Détail de l'algorithme de précalcul 91
Algorithme dans le cas à une dimension 92
Version simple sans occultations 94
Résultats 97
Compromis qualité / rapidité 97
Version optimale avec occultations 98
Résultats 99
Bilan 100
4.2.6 Géométrie englobante 101
Carte de profondeur 102
4.2.7 Comparaison 103
4.3 Conclusion 103

5 Représentations à base de reliefs 105
5.1 Représentativité d'un relief unique 105
5.1.1 Déformations applicables 106
Transformation affine 106
Transformation projective 106
Limitations 107
Décorrélation de l'information de couleur 108
5.1.2 Imposteur projectif 108
5.2 Combinaison de plusieurs reliefs 109
5.2.1 Relief d'une surface 110
Surface plane 110
Surface courbe 110
Courbure locale 110
Courbure définie par morceaux 111
5.2.2 Capture d'un objet entier 112
Recollage continu 112
Recollage discontinu 114
Segmentation de la surface 114
Segmentation du volume : plusieurs couches 115
Segmentation selon les directions de vue 115
5.3 Conclusion 116

6 Une application : éclairement réaliste dans un volume d'eau animé 119

6.1 Représentation d'un volume d'eau 120
6.1.1 Animation de surface d'eau 121
6.2 Travaux précédents 122
6.3 Modèle d'éclairement simplifié 123
6.4 Éléments à calculer 124
6.4.1 Réflexion exacte 124
6.4.2 Réfraction exacte 125
6.4.3 Absorption lumineuse 125
6.4.4 Caustiques 127
6.5 Algorithme complet 129
6.6 Résultats 130
6.7 Conclusion 132

III Représentations non géométriques 135

7 Représentations par échantillonnage de la parallaxe 137
7.1 Hypothèses et objectif 138
7.1.1 Hypothèses 139
7.1.2 Objectif 139
7.2 Light-field cylindrique 140
7.3 Travaux existants 141
7.4 Cycle d'imposteurs avec parallaxe 142
7.4.1 Définition 143
7.4.2 Algorithme de rendu sur GPU 144
Interpolation de la parallaxe 145
Discontinuités 145
Interpolation 146
Plus proche voisin 146
Linéaire 147
Déformation 149
Combinaison 151
Algorithme complet 151
Résultats 152
7.4.3 Bilan 152
7.5 Reliefs cylindriques 153
7.5.1 Définition 153
7.5.2 Expressivité 154
7.5.3 Algorithme de rendu sur GPU 155
7.5.4 Bilan 155
7.6 Conclusion 156

8 Light-field paramétrique 157
8.1 Définitions 158
8.1.1 Light-field paramétrique 158
8.1.2 Distance entre light-fields 160
8.1.3 Objectif 161
8.2 Travaux précédents 161
8.3 Application 162

8.3.1 Ensemble de polygones texturés 163
Limitations des techniques existantes 163
Représentation paramétrique 163
8.3.2 Plans parallèles additifs 164
Hypothèses 164
Échantillonnage du light-field 165
Linéarité du cas additif 166
Linéarité pour une texture unique 167
Texture additives 168
Optimisation par moindres carrés 169
Résultats 169
8.3.3 Prise en compte de l'occultation 174
Combinaison entre textures 174
Découplage couleur/ opacité 175
Opacité continue 177
Résultats 178
Opacité binaire 179
Résultats 180
8.3.4 Bilan 182
8.4 Possibilités offertes 182
Meilleure représentation 182
Meilleur échantillonnage du light-field 183
Algorithme d'optimisation adapté 183
Meilleur critère de distance entre light-fields 184
8.5 Conclusion 184

9 Conclusion **185**

IV Annexes **187**

A Shaders des algorithmes de rendu de relief **189**
A.1 Vertex shader appliqué à la boîte englobante du relief 189
A.2 Fragment shader pour le rendu sans précalcul 189
A.3 Fragment shader pour le rendu utilisant le précalcul du rayon de sûreté 190

B Algorithme de précalcul du rayon de sûreté **193**

Bibliographie **197**

Chapitre 1

Introduction

Depuis plusieurs décennies, la recherche en synthèse d'images n'a de cesse d'inventer de nouvelles méthodes pour produire les images les plus perfectionnées qui soient. En particulier, le réalisme des images que l'ont sait aujourd'hui produire à l'aide d'ordinateurs a atteint un tel niveau qu'il est généralement devenu impossible pour l'oeil non averti de distinguer une image de synthèse d'une photographie.

D'autre part, la puissance croissante des ordinateurs et notamment l'évolution récente et rapide des capacités du matériel dédié aux opérations graphiques, ont permis la création d'applications interactives : un utilisateur interagit avec un environnement virtuel, dont une représentation visuelle est affichée et rafraichie *en temps réel*, c'est à dire à une fréquence suffisamment élevée pour provoquer chez lui une sensation d'immersion.

Le principal problème auquel se consacre la recherche dans le domaine de la synthèse d'image en temps réel consiste à concevoir des méthodes autorisant la production à une fréquence élevée d'images les plus perfectionnées qui soient, alors que les techniques classiques produisant les images les plus réalistes, requierent généralement des calculs beaucoup trop complexes pour pouvoir être utilisées dans une application interactive.

Une part importante du réalisme d'une image provient de la richesse des détails qu'elle contient : la nature qui nous entoure est complexe, composée d'une grande diversité, d'une multitude de détails variés. C'est cette profusion de détails visuels qu'il faut savoir représenter et synthétiser pour qu'une image ait l'air réaliste, et ainsi produire la meilleure sensation d'immersion dans une application interactive. L'oeil humain étant très sensible à la moindre régularité (surfaces planes, zones uniformes, répétitions, etc.) dans une image, le manque de richesse visuelle dans le rendu d'une simulation interactive peut nuire à la sensation pour un utilisateur d'interagir avec un environnement réel.

La représentation classique et encore largement utilisée pour rendre une scène virtuelle dans le contexte d'une application temps réel consiste à discrétiser les surfaces qui la composent sous la forme d'un ensemble de primitives planes, que le matériel graphique sait dessiner très efficacement. Les surfaces parfaitement planes n'étant que très rarement rencontrées dans le monde qui nous entoure, il faut

généralement recourir à un très grand nombre de ces primitives pour produire une image réaliste, suffisamment riche en détails visuels, ce qui peut poser problème autant du point de vue de la représentation en mémoire que du temps de rendu.

La première approche généralement utilisée pour faire face à la quantité trop élevée de primitives à afficher consiste à simplifier les objets de la scène à rendre, en fonction de leur impact estimé sur l'image finale, notamment en fonction de la place qu'ils occupent à l'écran. Pour la représentation par primitives polygonales, la quantité de détail dépendant directement du nombre de primitives utilisées, simplifier un objet implique nécessairement une diminution du détail visuel qu'il présente. Cependant dans de nombreuses situations le détail visuel peut rester perceptible, même lorsque l'objet occupe peu de place à l'écran (le branchage complexe d'un arbre sans feuilles par exemple, même observé de loin, peut difficilement être représenté fidèlement par quelques primitives polygonales).

La deuxième solution consiste à changer de représentation. La représentation choisie est alors généralement adaptée au type d'objets que l'on cherche à rendre. Bien que cette voie ait déja été considérablement étudiée, l'évolution permanente du matériel graphique offre des possibilités toujours plus larges. Notamment certaines représentations connues mais auparavant inutilisables pour le rendu en temps réel par manque d'adaptation aux capacités de calcul des cartes graphiques, deviennent avantageuses grâce aux nouvelles fonctionnalités dont celles-ci disposent maintenant.

Motivation et contributions La motivation qui a catalysé l'ensemble des recherches effectuées pour cette thèse, est comme nous venons de le montrer, notre conviction de l'importance particulière du détail visuel pour la synthèse d'images en temps-réel, et de la nécessité de représentations adaptées, profitant au mieux de l'évolution récente des capacités du matériel graphique. Les propriétés recherchées pour de telles représentations sont les suivantes :

- encodage le plus concis qui soit d'une grande quantité de détails visuels;
- existence d'algorithmes de rendu efficaces, adaptés au capacités actuelles du matériel graphique, dont le coût est autant que possible indépendant de la quantité de détails représentée;
- existence d'algorithmes de simplification automatique permettant, connaissant l'apparence de l'objet que l'on veut représenter, de déterminer les paramètres d'une telle représentation.

Nos contributions, cherchant à répondre à ces questions, peuvent être classées en trois différents points :

- l'étude du relief comme représentation de détail géométrique, partie importante du détail visuel, avec en particulier la conception d'algorithmes de rendu exacts et efficaces;
- l'étude de représentations non géométriques, c'est à dire adaptées aux cas où il est mal adapté voire impossible de représenter correctement un objet à l'aide de surfaces;
- une méthode d'optimisation pour les paramètres d'une représentation, guidée uniquement par la donnée de l'apparence visuelle de l'objet.

Organisation de la thèse Cette thèse s'organise en trois grandes parties. La première partie constitue une réflexion générale sur les deux principales techniques utilisées pour le rendu, la rastérisation et le raycasting, à la lumière de l'évolution actuelle du matériel graphique. La deuxième partie consiste en l'étude d'une primitive de rendu puissante, le relief, une solution efficace à la représentation et au rendu de détail visuel.La troisième partie enfin traite des représentations non surfaciques, qui permettent la synthèse d'objets ou d'environnements dans des situations où les représentations surfaciques n'existent pas ou sont mal adaptées.

Première partie

Raycasting et rastérisation

Chapitre 2

Détermination de la visibilité directe

Nous nous intéressons dans ce chapitre aux différentes manières d'aborder le problème de la *complexité visuelle* : comment afficher un nombre potentiellement infini d'objets sur un écran constitué de plusieurs millions de pixels ? La représentation réaliste d'une scène implique un nombre de primitives géométriques toujours croissant, et les progrès technologiques demandent de produire des images de résolution en permanente augmentation. Même si l'on peut supposer que la précision limitée du système visuel humain peut un jour amener la résolution des systèmes d'affichage (comme les moniteurs) à cesser de croître, la quantité d'information nécessaire pour représenter une scène de la manière la plus réaliste qui soit ne sera, elle, jamais suffisante.

Nous proposons ici une classification des techniques d'affichage en deux catégories distinctes, permettant d'analyser leur comportement face à la complexité, d'identifier leurs limites respectives et de déterminer les stratégies qui permettront de tirer parti au mieux du matériel existant et à venir.

La première opération de tout système de rendu à être affectée par la complexité visuelle (c'est-à-dire le nombre de primitives à afficher) est la détermination de la visibilité directe (que l'on appele aussi *détermination des surfaces visibles* lorsque les primitives à afficher sont des surfaces, ce qui est le cas le plus souvent).

2.1 Hypothèses et définitions

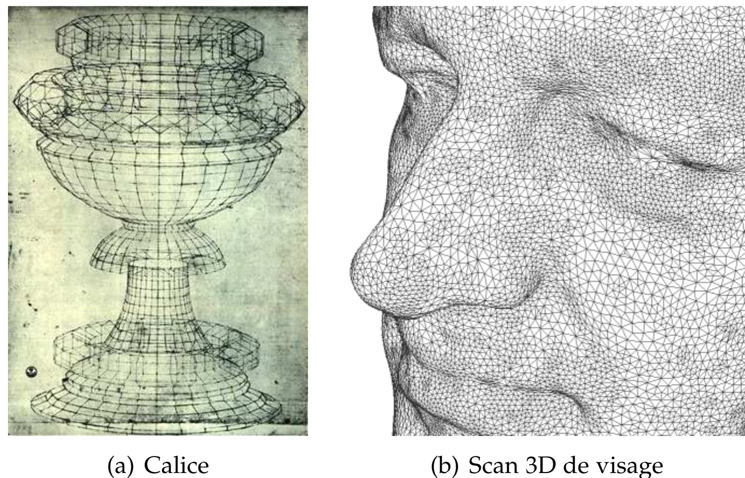
Synthétiser une image consiste à calculer la projection de la représentation d'une *scène* selon une certaine *vue*. Commençons par définir précisement ces deux com-

FIGURE 2.1 – Complexité visuelle

posantes.

2.1.1 Scène

La scène à afficher est représentée numériquement par un ensemble de N primitives géométriques, définies dans une repère de référence que nous appelerons R_M ou *repère monde*, que le matériel graphique sait dessiner très rapidement en très grand nombre. En pratique c'est le triangle qui est la primitive universellement adoptée dans le cadre du rendu en temps-réel (les processeurs actuels sont capables d'en afficher plusieurs centaines de millions à la seconde).



(a) Calice (b) Scan 3D de visage

FIGURE 2.2 – Exemples de polygonalisation : (a) un exemple ancien (dessiné par Paolo Uccello au XVème siecle), (b) une discrétisation obtenue par scanner (www.3dface.org)

2.1.2 Vue

La vue est définie par une transformation projective qui spécifie la manière dont la scène se projette à l'écran. Cette transformation sera notée $T_{V \triangleleft M}$, transformant le repère monde en un *repère vue*, noté R_V. Le *volume de rendu*, c'est-à-dire la partie de la scène se projetant à l'écran, est défini comme étant le cube $[-1, 1]^3$ dans le repère R_V : la projection se fait dans la direction de l'axe z, entre deux plans z_{near} et z_{far}, en faisant correspondre les axes x et y du repère R_V avec ceux de l'écran. La donnée de la transformation $T_{V \triangleleft M}$ définit donc l'angle de vision (horizontal et vertical), ainsi que la *position de l'oeil* ou *centre de projection*, que l'on notera c (éventuellement à l'infini, dans le cas des projections parallèles). L'axe z' obtenu par la transformation affine $z' = {}^{1-z}/_{2}$ est appelé la *profondeur* (voir figure 2.3).

Le choix de cette transformation projective vient d'une part du fait qu'elle possède de bonnes propriétés géométriques et que sa représentation algébrique en permet une implémentation efficace sur le matériel graphique, d'autre part de son adéquation avec le type d'images que l'être humain a historiquement été amené à produire pour représenter la perception qu'il a de ce qu'il voit (voir figure 2.4), d'abord par le dessin en suivant les lois de la perspective linéaire (procédé que

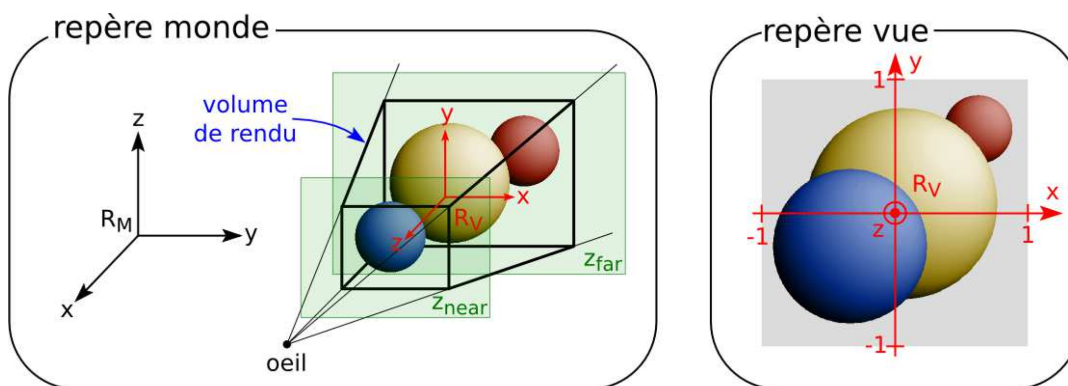


FIGURE 2.3 – Repère monde et repère vue

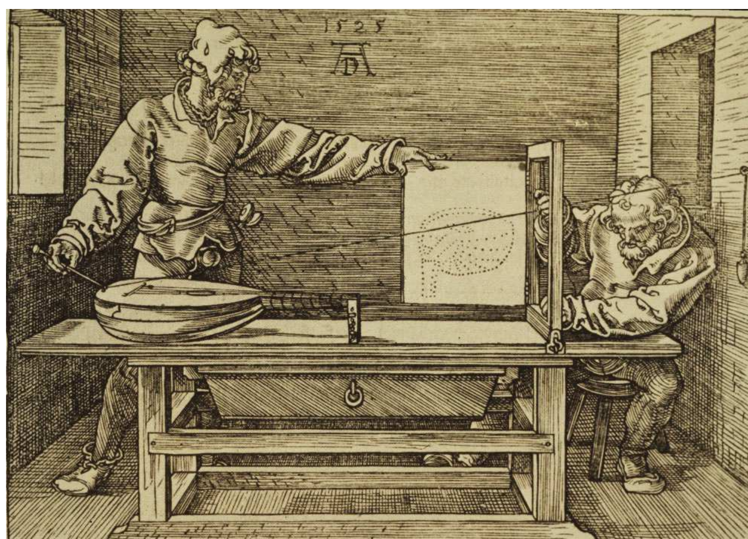


FIGURE 2.4 – Transformation projective : perspectographe de Dürer

l'on attribue à Brunelleschi en 1435), puis par la photographie, dont un modèle simplifié, le sténopé (*pinhole camera*), correspond précisément à une telle projection.

Sous l'effet de la projection ainsi définie, les points de l'espace se projettent à l'écran le long de droites concourantes au centre de projection c, à la manière de photons parcourant des rayons lumineux interceptés par l'oeil. Ainsi si l'on considère un point p quelconque de l'espace, l'ensemble des points de l'espace se projetant à la même position constitue une demi-droite d'origine c et de direction $\overrightarrow{cp}$.

Nous appelons *rayon lumineux* ou plus simplement *rayon* une demi droite de l'espace, définie par une origine et une direction. Chaque rayon définit une relation d'ordre sur l'ensemble des primitives convexes qu'elle intersecte : pour deux primitives convexes A et B intersectant un rayon r d'origine p et de direction $\overrightarrow{d}$, en des points p_A et p_B, on dira que A est intersecté avant B si $\|p_A - p\| < \|p_B - p\|$. On parlera ainsi pour un rayon de première primitive intersectée (ou rencontrée) ou encore

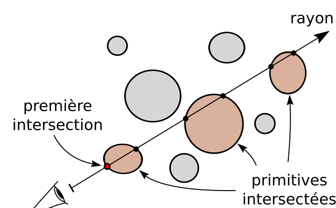


FIG. 2.5 *Ordre d'intersection le long d'un rayon lumineux*

de première intersection, et on notera $r \cap A$ la première intersection d'un rayon r avec une primitive A. Nous supposerons par la suite que les primitives géométriques utilisées sont convexes, ce qui est le cas notamment pour le triangle.

L'écran est discrétisé en P pixels, formant un échantillonnage de la scène projetée. Chaque pixel reçoit une couleur correspondant à l'échantillonnage en son centre de la projection de la scène, c'est-à-dire la couleur du premier objet rencontré le long du rayon d'origine c et passant par le centre du pixel. On appelera le rayon lumineux associé à un pixel un *rayon de vue*.

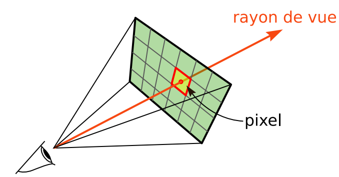
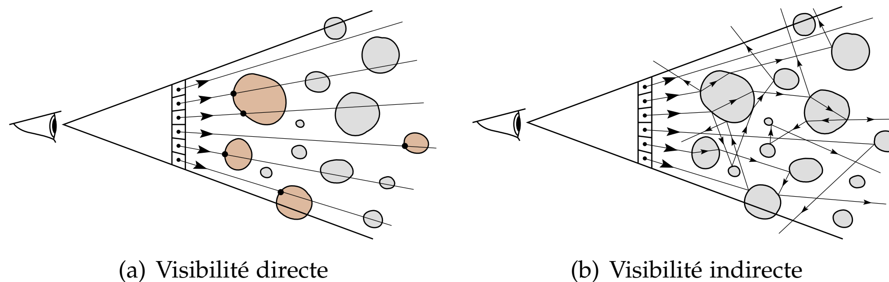


FIG. 2.6 *Pixels et rayons de vue*

2.2 Définition du problème

Le problème consiste donc pour chaque pixel à déterminer dans l'ensemble des N primitives représentant la scène quelle est celle qui intersecte en premier le rayon de vue associé. Une fois cette primitive localisée on peut utiliser ses attributs (locaux, au point d'intersection avec le rayon de vue) pour en évaluer l'éclairement et ainsi produire la couleur du pixel considéré.



(a) Visibilité directe (b) Visibilité indirecte

FIGURE 2.7 – Visibilité directe (a) et requêtes de visibilité indirecte (b).

Quels que soient le degré de réalisme visé (la précision du modèle d'éclairement utilisé) et la technique de rendu employée, cette requête (de visibilité directe) constitue une partie importante du coût de rendu. Ceci car le coût de calcul de l'éclairement, qui peut paraître plus important que celui de la visibilité directe dans le cas de modèles d'éclairement évolués, comprend en fait lui même le coût de requêtes supplémentaires de visibilité directe pour des calculs d'éclairement indirect (ombrage, réflexion, réfraction).

2.3 Deux approches possibles

Si l'on considère le rendu de l'image complète, il faut théoriquement effectuer $N \times P$ projections (tests d'intersection), correspondant à l'ensemble des couples (primitive, pixel) possiblement à examiner. Pour minimiser le coût de rendu il faut donc chercher à n'effectuer qu'un sous ensemble de ces tests.

Deux stratégies de rendu se distinguent alors (illustrées par la figure 2.8) :

1. on traite individuellement les primitives : pour chacune d'elles on cherche les pixels sur lesquels elle se projette. Cette approche est intrinsèquement *input sensitive* : son coût dépend directement du nombre de primitives à afficher (les données en entrée).
2. on traite individuellement les pixels : pour chacun d'eux on cherche la première primitive intersectée par le rayon de vue associé. Cette approche est intrinsèquement *output sensitive* : son coût dépend directement du nombre de pixels à rendre (les donnés en sortie).

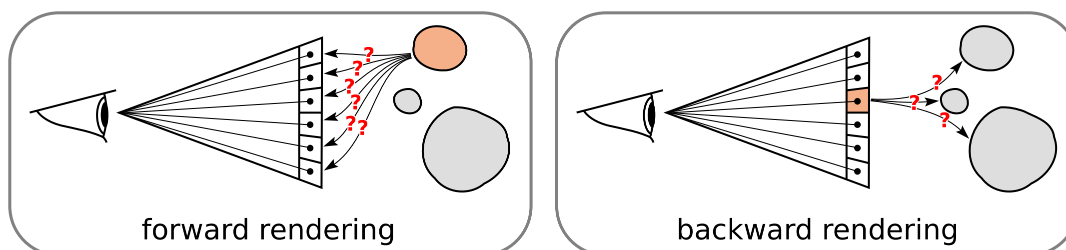


FIGURE 2.8 – Forward rendering et backward rendering.

Nous appelerons ces approches respectivement *forward rendering* et *backward rendering*, par analogie avec la terminologie utilisée pour distinguer les différentes formes de *raytracing* : *forward* raytracing lorsque les rayons se dirigent vers l'oeil, *backward* raytracing lorsque les rayons émanent de l'oeil (attention cependant, lors des débuts du raytracing ces termes étaient parfois inversés par certains auteurs comme Arvo [Arv86], car le mot *forward* désignait alors le sens habituel des rayons lumineux utilisés pour le rendu, c'est-à-dire partant de l'oeil).

Autrement dit, le *forward rendering* consiste à prendre chaque primitive et à la projeter à l'écran, pour obtenir finalement une image de la scène complète, alors que le *backward rendering* consiste à prendre chaque pixel individuellement et à chercher dans la scène ce qui doit s'y afficher.

Cette distinction peut aussi se définir en comparant le sens de la projection effectuée avec le sens de déplacement des photons sur les rayons de vue : le *forward rendering* effectue une projection des objets vers l'écran donc dans le même sens que les photons, alors qu'à l'inverse le *backward rendering* consiste à suivre le trajet des photons à rebours. Le fait que l'on puisse indifféremment choisir l'une ou l'autre de ces deux stratégies provient du principe de retour inverse de la lumière (conséquence du principe de Fermat) : le trajet suivi par la lumière pour aller d'un point à un autre ne dépend pas du sens de propagation de la lumière.

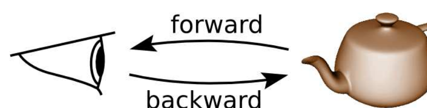


FIG. 2.9 *Sens de parcours de la lumière et stratégies pour le rendu.*

Les termes *forward* et *backward* sont utilisés de la même manière pour distinguer les deux sens de rendu par Teller et Allex [TA99]. Cette distinction est aussi effectuée par Foley et al. [FvDFH90] qui définissent les notions d'*object-precision* quand c'est la scène qui est d'abord parcourue et d'*image-precision* lorsque ce sont les pixels qui sont d'abord parcourus. Notons aussi que cette terminologie est cohérente avec celle utilisée pour distinguer deux approches pour rendre une image à l'écran : on

FIGURE 2.10 – Complexités interobjets (une forêt est constituée d'un grand nombre d'arbres) et intraobjets (chaque arbre a une géométrie complexe).

parle généralement de *forward mapping* lorsque chaque texel de l'image est projeté sur les pixels de l'écran, de *backward mapping*, lorsque pour chaque pixel de l'écran on cherche quel texel de l'image s'y projette.

Pour ces deux manières complémentaires d'aborder le problème de la visibilté directe, les stratégies d'accélération du rendu sont très différentes et elles disposent de propriétés très distinctes. Analyser ces propriétés permet d'en déduire dans quels cas les utiliser.

Nous présentons pour chacune des deux approches les techniques classiques qu'elles englobent, les avantages et désavantages respectifs, ainsi que les stratégies adoptées pour la gestion de la complexité (géométrique). Nous distinguons notamment deux types de complexité : la complexité *interobjet* et la complexité *intraobjet*, où la notion d'objet signifie un ensemble cohérent de primitives (voir figure 2.10).

2.4 Forward rendering

Les méthodes de rendu faisant partie de la classe du *forward rendering* effectuent un parcours des primitives et détermine pour chacune l'ensemble des rayons de vue qu'elle intersecte.

Pour que cela soit pratiquable il faut d'abord que l'ensemble des rayons soit structuré, pour pouvoir sans le parcourir entièrement en extraire efficacement le sous-ensemble concerné par la primitive traitée. La projection perspective (ou orthographique) sur un ensemble de pixels disposés en grille régulière, implique justement une telle structure entre les rayons de vue.

Ensuite comme les primitives sont traitées séparément, il est nécessaire de recourir à une structure intermédiaire permettant de maintenir l'information de visibilité entre ces primitives, pour n'en garder que les plus proches du point de vue. La solution la plus évidente, qui est celle utilisée par les systèmes de rendus actuels (et implémentée par le matériel graphique), consiste à l'aide d'une *carte de profondeur* (*depth-buffer* ou *z-buffer*) de même résolution que l'image à rendre, d'associer à chaque pixel la profondeur minimale des primitives qui s'y sont projeté. La profondeur correspond à la composante z dans le cube projeté, qui peut s'assimiler à une distance par rapport au point de vue, bien qu'elle ne soit pas linéaire,

comme son caractère projectif l'implique. On appele *fragment* l'échantillon (couleur, profondeur) considéré lorsqu'une partie d'une primitive se projette sur un pixel (on peut voir le fragment comme le morceau de la primitive recouvrant un pixel). Ainsi pour chaque nouveau fragment considéré, on compare sa profondeur à celle stockée dans la carte de profondeur à l'emplacement correspondant pour déterminer s'il faut le rejeter (lorsque sa profondeur est *supérieure* à celle actuellement donnée par la carte de profondeur) ou s'il faut le garder (dans le cas contraire) et alors actualiser les valeurs dans la carte de profondeur et l'image rendue par les celles de ce fragment.

2.4.1 Rastérisation

La primitive historiquement adoptée et encore universellement utilisée pour le rendu en temps-réel est le triangle. La technique qui permet de rendre un triangle efficacement par *forward rendering* est la *rastérisation*.

La rastérisation (on parle aussi de *scan-conversion*) d'un triangle consiste à en projeter les trois sommets à l'écran pour obtenir une projection en deux dimensions (qui reste un triangle par propriété de la transformation projective linéaire), puis remplir les pixels affectés par le triangle projeté à l'écran (plus précisément les pixels dont le centre est contenu par le triangle projeté) à l'aide d'algorithmes de parcours simples et efficaces. Cette opération génère un ensemble de fragments, qui sont ensuite testés avec la carte de profondeur pour déterminer lesquels sont visibles et doivent être affichés.

L'efficacité de la rastérisation provient de la propriété qu'a la transformation projective de conserver les droites. La projection p d'un triangle de sommets A, B, et C de l'espace reste un triangle (dans le plan de l'écran), de sommets $p(A)$, $p(B)$ et $p(C)$. Le problème revient donc à chercher les pixels dont le centre se trouve à l'intérieur du triangle projeté, ce que l'organisation régulière des pixels (en grille rectangulaire) permet d'effectuer très efficacement.

La rastérisation est donc un moyen très efficace d'afficher une scène représentée par des primitives planes, sous une vue perspective ou orthographique. La simplicité des primitives utilisées (le triangle comme unique primitive) ainsi que l'efficacité des algorithmes et structures impliqués (rastérisation et carte de profondeur) expliquent que c'est cette technique qui ait été adoptée et développée par les constructeurs de matériel graphique. Les cartes actuelles sont capables d'afficher en temps-réel des scènes composées de plusieurs millions de triangles (voir figure 2.11).

2.4.2 Autres primitives

D'autres primitives que le triangle peuvent aussi être parfois utilisées, principalement les lignes [DCSD02, MPSS05, WOL+05] et les points [LW85, GD98, KB04]. Le rendu à base de points (*point-based rendering* ou encore *point splatting*) à donné lieu à des nombreux travaux, et des techniques permettant un rendu efficace et de qualité ont été developpées [PZvBG00, ZPvBG01, BSK04] (voir figure 2.12).

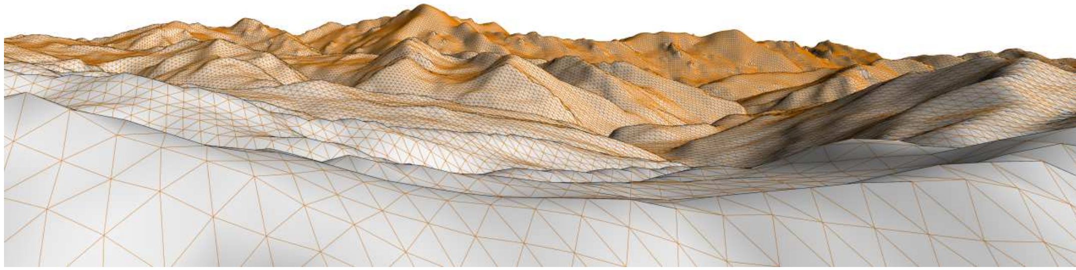


FIGURE 2.11 – Le matériel actuel est capable de traiter plusieurs millions de triangles à la seconde : ici un terrain constitué de 260 000 triangles, rendu à une fréquence d'environ 60*Hz* sur une carte graphique GeForce GTX 285.

FIGURE 2.12 – Exemple de rendu à base de points [BSK04].

Dans tous les cas la technique reste globalement la même, la visibilité est déterminée en utilisant la carte de profondeur et les primitives sont rastérisées d'une manière spécifique à leur forme.

Notons aussi qu'une technique appelée *Forward rastérisation* a été proposée par Popescu et Rosen [PR06] pour résoudre le problème du rendu de petits triangles, pour lesquels le coût de préparation du rastériseur devient non négligeable devant celui de la rastérisation en tant que telle : c'est une méthode à mi-chemin entre la rastérisation de triangles et le rendu à base de points (les primitives restent des triangles, mais ce sont des échantillons ponctuels qui sont projetés pour les dessiner).

2.4.3 Avantages

Les principaux avantages liés aux techniques de *forward rendering* sont :
- la cohérence entre pixels est exploitée : c'est de la structuration en grille des pixels que provient l'efficacité de la rastérisation.
- les scènes dynamiques sont naturellement supportées : le déplacement des primitives ne nécessite aucune structure à recalculer.
- le matériel graphique actuel est optimisé pour cet usage spécifique.

2.4.4 Désavantages

Cette approche comprend cependant certains désavantages :
- les vues que l'on peut ainsi rendre sont restreintes aux seules transformation projectives linéaires (c'est-à-dire les projections perspectives ou orthographiques). Une vue comme une perspective curviligne ne peut pas être rendue avec cette approche.

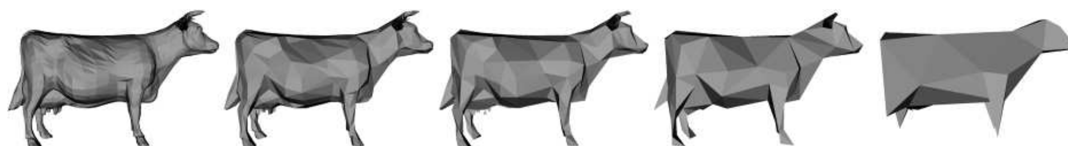


FIGURE 2.13 – Différents niveaux de détails pour un objet [GH97].

- les requêtes de visibilité indirecte ne peuvent pas être effectuées efficacement : par exemple l'ensemble de rayons associé à la réflexion par une surface courbe n'a pas de structure exploitable par rastérisation. Calculer des phénomènes d'éclairement autres que purement locaux (ombrage, réflexion, réfraction) nécessite de nombreuses requêtes de ce type.
- le coût d'affichage est *a priori* directement lié au nombre de primitives affichées (algorithme *input sensitive*), pas à la complexité de la vue affichée.

2.4.5 Gestion de la complexité

Il existe deux stratégies pour faire face à la complexité de la scène dans le cadre du *forward rendering*.

La complexité interobjets

Lors du rendu d'une scène complexe, une grande partie des objets est cachée par d'autres objets. Des techniques de *visibility culling* [BW03], qui s'appuient sur une structure globale de la scène demandant généralement un précalcul, permettent d'éviter d'avoir à dessiner une grande partie des objets de la scène dont on peut déterminer à l'avance qu'ils ne contribueront pas à l'image finale.

La complexité intraobjets

Pour faire en sorte d'adapter le coût de rendu d'un objet en à son importance à l'écran (la surface qu'il occupe), on doit recourir à des techniques de *niveaux de détails* [Hop96, GH97, LWC+02]. Cela consiste à disposer de plusieurs représentations d'un objet, plus ou moins détaillées (généralement obtenues à partir d'une version très détaillée progressivement simplifiée par *décimation*, voir figure 2.13), parmi lesquelles on choisit au moment du rendu laquelle utiliser en fonction de la taille occupée par l'objet à l'écran (qui dépend de sa distance au point du vue pour une vue perspective classique).

2.5 Backward rendering

La deuxième classe d'algorithmes de rendu consiste à traiter indépendamment chaque pixel et à rechercher la première primitive de la scène qui s'y projette. Si l'on considère le rayon de vue associé au pixel en question, cette opération s'exprime comme la recherche de la première primitive intersectée par le rayon. Cette opération est appelée le *raycasting*.

Comme c'est généralement l'usage, nous distinguons ici le terme *raycasting* qui désigne la simple opération consistant à déterminer le premier objet rencontré le long d'un rayon lumineux (et le point précis d'intersection), du terme *raytracing* ou *lancer de rayons*, représentant la grande variété de techniques de rendu qui s'appuient sur une simulation du parcours de la lumière le long de rayons pour estimer la quantité de lumière reçue en un point donné, selon une direction donnée. Cette famille de techniques permet de prendre en compte de manière simple les effets d'éclairement indirect (ombres, réflexions, réfractions) en lançant récursivement des rayons dans les directions nécessaires (en effectuant à chaque fois une requête de *raycasting*).

D'autres techniques de rendu couramment utilisées peuvent s'apparenter à du *backward rendering*. Le texturage ou *texture mapping* consiste à plaquer une image (un tableau de couleurs à deux dimensions) sur un polygone de la scène. Même si le polygone est généralement rendu par rastérisation (donc en *forward rendering*), le fait de le texturer est bel et bien une opération *backward* : pour chaque pixel (séparément) on va chercher dans la texture l'information à afficher. Encore une fois c'est l'aspect structuré des texels en grille qui permet de récupérer efficacement cette information parmi l'ensemble des texels de la texture. De la même manière, la technique de l'*environment mapping* [BN76, Gre86] (échantillonnage de l'environnement d'une scène dans une texture sphérique ou cubique), ainsi que de nombreuses techniques de rendu à base d'image, consistant à remplacer un objet complexe ou lointain par un imposteur 2D peuvent être considérés comme une forme de *backward rendering*.

2.5.1 Avantages

Le *backward rendering* dispose de plusieurs avantages :
- chaque pixel est calculé indépendamment (la structure des pixels en grille n'est pas exploitée), ce qui ne limite pas aux seules projections linéaires les vues que l'on peut calculer.
- des effets d'éclairement indirect (ombrage, réflexions, réfractions) peuvent être facilement intégrés en effectuant le raycasting sur des rayons intermédiaires (rayons réfléchis, rayons transmis, rayons dirigés vers les sources lumineuses).
- le coût d'affichage est proportionnel au nombre de pixels affichés (par définition), ce qui en fait une approche intrinsèquement *output sensitive*.

2.5.2 Désavantages

Si l'on prend cette approche telle quelle, sans technique d'optimisation particulière, elle a plusieurs désavantages :
- elle ne tire pas parti de la cohérence entre pixels voisins.
- il faut *a priori* pour chaque rayon parcourir l'ensemble des primitives de la scène pour déterminer laquelle l'intersecte en premier.
- cette approche est mal adaptée aux scènes dynamiques, car il faut en pratique organiser la scène de manière à diminuer le tests d'intersection à effectuer par rayon, ce qui peut généralement demander un précalcul coûteux, pas toujours faisable en temps-réel.

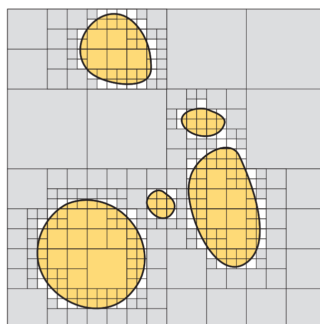


FIGURE 2.14 – Nécessité de structuration de la scène à rendre dans le cas du *backward rendering*.

2.5.3 Gestion de la complexité

Complexité interobjets

La gestion de la complexité interobjets se fait en organisant les objets de la scène à l'aide de structures d'accélération hiérarchiques ou de partitions de l'espace. Ces structures permettent de limiter le nombre d'intersections nécessaires à un nombre faible et généralement borné.

Complexité intraobjets

La complexité intraobjet dépend beaucoup du type d'objet que l'on cherche à rendre. Certaines classes d'objets permettent un *raycasting* efficace (certaines surfaces algébriques par exemple). La stratégie la plus générique consiste à tirer parti de la cohérence des primitives qui forment l'objet, en utilisant des structures similaires à celles utilisées pour structurer les objets entre eux (les octrees ou les kd-trees notamment sont couramment utilisés, voir figure 2.14).

2.6 Comparaison

Nous voyons donc après comparaison de ces deux manières d'aborder le problème de la visiblité directe que même si l'approche *forward rendering* dispose d'une implémentation très efficace à travers la rastérisation de surfaces polygonales, qui en fait la technique *a priori* la plus appropriée pour le rendu en temps-réel, elle possède néanmoins des limitations intrinsèques qui en empêchent l'utilisation exclusive dans certaines applications importantes.

D'abord comme nous l'avons déja expliqué, la complexité et la richesse d'une scène s'appuient fortement sur la quantité de détails représentés, ce qui peut pousser à ses limites une technique de *forward rendering*, intrinsèquement *input sensitive* comme nous l'avons vu. Certes il est toujours possible de s'attaquer à la complexité de la scène à l'aide de techniques de *culling* ou de niveaux de détail, mais elles ont aussi leur limites. D'une part les techniques de *culling* ne sont pas bien adaptées aux objets dont la visibilité est difficilement approchable par une primitive simple (par exemple un arbre au branchage dense, sans feuilles, voir figure

FIGURE 2.15 – Limitation des approches par décimation géométrique pour les objets à topologie complexe.

2.15). D'autre part il n'est pas possible de décimer à l'infini un modèle géométrique (conserver la topologie de l'objet, par exemple peut demander un nombre élevé de primitives), alors que même à une grande distance son influence visuelle peut rester notable. À condition de disposer de structures d'accélération adéquates, les approches de *backward rendering* semblent intrinsèquement plus adaptées à faire front à des scènes très détaillées.

Ensuite la complexité visuelle d'une scène dépend en grande partie du niveau de réalisme du modèle d'éclairement utilisé. Les modèles d'éclairement avancés font appel aux phénomènes d'éclairement indirect comme la réflexion (spéculaire ou diffuse), la réfraction et l'ombrage (ponctuel ou surfacique). Certains effets d'éclairement global commencent aussi à être intégrés aux applications de rendu en temps-réel. Tous ces effets ont en commun de nécessiter des requêtes d'éclairement direct supplémentaires, qui peuvent facilement dépasser le nombre de rayons primaires (le nombre de pixels) du rendu initial (par exemple la vue d'une grande étendue d'eau occupant tout l'écran nécessite pour chaque pixel de lancer au moins un rayon réfléchi supplémentaire, ce qui double le nombre total de rayon à calculer). Les cas où une stratégie de *forward rendering* fonctionne sont rares, puisqu'utiliser une telle technique ne permet que de lancer un ensemble structuré de rayons, définis par une transformation projective unique (une condition nécessaire –mais pas suffisante– étant que les rayons doivent être tous parallèles ou converger en un même point). Le *shadow mapping* et le *réflexion mapping* en sont des cas d'application classiques. Dans ces deux cas les rayons secondaires convergent en un même point (la source lumineuse ponctuelle dans le premier cas, le point de vue réfléchi dans le deuxième cas) ce qui permet l'utilisation du *forward rendering* en passant par des buffers *off-screen* intermédiaires. Cependant même dans ces cas des problèmes d'échantillonnage peuvent apparaître puisqu'il n'est généralement pas possible de trouver une transformation projective qui fasse correspondre l'échantillonnage des buffers *off-screen* et celui des rayons lumineux secondaires nécessaires au rendu de la scène. Voir par exemple à ce sujet les nombreux travaux comme les *Perspective Shadow Maps* [SD02, LGQ+08] qui se confrontent aux problèmes d'échantillonnage posés par les *shadow maps* classiques. Le *backward rendering* semble ici encore naturellement plus approprié, à condition de pouvoir calculer efficacement l'intersection de rayons lumineux avec les objets de la scène.

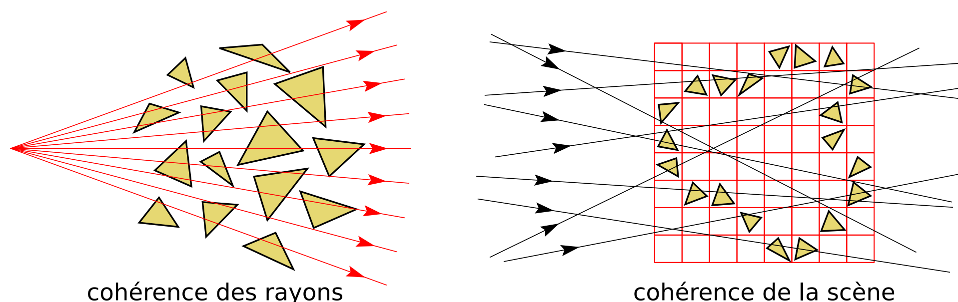


FIGURE 2.16 – Exploitation de la cohérence : des rayons (forward rendering) ou de la scène (backward rendering).

2.7 Conclusion

Nous venons de présenter une classification des méthodes de rendu en deux catégories. La première, le *forward rendering*, représentée par la technique de rastérisation est la plus généralement utilisée par les applications temps-réel et dispose d'une implémentation matérielle sur les cartes graphiques acutelles. La seconde, le *backward rendering*, représentée par la technique de *raycasting* a été traditionnellement attribuée aux algorithmes d'éclairement global comme de *raycasting* mais fait depuis quelques années son apparition dans les applications temps-réel, grâce à la flexibilité croissante du matériel graphique, et permet d'effectuer certains calculs coûteux ou impossibles à effectuer dans un contexte de rastérisation.

Chapitre 3

Raycasting sur GPU : une méthode hybride

Malgré l'évolution rapide du matériel graphique actuel, la fonction pour laquelle il reste le plus efficace est l'affichage d'un grand nombre de polygones par transformation projective, en s'aidant de la carte de profondeur pour déterminer la visibilité. Même si l'architecture des cartes graphiques se rapproche progressivement de celle de processeurs vectoriels parallèles généralistes, permettant leur utilisation pour des calculs beaucoup plus évolués que le simple rendu de triangles par rastérisation, c'est cependant cette technique qui reste adoptée par la majorité des applications nécessitant un rendu en temps-réel.

Comme nous venons de le voir cette technique possède des limites que la seule augmentation du nombre de triangles traitables par les processeurs graphiques, ne permettra jamais de dépasser. Dans de nombreux cas pouvoir effectuer l'opération inverse à la rastérisation, c'est-à-dire le raycasting, s'avère nécessaire.

Nous présentons dans cette partie une méthode hybride, permettant d'intégrer de manière simple et naturelle des éléments calculés par raycasting, dans une scène polygonale rendue par rastérisation. C'est une méthode générique pouvant tirer parti de techniques existantes de raycasting, tout en profitant des fonctionnalités récentes offertes par le matériel graphique. À la différence de nombreux travaux qui

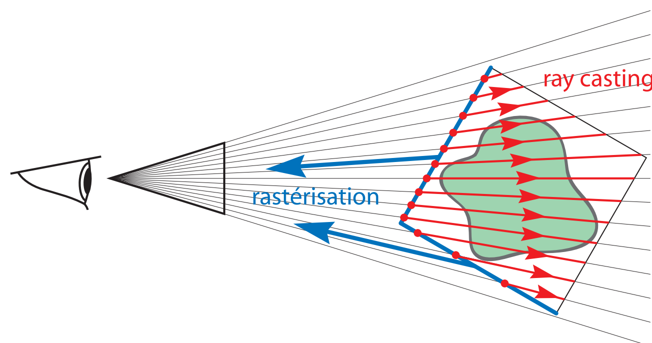


FIGURE 3.1 – Une méthode hybride : une enveloppe polygonale simple de l'objet est rendue par rastérisation, puis un calcul de raycasting est effectué à partir de chaque fragment généré.

utilisent le processeur graphique pour accélérer des calculs de lancer de rayons sur une scène complète, permettant généralement d'obtenir des fréquences d'affichage difficilement utilisables pour une application en temps-réel, nous cherchons ici à profiter au mieux des capacités offertes et proposons une technique utilisable dans un contexte pratique de rendu en temps-réel.

Détaillons d'abord les différentes approches suivies dans les travaux récents pour tirer parti des capacités nouvelles des processeurs graphiques.

3.1 État de l'art

Historiquement le lancer de rayons ou *raytracing*, technique envisagée par Appel [App68] et formalisée par Whitted [Whi80], a été une des première techniques à pouvoir simuler de manière réaliste des phénomènes lumineux avancés comme la réfraction ou la réflexion spéculaire. Par la suite de nombreuses variantes ont été développées pour prendre en compte des effets d'éclairement global [Kaj86]. Cependant, par l'intensité des calculs impliqués, l'architecture des processeurs classiques n'avait jusqu'à récemment pas permi l'utilisation de cette technique dans un contexte de rendu en temps-réel.

Au contraire, le matériel graphique s'est développé autour de la méthode de rastérisation de polygones, pour atteindre actuellement des fréquences d'affichage dépassant la centaine de millions de triangles à la seconde. Pour atteindre de telles performances les processeurs graphiques recourrent à la puissance du calcul parallèle, et se rapprochent progressivement d'architectures de *stream processors* génériques, capables d'effectuer efficacement une même opération sur un grand volume de données.

Dans des domaines divers autres que l'informatique graphique, les chercheurs se sont alors attaché à exploiter la puissance de calcul ainsi mise à disposition, donnant lieu à des méthodes dites *GPGPU* [OLG+07], pour *General-Purpose computations on GPUs*. Des environnements de programmation parallèle génériques dédiés au *stream processing*, permettant d'abstraire l'utilisation du matériel graphique à des applications autres que le rendu ont alors été proposés [BFH+04], et les constructeurs ont mis à disposition des interfaces de plus ou moins haut niveau permettant d'accéder directement aux ressources qu'offrent les cartes graphiques, indépendamment d'un contexte de rendu [NBGS08, Hen07].

Lancer de rayons sur GPU

Nécessitant d'effectuer un même calcul pour un grand ensemble de rayons lumineux, l'algorithme du lancer de rayons est connu pour être fortement parallélisable. C'est donc naturellement qu'il donne lieu à des recherches visant son implémentation pour le processeur graphique, vu comme un *stream processor* générique. Le but visé consiste, soit simplement en l'accélération du calcul de lancer de rayon par rapport à une implémentation sur CPU, soit en l'obtention de fréquences de rendu suffisamment élevées pour être utilisées dans une application interactive, voire temps-réel.

L'idée d'utiliser le *GPU* comme un *stream processor* pour effectuer des calculs de lancer de rayon a été proposée au même moment par Carr et al. [CHH02] et Purcell et al. [PBMH02]. Carr et al. proposent d'utiliser le GPU uniquement pour ce qu'il sait le mieux faire, c'est-à-dire effectuer un même calcul simple parallèlement sur un grand volume de donnés (dans le cas présent, l'intersection d'un triangle avec un ensemble de rayons lumineux), et de laisser au CPU la charge de coordonner les différents calculs de raycasting nécessaires au rendu final de l'image, par simple lancer de rayons récursif [Whi80] ou par *path tracing* [Kaj86]. Purcell et al. proposent eux une implémentation complète de lancer de rayons sur GPU, avec une extension au *path tracing*, grâce à l'utilisation d'une grille volumique régulière, décomposant la scène en cubes contenant chacun une courte liste de triangles. Dans les deux méthodes, de nombreuses passes de rendu dans des *buffers off-screen* sont nécessaires, empêchant l'utilisation d'une telle approche dans le cadre du temps-réel. Une extension au *photon mapping* a aussi été proposée [PDC+03], faisant appel à des techniques de tri et de recherche de plus proches voisins adaptées au GPU.

Structures d'accéleration

Pouvoir calculer le rendu d'une scène complète par lancer de rayons passe par l'utilisation de structures d'accélération adéquates [AK89, HPP00]. Ces structures peuvent être classées comme hierarchies de volumes englobants, partitionnements de l'espace (hierarchique ou non) ou partitionnements de l'ensemble des rayons lumineux (hiérarchiques ou non). Elles peuvent être coûteuses à précalculer, alors que l'apparition récente d'implémentations interactives de lancer de rayon produit un besoin de structures adaptées aux scènes dynamiques [WMG+07].

Les structures de partitionnement hiérarchique de l'espace, comme l'octree ou encore le kd-tree (qui est une des structures les plus utilisées pour le lancer de rayons), ont donné lieu à plusieurs implémentations sur GPU [PGSS07, FS05, HSHH07]. Le GPU ne permettant pas l'implémentation efficace d'une structure de pile, nécessaire lorsque l'on veut effectuer un parcours hiérarchique, plusieurs stratégies sont adoptées pour rendre séquentiel un tel parcours, soit en ajoutant des données supplémentaires (pointeurs entre certains noeuds de l'arbre), soit en effectuant des calculs redondants (reparcours de l'arbre à partir de la racine ou remontée dans l'arbre). Le problème se pose aussi de la construction d'une telle structure, nécessaire pour le rendu d'une scène animée, une implémentation efficace sur GPU est proposée par Zhou et al. [ZHWG08].

Les hiérarchies de volumes englobants (*BVH*, pour *Bounding Volumes Hierarchies*) font aussi partie des structures populaires pour le lancer de rayons et plusieurs implémentations sur GPU en ont été proposées [CHCH06, GPSS07].

Utiliser des structures régulières, non hiérarchiques, permet d'éviter le problème de la récursivité [PBMH02, PDC+03]. Patidar et Narayanan [PN08] proposent même le calcul dynamique d'une telle structure (une grille volumique régulière), alignée avec l'écran, en permettant ainsi un parcours encore plus simple par la suite.

Certains travaux enfin s'attachent à l'utilisation sur GPU de structurations hiérarchiques de l'espace des rayons lumineux [Szé06, RAH07].

Rendu volumique

Le rendu volumique et son implémentation sur GPU ont fait l'objet de nombreuses recherches, ayant abouti à des algorithmes efficaces généralement à base de *ray marching* effectué dans un *fragment shader* [KW03, HSS+05]. Plus récemment Gobetti et al. [GMIG08] proposent une méthode utilisant un octree augmenté de pointeurs entre noeuds voisins, permettant le rendu interactif de très gros volumes pouvant contenir plusieurs milliards de voxels. Crassin et al. [CNLE09] proposent une structure similaire mais plus évoluée permettant notamment de traiter correctement les problèmes de filtrage, ainsi que d'optimiser les transferts de données entre CPU et GPU.

Surfaces algébriques

Un des principaux défauts que présente le pipeline classique de rendu est l'impossibilité de rendre des surfaces courbes. Divers travaux sur le raycasting de surfaces algébriques ont cherché contourner cette limitation.

Les cas les plus simples de surfaces algébriques sont les surfaces quadratiques ou quadriques, comprenant notamment les sphères. Elles ont donné lieu assez tôt à l'implémentation de leur rendu sur GPU [Gum03, dTL04, SWBG06], notamment par leur intérêt en rendu à base de points [BSK04]. Plusieurs exemples avantageurs d'utilisation de quadriques quelconques pour le rendu de modèles complexe existent [KOKK06, SGS06, dTL08] : de manière similaire à la modélisation à base de points, la surface de l'objet est segmenté en un ensemble de morceaux de quadriques qui sont ensuite rendus de manière efficace grâce à une implémentation sur GPU.

Des surfaces algébriques plus complexes ont aussi donné lieu à des implémentations efficaces sur GPU, comme des surfaces de bézier cubiques et quartiques [LB06, dTLP07], des NURBS[PSS+06], des surfaces implicites [KHW+07, KHK+08, SN07], des metaballs [KSN08] (voir figure 3.2).

FIGURE 3.2 – Différents types de surfaces algébriques rendues par raycasting sur GPU : (a) surfaces de Bézier de degré 4 [LB06], (b) NURBS [PSS+06], (c) surfaces implicites [KHK+08] et (d) metaballs [KSN08].

3.2 Le raycasting dans une scène rastérisée

Une grande partie des travaux existants faisant appel à des opérations de raycasting en exploitant le matériel graphique utilisent de manière plus ou moins explicite la même technique, consistant à utiliser le pipeline standard de rendu pour

rastériser les faces d'un volume englobant l'objet à rendre, pour ensuite effectuer pour chaque fragment ainsi généré le calcul de raycasting dans un *fragment shader*.

C'est une méthode hybride s'appuyant à la fois sur la puissance éprouvée de la technique de rastérisation pour déterminer la visibilité entre objets et sur l'efficacité de la méthode de raycasting pour rendre certaines classes d'objets.

C'est à cette classe de techniques que nous nous intéressons dans cette partie. Nous en expliquons les détails pratiques permettant une implémentation simple et efficace, avant d'analyser les cas où elle peut se montrer le plus utile.

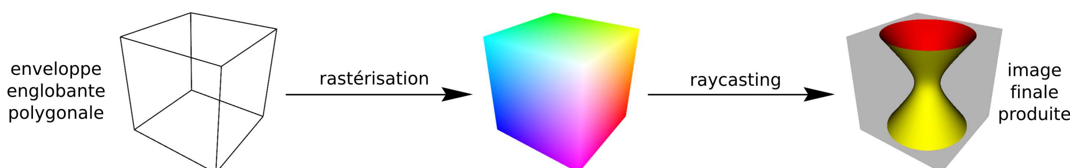


FIGURE 3.3 – Les étapes de la technique de rendu hybride.

3.2.1 La technique

Voici les divers composants de la technique de rendu hybride, permettant son implémentation sur GPU et son intégration dans une scène rastérisée classique.

Génération des rayons de vue

Pour pouvoir rendre l'objet considéré par raycasting, il faut d'abord pour chaque pixel de l'écran potentiellement concerné calculer le rayon de vue correspondant, pour pouvoir ensuite effectuer le calcul de raycasting à proprement parler.

Contrairement aux techniques classiques de lancer de rayons, on ne va pas considérer ici tous les pixels de l'écran : on utilise la capacité du pipeline de rastérisation à traiter la visibilité entre surfaces polygonales pour ramener au minimum le nombre de pixel à envisager par objet. Il suffit pour ceci d'utiliser une enveloppe polygonale simple de l'objet à rendre, puisque la projection à l'écran d'une telle surface englobe celle de l'objet. La solution la plus simple est d'en calculer une boîte englobante rectangulaire, alignée avec les axes du repère utilisé.

Chaque polygone constituant l'enveloppe est rendu de manière classique par rastérisation. Les coordonnées (3D) de ses sommets sont interpolées à sa surface (par interpolation barycentrique, en utilisant les coordonnées de textures par exemple), ce qui permet de connaître pour chaque fragment ainsi généré les coordonnées du point correspondant de la surface de l'enveloppe (c'est-à-dire le point d'intersection avec le rayon de vue associé au pixel).

Connaissant la position du centre de projection de la caméra courante (que l'on peut passer comme constante au *shader* concerné ou encore recalculer connaissant la transformation projective associée à la caméra), on peut alors déterminer pour chaque pixel dessiné les paramètres du rayon de vue correspondant (point de départ et direction).

Le calcul de raycasting est alors effectué dans un *fragment shader* dédié.

Repère normalisé

Selon l'objet que l'on cherche à rendre, les calculs de raycasting peuvent être plus ou moins faciles à effectuer selon le repère dans lequel on se place (voir par exemple le cas du rendu de relief, section 4). D'autre part certains objets ne gardent pas une position statique dans une scène, il faut donc pouvoir leur associer une position et une rotation arbitraire sans avoir à en modifier complètement la représentation.

Pour rendre la technique plus flexible sur ces deux points, nous définissons le *repère normalisé*, que l'on notera R_N, découplé du repère monde R_M dans lequel est exprimée la scène à rendre. Ce repère est défini par rapport à R_M par une transformation $T_{N \triangleleft M}$, allant du repère monde vers le repère normalisé, que nous supposons pour l'instant être une isométrie (c'est-à-dire la combinaison d'une rotation et d'une translation et éventuellement une symétrie). On notera $T_{M \triangleleft N} = (T_{N \triangleleft M})^{-1}$ la transformation inverse.

Cette transformation peut varier dynamiquement, mais dans le repère R_N, l'objet à rendre est toujours fixe. Nous supposerons de plus pour fixer les idées (mais sans perte de généralité) que dans ce repère l'objet à rendre est inclus dans le cube unité ($[0,1]^3$).

Ce cube unité de R_N définit dans le repère monde un cube de sommets $(P_{ijk})_{i,j,k \in \{0,1\}}$ pouvant être utilisé pour le rendu de l'objet. Les sommets de ce cube sont définis de la manière suivante :

$$\forall\, i,j,k \in \{0,1\}, \quad P_{ijk} = T_{M \triangleleft N}(i,j,k)$$

Isométrie L'utilisation d'une isométrie pour la transformation $T_{N \triangleleft M}$ permet de déplacer l'objet dynamiquement dans la scène par translation ou rotation (et éventuellement symétrie).

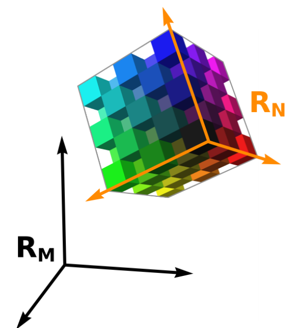


FIG. 3.4 *Isométrie.*

Par hypothèse, les calculs d'intersection des rayons de vue avec l'objet sont effectué dans le repère R_N (autrement dit, les positions et directions manipulées, notamment le point d'intersection et la normale à la surface de l'objet, sont exprimés dans ce repère). Les éléments extérieurs intervenant dans ce calcul sont les paramètres du rayon de vue, pour le calcul de raycasting proprement dit, ainsi que les positions et directions des sources lumineuses, pour le calcul d'éclairement final. Il faut donc exprimer les coordonnées des points correspondants dans le repère normalisé. Notons qu'en utilisant les coordonnées homogènes on peut représenter aussi bien une position de l'espace qu'une direction (point à l'infini, de quatrième coordonnée nulle). Ainsi les rayons parallèles issus d'une vue orthographique peuvent être considérés comme issus d'un même point de vue à l'infini. De la même manière une source lumineuse directionnelle peut être représenté comme une source ponctuelle placée à l'infini. Nous pouvons donc de manière suffisamment générale considérer uniquement les transformations de positions.

Concrètement, le *fragment shader*, qui effectue le calcul d'intersection des rayons de vue avec l'objet à rendre ainsi que le calcul final d'éclairement doit recevoir en entrée les coordonnées, exprimées dans le repère R_N, de l'origine du rayon lumineux, du point d'entrée dans la boîte englobante et des positions des sources lumineuses.

La position du point de vue ainsi que celles des sources lumineuses étant indépendantes du fragment traité, elles peuvent être transformées sur CPU et passées en constantes au shader (variable `uniform` en GLSL), ou encore transformées dans le *vertex shader*, à qui l'on doit alors passer la matrice de transformation $T_{N \triangleleft M}$, lequel les fait ensuite passer au *fragment shader* par des attributs interpolés (attributs `varying` en GLSL).

La position d'entrée du rayon de vue dépend par contre elle du fragment traité, il faut donc interpoler cette valeur aux sommets de la géométrie englobante rastérisée. Il suffit pour cela d'affecter comme attribut interpolé à chaque sommet de l'enveloppe polygonale (le cube P_{ijk}) ses coordonnés normalisées (les valeurs (i, j, k) pour $i, j, k \in \{0, 1\}$), la valeur interpolée en chaque fragment sera alors la position d'entrée dans la boîte du rayon de vue correspondant, exprimée dans le repère R_N. Si l'on a choisi un autre volume (polygonal) englobant, on procède de même mais en affectant en coordonnées de texture (3D) à chaque sommet le résultat de la transformation $T_{N \triangleleft M}$ appliquée à sa position.

Notons que comme les calculs d'éclairement sont effectués dans le repère R_N, les valeurs stockées dans les éventuelles cartes de normales utilisées (*normal map*) doivent être précalculées et exprimées dans ce repère.

Transformation affine Voyons maintenant comment étendre la transformation $T_{N \triangleleft M}$ aux tranformation affines quelconques. Cela peut être utile par exemple pour rendre par raycasting un parallélépipède quelconque lorsque l'on sait déja rendre un cube par raycasting.

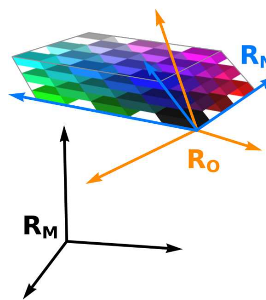


FIG. 3.5 *Transformation affine.*

Les transformations affines ne sont en général pas des similitudes, elles ne conservent pas les angles. Les calculs d'éclairement faisant généralement intervenir des angles (entre direction de vue, direction réfléchie, normale à la surface et direction d'éclairement), ils ne peuvent être effectués pour être corrects que dans un repère défini à une isométrie près du repère de la scène.

Pour éviter de devoir effectuer les calculs d'éclairement dans le repère monde et ainsi devoir recalculer les normales de l'objet (stockées par exemple sous la forme de cartes de normales) dans ce repère à chaque fois que l'objet est déplacé, on definit un repère intermédiaire orthonormé dans lequel seront exprimées les normales et effectués les calculs d'éclairement. Nous appelons ce repère le repère objet ou R_O, puisque c'est ce repère qui permet de positionner l'objet dans la scène.

On définit alors les transformations $T_{O \triangleleft M}$ et $T_{N \triangleleft O}$ (ainsi que leurs inverses respectifs $T_{M \triangleleft O}$ et $T_{O \triangleleft N}$ qui permettent respectivement de passer du repère monde au

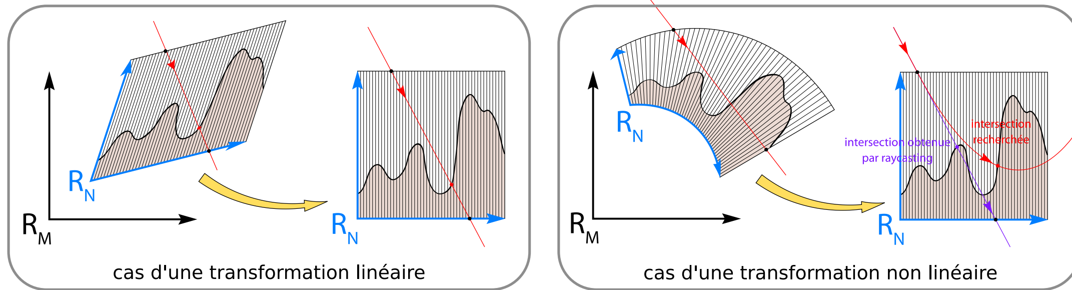


FIGURE 3.7 – Seules les transformations linéaires possèdent la propriété de conservation des droites, nécessaire pour pouvoir appliquer l'algorithme de raycasting dans le repère normalisé.

repère objet, et du repère objet au repère normalisé. Par définition, la transformation $T_{M \triangleleft O}$ est la composée d'une translation et d'une rotation (isométrie directe) et peut varier au cours du temps, alors que la transformation $T_{O \triangleleft N}$ est une transformation affine quelconque mais qui reste constante. Notons que $T_{N \triangleleft M} = T_{N \triangleleft O} \circ T_{O \triangleleft M}$.

Du point de vue de l'implémentation, cette modification implique que le *fragment shader* soit capable de transformer des coordonnées normalisées (utilisées pour le calcul d'intersection) en coordonnées objet (pour les calculs d'éclairement), ce qui implique de lui passer (en paramètre constant, `uniform` en GLSL) la matrice de transformation (généralement de dimensions 3×4) représentant $T_{O \triangleleft N}$. À noter que dans le cas (courant) où cette transformation est juste un changement d'échelle, on peut la représenter par un simple triplet de valeurs (la diagonale de la matrice 3×3 associée) et les opérations de passage entre repère objet et repère normalisés peuvent être implémentées plus efficacement, par de simples produits de vecteurs composantes par composantes.

Transformation projective Il faut bien voir que si l'algorithme de calcul d'intersection effectué dans le repère normalisé produit le même résultat que si on l'exécute dans le repère monde, c'est grâce à la propriété qu'ont les transformations affines de conserver les droites : le rayon lumineux exprimé dans le repère normalisé reste une droite et donc l'algorithme peut s'appliquer correctement.

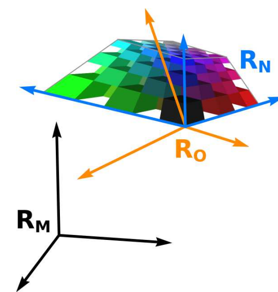


FIG. 3.6 *Transformation projective.*

Cette propriété est nécessaire, mais aussi suffisante : un algorithme d'intersection de rayon avec un objet produit le même résultat après toute transformation T qui conserve les droites (ou autrement dit la colinéarité). Autrement dit, si p est la première intersection d'un rayon r un objet $\mathcal{O}$, la première intersection du rayon $T(r)$ avec l'objet transformé $T(\mathcal{O})$ sera le point $T(p)$. Ou encore si l'on note ψ la fonction de raycasting, qui pour un rayon lumineux r et un objet $\mathcal{O}$ retourne le premier point d'intersection $\psi(r, \mathcal{O})$, on a :

$$T(\psi(r, \mathcal{O})) = \psi(T(r), T(\mathcal{O}))$$

Les transformations projectives (qui comprennent les transformations affines) vérifient cette propriété et ce sont les plus générales qui le font dans $\mathbb{R}^3$.

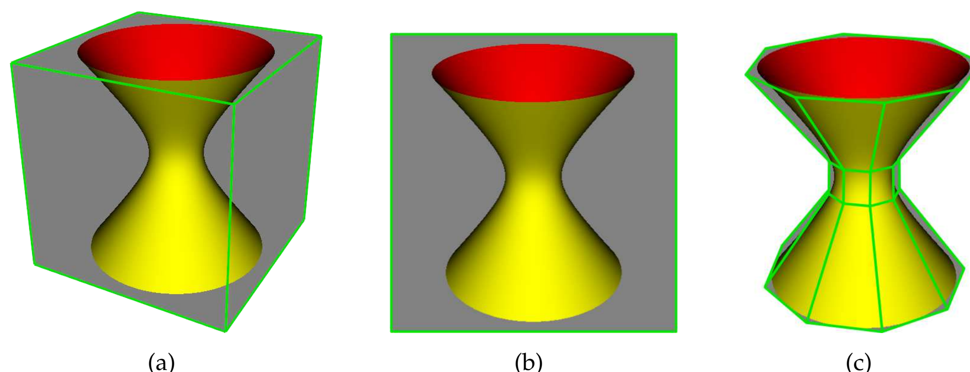


FIGURE 3.8 – Différentes stratégies pour l'enveloppe (arêtes définissant la géométrie utilisée en vert, fragments en excès en gris) : (a) boîte englobante; (b) rectangle faisant toujours face à l'écran; (c) maillage englobant.

On peut donc généraliser encore notre technique en étendant la transformation $T_{N \triangleleft O}$ aux transformations projectives.

Ajustement de l'enveloppe

Comme nous l'avons expliqué, la projection de l'enveloppe polygonale utilisée pour générer les rayons de vue doit englober celle de l'objet pour garantir de rendre tous les pixels qu'occupe l'objet à l'écran (voir figure 3.8). Nous avons vu que la solution la plus simple consiste à rendre la boîte englobante définie par la transformation $T_{M \triangleleft N}$ du cube unité (nécessitant la rastérisation de 12 triangles). De manière générale, n'importe quelle surface englobant complètement l'objet, définie de manière indépendante du point de vue, convient.

Une autre possibilité consiste à rendre une surface dépendant du point de vue. Par exemple s'il est possible quel que soit le point de point de vue de calculer facilement un rectangle, dans le repère de l'écran, englobant la projection de l'objet, il suffit de rendre ce rectangle pour être sûr de courvir tout l'objet. Prenons par exemple le cas simple de la sphère : sa projection est une ellipse dont il est facile de déterminer un rectangle englobant connaissant le centre et le rayon de la sphère. Pour encore réduire au strict minimum le nombre de sommets utilisés (et ainsi le nombre d'appels au *vertex shader*) il est possible d'utiliser un simple *point sprite*, un carré faisant toujours face à l'écran, centré sur la projection du point associé (le *vertex* envoyé à la carte graphique), de taille spécifiable.

Le critère permettant de décider quelle surface englobante choisir est le compromis optimal entre nombre de primitives polygonales rastérisées et nombre de fragments *en excès*. On appele ici fragment en excès un fragment pour lequel le test d'intersection du rayon de vue correspondant avec l'objet à rendre est négatif, c'est-à-dire un fragment qui ne fait pas partie de la projection de l'objet. Le nombre de fragments en excès est lié à la différence entre la surface projetée de l'enveloppe englobante avec celle effective de l'objet. Les fragments en excès s'avèrent d'autant plus coûteux que l'objet est complexe, et peuvent même être plus coûteux que les fragments de l'objet : par exemple lors du calcul d'intersection d'un rayon avec la surface d'un champ de hauteur (voir section 4), il faut parcourir toute la carte de hauteur dans la direction rayon pour se rendre compte qu'il ressort du volume

englobant sans intersecter la surface. Ainsi c'est la complexité de l'objet à rendre qui détermine le ratio optimal recherché entre coût d'une enveloppe polygonale détaillée et coût des fragments en excès.

Intégration avec la carte de profondeur

Lorsqu'un fragment est déterminé comme étant en excès (correspondant à un rayon de vue ressortant de l'enveloppe englobante sans avoir intersecté l'objet), on le rejette grâce à l'instruction correspondante dans le *fragment shader* (`discard` en GLSL), pour ne pas qu'il affecte l'image finale en recouvrant une partie visible de la scène.

Ainsi l'empreinte laissée dans la carte de profondeur par l'algorithme de rendu est celle de la boîte englobante utilisée moins l'ensemble des fragments rejetés. Les valeurs de profondeur écrites (les valeurs de profondeur de l'enveloppe englobante de l'objet), constituent une approximation conservative de la profondeur exacte de l'objet, dans le sens où si aucun objet de la scène ne pénètre le volume défini par l'enveloppe englobante, alors la visibilité entre l'objet et le reste de la scène sera résolue correctement. Autrement dit le objets de la scène se trouvant devant l'enveloppe englobante de l'objet cacheront ce dernier, alors qu'à l'inverse l'objet cachera les parties de la scène se trouvant derrière les faces avant de son enveloppe.

Par contre si l'on autorise des objets à rentrer à l'intérieur de ce volume, voire carrément intersecter la surface du champ de hauteur, il faut calculer la valeur précise de la profondeur au point d'intersection du rayon de vue avec la surface de l'objet et écrire cette valeur dans la carte de profondeur (voir figure 3.9).

Pour calculer cette valeur, il faut utiliser la transformation projective $T_{V \triangleleft M}$ qui fait passer du repère monde R_M au repère vue R_V. Le calcul d'intersection dans le *fragment shader* produit un point p_N exprimé dans le repère normalisé, auquel il suffit d'appliquer la transformation $T_{M \triangleleft N}$ (ou $T_{O \triangleleft N}$ suivie de $T_{M \triangleleft O}$), avant d'appliquer la transformation $T_{V \triangleleft M}$ pour obtenir le point p_V exprimé dans le repère vue. Après normalisation des coordonnées homogènes (c'est-à-dire division par la quatrième composante w), la composante z de ce point donne directement la profondeur recherchée que l'on peut ainsi associer comme nouvelle profondeur du fragment (la valeur initiale étant celle du polygone ayant provoqué la création de ce fragment). Le test de profondeur ensuite effectué par la carte graphique permet de déterminer si ce fragment est effectivement visible auquel cas la valeur calculée est écrite dans l'emplacement correspondant de la carte de profondeur, ou s'il est en fait caché par une partie de la scène se trouvant dans l'enveloppe englobante, auquel cas le fragment est rejeté.

Ce calcul précis de la profondeur est à ajouter au *fragment shader* uniquement dans le cas où c'est vraiment nécessaire, car modifier la valeur de profondeur d'un fragment dans le *fragment shader* désactive les optimisations du *early-z culling*, qui consiste à effectuer le test de profondeur le plus tôt possible dans le pipeline de rendu, avant l'execution du *fragment shader*, pour ainsi éviter des calculs éventuellement coûteux sur des fragments finalement cachés et donc inutiles. Le fait de modifier la valeur de profondeur empêche au processeur graphique de prévoir avant de

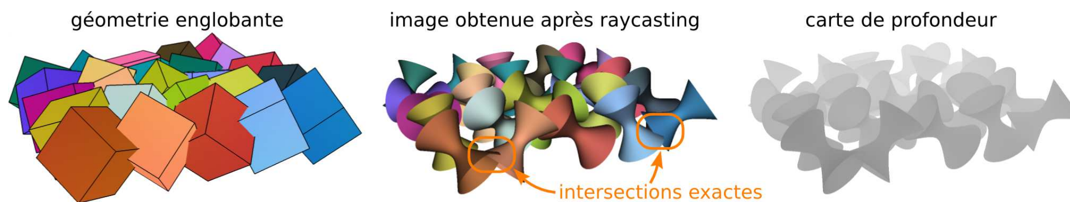


FIGURE 3.9 – Cas d'objets s'entrecoupant : le calcul des valeurs exactes de profondeur et l'actualisation en conséquence de la carte de profondeur est nécessaire.

le calculer si un fragment sera visible ou non. Notons que dans notre cas la valeur écrite est toujours supérieure à la valeur initiale (car la géométrie rendue englobe le relief), ce qui permettrait si la carte pouvait prendre en compte une telle information, d'éliminer avant leur traitement par le *fragment-shader*, tous les fragments se trouvant déja cachés par une partie de la scène déja rastérisée. Cela permettrait d'accélérer considérablement le rendu de scènes complexes où une grande partie des objets rendus par raycasting se trouvent cachés.

Implémentation

Nous donnons ici un exemple d'implémentation de la technique présentée, le plus simple et le plus générique qui soit. Selon les cas rencontrés en pratiques, de nombreuses optimisations sont possibles. Pour simplifier l'explication nous considérons le cas d'une unique source lumineuse, étendre à plusieurs sources l'implémentation ici présentée ne pose pas de problème particulier.

Le code est donné en trois parties : le code de dessin de l'enveloppe polygonale, le *vertex shader* servant à la génération des rayons de vue et enfin le *fragment shader* servant au calcul de raycasting suivi du calcul d'éclairement final.

Dessin de l'enveloppe polygonale La première étape consiste à afficher l'enveloppe polygonale de chaque objet que l'on cherche à rendre par raycasting (dénotés par la liste `objets`), en appliquant aux polygones concernés un couple (*vertex shader*, *fragment shader*) dénoté par la variable `prog`, auquel il faut passer toutes les valeurs uniformes nécessaires : les matrices de changement de repère nécessaires (notons que la matrice `M2V` est le produit des matrices `GL_PROJECTION` et `GL_MODELVIEW` en OpenGL), la position de l'oeil dans le repère objet (`oeil_O`) ainsi que la position de la source lumineuse, également exprimée dans le repère objet (`lum_O`). Le morceau de programme suivant est voué à être executé par le CPU, communiquant avec le GPU par l'intermédiaire d'un contexte dénoté `gpu`.

```
1   gpu->useShaders(prog);
2   foreach(Objet obj, objets) {
3       gpu->pushM2V();
4       gpu->multM2V(obj->O2M);   // M2V <-- M2V * O2M
5       // ainsi on se trouve dans le repère O (autrement dit M == O)
6       prog->setUniform(mat4, M2V);
7       prog->setUniform(mat4, obj->N2O);
8       prog->setUniform(mat4, obj->O2N);
9       prog->setUniform(vec4, oeil_O);
10      prog->setUniform(vec4, lum_O);
11
12      foreach(Poly p, obj->enveloppe->faces) {
13          gpu->beginPoly(p->type);
```

```
14          foreach(Sommet s, p−>sommets)
15              gpu−>drawVertex(s);    // sommets définis dans le repère R_N
16          gpu−>endPoly(p−>type);
17          }
18      gpu−>popM2V();
19      }
```

Vertex shader : génération des rayons La seconde étape, effectuée par le *vertex shader* consiste à générer les rayons de vue. Ceci se fait par l'interpolation d'un point `surf_N` à la surface du polygone traité, défini dans le repère R_N. Il faut aussi calculer et passer au *fragment shader* la position de l'oeil dans le repère normalisé (`oeil_N`). Enfin les position des sommets, spécifiées dans le repère normalisé, doivent être ramenées dans le repère objet avant d'être projetées dans le repère vue.

```
1   varying mat4 M2V;   // == gl_ModelViewProjectionMatrix en GLSL
2   uniform mat4 N2O;
3   uniform mat4 O2N;
4   uniform vec4 oeil_O;
5   uniform vec4 lum_O;
6
7   varying vec4 oeil_N;
8   varying vec4 surf_N;
9
10  void main() {
11      oeil_N = O2N * oeil_O;
12      surf_N = gl_Vertex;
13      gl_Position  = M2V * N2O * gl_Vertex;
14      }
```

Fragment shader : intersection rayon / surface La dernière étape est réalisée par le *fragment shader* : on suppose ici disposer de deux fonctions, dépendant du type d'objet que l'on veut rendre. La première, `raycast`, calcule par raycasting la position (dans le repère normalisé) du point d'intersection de l'objet avec le rayon défini dans le repère normalisé par une origine et une direction. La seconde, `shading`, calcule la couleur à rendre en effectuant le calcul d'éclairement dans le repère objet. Nous donnons ici le cas le plus général où la carte de profondeur est actualisée, avec la profondeur exacte du point d'intersection trouvé, passé du repère normalisé au repère vue.

```
1   varying mat4 M2V;   // == gl_ModelViewProjectionMatrix en GLSL
2   uniform mat4 N2O;
3   uniform vec4 oeil_O;
4   uniform vec4 lum_O;
5
6   varying vec4 oeil_N;
7   varying vec4 surf_N;
8
9   vec3 raycast(vec3 origine, vec3 direction) ;          // dans le repère N
10  vec4 shading(vec4 pos, vec4 oeil, vec4 lum);          // dans le repère O
11
12  void main() {
13      float  signe = mix(−1, 1, step(0.0,  oeil_N.w * surf_N.w));
14      vec3 dir   = signe * normalize(oeil_N.w * surf_N.xyz − surf_N.w * oeil_N.xyz);
15      vec3 orig = surf_N.xyz /  surf_N.w;
16      vec3 p_N = raycast(orig,  dir ) ;
17
18      vec4 p_O = N2O * vec4(p_N, 1);
19      vec4 p_V = M2V * p_O;
```

```
20
21     vec4 color = shading(p_O, oeil_O, source_O);
22     float depth = p_V.z / p_V.w;
23
24     gl_FragColor = color;
25     gl_FragDepth = (1 + depth) / 2;
26   }
```

Le calcul de la direction de vue est donné de la manière la plus générale qui soit, il est correct quelles que soient les transformation utilisées, notamment lorsque le point de vue se trouve à l'infini (dans le repère normalisé) ce qui peut être le cas pour les vues orthographiques. La division par les coordonnées homogènes, potentiellement nulles, est évitée par multiplication par le scalaire `oeil_N.w * surf_N.w`, la variable `signe` servant à rétablir le sens du vecteur calculé, qui se trouve inversé si le produit `oeil_N.w * surf_N.w` est négatif (`signe` prend les valeurs -1 ou 1, mais jamais 0). Selon l'application visée ce calcul peut être simplifié.

3.2.2 Utilité

Nous identifions quatre classes de situations où la technique que nous venons de présenter peut permettre de contourner les limitations inhérentes au pipeline de rendu classique, qui ne sait rendre que des surfaces planes par projection plane (perspective ou orthographique) sur un ensemble de pixels structurés en grille régulière :

- rendu de surfaces courbes;
- rendu d'effets volumiques;
- utilisation de projections non-linéaires;
- calcul de rayons lumineux non-structurés.

Surfaces courbes

Le matériel graphique actuel ne permet de rendre que des primitives planes alors que la quasi-majorité des surfaces rencontrées dans une scène réaliste sont courbes. La stratégie adoptée pour rendre une surface courbe consiste à l'approximer par un ensemble de primitives planes (généralement des triangles) puis de rendre ces primitives. Pour pouvoir approcher au mieux l'éclairement de la surface au moment du rendu, des vecteurs de normale sont généralement associées à chaque sommet et interpolées linéairement sur la surface des polygones.

Cette approche présente plusieurs inconvénients :

- la version polygonale n'est jamais visuellement fidèle à la surface courbe originale, ce qui est généralement perceptible au niveau des silhouettes, apparaissant alors en lignes brisées (continuité généralement C^0 mais pas C^1), ainsi qu'au niveau de l'éclairement à cause de l'approximation faite sur les normales;
- selon la complexité de la surface que l'on cherche à rendre et la qualité de rendu visée, le nombre de primitives à rendre peut être important;
- le coût mémoire d'une représentation polygonalisée est généralement plus important que la représentation originale, notamment dans le cas de surfaces algébriques, définies à l'aide de peu de coefficients;

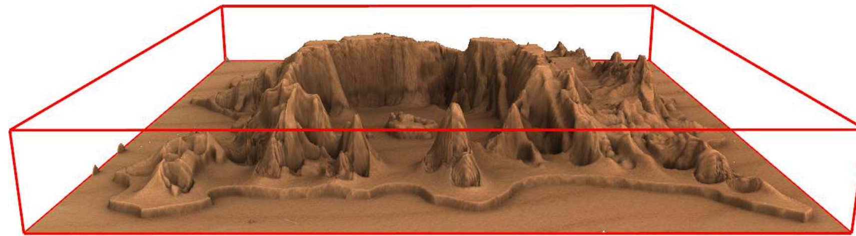


FIGURE 3.10 – Exemple de rendu de champ de hauteur par raycasting (la géométrie utilisée pour initialiser le raycasting est la boîte englobante indiquée en rouge).

– si la surface que l'on cherche à rendre est animée, il faut en réévaluer l'approximation à chaque image à rendre, ce qui peut être coûteux.

Pour permettre la visualisation à n'importe quelle échelle d'une surface courbe, sans avoir à la polygonaliser trop finement, on doit recourir à une méthode de niveaux de détails, consistant à en utiliser plusieurs versions plus ou moins détaillées, selon la surface qu'elle occupe à l'écran. Outre les éventuelles difficultés techniques de mise en œuvre, une telle approche pose le problème de la continuité entre représentations, qu'il est important de maintenir si l'on veut éviter des problèmes de saccades (*popping*), gênants car fortement perceptibles, même sur une petite surface.

C'est donc une situation où une approche par raycasting peut s'avérer avantageuse, à condition que le calcul d'intersection entre la surface est un rayon lumineux soit faisable et suffisamment peu coûteux. C'est le cas notamment lorsque ce calcul d'intersection revient à la résolution d'une équation analytique suffisamment simple (c'est-à-dire que l'on peut espérer résoudre analytiquement ou par une méthode numérique robuste), ou lorsque la surface possède une structure dont il est possible de tirer parti.

L'intersection d'un rayon avec une surface algébrique peut s'écrire sous la forme des racines d'un polynome réel d'une variable réelle, pouvant être résolue de manière exacte, analytiquement ou numériquement, si le degré du polynome n'est pas trop élevé. Les surfaces algébriques comprennent notamment les surfaces de Bézier, les splines ou encore les NURBS, flexibles et pratiques pour modéliser une grande variété de surfaces. Les quadriques notamment, surfaces algébriques de degré 2, sont particulièrement rapides à rendre par raycasting puisqu'il suffit pour cela de résoudre une équation polynomiale de degré 2 pour chaque pixel à rendre. C'est l'exemple que nous prendrons pour illustrer la technique présentée dans ce chapitre (section 3.2.3). Certaines surfaces implicites, algébriques ou non, ou encore des surfaces composées comme les surfaces de révolution, permettent aussi un rendu efficace par raycasting.

Enfin les champs de hauteur sont un exemple de surfaces courbes pour lesquelles une certaine structure existe, dont il est possible de tirer parti pour effectuer un raycasting efficace, comme nous le verrons dans le chapitre suivant (voir figure 3.10).

Effets volumiques

La carte de profondeur utilisée par les pipelines de rendu actuels suppose une visibilité binaire : le fragment généré par rastérisation d'une primitive est soit visi-

ble, soit caché. La prise en compte de semi-transparence est possible grâce au mécanisme de *blending* permettant de définir une opération de mélange, généralement non commutative, entre la couleur d'un fragment et la couleur déja présente dans le *frame buffer* (l'image calculée) à l'emplacement correspondant. Cependant cette technique a ses limitations, la principale étant qu'elle nécessite un rendu ordonné selon la direction de vue (de l'avant vers l'arrière ou de l'arrière vers l'avant) des primitives transparentes de la scène, de par la non-commutativité de la fonction de mélange et l'insuffisance des informations gardées dans la carte de profondeur.

Ainsi la prise en compte d'un milieu participatif pour simuler des effets comme la dispersion de la lumière dans du brouillard, de la fumée ou des nuages est rendue compliquée et nécessite souvent le développement de techniques adaptées [BNM+08].

De manière plus générale, le rendu d'objets ou d'effets volumiques s'intègre mal à un pipeline de rendu classique, à base de primitives surfaciques, alors qu'il s'avère très adapté pour le rendu par raycasting. Certains objets complexes représentables de manière surfacique mais possédant une géométrie dense voire fractale, comme par exemple les arbres, la mousse, la fourrure, les forêts, peuvent aussi avantageusement profiter d'une représentation surfacique, là où une représentation polygonale s'avère mal adaptée.

Là encore, la technique générique que nous avons présenté dans ce chapitre se prête parfaitement à ce type d'utilisation.

Projections non-linéaires

La projection effectuée au moment du rendu par les processeurs graphiques actuels est une transformation projective, autrement dit une projection perspective ou orthographique. Ce choix vient à la fois des bonnes propriétés géométriques et algébriques que présentent ce type de transformations, permettant une implémentation matérielle efficace, mais aussi de leur adéquation à un modèle optique, puisque l'image obtenue par projection d'un objet sur un plan, par un certain point p, provoque le même effet une fois plaquée sur le plan en question, pour un observateur regardant par le point p (voir figure 3.11).

Cependant cette perspective rectiligne, ne correspond qu'imparfaitement à la perception qu'un être humain peut avoir d'une scène qui l'environne, comme l'avaient remarqué relativement tôt certains artistes ou scientifiques (il se pourrait que cela remonte à Léonard de Vinci, bien qu'il n'en existe pas de preuve probante) et comme en témoignent de nombreuses œuvres d'art. Ainsi la perspective dite curviligne [FB68], qui peut être vue comme une projection centrale sur une sphère ou sur un cylindre, peut permettre de mieux rendre compte de la perception humaine que le ferait une perspective linéaire. Cela peut se comprendre par le fait que la représentation que se crée un observateur de son environnement est la composition de la suite continue d'images obtenues en tournant la tête [Com92]. C'est ce qui peut donner l'illusion lorsque l'on se trouve face à un mur que les lignes horizontales qui le composent (comme les bords de briques par exemple) sont courbes. La persepctive rectiligne s'avère mal adaptée à la représentation d'angles de vision

FIGURE 3.11 – Les promenades d'Euclide, de Magritte (1955) : une image plane dessinée selon les règles de la perspective se confond avec la scène représentée lorsqu'elle est observée depuis le bon point de vue.

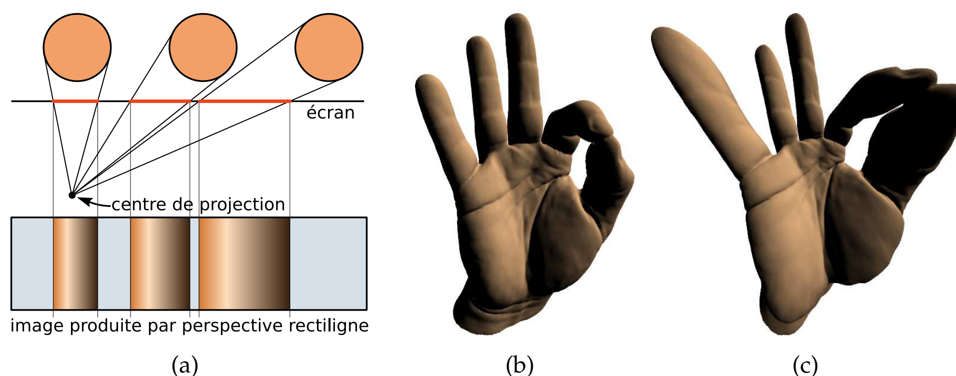
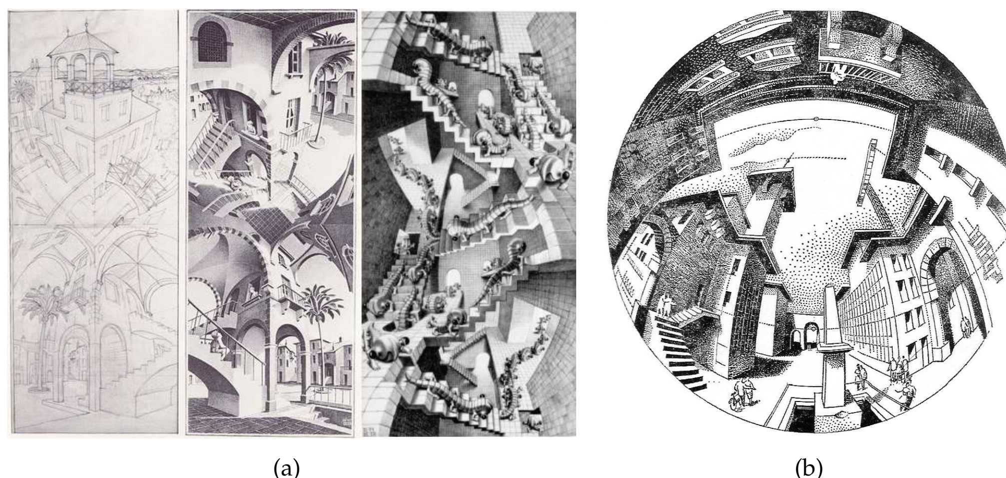


FIGURE 3.12 – Distortions produites par la perspective classique (rectiligne) : (a) illustration du problème en deux dimensions; (b) rendu d'un maillage en perspective avec un angle de vision de 45°; (c) rendu du même maillage avec un angle de vision de 144°.

élevés (voir figure 3.12, cela avait déja été remarqué par Léonard de Vinci [Pan75], et on peut trouver cette observation dans des ouvrages anciens sur la perspective comme celui de Vignola et Danti en 1583 [Com92]), alors que le problème ne se pose pas pour la perspective curviligne.

Outre l'intéret artistique ou perceptuel que pourraient présenter de telles projections non linéaires (voir figure 3.13), certaines sont utilisées indirectement en pratique dans des techniques de rendu. C'est le cas des projections utilisées pour calculer des *spherical maps* ou *paraboloid maps* [HS98], qui permettent de représenter une *environment map* avec de meilleures propriétés d'échantillonnage que les *cube maps* classiques. Une autre application encore de ce type de projection est la simulation d'images obtenues à travers un objectif de type *fisheye*, possédant un angle de champ pouvant aller jusque à 180 degrés.

Calculer les images obtenues par de telles projections ne peut pas se faire de manière exacte à l'aide du pipeline de rastérisation, puisque cela nécessiterait de



(a) (b)

FIGURE 3.13 – Exemples de dessins en perspective curviligne : (a) œuvres de l'artiste M.C. Escher [EE95]; (b) extrait du traité sur la perspective curviligne de Flocon et Barre [FB68].

pouvoir effectuer la rastérisation de la projection courbe d'un triangle plan. En pratique de telles images sont dessinées en appliquant à chaque sommet de la scène la projection non linéaire, en supposant que la scène est suffisamment finement discrétisée pour que la rastérisation de triangles plan approxime correctement la projection courbe théorique. Les efforts à effectuer pour calculer la projection courbe exacte d'un triangle [GHFP08], comparés à la simplicité avec laquelle de telles projections peuvent être calculées par une technique de raycasting, montrent bien à quel point ce dernier peut s'avérer utile.

Rayons secondaires

Enfin la dernière situation où le système de rendu défini par le matériel graphique actuel s'avère trop restrictif est le calcul d'un ensemble non structuré de rayons. L'ensemble de rayons associés à une image que sait rendre le pipeline de rendu actuel possède une structure régulière, correspondant à un échantillonnage régulier (selon une grille de pixels) des rayons partant d'un même point de vue (éventuellement placé à l'infini). C'est justement cette structure régulière (on parle aussi de *cohérence*) qui rend possible la méthode de rastérisation.

La simulation de phénomènes comme la réflexion ou la réfraction, importants car omniprésents dans une scène réaliste, nécessitent le calcul de rayons secondaires, réfléchis ou transmis, dont l'ensemble n'a généralement pas de structure cohérente. Le seul cas de surface pour laquelle la réflexion d'une ensemble structuré de rayons reste structuré, est le plan (voir figure 3.14). Le cas pourtant *a priori* géométriquement simple de la réflexion sur une sphère transforme un ensemble de rayons concourants en un ensemble rayons qui ne le sont plus. Calculer de manière exacte des effets de réflexion spéculaire dans un contexte de rastérisation est un problème difficile [RH06] auquel il n'existe pas de solution générale.

Même dans le cas où les rayons secondaires à calculer sont issus d'un même point, ce qui est le cas pour le calcul d'ombrage ponctuel, leur calcul efficace peut

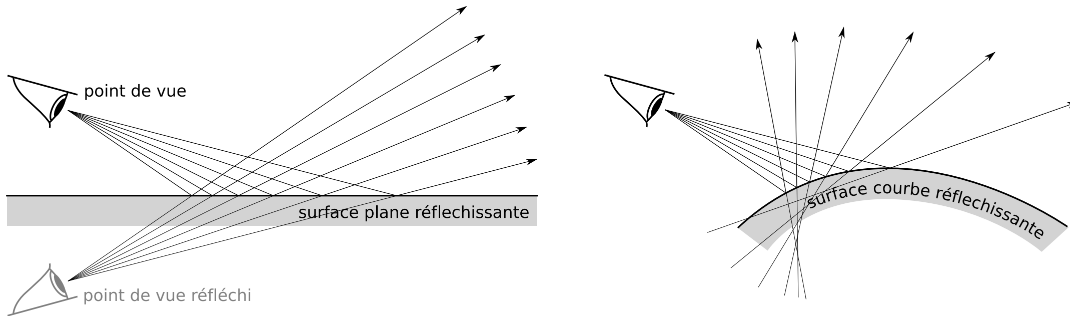


FIGURE 3.14 – Rayons secondaires pour le calcul de réflexions : la propriété de concourance des rayons secondaires dans le cas d'une surface plane ne s'étend pas au cas des surfaces courbes.

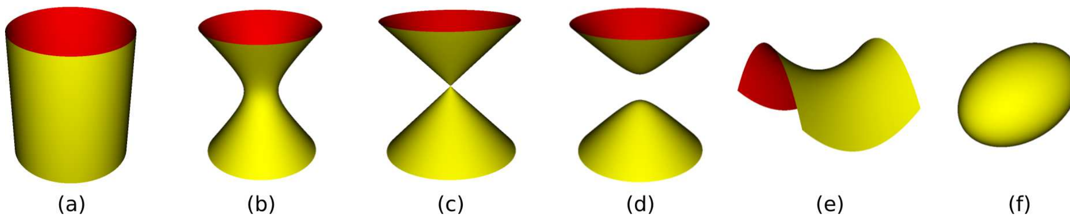


FIGURE 3.15 – Principaux types de quadriques : (a) cylindre; (b) hyperboloïde à une nappe; (c) cône; (d) hyperboloïde à deux nappes; (e) paraboloïde hyperbolique; (f) ellipsoïde.

être problématique s'ils ne correspondent pas à un échantillonnage régulier, comme le montre bien la difficulté toujours existante à utiliser les *shadow maps* pourtant déja proposées il y a une trentaine d'années [Wil78].

Encore une fois le raycasting est la technique appropriée pour le calcul d'un ensemble non structuré de rayons lumineux, nous en montrons au chapitre 6 un exemple d'application dans le cas des réflexions et réfractions spéculaires, pour un contexte de rendu d'eau en temps-réel.

3.2.3 Exemple : quadriques

Les *quadriques* sont les iso-surfaces définies par équation quadratique sur $\mathbb{R}^3$. Toute quadrique $\mathcal{S}$ peut être décrite comme l'ensemble des points $x \in \mathbb{R}^3$ vérifiant l'équation :

$$\langle Qx + L, x \rangle + C = 0$$

où Q est une matrice symétrique (de taille 3×3), L un vecteur de $\mathbb{R}^3$ et C un réel quelconque ($\langle ., . \rangle$ dénote le produit scalaire euclidien sur $\mathbb{R}^3$).

Le paramètre $t \in \mathbb{R}^+$ du premier point d'intersection $p(t) = p + t.d$ d'un rayon lumineux d'origine p et de direction d avec la surface $\mathcal{S}$ est la plus petite solution positive de l'équation :

$$at^2 + bt + c = 0$$

avec

$$\begin{aligned} a &= \langle Qd, d \rangle \\ b &= \langle 2Qp + L, d \rangle \\ c &= \langle Qp + L \rangle + C \end{aligned}$$

On en déduit la fonction `raycast` suivante pour le *fragment shader* permettant de rendre la surface $\mathcal{S}$ par raycasting :

```
1  vec3 raycast(vec3 p, vec3 d) {       // p = origine, d = direction
2      const vec3 Qp = Q * p;
3      const float
4          a = dot(Q*d, d),
5          b = dot(2*Qp + L, d),
6          c = dot(Qp, p + L) + C,
7          delta = b*b − 4*a*c;
8
9      if (delta < 0.0) discard;     // aucune intersection
10
11     const float
12         t1 = (−b − sqrt(delta)) / (2 * a),
13         t2 = (−b + sqrt(delta)) / (2 * a),
14         t = min(t1, t2) >= 0 ? min(t1,t2) : max(t1,t2);
15
16     if (t < 0) discard;
17     // = intersections avec le rayon *avant* son entrée dans la boîte
18
19     const vec3 t_out = (step(0.0, d) − p) / d;
20     if (any(greaterThan(t, t_out))) discard;
21     // = intersections avec le rayon *après* sa sortie de la boîte
22
23     return p + t*d;       // point sur la quadrique
24     }
```

Enfin notons que la normale en un point d'un isosurface définie par une équation de la forme $q(x) = 0$ est dirigée par le gradient de q en ce point. Ici $q(x) = \langle Qx + L, x \rangle + C$, donc

$$\nabla q(x) = 2Qx + L$$

On en déduit la fonction d'éclairement suivante :

```
1  vec4 shading(vec3 pos, vec4 oeil, vec4 lum);  // positions dans le repère O == N
2      vec3 n = normalize(2*Q*pos + L);          // normale en ce point
3      vec3 d = normalize(pos − lum.xyz / lum.w); // direction d'éclairage
4      float coef = dot(n, −d);
5      vec3 color = coef > 0 ? vec3(1,0,0) : vec3(1,1,0);      // couleur selon coté
6      return vec4(abs(coef) * color, 1);                      // éclairement diffus
7      }
```

Résultats

Les résultats que nous donnons maintenant ont été mesurés sur une carte graphique GeForce GTX 285. Le simple rendu de 100 000 boîtes englobantes couvrant un écran de résolution 1280×1024 (consituées de 8 sommets et 6 faces carrées), avec un *fragment shader* retournant une couleur constante demande 5,4 milisecondes (185 Hz). L'application du shader de raycasting de quadrique présenté dans ce chapitre demande 8 milisecondes (125 Hz). On voit bien que le rendu par raycasting s'avère beaucoup plus avantageux pour ce type de surfaces : en effet, un rendu par rastérisation uniquement nécessiterait d'afficher au minimum le double de polygones, et ce avec un algorithme de niveaux de détails géométriques sophistiqué.

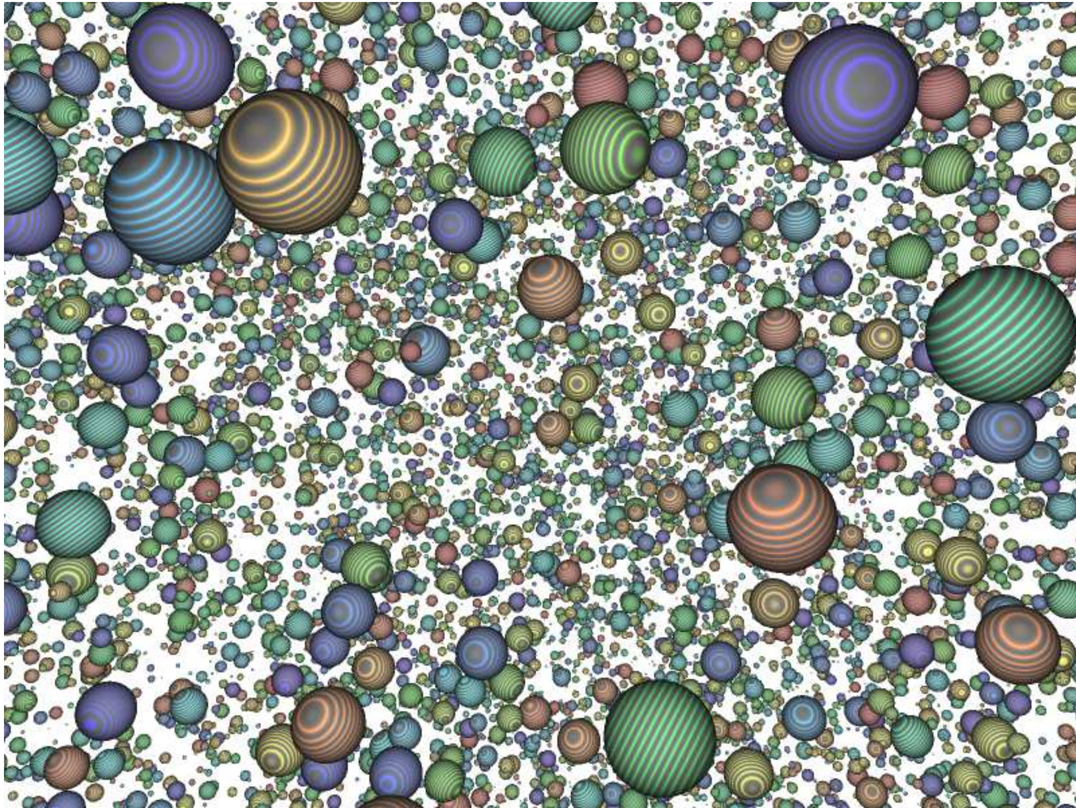


FIGURE 3.16 – Spheres rendues par raycasting sur GPU

3.3 Conclusion

Nous venons de présenter une méthode de rendu hybride tirant parti de la puissance du matériel graphique en termes de rastérisation de polygones, tout en permettant le traitement de cas où cette dernière n'est pas appropriée.

L'application de cette technique au cas particuliers du rendu de quadriques est un exemple de situation où le rendu par raycasting s'avère plus avantageux que la seule utilisation de rastérisation de polygones, tout en permettant une meilleure qualité de rendu.

Nous montrons dans la partie suivante une solution au problème du rendu efficace de relief tirant parti de cette méthode de rendu hybride.

Deuxième partie

Le relief comme primitive de rendu

Chapitre 4

Rendu efficace de relief

Il est rare dans le monde qui nous entoure de rencontrer des objets ou des environnements qui puissent être décrits fidèlement à l'aide de formes géométriques ou algébriques simples.

C'est cependant à l'aide de primitives géométriques très simples que les scènes destinées à être visualisées en temps-réel sont modélisées. La primitive unitaire universellement utilisée est le triangle, qui *a priori* ne permet de représenter que des surfaces planes. La seule possibilité offerte pour arriver à représenter avec suffisamment de fidélité une forme courbe ou présentant de nombreux détails consiste à en discrétiser la surface à l'aide d'un nombre élevé de petits triangles.

La géométrie fractale de la nature fait qu'une surface qui peut à une échelle fixée sembler lisse, dévoile des irrégularités importantes dès que l'on l'observe de plus près. La surface du globe terrestre observée depuis l'espace semble parfaitement lisse, mais si l'on s'en approche, les massifs montagneux s'élèvent de la surface terrestre, les vagues à la surface des océans deviennent perceptibles. La canopé des forêts ou la surface des prairies qui vues de loin présentent un aspect uni, révèlent vues de près la complexité qui les compose. Observés dans leur globalité, les rochers semblent être constitués de faces planes, alors que la rugosité de leur surface devient palpable de près. Même dans les environnements façonnés par l'homme, comme les milieux urbains, rares sont les surfaces qui à une certaine échelle ne présentent pas de détails complexes (sol pavé, mur de briques, mur de pierres, bitume, crépi, façade, carrelage, tissu, bas relief).

La notion importante que ces quelques exemples illustrent est la notion de *relief*. La définition qu'en donne le Littré est la suivante : « Ce qui est relevé, partie saillante d'un objet ». Du point de vue géométrique, le relief peut donc se définir

FIGURE 4.1 – Exemples de reliefs rencontrés dans le monde qui nous entoure.

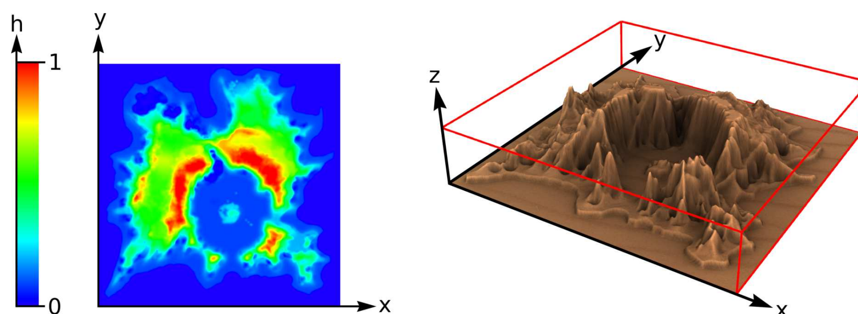


FIGURE 4.2 – Définition d'un relief comme le graphe d'une fonction continue définie sur le carré unité.

comme détail adjoint à une surface. Nous allons voir que la représentation à base de reliefs est une représentation compacte, permettant de rendre efficacement du détail géométrique, sans avoir à recourir à des discrétisations polygonales détaillées des objets concernés.

Ce chapitre se concentre uniquement sur la question du rendu efficace de relief, alors que le chapitre suivant traitera des diverses utilisations qui peuvent en être faites.

Après avoir donné une définition précise de ce que l'on endend par relief et de la représentation numérique associée, nous montrerons, par la présentation de nos contributions dans ce domaine, qu'il peut être rendu très efficacement par l'utilisation adéquate du matériel graphique actuel.

4.1 Définition

4.1.1 Représentation continue

Mathématiquement, le relief peut être représenté par un *champ de hauteur*, défini comme le graphe d'une fonction réelle h continue définie sur le carré unité :

$$h : [0,1]^2 \longrightarrow [0,1]$$

La surface S ainsi représentée est donc :

$$S = \left\{ (x,y,h(x,y)) \ / \ (x,y) \in [0,1]^2 \right\}$$

Cette surface est incluse dans le cube unitaire $[0,1]^3$. Nous appelons ce volume le *volume relief* (voir figure 4.2).

Par la suite nous appelerons h la *fonction de hauteur* (associée à la surface S).

4.1.2 Représentation échantillonnée

La manière la plus simple de représenter un champ de hauteur numériquement consiste à échantillonner la fonction de hauteur sur une grille régulière, appelée *carte de hauteur* (ou *heightmap*). Le carré cartésien $[0,1]^2$ est découpé en $N \times N$ zones

carrées que l'on appelera *texels*, dans chacune desquelles la fonction de hauteur est échantillonnée. L'ensemble d'échantillons obtenu est stocké dans un tableau carré H de N lignes et N colonnes. La valeur $H_{i,j}$ stockée à la ligne i et à la colonne j de ce tableau est définie comme l'échantillonnage de la fonction h au centre du texel correspondant, que l'on appelera τ_{ij}, autrement dit :

$$\forall\ i, j \in [\![0, N-1]\!], \quad H_{i,j} = h(x_i, y_j) \quad \text{avec} \quad (x_i, y_j) = \left(\frac{i + 1/2}{N}, \frac{j + 1/2}{N}\right)$$

Notons d'abord que par souci de simplicité des notations nous utilisons la même fréquence d'échantillonnage N en x et en y, mais que tout ce que nous allons voir ensuite pourrait aussi bien s'appliquer avec des échantillonnages différents selon chaque axe.

Contrairement à d'autres représentations comme les maillages classiques, la relation d'adjacence entre les sommets de la surface échantillonnée n'a pas besoin d'être ajoutée, elle est déja encodée implicitement par l'organisation des échantillons en tableau à deux dimensions. Cela représente un gros avantage en termes de coût mémoire par rapport à une représentation à l'aide d'un maillage.

D'autre part un tableau de valeurs n'est autre qu'une image, qui peut être stockée sur la mémoire de la carte graphique sous forme de texture, élément de base pour manipuler des données sur GPU, et donc optimisé de façon matérielle, notamment en ce qui concerne l'accès à ses valeurs (*texture fetch*, *texture lookup*).

Avant d'aborder le problème de la reconstruction de la surface ainsi échantillonnée, voici quelques définitions qui seront utiles par la suite (voir figure 4.3) :

– $\tau_{ij} = [i/N, i+1/N] \times [j/N, j+1/N]$ dénote le texel situé ligne i, colonne j;
– $c_{ij} = (x_i, y_j)$ dénote le centre du texel τ_{ij};
– les *bordures* sont les droites du plan (x, y) délimitant les texels;
– les *interbordures* sont les droites du plan passant par les centres des texels;
– les *x-interbordures* sont les droites d'équation $x = x_i$ pour $i \in [\![0, N-1]\!]$;
– les *y-interbordures* sont les droites d'équation $y = y_j$ pour $j \in [\![0, N-1]\!]$;
– $\bar{\tau}_{ij} = [x_i, x_{i+1}] \times [y_j, y_{j+1}]$ est l'*intertexel* situé ligne i, colonne j;
– les *cloisons* sont les plans verticaux passant par les bordures;
– les *intercloisons* sont les plans verticaux passant par les interbordures;
– la $i^{\text{ème}}$ *x-intercloison*, dénotée U_i est le plan d'équation $x = x_i$;
– la $j^{\text{ème}}$ *y-intercloison*, dénotée V_j est le plan d'équation $y = y_j$.

4.1.3 Reconstruction

Le problème se pose ensuite de pouvoir reconstruire une surface à partir de cette carte de hauteur. Il existe deux manières de voir ce problème : soit comme un problème de maillage entre sommets, soit comme un problème d'interpolation entre échantillons (reconstruction d'un signal échantillonné).

Les surfaces reconstruites par ces deux approches diffèrent, d'où l'intérêt de bien analyser leurs différences avant de pouvoir envisager un algorithme de rendu.

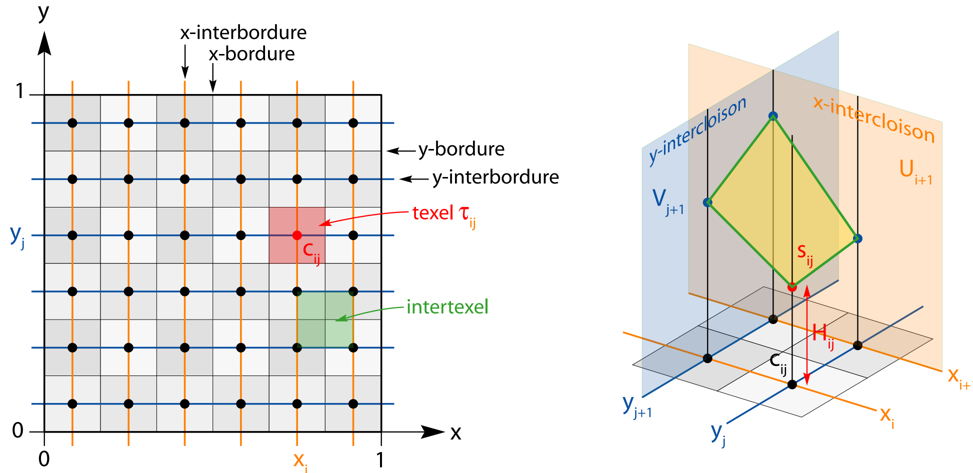


FIGURE 4.3 – Définitions des différents éléments utilisés pour les explications présentées dans ce chapitre.

Maillage polygonal

La carte de hauteur peut être vue comme un ensemble de sommets dont les relations d'adjacence sont encodées implicitement par le voisinage entre texels de la carte de hauteur (voir figure 4.4). Ainsi à l'échantillon H_{ij} est associé un sommet s_{ij}, défini de la manière suivante :

$$s_{ij} = (x_i, y_j, H_{ij}) = \left(\frac{i + {}^1\!/_2}{N}, \frac{j + {}^1\!/_2}{N}, H_{ij} \right)$$

et a pour voisins les sommets $s_{i,j\pm1}$, $s_{i\pm1,j}$, $s_{i\pm1,j\pm1}$.

Si l'on considère l'élément de surface à reconstruire entre quatre sommets voisins $s_{i,j}$, $s_{i+1,j}$, $s_{i+1,j+1}$ et $s_{i,j+1}$, on constate d'abord que ces quatre points n'étant pas nécessairement coplanaires, on ne peut pas utiliser un quadrilatère pour mailler ce morceau de surface. La solution la plus simple consiste à utiliser deux triangles, en choisissant sur laquelle des deux diagonales ($[s_{i,j}, s_{i+1,j+1}]$ ou $[s_{i+1,j}, s_{i,j+1}]$) mettre l'arête qui les sépare. Il faut donc $2(N-1)^2$ triangles pour mailler une carte de hauteur de résolution $N \times N$.

Si aucune information supplémentaire n'est disponible sur la surface représentée (comme un super-échantillonnage de la fonction de hauteur, voire sa donnée analytique), il n'est pas possible de décider quel choix permet une meilleur approximation de cette surface, il faut donc faire un choix arbitraire.

Notons que ce dilemme provient du fait que les échantillons sont répartis sur une grille carrée et que donc la relation de voisinage entre sommets est ambiguë (cas dégénérés pour la triangulation de Delaunay). Une meilleure répartition des échantillons permet de lever cette ambiguïté. La meilleure répartition peut être obtenue simplement en appliquant à la position des échantillons une transformation affine T_{iso}, composée d'une transvection inclinant l'axe y d'un angle ${}^{\pi}\!/_6$ et d'un changement d'échelle selon l'axe des x d'une valeur de ${}^2\!/\!\sqrt{3}$. Par l'application de ce déplacement, les positions d'échantillonnage c_{ij} se retrouvent disposées sur une grille triangulaire régulière (voir figure 4.5).

Pour une telle répartition des positions d'échantillonnage, les texels (régions de Voronoï associées aux positions d'échantillonnage) deviennent hexagonaux et ont

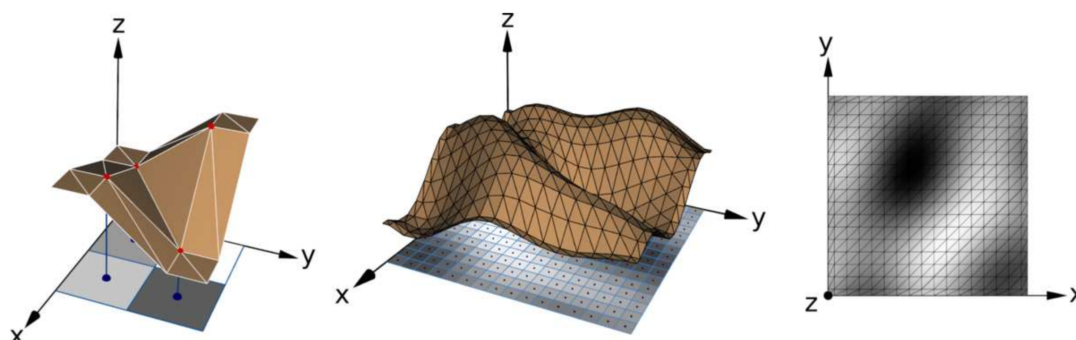


FIGURE 4.4 – Reconstruction d'un relief par maillage à base de triangles.

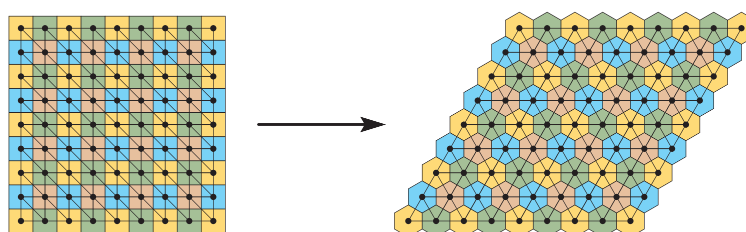


FIGURE 4.5 – Obtention d'une répartition hexagonale d'échantillons par simple application d'une transformation affine à une grille carrée.

chacun 6 voisins, et la méthode de triangulation devient évidente. L'échantillonnage reste représenté par une carte de hauteur carrée mais le domaine de définition du champ de hauteur représenté devient un losange au lieu du carré unité. En pratique il est possible de définir une topologie cylindrique sur la carte de hauteur (en faisant cycler l'axe des x) pour couvrir le carré unité au lieu du losange.

Interpolation

Une autre manière d'aborder le problème de la reconstruction est de voir la carte de hauteur comme un échantillonnage régulier de la fonction de hauteurs. Comme avec tout échantillonnage se pose le problème de l'interpolation entre échantillons. La méthode d'interpolation choisie détermine la forme de la surface reconstruite. Plusieurs options se présentent.

Plus proche voisin La technique d'interpolation la plus simple consiste à associer au point considéré la hauteur de l'échantillon le plus proche (voir figure 4.6). Cela revient à associer à tous les points du texel τ_{ij} la hauteur constante H_{ij}. La fonction ainsi définie est donc constante par morceaux et donc *a priori* discontinue. La surface reconstruite consiste en un ensemble de carrés horizontaux, un par échantillon (surface en escaliers).

Cette méthode d'interpolation pose plusieurs problèmes. D'abord la valeur interpolée à la bordure entre texels est ambiguë. D'autre part la surface reconstruite est discontinue, ce qui peut poser des problèmes de trous au rendu. Une solution consiste à ajouter à cette surface les parties verticales correspondant aux discontinuités entre texels (comme c'est le cas pour la figure 4.6). La surface obtenue n'est plus représentable par une fonction de hauteur (les points aux frontières de texels

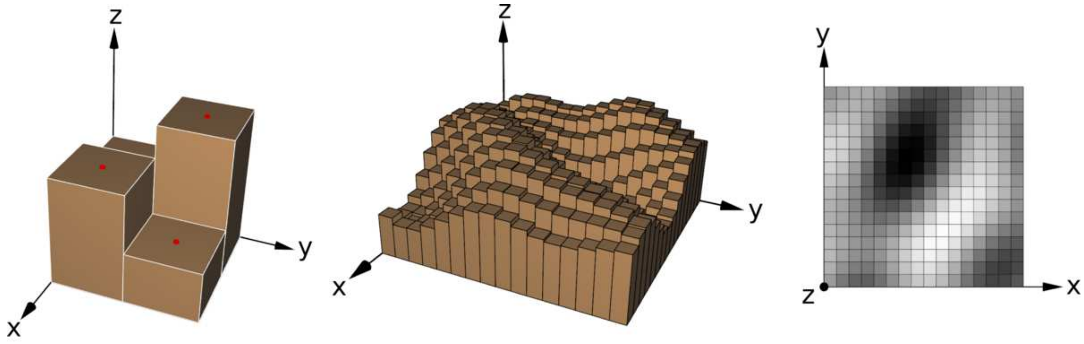


FIGURE 4.6 – Reconstruction d'un relief par interpolation de plus proche voisin.

auraient une infinité d'images), mais elle est continue et peut donc être utilisée pour rendre la surface sans trous.

Bilinéaire L'interpolation bilinéaire est l'expression en dimension deux sur une grille régulière de l'interpolation linéaire (polynomiale, de degré 1) par morceaux en dimension un (voir figure 4.7). Elle se définit par parties sur chaque intertexel $\bar{\tau}_{ij}$ de la manière suivante :

$$\begin{array}{rcrcr} \forall(x,y) \in [0,1[^2, \quad \tilde{h}(x_i + \frac{x}{N}, y_j + \frac{y}{N}) & = & (1-x)(1-y)H_{i,j} & + & x(1-y)H_{i+1,j} \\ & + & (1-x)yH_{i,j+1} & + & xyH_{i+1,j+1} \end{array}$$

La fonction ainsi reconstruite est continue (C^0) et il en va de même pour la surface associée, mais les raccords obtenus au niveau des intercloisons ne sont généralement pas continuement dérivables (C^1).

La surface définie au dessus de chaque τ_{ij} est une quadrique et plus précisément un paraboloïde hyperbolique (ou un plan dans les cas dégénérés), dont les coupes verticales sont des paraboles.

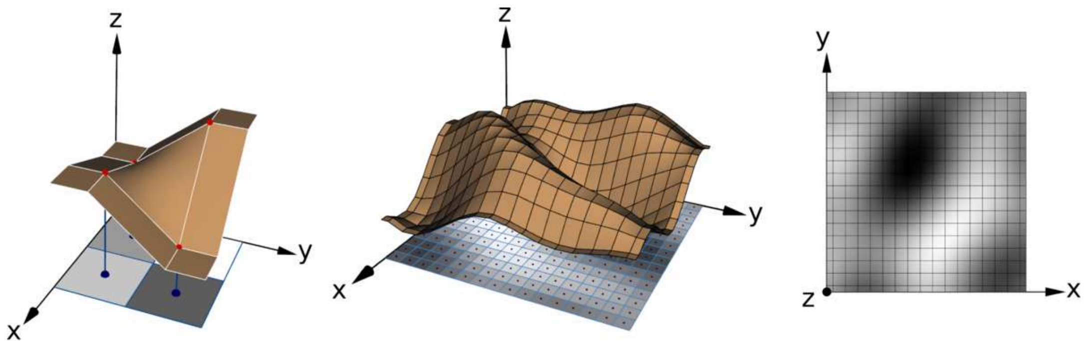


FIGURE 4.7 – Reconstruction d'un relief par interpolation bilinéaire.

Bicubique et plus L'interpolation bilinéaire présente le défaut de ne pas être continuement dérivable (C^1) aux interbordures. Les surfaces ainsi reconstruites présentent des arêtes vives qui peuvent s'avérer gênantes si l'on cherche à représenter une surface lisse avec peu d'échantillons. L'interpolation bicubique consiste à monter encore en degré dans l'interpolation polynomiale (degré 3) et résoud ce problème (voir figure 4.8).

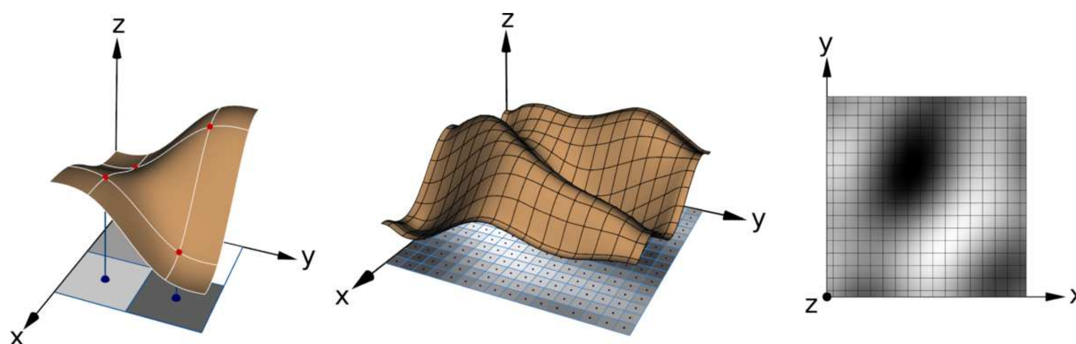


Figure 4.8 – Reconstruction d'un relief par interpolation bicubique.

Barycentrique Comme leurs noms l'indiquent, les interpolations bilinéaires et bicubiques s'appuient sur les coordonnées cartésiennes : l'interpolation (linéaire, cubique) est effectuée sur chaque coordonnée successivement, dans n'importe quel ordre. Ceci est rendu possible par le fait que les échantillons se trouvent sur une grille rectangulaire.

Nous avons vu précédemment qu'en utilisant une grille hexagonale on peut obtenir de meilleures propriétés d'échantillonnage. Le maillage triangulaire qui peut y être associé comme nous l'avons présenté peut être vu comme le graphe de l'interpolation linéaire des coordonnées barycentriques (associées aux positions des échantillons).

4.2 Rendu efficace sur GPU

Nous venons de voir que le relief peut être encodé à l'aide d'une carte de hauteur, ce qui s'avère une représentation compacte et adaptée au matériel graphique, puisque l'utilisation d'une simple texture (2D) permet de stocker un relief entièrement dans la mémoire associée au processeur graphique. Cela n'a d'intérêt que si l'on dispose en plus d'un algorithme de rendu efficace uniquement executé par ce dernier. C'est l'objet de la partie qui suit. Après une revue des méthodes existantes, nous exposons notre contribution dans ce domaine, consistant en deux algorithmes robustes dont la particularité par rapport aux méthodes existantes consiste en l'exactitude du rendu généré (l'absence d'artefacts visuels).

4.2.1 État de l'art

Il existe une vaste littérature sur le rendu de terrains, que cela soit par raycasting ou au moyen de maillages triangulaires. Dans le premier cas, il faut disposer d'une procédure efficace d'intersection de rayon avec la surface et la solution généralement utilisée consiste à disposer d'une structure hiérarchique du terrain à rendre (un quadtree généralement). Dans le second cas, il faut faire face à un nombre potentiellemnt très elevé de triangles, ce qui peut être évité en s'appuyant sur des représentations multirésolution du terrain, adaptatives en fonction du point de vue pour ne sélectionner que la géométrie nécessaire, à la bonne résolution (voir par exemple [DWS+97]). Notons cependant que les structures hiérarchiques utilisées

se prêtent généralement mal à une implémentation pour le processeur graphique, celui-ci manquant d'instructions de branchement.

L'idée d'utiliser des cartes de hauteur pour ajouter du détail à une surface est attribuée à Cook [Coo84]. La technique, appelée *displacement mapping*, consiste simplement à déplacer les sommets de la surface concernée selon les valeurs données par une carte de hauteur, quitte à ajouter suffisamment de sommets par triangulation pour obtenir le détail recherché. Bien que des techniques existent [GH99], cherchant à minimiser le nombre de triangles rendus en fonction du point de vue courant, une telle approche peut demander de rendre un grand nombre de triangles.

Sont alors apparues les premières approches inverses de ce problème, généralement appelées *inverse displacement mapping* ou encore *per-pixel displacement mapping*, où au lieu d'afficher la géométrie déplacée détaillée (c'est-à-dire modifiée par déplacement selon une carte de hauteur) de la surface, on cherche pour chaque pixel à rendre quel point de la surface déplacée s'y projette. Le problème revient alors à effectuer des calculs de raycasting avec la carte de hauteur, en se plaçant dans un repère lié à la surface déplacée, comme le montrent les premiers travaux à ce sujet [PHL91, LP95]. Une telle approche peut être vue comme une évolution de *bump mapping* ou *normal mapping* [Bli78], consistant à ajouter du détail à une surface sans en modifier la géométrie, par simple plaquage d'une carte de normales, provoquant après calcul d'éclairement une impression de relief. La principale limitation de cette technique, le fait qu'elle soit inappropriée pour les angles rasants et qu'elle ne permette pas de produire des silhouettes détaillées, est corrigée par le *displacement mapping* (inverse), au prix de calculs *a priori* plus complexes.

Avec les progrès du matériel graphique, il est devenu possible de rendre des surfaces en temps-réel par cette approche, en déléguant le calcul de raycasting au processeur graphique. Les premiers travaux dans lesquels cette possibilité a été étudiée utilisent un parcours dense du rayon de vue (*ray marching*), soit en testant successivement chaque texel de la carte de hauteur rencontré le long de ce rayon [QQZ+03, HS04], soit en utilisant un échantillonnage régulier du rayon [HEGD04].

Pour atteindre des temps de rendus compatibles avec une utilisation en temps-réel, des techniques de rendu approximatives ont été proposées, où une intersection d'un rayon lumineux avec la carte de hauteur est calculée rapidement mais sans la garantie d'obtenir la première intersection. Les premières techniques à avoir adopté cette approche sont celles dites de *parallax mapping* [KTI+01, Wel04], où une approximation grossière du calcul d'intersection avec le relief est utilisée pour améliorer le résultat que produirait un simple *bump mapping*. Parmi les techniques populaires plus évoluées, la première solution consiste à recourir à un échantillonnage régulier du rayon lumineux, en réglant le nombre de pas selon les objectifs de temps de rendu fixés [Tat06]. La deuxième solution consiste à utiliser un algorithme de recherche dichotomique, comme la bissection [POC05, OP05, PO06, dTWL07, dTWL08] ou la méthode de regula falsi [SKALP05, RSP07], après un éventuel parcours régulier visant à diminuer les artefacts produits (intersections manquées).

Dans le but d'obtenir des rendus sans erreurs visuelles tout en restant dans un contexte temps-réel, des méthodes ont été proposées qui recourent à des données additionnelles, généralement précalculées. L'information supplémentaire utilisée est soit la donnée d'une distance à la surface du relief [KRS05, Don05, PO07],

soit une structure hiérarchique proche d'un *quadtree* [OKL06, TIS08]. Ces méthodes sont généralement moins efficaces que les méthodes sans précalcul mais permettent de garantir une certaine qualité de rendu.

Notons enfin que le problème consistant à déformer l'image d'un objet, prise sous un angle de vue particulier, pour laquelle une carte de profondeur est connue, pour obtenir une image de l'objet en question sous un autre angle de vue, a été considérablement étudié dans le domaine du rendu à base d'images (*image-based rebdering*) et a donné lieu à des techniques dites d'*image warping* [MB95, SP99, OBM00]. Bien que le problème soit conceptuellement équivalent à celui du *displacement mapping*, les techniques utilisées sont généralement mal adaptées au matériel graphique actuel.

Bilan

Parmi les méthodes existantes qui présentent des temps de rendus suffisamment intéressants pour une utilisation en temps-réel, très peu sont exemptes d'erreurs de rendu. Les méthodes populaires [POC05, Tat06] ne sont utilisables que pour des reliefs suffisamment lisses et de faible dynamique. Seules certaines méthodes recourant à un precalcul arrivent à produire des rendus sans artefact, mais elles sont généralement plus lentes et ne supportent pas les reliefs dynamiques, voire nécessitent le stockage trop coûteux en mémoire de données supplémentaires.

Pour faire face à un tel besoin, nous nous sommes donc attachés à la recherche d'un algorithme présentant les propriétés suivantes :
- aucune erreur de rendu, quelle que soit la distance au point de vue,
- supportant les reliefs arbitrairement complexes,
- simple et efficace, implémentable sur processeur graphique.

4.2.2 Approche par raycasting

Les algorithmes de rendu efficace de relief que nous allons détailler maintenant se placent dans le cadre de la méthode hybride de raycasting sur GPU présentée au chapitre précédent. Comme nous allons le voir, la représentation d'une surface par une carte de hauteur se prête parfaitement à une telle approche, de par la structuration implicite en grille des éléments de surface encodés. Cela permet d'une part de profiter des avantages intrinsèques aux techniques de raycasting sur GPU, notamment l'aspect *output sensitive* du coût de rendu. D'autre part si l'on reprend la définition de relief, (détail ajouté à une surface), on peut justement considérer l'opération de raycasting sur la représentation polygonale simplifiée d'un objet comme une manière d'ajouter du détail à sa surface, sans avoir à modifier la géométrie de la surface elle même. Notons cependant que les algorithmes que nous présentons ici ne se restreignent pas à l'ajout de méso-détail à des surfaces, comme c'est généralement le cas dans la littérature existante à ce sujet, mais qu'ils peuvent aussi bien être utilisés pour rendre des objets entiers, voire même le terrain de la scène à rendre.

Rappelons qu'avec la définition du champ de hauteur que nous avons donnée, la surface à rendre est incluse dans le cube unité $[0, 1]^3$. On peut donc choisir de

rendre les six faces carrées englobant ce volume comme enveloppe polygonale servant à la génération des rayons de vue intersectant potentiellement la surface. Les méthodes de changement de repère décrites en section 3.2.1 peuvent dans ce cas s'appliquer pour déformer et placer la surface n'importe où dans la scène. On peut donc sans perte de généralité supposer que l'on se trouve dans le repère normalisé R_N, défini de sorte que le volume de relief corresponde au cube unité $[0,1]^3$. C'est cette convention que nous prenons dans ce qui va suivre.

Notons que cette façon d'aborder le problème du rendu de relief est similaire aux techniques dites de *warping* [SP99, OBM00]: on calcule, selon le point de vue, une version déformée de la carte de hauteur, que l'on plaque ensuite sur une boîte englobante. La différence cependant étant que dans notre cas, l'échantillonnage est effectué selon la vue (un rayon par pixel à rendre) alors que les approches par warping génèrent des images échantillonnées régulièrement selon la surface sur laquelle elles sont plaquées. Cela peut se voir comme une technique de trompe l'oeil ou anamorphose, consistant à dessiner un objet sur une surface destinée à être observée depuis un point de vue particulier, en appliquant la déformation nécessaire pour provoquer l'impression de relief.

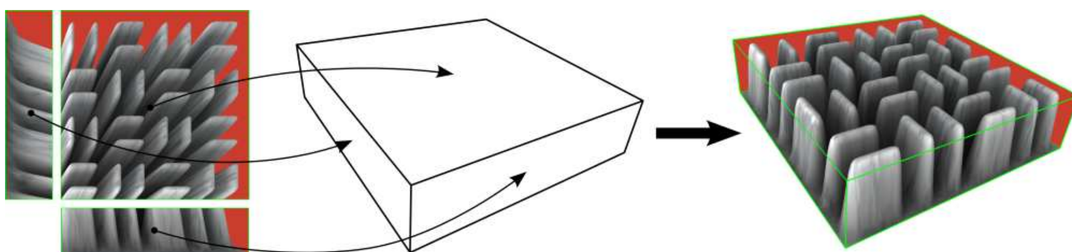


FIGURE 4.9 – Le rendu de relief par raycasting peut se voir comme une opération de warping : tout se passe comme si trois textures 2D étaient calculées en fonction du point de vue puis plaquées sur la boîte polygonale.

On est donc ramené au problème de raycasting consistant à chercher la première intersection d'une demi-droite avec la surface du relief (s'il en existe une). Ce problème à priori en trois dimensions peut se ramener en deux dimensions : prenons le plan vertical (c'est-à-dire orthogonal au plan de base du champ de hauteur) qui contient le rayon lumineux (nous l'appelerons P, ou plan de coupe), il existe un et un seul tel plan, sauf dans le cas dégénéré où le rayon est parfaitement vertical, cas où le raycasting s'avère trivial. L'intersection de ce plan avec la surface du champ de hauteur est le graphe d'une fonction g à une dimension (définie de $\mathbb{R}$ dans $\mathbb{R}$). Si l'on appelle p_0 et p_1 les points d'entrée et de sortie du rayon lumineux dans le volume de relief (le cube unité), on peut paramétrer la partie du rayon lumineux intersectant la boîte par une variable t comme suit :

$$\forall t \in [0,1],\ p(t) = (1-t).p_0 + t.p_1$$

La fonction g dont le graphe représente la coupe de la surface S par le plan P est paramétrée par la variable t :

$$\begin{aligned} g : [0,1] &\longmapsto [0,1] \\ t &\longrightarrow h\left(p(t)\right) = h\left((1-t).p_{0|xy} + t.p_{1|xy}\right) \end{aligned}$$

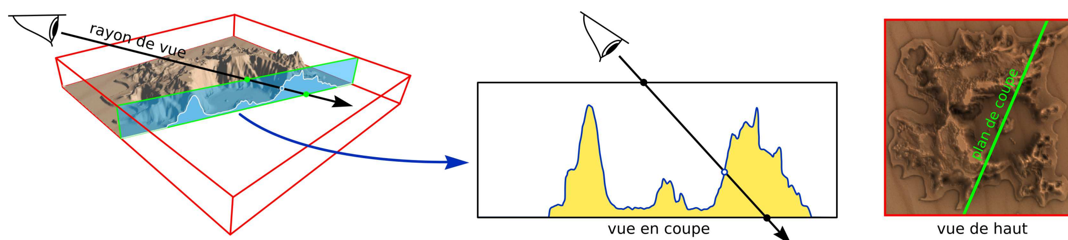


Figure 4.10 – La recherche de l'intersection d'un rayon de vue avec un relief peut être considérée en deux dimensions, dans le plan vertical contenant le rayon.

Le problème auquel on est ainsi ramené revient à déterminer si le segment σ d'équation

$$d(t) = (1-t).p_{0_{|z}} + t.p_{1_{|z}}$$

intersecte la fonction g et si c'est le cas chercher alors le paramètre t_{min} de la première intersection :

$$t_{min} = \min \{t \in [0,1] \; / \; d(t) = g(t)\}$$

Une fois ce point d'intersection trouvé il suffit d'en utiliser la projection $p_{|xy}(t_{min})$ dans le plan pour obtenir à l'aide d'un simple accès texture (*texture lookup*) les attributs (couleur, normale) attachés en ce point de la surface et effectuer les calculs d'éclairement nécessaires. Comme on dispose de l'information de position exacte du point d'intersection, on peut comme expliqué en section 3.2.1 calculer la profondeur exacte du fragment que l'on est en train de traiter. Ceci est nécessaire uniquement dans les cas où d'autres objets que la surface de relief elle-même peuvent se trouver à l'intérieur de l'enveloppe englobante.

Nous exposons maintenant les méthodes numériques classiques permettant de calculer l'intersection d'une droite avec le graphe d'une fonction quelconque, afin de montrer quelles approches sont adaptées au calcul sur GPU et enfin décrire quelles ont été nos contributions pour obtenir des algorithmes robustes et efficaces.

4.2.3 Intersection d'une droite avec le graphe d'une fonction

Pour simplifier encore l'énoncé du problème, observons qu'il peut être exprimé comme un problème classique de recherche de racines sur un intervalle borné d'une fonction $f = d - g$ continue quelconque :

$$f(t) = 0 \iff d(t) = g(t) \iff p(t) = h(t)$$

Trouver une solution numérique à cette question d'analyse numérique (*numerical root finding* dans la littérature scientifique en anglais) a été considérablement étudié [PTVF07] et a donné lieu à des algorithmes qui pour certains datent de l'antiquité.

Selon les propriétés connues de la fonction f, il existe des algorithmes plus ou moins efficaces. Notamment un grand nombre de techniques s'attache au problème spécifique des fonctions polynomiales en s'appuyant sur leurs propriétés algébriques.

Dans notre cas, la seule propriété connue de la fonction à inverser est qu'elle est continue. C'est la propriété minimale généralement requise par la plupart des méthodes numériques de recherche de racines, pour pouvoir s'appuyer sur le théorème des valeurs intermédiaires :

Théorème (Théorème des valeurs intermédiaires)**.** *Soit f une fonction réelle continue définie sur un intervalle $I \subset \mathbb{R}$ et a et b deux éléments de I. Alors pour tout réel k compris entre $f(a)$ et $f(b)$ il existe au moins un réel c compris entre a et b tel que $f(c) = k$.*

Autrement dit dans notre cas si l'on connaît un intervalle $[a, b]$ tel que $f(a) < 0$ et $f(b) > 0$ (ou inversement) alors on est sûr qu'il existe une valeur $c \in [a, b]$ telle que $f(c) = 0$. Les algorithmes de recherche de zéro d'une fonction supposent généralement la donnée d'un tel intervalle comme point de départ.

L'algorithme le plus connu est la **méthode de la bissection**, méthode dichotomique qui consiste à maintenir un intervalle $[a, b]$ dont on est sûr qu'il encadre une racine et une seule. En partant d'un intervalle qui satisfait cette propriété (c'est-à-dire tel que $f(a)$ et $f(b)$ soient de signes opposés), on évalue la fonction au milieu de l'intervalle (en $c = \frac{a+b}{2}$) et on sélectionne le demi-intervalle dont on est sûr qu'il contienne une racine, par test du signe de $f(c)$. Par exemple si $f(c)$ est du même signe que $f(a)$, on sélectionne le sous-intervalle $[c, b]$ en prenant c comme nouvelle valeur de la borne inférieure a de l'intervalle de recherche ($a \leftarrow c$). Ce processus est réitéré jusqu'à ce que la taille de l'intervalle de recherche devienne plus petite qu'un seuil de précision ϵ fixé (dès que $b - a < \epsilon$). L'algorithme est illustré en figure 4.11 (à gauche). Cet algorithme est connu pour converger à une vitesse linéaire puisque l'intervalle de recherche est simplement divisé par 2 à chaque itération (la précision augmente d'un bit par itération), il est donc considéré comme l'un des plus lents mais aussi l'un des plus robustes puisqu'il converge à coup sûr.

Algorithme 4.1 : Méthode de la bissection

Entrées : a, b tels que $a < b$, $f(a) < 0$ et $f(b) > 0$
Sorties : une racine c de la fonction f, approchée à ϵ près
tant que $b - a > \epsilon$ **faire**
  $x \leftarrow (b - a) \div 2$
  si $f(x) > 0$ **alors**
    $b \leftarrow x$
  sinon
    $a \leftarrow x$
$c \leftarrow a$

La méthode de **regula falsi** ou *méthode de la fausse position* est une amélioration de cet algorithme qui permet d'obtenir une vitesse de convergence super-linéaire (c'est-à-dire mieux que n'importe quelle méthode qui converge linéairement mais moins bien qu'une vitesse d'ordre quadratique par exemple) tout en restant robuste, contrairement à la *méthode de la sécante* (avec laquelle elle est parfois confondue), qui elle possède une vitesse de convergence d'ordre 1.6 environ (plus précisément le nombre d'or) si l'intervalle de départ encadre suffisamment bien la racine recherchée mais ne permet pas de garantir la convergence. C'est une méthode dichotomique où, comme pour la méthode de bissection, un intervalle contenant une racine est itérativement raffiné, à la différence près que le point utilisé pour subdiviser l'intervalle n'est plus le point mileu, mais le point d'intersection du segment $[(a, f(a)); (b, f(b))]$ avec l'axe des abscisses. Cela revient à approximer localement la fonction f par une droite et permet de converger plus rapidement vers la solution recherchée. L'algorithme est illustré en figure 4.11 (à droite).

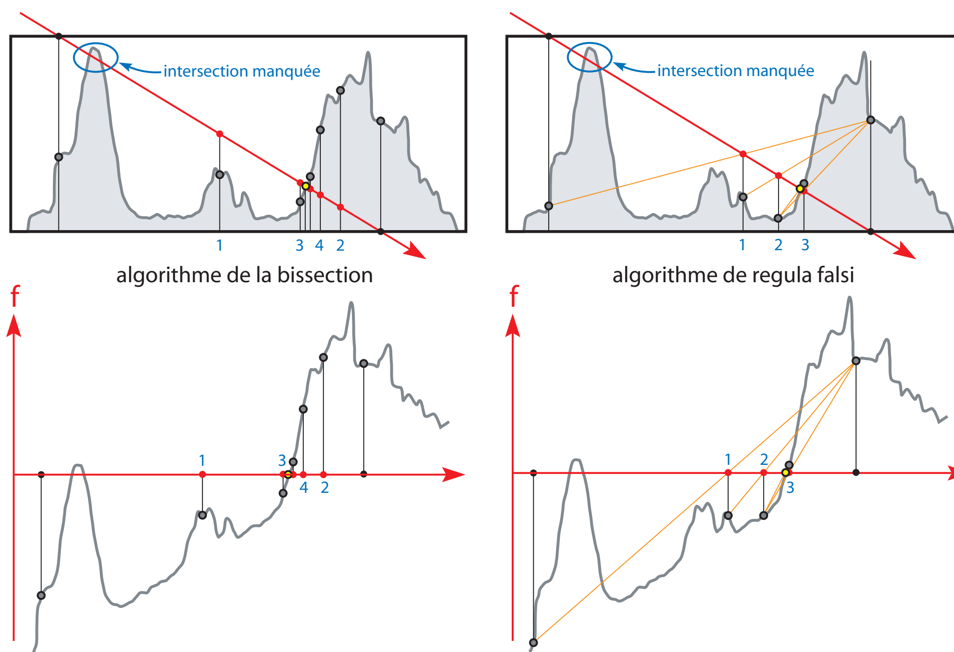


FIGURE 4.11 – Recherche de racine par dichotomie : méthode de la bissection (à gauche) et méthode de regula falsi (à droite).

Algorithme 4.2 : Méthode de regula falsi

Entrées : a, b tels que $a < b$, $f(a) < 0$ et $f(b) > 0$
Sorties : une racine c de la fonction f, approchée à ϵ près
tant que $b - a > \epsilon$ **faire**
  $x \leftarrow (a \times f(b) - b \times f(a)) \div (f(b) - f(a))$
  si $f(x) > 0$ **alors**
    $b \leftarrow x$
  sinon
    $a \leftarrow x$
$c \leftarrow a$

D'autres techniques plus évoluées existent mais elles s'appuient sur des itérations plus complexes et se prêtent mal à une implémentation efficace sur GPU, interdisant l'utilisation de branchements et nécessitant des itérations simples. Notamment la méthode de Brent [Bre73, PTVF07], généralement considérée comme la plus efficace des méthodes existantes, combine plusieurs techniques dont la bissection et la méthode de regula falsi, et choisit à chaque itération laquelle est la plus appropriée à utiliser.

Mentionnons enfin que la méthode de Newton-Raphson (qui utilise une approximation linéaire locale de la fonction à chaque itération), populaire pour sa simplicité et pour sa vitesse de convergence quadratique, n'est pas utilisable dans notre cas puisqu'elle ne permet pas de garantir la convergence.

Particularités de notre problème

Nous cherchons un algorithme qui puisse être executé par le processeur graphique, ce qui pose plusieurs contraintes :

– l'algorithme doit être robuste, c'est-à-dire assurer la convergence;
– le nombre d'itérations doit rester faible et idéalement constant;
– chaque itération doit être simple et sans branchement pour permettre la parallélisation.

La première particularité de notre problème est qu'on ne dispose pas comme donnée de départ d'un intervalle dont on soit sûr qu'il encadre la racine que l'on cherche. En fait on a généralement plusieurs racines, et parmi ces racines c'est la plus petite qui nous intéresse (la valeur t minimale telle que $f(t) = 0$). C'est important car rater la première racine génère des erreurs de rendu (artefacts) très saillants, comme l'apparition de trous sur la surface (voir figure 4.12). On a donc besoin avant tout d'une technique permettant de déterminer un intervalle contenant la première racine.

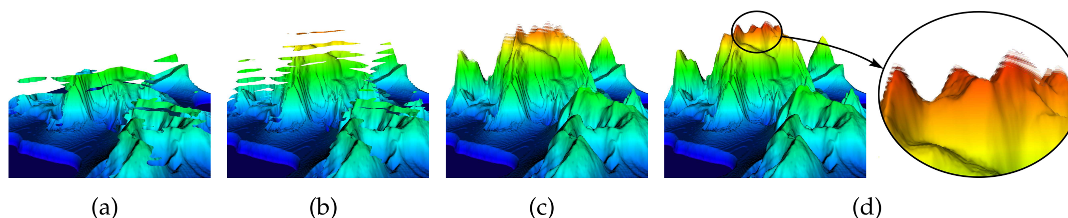
La deuxième particularité vient du fait que l'on se place dans un contexte de rendu en temps-réel, qui nécessite d'être capable d'exécuter l'algorithme d'intersection un grand nombre de fois à la seconde (une fois par trame rendue et par pixel occupé par le relief, soit 50 millions de requêtes à la seconde dans le cas du rendu à une fréquence de $100Hz$ d'un relief occupant une surface de 800×600 pixels à l'écran). Dans le cas où le relief est statique (c'est-à-dire quand h est indépendant du temps), c'est toujours sur la même surface que sont effectuées les requêtes d'intersection, il est donc intéressant de se demander comment *prétraiter* ce relief, hors de l'application de rendu temps-réel (*off-line*), pour pouvoir ensuite au moment du rendu disposer d'informations permettant d'accélérer le calcul d'intersection. La question se pose alors de savoir quelle information précalculer et comment la stocker, sans impliquer une augmentation significative de la taille de stockage en mémoire de la représentation.

Enfin la troisième particularité est que la précision numérique requise dans notre cas est relativement faible par rapport à celle qui est habituellement recherchée par les applications de calcul numérique : elle dépend de la résolution à laquelle le relief sera rendu et de la résolution de la carte de hauteur. C'est un avantage, permettant d'avoir relativement peu d'itérations à effectuer quel que soit l'algorithme utilisé. Ainsi la notion de vitesse asymptotique de convergence n'est pas un critère déterminant dans notre cas.

Pour palier au principal défaut des travaux existants sur le rendu de relief sur GPU, nous nous sommes attachés à chercher des algorithmes qui soient exempts d'erreurs de rendu (artefacts), notamment au niveau de la visibilité : on veut obtenir la garantie de toujours trouver la première intersection avec le relief, quelle que soit la finesse des détails qu'il contient.

Deux étapes nécessaires

Parmi les méthodes de recherche de racine que nous venons de présenter, celles qui permettent d'assurer la convergence et ainsi déterminer une racine de manière robuste, supposent toutes comme donnée préalable de connaître un intervalle dont on possède la garantie qu'il contient une racine. De plus, seule la garantie supplémentaire d'unicité d'une racine permet d'obtenir un résultat unique (autrement dit si plus d'une racine existent, aucun algorithme n'est capable de déterminer la



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

Figure 4.12 – Artefacts produits au rendu lorsque des intersections sont ratées : (a) pas de recherche d'intervalle (application directe de la bissection); (b) recherche de l'intervalle par parcours itératif en échantillonnat le rayon en 10 positions; (c) de même avec 100 échantillons; (d) de même avec 500 échantillons : les erreurs persistent.

première à coup sûr). Le problème doit donc être décomposé en deux étapes successives : d'abord déterminer un intervalle contenant la première racine que l'on cherche, ensuite calculer cette intersection précisément.

Ces deux étapes nécessaires au calcul exact d'intersection d'un rayon avec un champ de hauteur sont généralement mal identifiées dans les travaux existants sur le rendu de champ de hauteur. Voyons pour chacune d'elles l'ensemble des solutions qui peuvent être adoptées.

Recherche de l'intervalle englobant

Aucune recherche La première manière d'aborder le problème de la recherche d'un intervalle englobant la première intersection consiste simplement de ne pas le traiter ! Plusieurs travaux existants ignorent simplement cette étape [SKALP05] et appliquent directement un algorithme de recherche d'intersection sans s'être assuré d'en encadrer une et une seule. Cela peut génerer des artefacts généralement importants.

Parcours à pas constants Si l'on se restreint au cas où on ne veut (ou peut) pas utiliser d'information précalculée sur la fonction de hauteur, il n'existe pas de méthode exacte permettant de déterminer un intervalle contenant la première racine. Il faut donc procéder à un parcours itératif le long du rayon (*ray marching*), consistant à échantillonner, généralement uniformément (à pas constants), l'intervalle de recherche et évaluer la fonction f (différence de hauteur entre la surface et le rayon de vue) en ces positions échantillonnées prises dans l'ordre croissant pour déterminer à quel moment la fonction change de signe. Appelons $(t_i)_{i\in\mathbb{N}}$ la suite ordonnée de paramètres correspondant aux positions des pas effectués le long du rayon, et appelons k l'indice du premier pas effectué provoquant un changement de signe sur la fonction f, c'est-à-dire tel que $f(t_{k-1}).f(t_k) < 0$. On sait alors par le théorème des valeurs intermédiaires qu'il existe (au moins) une racine dans l'intervalle $[t_{k-1}, t_k]$, mais l'échantillonnage discret effectué ne permet pas de garantir que c'est la première.

Une solution évidente pour tenter de ne pas rater d'intersection consiste à augmenter la fréquence d'échantillonnage (autrement dit utiliser des pas plus fins). Rendre ainsi un relief comprenant des détails fins peut nécessiter un grand nombre

de pas pour obtenir un rendu correct, sans pour autant prévenir toute erreur. En fait quelles que soient la complexité du champ de hauteur et la méthode de d'interpolation utilisée pour la carte de profondeur, il n'existe pas d'échantillonnage régulier (quelle que soit la fréquence utilisée) qui fournisse la garantie de toujours trouver un intervalle contenant la première intersection. Malgré les défauts inhérents à une telle approche, c'est la plus généralement utilisée par les techniques de rendu de relief [HEGD04, POC05, OP05, dTWL08, RSP07, Tat06].

Comme nous le verrons ensuite (section 4.2.4), dans le cas où la fonction de hauteur est échantillonnée sur une grille régulière et interpolée bilinéairement, comme c'est généralement le cas, il existe une façon non uniforme d'échantillonner le rayon qui garantisse de toujours trouver un tel intervalle [QQZ+03, HS04].

Utilisation de données précalculées Échantillonner uniformément l'intervalle de recherche requiert potentiellement beaucoup d'itérations avant d'atteindre la surface, et ne permet pas de garantir l'exactitude de la recherche de la première racine. Pour s'affranchir de ces obstacles et obtenir un algorithme exact et efficace, la seule option est de disposer d'informations supplémentaires sur la fonction de hauteurs, qui permettront de parcourir rapidement les zones ne contenant pas de racines (les zones de vide) tout en s'arrêtant là où des détails fins ont des chances d'intersecter le rayon lumineux.

Idéalement on aimerait pouvoir connaître en n'importe quel point de l'espace, dans n'importe quelle direction, la distance qu'il faut parcourir jusqu'à la surface du relief, information qui peut être représentée par la fonction à cinq dimensions suivante (en prenant la convention $\min \emptyset = +\infty$) :

$$\begin{array}{rccl} \delta : & [0,1]^3 \times \mathbb{S}^2 & \longrightarrow & \mathbb{R}^+ \cup \{+\infty\} \\ & (p,d) & \longmapsto & \min \left\{ t \in \mathbb{R}^+ \ / \ p + t.d \in S \right\} \end{array}$$

Une telle donnée permet pour n'importe quel rayon lumineux d'obtenir directement le point d'intersection recherché sans effectuer de parcours le long du rayon. La complexité de cette fonction pour un relief quelconque, notamment son aspect fortement discontinu, rend illusoire la possibilité de la représenter avec peu de paramètres.

Toutes les approches de rendu de relief existantes ayant recours à un précalcul peuvent être considérées comme des manières plus ou moins approchées d'encoder cette fonction δ. La technique proposée par les *Generalized Displacement Maps* [WTL+04] est celle qui tente d'approcher au mieux cette fonction, par une stratégie d'échantillonnage dense selon ses cinq dimensions, suivie d'une méthode de compression efficace. Cependant malgré les bons taux de compression, la taille occupée en mémoire reste très importante (4 Mo pour une carte de hauteur de taille 128×128, soit 256 fois plus que la place occupée par cette seule carte de hauteur). L'approche utilisée par les *View-dependent Displacement Maps* [WWT+03] est similaire, la dimension z en moins (seuls les rayons provenant de l'extérieur du volume de relief sont considérés, ce qui permet cette simplification).

Pour réduire la dimensionnalité du problème, Donnelly [Don05] enlève la composante directionnelle de la fonction δ en ne conservant pour chaque point du

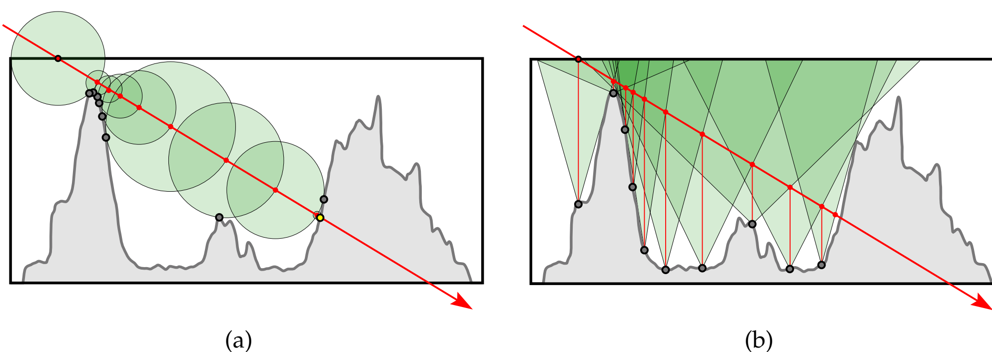


FIGURE 4.13 – Utilisation de données précalculées pour avancer efficacement le long du rayon sans rater d'intersection : (a) sphere tracing; (b) cone tracing

volume de rendu que la distance minimale prise dans toutes les directions :

$$\begin{array}{rccl} \tilde{\delta}: & [0,1]^3 & \longrightarrow & \mathbb{R}^+ \cup \{+\infty\} \\ & p & \longmapsto & \min_{d \in \mathbb{S}^2} \delta(p,d) \end{array}$$

Ceci oblige alors à effectuer plusieurs pas le long du rayon par une approche de *sphere tracing* [Har96] (voir figure 4.13(a)), et il faut échantillonner et stocker la fonction de distance dans une texture 3D ce qui représente encore un surcoût mémoire non négligeable.

Pour que le coût mémoire additionnel impliqué par le stockage d'un précalcul soit du même ordre que la taille qu'occupe en mémoire la carte de hauteur, il semble nécessaire de recourir à une information 2D. La solution la plus directe consiste à associer une ou plusieurs valeurs supplémentaires à chaque texel de la carte de hauteur. Prendre simplement le minimum de la fonction $\tilde{\delta}$ sur l'axe des z ne convient pas car la valeur obtenue serait uniformément nulle.

Paglieroni [PP94, Pag98] propose de stocker au dessus de chaque texel un cône inversé d'espace vide maximal, pouvant être représenté par peu de paramètres (un angle dans le cas le plus simple). De manière similaire, Kolb et Rezk-Salama [KRS05] proposent de précalculer un ou plusieurs cylindres d'espace vide au-dessus de chaque texel. Cela permet d'effectuer de grands pas lors du parcours du rayon sans occasionner de surcoût mémoire important (voir figure 4.13(a)).

Enfin notons que toutes ces approches demandent des précalculs coûteux, qu'il n'est pas possible de réaliser en temps-réel, empêchant leur utilisation pour rendre un relief dynamique (une surface d'eau par exemple). Une solution est proposée par Tevs et al. [TIS08], qui utilise une hiérarchie de volumes englobants calculée dynamiquement, ce qui est rendu possible par la structure de la carte de hauteur (les volumes englobants sont obtenus en calculant récursivement les hauteurs maximum entre texels voisins). Une approche similaire pour un relief statique seulement avait été donnée par Schroders et Gulik [SG06] et Oh et al. [OKL06]. Le parcours demandé est par contre plus compliqué que pour les approches statiques, par la nécessité de contourner l'absence de récursivité qui caractérise le processeur graphique.

Calcul précis de la racine Supposons maintenant disposer d'un intervalle $[a, b]$ contenant la première racine (et elle seule) que l'on recherche.

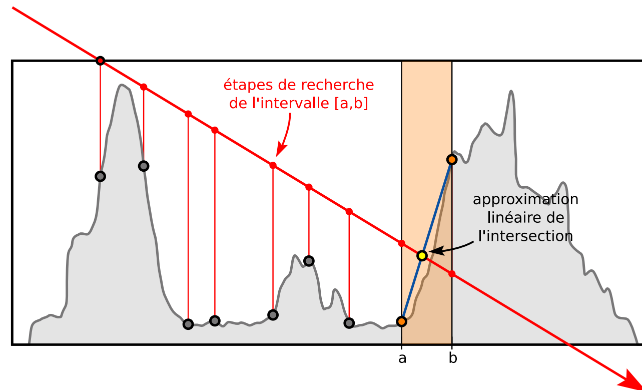


FIGURE 4.14 – Approximation linéaire sur l'intervalle $[a, b]$ de l'intersection recherchée.

Pas de calcul précis De la même façon que certaines techniques ne se préoccupent pas de la première étape de recherche de l'intervalle englobant la première racine, certaines n'effectuent pas le calcul exact de la racine. C'est souvent le cas pour les algorithmes qui utilisent un parcours à pas fixes et qui conservent simplement le dernier point échantillonné (le premier à provoquer un changement de signe de la fonction f) comme approximation de la racine recherchée. Cela peut générer des erreurs de rendu où le relief apparaît comme formé de tranches horizontales (artefacts de *slicing*) qui ne peuvent être atténuées qu'en diminuant la taille des pas effectués.

Approximation linéaire Lorsque l'intervalle englobant la racine recherchée est suffisamment petit par rapport aux variations que contient le champ de hauteur, une approximation linéaire locale de la fonction de hauteur permet d'obtenir une bonne estimation de la valeur que l'on cherche à calculer. Cela revient à prendre comme approximation t de la racine :

$$t = \frac{a.f(b) - b.f(a)}{f(b) - f(a)}$$

Cette approximation est correcte à condition de s'être assuré que la fonction f peut être supposée linéaire sur l'intervalle $[a, b]$. On peut noter que cela revient à effectuer une itération de la méthode de regula falsi.

Recherche dichotomique Dans le cas général où l'on ne peut pas faire d'hypothèse sur la régularité de la fonction à l'intérieur de l'intervalle $[a, b]$, la seule solution est de recourir à une méthode numérique itérative de recherche de racine. La méthode de la bissection répond bien à nos besoins : c'est une méthode simple et robuste qui garantit la convergence vers la racine recherchée en peu d'itérations. On peut même déterminer à l'avance le nombre d'itérations nécessaires pour satisfaire un seuil de tolérance d'erreur fixé, puisqu'on connait la taille exacte δ_n de l'intervalle de recherche après n itérations :

$$\delta_n = 2^{-n}(b - a)$$

donc si ϵ est l'erreur maximum que l'on s'autorise, le nombre n_ϵ d'itérations nécessaire pour obtenir cette précision vaut :

$$n_\epsilon = \left\lceil \log_2 \left(\frac{b-a}{\epsilon} \right) \right\rceil$$

La méthode de regula falsi qui peut être vue comme une variante plus efficace est aussi une bonne option puisque les itérations nécessaires restent simples et comme avec la dichotomie la convergence est assurée. Le meilleur choix entre ces deux méthodes n'est pas décidable *a priori* car bien que la méthode de regula falsi permette, pour une précision visée d'obtenir une solution en moins d'itérations qu'avec la méthode de la dichotomie, le calcul effectué à chaque itération est en revanche légèrement plus coûteux. Comme ces algorithmes seront executés par le processeur graphique, pour lequel le compilateur effectue de nombreuses optimisations parfois délicates à anticiper, il est difficile de déterminer quelle méthode est la plus efficace du point de vue théorique. D'autre part le calcul précis de la racine n'est qu'une partie du calcul d'intersection, et c'est généralement la moins coûteuse (l'autre partie étant la recherche d'un intervalle englobant la première racine).

Il est important de noter ici que l'utilisation de ces techniques de recherche numérique de racine n'est correcte qu'à condition de s'être auparavant assuré que l'intervalle considéré contient une et une seule racine, point généralement ignoré par la plupart des méthodes traitant du rendu de relief en temps-réel.

Conclusion

Nous venons de voir que pour obtenir un algorithme de rendu de champ de hauteur exempt d'artefacts, il faut pouvoir effectuer efficacement les deux étapes suivantes :

1. recherche d'un intervalle englobant la première intersection,
2. calcul précis de ce point d'intersection.

Ces deux étapes ne sont pas prises en compte correctement par les méthodes existantes, ce qui mène soit à des erreurs visuelles, soit à un coût de rendu inutilement élevé. Nous proposons deux approches pour répondre à ce manque. La première ne nécessite pas de précalcul et peut donc être utilisée pour des reliefs animés (c'est-à-dire dont la fonction de hauteur varie au cours du temps). La deuxième se restreint au cas de reliefs statiques mais propose un précalcul efficace permettant, au moment du rendu, d'atteindre l'intersection avec le relief en très peu d'itérations, tout en garantissant l'absence d'erreurs de rendu (aucune intersection avec la surface manquée).

4.2.4 Algorithme sans précalcul

L'algorithme que nous présentons ici ne requiert aucun précalcul et peut donc être utilisé pour rendre un champ de hauteur dynamique. Comme nous l'avons vu en section 4.1.3 il existe plusieurs manières d'interpoler les échantillons de la carte

de hauteur. L'algorithme que nous présentons ici s'applique à parcourir itérativement le rayon lumineux considéré en tenant compte de la manière dont la carte de hauteur a été échantillonnée, en l'occurence sur une grille régulière (carrée ou hexagonale). Selon la méthode d'interpolation utilisée, la fonction de hauteur reconstruite présente des caractéristiques exploitables pour déterminer la manière optimale (juste le nombre de pas nécessaire) et exacte (aucune intersection manquée) de la parcourir.

Certains détails de l'algorithme de parcours diffèrent selon la méthode d'interpolation choisie, nous commençons par le présenter dans le cas de l'interpolation bilinéaire, cas le plus utile en pratique, puis nous montrerons les changements à apporter dans le cas de l'interpolation de plus proche voisin puis et enfin dans le cas de l'interpolation linéaire des coordonnées barycentriques sur un réseau triangulaire. Le choix de ces trois méthodes d'interpolation provient d'une part de l'utilité pratique qu'ils peuvent présenter et d'autre part de leur bonne adéquation aux capacités du matériel graphique actuel (qui notamment permet des accès texture optimisés pour les interpolations de plus proche voisin et bilinéaire).

Interpolation bilinéaire

Comme rappelé en section 4.1.3, l'interpolation bilinéaire d'une carte de hauteur revient dans chaque intertexel à interpoler les quatre échantillons associés par la surface d'une quadrique. La surface globale ainsi définie est quadratique par morceaux. Si l'on se place dans le plan de coupe vertical P, la fonction g obtenue est aussi quadratique par morceaux, en effet, la coupe verticale du paraboloïde hyperbolique associé à un intertexel est une parabole.

Rappelons que l'intersection σ de notre rayon lumineux avec le volume de relief est paramétrée par un point d'entrée p_0, un point de sortie p_1 et une variable t balayant l'intervalle $[0,1]$: $p(t) = (1-t).p_0 + t.p_1$.

Considérons l'ensemble I_x des points d'intersection du segment σ avec les x-intercloisons, notées $(U_i)_{i \in [\![0,N-1]\!]}$, c'est-à-dire :

$$I_x = \bigcup_{i=0}^{N-1} \sigma \cap U_i = \left\{ p \in \sigma \ \ / \ \ p_{|x} \in \{ x_i \ / \ i \in [\![0,N]\!] \} \right\}$$

Appelons de la même manière I_y les points de σ qui appartiennent à des y-intercloisons. L'ensemble des points $I_x \cup I_y$ correspond à toutes les positions où le segment σ change d'intertexel, supposons qu'il y en a K et appelons $(t_k)_{0<k\leqslant K}$ la suite des paramètres ordonnés de ces points sur le segment σ :

$$0 \leqslant t_1 < t_2 < \ldots < t_K \leqslant 1 \quad \text{et} \quad \{ p(t_k) \ / \ 1 < k \leqslant K \} = I_x \cup I_y$$

La suite (t_k) représente une subdivision de l'intervalle $[0,1]$ telle que sur chacun des sous-intervalles la coupe g à une dimension de la fonction h est quadratique (son graphe est une parabole).

Nous procédons à l'approximation consistant à considérer que g est non pas quadratique mais linéaire sur chaque intervalle de la subdivision, autrement dit :

$$g_{|[t_k,t_{k+1}]}(t) = \frac{t_{k+1} - t}{t_{k+1} - t_k}.h(t_k) + \frac{t - t_k}{t_{k+1} - t_k}.h(t_{k+1})$$

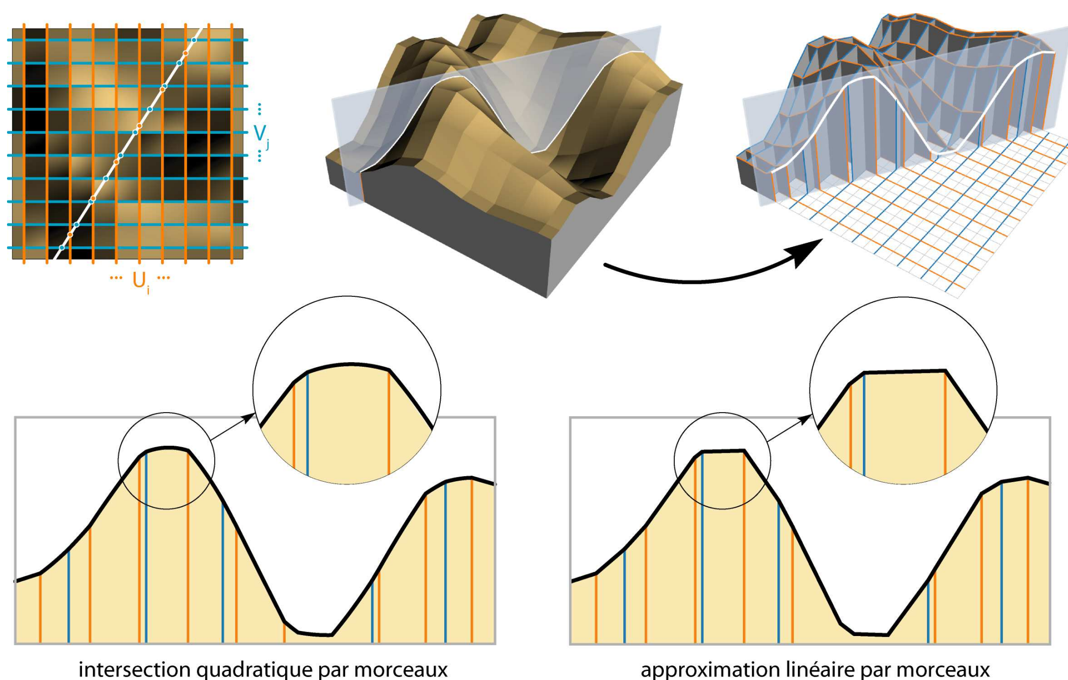


FIGURE 4.15 – La coupe verticale d'un relief obtenu par interpolation bilinéaire est une courbe quadratique par morceaux, dont l'approximation linéaire par morceaux est généralement très proche.

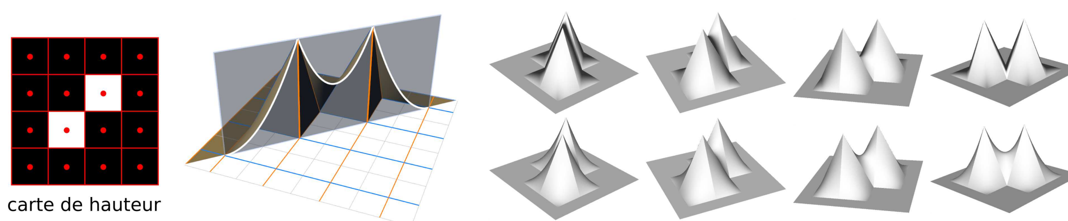


FIGURE 4.16 – Cas pathologique où l'erreur induite par l'approximation linéaire est la plus élevée, pour une carte de hauteur de résolution 4×4 : la série d'images de droite correspond aux rendus de la surface correspondante sous plusieurs angles de vue différents, avec l'approximation linéaire (en haut) et sans approximation (en bas).

Cette approximation est raisonnable car les morceaux de paraboles obtenus par coupe verticale d'une forme bilinéaire sont généralement très proches de segments de droites, et c'est seulement pour les vues de très près de cas pathologiques que la différence avec une vraie interpolation bilinéaire peut être perceptible. Le cas correspondant à l'erreur d'approximation la plus élevée est celui d'une selle de cheval très étirée (deux échantillons de hauteur minimale et deux autres de hauteur maximale, disposés en diagonale, voir figure 4.16).

On pourrait en fait éviter cette approximation et calculer l'intersection exacte du rayon lumineux avec les surfaces quadriques successivement rencontrées puisque le calcul analytique est suffisamment simple (cela reviendrait à calculer les racines d'un polynome du second degré, voir section 3.2.3). Cependant le surcoût non négligeable par rapport à l'approximation linéaire ne serait justifié que si la surface est vouée à être rendue de très près, de sorte que le morceau de surface associé à un intertexel occupe une place importante à l'écran (plusieurs dizaines de pixels au moins, mettons), mais c'est rarement le cas dans les applications pratiques. En effet

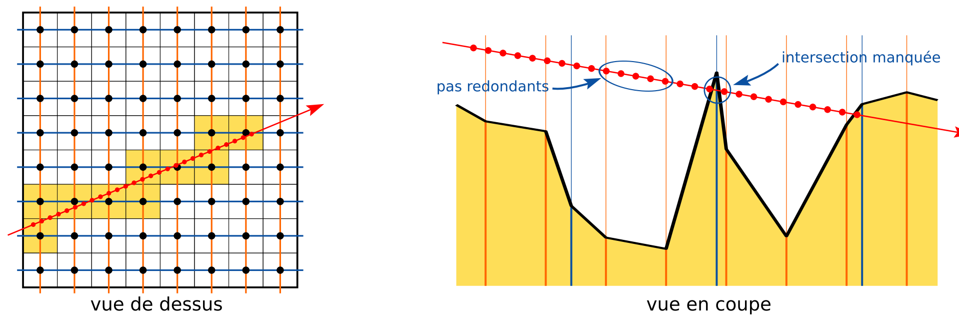


FIGURE 4.17 – Deux problèmes se posent lors du parcours à pas constant d'un rayon lumineux pour déterminer son intersection avec une fonction affine par morceaux : d'une part certains pas effectués sont redondants, d'autre part des intersections peuvent être ratées.

comme nous l'avons vu, le graphe d'une fonction quadratique par morceaux est continu mais pas C^1, ce qui implique, pour pouvoir représenter correctement une surface lisse, d'utiliser une fréquence d'échantillonnage suffisamment élevée pour que les erreurs d'approximation ne soient pas perceptibles. Ainsi un champ de hauteur échantillonné correctement ne présente pas de cas pathologiques comme celui de l'exemple montré figure 4.16.

Notons enfin que l'on peut se représenter cette approximation comme une modification de la méthode d'interpolation, dépendant continuement du point de vue. En effet, la surface reconstruite pour un point de vue donné est séparée selon deux segments en deux triangles et un morceau de paraboloïde hyperbolique, évoluant continuement en fonction de la position du point de vue.

Le problème se ramène donc à chercher la première intersection d'une droite avec une fonction continue affine par morceaux. Remarquons d'abord qu'un parcours par pas constants le long du rayon ne permet pas de résoudre ce problème correctement pour deux raisons :

1. ce n'est pas optimal en termes de nombre de pas : on est amené à effectuer beaucoup de pas sur un intervalle où la fonction reste linéaire;
2. cela n'est jamais exact : on n'a pas la garantie de ne jamais manquer un pic fin situé à une transition t_k.

On voit en fait intuitivement que ce sont aux transitions (t_k) qu'il faut échantillonner le rayon et effectuer la comparaison avec la fonction de hauteur. En effet, les fonctions continues affines par morceaux possèdent la propriété suivante :

Propriété (Propriété des fonctions affines par morceaux)**.** *Soit f une fontion réelle à une variable réelle continue affine par morceaux, soit $(t_k)_{0<k\leqslant K}$ la subdivision correspondante.*
Pour tout intervalle $[t_k, t_{k+1}]$ à l'intérieur duquel f s'annule,

$$f(t_k).f(t_{k+1}) \leqslant 0$$

Autrement dit, si l'on parcourt par ordre croissant les valeurs (t_k) et que l'on calcule chaque fois la différence de hauteur entre le point sur le rayon et la surface du relief prise à la verticale de ce point (autrement dit $p(t_k)_{|z} - g(t_k)$), tant que cette différence reste du même signe on est sûr de ne pas avoir raté d'intersection,

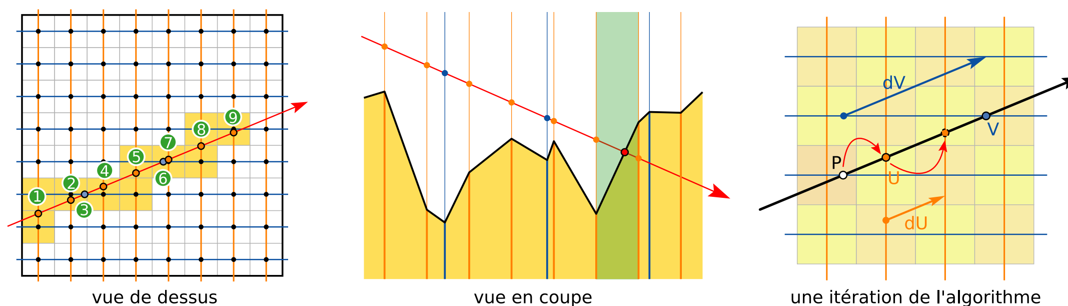


Figure 4.18 – Illustration de l'algorithme de parcours : l'image de droite montre une itération dans le cas $U_{|t} < V_{|t}$: P prend la place de U, qui est alors translaté d'une quantité dU.

et dès que cette différence change de signe, on sait qu'il existe une intersection (et une seule, sauf cas dégénéré) entre les deux derniers points visités. C'est la base de l'algorithme que nous proposons :

1. on parcourt les intercloisons rencontrées le long du rayon
2. à chaque itération on compare l'altitude du point considéré sur le rayon avec celle de la surface à la verticale de ce point
3. on itére tant que cette comparaison donne le même résulat
4. les deux derniers points obtenus définissent un intervalle englobant de la première intersection du rayon avec le relief

La clé de la technique consiste donc à pouvoir parcourir efficacement le rayon à toutes les positions où il intersecte les interbords, et ce dans le bon ordre. C'est un problème classique de traversée de grille régulière. Nous nous appuyons sur l'algorithme de parcours de grille de voxels proposé par Amanatides et Woo [AW87] qui se prête parfaitement à ce problème. Voici une manière légèrement différente de considérer le problème : on dispose de deux échantillonnages réguliers de l'intervalle de recherche (un pour les interbords selon l'axe des abscisses, un pour les interbords selon l'axe des ordonnées) et on veut entrecroiser ces deux échantillonnages de manière à ce que la liste de valeurs obtenues soit triée. Cela s'apparente à l'étape de fusion d'un tri fusion. Il suffit itérativement de choisir le plus petit élément parmi le premier de chaque liste pour l'en ôter et le placer en fin de la liste triée. Cela demande un phase d'initialisation pour positionner le premier point sur le premier interbord rencontré avant le point d'entrée du rayon dans le volume de relief, suivie d'une boucle faite d'itérations très simples. Le processus complet est décrit par l'algorithme 4.3, et est illustré par la figure 4.18

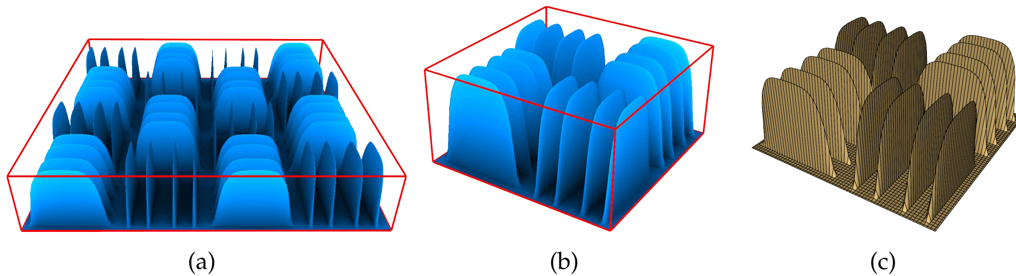
Le parcours s'arrête donc dès que le dernier point testé se retrouve sous la surface du champ de hauteur (puisque l'origine du rayon lumineux se trouve au dessus du relief), et alors on est assuré d'avoir cerné l'intersection recherchée, entre les deux derniers points testés, appelons les t_{n-1} et t_n.

Il reste à calculer précisément l'intersection avec la surface du relief. Comme on a supposé la fonction affine par morceaux entre les interbords, on peut appliquer directement le calcul de l'approximation linéaire détaillé précédemment. Connaissant $h(t_{n-1})$ et $h(t_n)$, on calcule le paramètre t correspondant à l'intersection :

$$t = \frac{t_{n-1}.f(t_n) - t_n.f(t_{n-1})}{f(t_n) - f(t_{n-1})}$$

Algorithme 4.3 : Algorithme d'intersection d'un rayon de vue avec le relief

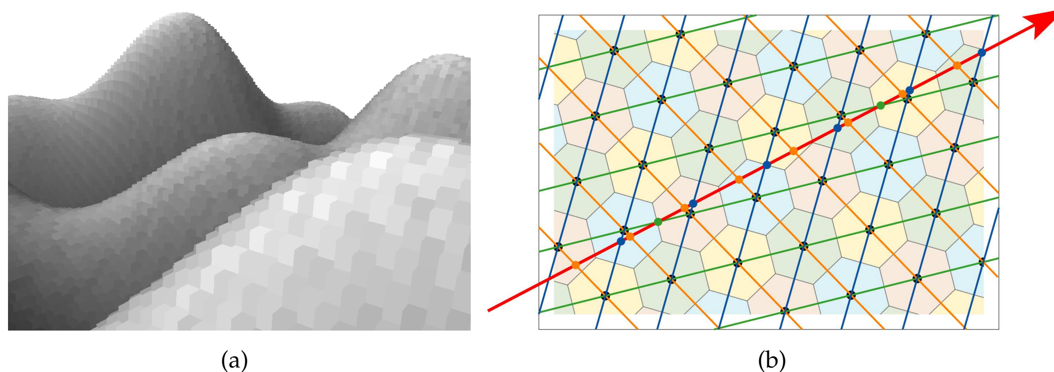
$p_0 \leftarrow$ point d'entrée du rayon dans le volume de relief, exprimé dans R_N
$p_1 \leftarrow$ point de sortie du rayon hors du volume de relief, exprimé dans R_N
$d \leftarrow p_1 - p_0$
$\delta_{|x} \leftarrow$ si $d_{|x} \geqslant 0$ alors 1 sinon 0
$\delta_{|y} \leftarrow$ si $d_{|y} \geqslant 0$ alors 1 sinon 0
$D_{|xyzt} \leftarrow (d, 1)$
$P_{0|xyzt} \leftarrow (p_0, 0)$
$dU_{|xyzt} \leftarrow D \div (N \times |d_{|x}|)$
$dV_{|xyzt} \leftarrow D \div (N \times |d_{|y}|)$
$U_{|xyzt} \leftarrow P_0 + \left|\delta_{|x} - \{0.5 - N \times p_{0|x}\}\right| \times dU$
$V_{|xyzt} \leftarrow P_0 + \left|\delta_{|y} - \{0.5 - N \times p_{0|y}\}\right| \times dU$
`// {x} dénote la partie fractionnaire de x`
$z_{relief} \leftarrow -1$
$D_{|xyzt} \leftarrow (0, 0, +\infty, 0)$
tant que $P_{|z} > z_{relief}$ *et* $P_{|t} \geqslant 1$ **faire**
  si $U_{|t} < V_{|t}$ **alors**
    $P \leftarrow U$
    $U \leftarrow U + dU$
  sinon
    $P \leftarrow V$
    $V \leftarrow V + dV$
  $z_{relief} \leftarrow \tilde{h}(P_{|xy})$
si $P_{|z} > z_{relief}$ **alors**
  aucune intersection trouvée
sinon
  intersection trouvée



(a) (b) (c)

FIGURE 4.19 – Robustesse de l'algorithme de rendu dans le cas de reliefs contenant des détails fins : (a) et (b) rendus obtenus avec la méthodes présentée; (c) mise en valeur des bordures de texels (carte de hauteur de résolution 64^2).

Le coût de rendu de cet algorithme est d'au plus $2N$ itérations. L'implémentation en GLSL de cet algorithme est donnée en annexe A. Les résultats obtenus sont rassemblés en fin de chapitre (section 4.2.7), pour pouvoir être comparés avec ceux produits par la méthode avec précalcul que nous allons maintenant décrire, ainsi qu'avec les autres méthodes existantes.



(a) (b)

FIGURE 4.20 – (a) Exemple de rendu de relief obtenu par interpolation de plus proche voisin; (b) Le parcours d'une grille triangulaire peut être effectué par application d'un algorithme similaire à celui présenté pour les grilles carrées. Il y a dans ce cas trois réseaux de droites parallèles à intersecter, ce qui est équivalent à parcourir une grille à trois dimensions.

Autres interpolations

Le cas de l'interpolation de plus proche voisin peut être traité par un algorithme similaire, à la différence près qu'il faut dans ce cas intersecter le rayon lumineux avec les cloisons au lieu des intercloisons. Un exemple de rendu est donné figure 4.20(a).

Le cas de l'interpolation barycentrique sur une grille triangulaire peut aussi être traité par un algorithme similaire (figure 4.20(b)). La différence étant dans ce cas qu'il faut cette fois intersecter le rayon avec trois réseaux d'intercloisons au lieu de seulement deux. La seule difficulté qui se pose alors est que pour ce troisième réseau d'intercloisons, correspondant aux diagonales dans la grille carrée correspondante, les valeurs données par l'interpolation bilinéaire ne produit pas les mêmes valeurs que l'interpolation barycentrique. Il n'est donc plus possible de recourir aux fonction d'interpolation de la carte graphique, il faut à chaque itération de l'algorithme aboutissant sur une diagonale, récupérer dans la carte de hauteur les deux échantillons encadrant la position courante et effectuer l'interpolation linéaire explicitement. Outre le surcoût en termes d'opérations algébriques et de *texture fetchs* (deux au lieu d'un), cela implique une disymétrie dans l'algorithme de parcours puisque ce sont seulement les cloisons diagonales qui posent problème. Nous n'avons pas poussé plus avant les recherches à ce sujet, mais il pourrait être intéressant de disposer d'un algorithme efficace permettant de rendre ce type de surfaces, de par les meilleurs propriétés d'échantillonnage présentées par rapport aux surfaces reconstruites par interpolation bilinéaire.

4.2.5 Algorithme avec précalcul

La technique que nous venons de présenter ne s'appuie sur aucune autre information que la donnée de la carte de hauteur, ce qui lui permet d'être utilisée pour rendre un champ de hauteur animé de manière interactive, dont on ne connaît pas à l'avance les variations dans le temps. Cependant de nombreuses applications font appel à des champs de hauteur statiques, situation qui présente l'avantage de permettre un éventuel pré-traitement en vue d'accélerer la recherche de la première

intersection tout en assurant l'exactitude du résultat produit (pas d'erreur de visibilté). Nous analysons d'abord quelles propriétés l'information précalculée doit nous permettre de garantir lors du parcours le long du rayon, pour ensuite proposer un précalcul induisant un faible surcoût mémoire et permettant d'accélérer considérablement le rendu.

Propriétés à garantir

S'autoriser à utiliser une information précalculée pour accélérer le calcul d'intersection ouvre la voie à un vaste champ de possibilités. La difficulté consiste à trouver le compromis optimal entre quantité de mémoire nécessaire pour stocker cette information et l'accélération induite pour le calcul d'intersection. Comme nous l'avons déja vu, la solution permettant de trouver l'intersection le plus directement consiste à précalculer un échantillonnage dense de l'ensemble des rayons lumineux intersectant le volume de relief et retenir pour chaque rayon son premier point d'intersection avec le champ de hauteur (par exemple les coordonnées 2D de ce point dans le plan de base du champ de hauteur ou plus simplement sa distance par rapport à l'origine du rayon, procédant ainsi à un échantillonnage de la fonction δ définie en section 4.2.3).

Rappelons que nous nous restreignons à l'étude du cas où le point de vue est situé hors du relief de rendu. Appelons *rayons extérieurs* les rayons lumineux dont l'origine se trouve hors du volume de relief, par opposition aux *rayons intérieurs* ayant leur origine dans le volume de relief. Deux rayons lumineux extérieurs d'origine différente mais de même droite de support (et de même sens) mènent toujours à la même intersection avec la surface du relief, puisqu'aucune occultation par le relief ne peut se produire hors de son volume. Cette relation d'équivalence permet de ne considérer que l'ensemble des classes d'équivalence entre rayons extérieurs (c'est-à-dire l'ensemble quotient pour cette relation d'équivalence), espace à quatre dimensions au lieu de cinq. Une manière simple de paramétrer cet espace consiste à utiliser une surface englobante convexe $\mathcal{B}$, une sphère ou un cube par exemple, et de paramétrer un rayon lumineux par ses deux points d'intersection (entrant et sortant) avec cette surface (pour être précis l'ensemble des rayons représentés ainsi n'est qu'un sous ensemble des rayons extérieurs, excluant une partie des rayons qui n'intersecte pas le volume de relief et qui ne nous intéressent donc pas). La fonction $\bar{\delta}$ qu'il faudrait échantillonner peut alors s'écrire comme suit :

$$\begin{aligned} \bar{\delta} : \mathcal{B} \times \mathcal{B} &\longrightarrow \mathbb{R}^+ \cup \{+\infty\} \\ (q_0, q_1) &\longmapsto \min\left\{t \in \mathbb{R}^+ \ / \ (1-t)q_0 + tq_1 \in S\right\} \end{aligned}$$

C'est une fonction à quatre paramètres réels puisque la surface $\mathcal{B}$ peut être paramétrée par deux valeurs (les coordonnées sphériques par exemple dans le cas d'une sphère). C'est une fonction liée à la visibilité, présentant de fortes discontinuités, au niveau des rayons tangents à la surface S, ce qui rend diffile son approximation par des fonctions paramétriques (analytiques ou algébriques). C'est pour cette raison que l'approche généralement utilisée pour tenter de représenter au mieux cette fonction consiste à en effectuer un échantillonnage dense [WWT+03, WTL+04]. Le principal problème qui se pose alors est d'arriver à stocker

efficacement l'importante quantité de données que cela représente, autrement dit un problème classique de compression de données. Le principal inconvénient de cette approche vient du fait qu'elle s'appuie sur un échantillonnage : elle ne permet de rendre le détail représenté qu'à une résolution plus basse que la fréquence d'échantillonnage, elle n'est d'ailleurs utilisée que pour rendre du micro-détail sur une surface.

Nous cherchons donc une technique moins coûteuse en mémoire et permettant un rendu à n'importe quelle échelle, quitte à nécessiter plusieurs itérations pour s'approcher de l'intersection recherchée.

La méthode que nous proposons s'inspire en partie de la technique du *sphere tracing* [Har96], qui s'appuie sur la possibilité d'évaluer en tout point de l'espace une borne inférieure de la distance à la surface que l'on cherche à rendre. À l'aide d'une telle information, permettant en tout point de l'espace de connaître une sphère maximale vide de matière, on peut avancer à grands pas vers l'intersection recherchée en s'assurant de ne pas la rater. Cette technique était vouée initialement au rendu de surfaces implicites, pour lesquelles certaines propriétés de la fonction analytique qui les définit sont exploitées pour déduire une borne sur la distance de n'importe quel point à la surface.

Algorithme 4.4 : Algorithme du *sphere tracing*

Entrées : p_0, p_1 intervalle de recherche de l'intersection
$d \leftarrow (p_1 - p_0) \div \|p_1 - p_0\|$
$q \leftarrow p_0$
$\delta \leftarrow +\infty$
tant que $\delta > \epsilon$ *et* $q \in [p_0, p_1]$ **faire**
  | $\delta \leftarrow distance(q, S)$
  | $q \leftarrow q + \delta.d$
si $\delta > \epsilon$ **alors**
  | intersection q trouvée
sinon
  | pas d'intersection

Cette technique a été appliquée au cadre du rendu sur GPU par Donnelly [Don05] qui précalcule cette information de distance dans une grille 3D. Cela permet d'obtenir un algorithme de rendu efficace mais n'est utilisable que pour des reliefs peu profonds, puisque l'échantillonnage dense selon l'axe vertical d'une texture 3D serait trop coûteux à stocker en mémoire (dans les exemples donnés par Donnelly, la fréquence d'échantillonnage utilisée verticalement est 16 fois moins élevée que la fréquence horizontale). D'autre part, le problème de l'interpolation d'une telle texture n'est pas abordé, or c'est un point important et problématique si la surface doit être observée de près. Notamment l'interpolation trilinéaire de valeurs de distance n'a pas de signification géométrique bien définie, alors que l'interpolation de plus proche voisin nécessiterait d'avoir précalculé pour chaque texel la distance minimale pour l'ensemble des points qu'il contient, pour délivrer une information correcte et utilisable.

Comme nous l'avons expliqué plus tôt, nous aimerions recourir à une information 2D pour ne pas augmenter considérablement la taille mémoire nécessaire pour

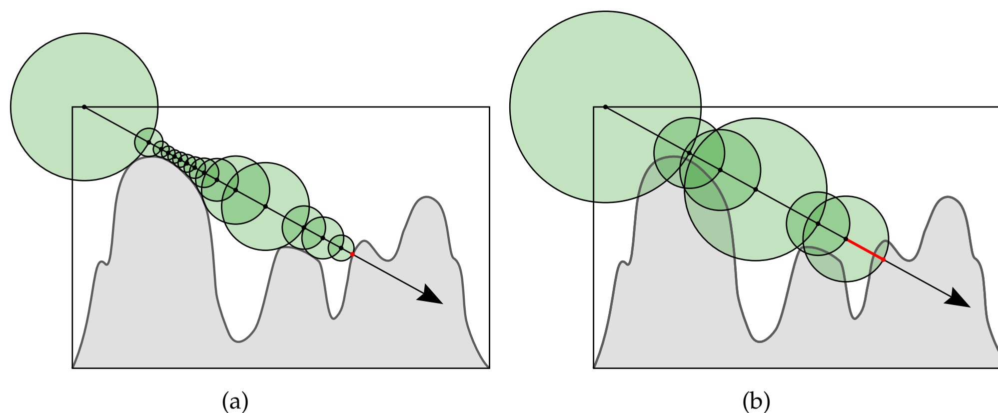


Figure 4.21 – Algorithme du *sphere tracing* : (a) technique classique utilisant la distance euclidienne; (b) accéleration par utilisation de la distance élargie : le nombre d'itération requises est considérablement réduit et les deux derniers points considérés définissent un intervalle contenant la première intersection recherchée (et elle seule).

représenter un relief. La solution simple consistant à rassembler toutes les valeurs de distance stockées à la verticale d'un texel, en une seule valeur en ne gardant que la plus petite, ne fonctionne pas car la valeur ainsi obtenue serait uniformément nulle.

Nous proposons d'utiliser non pas la distance à la surface la plus proche (autrement dit la distance de la première intersection rencontrée le long d'un rayon), comme dans le cas du *sphere tracing*, mais la **distance de la deuxième intersection avec la surface**. Nous appelons cette distance la *distance élargie*, que nous définissons d'abord sur l'ensemble des rayons lumineux :

Définition. *Soit S une surface continue, on définit la* distance élargie 5D, *notée ξ_S^5, entre cette surface et l'ensemble des rayons lumineux, comme suit :*

$$\begin{array}{rcccl} \xi_S^5 & : & \mathbb{R}^3 \times \mathbb{S}^2 & \longrightarrow & \mathbb{R}^+ \cup \{+\infty\} \\ & & (p,d) & \longmapsto & \sup \left\{ t \in \mathbb{R}^+ \ / \ \mathrm{Card}(]p, p+t.d] \cap S) < 2 \right\} \end{array}$$

On définit ensuite une version spatiale de cette distance en en prenant la borne inférieure sur l'ensemble des directions :

Définition. *Soit S une surface continue, on définit la* distance élargie 3D, *notée ξ_S^3, entre cette surface et l'ensemble des points de l'espace, comme suit :*

$$\begin{array}{rcccl} \xi_S^3 & : & \mathbb{R}^3 & \longrightarrow & \mathbb{R}^{+*} \\ & & p & \longmapsto & \inf_{d \in \mathbb{S}^2} \xi_S^5(p,d) \end{array}$$

Il est facile de voir que cette distance est partout supérieure à la distance euclidienne (entre un point et une surface) utilisée par le *sphere tracing*, d'où l'appelation de distance élargie.

Posséder une telle information permet d'avancer le long d'un rayon en s'assurant de ne pas traverser la surface du relief plus d'une fois (on autorise à passer

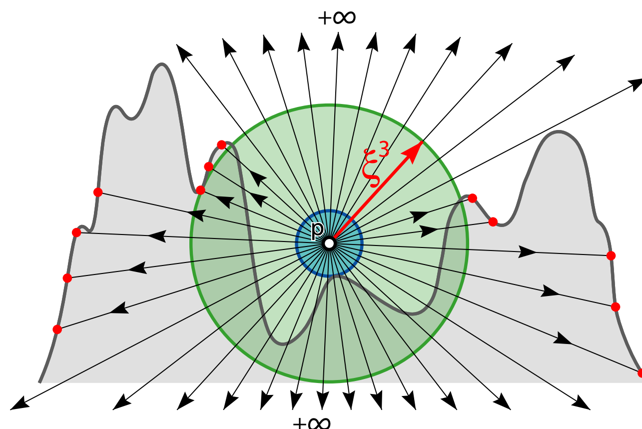


FIGURE 4.22 – Calcul en un point p de la distance élargie 5D pour un ensemble de rayons lumineux. La distance élargie 3D (dénotée par le cercle vert) est déduite en conservant la plus petite de ces distances. La distance euclidienne du point p à la surface est dénotée par le cercle bleu.

sous la surface mais pas à en ressortir). Parcourir le rayon en avançant par pas de taille inférieure à la distance élargie permet de rapidement déterminer un intervalle englobant la première intersection recherchée (avec la garantie de ne jamais la rater). Le point précis d'intersection peut ensuite être déterminé efficacement en appliquant un algorithme de recherche rapide de racine. Cela peut se comprendre grace à la propriété suivante, qui montre bien l'information qu'apporte la possession d'une borne inférieure de la distance élargie :

Propriété. *Soit S une surface continue et soient $p, q \in \mathbb{R}^3$ tels que*

$$d(p,q) < \xi_S^5(p, \overrightarrow{pq})$$

Alors,
- *soit le segment $[p,q]$ traverse la surface S exactement une fois,*
- *soit p et q se trouvent du même coté de la surface.*

Donc par exemple, si l'on dispose au moment du rendu de la donnée de la fonction élargie 3D, qui fournit une borne inférieure de la distance élargie 5D, on peut déterminer la première intersection d'un rayon d'origine $p \in \mathbb{R}^3$ et de direction $d \in \mathbb{S}^2$ avec une surface S à l'aide de l'algorithme suivant :

Algorithme 4.5 : Algorithme de parcours utilisant la distance élargie

Entrées : rayon lumineux de paramètres (p, d)
$q_1, q_2 \longleftarrow p$
tant que $h(q_2) < q_{2|z}$ *et* $q_2 \in$ *volume englobant de* S **faire**
  $q_1 \longleftarrow q_2$
  $q_2 \longleftarrow q_2 + \xi_S^3(q_2).d$
si $h(q_2) \geqslant q_{2|z}$ **alors**
  l'intervalle $[q_1, q_2]$ contient l'intersection
sinon
  on est sorti du volume, il n'y a pas d'intersection

L'algorithme peut se reformuler comme suit : chercher le plus rapidement possible (par grands pas) un point sous le relief en s'assurant de ne le traverser qu'une

seule fois, puis calculer l'intersection exacte en utilisant un algorithme rapide comme la dichotomie.

La distance élargie utilisée pour chaque pas permet de réduire considérablement le nombre d'itérations nécessaires pour la première étape par rapport à une approche de *sphere tracing*, notamment pour les rayons proches des silhouettes pour lesquels cette dernière se voit fortement ralentie par cette proximité qui implique des sphères de petits rayons.

Le rayon de sûreté

Pris tel quel, l'algorithme que nous venons de décrire nécessiterait d'échantillonner et stocker la fonction de distance ξ_S^3 dans une grille 3D, ce que l'on cherche justement à éviter.

Réduisons encore d'une dimension la fonction de distance élargie, en ne considérant que le point du volume de relief se trouvant au dessus de la surface :

Définition. *Soit S la surface d'un champ de hauteur h, on définit la* distance élargie 2D, *notée ξ_S^2, entre cette surface et l'ensemble des points du plan, comme suit :*

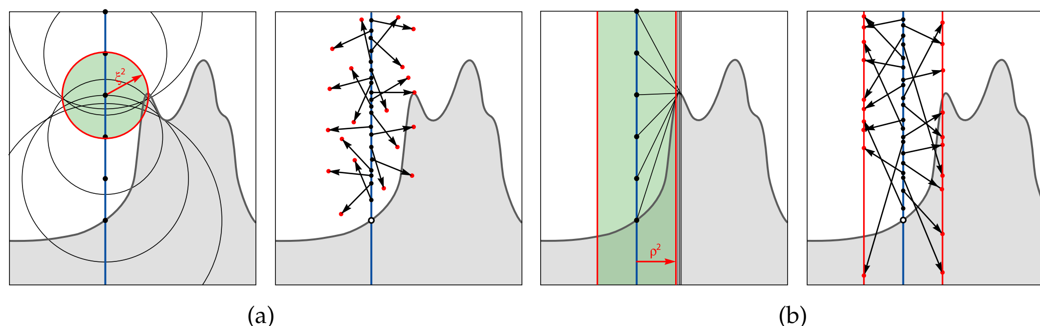
$$\begin{array}{rcccl} \xi_S^2 & : & [0,1]^2 & \longrightarrow & \mathbb{R}^+ \\ & & (x,y) & \longmapsto & \inf\limits_{z \in [h(x,y),1]} \xi_S^3(x,y,z) \end{array}$$

Le problème rencontré avec la distance euclidienne pour passer à un échantillonnage 2D, qui était qu'en prenant la valeur minimale à la verticale d'un texel on obtenait partout la valeur zéro, ne se pose plus avec la distance élargie 2D : elle ne s'annule que rarement à la surface du relief, et son caractère élargi lui permet de conserver une valeur élevée même après réduction à deux dimensions. Cependant comme on peut le voir sur la figure 4.23, dans le cas particulier des reliefs, l'ensemble des distances élargies 3D mesurées à la verticale d'un point du plan (x,y) pour, en en gardant la plus petite, obtenir une distance élargie 2D, peut avoir une variance élevée. Alors qu'au contraire, si l'on considère non plus la distance euclidienne spatiale mais la distance euclidienne dans le plan (x,y), les valeurs mesurées sont généralement beaucoup plus similaires. Lorsque l'on se trouve sur un rayon lumineux non vertical, connaître la distance projetée dans le plan d'un point du rayon est équivalent à connaître sa distance spatiale, on peut donc dans notre cas utiliser indiféremment l'une ou l'autre. Nous définissons alors la *distance projetée élargie* :

Définition. *Soit S une surface continue, on définit la* distance projetée élargie 5D, *notée ρ_S^5, entre cette surface et l'ensemble des rayons lumineux, comme suit :*

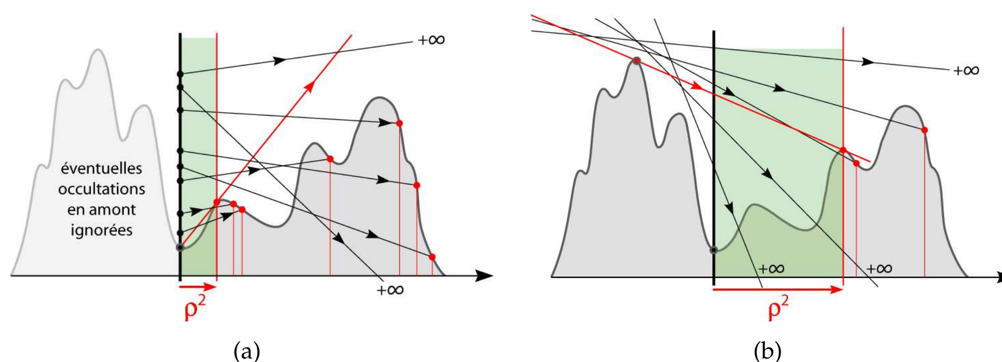
$$\begin{array}{rcccl} \rho_S^5 & : & \mathbb{R}^3 \times \mathbb{S}^2 & \longrightarrow & \mathbb{R}^+ \cup \{+\infty\} \\ & & (p,d) & \longmapsto & \|d_{|xy}\|_2 \cdot \xi_S^5(p,d) \end{array}$$

Les définitions des distances projetées ρ_S^3 et ρ_S^2 découlent ensuite de la même manière que pour ξ_S. C'est la distance projetée élargie 2D, ρ_S^2 que nous allons



(a) (b)

FIGURE 4.23 – Avantage de l'utilisation d'une distance projetée dans le plan pour la valeur de la distance élargie 2D : (a) calcul de ξ_S^2 comme le minimum de ξ_S^3 à la verticale d'un point du plan; (b) calcul de ρ_S^2 comme le minimum de ρ_S^3 à la verticale d'un texel. Les schémas de droite respectifs montrent comment l'information ξ_S^2 ou ρ_S^2 permet d'avancer le long d'un rayon quelconque : même si la valeur de ξ_S^2 peut être supérieure à celle de ρ_S^2, cette dernière permet généralement d'effectuer de plus grands pas.



(a) (b)

FIGURE 4.24 – Intérêt de la prise en compte des occultations pour le calcul du rayon de sûreté : calcul (a) sans et (b) avec prise en compte des occultations. Le rayon lumineux dénoté en rouge correspond au rayon limite définissant le rayon de sûreté.

utiliser, nous l'appelons le *rayon de sûreté* ou *safety radius* [BD06], et nous allons voir qu'en pratique, l'utilisation de cette quantité s'avère être un choix judicieux.

Notons ici un point important : comme nous l'avons précisé, par hypothèse nous ne considérons que les rayons lumineux extérieurs. C'est donc uniquement ces rayons qui doivent être pris en compte pour le calcul de la distance ρ_S^5 (et par conséquent ρ_S^2). On peut voir ceci comme une prise en compte des occultations à l'intérieur du volume de relief pour maximiser les valeurs du rayon de sécurité : un rayon qui possède une petite distance ρ_S^5, et qui donc majorerait la distance ρ_S^3 de son point d'origine (et donc la distance ρ_S^2 de la projection 2D de ce point), mais ne provient pas de l'extérieur (tel qu'une occultation en amont par le relief empêche d'aboutir à un tel rayon en venant de l'extérieur, voir figure 4.24) n'a pas a être pris en compte dans le calcul de cette distance, ce qui ne peut qu'être avantageux.

Préparation pour l'algorithme de rendu

Avant d'exposer en détail l'algorithme de rendu il faut encore effectuer certains choix ainsi qu'analyser et détailler quelques points importants.

Le premier choix qu'il faut faire, c'est la manière de représenter les rayons de sûreté. Nous choisissons d'utiliser une carte de la même résolution que la carte de hauteur, en associant une valeur unique ρ_{ij} à chaque texel τ_{ij} (nous discuterons plus tard des alternatives possibles et des éventuelles modifications qu'elles impliquent). Interpoler linéairement des valeurs du rayon de sûreté produirait une information difficilement interprétable, nous choisissons donc d'utiliser une interpolation de plus proche voisin, ce qui a pour effet d'attribuer la même valeur ρ_{ij} à tous les points du texel τ_{ij}.

Le deuxième choix concerne la méthode d'interpolation de la carte de hauteur, qui rappelons-le détermine la forme de la surface reconstruite, et influe alors inévitablement sur la fonction de distance élargie et par conséquent le rayon de sécurité. Pour notre première méthode de rendu (l'algorithme sans précalcul) nous avons montré qu'on pouvait utiliser l'interpolation de plus proche voisin, l'interpolation bilinéaire et l'interpolation barycentrique linéaire sur grille triangulaire. L'interpolation de plus proche voisin définit une surface discontinue en escaliers, qui risque d'impliquer des valeurs faibles pour la fonction de distance élargie. L'interpolation linéaire sur grille triangulaire demande de reprogammer l'interpolation barycentrique, ce qui peut représenter un surcoût rendant la technique *a priori* moins intéressante que l'interpolation bilinéaire, implémentée efficacement sur le matériel graphique. Nous nous restreignons donc à cette dernière.

La surface reconstruite par interpolation bilinéaire est quadratique par morceaux et n'est généralement pas C^1 aux bordures de texels. Des arêtes vives peuvent exister au niveau de ces bordures, limitant pour les rayons lumineux concernés la valeur de distance élargie ρ_S.

Autoriser des rayons de sûreté nuls obligerait à ajouter un branchement dans la boucle itérative de l'algorithme de parcours du rayon, puisqu'il faudrait dans ce cas, pour sortir de cette zone de rayon nul, basculer provisoirement sur un parcours du rayon au niveau des intercloisons, du même type que celui utilisé par notre algorithme sans précalcul (voir section 4.2.4). Les branchements dynamiques sont encore très limités sur le processeur graphique et cassent complètement le parallélisme qui en fait la force, il faut donc chercher à conserver la simplicité des itérations à effectuer.

Pour ce faire nous proposons la solution suivante : lors du parcours du rayon lumineux, les pas effectués ne seront permis qu'au niveau des interbordures orthogonales à la direction principale du rayon lumineux (deux possibilités selon que le rayon est plutôt dirigé selon l'axe des x ou celui des y). Ainsi si l'on considère un texel particulier, seuls les points appartenant aux deux segments orthogonaux définis par les interbordures peuvent avoir une chance d'être atteints lors du parcours d'un rayon. Il est donc inutile de considérer le reste des points du texel dans le calcul du rayon de sûreté. Pour être plus précis notons, toujours en considérant un texel isolé, qu'avec le parcours proposé, le segment correspondant à l'x-interbordure (resp. y-interbordure) ne peut être atteint que lors du parcours d'un rayon dont la direction principale est l'axe des x (resp. l'axe des y). Enfin comme le parcours se fait d'intercloison en intercloison (selon l'axe x ou selon l'axe x), il est plus pratique et efficace d'utiliser un rayon de sûreté dont la valeur corresponde à une distance projetée sur l'axe concerné plutôt que la distance euclidienne

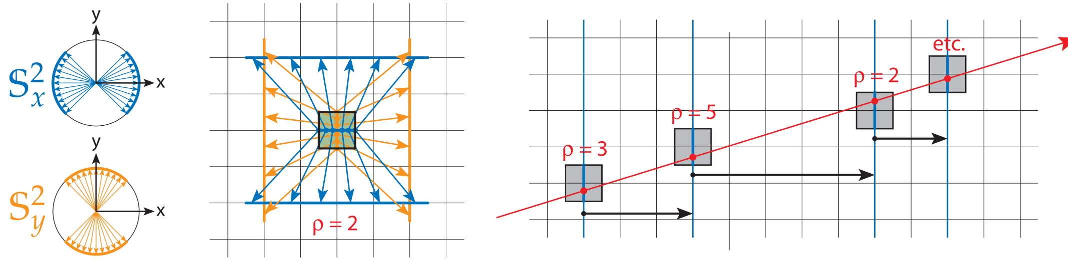


FIGURE 4.25 – Définition et utilisation des valeurs du rayon de sûreté ρ précalculées : la sphère des directions est partionnée en deux ensemble $\mathbb{S}^2_x$ et $\mathbb{S}^2_y$ (figure de gauche). À l'intérieur de chaque texel, seuls deux segments peuvent être atteints lors de l'algorithme de parcours, un segment par ensemble de directions : un exemple de positions courantes et déplacement possibles est donné dans le cas $\rho = 2$ (figure centrale). À droite, un exemple de parcours pour un rayon dont la direction appartient à $\mathbb{S}^2_x$.

(ce qui revient à utiliser la norme infini dans le plan : $\|(x,y)\|_\infty = \max(|x|,|y|)$). En restreignant les valeurs de cette distance aux multiples entiers de la distance entre cloisons (c'est-à-dire $e_N = {}^1/_N$), on s'assure en effectuant un pas depuis une intercloison de retomber sur une autre intercloison.

Ces considérations mènent à la définition suivante du rayon de sûreté ρ_{ij} associé à un texel τ_{ij} :

$$\rho_{ij} = \frac{1}{N}\left\lfloor N.\min\left\{\min_{y\in I_j^N}\xi_S^{2x}(x_i,y);\min_{x\in I_i^N}\xi_S^{2y}(x,y_j)\right\}\right\rfloor$$

$$\text{avec}\qquad \xi_S^{2x}(x,y) = \inf_{d\in\mathbb{S}^2_x,z\in[0,1]}\xi_S^5(x,y,z,d).|d_{|x}|$$

$$\text{et}\qquad \xi_S^{2y}(x,y) = \inf_{d\in\mathbb{S}^2_y,z\in[0,1]}\xi_S^5(x,y,z,d).|d_{|y}|$$

en notant I_i^N l'intervalle $[\frac{i}{N},\frac{i+1}{N}]$, $\mathbb{S}^2_x$ le sous-ensemble des directions dont la composante x est supérieure à la composante y en valeur absolue, et $\mathbb{S}^2_y$ son complémentaire.

Enfin le dernier point à mentionner est que comme nous l'avons expliqué nous interdisons la valeur nulle pour le rayon de sûreté ρ_{ij}, ce qui revient imposer un rayon minimum de e_N, correspondant à la distance entre deux intercloisons voisines. C'est une approximation que nous faisons là (et c'est la seule) puisqu'il existe des situations où la valeur théorique de ρ_{ij} s'annule. En fait on peut voir cette hypothèse comme une contrainte sur les cartes de hauteur acceptées par notre algorithme. Cette contrainte peut être exprimée grossièrement comme suit : la carte de hauteur ne doit pas contenir de pic élevé ou d'arête aiguë de la taille d'un texel. Dans la pratique c'est peu contraignant car ce genre de surface est rare et dans le cas où l'on veut rendre un relief contenant des arêtes fines, il suffit d'échantillonner la fonction de hauteur convenablement, de sorte que les arêtes fines occupent au moins deux texels.

Maintenant que nous avons défini précisément l'information précalculée, voyons les détails de l'algorithme de rendu et son implémentation sur le processeur graphique. Nous expliquerons ensuite en détail comment calculer efficacement et de manière robuste cette information du rayon de sûreté en pratique.

Algorithme de rendu

Avec la définition précise du rayon de sûreté que nous venons de donner, l'algorithme suivant de calcul d'intersection peut être appliqué :

Algorithme 4.6 : Algorithme d'intersection utilisant le rayon de sûreté

```
p0 ← point d'entrée du rayon dans le volume de relief
p1 ← point de sortie du rayon hors du volume de relief
d ← p1 − p0
si |dx| > |dy| alors
  | du ← d ÷ (N × |dx|)
  | s ← si (dx > 0) alors 0 sinon 1
  | dt ← |s − {N × p0x − 0.5}|
sinon
  | du ← d ÷ (N × |dy|)
  | s ← si (dy > 0) alors 0 sinon 1
  | dt ← |s − {N × p0y − 0.5}|
// {x} dénote la partie fractionnaire de x
a, b ← p0 − dt × du
tant que h(bxy) < bz et ||b − p0|| ⩽ ||p1 − p0|| faire
  | a ←− b
  | b ←− b + ρplus_proche(bxy) ∗ du
si h(bxy) < bz alors
  | ⇒ on est sorti du volume sans trouver d'intersection
sinon
  | // l'intervalle [a, b] contient l'intersection
  | tant que ||bxy − axy|| > ϵ faire
  |   | q ← (a + b) ÷ 2
  |   | si qz > h(qxy) alors
  |   |   | a ← q
  |   | sinon
  |   |   | b ← q
  | ⇒ intersection trouvée en a
```

La méthode dichotomique choisie ici pour la recherche de racine est la méthode de la bissection, une légère modification dans le calcul de q permet d'obtenir la méthode de regula falsi :

$$q \leftarrow (a \times hb - b \times ha) \div (hb - ha)$$

où ha et hb correspondent aux valeurs $h(a_{xy}) - a_z$ et $h(b_{xy}) - b_z$ stockées dans des variables intermédiaires pour éviter les calculs redondants.

Une implémentation en GLSL complète de cet algorithme de rendu est donnée en annexe (voir section A).

La figure 4.26 montre un exemble de nombre d'itérations nécessaires pour déterminer l'intervalle de recherche en utilisant les données du précalcul (se référer à la section qui suit pour la disctinction entre les deux types de précalcul utilisés). On

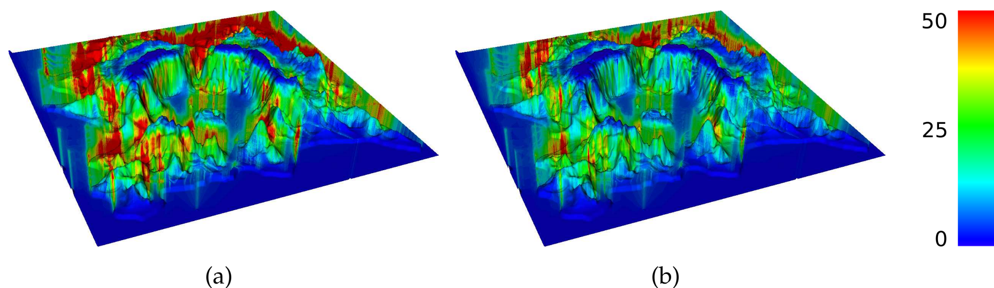


FIGURE 4.26 – Rendu en fausse couleurs (échelle donné à droite) du nombre d'itérations requis par l'algorithme de rendu pour le cas d'un champ de hauteur de résolution 1024^2 : (a) avec le précalcul sans prise en compte des prébloquages; (b) avec le précalcul prenant en compte les prébloquages.

constate qu'en moyenne 25 itérations sont suffisantes pour déterminer l'intervalle de recherche sur un champ de hauteur de résolution 1024^2, ce qui est peu par rapport aux quelques centaines de pas que requiert l'algorithme sans précalcul. Les résultats en termes de rapidité de rendu sont comparés en section 4.2.7.

Détail de l'algorithme de précalcul

Voyons maintenant comment calculer la carte de rayons de sûreté, de manière robuste et efficace. Considérons la valeur ρ_{ij} associée au texel τ_{ij}. Notons d'abord que pour calculer cette valeur il faut *a priori* parcourir un ensemble de rayons lumineux à quatre ou cinq dimensions (il faut considérer toutes les directions à partir de tous les points du volume à la verticale d'un texel). La première approche que nous avons appliquée [BD06] consistait à échantillonner densément cet ensemble de rayons. Le premier inconvénient d'une telle approche est son coût *a priori* élevé, nécessitant de traiter une grosse quantité de donnée, difficulté que nous contournions en utilisant une implémentation efficace sur GPU de l'algorithme de précalcul, permettant d'obtenir des temps de précalcul raisonnables (de l'ordre d'une dizaine de secondes pour une carte de taille 128×128). Le deuxième inconvénient, plus problématique, vient de la nature même d'une approche par échantillonnage : la fonction que l'on cherche à échantillonner (la distance élargie) peut présenter des discontinuités et des hautes fréquences, on n'a donc jamais la garantie d'avoir considéré toutes les valeurs possibles. C'est exactement le même problème que celui que pose le parcours d'un rayon lumineux par pas constants pour calculer son intersection avec un relief. Or l'objectif ce ce précalcul est justement d'obtenir une valeur robuste du rayon de sûreté, permettant de garantir que l'on ne rate aucune intersection lors du rendu !

Nous proposons donc une méthode robuste utilisant des notions de visibilité et manipulant des sous-ensembles continus de rayons lumineux pour s'assurer que la valeur calculée ρ_{ij} est exacte, tout en évitant tout calcul redondant.

L'intuition principale menant à l'algorithme que nous allons présenter est la suivante : si la surface du relief est C^1, les rayons lumineux correspondant aux valeurs minimales de la distance élargie sont les rayons tangents à la surface (voir figure 4.27). Plus précisément, cette propriété s'énonce comme suit :

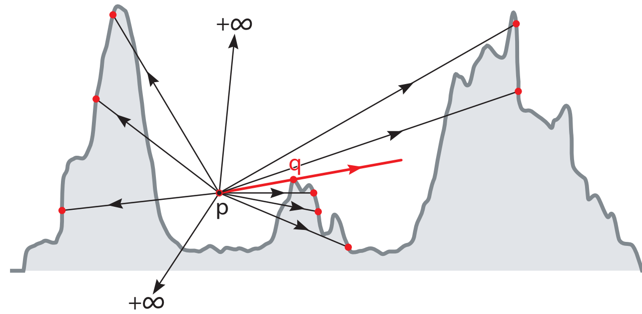


FIGURE 4.27 – La distance élargie en un point p est donnée par sa distance à un point q de la surface du relief, où la droite (pq) y est tangente.

Propriété. *Soient S la surface d'un champ de hauteur C^1 et $p \in \mathbb{R}^3$ tel que*

$$0 < \rho_S^3(p) < +\infty$$

Alors il existe $q \in S$ tel que

$$\|q - p\|_{2_{xy}} = \rho_S^3(p) \quad \text{et} \quad \text{la droite } (pq) \text{ est tangente à } S \text{ en } q$$

Cela nous donne donc une indication importante sur les rayons lumineux qu'il faut examiner pour déterminer le rayon de sûreté, puisqu'il suffit de ne considérer que ceux qui sont tangents à la surface. Nous étudions d'abord le cas à une dimension, correspondant à la coupe du champ de hauteur selon un plan vertical P, avant de décrire la généralisation en deux dimensions.

Algorithme dans le cas à une dimension Nous avons supposé que la surface S était obtenue par interpolation bilinéaire de la carte de hauteur, que nous approchons en supposant que l'intersection par un plan vertical est le graphe d'une fonction linéaire par morceaux (comme nous l'avons supposé plus tôt). Appelons g la restriction de la fonction de hauteur h au plan P, reparamétrée sur $[0, 1]$. Pour simplifier supposons que g est linéaire par morceaux selon une subdivision régulière $(x_k) \in [0, 1]^{[\![0,K]\!]}$ avec $K \in \mathbb{N}$ telle que

$$0 = x_0 < \ldots < x_k < x_{k+1} < \ldots < x_N = 1$$

Comme nous l'avons montré plus tôt, cette subdivision n'est en pratique pas régulière mais cela n'affecte pas les explications qui suivent.

Le problème consiste pour un point x_c fixé, de calculer le rayon ρ_c associé, qui peut dans ce cas simplifié s'exprimer comme suit :

$$\rho_c = \inf_{g(x_c) < z \leqslant 1} \inf_{d \in \mathbb{S}^1_+} \rho_S^5((x_c, z), d)$$

(notons que nous utilisons implicitement une redéfinition de la fonction ρ_S^5 sur $\mathbb{R}^2 \times \mathbb{S}^1$ en utilisant la correspondance induite par le plan P sur les positions et directions, planaires ou spatiales).

Il faut donc théoriquement considérer tous les points à la verticale de x_c (au dessus du point $p = (x_c, g(x_c))$ et toutes les directions d'abscisse positive, problème à une dimension spatiale et une dimension directionnelle. La propriété suivante permet de mieux cerner le problème (la notation $[ab](x)$ représente l'interpolation linéaire entre a et b, c'est à dire $[ab](x) = ((b_x - x).a_y + (x - a_x).b_y).(b_x - a_x)^{-1})$:

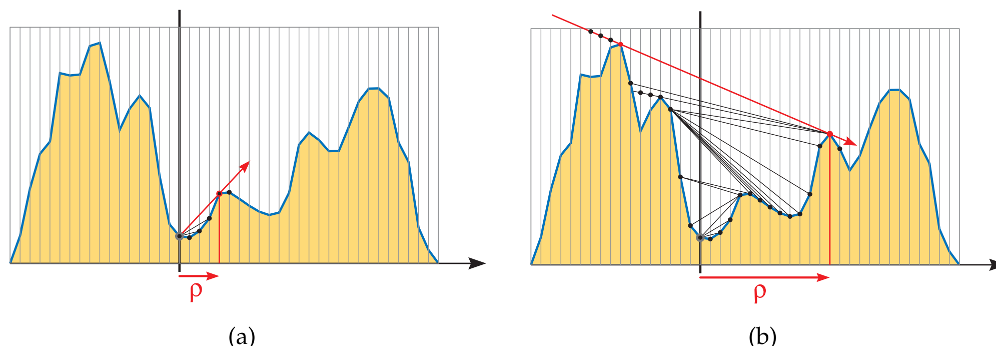


Figure 4.28 – Algorithmes itératifs de calcul du rayon de sûreté dans le cas à une dimension (pour les rayons lumineux allant de la gauche vers la droite) : (a) sans prise en ompte des occultations; (b) avec prise en compte des occultations. Dans chaque cas, le rayon dénoté ne rouge correspond au rayon lumineux limite définissant le rayon de sûreté. Les segments verticaux correspondent à la subdivision (x_k).

Propriété. *Si $\rho_c > 0$, alors il existe $n > c$ tel que*

$$\begin{cases} \rho_c = x_n - x_c \\ \forall x \in]x_c, x_n[,\ g(x) < [p_c p_n](x) \\ \exists \epsilon > 0 \textit{ tel que } \forall x < x_n + \epsilon,\ g(x) \leqslant [p_c p_n](x) \end{cases}$$

Autrement dit, si ρ_c est positif, il est donné par un point q sur le graphe de g tel que le rayon d'origine p et de direction $\overrightarrow{pq}$ n'intersecte pas le graphe de g entre p et q et est tangent à g en q (pour une définition de la tangence étendue aux fonctions C^0 : une droite sera dite tangente à g en un point q si et seulement s'il existe un voisinage ouvert du point q dans lequel g se trouve d'un seul coté de cette droite).

Réciproquement, tout rayon $(p, \overrightarrow{pq})$ vérifiant ces propriétés donne une borne sur la valeur de ρ, il suffit de déterminer celui pour lequel l'abscisse de q soit minimale pour trouver la valeur de ρ. Cette observation permet d'utiliser l'algorithme itératif suivant pour calculer ρ :

Algorithme 4.7 : Algorithme de calcul du rayon de sûreté dans le cas à une dimension, sans prise en compte des prébloquages.

Entrées : c = indice de l'abscisse x_c où le rayon de sûreté est calculé
$p \leftarrow (x_c, g(x_c))$
$q \leftarrow (x_{c+1}, g(x_{c+1}))$
$k \leftarrow c + 2$
tant que $g(x_k) > (pq)(x_k)$ **faire**
  $q \leftarrow (x_k, g(x_k))$
  $k \leftarrow k + 1$
$\rho_c \leftarrow q_x - x_c$

Un exemple de parcours effectué par cet algorithme est illustré figure 4.28(a).

Rappelons maintenant un point que nous avions mentionné précédemment : si l'on veut calculer une valeur de ρ optimale pour l'algorithme de rendu, il ne faut pas prendre en compte les rayons intérieurs, c'est-à-dire ceux qui ne peuvent pas

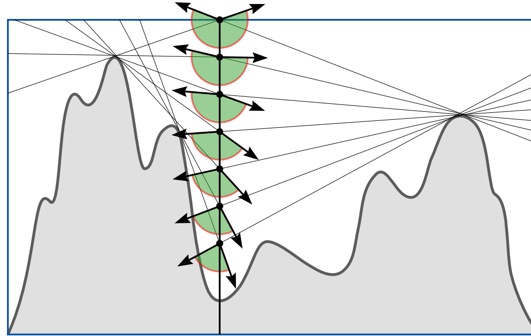


FIGURE 4.29 – Prise en compte des occultations pour le calcul du rayon de sûreté : en chaque point du volume de relief, seul un sous-ensemble restreint des directions possibles pour les rayons lumineux est à prendre en compte.

provenir de l'extérieur, parce que bloqués en amont par une partie du relief. La figure 4.24 illustre le gain que cela peut représenter pour la valeur de ρ.

Il est possible de montrer que dans ce cas là aussi ce sont les rayons tangents qui fixent la valeur de ρ, à la différence que dans ce cas la tangence est double : le rayon limite est un rayon défini par deux points p et q du graphe de g, tangent en ces deux points. L'algorithme de parcours est plus compliqué mais reste similiaire : on a comme précédemment un point q qui avance sur le graphe de g, tandis qu'un point p recule sur ce graphe pour détecter les éventuels prébloquages. On applique le même test que précédemment pour faire avancer q, mais on a maintenant la possibilité de faire remonter p (en reculant sur le graphe de g) dès que q est bloqué. L'algorithme est illustré par la figure 4.28(b).

Nous appelons cet algorithme l'algorithme de précalcul avec prise en compte des occultations. On voit bien déja dans sa version à une dimension qu'il peut s'avérer beaucoup plus coûteux à calculer puisqu'il peut nécessiter de remonter loin derrière le rayon lumineux pour détecter une éventuelle occultation. Nous en avons implémenté une version dans le cas à deux dimensions, plus complexe puisque le passage aux deux dimensions nécessite d'utiliser des notions de visibilité évoluées comme la fusion d'occulteurs. Nous proposons donc en premier lieu, pour le calcul des rayons de sûreté en deux dimensions, un algorithme simple dans le cas où les prébloquages ne sont pas pris en compte. Nous expliquerons ensuite les modifications à apporter pour prendre en compte ces prébloquages et ainsi obtenir de meilleurs valeurs du rayon de sûreté.

Version simple sans occultations Concentrons nous sur le calcul de la valeur ρ_{ij} associée au texel τ_{ij}. Comme nous l'avons vu lors de la définition de ρ_{ij}, seuls les deux segments orhogonaux $t^x = \{x_i\} \times I_j^N$ et $t^y = I_i^N \times \{y_j\}$ correspondant aux interbordures de ce texel sont à considérer. De plus les ensembles de directions à considérer pour chacun de ces deux segments sont disjoints : $\mathbb{S}_x^2$ pour t^x et $\mathbb{S}_y^2$ pour t^y. Nous montrons d'abord comment calculer le rayon de sûreté ρ_{ij}^{x+} pour le sous-ensemble des rayons dont la direction appartient à $\mathbb{S}_{x+}^2$, l'ensemble des directions dont la composante selon x est positive et supérieure à la composante y. Pour calculer les trois autres valeurs (correspondant aux sous-ensembles de directions $\mathbb{S}_{y+}^2$, $\mathbb{S}_{x-}^2$, $\mathbb{S}_{y-}^2$), il suffira ensuite de tourner successivement le plan xy d'un angle $\pi/2$ avant d'appliquer la méthode que nous allons donner. Le rayon de sûreté ρ_{ij}

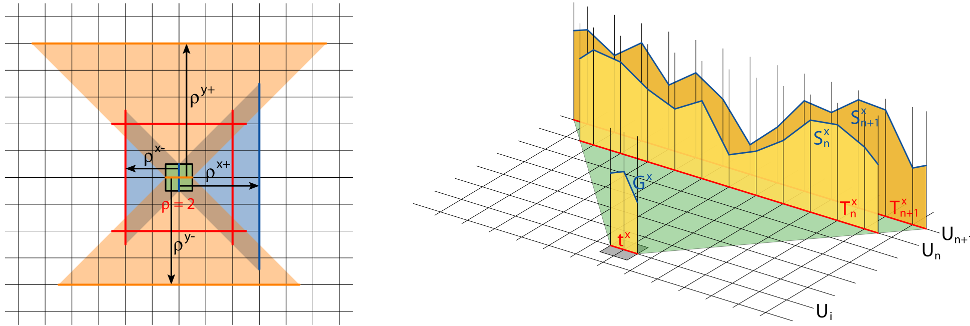


FIGURE 4.30 – Définition du rayon de sûreté ρ_{ij} comme la plus petite des quatre valeurs directionnelles ρ_{ij}^{x+}, ρ_{ij}^{x-}, ρ_{ij}^{y+} et ρ_{ij}^{y-} (à gauche); notations utilisées dans cette section (à droite).

pourra enfin être obtenu en prenant la plus petite des quatre valeurs :

$$\rho_{ij} = \min(\rho_{ij}^{x+}, \rho_{ij}^{x-}, \rho_{ij}^{y+}, \rho_{ij}^{y-})$$

Nous cherchons donc à déterminer la différence minimale ρ_{ij}^{x+} entre les abscisses d'un point p situé à la verticale du segment $t^x i$ et un point q tel que la droite (pq) est tangente à la surface du relief. Comme dans le cas 1D, Il est facile de voir que l'on peut se restreindre pour la position de p aux points de l'image G^x du segment t^x par la fonction de hauteur ($G^x = \{(x, y, h(x, y)) \ / \ (x, y) \in t^x\}$) : comme nous ne considérons pas les prébloquages, connaissant un rayon tangent partant d'un point p, redescendre le point p au niveau de G^x ne peut que faire diminuer la valeur de ρ_{ij}^{x+}. Cela permet déja d'enlever une dimension au problème.

C'est l'idée du parcours itératif 1D, donné précédemment, qui guide notre algorithme en 2D. L'algorithme consiste à tester les valeurs successives possibles du rayon de sûreté (multiple entier de $e_N = 1/N$), par incrémentation de e_N à chaque itération. Applelons T_k^x l'ensemble des points situés à une distance $k.e_N$ en abscisse du segment t^x, dans une direction incluse dans $\mathbb{S}^2_{x+}$, autrement dit :

$$\begin{aligned} T_k^x &= \left\{ (x_i, y) + \frac{k}{N} \cdot \frac{d}{|d_{|x}|} \quad / \quad y \in I_j^N, d \in \mathbb{S}^2_{x+} \right\} \\ &= \left\{ (x_i + \frac{k}{N}, y) \quad / \quad |y - y_j| \leqslant r + \frac{1}{2N} \right\} \\ &= \{x_{i+k}\} \times \bigcup_{k'=j-k}^{j+k} I_{k'}^N \end{aligned}$$

Définissons l'hypothèse $H(r)$:

$$H(k) \iff (\forall p \in t^x, \forall q \in T_k^x \quad [pq] \text{ ne passe pas sous la surface } S)$$

Cela permet d'exprimer la propriété intéressante qui est la suivante :

Propriété. *Soit $n \in \mathbb{N}$. On a l'équivalence suivante :*

$$(\forall k \in [\![1, n]\!] \ H(k)) \iff \rho_{ij}^{x+} \geqslant n.e_N$$

Cette propriété peut se prouver par un raisonnement géométrique similaire à celui que nous avions effectué dans le cas 1D. Cette propriété implique une nouvelle manière de définir ρ_{ij}^{x+} :

$$\rho_{ij}^{x+} = \min\{n \in \mathbb{N} \ / \ \forall k \in [\![1,n]\!],\ H(k)\}$$

Donc si l'on sait évaluer la propriété $H(n)$, cela donne un moyen de calculer ρ_{ij}^{x+} de manière itérative. Elle peut se calculer par induction grâce à l'équivalence suivante :

$$\forall n \in \mathbb{N}, \quad H(n+1) \iff (H(n) \ \text{ et } \ Q(n))$$

avec

$$Q(n) \iff (\forall p \in t^x, \forall q \in T_{n+1}^x,\ h([p,q](x_{i+n})) \leqslant [h(p),h(q)](x_{i+n}))$$

Si l'on appele S_n^x l'image du segment T_n^x par la fonction de hauteur h, c'est à dire $S_n^x = \{(x,y,h(x,y)) \ / \ (x,y) \in T_n^x\}$, la propriété $Q(n)$ se traduit par le fait que S_n^x se trouve *en dessous* de tous les segments existants entre un point de G^x et un point de S_{n+1}^x (voir figure 4.31(a)). Ainsi si à la $n^{\text{ième}}$ itération on a prouvé que $H(n)$ est vérifiée, et que l'on est capable d'évaluer $Q(n)$, c'est-à-dire s'il existe ou non un segment entre G^x et S_{n+1}^x qui passe sous la surface au niveau du segment T_{n+1}^x, on peut déterminer si $H(n+1)$ est vérifiée (dans le cas où un tel segment n'existe pas), auquel cas ρ_{ij}^{x+} est strictement supérieur à $n.e_N$, et il faut itérer, ou si $H(n+1)$ n'est pas vérifée (dans le cas contraire), auquel cas on a déterminé $\rho_{ij}^{x+} = n.e_N$.

Donc il faut à la $n^{\text{ième}}$ itération considérer l'ensemble des segments qu'il est possible de former entre un point de G^x et S_{n+1}^x, pour les comparer à la hauteur du relief au dessus du segment T_n^x, c'est-à-dire au niveau de l'x-intercloison U_{i+n}. C'est un ensemble à deux dimensions que l'on peut partitionner en éléments plus simples : aussi bien G^x que S_{n+1}^x sont des courbes polygonales, composées de deux segments pour G^x et $2(n+3)$ pour S_{n+1}^x. Cela représente $4(n+3)$ couples de segments à tester.

Soient a et b deux segments d'extrémités $a_0, a_1, b_0, b_1 \in \mathbb{R}^3$, appelons $L_{a\mapsto b}$ l'ensemble des segments formés entre un point de a et un point de b, c'est-à-dire :

$$L_{a\mapsto b} = \{[p_a, p_b] \ / \ p_a, p_b \in a \times b\}$$

L'union des segments contenus dans $L_{a\mapsto b}$ forme un tétraèdre, d'arètes a, b, $[a_0, b_0]$, $[a_0, b_1]$, $[a_1, b_0]$ et $[a_1, b_1]$, comme le montre la figure 4.31(b).

Si l'on prend pour segment a un des deux segments de G^x et pour segment b un des $2(n+3)$ segments de S_{n+1}^x, l'intersection du tétraèdre formé par les segments de $L_{a\mapsto b}$ avec l'x-intercloison U_{i+n} est un parallélogramme qu'il est facile de calculer connaissant a et b. Cette propriété vient du fait que les segments a et b sont contenus dans les plans U_i et U_{i+n+1}, parallèles à U_{i+n}. Ce parallélogramme déterminé, il suffit de tester si ses deux segments inférieurs passent en dessous du relief ou non, pour déterminer s'il existe un segment dans $L_{a\mapsto b}$ qui passe sous le relief au niveau de l'intercloison U_{i+n}.

Donc en résumé, pour tester la propriété $Q(n)$, il suffit de tester pour l'ensemble des $4(n+3)$ tétraèdres formés entre G^x et S_{n+1}^x s'il en existe un qui passe sous la surface du relief au niveau de l'intercloison U_{i+n}.

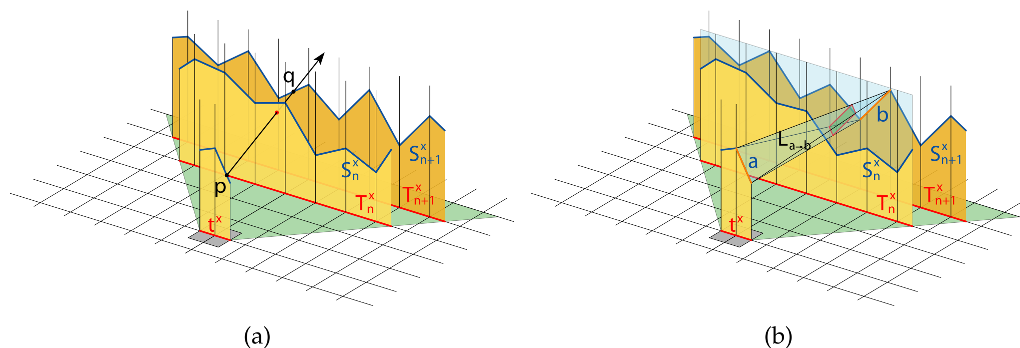


FIGURE 4.31 – (a) un exemple de rayon allant d'un point p vers un point q qui empêche la propriété $Q(n)$ d'être vérifiée; (b) le tétraèdre formé par les segments de $L_{a \mapsto b}$, dans un cas où il passe au dessus de S_n^x.

résolution	128^2	256^2	512^2	1024^2
temps précalcul	4	20	50	1000

TABLE 4.1 – Temps de précalcul en secondes, pour plusieurs résolutions de champs de hauteur, obtenus sur un processeur Intel Pentium 4 à 3 GHz.

L'algorithme complet et détaillé est donné en annexe B. Il faut pour chaque texel τ_{ij} effectuer $O(\rho_{ij}^2)$ tests de tétraèdres, donc si l'on appele $\bar{\rho}$ le rayon de sûreté moyen, la complexité de l'algorithme est en $O(\bar{\rho}^2 N^2)$.

Résultats L'implémentation relativement optimisée de l'algorithme de précalcul, celle donnée en annexe B, permet de calculer une carte de rayons de sûreté en des temps raisonnables, généralement de l'ordre de la dizaine de secondes, comme le montre le tableau 4.1. Ces temps s'avérant convenables pour notre utilisation, nous n'avons pas poussé plus loin les optimisations, mais notons que le parallélisme qu'offre l'algorithme (les valeurs ρ_{ij} sont calculées indépendamment les unes des autres) se prêterait bien à une implémentation sur GPU.

Compromis qualité / rapidité Pour définir un compromis réglable entre rapidité de rendu et qualité de rendu, il est possible de définir un seuil ϵ, permettant de ne pas considérer comme traversant la surface les rayons qui le font à une épaisseur inférieure à ϵ (voir figure 4.32). Cette erreur ϵ correspond alors à l'erreur maximale autorisée sur les silhouettes, qu'il est possible de régler si l'on connait à l'avance la résolution maximale à laquelle sera rendu le relief, de sorte qu'elle ne dépasse pas la taille d'un pixel de l'écran, rendant les erreurs ainsi effectuées imperceptibles.

On peut ainsi définir plusieurs niveaux de détails, indépendamment de la résolution de la carte de hauteur, en calculant plusieurs cartes de rayons de sûreté, pour des valeurs différentes du de l'erreur ϵ.

Notons que dans l'implémentation de l'algorithme de précalcul fournie en annexe, la valeur ϵ est utilisée au moment de la comparaison de la hauteur d'un tétraèdre avec celle du relief au niveau d'une intercloison, en ajoutant simplement la valeur ϵ à la hauteur des points du tétraèdre, indépendamment de l'inclinaison des rayons lumineux qu'il représente, ce qui n'est pas exactement la définition que

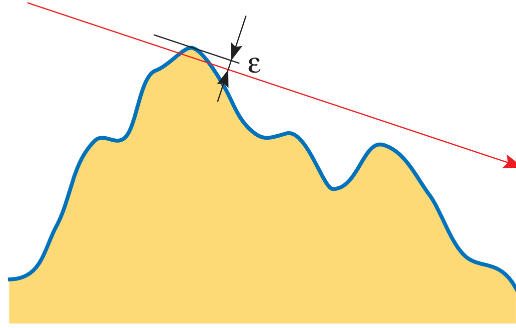


FIGURE 4.32 – Utilisation d'une erreur aux silhouettes ϵ réglable, permettant d'accélérer encore l'algorithme de rendu.

nous venons d'en donner. Il faudrait idéalement augmenter cette valeur pour les rayons inclinés (par divison de la valeur ϵ par le cosinus de l'angle de ces rayons par rapport à l'horizontale). Les valeurs de rayons de sûreté que nous calculons ainsi sont donc inférieures à celles théoriques correspondant à la prise en compte exacte de la valeur ϵ, mais elles restent proches et autorisent un calcul plus simple.

Version optimale avec occultations La prise en compte des occultations créées par la surface du relief, pour obtenir des valeurs optimales des rayons de sûreté est plus complexe. Replaçons nous dans le cas où l'on cherche à calculer le rayon ρ_{ij}^{x+} d'un texel τ_{ij}, en ne prenant donc en compte que les directions de rayons lumineux dont la composante en x est positive et supérieure à la composante en y. Supposons que l'on se trouve à la $n^{\text{ième}}$ itération de l'algorithme que nous venons de voir, et que nous cherchons à prouver la propriété $Q'(n)$, expression de la propriété $Q(n)$ dans le cas où les occultations sont prises en compte, qui peut s'écrire ainsi :

$$\begin{aligned} Q'(n) \iff & \big(\forall p \in t^x, \forall q \in T^x_{n+1}, \\ & \exists p' \in (pq) \text{ tel que } p'_{|x} \leqslant p_{|x} \text{ et } h([p',q](x_{i+n})) \leqslant [h(p'),h(q)](x_{i+n})\big) \end{aligned}$$

On peut reformuler cela de la manière suivante : $Q'(n)$ est vraie si et seulement si tout rayon lumineux partant d'un point p à la verticale du segment t^x pour atteindre un point q du graphe S^x_{n+1} en traversant la surface S au niveau de l'intercloison U_{i+n}, est bloqué par S en amont (la demi-droite $(p,q]$ traverse S).

Comme dans l'algorithme précédent, nous considérons deux segments a et b, a faisant cette fois partie d'une x-intercloison U_j située *avant* le texel ρ_{ij} (plus précisément telle que $j \leqslant i$) et b étant un des $4(n+3)$ segments de S^x_{n+1} (voir figure 4.33). Notons $L'_{a\mapsto b}$ l'ensemble des segments formés entre un point de a et un point de b dont la projection sur le plan (xy) passe par le segment t^x et dont la direction est incluse dans $\mathbb{S}^2_x$, autrement dit :

$$L'_{a\mapsto b} = \left\{[p,q] \quad / \quad p \in a,\ q \in b,\ \overrightarrow{pq} \in \mathbb{S}^2_x,\ [pq](x_i)_{|xy} \in t^x\right\}$$

On dira que a est un segment *bloquant* (pour le segment b, par rapport au texel τ_{ij}) si :

$$\begin{aligned} &\forall p \in a, \quad p_{|z} \leqslant h(p_{|xy}) \\ &\forall p \in a, \forall q \in b, \quad [p,q] \in L'_{a\mapsto b} \Rightarrow h([pq](x_{i+n})_{|xy}) \leqslant [pq](x_{i+n})_{|z} \end{aligned}$$

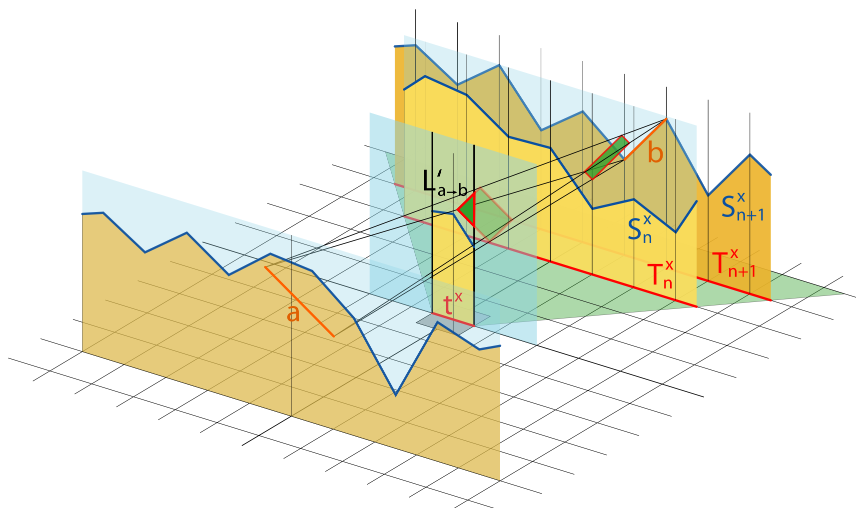


FIGURE 4.33 – Exemple de segment bloquant a : tous les points de a se trouvent sous le relief et tous les segments de l'ensmble $L'_{a\mapsto b}$ passent au dessus de S_n^x.

Autrement dit un segment est dit bloquant lorsqu'il bloque une partie des rayons qui en passant au-dessus de t^x pour traverser la surface S au-dessus de T_n^x avant d'atteindre le graphe S_{n+1}^x auraient impliqué une borne sur la valeur de ρ_{ij}^{x+}. Donc si l'on est capable de trouver un ensemble de segments bloquants dont la réunion des rayons bloqués couvre l'ensemble de rayons partant d'un point à la verticale de t^x, dans une direction de $\mathbb{S}_{x+}^2$, alors on à démontré que la propriété $Q'(n)$ est vraie. Réciproquement, si l'on arrive à exhiber un rayon lumineux passant au dessus de t^x pour atteindre S_{n+1}^x après avoir traversé la surface sous S_n^x, qui ne soit pas occulté en amont par le relief, on montre que $Q'(n)$ est fausse. C'est la base de l'algorithme que nous avons implémenté. La recherche de l'ensemble de segments bloquants peut se voir comme un problème de visibilité 2D : les segments candidats sont des occulteurs, qu'il faut arriver à fusionner, du point de vue de la visibilité, pour déterminer s'ils couvrent toutes les directions considérées. Le test pour déterminer si un segment est bloquant se fait de manière similaire aux tests de tétraèdres utilisés dans la version sans occultations. Nous prenons comme segments occulteurs candidats, les segments formés par intersection de S avec les intercloisons U_j pour $j \leqslant i$. L'algorithme nécessaire pour calculer la fusion entre ces occulteurs nécessiterait encore plusieurs pages d'explication, sortant du cadre de cette thèse, nous en donnons simplement l'idée par la figure 4.34.

Résultats Nous comparons ici seulement les résultats obtenus par cet algorithme avec ceux de l'algorithme précédent. La complexité en pire cas est en $O(\bar{\rho}^2 N^4)$, en pratique le temps de calcul dépend fortement de la forme du champ de hauteur traité (notamment, la présence de grandes zones plates peut nécessiter des parcours longs pour trouver des occulteurs). Les temps de précalcul que nous obtenons sont donnés dans le tableau 4.2, il en ressort qu'il faut grossièrement 20 fois plus de temps pour le calcul avec prise en compte des occultations, ce qui est considérable.

Les gains en terme de temps de rendu dépendent là encore du type de relief considéré, mais ils sont généralement de l'ordre de 15%. Le tableau 4.3 donne un exemple des gains obtenus.

résolution	128^2	256^2	512^2	1024^2
temps précalcul	50s	400s	1200s	7h

TABLE 4.2 – Temps de précalcul avec prise en compte des occultations, pour plusieurs résolutions de champs de hauteur, obtenus sur un processeur Intel Pentium 4 à 3 GHz.

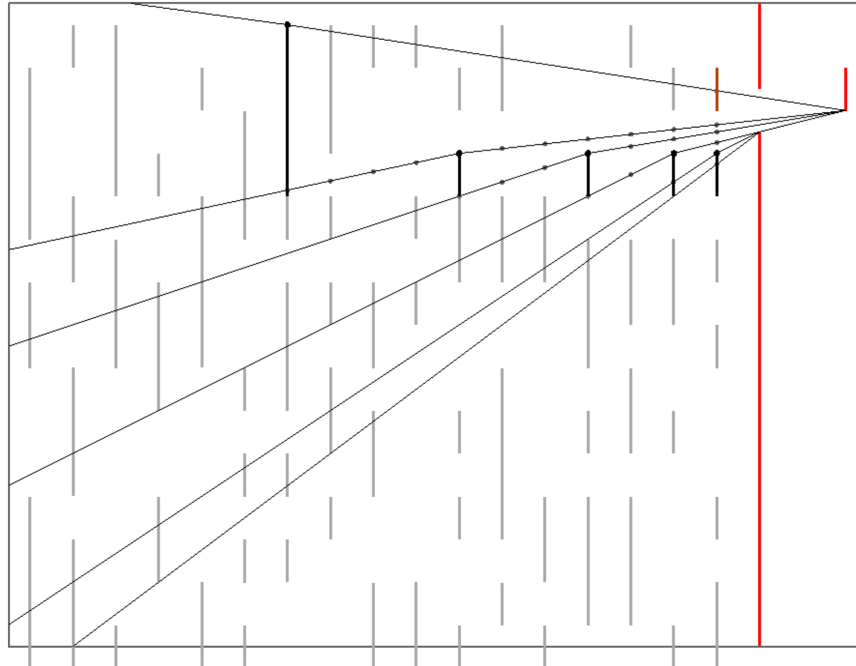


FIGURE 4.34 – Méthodes de visibilité 2D requises par l'algorithme de précalcul avec prise en compte des occultations : on est à amené considérer un ensemble de rayons passant par une « porte » pour atteindre un segment particulier, et vérifier si cet ensemble est bloqué ou non par un ensemble de segments occulteurs

Bilan

Les résultats en termes de rapidité de rendu obtenus grâce à l'information fournie par le rayon de sûreté, détaillés plus tard (section 4.2.7), montrent l'intérêt de posséder une telle information, permettant d'obtenir un rendu robuste et plus efficace que les autres méthodes existantes.

Cependant cette information peut encore être perfectionnée. Le principal défaut du rayon de sûreté vient de sa sensibilité aux petites variations de surface : un relief globalement plat mais avec de petites variations de haute fréquence impliquera des rayons de sûreté faibles. Ce problème vient de la réduction de la distance élargie à seulement deux dimensions, ce qui bien que déja satisfaisant semble pouvoir être amélioré. Notamment il serait intéressant d'étudier d'autres possibilités de réduire la dimensionnalité de l'information de distance élargie 5D, en conservant par exemple certaines dimensions angulaires.

Notons que l'information de distance élargie pourrait être utilisée pour accélerer la méthode du *sphere tracing* dans le cas du rendu de surfaces plus générales que les simples reliefs (surfaces algébriques par exemple).

Mentionnons aussi qu'il serait possible de modifier l'algorithme de précalcul pour prendre en compte d'éventuelles contraintes sur la position du point de vue

connues *a priori*, ce qui aurait pour effet d'augmenter encore les valeurs du rayon de sûreté et ainsi accélérer le rendu.

4.2.6 Géométrie englobante

Nous venons de décrire deux algorithmes permettant de rendre un relief par raycasting. Ces algorithmes sont destinés à être executés par le processeur graphique, dans un *fragment shader*. Voyons maintenant quelques points à prendre en compte à propos de la géométrie englobante du relief, utilisée pour générer les rayons de vue pris en entrée par le *fragment shader*.

Pour les deux algorithmes présentés, les pixels rejetés (fragments en excès) sont les plus coûteux, puisqu'il faut pour les deux techniques parcourir tout le rayon jusqu'à ressortir du volume de relief sans avoir rencontré la surface du champ de hauteur pour déterminer que le rayon de vue considéré n'intersecte pas le relief et en conséquence rejeter le fragment correspondant (instruction `discard` en GLSL).

Ces fragments sont d'autant plus nombreux que la boîte utilisée pour générer les rayons de vue englobe de manière grossière la surface du relief. Il peut donc être avantageux de chercher à englober le relief à l'aide d'une enveloppe mieux ajustée. Rendre une enveloppe mieux ajustée vient forcément au prix d'un plus grand nombre de polygones à rasteriser. Cependant la prise en compte de l'efficacité du matériel graphique en terme de rastérisation de polygones par rapport au coût moyen que représentent les fragments en excès permet de définir le bon compromis. C'est un choix à faire en fonction du nombre d'instances de reliefs de ce type que l'on cherche à rendre, ainsi que de la surface (en pixels) qu'ils risquent d'occuper à l'écran : un grand nombre de reliefs occupant chacun une faible surface à l'écran seront sans doute plus efficacement rendus avec une boîte englobante grossière utilisant peu de polygones, alors qu'un relief détaillé vu de près justifiera l'utilisation de plus de polygones pour mieux s'en rapprocher et ainsi éviter de mobiliser les ressources graphiques pour dessiner du vide.

Comme expliqué au chapitre 3, n'importe quelle enveloppe polygonale englobant le relief peut être utilisée à la place de la boîte unité, il suffit juste d'interpoler linéairement à la surface de chacun des triangles de l'enveloppe les coordonnées de ses sommets (exprimées dans le repère normalisé R_N) pour qu'à la rasterisation chaque fragment dispose des coordonnées du point d'entrée dans cette enveloppe. Notons de plus que dans le cas du rendu de relief, disposer d'un point d'entrée plus proche (que celui donné par le cube unité) peut représenter un gain important aussi pour les fragments qui ne sont pas en excès (pour lesquels on trouve une intersection) : les algorithmes que nous avons présentés se servent d'un point de départ sur le rayon de vue pour avancer le long de celui-ci en se rapprochant de l'intersection recherchée. Ainsi, disposer d'un point de départ déja proche de cette intersection permet de réduire considérablement le nombre d'itérations nécessaires, notamment pour l'algorithme sans précalcul pour lequel le nombre d'itérations effectuées est directement proportionnel à la distance (en norme infini dans le plan) parcourue jusqu'à l'intersection.

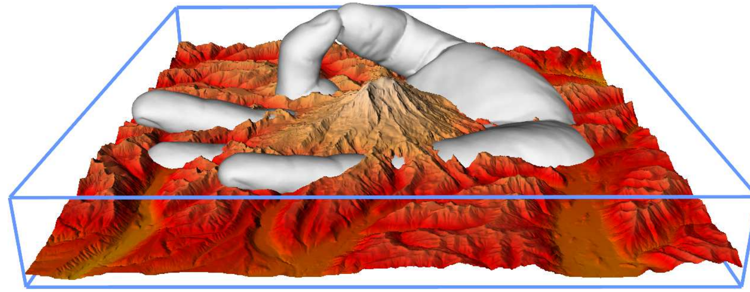


FIGURE 4.35 – Actualisation de la carte de profondeur pour la prise en compte exacte des occultations entre un relief rendu par raycasting et un objet représenté par un maillage polygonal.

méthode de rendu		fréquence (en Hz)
Géométrie (2 triangles par texel)		178
Policarpo [POC05]	50 itérations	360
	100 itérations	210
	200 itérations	118
Maximum Mip-maps [TIS08]		13
Algorithme sans précalcul		116
Rayon de sûreté	$\epsilon = 0$	242 / 292
(sans / avec visibilité)	$\epsilon = 0.001$	295 / 332
	$\epsilon = 0.005$	412 / 483
	$\epsilon = 0.01$	509 / 608

TABLE 4.3 – Comparaison de plusieurs méthodes de rendu sur un champ de hauteur de résolution 1024^2, occupant complètement un écran de résolution 1280×1024.

Carte de profondeur

Rappelons enfin qu'il est possible de calculer pour chaque pixel rendu sa profondeur exacte et actualiser en conséquence la carte de profondeur, puisque l'on dispose des coordonnées normalisées du point d'intersection de chaque rayon de vue avec le relief. Cela est nécessaire pour que les occultations soient traitées correctement lorsque l'on autorise d'autres objets à pénétrer dans le volume de relief, voire intersecter sa surface.

méthode de rendu		fréquence (en Hz)
Algorithme sans précalcul		310
Maximum Mip-maps [TIS08]		22
Relaxed Cone Stepping [PO07]		525
Rayon de sûreté	$\epsilon = 0$	650
(sans visibilité)	$\epsilon = 0.001$	711
	$\epsilon = 0.01$	851

TABLE 4.4 – Comparaison de plusieurs méthodes de rendu sur un champ de hauteur de résolution 256^2, occupant complètement un écran de résolution 1280×1024.

4.2.7 Comparaison

Les tableaux (4.3) et (4.4) montrent les résultats obtenus avec les deux méthodes que nous venons de présenter (algorithme sans précalcul et algorithme avec précalcul du rayon de sûreté), en comparaison avec les autres méthodes existantes, avec une carte graphique GeForce GTX 285.

Plusieurs observations peuvent être tirées de ces résultats :

- pour un champ de hauteur de résolution 1024^2, l'algorithme sans précalcul est aussi rapide que l'alorithme approximatif de Policarpo et al. [POC05] appliqué avec 200 itérations de recherche à pas constants. Comme le montre la figure 4.12, des artefacts persistent pour un tel nombre d'itérations, ce qui n'est pas le cas avec notre algorithme;
- la méthode des Maximum Mip-maps [TIS08], bien qu'elle puisse sembler *a priori* mieux adaptée pour les champ de hauteur de grande résolution, est beaucoup moins performante que notre algorithme sans précalcul. Ceci vient du fait que cette méthode intrinsèquement récursive ne permet pas d'exploiter le parallélisme du processeur graphique aussi bien que notre algorithme, requérant plus d'itérations mais beaucoup plus simples et parallélisables;
- notre algorithme avec précalcul s'avère plus rapide que la méthode du Relaxed Cone Stepping [PO07], qui en plus ne garantit pas l'absence d'erreurs de rendu, de par les approximations faites pour le précalcul;
- la prise en compte de la visibilité lors du précalcul du rayon de sûreté permet en moyenne de gagner 15% de temps de rendu;
- le rendu d'une version maillée du champ de hauteur (optimisée au maximum, utilisant des *vertex arrays*) est plus lent que celui faisant appel à notre méthode avec précalcul, ce malgré la grande résolution de rendu appliquée (le coût de rendu des méthodes par raycasting décroissant linéairement avec le nombre de pixels affichés, contrairement à celui du maillage triangulaire qui reste constant).

4.3 Conclusion

Nous venons de présenter une primitive de rendu efficace, le relief. Elle permet de représenter de manière compacte une grande quantité de détails géométriques et se prête bien au raycasting. Le chapitre suivant étudie le potentiel de cette primitive.

Nous avons montré qu'il est possible de rendre un champ de hauteur à la fois efficacement et de manière exacte. Dans le cas où le champ de hauteur est dynamique et où un précalcul coûteux n'est pas acceptable, nous avons proposé une méthode de rendu simple et efficace, présentant l'avantage par rapport aux autre méthodes existantes d'être exempte d'artefacts.

Dans le cas où le relief à rendre est statique, nous avons introduit le *rayon de sûreté*, qui constitue une quantité avantageuse à précalculer, permettant un rendu à la fois exempt d'artefacts et plus rapide que le reste des méthodes existantes. Cette idée du rayon de sûreté pourrait être étendue à des utilisations plus générales que le rendu de relief.

Chapitre 5

Représentations à base de reliefs

Nous avons montré dans le chapitre précédent que le relief, représenté par une carte de hauteur, s'adapte bien au matériel graphique actuel et peut être rendu très efficacement. Il constitue donc une primitive de rendu intéressante, pouvant remplacer avantageusement une représentation classique à base de polygones, dans de nombreuses situations. Nous étudions dans ce chapitre quel est le spectre de possibilités qu'offre cette primitive.

Les utilisations classiques du relief sont la représentation de terrains et l'ajout de petits détails à une surface. L'accent est mis dans ce chapitre sur la possibilité de représenter des objets entiers uniquement à l'aide de quelques champs de hauteur.

Commençons par voir ce qu'il est possible représenter avec un relief unique avant de passer en revue les différentes manières de combiner plusieurs reliefs pour en étendre les possibilités.

5.1 Représentativité d'un relief unique

Nous nous posons donc pour l'instant la question de savoir ce qu'il est possible de représenter à l'aide d'un unique relief, selon la définition que nous en avons donnée au chapitre précédent (voir section 4.1). Nous nous restreignons ici aux reliefs que savent rendre les algorithmes de raycasting tels que ceux nous avons présenté, c'est-à-dire uniquement les reliefs dont le repère normalisé est défini à une transformation projective près du repère de la scène. Autrement dit les champs de hauteur dont le volume de relief (tel que défini section 4.1) est obtenu par transformation projective du cube unité.

Nous montrons d'abord quels types de déformations cette représentation autorise, et discutons des transformations utiles mais non supportées, et des modifications de l'algorithme de rendu que cela impliquerait. Nous montrons ensuite plus concrètement les applications pratiques de cette représentation pour rendre des objets complets efficacement.

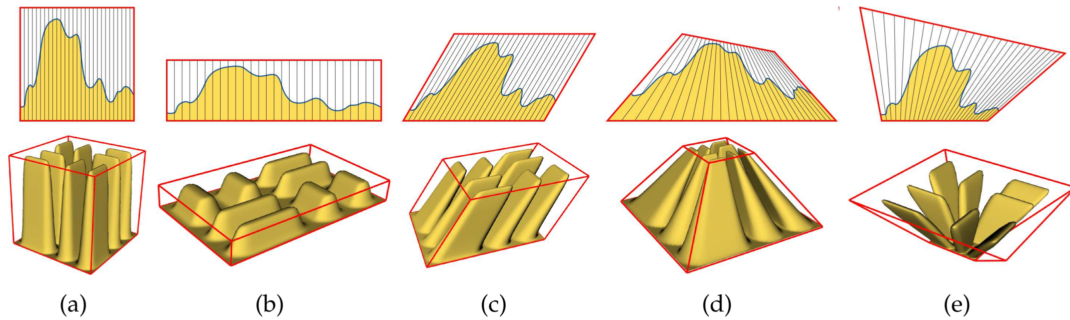


FIGURE 5.1 – Transformations applicables à un relief sans modification de l'algorithme de rendu : (a) isométrie; (b) changement d'échelles; (c) transvection; (d) perspective directe, (e) perspective inversée.

5.1.1 Déformations applicables

Comme nous l'avons vu au chapitre 3, les approches de rendu par raycasting nécessitent de se placer dans un repère dans lequel les droites sont conservées. La transformation la plus générale qui possède cette propriété est la transformation projective, qui inclut notamment la transformation affine.

Transformation affine

Si l'on appele *droites de hauteur* (ou simplement *hauteurs*) les droites d'équation $(x, y) = (x_0, y_0)$ dans le repère normalisé R_N, pour (x_0, y_0) fixé, les hauteurs d'un volume de relief déformé par transformation affine sont parallèles. On peut donc avec un champ de hauteur représenter fidèlement toute surface pour laquelle on peut trouver une direction telle que toute droite parallèle à cette direction n'intersecte qu'une seule fois la surface.

Une fois un champ de hauteur défini, lui appliquer une transformation affine permet principalement d'en changer la position, l'orientation, les échelles (selon x, y et z) et éventuellement en incliner les droites de hauteur (par application d'une transvection).

Les limites d'un tel échantillonnage apparaissent pour les surfaces presques parallèles à la direction des hauteurs : par exemple, si l'on veut représenter un bâtiment entier à l'aide d'un champ de hauteur dont les droites de hauteur sont verticales, bien que le toit soit ainsi capturé correctement, on disposera de peu d'échantillons sur les façades verticales (voire même aucun si les murs sont parfaitement verticaux).

Transformation projective

Une solution simple au problème de l'échantillonnage de toutes les façades d'un bâtiment par un unique champ de hauteur est d'utiliser une transformation non plus seulement affine mais projective, avec laquelle les droites de hauteur ne sont plus parallèles mais toutes concourantes en un même point (le centre de projection).

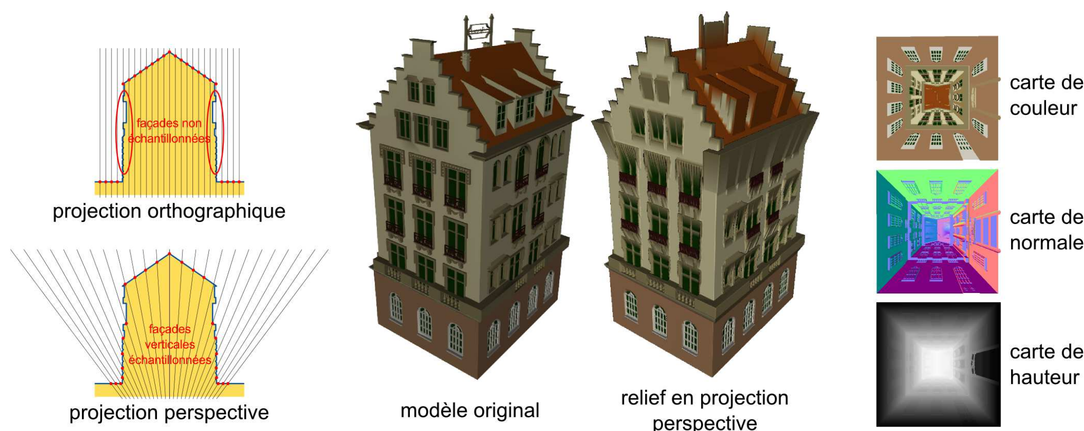


FIGURE 5.2 – Exemple d'utilisation avantageuse d'un relief défini par projection perspective : une telle transformation permet de capturer des détails sur les quatre façades verticales d'un bâtiment, ce qui n'est pas possible avec une simple projection orthographique.

Ainsi en prenant un point au niveau du sol, au milieu du bâtiment, on peut en capturer les façades.

Cette idée est proopsée par Hanson et Wernert [HW98] justement dans le cas des bâtiments, ainsi que par Schaufler [Sch98], qui remarque que l'utilisation de projection perspective a un meilleur pouvoir de représentativité dans certaines situations.

L'utilisation d'une transformation perspective pour représenter une surface offre deux possibilités : soit en prenant un centre de projection *au dessus* de la surface, soit en prenant un centre de projection *en dessous* (voir figures 5.1(d) et 5.1(e)). Dans le premier cas, les droites de hauteur convergent en un point au dessus de la surface, ce qui permet de représenter des surfaces globalement concaves, alors que dans le deuxième cas les droites de hauteur proviennent d'un point sous la surface, ce qui permet de représenter des surfaces globalement convexes.

On appelera *perspective directe* la classe de transformations correspondant au premier cas, puisque une telle carte de hauteur correspond à la carte de profondeur que l'on obtiendrait par un rendu de la surface depuis le centre de projection, en utilisant une perspective classique. Dans le second cas on appelera *perspective inverse* la classe de transformations correspondante, car le sens de projection est inversé par rapport au sens habituel (il part du centre de projection au lieu d'y converger).

Limitations

Malgré l'expressivité permise par les transformations projectives, les déformations autorisées sur les reliefs par l'algorithme de rendu se limitent à celles-ci. Ainsi des déformations non linéaires telle qu'une courbure cylindrique par exemple ne peuvent pas être appliquées. En effet, sous l'action de telles déformations, les rayons lumineux ramenés au repère normalisé ne sont plus des droites, ce qui empêche le rendu par raycasting tel que nous l'avons présenté. Des algorithmes existent qui prennent en compte cette courbure pour marcher le long du rayon, mais il n'en existe pas d'exact (voir plus loin la section 5.2.1 pour plus de détails).

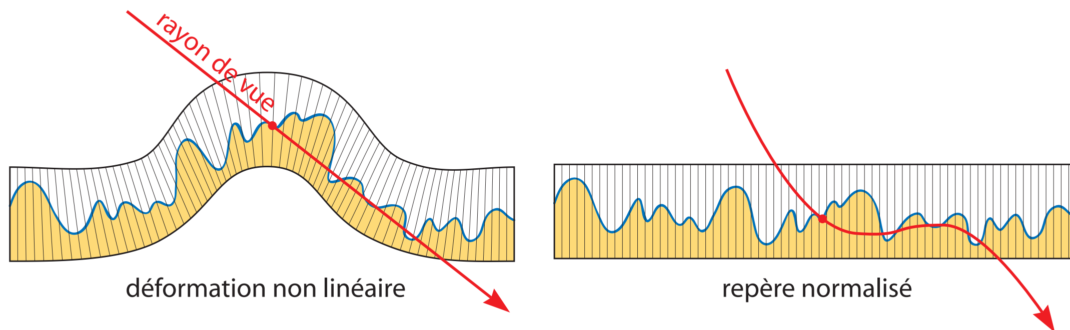


FIGURE 5.3 – Déformation non linéaire du volume de relief : la propriété de conservation des droites n'est pas vérifiée.

Décorrélation de l'information de couleur

Notons enfin que l'information de couleur et l'information géométrique de la surface que l'on cherche à représenter peuvent être complètement décorrélées : l'algorithme de rendu de champ de hauteur sur GPU nécessite que la géométrie soit représentée par une carte de hauteur dont l'élévation en trois dimensions est définie par une transformation projective, mais bien que cela soit la solution la plus directe, rien n'oblige à utiliser la même transformation pour le plaquage de la texture de couleur.

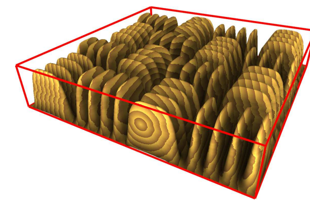


FIG. 5.4 *Rendu de relief avec une texture volumique.*

L'algorithme de rendu par raycasting produit, pour chaque rayon de vue, le point d'intersection exact avec la surface du champ de hauteur, ce qui permet d'utiliser un grand nombre de méthodes de plaquage de texture où cette information est suffisante. Il est par exemple possible d'associer à chaque échantillon du champ de hauteur une ou plusieurs coordonnées de texture, pour pouvoir au moment du rendu, récupérer par indirection l'information de couleur dans un atlas de textures (comme sur un maillage classique, les sommets du maillages étant ici les échantillons du champ de hauteur). Selon l'application il peut être approprié d'utiliser une texture volumique, qui peut éventuellement être stockée de manière efficace dans une *octree texture* [LHN05].

Une possibilité est démontrée par McGuire et Whitson [MW08], qui par un procédé d'optimisation calculenet une carte de couleur continue mais déformée et associe à chaque échantillon du champ de hauteur un couple de coordonnées de texture permettant au rendu de plaquer cette texture sur le relief par une simple indirection.

5.1.2 Imposteur projectif

Une technique efficace pour accélerer le rendu d'un objet dans certaines situations consiste à en créer un *imposteur*, c'est-à-dire une simple image rendue au préalable et stockée dans une texture, puis affichée au moment du rendu à la place de l'objet original (par plaquage sur un polygone faisant généralement face à l'écran). Cette technique, qui fait partie des méthodes de rendu à base d'images, est généralement utilisée pour rendre des objets suffisamment lointains, car l'absence

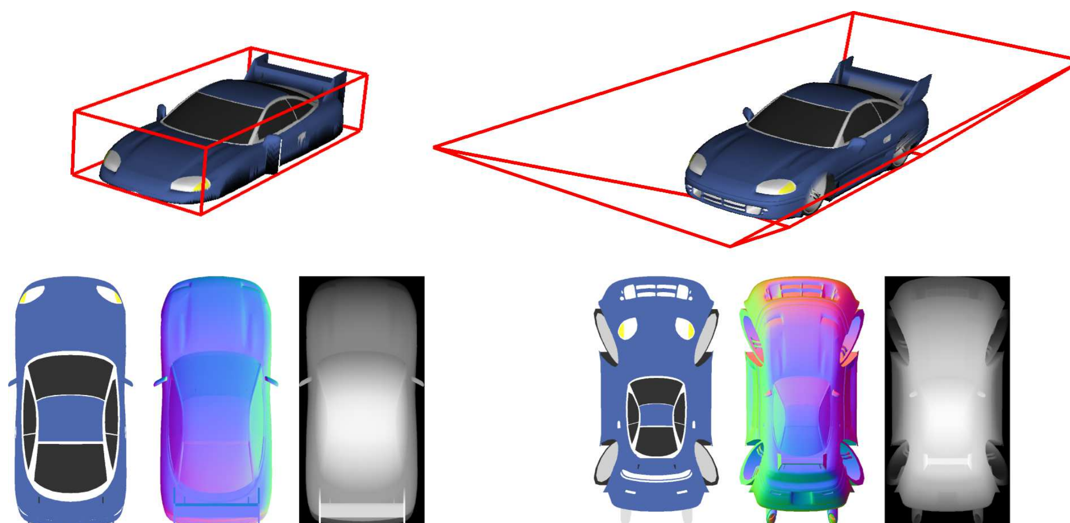


FIGURE 5.5 – Imposteur orthographique (à gauche) et imposteur projectif (à droite) : l'utilisation d'une perspective inversée peut permettre de capturer une plus grande partie d'un objet que ne le fait une simple projection orthographique.

de parallaxe devient gênante de près. C'est notamment très avantageux pour des objets complexes comme les arbres ou l'arrière plan d'un environnement urbain.

Pour faire face à l'absence de parallaxe, une solution couramment adoptée consiste à ajouter à l'image une information géométrique de profondeur, permettant ensuite au moment du rendu, par l'application de la déformation adéquate, de restituer correctement la parallaxe de l'objet. Cela revient à représenter la surface de l'objet par un champ de hauteur. Le principal défaut est qu'on n'échantillonne ainsi que la partie visible de l'objet depuis un certain point de vue, ce qui produit au moment du rendu des trous aux endroits où des parties de l'objet cachées dans la vue précalculée deviennent visibles.

C'est là que l'utilisation pour la capture de l'imposteur d'une projection non pas simplement orthographique mais perspective, éventuellement inversée, peut s'avérer avantageux, puisqu'elle permet dans de nombreux cas de capturer plus d'information.

Prenons l'exemple d'une voiture : avec une unique projection orthographique, quelle que soit la direction de vue utilisée pour la capture, une partie importante de l'objet manquera. Au contraire, si l'on positionne le centre de projection sous la voiture, et que l'on capture le champ de hauteur obtenu avec une projection perspective inversée, on en obtient un bon échantillonnage (voir figure 5.5).

L'*imposteur projectif* ainsi défini est une représentation très compacte de l'objet et en permet un rendu efficace, avec tous les avantages des algorithmes de raycasting sur GPU, notamment le coût de rendu proportionnel au nombre de pixels occupés à l'écran.

5.2 Combinaison de plusieurs reliefs

Nous venons de montrer que l'utilisation d'un unique relief offre une expressivité assez large, cependant ce n'est pas suffisant pour représenter n'importe quelle

surface. Pour pouvoir exprimer une surface quelconque sous la forme de relief il est possible d'utiliser plusieurs champs de hauteur. Nous décrivons ici les différentes stratégies possibles pour combiner plusieurs reliefs, dans le but de former une représentation plus expressive. Nous classons ces approches en deux catégories, selon que l'on cherche à représenter le relief d'une surface, éventuellement animée, ou un objet entier.

5.2.1 Relief d'une surface

L'utilisation la plus évidente du relief est la représentation du détail d'une surface. Nous nous plaçons dans le cas le plus général où la surface sur laquelle est plaqué le relief est dynamique. Dans le cas où cette surface est statique certains pré-calculs deviennent possibles pour rendre l'algorithme de raycasting plus efficace ou plus exact, comme nous l'avons montré au précédent chapitre.

Surface plane

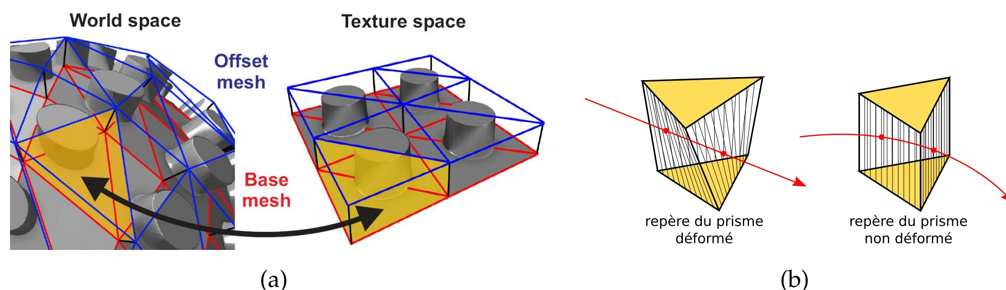
Le cas le plus simple de surface est le plan. Ce cas ne pose pas de problème particulier, il suffit d'utiliser un repère normalisé R_N dont le plan d'équation $z = 0$ se confond avec le plan sur lequel on cherche à ajouter du relief. Appliquer du relief à une surface plane peut être utile par exemple pour représenter le relief d'un mur de briques ou de pierres.

Surface courbe

Pour plaquer un relief sur une surface courbe il faut en déformer le volume par une déformation non linéaire. Si la surface est statique, et dans certaines conditions de déformation, il est possible d'exprimer le relief déformé comme un nouveau champ de hauteur défini par rapport à une surface plane. Cependant ce n'est pas possible dans le cas général, et comme nous avons supposé la surface dynamique il faudrait recalculer le champ de hauteur déformé avant chaque rendu, ce qui peut être coûteux.

Comme nous l'avons vu plus tôt, après déformation non linéaire, l'algorithme de raycasting ne peut plus être appliqué tel quel. Les deux problèmes auxquels on est alors confronté sont d'une part l'expression dans le repère R_N du rayon lumineux déformé, d'autre part l'intersection de ce rayon courbe avec le relief. Le rayon déformé pouvant prendre *a priori* une forme arbitraire, il n'existe pas de solution exacte à ces deux problèmes. L'intersection du relief avec le rayon déformé est généralement calculée en effectuant une approximation linéaire par morceaux de ce dernier. Pour calculer le rayon déformé deux approches existent.

Courbure locale La première méthode [PHL91, LP95, WWT^{+}03, OP05] consiste à supposer la courbure localement constante : la courbure de la surface est évaluée au point d'entrée du rayon de vue, et le rayon lumineux est déformé en conséquence,



(a) (b)

Figure 5.6 – Technique du *shell mapping* : (a) correspondance entre repère monde et repère normalisé (image tirée de [JMW07]); (b) difficulté liée à la déformation trilinéaire définie par un prisme : les droites ne sont pas conservées.

par l'application d'une courbure inverse. Cela revient à approximer localement la surface par une quadrique, le rayon lumineux déformé étant alors une parabole.

Cette méthode présente le problème de n'être que locale : des artefacts importants peuvent se produire aux endroits où la courbure de la surface varie fortement, cela est partiellement résolu par Na et Jung [NJ08], à condition de disposer d'une paramétrisation continue de la surface complète de l'objet.

Courbure définie par morceaux Une autre alternative consiste à définir la déformation de la surface par morceaux, en supposant que l'on sache calculer dans chaque morceau la déformation du rayon lumineux. La méthode généralement utilisée, appelée *shell mapping* [Ney98, WTL+04, PBFJ05], consiste à extruder chaque triangle du maillage de la surface en un prisme (selon les normales aux sommets du triangle), de manière à former un volume continu dans lequel le relief est placé. Ainsi la déformation globale du rayon lumineux est définie par l'ensemble des prismes qu'il intersecte, la déformation de chacun de ces prismes permettant de déterminer la déformation locale du rayon sur le segment correspondant.

L'extrusion d'un triangle le long d'une direction distincte par sommet (généralement les normales associées à ces sommets) forme un prisme dont les trois cotés sont des surfaces quadriques. La transformation utilisée à l'intérieur du prisme pour y placer le relief est une transformation trilinéaire, qui ne conserve pas les droites (voir figure 5.6).

Nous ne rentrerons pas plus dans les détails car cela sortirait du sujet traité dans ce chapitre, mais notons quand même que cette approche présente plusieurs problèmes :

- selon la surface, les prismes ainsi extrudés peuvent se recroiser;
- les prismes doivent être rendus séparément, ce qui pose problème au niveau des jonctions non planes (surfaces quadriques);
- la transformation trilinéaire ne conserve pas les droites ce qui complique le calcul d'intersection du rayon;
- la transfomation globale du relief ainsi définie n'est pas C^1, car les transformations trilinéaires à la jonction entre deux prismes ne se raccordent pas de manière C^1.

Une solution au premier problème est donné par Peng et al. [PKZ04], nécéssitant un précalcul, alors que les trois autres problèmes sont partiellement résolus par Jeschke et al. [JMW07], bien que l'algorithme qui en résulte soit relativement coûteux et difficilement utilisable pour une application de temps-réel.

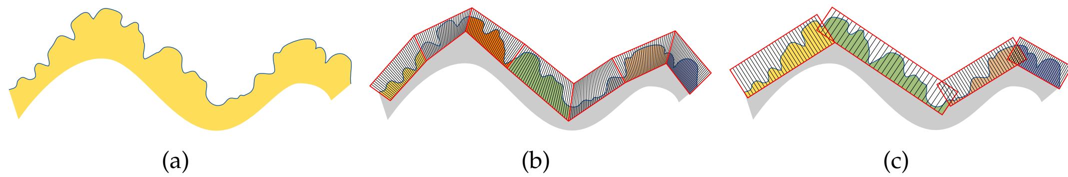


FIGURE 5.7 – Stratégies de combinaison entre plusieurs reliefs : (a) recollage continu : les faces des volumes englobants coïncident; (b) recollage discontinu.

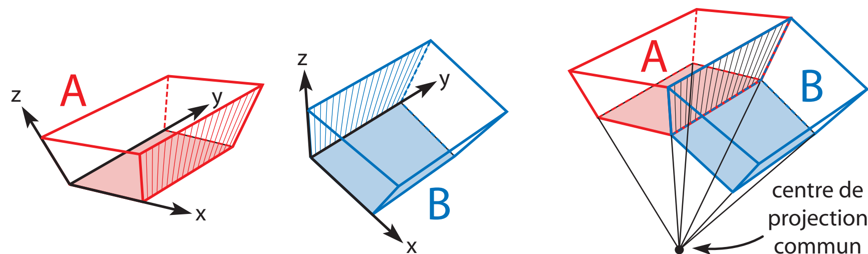


FIGURE 5.8 – Le recollage continu entre deux volumes de reliefs A et B nécessite que les centres de projection de leur transformation respective coïncident.

5.2.2 Capture d'un objet entier

Nous étudions enfin comment plusieurs champs de hauteur peuvent être combinés pour représenter un objet statique complet.

Recollage continu

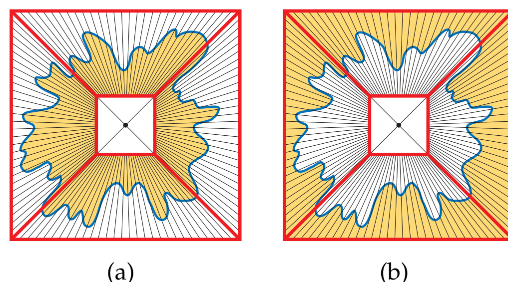
La première option qui s'offre pour combiner plusieurs champs de hauteur consiste à appliquer un recollage continu, c'est-à-dire positionner les volumes de reliefs de sorte que leur jonction soit continue. Considérons par exemple deux champs de hauteur A et B de repères normalisés respectifs R_N^A et R_N^B, que l'on cherche à recoller le long de la face de A qui correspond au plan d'équation $x = 1$ dans le repère R_N^A. On dira que A et B se recollent continuement si et seulement si :

$$\forall\ (y,z) \in [0,1]^2, \quad (1,y,z)_{R_N^A} = (0,y,z)_{R_N^B}$$

Si cette propriété est vérifiée, il suffit d'assurer la continuité entre les deux champs de hauteur à leur jonction pour que la surface combinée soit elle-même continue.

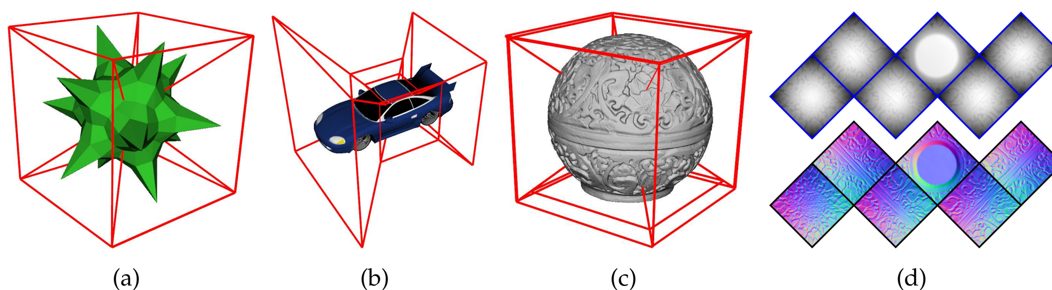
Dans le cas de volumes de reliefs définis par transformation projective, imposer un recollage continu implique une contrainte importante : les centres de projection coïncident. En effet, il est facile de voir que la contrainte de recollage continu à la jonction entre deux reliefs implique que les droites de hauteur à cette jonction coïncident, et donc il en va de même pour leur point de concours (voir figure 5.8). Cette propriété tient aussi lorsque le centre de projection se trouve à l'infini, autrement dit lorsque les déformations sont définies par transfomation affine (puisque dans ce cas toutes les droites de hauteur doivent nécessairement posséder la même direction).

On en déduit donc que pour un ensemble de reliefs recollés continuement, on peut distinguer trois configuration possibles :



(a) (b)

FIGURE 5.9 – Le recollage continu de reliefs mène dans le cas général à un relief sphérique, qui permet de représenter des surfaces étoilées, (a) extérieures par perspective inversée; (b) intérieures par perspective directe.



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

FIGURE 5.10 – Représentation sous forme de relief sphérique : (a), (b) et (c) divers exemples d'objets; (d) cartes de hauteur (en haut) et cartes de normale (en bas) de l'objet (c).

- au moins un des reliefs est déformé par transformation affine, auquel cas le reste des reliefs aussi et les droites de hauteur sont parallèles;
- au moins un des reliefs est déformé par perspective directe, auquel cas le reste des reliefs aussi, et les centres de projection coïncident;
- au moins un des reliefs est déformé par perspective inverse, auquel cas le reste des reliefs aussi, et les centres de projection coïncident.

Le premier cas n'est pas très intéressant car la surface combinée pourrait être représentée à l'aide d'un seul champ de hauteur. Les deux autres cas sont plus intéressants car ils permettent à l'aide de plusieurs champs de hauteur de couvrir l'intégralité de la sphère des directions : une solution simple consiste, à la manière des *cube maps*, de partitionner cette sphère en six pyramides de directions, chacune correspondant à une face d'un cube, et d'associer à chacune de ces pyramides une transformation projective ayant pour centre de projection le centre du cube (voir figure 5.9).

Si l'on prend le cas de la perspective inverse, une telle représentation (faite de six champs de hauteur) permet de représenter de manière exacte (à l'échantillonnage près) n'importe quel objet à géométrie *étoilée*, c'est-à-dire contenant un point depuis lequel tous les points de la surface sont visibles. Même si les objets parfaitement étoilés sont rares, nombreux sont ceux pour lesquels ceci constitue une bonne approximation. Cette idée, proposée notamment par Schaufler et Priglinger [SP99], que nous avions aussi exploitée [BD06], a été appliquée par exemple par Risser [Ris07] pour rendre en temps-réel plusieurs centaines de milliers d'astéroides.

Si l'on inverse le sens de projection en prenant une perspective directe, on peut représenter l'intérieur d'un objet à géométrie étoilée. Cela peut être utile par exem-

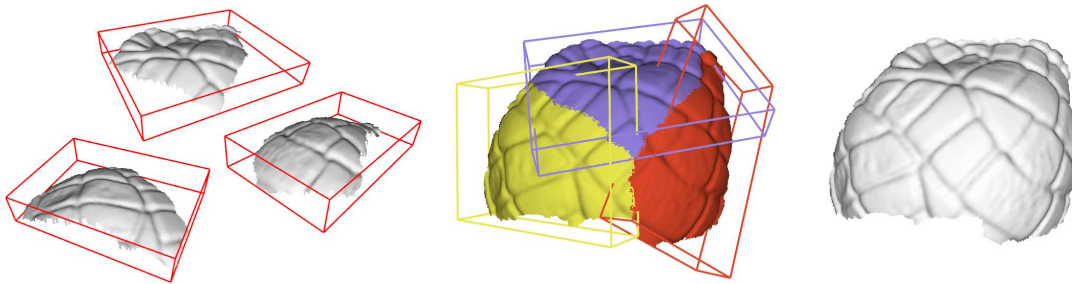


FIGURE 5.11 – Recollage discontinu de trois reliefs (à gauche) pour former une surface continue (à droite).

ple pour représenter un environnement proche, l'intérieur d'une pièce par exemple, ou une partie creuse d'un objet. Cette idée a été utilisée par Szirmay-Kalos et al. [SKALP05] pour calculer des réflexions ou des réfractions en utilisant une approximation de l'environnement sous forme de *cube map* de reliefs (les réflexions et réfractions ainsi obtenues sont plus précises que celles produites par un *environment mapping* classique, de par l'information de distance qui est utilisée).

Recollage discontinu

Nous venons de voir que le recollage continu entre champs de hauteur défini par transformation projective est contraignant, puisqu'il impose aux transformations d'avoir le même centre de projection. Bien que cela permette déjà de représenter un grand nombre d'objets, une surface quelconque n'est généralement pas approchable correctement par une surface étoilée. Il faut donc lever la contrainte de recollage continu, ce qui n'est pas gênant dans le cas d'objets statiques où la continuité entre les différents champs de hauteur utilisés peut être assurée au moment du calcul de la segmentation de l'objet en reliefs. Cependant, contrairement à un recollage continu, si l'on veut par la suite appliquer une transformation continue aux champs de hauteur, on n'aura pas la garantie que l'objet ainsi déformé reste continu.

Nous présentons ici trois stratégies possibles pour la segmentation d'un objet en champs de hauteur sans recollage continu.

Segmentation de la surface La première stratégie consiste à segmenter la surface de l'objet en parties connexes qui soient exprimables sous forme de champs de hauteur. Une solution est donnée par les *geometry textures* [dTWL07, dTWL08], où l'algorithme de *Variational Shape Approximation* (VSA) [CSAD04] est appliqué pour calculer une telle segmentation. Cela permet de représenter des objets potentiellement complexes avec relativement peu de champs de hauteur. La question de la segmentation en champs de hauteur reste cependant ouverte car l'algorithme VSA ne garantit pas que les éléments de surface de la partition calculée soient toujours exprimables comme champs de hauteur (bien que cela soit le cas en général).

Un algorithme simple de segmentation d'une surface en champs de hauteur est donné par Boubekeur et al. [BHGS06] où la surface de l'objet est subdivisée hiérarchiquement selon un octree. La subdivision d'une cellule s'arrête dès que la

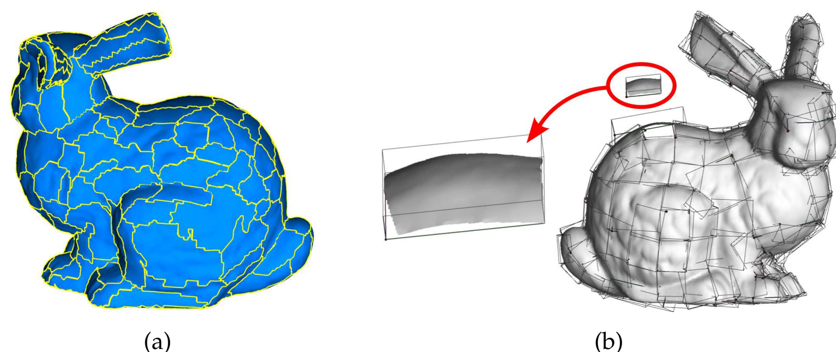


Figure 5.12 – Segmentation en reliefs à l'aide des *geometry textures* [dTWL07, dTWL08] : (a) segmentation de la surface par l'algorithme VSA; (b) rendu de la surface : chaque relief est rendu individuellement par raycasting à partir d'une boîte polygonale englobante.

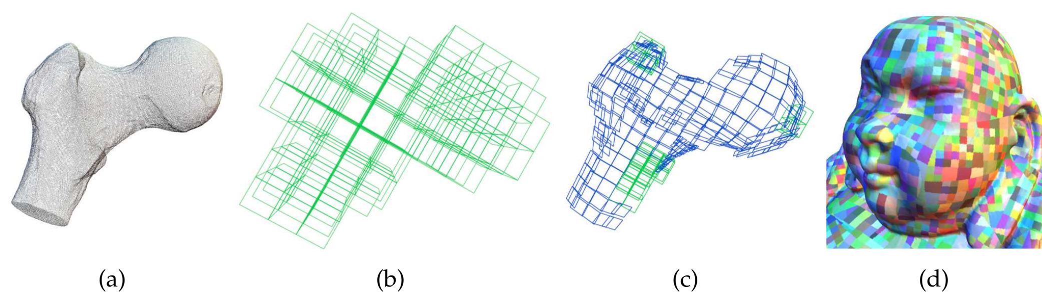


Figure 5.13 – Segmentation en reliefs à l'aide des *volume-surface trees* [BHGS06] : (a) une surface que l'on cherche à segmenter; (b) octree; (c) chaque cellule contient un relief; (d) autre exemple (les variations de couleurs représentent la segmentation) : le nombre de reliefs produit est beaucoup plus élevé que nécessaire (une telle surface est représentable à l'aide de moins d'une dizaine de reliefs).

surface de l'objet contenue dans celle-ci peut être exprimée comme un champ de hauteur. La partition ainsi définie est correcte (chaque morceau de surface est bien un champ de hauteur) mais *a priori* pas la mieux adaptée pour le rendu car le nombre de reliefs produits peut être plus élevé que nécessaire.

Segmentation du volume : plusieurs couches La seconde stratégie consiste à considérer le volume de l'objet et de le segmenter en utilisant plusieurs couches de champs de hauteur, partageant le même volume de relief [PO06, SdTG08]. Cette méthode a l'intérêt d'être très simple à mettre en œuvre, et permet de factoriser les calculs de raycasting avec les différentes couches. Cependant l'échantillonnage de la surface défini de cette manière est moins bon qu'avec la stratégie précédente, d'une part car selon la topologie de l'objet, certaines couches peuvent se trouver presque vides (nécessitant le stockage de textures contenant de grandes zones vides), d'autre part car les parties tangentes aux droites de hauteur sont sous-échantillonnées par rapport au reste de l'objet.

Segmentation selon les directions de vue La troisième stratégie consiste à partitionner l'ensemble des directions de vue de l'objet, en gardant pour chaque élément de cette partition le relief formé par la partie visible de l'objet dans la direction de

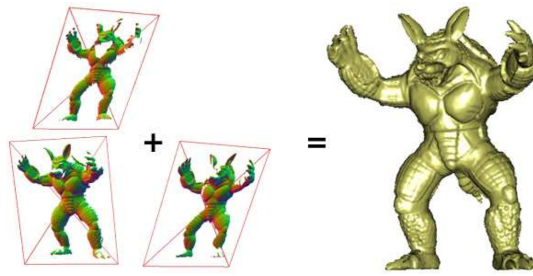


FIGURE 5.14 – Représentation d'un objet par les *omni-directional impostors*[ABB+07] : pour rendre l'objet selon un point de vue donné, les trois imposteurs correspondant aux points de vue les plus proches sont sélectionnés puis rendus par raycasting.

vue correspondante. La solution généralement utilisée consiste à répartir les vues uniformément sur la sphère des directions [Sch98], le cas le plus simple consistant à prendre les 6 directions principales défines par les trois axes d'un repère orthonormé [OBM00].

Une technique d'optimisation des directions d'échantillonnage est donnée par Andújar et al. [ABB+07], qui reste malgré tout basée sur des heuristiques et ne permet pas de garantir que toute la surface de l'objet est échantillonnée correctement.

Les problèmes qui peuvent donc se poser avec une telle approche sont d'abord la détermination de l'ensemble de vues optimales permettant de représenter l'objet (c'est-à-dire comment éviter les redondances tout en couvrant toute la surface de l'objet), puis ensuite au moment du rendu la selection des vues à rendre et la manière de les combiner là où les surfaces qu'elles définissent se chevauchent.

5.3 Conclusion

Nous venons de voir que le relief est une primitive de rendu qui offre de nombreuses possibilités de représentation. Notamment, outre l'utilisation classique pour représenter des terrains ou le détail d'une surface, il peut être utilisé pour obtenir des représentations efficaces autant du point de vue de la taille mémoire que du coût de rendu, pour des types d'objets très variés.

L'utilisation de cette primitive présente un fort potentiel pour le rendu de surfaces très détaillées, généralement représentées à l'aide de maillages comportant un nombre de polygones trop élevé pour pouvoir être rendus en temps-réel. La possibilité de rendre efficacement du relief grace aux algorithmes que nous avons présentés dans le chapitre précédent, qui possèdent notamment la propriété très avantageuse d'être *output sensitive* (c'est à dire un coût de rendu proportionnel à la place occupée à l'écran), en fait un bon condidat pour remplacer les représentations polygonales dans de nombreuses situations.

Cependant comme nous venons de le voir dans ce chapitre, il existe encore plusieurs problèmes ouverts, qu'il reste à résoudre afin d'exploiter pleinement le potentiel de cette représentation. Un premier problème est le manque de méthode automatique de segmentation d'une surface en reliefs. Des méthodes existent allant dans ce sens mais aucune ne possède la garantie de produire la meilleure segmentation (notion qu'il faudrait aussi définir, dépendant notamment de la méthode

utilisée pour le rendu) ou une segmentation exacte (c'est à dire pour laquelle toute la surface de l'objet est correctement échantillonnée). Un deuxième problème vient de la relative rigidité de cette représentation : la possibilité de pouvoir lui appliquer des déformations courbes étendrait fortement son pouvoir représentatif, notamment pour les objets non statiques. Rendre un relief déformé non linéairement demande un algorithme de rendu plus complexe, et il n'en existe actuellement aucun qui soit à la fois efficace et exact.

Il resterait donc à inventer une représentation à mi-chemin entre les *geometry textures* [dTWL07, dTWL08], efficaces pour le rendu mais complètement statiques, et le *smooth curved mapping* [JMW07], beaucoup plus flexible et permettant la représentation de surfaces animées mais beaucoup plus coûteux à rendre, ainsi qu'une méthode permettant le passage automatique d'une représentation classique d'une surface quelconque à cette nouvelle représentation.

Chapitre 6

Une application : éclairement réaliste dans un volume d'eau animé

Nous avons vu pour l'instant plusieurs moyens de représenter et rendre la complexité géométrique que contiennent les scènes réalistes. C'est une part importante de ce qui fait le réalisme d'une scène, permettant d'inclure les détails, les variations, les imperfections, la profusion présents aussi bien dans la nature que dans les environnements urbains.

Cependant cela ne suffit pas pour rendre une scène de manière réaliste, il faut aussi pour produire l'image finale pouvoir simuler l'interaction de la lumière avec la matière de la manière la plus conforme à ce qui peut être perçu dans la réalité. C'est l'autre partie de la complexité visuelle.

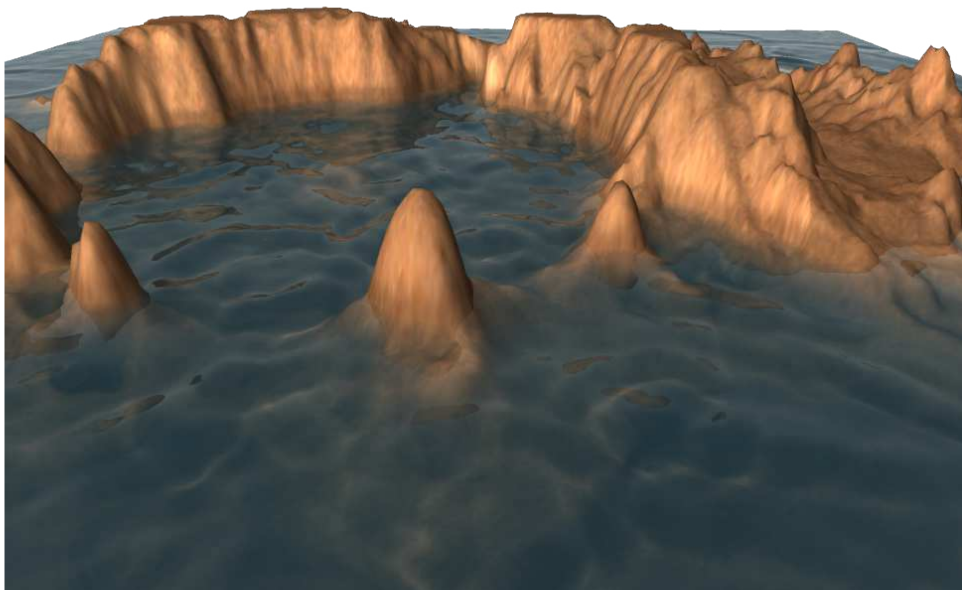


FIGURE 6.1 – Importance de l'éclairement dans la complexité visuelle : une image synthétisée à l'aide de la technique présentée dans ce chapitre.

De nombreux modèles d'éclairement, basés sur l'optique géométrique qui explique bien la plupart des phénomènes observables à l'oeil nu, décrivent comment

la lumière émise depuis des sources interagit avec la matière avant d'arriver jusqu'à l'oeil. Les phénomènes impliqués sont généralement globaux, dans le sens où l'expression locale de la radiance provenant d'un point d'une surface, dans une direction donnée, peut faire intervenir des mesures de radiance incidente provenant de toutes les directions, donc de surfaces ou de sources lumineuses situées arbitrairement loin du point considéré.

Une vaste gamme d'algorithmes existent qui essaient de simuler au mieux et le plus efficacement possible ces phénomènes, mais la complexité des calculs impliqués nécéssite généralement, dans une application en temps-réel, d'effectuer certaines approximations simplificatrices. Certains effets d'éclairement indirect sont plus perceptibles que d'autres et c'est donc généralement sur ceux-ci que les efforts d'intégration à un contexte de rendu en temps-réel se portent en premier. C'est notamment le cas des effets spéculaires que sont la réflexion, la réfraction et les caustiques.

Rendre un volume d'eau (et plus généralement un fluide transparent ou translucide) est un cas pratique où les phénomènes spéculaires sont particulièrement importants et où l'exactitude de la restitution qui en est faite influe directement sur le réalisme perçu. C'est d'autant plus important pour certaines scènes comme les paysages côtiers ou marins, pour lesquelles les surfaces d'eau constituent une partie importante de l'image à rendre.

C'est à ce problème particulier que nous nous attachons maintenant. Nous montrons comment la représentation à base de champ de hauteur ainsi que les algorithmes associés de rendu efficace (tirant parti des capacités du matériel graphique actuel), permettent directement et simplement de reproduire en temps-réel et de manière exacte les effets de réflexion, de réfraction et de caustiques produits par un volume d'eau. Nous montrons aussi comment certaines techniques que nous décrivons ici peuvent dépasser le simple cadre du rendu d'eau.

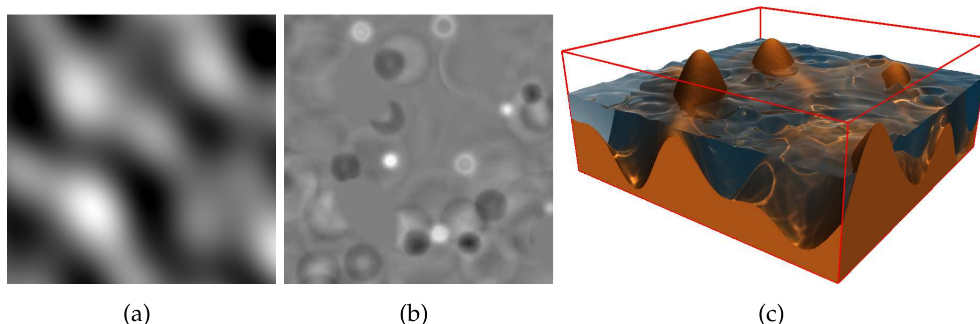
6.1 Représentation d'un volume d'eau

Nous représentons le volume d'eau à rendre comme le volume compris entre les surfaces de deux champs de hauteur h_{eau} et h_{fond}. Plus précisément, le volume d'eau V se définit comme suit :

$$V = \left\{(x,y,z) \in [0,1]^3 \ / \ h_{fond}(x,y) < z \leqslant h_{eau}(x,y)\right\}$$

Cette représentation est valide pour la plupart des situations rencontrées dans la nature, car les effets combinés du caractère liquide de l'eau et de la gravité terrestre impliquent une surface d'eau sans replis, donc représentable par un champ de hauteur, sauf dans des conditions particulières impliquant un état instable (par exemple lorsqu'une vague sur un océan se transforme en rouleau, surface non représentable par un champ de hauteur).

Nous supposerons par la suite que le champ de hauteur h_{fond} est statique (indépendant du temps), alors que h_{eau} pourra être dynamique pour permettre à la surface de l'eau d'être animée.



(a) (b) (c)

FIGURE 6.2 – Les cartes de hauteur de la surface de fond (a) et de la surface d'eau (b) définissent le volume d'eau (c).

Les éléments de l'algorithme de rendu que nous décrivons par la suite font tous appel au raycasting des surfaces S_{fond} et S_{eau}. On pourra donc pour la surface S_{fond} utiliser notre méthode avec précalcul, alors que pour S_{eau} on ne pourra utiliser que l'algorithme sans précalcul. Notons que la surface d'eau possédant généralement peu de détails fins de haute amplitude, l'application directe d'une recherche dichotomique d'intersection (comme la bissection) est parfois suffisante.

6.1.1 Animation de surface d'eau

Rendre une surface d'eau en mouvement suppose de disposer d'une technique d'animation réaliste. C'est un domaine de recherche à part entière, indépendant du problème du rendu. Nous passons ici simplement en revue les techniques existantes utilisables dans un contexte interactif comme celui que nous visons, et dont le résultat peut être facilement représenté sous la forme d'un champ de hauteur.

On sait depuis plusieurs siècles que le mouvement d'un fluide peut être expliqué par les équations de Navier-Stokes. Plus récemment, en calcul scientifique puis en informatique graphique, ces équations ont été utilisées à différents niveaux de précision en fonction des applications visées.

L'utilisation de représentations volumiques ou à base de particules permettent des simulations physiquement très réalistes, reproduisant des mouvements complexes comme les éclaboussures ou des vagues déferlantes, mais les temps de calculs impliqués sont généralement très longs [FF01, EMF02].

Dans le contexte de la modélisation de surfaces d'océans, ces équations débouchent sur le modèle de houle de Gerstner et Biesel, qui décompose les vagues à la surface de l'océan en une somme de trains d'onde à profils trochoïdaux. Des résultats réalistes ont été obtenus [FR86, HNC02] en combinant ce modèle avec d'autres modèles empiriques fournissant des expressions pour les spectres d'amplitude, de fréquence et de direction des vagues [PM64, HDE80], et des propriétés de diffraction des vagues proches des côtes [Pea86, TB87, Tes99, GM02]. La représentation de la surface d'eau utilisée par ces méthodes peut généralement s'exprimer sous la forme d'un champ de hauteur.

Dans le cas de surfaces d'eau bornées, le champ de hauteur peut être exprimé comme la solution de simples équations aux dérivées partielles [KM90], qui peu-

vent être résolues numériquement par des méthodes efficaces de diffusion locale [Gom00, Sta03].

Les approches spectrales [MWM87, Tes99, PA00] calculent dans un premier temps un spectre de Fourier de la surface d'eau en se basant sur des distributions mesurées empiriquement, puis le transforment en une carte de hauteur en appliquant un transformée de Fourier rapide (*FFT*). Le champ de hauteur obtenu est borné mais périodique.

Nous voyons donc qu'il existe une grande variété de méthodes, qui produisent généralement un résultat sous forme de champ de hauteur. Notamment la méthode proposée par Gomez [Gom00] peut s'implémenter très simplement et efficacement pour le GPU et permet de réagir en temps-réel à des impulsions appliquées de manière interactive à la surface de l'eau. C'est cette technique que nous utilisons dans notre implémentation et notamment pour les images de résultats montrées dans ce chapitre.

6.2 Travaux précédents

Rendre une volume d'eau de manière réaliste demande de bien prendre en compte les divers phénomènes d'éclairement impliqués. L'étude des interactions entre la lumière et l'eau consiste un domaine de recherche important en physique, notamment dans le domaine de l'océanologie. En informatique graphique plusieurs travaux se sont attelés à transcrire ces résultats de la physique en images de synthèse [Wat90, PA00, IDN03], réalistes mais généralement coûteuses à calculer. Des solutions temps-réel existent mais s'appuient généralement sur des hypothèses simplificatrices [NN94, HVT+06].

Les effets spéculaires que sont la réflexion et la réfraction sont importants à prendre en compte pour rendre correctement une surface d'eau. Jusqu'à récemment, seules les réflexions planes ou les réflexions courbes d'environnements théoriquement infiniment lointains pouvaient être synthétisées en temps-réel par *reflection mapping* ou *environment mapping* [BN76, Gre86]. Certains travaux s'appuyant sur un échantillonnage de l'espace des rayons lumineux [HLCS99, YYM05, LFY+06] permettent d'obtenir de bons résultats en temps-réel au prix d'un précalcul coûteux à stocker en mémoire. Plus récemment des approches utilisant des imposteurs en relief, éventuellement recalculés dynamiquement, et des algorithmes de raycasting sur GPU arrivent à obtenir des réflexions correctes en temps-réel [SKALP05, PMDS06]. D'autres approches s'appuient sur une déformation de la géométrie pour pouvoir la plaquer sur une surface courbe, généralement simple [RH06].

Les travaux existants sur le rendu d'effets de réfraction considèrent généralement l'objet isolé dans un environnement infiniment loin [GLD06]. Les travaux récents s'appuient de façon plus ou moins exacte sur une forme simplifiée de raycasting effectuée dans une représentation à base d'images de l'objet transparent et de la scène environnante [Wym05, DW07, HQ07, OB07].

Les caustiques sont la conséquence de réflexions et réfractions par des surfaces courbes, faisant se concentrer les rayons lumineux à certains endroits. Elles

sont généralement calculées à l'aide de *photon mapping* [Jen01] et peuvent être coûteuses à rendre. Des méthodes interactives existent, basées sur différentes stratégies d'émission et de rassemblement de photons [EAMJ05, DS06, WD06, SKP07, Wym08].

Un récent état de l'art recensant les techniques existantes relatives au rendu en temps-réel de réflexions, de refractions et de caustiques en fait une classification intéressante [SKUP+09].

6.3 Modèle d'éclairement simplifié

L'algorithme de rendu que nous proposons s'appuie sur une combinaison simplifiée de *raytracing* et de *photon mapping*. La phase de raytracing permet en lançant un rayon depuis l'oeil de déterminer quelles sont les surfaces visibles après réflexion et réfraction. L'éclairement de la surface de fond, *a priori* complexe à cause de la réfraction subie par les rayons provenant de la source lumineuse avant d'atteindre cette surface, sera calculé par une phase de *photon mapping* (qui appartient à la classe des algorithmes de *forward rendering*).

Nous effectuons les hypothèses suivantes :
- le rayon réfléchi n'intersecte pas la surface d'eau une seconde fois;
- le rayon réfracté ne ressort pas de l'eau, il intersecte toujours la surface de fond;
- le lumineux ne subit pas d'effet de diffusion multiple (*multiple scattering*) dans l'eau : il la traverse en ligne droite;
- la surface de fond possède une réflectance lambertienne.

Les deux premières hypothèses permettent d'éviter de devoir effectuer un lancer de rayons récursif coûteux. Elles sont acceptables car des effets de réflexion secondaires (après deux rebonds) sur une surface irrégulière comme celle de l'eau sont généralement imperceptibles, et l'effet d'inclinaison des rayons induit par la réfraction les dirige vers le bas, ne leur laissant qu'une faible probabilité d'intersecter la surface d'eau une seconde fois. Encore une fois c'est un effet secondaire qui est difficilement perceptible. Les radiances obtenues pour ces deux rayons sont combinées par le terme de Fresnel dont une bonne approximation est donnée par Schlick [Sch93].

La troisième hypothèse permet d'effectuer un simple raycasting à l'intérieur du volume d'eau pour obtenir la radiance issue du fond. Elle n'empèche cependant pas d'utiliser un modèle réaliste de diffusion simple (*single scattering*) permettant de rendre les variations de couleur produites par le parcours du rayon à travers l'eau, facteur important de réalisme. Cela suppose aussi que le volume d'eau ne contienne rien d'autre que ce qui est représenté dans la carte de hauteur pour S_{fond} (on peut ainsi inclure des objets fixes au fond de l'eau, comme des cailloux par exemple, mais on ne pourra pas rendre un objet comme un poisson en mouvement à l'intérieur de l'eau).

Enfin la quatrième hypothèse nous permet de recourir à une version simple de photon mapping pour calculer l'éclairement de la surface de fond. Ainsi cet éclairement pourra être échantillonné dans une *carte d'éclairement* (*illumination map*) sur

une idée similaire à celle présentée par Arvo [Arv86]. Supposer la réflectance lambertienne permet d'accumuler dans cette carte l'intensité lumineuse des photons incidents, pondérée par le cosinus de leur angle d'incidence, indépendamment de l'angle de vue qui sera utilisé pour le rendu final. De plus nous supposerons que l'éclairement est dû à une source ponctuelle unique, plus éventuellement un terme d'éclairement ambiant, bien que la méthode que nous allons présenter puisse ensuite être facilement étendue à la prise en compte de plusieurs sources lumineuses, éventuellement directionnelles (mais pas surfaciques).

6.4 Éléments à calculer

Le modèle d'éclairement ainsi défini demande, pour un rayon de vue intersectant la suface de l'eau, la prise en compte de quatre phénomènes lumineux : la réflexion, la réfraction, l'absoption lumineuse à travers l'eau et enfin les caustiques produites au fond de l'eau. Nous détaillons maintenant comment sont calculés chacuns de ces phénomènes.

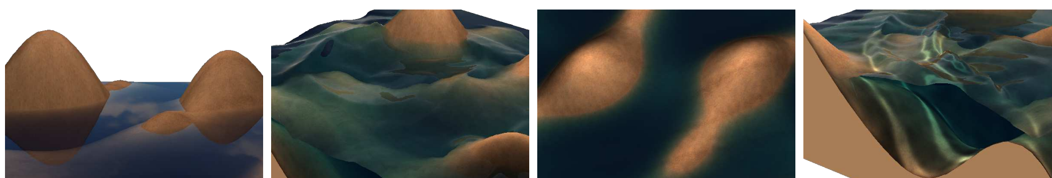


FIGURE 6.3 – Les quatres phénomènes lumineux importants à calculer dans une volume d'eau : (a) réflexion, (b) réfraction, (c) absorption, (d) caustiques.

6.4.1 Réflexion exacte

La surface d'eau n'étant généralement pas plane, l'ensemble des rayons obtenus par réflexion des rayons de vue issus de la caméra ne peut pas être représenté correctement comme un ensemble de rayons de vue issus d'une seule caméra projective classique puisqu'ils ne sont génералement pas concourants (dans le cas particulier d'une surface plane, les rayons de vue réfléchis sont les rayons provenant de l'image par symétrie par rapport au plan de la caméra de vue, ce qui permet à une technique comme le *reflexion mapping* de s'appliquer). Dans le cas de surfaces courbes, les applications temps-réel utilisent généralement une approximation grossière consistant à déformer l'image obtenue par réflexion plane et plaquer le résultat sur la surface de l'eau. Cette technique ne fonctionne que dans des cas restreints et demande un certain travail d'ajustement pour un résultat de toutes façons peu réaliste.

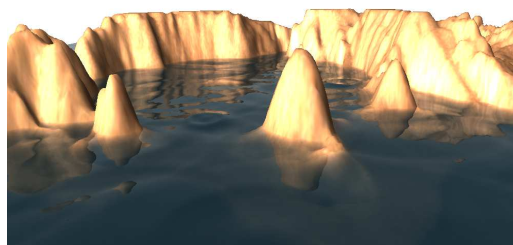


FIG. 6.4 *Réflexions sur une surface d'eau.*

L'application de notre algorithme de raycasting de champ de hauteur permet de calculer de manière exacte la réflexion du relief environnant, encodé dans la carte de hauteur de S_{fond}, sur la surface de l'eau.

Il suffit pour ceci pour chaque rayon de vue intersectant S_{eau} de calculer le rayon réfléchi correspondant (en lisant la normale en ce point dans la carte de normales de S_{eau}, calculée dynamiquement chaque fois que la carte de hauteur de cette surface est actualisée) et de chercher son intersection avec la surface S_{fond}. On peut pour ceci utiliser l'algorithme robuste avec précalcul puisque S_{fond} est statique. Il faut noter cependant que lors du précalcul de la carte de rayons de sûreté, tous les rayons lumineux doivent être pris en compte (intérieurs et extérieurs), puisque le rayon produit par réflexion à la surface de l'eau est un rayon intérieur (il a pour origine un point intérieur au volume de relief). Lorsque le rayon n'intersecte pas le relief environnant et ressort du volume de relief, on peut faire appel à une carte d'environnement (*environment map*) échantillonnée au préalable (en précalcul ou dynamiquement) dans une *cube-map*.

6.4.2 Réfraction exacte

De la même manière que pour la réflexion, la réfraction par une surface courbe est généralement difficile à synthétiser à l'aide d'un rendu par rastérisation, car la réfraction d'un ensemble de rayons provenant d'un même point de vue n'est généralement pas un ensemble de rayons lumineux concourants et donc pas représentable par une projection linéaire unique. Là encore les applications en temps-réel recourent souvent à des approximations grossières peu réalistes. La perception humaine étant assez sensible aux effets induits par la réfraction (et à leur éventuelle absence) il est important de pouvoir les rendre de manière exacte.

FIG. 6.5 *Distortions dues à la réfraction.*

Comme pour la réflexion, c'est très naturellement que notre algorithme de rendu de relief permet d'effectuer ce calcul. Il suffit de déterminer la direction du rayon réfracté (par la loi de Snell-Descartes, faisant intervenir le rapport d'indices de réfraction et la normale à la surface de l'eau : on ignore ici le phénomène de dispersion lumineuse, dû à la dépendance de cet indice par rapport à la longeur d'onde) et d'intersecter le rayon obtenu avec la surface S_{fond} (ou l'environnement extérieur dans certains cas rares où le rayon ressort du volume de relief sans n'avoir rien intersecté).

Les coordonnées au sol (2D) ainsi obtenues sont utilisées pour récupérer dans la carte d'illumination l'irradiance au point intersecté sur la surface de fond.

6.4.3 Absorption lumineuse

Un élément important pour le rendu d'eau (et le rendu de liquides de manière plus générale) est de bien prendre en compte l'absorption lumineuse prenant effet le long du trajet effectué par un rayon lumineux à travers le volume. Les effets parfois subtils apportés ajoutent en fait beaucoup au réalisme perçu. C'est notamment important au niveau des rivages, où une absence de prise en compte de ce phénomène au moment du rendu produit une frontière nette entre l'eau et le rivage, affectant ainsi le réalisme de l'image.

(a) (b)

FIGURE 6.6 – Importance de l'absorption lumineuse pour le réalisme : (a) une image tirée du jeu Oblivion, avec des discontinuités nettes aux bords de l'eau (zones entourées), (b) une image générée avec la technique présentée.

En s'appuyant sur des modèles empiriques de la diffusion de la lumière dans l'eau, bien connus dans la littérature sur l'océanographie, Premoze et Ashikhmin [PA00] proposent un modèle simple permettant pour une longueur d'onde λ donnée d'exprimer la radiance transmise en un point p_e à la surface de l'eau depuis un point p_f situé à une profondeur z sous cette surface :

$$L_\lambda(p_e,\vec{\omega}) = \alpha_\lambda(d,0)L_\lambda(p_f,\vec{\omega}) + (1-\alpha_\lambda(d,z))L_\lambda^d$$

$$\text{avec} \quad \begin{cases} d &= \|p_e - p_f\|_2 \\ z &= p_e|_z - p_f|_z \\ \vec{\omega} &= \text{direction du vecteur } \overrightarrow{p_f p_e} \end{cases}$$

Le premier terme de cette équation correspond à la radiance provenant du point p_f dans la direction $\vec{\omega}$, $L_\lambda(p_f,\vec{\omega})$, atténuée par la distance traversée. Cela correspond à l'éclairement de la surface S_{fond}, nous expliquons par la suite comment le calculer.

Le deuxième terme correspond à la contribution due à la diffusion multiple à travers l'eau, faisant intervenir une radiance diffuse constante L_λ^d calculée à partir de mesures de l'éclairement provenant du soleil et du ciel (ou dans le cas d'un environnement fermé, des principales sources lumineuses et de l'éclairement ambiant).

Le coefficient $\alpha_\lambda(d,z)$ décrit l'attenuation exponentielle que subit la lumière en fonction de la distance d et de la profondeur z traversées :

$$\alpha_\lambda(d,z) = e^{-a_\lambda d - b_\lambda z}$$

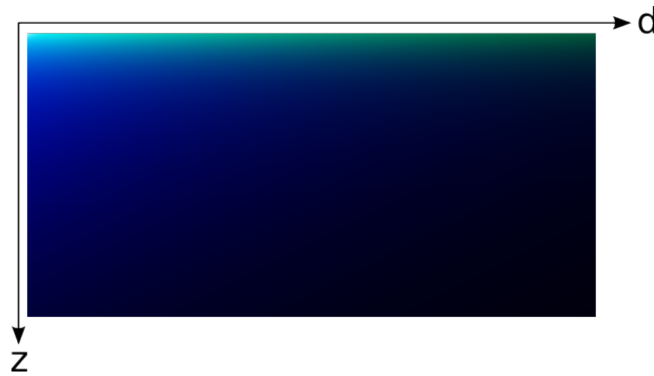


FIGURE 6.7 – Variations de couleur dues à l'absorption lumineuse, en fonction de la distance et de la profondeur traversées, obtenues pour des valeurs fixées de a et b, avec $L^d = (0,0,0)$.

En pratique ces calculs sont effectués dans l'espace de couleurs RGB. Bien que théoriquement il faudrait utiliser une discrétisation fine des longueurs d'ondes lumineuses, ceci donne de suffisamment bons résultats. L'absorption lumineuse est alors contrôlée par trois couleurs définies en RGB :

$a = (a_R, a_G, a_B)$	contrôle l'attenuation en fonction de la distance;
$b = (b_R, b_G, b_B)$	contrôle l'attenuation en fonction de la profondeur;
$L^d = (L^d_R, L^d_G, L^d_B)$	définit la couleur diffuse de l'eau causée par l'éclairement ambient et les propriétés de diffusion de l'eau.

Le calcul de α_λ peut se faire analytiquement de façon peu coûteuse en profitant des opérations vectorielles composantes par composantes disponibles sur le GPU (somme, multiplication et exponentielle). Il est aussi possible d'échantillonner cette fonction dans une texture 2D.

Notons enfin que dans le cas du rendu d'océans ou d'eaux troubles, où le fond de l'eau n'est visible que pour des profondeurs faibles, ce modèle d'atténuation exponentielle permet de calculer la distance à partir de laquelle il est inutile de continuer à avancer le long du rayon de vue, puisque le fond ne sera de toute façon pas visible. Plus précisément on cherche à déterminer la profondeur d_{max} telle qu'un rayon traversant une distance supérieure à cette valeur sera atténué d'un facteur inférieur à ϵ :

$$\forall\lambda, \forall z > 0, \forall d > d_{max} \quad \alpha_\lambda(d,z) < \epsilon \iff \forall\lambda \quad e^{-a_\lambda d_{max}} < \epsilon$$

On a alors

$$d_{max} = \max_\lambda \left(-\frac{\log(\epsilon)}{a_\lambda} \right) = \frac{\log(\epsilon)}{\min_\lambda a_\lambda}$$

Connaître cette valeur permet d'éviter tout calcul d'intersection inutile et ainsi d'accélérer le rendu quand les propriétés d'absorption lumineuse de l'eau le permettent.

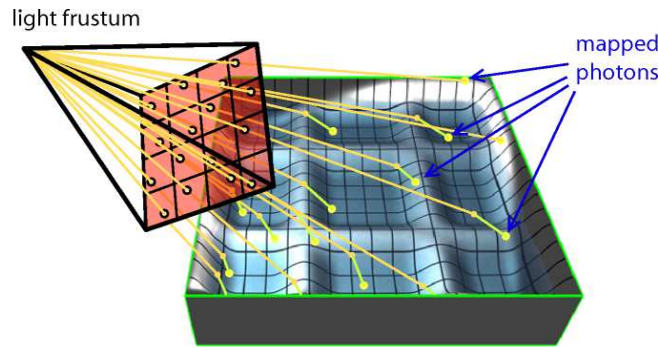


FIGURE 6.8 – Les photons sont émis depuis la source lumineuse en passant par les positions perturbées des pixels du frustum contenant le cône de lumière, avant d'être déposés sur la surface de fond.

6.4.4 Caustiques

Pour calculer l'éclairement en un point de la surface du fond de l'eau, il faut être capable d'estimer la quantité de lumière issue de la source, qui l'atteint après réfraction par la surface d'eau. La surface d'eau n'étant pas plane, plusieurs rayons (ayant suivi des chemins différents) peuvent converger en un même point sous l'eau, donnant ainsi naissance à un phénomène de caustiques. Pour un point donné, estimer l'ensemble des chemins lumineux qui y aboutissent en partant de la source est un problème difficile, d'autant plus qu'il faut aussi prendre en compte les éventuelles occultations produites par les parties saillantes du relief, causes d'ombres refractées.

Il est donc plus simple de recourir à une approche inverse, consistant à suivre les chemins lumineux provenant de la source et ajouter localement aux surfaces rencontrées l'illumination correspondante. C'est le principe du *photon mapping* [Jen01]. Un ensemble de *photons* est lancé depuis la source lumineuse, chacun portant une fraction de l'intensité lumineuse de celle-ci, ces photons subissent lors de leur parcours les lois de l'optique (réflexion, réfraction) avant d'être déposés sur les surfaces de la scène. Deux principaux problèmes se posent alors : d'abord celui de la bonne manière de stocker les photons pour pouvoir ensuite efficacement y accéder, ensuite celui du regroupement de ces photons (*photon gathering*) et du filtrage de leurs intensités pour le rendu de l'image finale.

Dans notre cas, la surface à éclairer est la surface de fond du champ de hauteur, qui par construction est paramétrique : un point (x, y, z) de cette surface peut être représenté de manière unique par ses coordonnées x et y. C'est cette paramétrisation que nous prendrons pour représenter la position des photons.

Pour lancer les photons nous effectuons un rendu en fausses couleurs du volume d'eau depuis la position de la lumière (en prenant pour *frustum* le cône de lumière). Chaque pixel de l'image rendue (que nous appelerons la *carte de photons*) correspond à un photon qui après refraction va se projeter sur S_{fond}, les canaux RGB servant à stocker les coordonnées (x, y) du photon et son intensité lumineuse (pondérée par le cosinus de l'angle d'incidence). Pour minimiser les problèmes de moiré (*aliasing*) dus à l'échantillonnage régulier des rayons lumineux de la source, causé par le passage de ces rayons par une grille régulière, on effectue un *jittering* consistant à perturber aléatoirement chaque rayon d'une petite quantité autour de

sa position de base. Une texture 2D précalculée permet de stocker un vecteur de perturbation par rayon de vue. En fait par cette méthode on pourrait même utiliser n'importe quel échantillonnage des rayons lumineux issus de la source. Il est possible aussi d'ajouter à cette texture une information de masquage permettant à la fois de définir la forme de la source lumineuse (un spot circulaire au lieu de la pyramide carrée par défaut) et de pondérer l'intensité des photons en fonction de leur fréquence d'échantillonnage et de l'intensité lumineuse directionnelle de la source lumineuse.

Une fois cette carte de photons calculée, il faut l'inverser, c'est-à-dire calculer une *carte d'illumination* plaquée sur la surface de fond, représentant dans chaque texel la quantité d'énergie lumineuse apportée par les photons incidents (ce qui revient à compter le nombre de photons présents dans chaque texel, grossièrement car les photons peuvent avoir des intensités différentes), à la manière de l'*illumination map* proposée par [Arv86]. Cela revient à calculer un histogramme à deux dimensions de la carte de photons (selon ses composantes R et G).

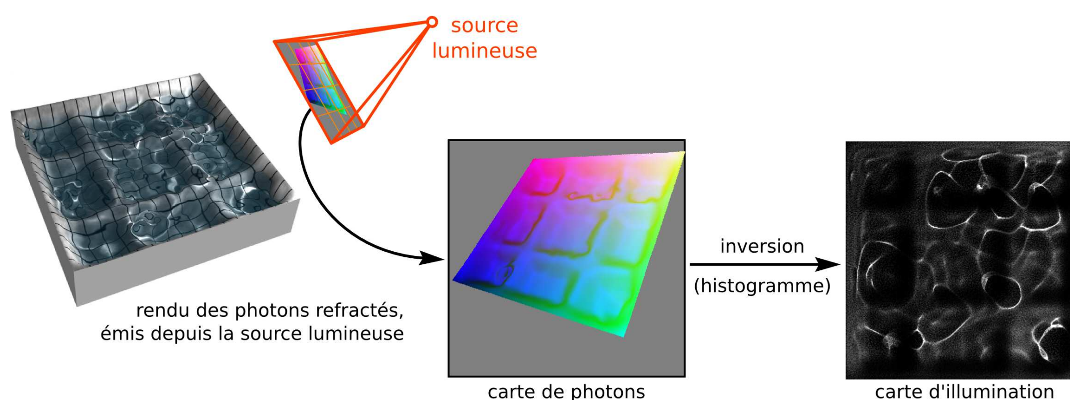


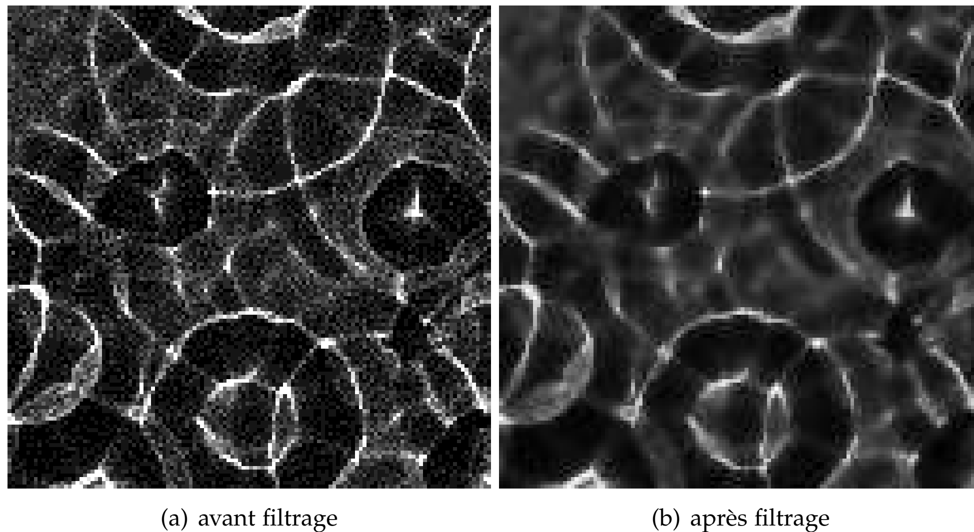
FIGURE 6.9 – Les étapes du calcul de la carte d'illumination.

C'est un problème de *gathering*, qui à l'époque où nous avons développé cette technique était difficilement implémentable sur GPU. Nous avons donc opté pour la solution hybride suivante : la carte de photons est rapatriée en mémoire du CPU, qui la parcourt complètement pour en créer l'histogramme 2D (en accumulant à chaque position (x, y) l'intensité des photons qui s'y projettent). La carte d'illumination ainsi créée est renvoyée en mémoire GPU et utilisée pour l'éclairage de la surface de fond.

Avec le matériel graphique actuel il est maintenant possible d'effectuer des accès texture efficacement dans le *vertex shader*, ce qui permettrait de résoudre le problème de l'accumulation des photons par *splatting* en effectuant le rendu d'une grille de points, un point par photon, déplacé par le *vertex shader* à la position donnée par la carte de photons.

En raison du nombre relativement faible de photons utilisés (du même ordre que le nombre d'échantillons de la carte d'illumination), la carte d'illumination obtenue est généralement bruitée. Il faut donc recourir à un filtrage de cette carte pour résoudre ce problème. Une caractéristique importante des caustiques est qu'elles contiennent des motifs de haute fréquence, importants pour le réalisme et

qu'il faut donc éviter de perdre par filtrage. Les filtres préservant les contours (*edge-preserving filters*), la diffusion anisotrope [PM90] ou le filtre bilatéral [TM98, DD02], présentent de bons candidats. Ce dernier a été utilisé pour le filtrage de *photon maps* [WMM+04] et peut être implémenté efficacement sur le matériel graphique actuel [CPD07, PD09]. Nous l'appliquons à la carte d'illumination après son transfert en mémoire GPU, en une passe rapide effectuée par un *fragment shader*. Cela permet de supprimer le bruit de la carte d'illumination tout en gardant les motifs fins caractéristiques des caustiques.



(a) avant filtrage (b) après filtrage

FIGURE 6.10 – Carte d'illumination obtenue avant et après filtrage bilatéral (résolution 128^2).

6.5 Algorithme complet

Si l'on récapitule l'algorithme complet, il consiste à effectuer à chaque nouveau rendu les trois étapes suivantes :

1. actualisation de la surface d'eau animée
2. calcul de la carte d'illumination :
   (a) tracé des photons par raycasting depuis la source
   (b) calcul de l'histogramme sur CPU
   (c) filtrage bilatéral sur GPU
3. rendu par raycasting du volume de relief :
   (a) intersection du rayon de vue avec S_{eau} et S_{fond}
   (b) si le rayon intersecte S_{eau} en premier
      i. intersection du rayon réflechi avec S_{fond}
      ii. intersection du rayon refracté avec S_{fond}, et récupération de la couleur de fond et de l'illumination à la position correspondante, puis pondération par l'absorption lumineuse en fonction de la distance traversée dans l'eau

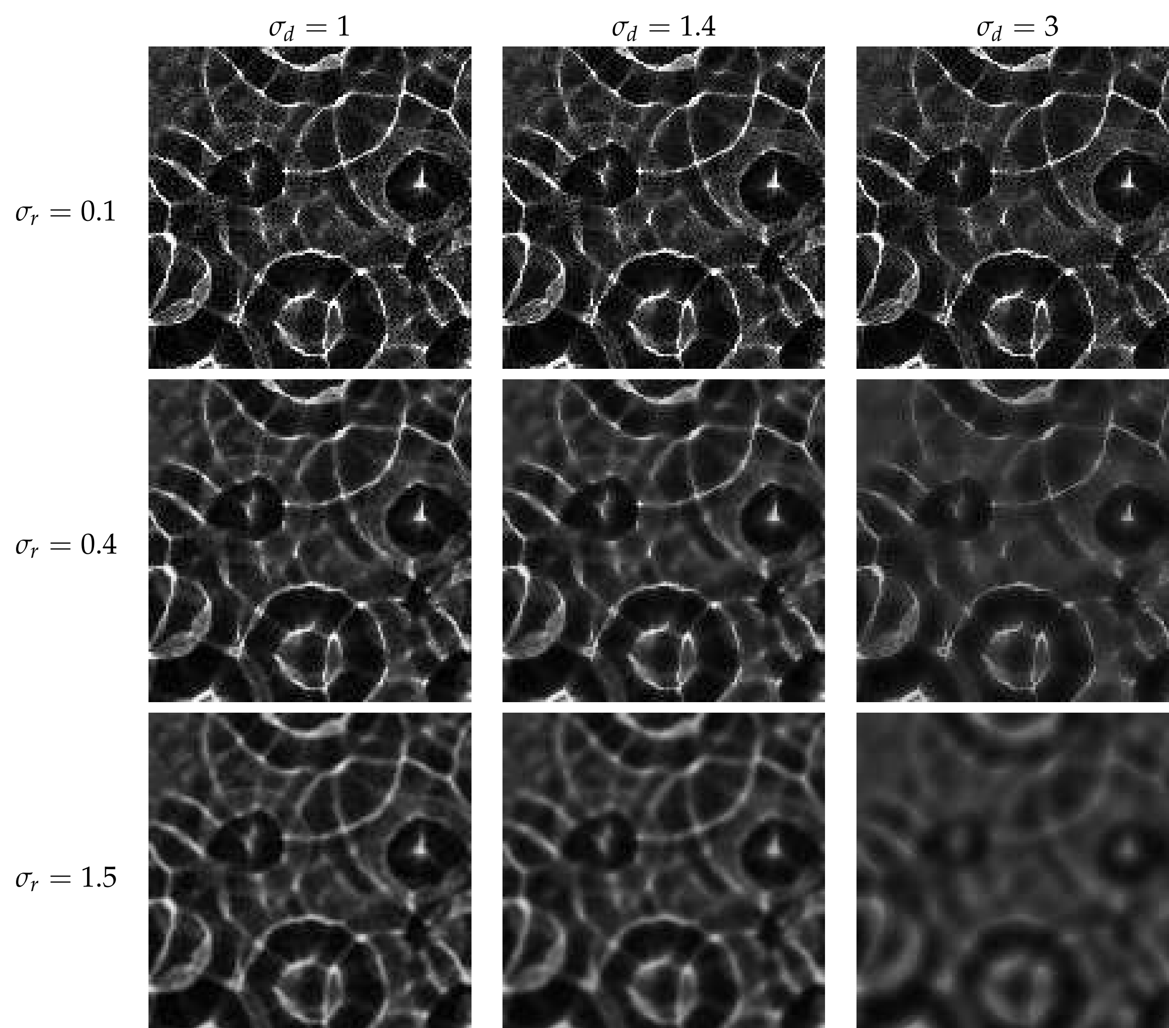


TABLE 6.1 – Carte de caustiques filtrée pour différentes tailles du filtre bilatéral, pour le domaine spatial (σ_d) et pour le domaine des intensités (σ_r). L'image centrale est un bon compromis entre l'élimination du bruit et la conservation des détails.

6.6 Résultats

Les résultats que nous donnons ici ont été obtenus avec une carte graphique GeForce 8600 et un processeur Intel Core 2 Quad Q6700 à 2.66 GHz. La méthode étant intrinsèquement *output sensitive*, nous avons pris comme référence le rendu d'un vue de 800×600 pixels complètement occupés par le volume d'eau, pour un angle de vue moyen d'environ 45 degrés. Un bon compromis entre précision de la carte d'illumination et temps de rendu est obtenu en utilisant une résolution de 128^2 pour la carte d'illumination et de 256^2 pour la carte de photons, on atteint alors une fréquence d'affichage de 26 Hz.

Nous avons déterminé expérimentalement qu'il faut utiliser environ quatre fois plus de photons que d'échantillons de la carte d'illumination pour obtenir une version suffisamment nette et non bruitée de cette dernière (après filtrage bilatéral). Le tableau 6.2 détaille les temps de calcul des différentes parties de l'algorithme complet, pour des résolutions des cartes de photons et d'illumination choisies de sorte à maintenir ce rapport de quatre. La première remarque que l'on peut tirer de

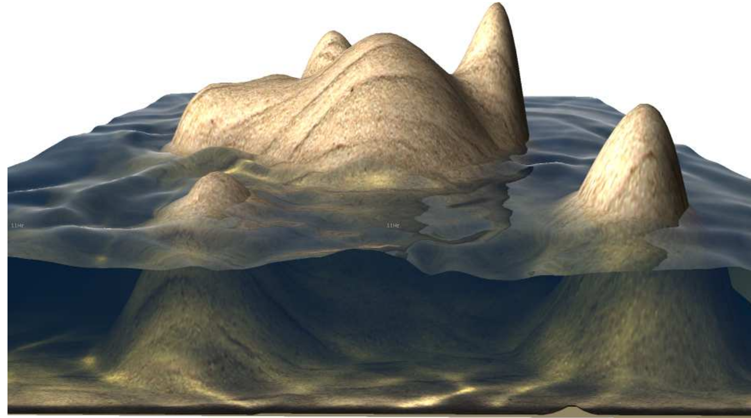


FIGURE 6.11 – Rendu en temps-réel produit par la technique présentée.

résolutions	anim.	photons	illum.	filtrage	rendu	total	fréq.
$(64^2, 128^2)$	0.4	0.64	0.45	0.3	33.65	35.77	28
$(128^2, 256^2)$	0.34	2.29	2.07	0.72	33.55	38.97	25.7
$(256^2, 512^2)$	0.45	7.77	10.26	2.46	33.5	54.44	18.4
$(512^2, 1024^2)$	0.4	29.47	102.66	8.72	33.65	174.9	5.7

TABLE 6.2 – Temps de rendus obtenus pour une vue de 800×600 pixels complètement occupés par le volume d'eau, pour une carte de hauteur de résolution 128^2 (eau et fond). La première colonne dénote les différentes résolutions utilisées pour la carte d'illumination et la carte de phtotons. Les colonnes suivantes donnent les temps en milisecondes pour le calcul de la surface d'eau animée, le calcul de la carte de photons, le calcul de la carte d'illumination, le filtrage bilatéral de celle-ci, et enfin le rendu final. L'avant dernière colonne donne le temps de rendu total en milisecondes, et la dernière la fréquence d'affichage atteinte en Hz.

ce tableau est que logiquement, les temps de calcul de la partie rendu proprement dite sont indépendants de la résolution de la carte d'illumination. Ensuite on peut remarquer que pour une résolution de la carte d'illumination inférieure ou égale à 256^2, c'est le temps de rendu qui prédomine, alors qu'au delà de cette valeur, le temps de calcul de la carte d'illumination devient largement prédominant, ce qui peut se comprendre par le fait qu'elle est effectuée par le CPU, mal adapté au traitement efficace de grandes images.

Bien que les temps de calculs obtenus pour des résolutions élevées de la carte d'illumination restent compatibles avec une utilisation interactive, le passage à plus grande échelle de la technique demanderait l'implémentation intégrale sur GPU du calcul de la carte d'illumination, qui représente pour l'instant le goulot d'étranglement de l'algorithme.

6.7 Conclusion

La technique que nous venons de présenter est une application efficace de la représentation par champs de hauteur : elle s'appuie sur l'efficacité de l'algorithme de raycasting de reliefs pour le processeur graphique pour produire en temps-réel des images très réalistes de volumes d'eau animés.

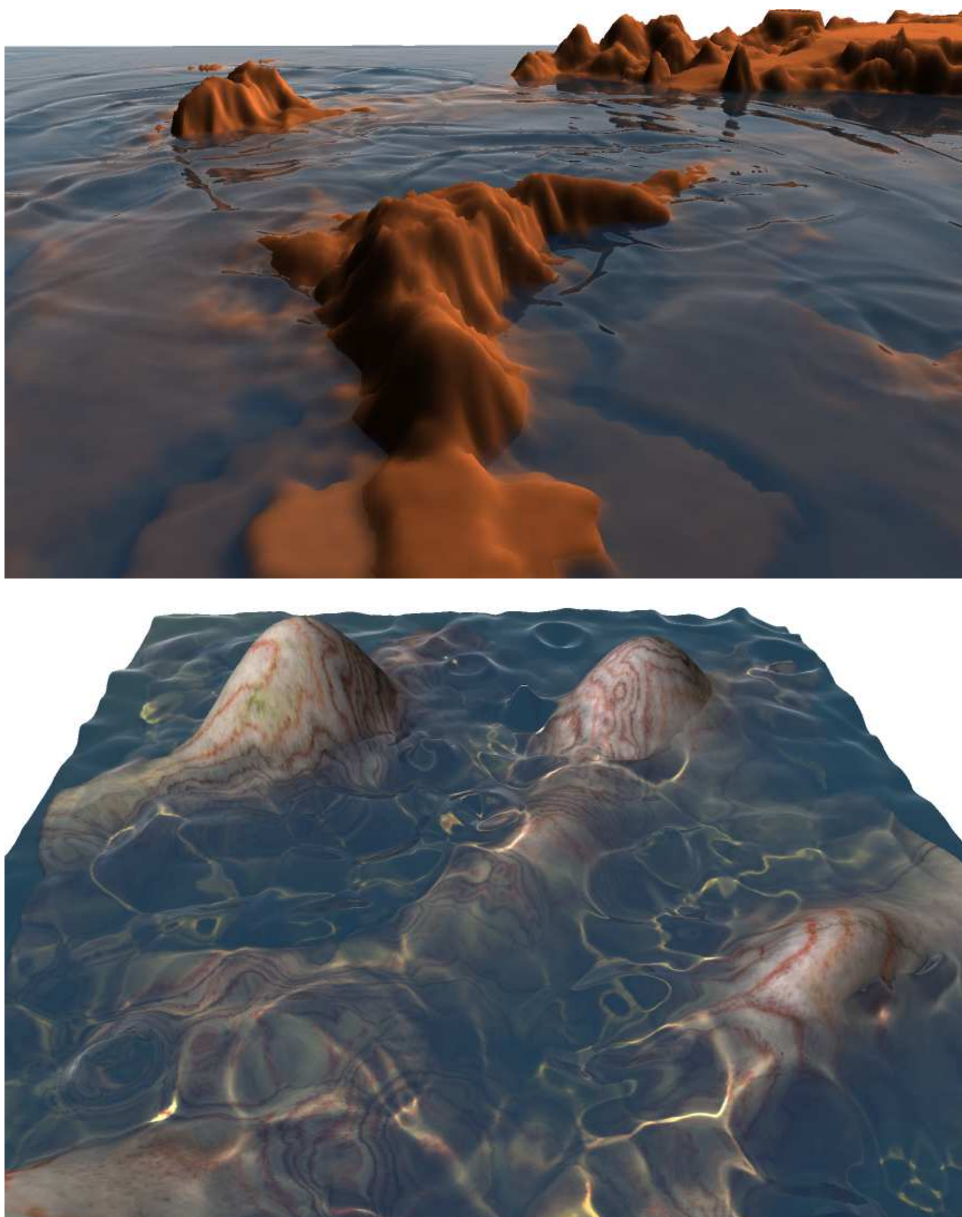


FIGURE 6.12 – Rendus en temps-réel produits par la technique présentée.

Certains des algorithmes présentés ici pourraient être généralisés à d'autres situations que le simple rendu d'eau, notamment le rendu d'effets spéculaires dans les objets translucides pouvant être représentés sous forme de champs de hauteur (voir les nombreuses possibilités présentées au chapitre précédent).

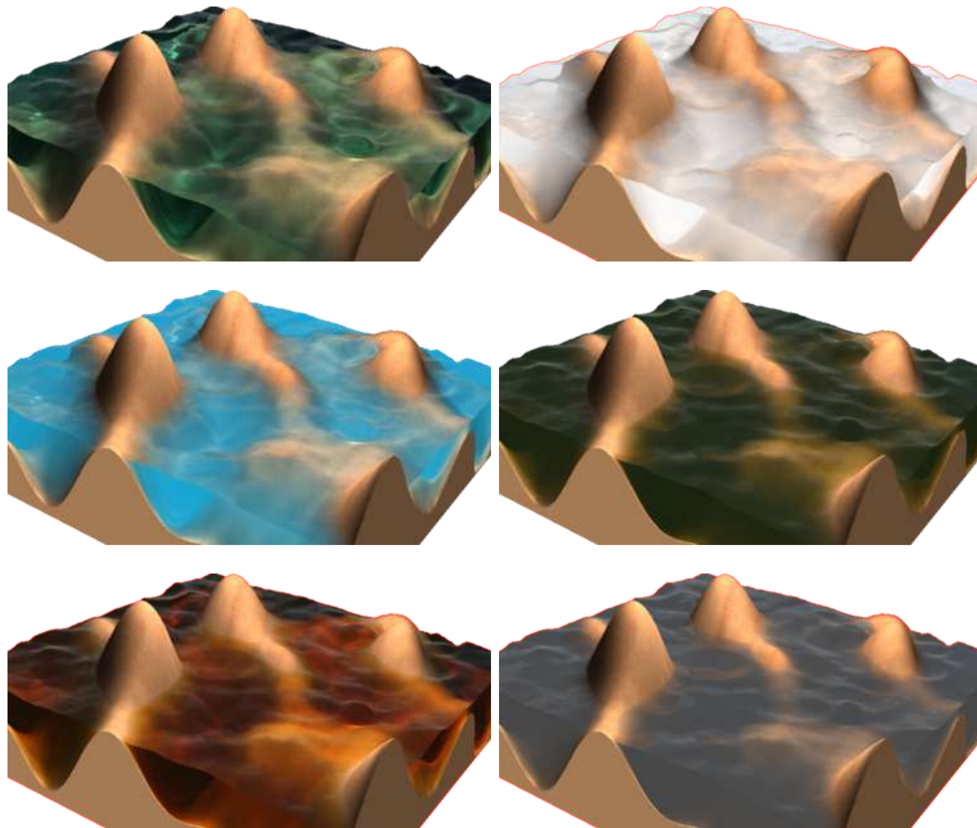


FIGURE 6.13 – Différents aspects du volume d'eau obtenus en faisant varier les paramètres d'absorption a et b.

Troisième partie

Représentations non géométriques

Chapitre 7

Représentations par échantillonnage de la parallaxe

Nous avons analysé en première partie les différentes stratégies possibles pour déterminer la visibilité directe dans une scène, c'est-à-dire l'élimination des parties qui ne contribuent pas à l'image rendue. Cependant même après cette phase de sélection, une scène réaliste quelconque regorge de détails qui, si représentés de manière traditionnelle c'est-à-dire à base de primitives géométriques, constituent une quantité d'information qu'il est inutile d'espérer pouvoir rendre avec le matériel graphique dans un futur proche.

Lorsque le champ de vision s'étend sur une grande distance, comme c'est généralement le cas pour les vues en extérieur, tout ce qui est visible à l'arrière-plan, joue un rôle important pour le réalisme perçu de la scène. Pris individuellement, les objets concernés (à une certaine distance du point de vue) n'occupent qu'une petite surface de l'écran, mais par leur potentielle multitude, les objets de l'arrière-plan considérés dans leur ensemble peuvent avoir un impact important sur l'image à rendre.

Pour faire face à cette profusion de détails, abondant dans les scènes réalistes, la stratégie traditionnellement utilisée consiste à recourir à une simplification des objets présents, en fonction de l'importance qu'ils occupent dans l'image. Ainsi les objets lointains, les objets de petite taille ou partiellement cachés sont rendus avec un niveau de détail dégradé. Un certain nombre de niveaux de détails est déterminé à l'avance et pour chaque niveau de détail une version de l'objet est

FIGURE 7.1 – Importance des objets lointains pour le réalisme d'une scène et nécessité de représentations adaptées.

précalculée, permettant ensuite lors du rendu de choisir parmi ces représentations celle qu'il est le plus approprié de rendre. Se pose alors la question du nombre de niveaux de détails à utiliser et de la transition entre ces niveaux. Rares sont les techniques qui permettent de faire varier continuement le niveau détail d'un objet, on est alors généralement confronté au problème de parvenir à effectuer une transition visuellement continue entre deux représentations distinctes de l'objet.

Le système perceptif humain est particulièrement sensible aux effets de sursauts (*popping*) qui peuvent apparaître lors d'une transition brusque entre deux versions visuellement distinctes d'un objet, alors que l'éventuelle dégradation visuelle impliquée par la simplification de l'objet peut s'avérer imperceptible sur une image fixe. C'est un problème gênant lorsque l'on utilise une représentation utilisant un maillage polygonal, de par son caractère discret. Un autre problème qui peut se poser est celui des très faibles niveaux de détails. Avec les représentations surfaciques on ne peut pas diminuer infiniment le niveau de détail d'un objet, sa topologie en est notamment une limitation (la géométrie d'une arbre par exemple, constituée d'un grand nombre de branches, est difficilement réductible à quelques triangles, même observée de loin). On peut cependant vouloir conserver des détails perceptibles à grande distance (la silhouette notamment peut conserver un aspect complexe quelle que soit l'échelle; la sensation de mouvement, la parallaxe, est aussi perceptible à une grande distance), tout en utilisant un nombre très faible de primitives (puisqu'il se multiplie avec le nombre d'objets à afficher, potentiellement très élevé).

Nous nous intéressons dans ce chapitre ainsi que dans le chapitre suivant aux *représentations non géométrique*, c'est-à-dire qui ne dépendent pas d'une notion de surface, permettant le rendu efficace de géométries complexes et d'un très grand nombre d'objets, là où les méthodes traditionnelles de simplification s'avèrent mal adaptées. Trois cas particulièrement nous intéressent, où il est crucial de disposer d'une représentation non surfacique :

– le rendu d'objets à l'arrière-plan, lointains et potentiellement très nombreux
– le rendu de petits objets, secondaires mais contribuant au réalisme d'une scène
– le rendu de géométries complexes, denses, comme le branchage ou le feuillage d'un arbre, mousse, prairies, fourrure, etc.

Les deux parties qui suivent servent à définir le cadre des représentations développées dans ce chapitre ainsi que dans le chapitre suivant, d'abord par la définition précise des hypothèses et des objectifs visés, puis par la définition du *light-field cylindrique*, qui est l'objet que l'on cherche à approcher. Les parties suivantes enfin se concentrent sur l'objet de ce chapitre, les représentations par échantillonnage de la parallaxe.

7.1 Hypothèses et objectif

Nous cherchons une représentation pour pouvoir rendre le plus fidèlement et le plus efficacement possible un objet O statique. Voici plus précisément les hypothèses que nous faisons sur les conditions dans lesquelles l'objet sera rendu, puis les objectifs que nous nous fixons quant à la fidélité du rendu que la représentation de l'objet devra permettre de reproduire.

FIGURE 7.2 – Hypothèses sur les conditions de vue : seules les vues orthographiques prises en trounant autour d'un axe de rotation fixé sont considérées

7.1.1 Hypothèses

La première hypothèse que nous effectuons est qu'au moment du rendu, O occupera une faible proportion de l'écran (objet lointain ou petit objet).

La deuxième hypothèse consiste à restreindre l'angle de vue sous lequel O pourra être visualisé. Nous définissons un axe privilégié, que nous appelerons *axe de rotation*, qui devra se trouver orthogonal à la direction de vue lors du rendu.

Ces deux hypothèse mises ensemble impliquent que nous ne considérons que les vues orthographiques, de direction orthogonale à l'axe de rotation. Bien que cela puisse à première vue sembler restrictif, cela permet en fait de répondre déja à beaucoup de besoins pratiques. Un grand nombre d'applications imposent au point de vue de se déplacer sur un plan (ou une surface localement plane), notamment les applications en vue à la première personne. Les objets distants (comme des arbres par exemple) sont alors toujours vus selon une direction quasiment orthogonale à leur axe vertical. En prenant cet axe comme axe de rotation on se retrouve alors bien dans le cadre de nos hypothèses. Nous montrons plus tard en quoi cette hypothèse est avantageuse.

La troisième hypothèse concerne le modèle d'éclairement. Nous nous placerons dans un premier temps pour simplifier dans le cas où les conditions d'éclairage (sources lumineuses) sont statiques. Nous discuterons par la suite en fonction de la représentation étudiée de la possibilité éventuelle de la rendre plus flexible sur ce point, c'est-à-dire de permettre de calculer l'éclairement dynamiquement.

Une quatrième hypothèse optionnelle consiste à restreindre encore les conditions de vue possible. Dans les applications pratiques il arrive souvent qu'on puisse à l'avance déterminer qu'un objet ne pourra être vu que pour un ensemble de points de vue restreint (par exemple dans une simulation de course automobile, l'angle sous lequel peuvent être vus les objets en bord de route est borné). La connaissance d'une telle restriction permet généralement d'obtenir une représentation plus efficace de l'objet (en mémoire et en rapidité de rendu).

7.1.2 Objectif

Nous recherchons une représentation de l'objet O qui soit suffisamment compacte et qui permette d'en effectuer un rendu efficacement, en tirant parti des ressources du matériel graphique, de manière la plus visuellement fidèle au modèle de référence qui soit.

Cette représentation est vouée à être utilisée comme dernier niveau de détail au rendu, elle doit donc être très efficace quand l'objet est situé loin du point de vue, autrement dit le coût de rendu doit être proportionnel à la surface occupée à l'écran (l'algorithme rendu doit être *output sensitive*), et d'autre part elle doit traiter correctement les problèmes de filtrage (*aliasing*). Enfin la transition entre cette représentation et une autre plus détaillée (polygonale par exemple) doit pouvoir être effectuée continument.

Parmi les aspects perceptuels que nous cherchons à reproduire, notons l'importance de la reproduction de silhouettes correctes, la restitution de l'impression de parallaxe (impression de rotation lorsque l'objet tourne par rapport au point de vue, perceptible de loin) et celle de l'éclairement perçu.

7.2 Light-field cylindrique

Comme expliqué nous nous intéressons à l'aspect visuel de l'objet à rendre, et nous cherchons une représentation qui permette de le restituer au mieux. Il nous en faut donc une mesure, indépendente de la représentation originale de l'objet.

Rendre un objet revient à en échantillonner la *fonction plénoptique* pour un certain ensemble de rayons lumineux. La définition la plus générale de la fonction plénoptique [AB91] est la fonction à sept dimensions qui à une position de l'espace, une direction, une longueur d'onde et un temps donné associe une intensité lumineuse.

Avec les hypothèse que nous nous sommes fixés, cette fonction peut se ramener à seulement trois dimensions, d'abord en supprimant les dimensions temporelles et de longueur d'onde, par l'hypothèse que l'objet est statique et en prenant simplement l'espace RGB comme espace d'arrivée, ensuite en ramenant les deux dimensions directionnelles à une seule de par l'hypothèse qu'on ne considère que les rayons lumineux horizontaux, enfin en supprimant une dimension spatiale en ne considérant pour un rayon lumineux que sa première intersection avec l'objet.

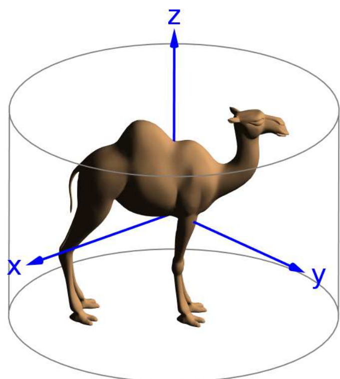


FIG. 7.3 *Repère R_O*

Plus précisément, on définit un repère orthonormé R_O dont l'axe z est dirigé selon l'axe de rotation de l'objet, et tel que l'objet soit contenu dans le cylindre droit d'axe z, de rayon 1 et délimité par les plans $(z = -1)$ et $(z = 1)$. Ce repère sera pris comme repère de référence pour tout ce qui suit. On définit ensuite le repère image R_θ comme le repère orthonormé de même origine que le repère de référence R_O, d'axes définis comme suit :

$$\begin{aligned} X_{R_\theta} &= -\sin(\theta)X_{R_O} + \cos(\theta)Y_{R_O} \\ Y_{R_\theta} &= Z_{R_O} \\ Z_{R_\theta} &= \cos(\theta)X_{R_O} + \sin(\theta)Y_{R_O} \end{aligned}$$

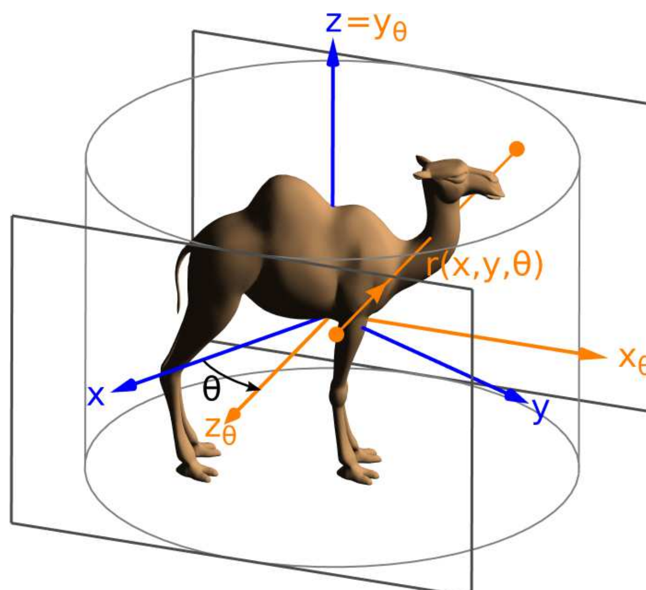


FIGURE 7.4 – Repère R_θ et paramétrisation des rayons lumineux $r(x,y,\theta)$.

On peut alors maintenant définir le *light-field cylindrique* de l'objet O comme la fonction :

$$f: \begin{array}{ccc} [-1,1]^2 \times [0,2\pi[& \longrightarrow & RGB \\ (x,y,\theta) & \longmapsto & \text{couleur}(r(x,y,\theta)) \end{array}$$

où $r(x,y,\theta)$ représente le rayon lumineux allant du point $(x,y,1)$ au point $(x,y,-1)$ dans le repère R_θ.

Ce que nous cherchons est alors une représentation de l'objet qui en permette un rendu efficace tout en approchant au mieux son light-field cylindrique.

7.3 Travaux existants

Pour représenter le light-field d'un objet, la solution la plus simple consiste à échantillonner celui-ci de manière dense [LH96, GGSC96]. L'interpolation entre échantillons d'un light-field étant mal définie [CTCS00, SYGM03], on ne peut généralement pas synthétiser de nouvelles vues de l'objet à une résolution supérieure à celle de l'échantillonnage du light-field. Les travaux traitant de cette représentation s'attachent alors principalement au problème de la compression efficace des données échantillonnées. Malgré les forts taux de compression atteignables, de par la forte redondance des données, la taille mémoire d'une telle représentation reste inadaptée à notre problème.

Les travaux de rendu à base d'images [CW93, MB95, DTM96, MMB97, SGHS98, SP99, OBM00] peuvent être vus comme des manières d'échantillonner le light-field d'un objet de manière peu dense. Pour compenser le manque d'information d'un tel sous-échantillonnage, la stratégie appliquée consiste généralement à ajouter des données géométriques supplémentaires, une information de profondeur par exemple. Cependant il existe peu de travaux cherchant à profiter au mieux des ressources offertes par le matériel graphique actuel pour en déduire des représentations efficaces.

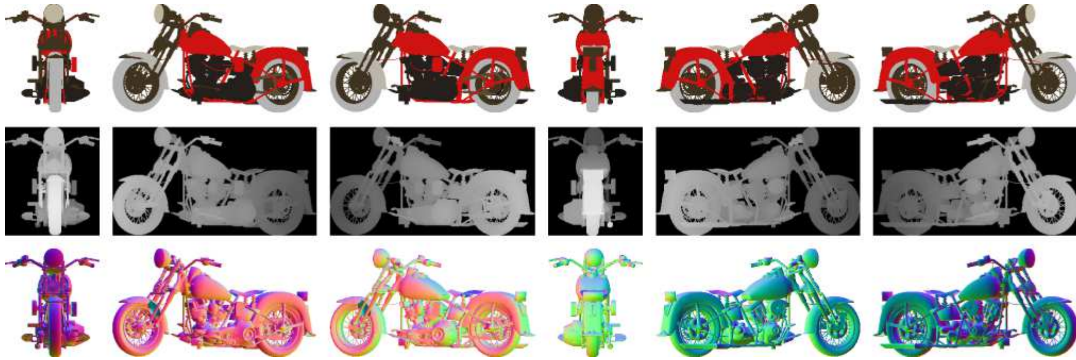


FIGURE 7.5 – Cycle de six imposteurs avec parallaxe (de en haut en bas : cartes de couleur, cartes de parallaxe et cartes de normales).

7.4 Cycle d'imposteurs avec parallaxe

L'option la plus intuitive pour capturer le light-field associé à un objet consiste à échantillonner un certain nombre K de vues $(I_k)_{k \in [\![0,K[\![}$, à une certaine résolution N, en tournant autour de cet objet selon des angles de vue (θ_k) espacés régulièrement ($\theta_k = {}^{2k\pi}/_K$). Pour qu'une telle représentation reste avantageuse en terme de coût mémoire, il faut que le nombre K de vues utilisées reste faible (intuitivement, l'information représentée par 8 vues de l'objet devrait largement suffire pour en capturer tous les aspects).

Le cas extrême consiste à prendre une seule vue : c'est la technique de l'imposteur unique, faisant toujours face à la caméra, couramment utilisée notamment pour les objets lointains, fixés à l'environnement (les arbres, par exemple). Cette technique fonctionne bien dans ce cas car lors d'un mouvement du point de vue, la parallaxe perceptible sur un objet fixe suffisamment distant est quasiment nulle. Cependant dès que l'objet n'est plus forcé de garder une orientation fixe (pour un véhicule au loin, par exemple), la parallaxe devient perceptible, même de loin. Comme c'est un effet que nous cherchons à reproduire, nous utiliserons plusieurs vues.

La question se pose alors de savoir comment synthétiser à partir de ces K vues, une nouvelle vue qui n'a pas été échantillonnée. La solution la plus simple consisterait à simplement sélectionner la vue I_k la plus proche du point de vue courant (telle que θ_k soit le plus proche de l'angle de vue θ courant) et de l'afficher telle quelle. Si l'on se restreint à un nombre de vues faible, ce qui correspond à notre objectif, une telle approche génère un effet de saccades quand l'objet tourne. Il faudrait un nombre de vues très élevé (de l'ordre de $2\pi N$) pour que cet effet ne soit plus perceptible. Pour réduire cet effet de saccades on peut être tenté de sélectionner les deux vues les plus proches (I_k et I_{k+1} telles que $\theta \in [\theta_k, \theta_{k+1}[$) et en effectuer un mélange linéaire (avec un poids dépendant de la position relative du point de vue courant entre ces deux vues). Ceci a pour effet de remplacer les saccades précédentes par un effet d'images fantômes, qui nécessiterait là encore d'utiliser un trop grand nombre de vues pour être rendu imperceptible.

L'effet de flou induit par l'interpolation est un problème bien connu des travaux sur les *light-fields* : il correspond à un sous-échantillonnage du *light-field* et au manque d'information qui en découle. L'information manquante, est la quantité de

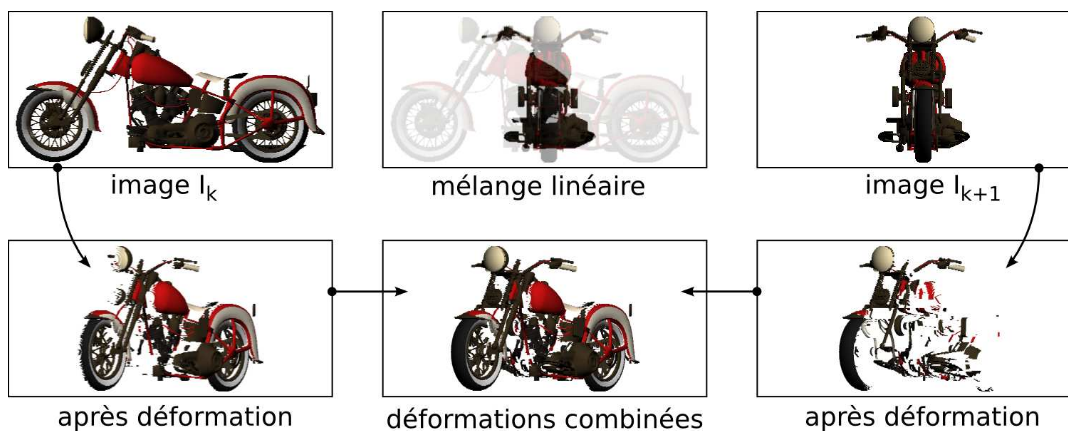


FIGURE 7.6 – Déformations des deux points de vue I_k et I_{k+1} les plus proche selon leurs cartes de parallaxe respectives, puis combinaison (en bas, au milieu) : l'image résultat n'a pas les problèmes de fantômes que produit un simple mélange linéaire (en haut, au milieu).

parallaxe associée à chaque échantillon : si l'on prend le point de l'objet se trouvant sous un pixel d'une vue I_k donnée, lorsque l'on fait tourner l'objet verticalement, ce point se déplace horizontalement d'une quantité dépendant de sa distance au point de vue. Comme nous allons le voir, garder cette information permet de résoudre le problème de synthèse de vue quelconque pour un faible surcoût de stockage en mémoire.

7.4.1 Définition

On définit le *cycle d'imposteurs avec parallaxe* de l'objet O comme la famille des K imposteurs avec parallaxe $(I_k)_{k\in[\![0,K[\![}$ de résolution $N \times N$ obtenus par rendu orthographique de l'objet O dans les direction $(\theta_k = \frac{2k\pi}{K})_{k\in[\![0,K[\![}$. L'imposteur avec parallaxe I_k est un triplet constitué d'une carte de couleur C_k, d'une carte de normales N_k et d'une carte de parallaxe P_k. La carte de couleur est constituée de trois canaux RGB contenant la composante de couleur diffuse ainsi qu'un canal d'opacité contenant des valeurs binaires 1 ou 0 selon que les pixels correspondants sont couverts par l'objet ou pas. Avec cette représentation il est possible de rendre les imposteurs avec un éclairement diffus (réflectance Lambertienne). L'utilisation de modèles d'éclairement paramétriques plus complexes ne poserait pas d'autre problème que le surcoût mémoire dû au stockage d'éventuelles cartes de coefficients supplémentaires (un coefficient spéculaire par exemple).

Le pixel (i,j) de l'imposteur I_k correspond à l'échantillonnage du rayon lumineux $r(x_i, y_j, \theta_k)$ allant, dans le repère R_{θ_k}, du point $(x_i, y_j, 1)$ vers le point $(x_i, y_j, -1)$, avec $x_i = {}^{2i}/_{N} - 1$ et $y_j = {}^{2j}/_{N} - 1$. La parallaxe $P_k(i,j)$ correspond à la coordonnée selon z (toujours dans le repère R_{θ_k}) du premier point de l'objet O rencontré le long de ce rayon. Autrement dit, le point échantillonné au pixel (i,j) a pour coordonnées $(x_i, y_j, P_k(i,j))$ dans le repère R_{θ_k}. Lorsque le rayon lumineux n'intersecte pas l'objet, on associe au pixel (i,j) une valeur de parallaxe arbitraire hors de l'intervalle $[-1,1]$.

La carte de parallaxe peut être vue comme la représentation de la surface visible selon la direction θ_k de l'objet O, sous la forme d'un champ de hauteur ayant pour

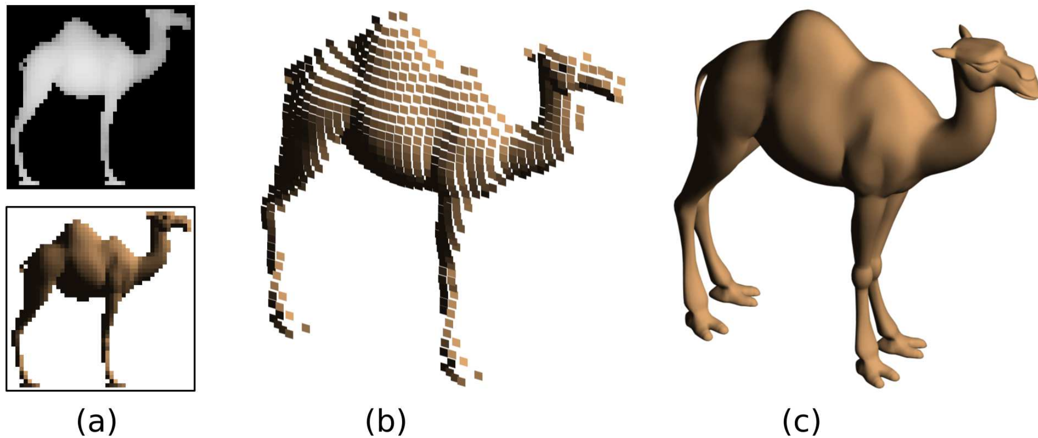


FIGURE 7.7 – Représentation de la parallaxe : (a) un imposteur avec parallaxe, (b) les échantillons correspondants représentés en 3D, (c) l'objet original rendu sous le même point de vue.

base le carré de sommets $(\pm 1, \pm 1, 0)$ dans le repère R_{θ_k}. A la différence près que contrairement à la définition du champ de hauteur que nous avons donné en section 4.1, la carte de parallaxe ne représente pas forcément une surface continue.

7.4.2 Algorithme de rendu sur GPU

De nombreux travaux existent s'attachant à rendre des imposteurs avec une information de parallaxe [MMB97, SGHS98, SP99, OBM00], mais il en existe peu de récents qui cherchent à profiter des capacités du matériel graphique actuel. Nous montrons ici comment notre algorithme de rendu de relief peut être adapté pour rendre des imposteurs avec parallaxe. Les principales différences avec le cadre du rendu de champ de hauteur comme présenté au chapitre 4 viennent d'abord du fait que les rayons lumineux sont toujours horizontaux ce qui enlève une dimension au problème, ensuite de la présence possible de discontinuités dans la carte de parallaxe qu'il faut prendre en compte, et enfin de l'utilisation de deux cartes de parallaxe à la fois pour chaque rendu (celles correspondant aux deux vues les plus proches du point de vue courant), au lieu d'une seule carte de hauteur.

Pour synthétiser une vue de l'objet selon un angle θ, on sélectionne les deux vues les plus proches I_k et I_{k+1} (par souci de simplicité des notations nous écrirons seulement $k+1$ au lieu de $k+1 \mod K$) du cycle d'imposteurs (telles que $\theta \in [\theta_k, \theta_{k+1}[$). L'image produite est le mélange de ces deux vues après déformation par utilisation de l'information de parallaxe et de la valeur de l'angle θ.

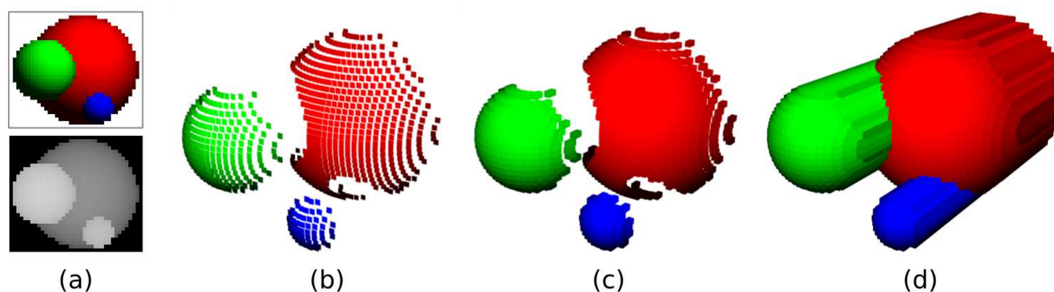


FIGURE 7.9 – Prise en compte des discontinuités : (a) imposteur avec parallaxe, (b) rendu des échantillons tels quels, sans seuil de continuité, (c) rendu avec un seuil $\delta p = 2\delta x$, (d) rendu comme un champ de hauteur continu.

Interpolation de la parallaxe

La carte de parallaxe correspond à un échantillonnage discret de la surface de l'objet O, il faut donc déterminer comment cette information doit être interpolée pour reconstruire la surface en d'autres points que ceux échantillonnés. On pourrait être tenté d'utiliser la même approche que pour les champs de hauteurs, c'est-à-dire par exemple une interpolation bilinéaire. Cependant à la différence de l'information échantillonnée dans une carte de hauteur, la parallaxe n'est pas une information continue : les valeurs de parallaxe sont discontinues aux silhouettes (internes ou externes) des objets. Supposer la parallaxe continue au niveau d'une silhouette amenerait à afficher un morceau de surface à la place d'un trou, ce qui produirait un artefact visuellement gênant.

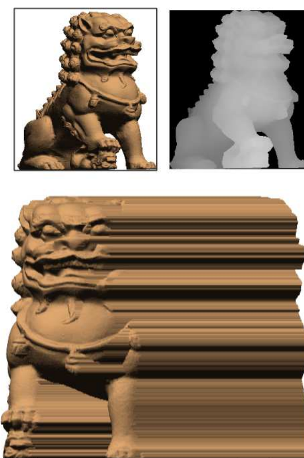


FIG. 7.8 *Artefacts dûs à l'absence de prise en compte des discontinuités*

Notons maintenant que seule l'interpolation horizontale de la carte de parallaxe est à considérer : comme nous avons supposé que les rayons de vue lors d'un rendu sont tous horizontaux, il n'y a pas de parallaxe verticale (autrement dit la déformation d'une image après application de la parallaxe ne déplace pas ses pixels verticalement). Nous utiliserons pour cet axe une interpolation de plus proche voisin, quelle que soit l'interpolation choisie pour l'axe horizontal.

Discontinuités Un moyen simple de définir les discontinutés dans la carte de parallaxe consiste à utiliser un seuil δp à partir duquel deux valeurs de parallaxe adjacentes sont considérées déconnectées (c'est-à-dire appartenant à des surfaces distinctes). Cette approche a pour inconvénient de considérer discontinue une surface vue à angle rasant pour le point de vue de l'imposteur (c'est-à-dire faisant un angle supérieur à $\arctan(\delta p/\delta x)$ par rapport à la direction de vue de l'imposteur, où δx est la largeur d'un pixel). Ce n'est pas une vraie limitation car une telle surface est de toutes façons mal échantillonnée selon un tel point de vue, cela n'a donc pas vraiment de sens d'en interpoler les échantillons continuement.

En fait le manque d'information occasionné sur une telle surface est généralement comblé par l'utilisation de l'information capturée par un imposteur voisin du

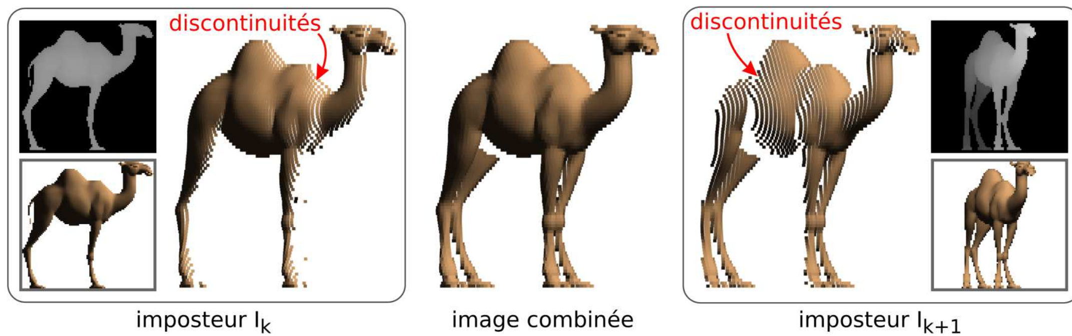


FIGURE 7.10 – L'utilisation d'une valeur δp faible (ici $\delta x/2$) fait apparaître des discontinuités dans les deux vues déformées individuelles, qui viennent se combler lorsque les deux images sont combinées.

cycle (voir figure 7.10). Le seul problème pouvant alors se poser est la présence d'une occultation dans ce point de vue voisin, occultant une partie de la surface en question, empêchant ainsi de disposer de l'information nécessaire pour la rendre.

Interpolation Les discontinuités étant définies, il reste à savoir comment interpoler les valeurs de parallaxe là où elle est continue. Comme pour les cartes de hauteurs plusieurs possibilités existent, on a en revanche ici plusieurs critères à prendre en compte dans ce choix. Le critère le plus important est que l'image produite, lorsque l'on se place dans la même vue que celle qui a servi à capturer l'imposteur, doit être la même que l'image C_k. C'est nécessaire si l'on veut assurer la continuité entre la représentation par cycle d'imposteurs et la représentation originale de l'objet. Le second critère est la facilité d'interprétation de la valeur interpolée : il faut garder à l'esprit que cette information va être utilisée par l'algorithme de rendu pour calculer la déformation de l'image. Nous verrons par la suite que la principale opération requise par notre algorithme de rendu est d'évaluer localement (entre deux texels) si une droite (rayon de vue) intersecte la surface définie par la carte de parallaxe. C'est donc l'efficacité de cette opération qu'il faut favoriser.

Les choix que laisse le matériel graphique sont l'interpolation de plus proche voisin et l'interpolation linéaire. Voyons comment l'information de discontinuité peut être intégrée dans ces deux cas.

Plus proche voisin La manière logique de traiter les discontinuités pour l'interpolation de plus proche voisin, consiste à considérer chaque texel (i, j) de la carte de parallaxe comme la représentation d'un bloc $B(i, j)$ correspondant à l'extrusion de la représentation surfacique du texel sur une profondeur δp, c'est-à-dire :

$$B(i, j) = [x_i \pm 1/2N] \times [y_i \pm 1/2N] \times [P_k(i, j) - \delta p, P_k(i, j)]$$

L'épaisseur ainsi ajoutée aux morceaux de surfaces de la carte de parallaxe permet de combler les discontinuités correspondant à une différence de parallaxe inférieure à δp, sans occulter les endroits où cette différence dépasse le seuil fixé.

Notons d'abord que cette approche permet éventuellement d'utiliser un seuil de discontinuité variable, défini pour chaque texel séparemment (par exemple une valeur supplémentaire ajoutée à la carte de parallaxe en précalcul).

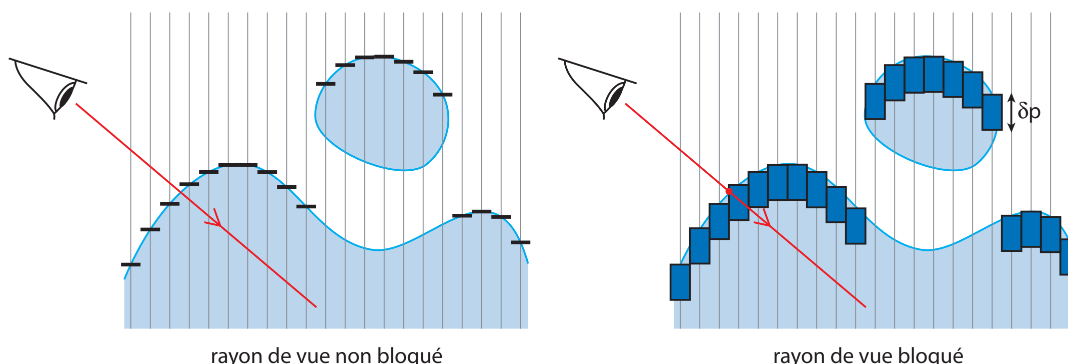


FIGURE 7.11 – Choix de la représentation des échantillons de la carte de parallaxe : sans épaisseur (à gauche) ou avec épaisseur (à droite).

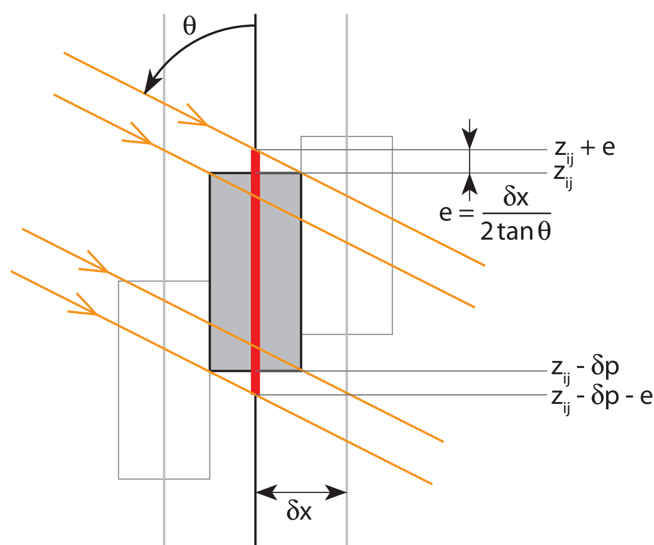


FIGURE 7.12 – Intersection avec un bloc $B(i,j)$ des rayons de vue inclinés d'un angle θ.

Cette représentation permet une implémentation simple du test d'intersection d'une droite (rayon de vue) avec un texel de la carte de parallaxe. Prenons un rayon de vue incliné d'un angle $d\theta$, et plaçons nous dans le plan horizontal contenant ce rayon. Appelons z la coordonnée selon l'axe Z_{R_θ} du point où le rayon atteint l'abscisse x_i du centre du texel (i,j). Alors ce rayon intersecte le bloc $B(i,j)$ si et seulement si

$$\left| z - P_k(i,j) + \frac{\delta p}{2} \right| < \frac{\delta p}{2} + \frac{\delta x}{2 \tan d\theta}$$

C'est un test simple à effectuer, sachant que le calcul de la valeur $\frac{\delta x}{2 \tan d\theta}$ peut être factorisé pour tous les pixels à dessiner avant chaque rendu, réduisant ainsi l'opération au calcul de trois sommes (deux si l'on pré-soustrait la valeur δp aux $P_k(i,j)$, et une seule si δp est uniforme), une valeur absolue et une comparaison.

Linéaire Nous avons vu dans le cas des champs de hauteur que le test d'intersection d'une droite avec une fonction continue affine par morceaux pouvait être effectué de manière simple. Cependant l'ajout de discontinuité complique cette opération.

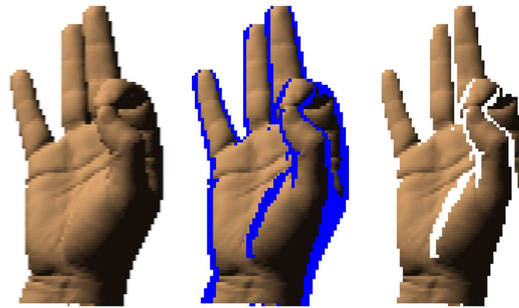


FIGURE 7.13 – La suppression des parties discontinues (en bleu dans l'image centrale) produit des trous au rendu (image de droite), même lorsque l'imposteur (image de gauche) est rendu sous son point de vue de référence.

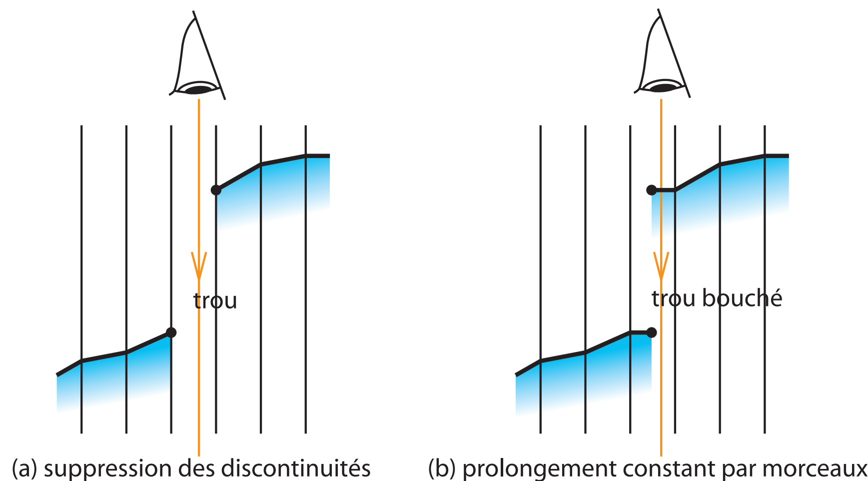


FIGURE 7.14 – Interpolation linéaire de la parallaxe et problèmes des discontinuités (à gauche), résolution par prolongation constante par morceaux des valeurs de parallaxe (à droite).

Pour gérer les discontinuités, on pourrait être tenté d'effectuer une interpolation linéaire par morceaux habituelle et de simplement supprimer les portions correspondant à l'interpolation de deux valeurs de parallaxe différrant de plus du seuil de discontinuité δp. Cependant une telle approche ne répondrait pas au premier critère que nous nous sommes fixé : cela induirait des trous au rendu, visibles même lorsque la vue utilisée est celle de l'imposteur (voir figure 7.13). Par exemple un texel isolé ayant une parallaxe différente de tous ses voisins sera complètement supprimé et remplacé par un trou.

Il faut donc pour palier à ce problème définir une surface, discontinue, entre deux valeurs disjointes, dont la projection dans la direction de vue de l'imposteur occupe tout le texel.

Une solution simple consiste à séparer par leur bordure deux texels disjoints et de prolonger de manière constante la valeur de parallaxe dans chacune de deux zones ainsi définies.

Cela permet de répondre au premier critère mais rend plus difficile la vérification du deuxième : les partie discontinues doivent être traitées différemment des parties continues, ce qui complique le test d'intersection avec une droite. De plus deux intervalles sont à considérer lors du test d'intersection d'une droite avec la

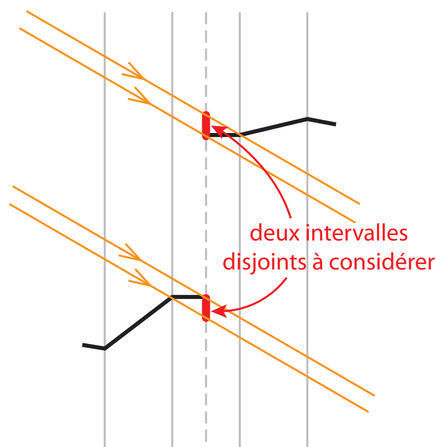


Figure 7.15 – Le test d'intersection d'un rayon de vue avec la carte de parallaxe P_k revient à considérer deux intervalles disjoints selon l'axe z_{θ_k}.

zone comprise entre deux texels disjoints, là où la méthode d'interpolation de plus proche voisin en donnait seulement un.

En considérant que la représentation par cycle d'imposteur est vouée à être utilisée pour synthétiser des vues à des résolutions inférieures ou égales à la résolution des imposteurs, disposer pour la parallaxe d'une précision de reconstruction supérieure à la taille d'une texel est superflue. Par le surcoût au rendu qu'elle implique, l'interpolation linéaire présente donc un faible intérêt par rapport à l'interpolation de plus proche voisin.

Déformation

Contrairement aux approches classiques par *warping* [MMB97, SP99, OBM00], qui déplacent chaque texel de l'imposteur d'une quantité dépendant de la parallaxe associée, pour en produire une version déformée, nous appliquons une méthode de *raycasting*, cherchant pour chaque rayon de vue le texel de l'imposteur qui s'y projette après déplacement par parallaxe, tirant ainsi parti au mieux du parallélisme qu'offre le GPU (autrement dit on adopte une approche de *backward rendering*, contrairement aux méthodes de *warping* qui s'apparentent à du *forward rendering*, ne permettant pas l'exploitation efficace de la puissance de calcul paralléle offerte par le matériel actuel).

Plaçons nous dans le repère R_{θ_k} et appelons $d\theta$ l'angle sous lequel doit être rendue la vue I_k (c'est-à-dire $d\theta = \theta - \theta_k$). D'après notre hypothèse sur les conditions de rendu, les rayons de vue à calculer sont horizontaux. Considérons un rayon de vue à rendre et le plan horizontal qui le contient.

Ce plan intersecte une ligne unique de l'image I_k, d'indice i. Si l'on se place dans ce plan, on cherche à calculer la première intersection du rayon de vue avec les blocs définis par les texels de la ligne i de l'image. Pour déterminer cette intersection, nous proposons une méthode similaire à notre algorithme de rendu de relief sans précalcul, consistant à parcourir dans l'ordre chaque texel rencontré le long du rayon pour s'arrêter à la première intersection avec un bloc. Les pas sont effectués aux centres des texels, ce qui permet de ne rater aucune intersection et

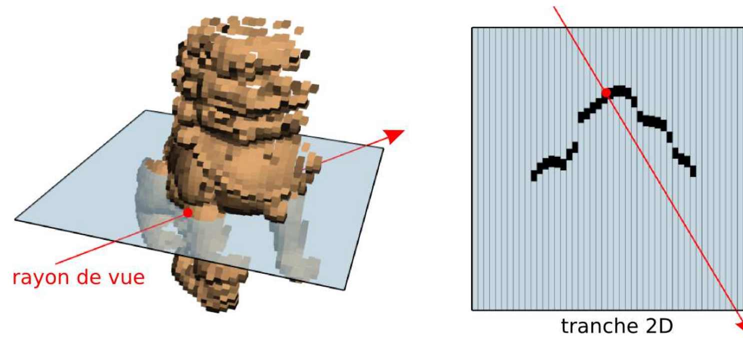


FIGURE 7.16 – Rayon de vue et tranche horizontale associée : les échantillons intersectés correspondent à une ligne horizontale de la carte de parallaxe.

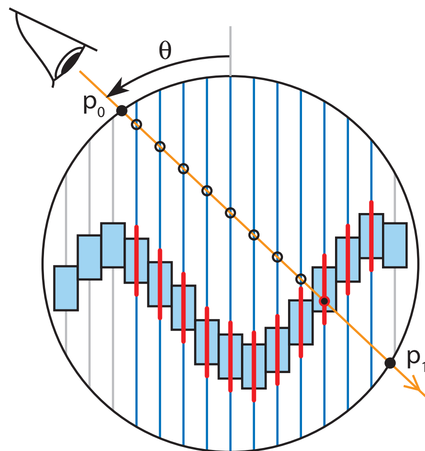


FIGURE 7.17 – Algorithme de parcours pour l'intersection d'un rayon de vue faisant un angle θ avec la carte de parallaxe considérée.

d'utiliser le test d'intersection décrit précédemment. L'algorithme est le suivant :

Algorithme 7.1 : Intersection d'un rayon lumineux avec la carte de parallaxe

`// les calculs se font dans le repère` R_{θ_k}
$p_0 \leftarrow (x_0, z_0) =$ point d'entrée du rayon lumineux dans le cylindre
$p_1 \leftarrow (x_1, z_1) =$ point de sortie du rayon lumineux hors du cylindre
$j \leftarrow$ indice de la tranche horizontale dans laquelle se trouve le rayon
$d \leftarrow p_1 - p_0$
$di \leftarrow d_x / |d_x|$
$dz \leftarrow 2 \times d_z \div (|d_x| * N)$
$i0 \leftarrow N \times (p_{0_x} + 1) \div 2$
$i \leftarrow \lfloor i0 + di \times 0.5 \rfloor$
$z \leftarrow p_{0_z} + (i + 0.5 - i0) * dz$
$e \leftarrow (\delta p + \delta x \div \tan d\theta) \div 2$
tant que $z > p_{1_z}$ *et* $|P(i,j) - \delta p \div 2| < e$ **faire**
  $z \leftarrow z + dz$
  $i \leftarrow i + di$
si $|P(i,j) - \delta p \div 2| < e$ **alors**
  afficher couleur $C(i,j)$
sinon
  ne rien afficher

De par la définition des points d'entrée et de sortie p_0 et p_1, le nombre de pas maximum effectué est $N \sin \theta$. On pourrait être tenté d'utiliser notre algorithme de rendu de relief avec précalcul pour réduire le nombre d'itérations nécessaires pour atteindre l'intersection, cependant la définition du rayon de sûreté telle que nous l'avons donnée n'a de sens que dans le cas d'un champ de hauteur continu.

Une bonne manière d'accélérer le calcul d'intersection, consiste à réduire l'intervalle de recherche $[p_0, p_1]$. Sachant qu'un imposteur n'est rendu que sous des angles de vue appartenant à un intervalle réduit ($[-\delta\theta, +\delta\theta]$ avec $\delta\theta = \frac{2\pi}{K}$), on peut par exemple pour chaque point d'entrée dans le cylindre englobant, précalculer une distance minimale sur l'ensemble de ces direction, nous indiquant de combien le point de départ p_0 peut être avancé tout en s'assurant de ne pas rater d'intersection (une texture 2D supplémentaire est nécessaire pour stocker cette information). Il y aurait beaucoup de possibilités à explorer. Nous en avons étudié quelques unes [Bab05] et les résultats obtenus montrent qu'une telle approche permet de diminuer considérablement le temps de rendu.

Combinaison

La couleur calculée pour un pixel de la vue synthétisée est obtenue par combinaison des échantillons p_k et p_{k+1} obtenus par intersection du rayon de vue avec les deux imposteurs I_k et I_{k+1} encadrant l'angle de vue courant. L'échantillon p_k est un couple formé d'une couleur c_k et d'une profondeur z_k (calculable facilement à la fin de l'algorithme d'intersection). Il se peut qu'aucune intersection n'ait lieu auquel cas une valeur particulière est affectée à z_k, que nous noterons ∞ (et dans le cas contraire, nous noterons $z_k < \infty$).

Plusieurs cas peuvent alors se produire :

1. $z_k = \infty$ et $z_{k+1} = \infty$: dans chacune des vues, aucune intersection n'a été trouvée, par défaut on ne dessine rien (il est possible qu'une partie de l'objet se projette sur le rayon de vue mais cette partie est cachée dans les deux vues utilisées).
2. $z_k < \infty$ et $z_{k+1} = \infty$: le point visible dans I_k est caché dans I_{k+1}, on affiche donc la seule information disponible, c_k.
3. $z_k = \infty$ et $z_{k+1} < \infty$: même cas en inversant les deux vues.
4. $z_k < z_{k+1} < \infty$: on affiche la couleur c_k, puisque elle correspond à l'échantillon le plus proche du point de vue.
5. $z_{k+1} \leqslant z_k < \infty$: pour la même raison que précédemment, on affiche c_{k+1}

Algorithme complet

Nous appliquons une technique de raycasting sur GPU selon l'approche expliquée au chapitre 3. Le volume englobant de l'objet est ici un cylindre dont l'axe vertical est toujours orthogonal à la direction de vue, la projection en est donc un rectangle. On peut donc se contenter de rendre un simple *quad* faisant toujours face à l'écran. Les informations de position et d'orientation de l'objet sont passées

N \ K	4	8	16	32	64
32	135	192	260	312	345
64	79	129	194	260	312
128	43	75	127	192	260
256	22	41	74	123	189

TABLE 7.1 – Fréquences d'affichage (en Hz) obtenus pour le rendu d'un cycle d'imposteur avec parallaxe à une résolution de 800×600 pixels, pour différentes résolutions N^2 des imposteurs et pour différents nombres de vues K (sur une carte graphique GeForce 8600 et un processeur Intel Core 2 Quad Q6700 à 2.66 GHz).

au *vertex shader* (par exemple comme attributs associés aux sommets) pour d'une part placer le *quad* correctement à l'écran (en fonction de la distance notamment) et d'autre part calculer la direction sous laquelle est vu l'objet et passer cette information au *fragment shader*.

Le *fragment shader* connaissant la direction de vue peut sélectionner les imposteurs les plus proches ainsi que les angles de vue relatifs, et appliquer enfin l'algorithme que nous venons de décrire.

Résultats

L'application de cette représentation sur une grande variété d'objets montre que l'utilisation de 8 vues ($K = 8$) est généralement largement suffisante pour capturer l'intégralité de la partie visible d'un objet. De plus les éventuels trous occasionnés par le manque d'échantillons sur certaines parties de l'objet occultées dans tous les imposteurs sont généralement réduits et peu perceptibles.

Les temps de rendus de l'algorithme complet (raycasting des deux imposteurs les plus proches du point de vue et combinaison) sont donnés dans le tableau 7.1. Deux points sont à remarquer : d'abord le temps de rendu est proportionnel à la résolution horizontale N de la carte de parallaxe, ce qui est logique puisque la principale partie du coût de l'algorithme est la parcours de la carte de parallaxe, dépendant du nombre de pas effectués. Ensuite le temps de rendu décroit lors que l'on augmente le nombre K de vues utilisées. Cela s'explique par le fait que plus K est élevé, plus l'angle maximal $\delta\theta$ sous lequel doivent être rendus les imposteurs est faible, or plus cet angle est faible, plus le nombre de pas à effectuer lors du raycasting est faible lui aussi : le nombre maximal de pas à effectuer étant $N \sin \delta\theta$, le coût de rendu est proportionnel à $N \sin {}^{2\pi}/_{K}$. Cela fournit un moyen simple d'établir un compromis entre vitesse d'affichage et taille en mémoire de la représentation.

7.4.3 Bilan

Les cycles d'images avec parallaxe sont une solution avantageuse d'échantillonnage peu dense du *light-field* cylindrique d'une grande classe d'objets. D'une part grace à l'ajout de l'information géométrique de parallaxe, il est possible de représenter fidèlement un grand nombre d'objets avec peu de vues. D'autre part cette représentation s'adapte bien au matériel graphique actuel, pouvant être rendue efficacement grâce à une approche par raycasting dans un *fragment shader*.

Cependant la question du choix optimal des vues reste ouverte. Nous avons adopté une stratégie de distribution uniforme des angles de vue, qui a l'avantage de simplifier l'étape de rendu : pour une direction de vue donnée, retrouver les deux vues les plus proches est ainsi une opération triviale. Toutefois en levant cette contrainte, c'est à dire en permettant un placement arbitraire des vues, il serait possible *a priori* à nombres de vues égaux, d'obtenir une représentation plus fidèle d'un objet donné.

De manière plus générale, il serait intéressant d'étendre la représentation et l'algorithme de rendu associé à la visualisation d'un objet sous n'importe quel angle de vue, pas forcément horizontal. Une solution est donnée par Andújar et al. [ABB+07] : une mesure d'entropie est proposée pour le placement des vues et un algorithme de raycasting similaire à celui que nous venons de présenter est utilisé pour rendre une telle représentation. Le placement des vues reste néanmoins un problème ouvert : la solution proposée est une heuristique et ne garantit aucune optimalité.

Enfin, bien que peu de vues soient utilisées pour la représentation d'un objet par un cycle d'images avec parallaxe, il existe souvent une forte redondance entre elles, qui n'est pas exploitée. La plupart de la surface d'un objet se voit représentée dans plusieurs vues à la fois. C'est ce qui a motivé l'étude de la deuxième représentation que nous présentons maintenant.

7.5 Reliefs cylindriques

La représentation précédente cherchait à échantillonner le *light-field* cylindrique de l'objet en utilisant un nombre faible d'angles de vue. Nous présentons ici une autre stratégie d'échantillonnage : les directions de vue sont échantillonnées de manière dense, mais pour chacune on ne garde qu'un échantillon par tranche verticale. L'idée est ici d'éviter la redondance présente entre les différentes vues d'un cycle d'imposteurs.

7.5.1 Définition

On définit le relief cylindrique de l'objet O comme le triplet des cartes cylindriques de parallaxe P, de couleur C et de normales N, de même résolution $K \times M$, dont les valeurs pour le texel (i, j) correspondent respectivement à la parallaxe, à la couleur (diffuse) et à la normale échantillonnées par intersection de l'objet O avec le rayon r_{ij}, d'origine p_{ij} et de direction d_{ij} définis comme suit :

$$\begin{aligned} p_{ij} &= (\cos\theta_i, \sin\theta_i, z_j) \\ d_{ij} &= (-\cos\theta_i, -\sin\theta_i, 0) \end{aligned}$$

avec $\theta_i = {}^{2i\pi}/_K$ et $z_j = 2^{j+1/2}/_M - 1$.

La parallaxe $P(i, j)$, à valeurs dans $[-1, 1]$, est définie comme la distance signée du premier point d'intersection p_{ij} du rayon r_{ij} avec l'objet O par rapport à l'axe de

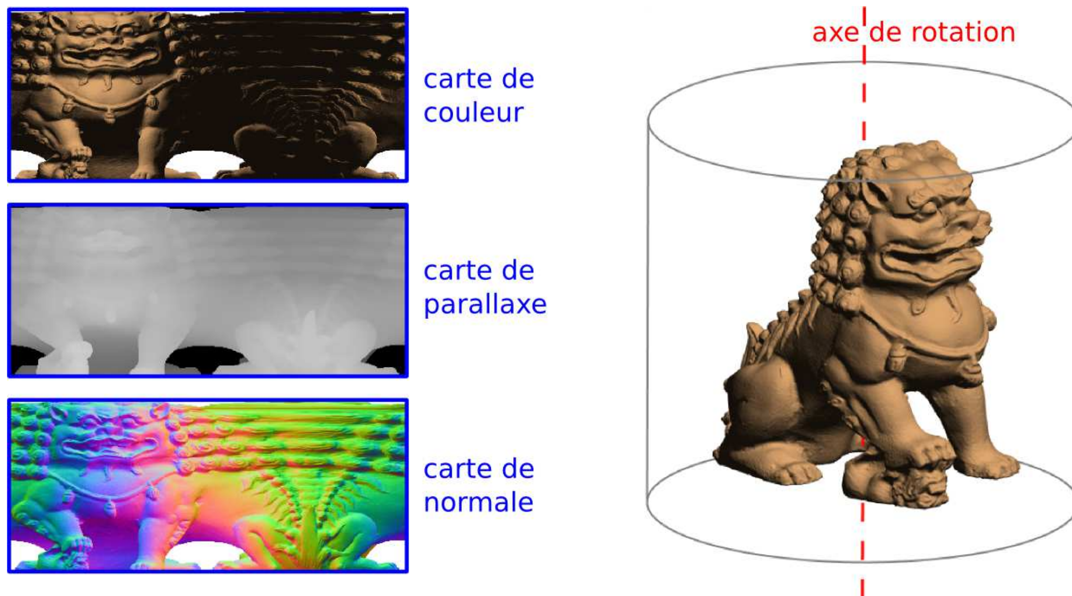


FIGURE 7.18 – Les trois cartes représentant un relief cylindrique (la carte de couleur a été éclairée à l'aide de la carte de normales pour une meilleure visualisation).

rotation de l'objet :

$$p_{ij} = \begin{bmatrix} 0 \\ 0 \\ z_j \end{bmatrix} + P(i,j)d_{ij} = \begin{bmatrix} P(i,j)\cos\theta_i \\ P(i,j)\sin\theta_i \\ z_j \end{bmatrix}$$

Cette représentation s'apparente d'une certain manière aux *Geometry Images* [GGH02], du fait qu'elle consiste en une paramétrisation continue à deux dimensions de la surface de l'objet, représenté sous la forme d'une image (voir figure 7.18). Cependant dans le cas des reliefs cylindriques la paramétrisation est la même quel que soit l'objet, précisément une paramétrisation cylindrique. C'est un compromis qui en rendant la représentation moins expressive en permet en contrepartie un rendu plus efficace, grâce à la structuration des échantillons de surface qu'induit une telle paramétrisation.

7.5.2 Expressivité

La classe d'objets exprimables de manière exacte à l'aide de cette représentation est relativement restreinte. Nous avons montré [Bab05] que si l'on se restreint aux valeurs de parallaxe positives, les surfaces continues représentables sont étoilées par tranches horizontales (chaque tranche horizontale est étoilée, par rapport à son intersection avec l'axe de rotation). C'est le cas par exemple des surfaces de révolution. L'autorisation de valeurs négatives augmente l'expressivité mais complique l'interprétation de la carte de parallaxe par la prise en compte de discontinuités.

Malgré tout cette représentation a pour intérêt de capturer, de manière compacte, une partie importante de l'aspect d'une grande variété d'objets. Même si l'intégralité de la surface d'un objet n'est pas capturée, son aspect et sa parallaxe sont généralement bien reproduits. Notamment les voitures ou les bâtiments sont de bons candidats pour cette représentation.

FIGURE 7.19 – Objets originaux (en haut) et rendu de leur représentation par relief cylindrique (en bas).

7.5.3 Algorithme de rendu sur GPU

Le rendu d'un objet représenté par un relief cylindrique peut se faire de manière similaire à celui d'un imposteur avec parallaxe, par raycasting de la carte de parallaxe. Pour réduire le nombre d'itérations à effectuer, nous avons proposé une technique de rendu approximatif [Bab05], permettant à l'aide d'un précalcul de disposer d'une approximation linéaire par morceaux pour chaque point de vue du décalage horizontal à effectuer dans la carte de parallaxe pour retrouver l'échantillon se projettant sur un pixel à rendre. Malgré les résultats approximatifs, cette méthode peut être intéressante pour rendre un objet suffisamment lointain tout en conservant l'impression de parallaxe.

Bien que nous n'ayons pas eu le temps de nous y consacrer, il serait intéressant d'adapter notre méthode de rendu de relief avec précalcul, notamment dans le cas où la surface est considérée continue. Cela permettrait d'obtenir un rendu exact en effectuant potentiellement très peu de pas le long des rayons de vue.

7.5.4 Bilan

Comme dans le cas des cycles d'imposteurs avec parallaxe, nous avons présenté une stratégie d'échantillonnage peu dense du *light-field* qui en s'appuyant sur l'ajout d'une information géométrique (parallaxe), permet de resynthétiser des vues quelconques d'un objet.

La représentation obtenue se restraint théoriquement à une petite classe d'objets (objets étoilés par rapport à un axe fixé), mais elle permet néanmoins de relativement bien approcher un grand nombre d'objets n'en faisant pas partie.

Notons enfin que cette représentation peut être considérée comme celle d'un relief appliqué à un cylindre, il serait alors intéressant d'étudier la possibilité d'é-

tendre la stratégie de précalcul proposée au chapitre 4 pour accélerer encore le rendu d'une telle représentation.

7.6 Conclusion

Nous venons de présenter deux solutions pour représenter un objet en échantillonnant son *light-field* cylindrique. Dans les deux cas, un échantillonnage dense est évité en recourant à une information supplémentaire de parallaxe.

Le fait de s'appuyer sur un échantillonnage du *light-field* de l'objet permet d'obtenir des représentations simplifiées fidèles à son apparence. Ceci permet d'effectuer une transition non perceptible lors du rendu entre la représentation originale de l'objet et sa représentation simplifiée, puisque les deux apparences concordent.

Ces deux représentations apportent une bonne solution au problème du rendu d'objets lointains. Dans les deux cas cependant l'algorithme de rendu demande un parcours itératif dans une ou plusieurs cartes de parallaxe, qui bien que nécéssitant généralement peu d'itérations, peut s'avérer trop coûteux lorsque l'on cherche à rendre un très grand nombre d'objets. La partie qui suit apporte une solution à ce problème en adoptant une approche inverse : une représentation très efficace à rendre est d'abord fixée, puis le *light-field* de l'objet est utilisé pour guider l'optimisation des paramètres de la représentation.

Chapitre 8

Light-field paramétrique

Les deux approches précédentes constituent deux manières différentes d'échantillonner le *light-field cylindrique* d'un objet, qui pour éviter de recourir à un échantillonnage dense s'appuient sur l'ajout d'une information géométrique (la parallaxe). L'information échantillonnée est utilisée telle quelle pour le rendu, et constitue donc la nouvelle représentation de l'objet.

La nécessité d'ajouter une information géométrique pose cependant plusieurs problèmes. D'abord cela suppose de disposer, pour l'objet que l'on veut représenter, d'une notion de surface, ce qui est restrictif. La plupart des objets rencontrés dans le monde réel ne peuvent pas être représentés simplement à l'aide d'une surface lisse à une certaine échelle. D'autre part cela ne permet pas de capturer des effets d'éclairements complexe, liés à une reflectance autre que celles données par les modèles paramétriques classiques (le modèle de Phong par exemple). Enfin on ne dispose pas toujours de cette information géométrique, dans le cas par exemple où l'on veut représenter un objet réel dont on ne possède que des données photographiques.

Si l'on considère le problème du rendu efficace, les représentations surfaciques ont pour inconvénient de nécessiter d'effectuer le parcours d'un ensemble de primitives pour déterminer un point d'intersection (que l'approche de rendu soit dans le sens *forward* ou *backward*). On aimerait idéalement disposer d'un algorithme de rendu dont le coût par pixel soit constant.

Cela suggère une approche inverse du problème : on aimerait, pour représenter un objet donné, pouvoir se fixer un ensemble de primitives bien défini que l'on sait pouvoir rendre efficacement, notamment en satisfaisant la propriété de coût constant par pixel rendu. La question que l'on se pose alors est : quelle est la meilleure approximation de l'objet original que l'on peut espérer obtenir avec une telle représentation, et comment la déterminer ?

Prenons comme exemple la technique courante consistant à représenter un objet par un ensemble de polygones texturés, éventuellement partiellement transparents. Cette technique est très utilisée en pratique, notamment pour représenter les objets à géométrie dense comme la végétation ou encore pour simplifier fortement un

objet. Les paramètres d'une telle représentation sont le nombre de plans, leurs positions, la résolution des textures et les valeurs qu'elles contiennent. Le placement et le dessin de telles textures demande généralement un travail d'artiste fastidieux.

On aimerait pour profiter au mieux de cette représentation, pouvoir y "projeter" de manière optimale et automatique l'aspect de l'objet initial, c'est-à-dire déterminer les valeurs des paramètres qui approchent au mieux son light-field.

Il n'existe aucun algorithme permettant de déterminer cet ensemble optimal de paramètres. La technique des *billboard clouds* [DSDD02, DDSD03] permet à partir d'une représentation polygonale de l'objet de déterminer la position optimale des plans (avec quand même la limitation que ce n'est pas un optimum global mais seulement local), mais les valeurs échantillonnées dans les textures associées sont déterminée par une simple heuristique ne garantissant pas l'optimalité en termes d'aspect final. Seul le caractère géométrique de l'objet est considéré pour l'optimisation.

Nous proposons dans cette partie une méthode qui se sert de l'information du light-field d'un objet pour en déduire une représentation optimale. Nous en montrons l'application dans le cas de la représentation par plans parallèles texturés et discutons ensuite des généralisations possibles.

8.1 Définitions

8.1.1 Light-field paramétrique

Soit une représentation R de l'objet O auquel on s'intéresse, définie par un ensemble de paramètres $(t_1, ..., t_p) \in \mathbb{R}^p$. Contrairement aux approches existantes qui définissent ces paramètres à partir de considérations géométriques, nous cherchons à déterminer les valeurs qu'il faut choisir pour cet ensemble de paramètres pour représenter l'*apparence* de l'objet de manière optimale. Par définition, le light-field f de l'objet O donne une description complète de son apparence, c'est donc lui que nous allons utiliser pour guider l'optimisation des paramètres $(t_1, ..., t_p)$.

Il faut d'abord définir l'ensemble des rayons lumineux que l'on doit considérer. Pour ceci nous faisons l'hypothèse que l'on connait une surface fermée $\mathcal{B}$ dont le volume englobé est convexe et contient l'objet O uniquement. Bien qu'il ne soit pas toujours possible de définir une telle surface, c'est généralement le cas (nous discutons plus tard des possibilités pour lever cette restriction). Cette simplification permet d'enlever une dimension à l'ensemble des rayons à prendre en compte : il est suffisant de considérer les rayons ayant pour origine un point de la surface $\mathcal{B}$ et l'intersectant une deuxième fois en un autre point : nous appelerons $\mathcal{R}$ cet ensemble, il peut être paramétré par un sous-ensemble de $\mathbb{R}^4$.

Faisons maintenant deux hypothèses sur la représentation R. La première hypothèse consiste à supposer que le rendu par cette représentation d'un rayon de vue r produit un résultat indépendant de la vue globale appliquée (ce n'est pas le cas de certaines représentations dites *view-dependent*) et le coût correspondant est indépendant des valeurs des paramètres (t_i).

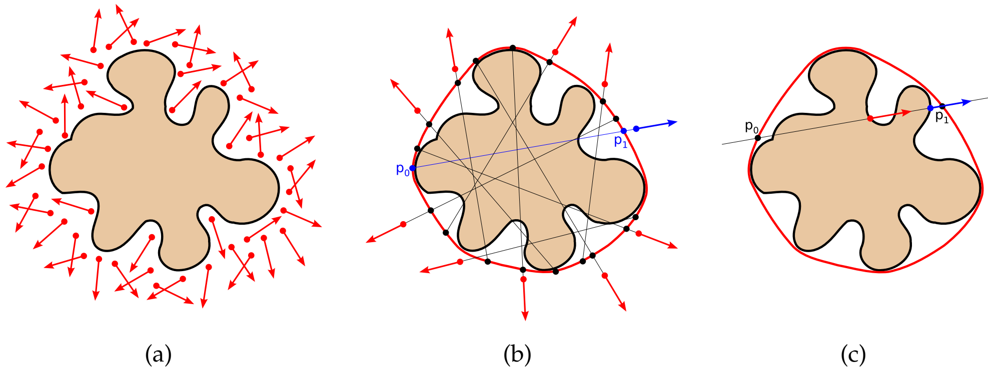


Figure 8.1 – Paramétrisation de l'ensemble des rayons lumineux : (a) paramétrisation 5D (position, direction); (b) paramétrisation 4D : points d'intersection p_0 et p_1 du support du rayon lumineux avec une enveloppe convexe; (c) limite de la paramétrisation 4D : deux rayons lumineux issus de points différents de la surface de l'objet peuvent posséder les même paramètres.

La deuxième hypothèse est que lors du rendu de l'objet O par la représentation R, à l'intérieur d'une scène S, la couleur $c(r)$ affichée en un pixel associé à un rayon de vue r est la somme d'une couleur $c_R(r)$ avec la valeur modulée par une transparence $\tau_R(r)$ de la couleur de fond c_f, c'est-à-dire la couleur qui serait obtenue par intersection du rayon r avec le reste de la scène S se trouvant après l'objet O. Autrement dit :

$$c(r) = c_R(r) + \tau_R(r)c_f$$

Cette hypothèse permet d'une part de représenter de la même manière les rayons qui intersectent l'objet et ceux qui ne l'intersectent pas (ceux-ci sont simplement exprimés à l'aide d'une couleur $c_R(r) = 0$ et d'une transparence $\tau_R(r) = 1$), et d'autre part de prendre en compte des objets-semi transparents ou des objets à géométrie dense (auquel cas la valeur d'opacité $\alpha = 1 - \tau$ représente une probabilité d'occultation). Notons que c'est l'équation de mélange traditionnellement utilisée pour le rendu d'objets avec transparence (technique de l'*alpha-blending*).

Ainsi le rendu par la représentation R peut être défini comme la correspondance entre chaque rayon lumineux $r \in \mathcal{R}$ et un couple $(c_R(r), \tau_R(r))$ où $c_R(r)$ est une couleur représenté par un triplet RGB (nous noterons $\mathcal{C} = [0,1]^3$ l'ensemble correspondant) et où $\tau_R(r)$ est une valeur de transparence dans $[0,1]$. Par cohérence avec l'utilisation classique en rendu d'un canal d'opacité $\alpha = 1 - \tau$ plutôt qu'une transparence, nous considérerons les valeurs $(c_R(r), \alpha_R(r))$ et nous noterons $\mathcal{C}_\mathcal{A} = \mathcal{C} \times [0,1]$ l'ensemble correspondant.

On peut maintenant définir le *light-field paramétrique* associé à la représentation R comme la fonction paramétrique $f^R_{t_1,\ldots,t_p}$ qui à un rayon lumineux fait correspondre la couleur obtenue :

$$\begin{aligned} f^R_{t_1,\ldots,t_p} : \ \mathcal{R} &\longrightarrow \mathcal{C}_\mathcal{A} \\ r &\longmapsto (c_R(r), \alpha_R(r)) \end{aligned}$$

La donnée de cette fonction définit entièrement la représentation R, du point de vue visuel.

8.1.2 Distance entre light-fields

Pour pouvoir comparer l'apparence de l'objet O original avec celle que lui donne une représentation paramétrique R, il faut pouvoir comparer leurs light-fields respectifs, ce qui nécessite de définir une distance sur l'ensemble des light-fields. Une telle distance doit idéalement prendre en compte les caractéristiques de la perception humaine pour évaluer à quel point deux light-field seront perçus comme proches par un observateur. Nous proposons ici une distance simple qui ne prend en compte que les aspects purement photométriques des light-fields, mais qui a l'intérêt de présenter de bonnes propriétés ensuite pour l'expression du problème d'optimisation qui nous intéresse.

Étant donné un light-field f et une famille $\mathcal{E} = (r_i)_{i \in [\![1,q]\!]} \in \mathcal{R}^q$ de rayons lumineux, nous applelons $f^{\mathcal{E}}$ l'échantillonnage de f par cet ensemble de rayons :

$$f^{\mathcal{E}} = (f(r_i))_{i \in [\![1,q]\!]}$$

Nous définissons alors sur l'espace $\mathcal{C}_{\mathcal{A}}{}^q$ des light-fields échantillonnés selon $\mathcal{E}$, à partir du produit scalaire euclidien sur $\mathcal{C}_{\mathcal{A}}$ (le produit scalaire euclidien de $\mathbb{R}^4$), le produit scalaire :

$$\left\langle f^{\mathcal{E}} | g^{\mathcal{E}} \right\rangle = \sum_{i=1}^{q} \left\langle f_i^{\mathcal{E}} | g_i^{\mathcal{E}} \right\rangle = \sum_{i=1}^{q} \langle f(r_i) | g(r_i) \rangle$$

et la norme euclidienne :

$$\|f^{\mathcal{E}}\|^2 = \left\langle f^{\mathcal{E}} | f^{\mathcal{E}} \right\rangle$$

ainsi que la distance associée :

$$d(f^{\mathcal{E}}, g^{\mathcal{E}}) = \|f^{\mathcal{E}} - g^{\mathcal{E}}\|$$

On peut ainsi comparer deux light-fields pour un échantillonnage donné. Cette définition de distance ne permet donc pas de comparer deux light-fields de manière continue, c'est-à-dire indépendamment d'un échantillonnage particulier, mais il faut noter qu'en pratique, la seule information dont on dispose sur le light-field que l'on cherche à représenter, consiste généralement en un échantillonnage discret. Bien que nous n'ayons pas eu le temps d'explorer cette voie, il serait cependant intéressant de pouvoir définir une distance sur l'espace continu des light-fields dans le cas où le light-field original de l'objet est donné sous une forme continue (le light-field d'un maillage polygonal par exemple peut être exprimé de manière analytique).

L'échantillonnage $\mathcal{E}$ doit être choisi de manière adaptée au cas qui nous intéresse. Nous supposons ici disposer de la liberté d'échantillonner de manière quelconque l'objet O considéré. C'est notamment le cas si l'objet est une représentation numérique (par exemple un maillage polygonal texturé) dont on peut effectuer des rendus pour des vues quelconques à des résolutions quelconques, ou encore dans le cas d'un objet réel dont on peut contrôler l'échantillonnage (par exemple si l'objet et l'écrairage sont statiques et si l'on dispose du matériel photographique adéquat).

Cet échantillonnage doit faire en sorte que chaque rayon lumineux de $\mathcal{E}$ ait la même importance, autrement dit que chacun de ces rayons ait la même probabilité d'intervenir lors d'un rendu de l'objet. Cela dépend des conditions de vue sous lesquelles l'objet peut être rendu. Nous précisons par la suite dans un cas concrêt quel échantillonnage peut être utilisé.

8.1.3 Objectif

Étant donnée l'expression paramétrique d'une représentation R ainsi qu'une distance sur l'ensemble des light-fields, le problème qui nous intéresse consiste à déterminer les paramètres optimaux $(t_1^\star, ..., t_p^\star) \in \mathbb{R}^n$ à utiliser pour la représentation R de manière à approcher au mieux l'apparence de l'objet O dont on connait le light-field f (sous forme échantillonnée). Ces paramètres sont définis de la manière suivante :

$$(t_1^\star, ..., t_p^\star) = \underset{t_1,...,t_p \in \mathbb{R}}{\arg\min}\, d(f^R_{t_1,...,t_p}, f)$$

Une autre manière de voir le problème est de le considérer comme un problème de compression du light-field en un ensemble de paramètres $(t_1, ...t_p)$: étant donné un budget précis de coût mémoire (le nombre p de paramètres t_i) et de temps de rendu (défini par la représentation R, indépendamment des valeurs t_i), on cherche à obtenir la meilleure apparence possible. De nombreux travaux existent sur la compression de light-fields (voir par exemple [CZRG06]), mais ils s'attachent généralement à obtenir de forts taux de compression indépendamment de l'efficacité permise au moment du rendu par la représentation obtenue. Nous prenons l'approche inverse en fixant une représentation adaptée au matériel graphique pour ensuite y projeter le light-field de manière optimale.

Notons enfin que cette approche permet la prise en compte du contexte dans lequel l'objet devra être rendu : la distance d étant définie sur un ensemble de rayons répartis uniformément parmi les vues sous-lesquelles l'objet peut être vu, la représentation optimale calculée prendra naturellement en compte d'éventuelles restrictions sur les points de vue possible pour répartir au mieux le détail que permet la représentation R (par exemple pour un objet voué à être adossé à un mur, la moitié de l'ensemble des rayons lumineux peut être ignorée, ce qui à budget fixé permet d'obtenir une représentation visuellement plus fidèle que si cette restriction n'est pas prise en compte). Cela permet aussi de prendre en compte des probabilités non uniformes sur les points de vue, lorsqu'elle sont connues *a priori*.

8.2 Travaux précédents

Le problème qui nous intéresse consiste à déterminer une certaine représentation d'un objet connaissant son light-field. Ceci peut s'apparenter dans une certaine mesure au problème de la reconstruction à base de vues multiples (*multi-view stereo reconstruction*), pour lequel à partir d'un ensemble (plus ou moins dense) de vues d'un objet ou d'une scène, l'on cherche à en reconstruire la géométrie, généralement représentée sous la forme d'une grille volumique (*voxel coloring* [SD99], *space*

carving [KS00]), de cartes de distance ou de maillages polygonaux. De nombreuses techniques ont été proposées (voir notamment Seitz et al. [SCD+06], Zeng et al. [ZPQS07] pour des états de l'art récents), le principal objectif visé étant la fidélité géométrique de la représentation calculée, avec comme principales difficultés le bruit ou le manque d'information dans les données en entrée.

Au contraire, nous cherchons à obtenir une représentation visuellement fidèle et efficace à rendre, en supposant disposer d'un échantillonnage dense et précis du light-field de l'objet considéré. Il existe peu de travaux se plaçant dans ce cadre-là. Certains travaux sur la simplification de maillages polygonaux [Hop99, LT00, ZT02], cherchant à prendre en compte l'apparence de l'objet lors du processus de simplification géométrique, abordent ce problème en partie. Cependant ces approches se restreignent aux maillage polygonaux et font appel à certaines heuristiques n'assurant pas l'obtention de la représentation optimale.

La technique des *billboard clouds* [DSDD02, DDSD03] consiste à déterminer par un processus d'optimisation globale la meilleure représentation d'un objet sous la forme d'un ensemble plans texturés semi-transparents. Seule la géométrie de l'objet fait l'objet de l'optimisation, les textures affectées à chaque plan étant déterminées par simple projection orthogonale de la géométrie associée (triangles d'orientation et position proches de celles du plan). De plus l'optimisation est réalisée par un algorithme glouton, ne garantissant pas l'obtention de l'optimum global.

Reche et al. [RMD04] proposent une méthode permettant de déterminer une représentation sous forme de grille volumique dont chaque cellule contient un polygone texturé faisant toujours face au point de vue, en utilisant comme seules données de départ plusieurs photographies d'un objet réel. Une grille volumique d'opacités est déterminée par un processus d'optimisation exact, puis une heuristique est appliquée pour déduire les textures contenues dans chaque cellule ayant été déterminée comme étant non transparente. Le peu d'information utilisée (une vingtaine de vues par objet) et les simplifications effectuées rendent cependant difficile la reproduction fidèle de l'aspect de l'objet original pour un point de vue quelconque.

Notons enfin que la question qui nous intéresse peut être vue comme un problème particulier de compression de *light-fields*. En raison de la quantité d'information importante que représente l'échantillonnage d'un light-field, il existe de nombreux travaux cherchant à compresser au mieux ces données [LH96, GGSC96, WAA+00, CBCG02, MRG03]. Cependant le but est généralement l'obtention de taux de compression élevés, tandis que notre principal objectif est la détermination des paramètres d'une représentation adaptée au matériel graphique actuel.

8.3 Application

Le cadre général dans lequel nous nous plaçons étant défini, montrons maintenant plus concrètement comment appliquer cette méthode à une représentation adaptée pour le rendu efficace sur GPU. Nous avons montré que l'objectif recherché, le calcul des paramètres optimaux pour une représentation donnée revient à

minimiser son light-field paramétrique pour une distance d déterminée. L'expression de ce problème de minimisation dépend complètement de la représentation étudiée.

Le choix de le représentation doit être guidé par l'efficacité qu'elle autorise au rendu, notamment la possibilité d'exploiter efficacement les capacités du matériel graphique. Idéalement on cherche une représentation dont le coût de rendu dépend du nombre de pixel occupés à l'écran (*output sensitive*), tel que le calcul de chaque pixel fasse intervenir peu de requêtes d'accès texture et pas de branchements.

8.3.1 Ensemble de polygones texturés

Nous nous intéressons au cas de la représentation par un ensemble de plans texturés. C'est un cas intéressant car bien supporté par le matériel graphique et beaucoup utilisé en pratique. Comme nous l'avons vu au chapitre 2, le rendu d'un polygone texturé peut être vu comme un compromis entre *forward rendering* et *backward rendering* permettant d'exploiter la cohérence de parallaxe entre plusieurs échantillons de couleur. Une telle représentation correspond bien aux critères d'efficacité que nous nous sommes fixé.

Limitations des techniques existantes

Bien que la représentation par polygones texturés soit très utilisée dans les applications en temps-réel, il n'existe pas de méthode automatique permettant de déterminer l'ensemble de textures optimal qu'il faut utiliser pour un objet donné. Cette tâche est généralement effectuée manuellement par un artiste, ce qui d'une part est fastidieux et d'autre part ne permet pas d'assurer l'optimalité.

La technique des *billboard clouds* [DSDD02, DDSD03] s'attaque à une sous-partie du problème : pour un objet O dont on possède une représentation polygonale, cette technique est capable de déterminer un ensemble de plans permettant de représenter au mieux la géométrie de l'objet. Cependant la deuxième partie de la représentation, l'ensemble des textures appliquées sur les plans, est déterminée par une heuristique (projection orthogonale sur chaque plan des triangles les plus proches) qui ne permet pas de garantir qu'on capture ainsi l'apparence optimale. D'autre part dans l'approche originale proposée, l'optimisation permettant de déterminer la position des plans n'est pas globale, elle effectue une recherche gloutonne. Des travaux plus récents s'attaquent à ce problème en faisant notamment appel à des algorithmes de *clustering* [GSSK05], mais aucun ne traite correctement le problème de la détermination des textures.

Idéalement on aimerait un algorithme automatique qui étant donné un certain nombre de plans texturés à une certaine résolution, nous donne les positions à utiliser pour ces plans, ainsi que les textures à y appliquer pour représenter de manière la plus fidèle possible l'apparence de l'objet original.

Représentation paramétrique

Les paramètres de cette représentation sont :

- le nombre K de plans utilisés
- pour chaque plan d'indice $k \in [\![0, K[\![$:
  - la position et l'orientation;
  - la résolution $m_k \times n_k$ de la texure plaquée;
  - la couleur de chaque texel $(i, j) \in [\![0, m_k[\![\times [\![0, n_k[\![$;
  - la paramétrisation utilisée : une simple transformation affine (matrice 3×2), une homographie (matrice 3×3) ou encore n'importe quelle transformation paramétrique de $\mathbb{R}^2$ qui peut s'avérer intéressante;
  - le mode de plaquage de la texture (répétition, couleur de bordure, etc.);
  - le mode d'interpolation des échantillons de la texture (plus proche voisin, bilinéaire);
  - la fonction de combinaison des différentes textures pour le rendu.

Si l'on suppose que les plans sont texturés à une même résolution $m \times n$ suffisamment élevée, cela représente grossièrement $K \times m \times n$ paramètres à optimiser.

Nous allons maintenant voir que sous certaines hypothèses, le light-field paramétrique associé à cette représentation est une combinaison linéaire de ses paramètres, ce qui permet d'en déterminer efficacement la valeur optimale, dans le cadre d'une distance euclidienne entre light-fields.

8.3.2 Plans parallèles additifs

L'hypothèse la plus contraignante que nous faisons dans cette partie, consiste à fixer la position des plans texturés (ainsi que la transformation de plaquage de texture). Nous allons montrer que sous cette hypothèse le light-field paramétrique associé à la représentation est une combinaison linéaire de ses paramètres (les valeurs de texels), ce qui présente l'intérêt de permettre son optimisation par l'application de la méthode des moindres carrés (linéaires), à condition d'utiliser une distance d euclidienne pour comparer les light-fields, ce qui est le cas de la distance que nous avons défini précédemment.

Les hypothèse que nous faisons ensuite servent uniquement à alléger les notations et l'explication mais pourraient être évitées sans perte de généralité.

Commençons par définir précisément la représentation étudiée pour pouvoir ensuite expliciter comment le problème d'optimisation s'exprime et comment il se resoud.

Hypothèses

Nous restreignons le problème en supposant fixe la position des plans texturés, et nous faisons le choix arbitraire de les répartir parallèlement, à intervalles réguliers le long de l'axe x. Notons que tout ce que nous allons voir par la suite s'applique pour des positions de plans arbitraires, simplement les notations dans ce cas particulier s'expriment plus simplement. Ainsi pour $k \in [\![0, K[\![$, le plan d'indice k a pour équation $x = x_k = \frac{2k}{K-1} - 1$. La texture correspondante, de résolution $m \times n$ est plaquée par transformation affine sur le carré $(x_k, \pm 1, \pm 1)$. Ainsi le centre du texel (i, j) de la texture k est positionné en (x_k, y_i, z_j) avec $y_i = 2\frac{i+1/2}{m} - 1$ et $z_j = 2\frac{j+1/2}{n} - 1$.

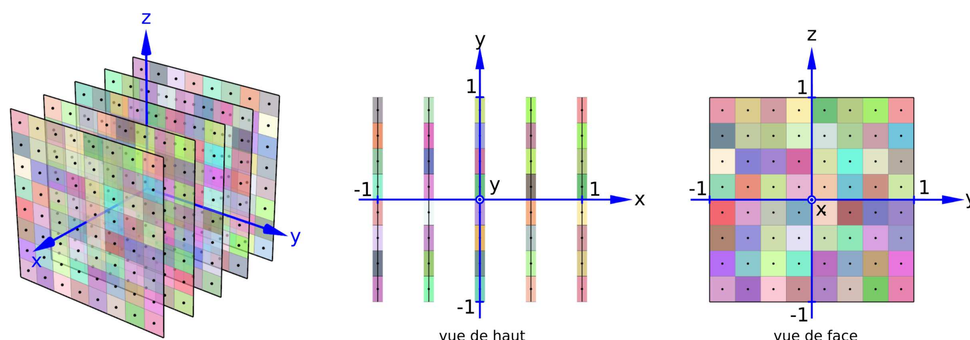


FIGURE 8.2 – Exemple de représentation composée de $K = 5$ plans parallèles texturés, de résolution $m \times n$ avec $m = n = 8$.

L'interpolation entre texels est l'interpolation de plus proche voisin ou l'interpolation bilinéaire (nous verrons plus tard que ces deux cas peuvent être traités de la même manière). Pour l'interpolation bilinéaire, l'interpolation des texels de bords se fait avec une couleur de de bordure nulle (comme si une ceinture de texels de valeur nulle entourait chaque texture). Ce sont les deux modes d'interpolation sur une texture 2D supportés par le matériel graphique.

Cette représentation en tranches parallèles est couramment utilisée pour rendre certains objets, notamment les objets à géométrie complexe comme les arbres [DN04] ou les représentations volumiques. Les valeurs des textures sont généralement calculées par une heuristique (comme pour les *billboard clouds*, en y plaquant la géométrie avoisinante). Calculer ces textures par optimisation de l'apparence, ne peut donc que produire de meilleurs résultats. D'autre part la structure due au placement régulier des plans en permet un raycasting très simple : ainsi il peut être plus efficace de n'afficher que les faces du cube $[-1, 1]$ pour ensuite combiner les couleurs trouvées dans chacune des textures par raycasting des plans parallèles.

Notons enfin que si cette représentation peut s'apparenter fortement à une représentation en grille volumique régulière, elle en diffère cependant par la manière dont sont interprétés les échantillons de couleur. Le principal avantage étant que le calcul de raycasting est plus simple et donc moins coûteux pour la représentation en ensemble de textures.

Le nombre de plans et la résolution des textures sont fixés, les paramètres de cette représentation sont donc les $N = m \times n \times K$ valeurs $(c_{ijk}) \in \mathcal{C}^N$, ce sont ces valeurs que l'on cherche à optimiser.

Échantillonnage du light-field

Rappelons que nous avons défini la distance entre light-fields à partir d'un échantillonnage $\mathcal{E}$ de l'ensemble des rayons $\mathcal{R}$, il faut donc définir cet échantillonnage.

La représentation par plans texturés parallèles, que nous nous proposons d'optimiser ne peut pas être utilisée pour rendre l'objet sous n'importe quel angle de vue : le pire cas correspondant aux directions de vue orthogonales à l'axe x, pour lesquelles les plans sont vus à angle rasant, ce qui empêche une restitution fidèle de l'apparence de l'objet, quel qu'il soit. D'autre part comme nous avons considéré

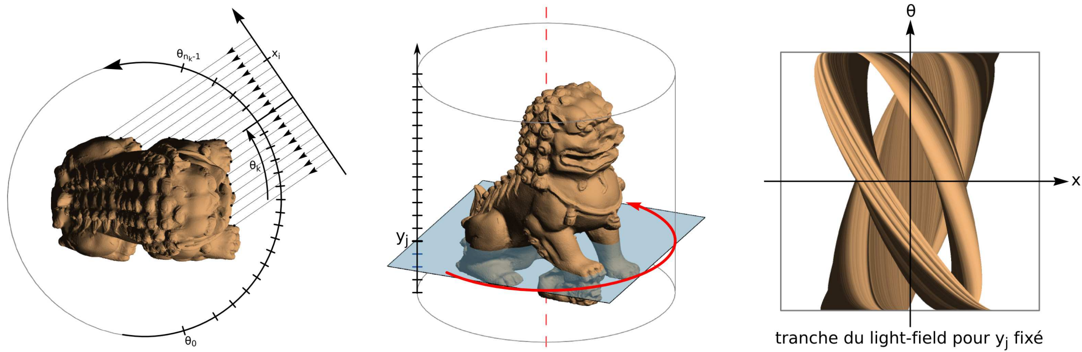


FIGURE 8.3 – Paramètres d'échantillonnage du *light-field* : l'image de droite correspond à l'échantillonnage en (x_i, θ_k) pour y_j fixé, c'est à dire à un ensemble de rayons lumineux contenus dans un plan horizontal (plan bleu clair de l'image centrale).

les textures additives, la couleur produite sur deux rayons de même support mais de sens opposés sera la même, ce qui là encore empêche l'utilisation de cette même représentation pour effectuer des rendus fidèles de l'objet initial à des angles de vues opposés. Pour ces raisons on supposera que la représentation ne sera utilisée que pour rendre des vues de direction $\theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$, avec une probabilité définie par une densité $p(\theta)$ décroissant avec l'écart angulaire par rapport à la vue de face : $p(\theta) = \cos\theta$. Cette probabilité peut être interprétée de deux manières différentes : soit elle correspond au fait que l'objet sera plus rarement observé à angle rasant que de face (et jamais de derrière), ce qui donne une importance plus faible à ces anges de vue, soit plusieurs représentations de ce type, à des angles différents, sont utilisées pour le rendu complet de l'objet, auquel cas la probabilité représente l'importance de la représentation actuelle pour le rendu final (par exemple par une vue presque rasante, la représentation actuelle ne donne presque aucune information utile sur l'image finale).

Pour prendre en compte ces conditions restreintes de rendu, nous utilisons sur l'échantillonnage $\mathcal{E}$ de l'ensemble des $q = n_i \times n_j \times n_k$ rayons lumineux suivants :

$$\forall (i,j,k) \in [\![0, n_i[\![\times [\![0, n_j[\![\times [\![0, n_k[\![\quad r_{ijk} = \left(d_k + x_i u_k + y_j e_z, -d_k\right)$$

avec

$$\begin{aligned} d_k &= (\cos\theta_k, \sin\theta_k, 0) \\ u_k &= (-\sin\theta_k, \cos\theta_k, 0) \\ \theta_k &= \arcsin(\frac{2k+1}{n_k} - 1) \\ x_i &= \frac{2i+1}{n_i} - 1 \\ y_j &= \frac{2j+1}{n_j} - 1 \end{aligned}$$

On peut se représenter cet échantillonnage comme l'ensemble des vues orthographiques horizontales de résolution $n_i \times n_j$ prises pour chaque angle θ_k.

Linéarité du cas additif

Nous montrons d'abord que dans le cas où les textures sont considérées additives (c'est-à-dire quand la fonction de mélange utilisée pour combiner les couleurs

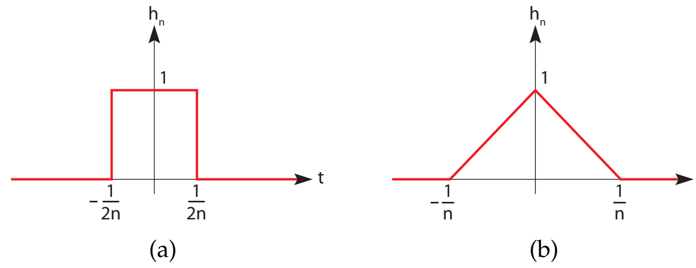


Figure 8.4 – Noyaux d'interpolation : (a) plus proche voisin; (b) linéaire.

rencontrées le long d'un rayon est l'addition), le light-field paramétrique associé à la représentation est une combinaison linéaire de ses paramètres, c'est-à-dire l'ensemble des valeurs de couleur des texels.

Nous faisons la simplification importante ici de ne pas chercher à représenter les occlusions : la quatrième composante des valeurs du light-field que l'on cherche à approcher est ignorée, on ne considère que des valeurs dans $\mathcal{C}$. De même, les texels de la représentation seront considérés à valeur dans $\mathcal{C}$. La représentation calculée ne pourra donc pas être insérée dans une scène en prenant correctement en compte la visibilité (les rayons n'intersectant pas l'objet O retourneront la couleur 0 c'est-à-dire noir), mais c'est un cas intéressant à étudier avant de traiter le modèle plus général permettant de prendre en compte les occultations.

Linéarité pour une texture unique Considérons le plan d'indice k. Soit $r = (q, d)$ un rayon lumineux et soit p son point d'intersection avec ce plan :

$$p = q + \frac{x_k - q_x}{d_x} d$$

Appelons $(u_k(r), v_k(r))$ l'expression de ce point dans le repère de la texture associée au plan, c'est-à-dire :

$$(u_k(r), v_k(r)) = \frac{p_{yz} + 1}{2} = \frac{1}{2}\left(q_{yz} + \frac{x_k - q_x}{d_x} d_{yz} + 1\right)$$

Appelons h_n la fonction noyau correspondant au mode d'interpolation de texture choisi, c'est-à-dire dans le cas de l'interpolation de plus proche voisin (figure 8.4(a)) :

$$h_n(t) = \begin{cases} 1 & \text{si} \quad |t| < \frac{1}{2n} \\ 0 & \text{sinon} \end{cases}$$

ou dans le cas de l'interpolation linéaire (figure 8.4(b)) :

$$h_n(t) = \begin{cases} 1 - n|t| & \text{si} \quad |t| < \frac{1}{n} \\ 0 & \text{sinon} \end{cases}$$

La valeur $c_k(u, v)$ obtenue par interpolation dans la texture au point (u, v) peut être exprimée comme suit :

$$c_k(u, v) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} h_m(u - u_i) h_n(v - v_j) c_{ijk}$$

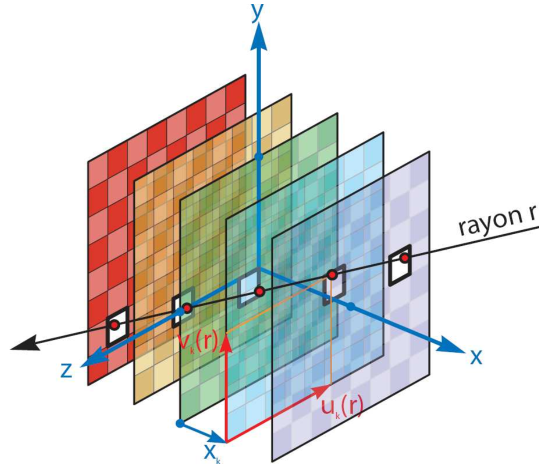


FIGURE 8.5 – La couleur associée à un rayon lumineux r correspond à la somme des couleurs obtenues aux intersections avec les plans texturés.

où (u_i, v_j) sont les coordonnées dans le repère de la texture du centre du texel (i, j) :

$$(u_i, v_j) = \left(\frac{i + 1/2}{m}, \frac{j + 1/2}{n}\right)$$

On voit donc que la couleur obtenue par interpolation dans la texture d'indice k est une combinaison linéaire des valeurs de ses texels, par les coefficients $\beta(u, v, i, j)$ définis comme suit :

$$\beta(u, v, i, j) = h_m(u - u_i)h_n(v - v_j)$$

Texture additives Si l'on considère maintenant le mélange additif des K textures, la couleur retournée pour un rayon r peut s'écrire :

$$\begin{aligned} f^R_{c_{ijk}}(r) &= \sum_{k=0}^{K-1} c_k\left(u_k(r), v_k(r)\right) \\ &= \sum_{k=0}^{K-1}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} \beta_{ijk}(r)c_{ijk} \end{aligned}$$

où les coefficients $\beta_{ijk}(r)$ sont définis par :

$$\beta_{ijk}(r) = \beta(u_k(r), v_k(r), i, j)$$

On voit donc que la valeur donnée par le light-field paramétrique de la représentation R pour un rayon r est une combinaison linéaire des valeurs de couleur des texels de la représentation, avec des coefficients dépendant uniquement du rayon considéré.

Notons que l'on peut se représenter la fonction qui à r associe le coefficient $\beta_{ijk}(r)$ comme un light-field (à valeurs réelles). Le problème revient alors à exprimer le light-field paramétrique de la représentation R comme une somme de N light-fields fixés (les β_{ijk}) avec des coefficients c_{ijk} à déterminer.

Optimisation par moindres carrés Voyons maintenant comment le problème d'optimisation du light-field paramétrique s'exprime dans ce cas :

$$\begin{aligned}(c_{ijk})^\star &= \underset{(c_{ijk})\in\mathcal{C}^N}{\arg\min}\, d(f^R_{c_{ijk}}, f) \\ &= \underset{(c_{ijk})\in\mathcal{C}^N}{\arg\min}\, \|f^R_{c_{ijk}} - f\|^2 \\ &= \underset{(c_{ijk})\in\mathcal{C}^N}{\arg\min}\, \|\sum_{i,j,k}\beta_{ijk}c_{ijk} - f\|^2\end{aligned}$$

C'est un problème de moindres carrés, linéaire car $f^R_{c_{ijk}}$ dépend linéairement des coefficients c_{ijk}, qui peut se ramener à la résolution d'un simple système linéaire, et pour lequel des algorithmes efficaces existent, permettant de déterminer le minimum global de manière exacte (nous utilisons la décompotition QR, calculée par la méthode de Householder [GVL96]).

Par ailleurs la matrice définissant le système est très creuse : une ligne de cette matrice, correspondant à la relation entre un rayon lumineux de $\mathcal{E}$ avec les N coefficients (c_{ijk}) contient au plus $4 \times K$ valeurs non nulles (pour chacun des K plans, le rayon intersecte le domaine d'au plus quatre texels), ce qui fait une proportion de seulement $4/mn$ valeurs non nulles. Il est donc possible d'utiliser une méthode de résolution basée sur la décomposition QR adaptée aux matrices creuses [Dav06], qui accélère de manière significative le calcul d'optimisation.

Enfin notons un point important sur le domaine des valeurs (c_{ijk}) : ces valeurs sont, dans la représentation R originale, restreintes à l'ensemble borné $\mathcal{C} = [0,1]^3 \subset \mathbb{R}^3$. Cependant la méthode des moindres carrés classique ne permet pas de prendre en compte cette contrainte, les valeurs calculées par optimisation sont donc non bornées, c'est-à-dire dans $\mathbb{R}^3$. En fait rien n'oblige la représentation par plans texturés à se cantonner à des composantes de couleur dans $[0,1]$: on peut tout à fait utiliser des valeurs quelconques si cela permet d'obtenir un résultat plus optimal, à condition de pouvoir produire au rendu final (par combinaison des plans texturés) des couleurs dans $\mathcal{C}$. Les couleurs du light-field objectif f que l'on cherche à approcher étant à valeurs dans $\mathcal{C}$, le résultat produit par optimisation doit s'en approcher au mieux.

Il est cependant possible de prendre en compte les contraintes de bornes sur les valeurs (c_{ijk}) en observant que le problème se présente alors comme un problème de *programmation quadratique*, consistant à minimiser une fonction quadratique sous un ensemble de contraintes linéaires. C'est précisément notre cas puisque nous cherchons à minimiser $\|f^R_{c_{ijk}} - f\|^2$, fonction quadratique des (c_{ijk}), sous les contraintes linéaires $c_{ijk} \geqslant 0$ et $c_{ijk} \leqslant 1$. Pour résoudre ce problème nous utilisons une implémentation de l'algorithme de Goldfarb et Idnani [GI83].

Résultats

La résolution par la méthode des moindres carrés implique l'inversion d'une matrice de taille $N \times N$. Sachant que $N = m \times n \times K$, cette matrice peut s'avérer être très grande si l'on veut optimiser une représentation faites de textures d'une

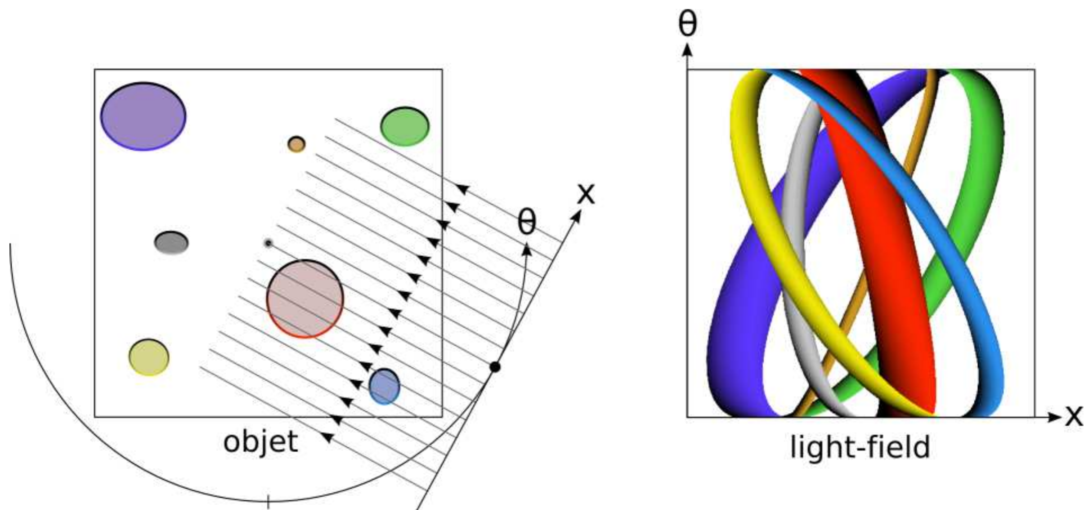


FIGURE 8.6 – À gauche : tranche 2D horizontale de l'objet que l'on cherche à représenter, à droite : le light-field 2D correspondant, pour l'échantillonnage des rayons lumineux en (x, θ) décrit précédemment.

résolution suffisante. Par exemple, l'optimisation de 32 plans texturés d'une résolution 64^2 implique une matrice de taille 524288^2, ce qui est impraticable.

Remarquons que si l'on fait le choix pour la représentation R que l'interpolation verticale (selon l'axe z) des textures est une interpolation de plus proche voisin, alors on peut décomposer le système complet de taille $m \times n \times K$ en n sous-systèmes indépendants de taille $m \times K$: en effet, par le fait que les rayons de l'ensemble $\mathcal{E}$ sont horizontaux, il n'existe pas de relation entre un texel c_{ijk} et un texel $c_{i'j'k'}$ si $j \neq j'$. Ainsi pour l'exemple que nous venons de donner, on est ramené à la résolution de 64 systèmes de taille 512^2 ce qui devient faisable. Notons que cela revient à considérer séparément n light-fields 2D correspondant à des tranches horizontales de l'objet.

Nous montrons d'abord des résultats sur une tranche 2D d'un objet, ce qui permet d'en visualiser le light-field 2D simplement à l'aide d'une seule image. Nous prenons comme référence la tranche d'objet donnée par la figure 8.6, qui en montre aussi le light-field associé.

La figure 8.7 montre, pour différentes valeurs de K et m, l'ensemble de textures calculées par optimisation et le light-field correspondant. La méthode utilisée est la méthode des moindres carrés sans contraintes, les valeurs de texels peuvent donc prendre des valeurs négatives (les couleurs de texels montrées sur la figure sont ramenées dans $\mathcal{C}$ par simple application de seuils). On constate qu'en augmentant la taille de la représentation (nombre de textures et résolutions), le light-field approché converge vers le light-field initial de l'objet. Un point intéressant à noter est que les textures calculées diffèrent complètement de ce que l'on aurait obtenu en plaquant sur chaque plan texturé les parties proches de l'objet.

L'utilisation de la méthode d'optimisation par programmation quadratique permet de borner les valeurs calculées dans $[0, 1]$. La figure 8.8 montre l'effet produit. On peut constater que le light-field ainsi produit est de moins bonne qualité que celui obtenu sans la contrainte des valeurs bornées. D'autre part, les valeurs des texels calculées sont proches des couleurs de l'objet original (on voit apparaître les

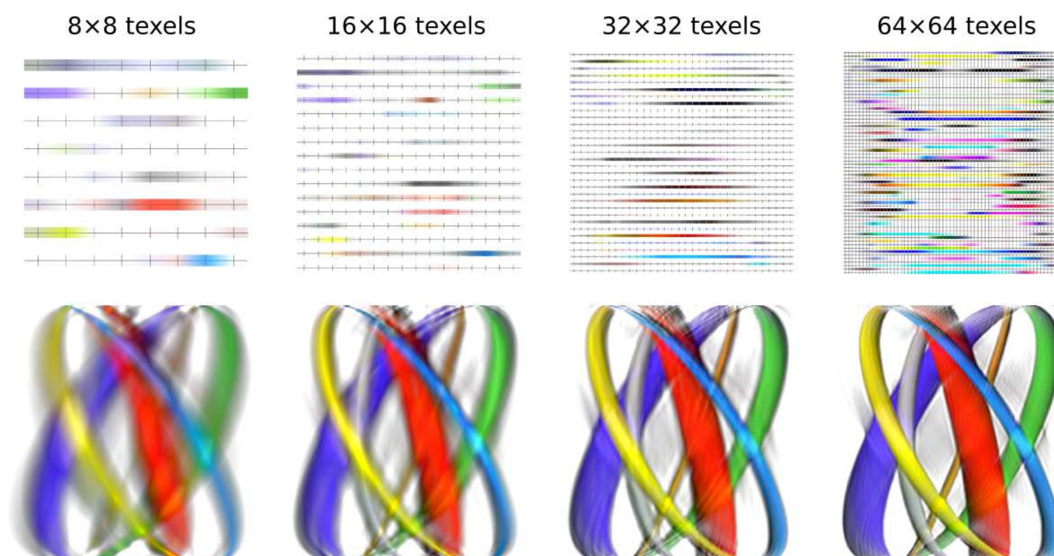


FIGURE 8.7 – En haut : ensemble de K textures de résolution m pour des valeurs de $K = m \in \{8, 16, 32, 64\}$. En bas : les light-fields correspondants.

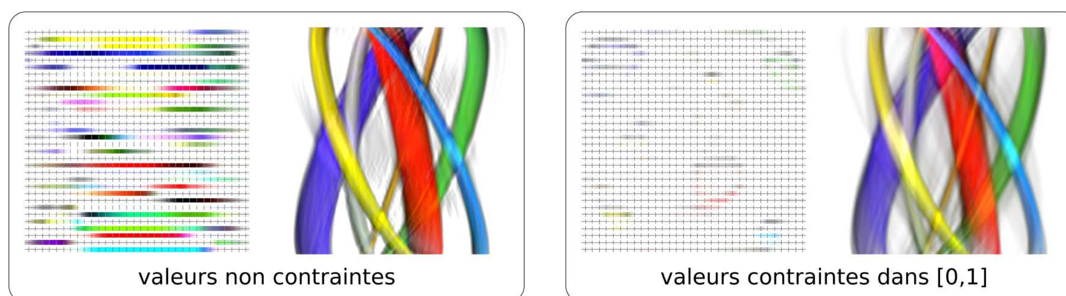


FIGURE 8.8 – Textures optimales calculées pour $K = m = 32$, par moindres carrés sans contraintes (à gauche), par programmation quadratique avec contrainte de valeurs dans $[0, 1]$ (à droite).

formes de l'objet dans les textures). Ce n'est pas le cas lorsque les valeurs sont non contraintes car il est alors possible d'utiliser par exemple des valeurs négatives pour compenser des valeurs trop élevées. Les temps de calculs nécéssaires pour effectuer une optimisation par programmation quadratique sont beaucoup plus élevés qu'avec la méthode de moindres carrés sans contrainte. Le tableau 8.1 donne des temps de calcul pour différentes valeurs de q et N. Sauf indication du contraire, les résultats qui suivent sont calculés par la méthode de moindres carrés sans contraintes.

Il est possible de réduire l'intervalle $[\theta_{min}, \theta_{max}]$ utilisé pour échantillonner le light-field. La figure 8.9 montre qu'en restreignant cet intervalle on peut calculer une représentation permettant de mieux approcher l'apparence de l'objet, ce qui peut être intéressant dans le cas où l'objet est voué à être rendu dans des conditions de vue restreintes.

Le simple changement du noyau d'interpolation h_n permet d'optimiser la représentation pour différentes méthodes d'interpolation des textures. Comme on peut le voir sur la figure 8.10, les texures calculées dans le cas de l'interpolation linéaire et dans le cas de l'interpolation de plus proche voisin diffèrent complètement.

(a)

N \ q	32^2	64^2	128^2	256^2
8^2	0.02	0.07	0.3	1.2
16^2	0.13	0.6	2.5	10.5
32^2		6.9	31.4	130
64^2			413	1908

(b)

N \ q	32^2	64^2	128^2
8^2	0.35	1.25	17.4
16^2	11.2	84.2	319
32^2		3174	7330

TABLE 8.1 – Temps de calcul en secondes pour l'optimisation de N texels par un échantillonnage de q rayons lumineux, (a) par la méthode des moindres carrés sans contraintes et (b) par programmation quadratique, sur un processeur Intel Core 2 Quad Q6700 à 2.66 GHz (les cases vides correspondent aux cas où le système est sous-dimensionné).

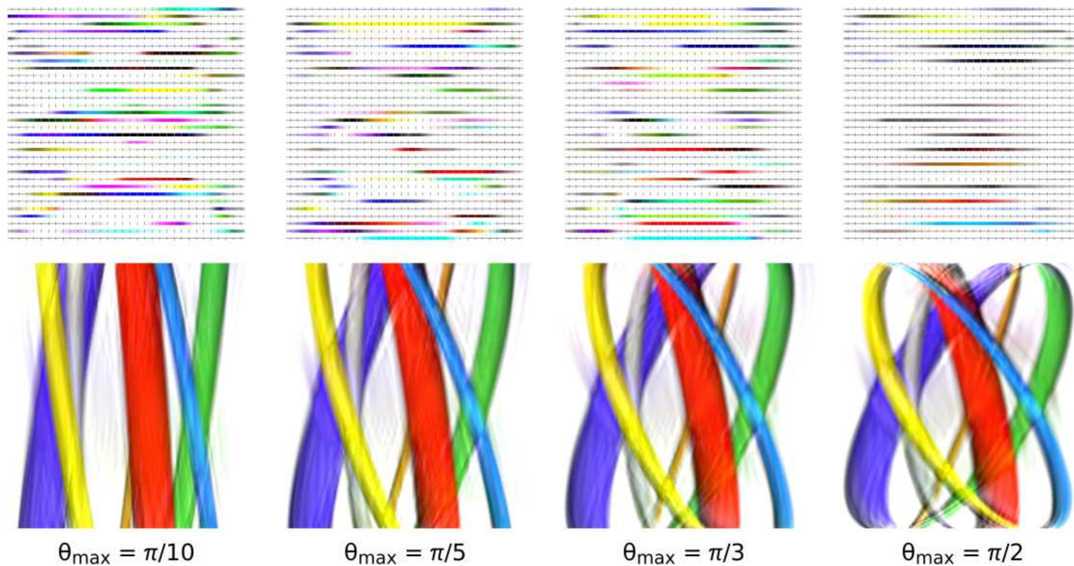


FIGURE 8.9 – Textures optimales et light-fields associés pour θ échantillonné dans l'intervalle $[-\theta_{max}, \theta_{max}]$ avec $\theta_{max} \in \{\pi/10, \pi/5, \pi/3, \pi/2\}$.

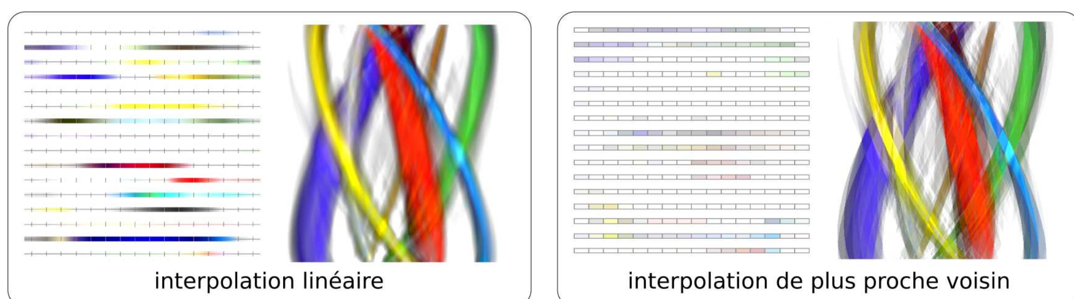


FIGURE 8.10 – Optimisation de 16 textures constituées de 16 texels, pour l'interpolation linéaire (à gauche) et pour l'interpolation de plus proche voisin (à droite).

FIGURE 8.11 – Représentation optimisée de 32 textures de résolution 32×64 (en bas) et rendus obtenus pour différents angles de vue (en haut).

Regardons maintenant ce que cela donne sur un objet entier. La figure 8.11 montre un exemple d'objet représenté par 32 textures de résolution 32×64 calculées par optimisation. Cette représentation est équivalente en taille mémoire à une image de taille 256×512. Les rendus synthétisés en utilisant cette représentation sont visuellement fidèles à l'objet original, sauf pour les directions de vue presque parallèles aux plans texturés. Encore une fois on remarque que les rendus obtenus sont *a priori* meilleurs de ceux que l'on aurait obtenu en prenant simplement la géométrie environnante des plans texturés pour déterminer le contenu de leurs textures associées, et ce malgré la fonction de combinaison simple (additive) que nous avons utilisée.

Au vu des résultats obtenus, cette approche par optimisation semble prometteuse, malgré la quantité de degrés de libertés pas encore exploités (pas exemple la disposition des plans texturés). Avec une simple combinaison additive il est déja possible d'approcher l'apparence d'un objet fidèlement avec relativement peu de texels.

Cependant le plus gros défaut reste lié aux hautes fréquences du light-field : les silhouettes nettes du light-field original se retrouvent floues après approximation, et il apparait aussi des effets de fantômes, des couleurs appartenant à différentes partie de l'objet se mélangeant, qui deviennent plus perceptibles et gênants lorsque l'objet tourne.

Ces problèmes peuvent être en partie évités en ajoutant une prise en compte de la visibilité dans le calcul d'optimisation de la représentation, que nous traitons maintenant.

8.3.3 Prise en compte de l'occultation

Il faut garder à l'esprit que la représentation que nous cherchons à calculer est vouée à être rendue parmi d'autres objets de la scène complète. Pour pouvoir traiter correctement la visibilité entre l'objet que l'on cherche à représenter et la partie de la scène se trouvant derrière son cylindre englobant, il est nécesaire de disposer d'une information d'opacité en plus de l'information de couleur pour chaque rayon lumineux.

Pour que le light-field paramétrique de la représentation R à base de textures que nous étudions puisse produire des valeurs d'opacité, le moyen le plus simple consiste à intégrer cette information aux textures, c'est-à-dire leur ajouter un canal α d'opacité. Pour ceci on associe à chaque couleur de texel c_{ijk} de la représentation une valeur d'opacité qui sera notée α_{ijk}. Nous noterons $p_{ijk} \in \mathcal{C}_{\mathcal{A}}$ la valeur associée au texel (i, j) de la texture d'indice k, autrement dit $p_{ijk} = (c_{ijk}, \alpha_{ijk})$.

Le light-field paramétrique que nous cherchons à optimiser est donc maintenant le suivant :

$$f^R_{p_{ijk}} : \mathcal{R} \longrightarrow \mathcal{C}_{\mathcal{A}}$$

Se posent alors principalement deux options pour le choix de l'ensemble des valeurs que peut prendre cette information d'opacité : soit l'intervalle $[0, 1]$, pour une opacité continue, soit l'ensemble $\{0, 1\}$, pour une opacité binaire. Le choix entre opacité continue et opacité binaire doit se faire en fonction des besoins de l'application et des possibilités qu'elle offre. Utiliser une opacité continue permet de représenter des objets transparents mais impose de les rendre dans un ordre précis, de l'arrière vers l'avant ou inversement (car la fonction de mélange n'est pas commutative), alors qu'utiliser une opacité binaire permet par l'utilisation du mechanisme de résolution des occultations fourni par la carte de profondeur de rendre les objets dans un ordre quelconque.

Nous montrons d'abord comment l'information d'opacité peut être utilisée pour combiner les valeurs obtenues dans les textures rencontrées successivement le long d'un rayon lumineux, puis nous traitons les deux cas d'opacité, continue et binaire.

Combinaison entre textures

Il faut maintenant redéfinir la fonction utilisée pour combiner les textures entre elles, c'est-à-dire la fonction appliquée pour combiner la succession de couleurs obtenues par interpolation dans chaque texture à la position rencontrée un le rayon de vue. Nous cherchons donc une fonction b qui à K valeurs de $\mathcal{C}_{\mathcal{A}}$ fasse correspondre une valeur dans $\mathcal{C}_{\mathcal{A}}$:

$$b : {\mathcal{C}_{\mathcal{A}}}^K \longrightarrow \mathcal{C}_{\mathcal{A}}$$

La fonction de combinaison la plus couramment utilisée pour mélanger des textures semi-transparentes consiste à appliquer successivement la fonction de mélange donnée précédemment. Ceci s'explique par le sens physique que revet cette fonction pour la prise en compte d'occultations ou de semi-transparence. En appliquant cette fonction de mélange successivement à K valeurs $p_k = (c_k, \alpha_k) \in \mathcal{C}_{\mathcal{A}}$

ordonnées de la plus proche à la plus lointaine, on obtient l'expression suivante de la fonction b :

$$b(p_1, ..., p_K) = \sum_{k=1}^{K} \left(\prod_{i=1}^{k-1} (1 - \alpha_i) \right) p_k$$

Notons d'abord que la fonction b n'est pas commutative : son résultat dépend généralement de l'ordre dans lequel les valeurs (p_i) sont considérées. Cependant si l'on considère uniquement l'expression de la composante d'opacité retournée par la fonction b, celle-ci peut se simplifier :

$$\begin{aligned}
\alpha(\alpha_1, ..., \alpha_n) &= b(p_1, ..., p_n)|_\alpha = \sum_{k=1}^{n} \alpha_k \prod_{i=1}^{k-1} (1 - \alpha_i) \\
&= \sum_{k=1}^{n} (1 - (1 - \alpha_k)) \prod_{i=1}^{k-1} (1 - \alpha_i) \\
&= \sum_{k=1}^{n} \left(\prod_{i=1}^{k-1} (1 - \alpha_i) - \prod_{i=1}^{k} (1 - \alpha_i) \right) \\
&= 1 - \prod_{i=1}^{n} (1 - \alpha_i)
\end{aligned}$$

Si l'on exprime cette relation en fonction des transparences au lieu des opacités, on constate qu'elle est en accord avec le caractère multiplicatif de la notion physique de transparence :

$$\tau(\tau_1, ...\tau_n) = \prod_{i=1}^{n} \tau_i$$

Cette formule possède la bonne propriété d'être commutative : la transparence résultant de la combinaison de plusieurs objets semi-transparents est indépendante de l'ordre dans lequel ces objets sont traversés par la lumière. Nous montrons plus loin quel avantage cela représente pour le problème qui nous intéresse.

Découplage couleur/ opacité

Maintenant que nous avons défini les changements induits par l'ajout du canal d'opacité, exprimons la valeur (dans $\mathcal{C}_\mathcal{A}$) du light-field paramétrique de cette nouvelle représentation, pour un rayon lumineux r intersectant successivement les plans d'indices croissants :

$$\begin{aligned}
f^R_{p_{ijk}}(r) &= b(p_0(r), ..., p_{K-1}(r)) \\
&= \sum_{k=0}^{K-1} \prod_{k'=0}^{k-1} (1 - \alpha'_k(r)) p_k(r) \\
&= \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \sum_{k=0}^{K-1} \left(\prod_{k'=0}^{k-1} \sum_{i'=0}^{m-1} \sum_{j'=0}^{n-1} \beta_{i'j'k'}(r) \left(1 - \alpha_{i'j'k'}\right) \right) \beta_{ijk}(r) p_{ijk}
\end{aligned}$$

Cette fois l'expression n'est pas linéaire par rapport aux paramètres (p_{ijk}) de la représentation : les termes de la somme sont composés de produits de plusieurs de

ces paramètres. On ne peut donc pas recourir à la même méthode d'optimisation que précédemment.

Constatons que la non-linéarité a été introduite par le produit de paramètres d'opacité avec des paramètres de couleur. Si l'on considère fixées les valeurs d'opacité, le problème redevient linéaire, seuls les coefficients affectés aux paramètres de couleur sont modifiés. Donc si l'on suppose avoir trouvé les paramètres d'opacité correspondant à la valeur optimale du light-field paramétrique, on peut directement en déduire les paramètres de couleur en faisant appel à une méthode de résolution de moindres carrés linéaires, comme précédemment à une modification près des coefficients du sytème d'équations. D'autre part, on peut remarquer que les valeurs d'opacité du light-field paramétrique ne dépendent que des valeurs d'opacité de ses paramètres :

$$f_{(p_{ijk})}(r)\big|_{\alpha} = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\sum_{k=0}^{K-1}\left(\prod_{k'=0}^{k-1}\sum_{i'=0}^{m-1}\sum_{j'=0}^{n-1}\beta_{i'j'k'}(r)\left(1-\alpha_{i'j'k'}\right)\right)\beta_{ijk}(r)\alpha_{ijk}$$

Nous avons suivi cette approche, consistant à découpler l'optimisation des paramètres d'opacité de celle des paramètres de couleur. Les valeurs d'opacité sont déterminées indépendamment des valeurs de couleur, puis les valeurs de couleur sont déterminées par optimisation, en connaissant les valeurs d'opacité.

Autrement dit, pour simplifier on peut se représenter la fonction à minimiser de la manière suivante :

$$\begin{aligned} f : \mathcal{C}\times[0,1] &\longrightarrow \mathcal{C}\times[0,1] \\ (c,\alpha) &\longmapsto (f_1(c,\alpha), f_2(\alpha)) \end{aligned}$$

Notons que f_2 ne dépend que de α. On cherche d'abord la valeur $\hat{\alpha}$ optimale pour cette fonction :

$$\widehat{\alpha} = \arg\min_{\alpha\in[0,1]} f_2(\alpha)$$

Ensuite, en fixant cette valeur on détermine la valeur $\widehat{c}$ optimale pour la fonction f_1 :

$$\widehat{c} = \arg\min_{c\in\mathcal{C}} f_1(c,\widehat{\alpha})$$

Il est important cependant de noter ici qu'une telle approche ne garantit pas l'obtention de l'optimum global, le couple $(c^\star,\alpha^\star)$ tel que :

$$(c^\star,\alpha^\star) = \arg\min_{(c,\alpha)\in\mathcal{C}\times[0,1]} f(c,\alpha)$$

Rien ne permet d'affirmer que la composante $\alpha^\star$ de cet optimum global soit optimale pour la fonction f_2, c'est-à-dire que $\alpha^\star = \widehat{\alpha}$ (la seule chose que l'on puisse affirmer c'est que $f_2(\alpha^\star) \geqslant f_2(\widehat{\alpha})$).

Nous montrons qu'une telle approche sous-optimale produit déja des résultats très intéressants, donnant une idée de la qualité minimum de ce que pourrait produire une technique d'optimisation globale, sur laquelle il serait alors intéressant de se pencher.

Nous traitons d'abord le cas où l'information d'opacité est considérée comme une valeur continue, avant d'étudier la cas particulier d'une opacité à valeurs binaires.

Opacité continue

Nous cherchons donc d'abord à calculer la valeur optimale de l'opacité. Rappelons que l'opacité est le complément à 1 de la transparence, dont la valeur paramétrique pour un rayon r peut s'écrire :

$$\tau(r) = \prod_{k=1}^{K} \tau_k(r)$$

Pour chercher à linéariser cette expression, on pourrait être tenté d'y appliquer un logarithme en posant $\delta = -\ln \tau$ (donc $\delta \in \mathbb{R}^+ \cup \{+\infty\}$) et ainsi transformer le produit en somme :

$$\delta(r) = \sum_{k=1}^{K} \delta_k(r)$$

Cependant, les valeurs $\delta_k(r) = -\ln \tau_k(r)$ ne peuvent pas être exprimées comme combinaisons linéaires des paramètres $\delta_{ijk} = -\ln \tau_{ijk}$, car l'interpolation des texels s'effectue sur les valeurs de transparence, et non pas sur leurs logarithmes :

$$\delta_k(r) = -\ln \left(\sum_{i,j=0}^{n-1} \beta_{ijk}(r) \tau_{ijk} \right) \neq \sum_{i,j=0}^{n-1} \beta_{ijk}(r) \delta_{ijk}$$

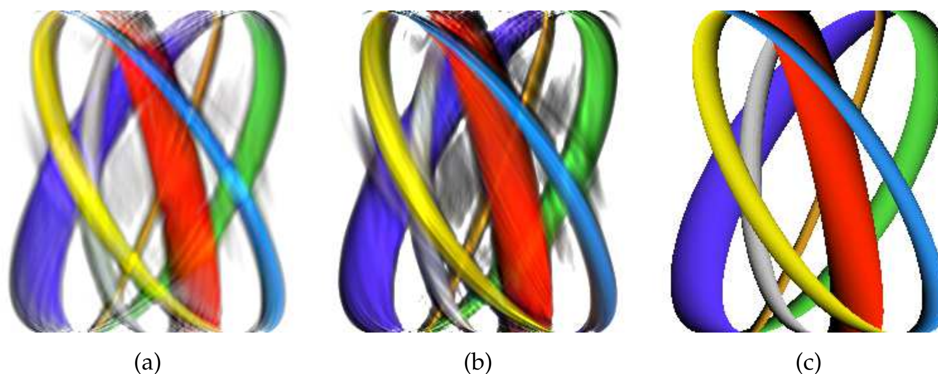
Il existe une solution simple à ce problème : il suffit de modifier légèrement la représentation en remplaçant simplement les valeurs d'opacité α associées à la quatrième composante de chaque texel par les valeur $\delta = -\ln(1-\alpha)$ associées, pour que l'interpolation s'effectue directement sur ces valeurs au lieu d'être calculée sur les valeurs d'opacité. Cette transformation revet une signification physique : le paramètre δ correspond à la notion d'*épaisseur optique* (ou profondeur optique). Il représente à une constante près la transparence $\tau = e^{-\delta}$ associée à un matériau homogène d'épaisseur δ. L'épaisseur optique est par nature une grandeur additive alors que la transparence est multiplicative. L'interpolation de texture s'exprimant comme une somme pondérée des valeurs contenues dans les texels, il parrait donc plus naturel que cette interpolation soit effectuée sur des valeurs d'épaisseur optique plutôt que sur des valeurs de transparence. Notons un pont qui peut sembler problématique : une valeur d'opacité de 1 correspond à une valeur δ infinie. Autrement dit une profondeur optique finie ne peut pas représenter un objet complètement opaque. Cependant la transparence converge très vite vers zéro lorsque $delta$ croît : par exemple une profondeur optique de 20 correspond déja à une transparence de 2.10^{-9}. Il suffit donc de fixer un seuil à partir duquel une valeur d'opacité est considérée comme égale à 1.

Ainsi dans cette nouvelle représentation l'interpolation effectuée sur la composante δ devient simplement :

$$\delta_k(r) = \sum_{i,j=0}^{n-1} \beta_{ijk}(r) \delta_{ijk}$$

ce qui correspond à l'interpolation “multiplicative” suivante pour les valeurs de transparence :

$$\tau_k(r) = \prod_{i,j=0}^{n-1} {\tau_{ijk}}^{\beta_{ijk}(r)}$$



(a) (b) (c)

FIGURE 8.12 – Optimisation de 32 textures constituées de 32 texels, (a) sans et (b) avec prise en compte de l'information d'opacité; (c) light-field de l'objet original.

La fonction de combinaison des valeurs de texture, doit aussi être adaptée à cette nouvelle représentation, définie sur l'enssemble $\mathcal{C}_{\mathcal{D}} = \mathcal{C} \times \mathbb{R}^{+\star}$:

$$\begin{aligned} b_\delta : {\mathcal{C}_{\mathcal{D}}}^N &\longrightarrow \mathcal{C}_{\mathcal{D}} \\ \left(\begin{bmatrix} c_1 \\ \delta_1 \end{bmatrix}, \ldots, \begin{bmatrix} c_N \\ \delta_N \end{bmatrix} \right) &\longmapsto \sum_{k=1}^{n} \begin{bmatrix} e^{-\sum_{i=1}^{k-1} \delta_i} c_k \\ \delta_k \end{bmatrix} \end{aligned}$$

Revenons maintenant à notre problème initial, consistant à calculer les paramètres optimaux de transparence de notre représentation pour approcher aux mieux celles du light-field initial. Ce problème s'exprime maintenant de manière linéaire, puisque :

$$f^R_{p_{ijk}}(r)\big|_\delta = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \sum_{k=0}^{K-1} \beta_{ijk}(r) \delta_{ijk}$$

On obtient exactement le même système d'équations que dans le cas sans prise en compte des occulsions, qui peut alors être résolu de la même manière, en appliquant l'approche des moindres carrés linéaires.

Résultats Comme le montre la figure 8.12, les résultats obtenus sont intéressants, et confirment l'intérêt de cette approche malgré son caractère sous-optimal, puisque les images obtenues sont relativement fidèles aux images originales. Par rapport à la méthode précédente sans prise en compte de l'opacité, le light-field approché est plus net et les différentes parties ont moins tendance à se mélanger.

Le gain apporté est cependant décevant, il reste des artefacts visuels, les plus gênants étant les effets de fantômes aux silhouettes (silhouettes floues). Ils proviennent des hautes fréquences comprises dans la fonction de visibilité (l'opacité) des objets que nous cherchons à représenter : ces objets ne présentent pas de parties semi-transparentes, donc l'opacité prend pour chaque rayon lumineux la valeur 0 ou 1. C'est le cas pour la grande majorité des objets du monde réel. La fonction de combinaison que nous utilisons est continue, ce qui la rend inadaptée à ce type d'objets. Une façon simple de la rendre binaire serait d'ajouter un seuil (par exemple 1/2) sur les valeurs que prend cette fonction, déterminant si la valeur de sortie doit valoir 0 ou 1. Cependant il ne suffit pas de rajouter un tel test à la fonction de combinaison après optimisation, il faut que cette nouvelle expression du light-field

paramétrique soit prise en compte au moment de l'optimisation. Un tel test binaire ajoute une non-linéarité qui ne permet pas d'utiliser l'approche des moindres carrés linéaires. Nous montrons dans la partie suivante comment effectuer une telle optimisation, ainsi que les autres possibilités pour représenter une opacité binaire.

Opacité binaire

L'aspect discontinu de la fonction de visibilité pour un objet opaque rend difficile sa représentation à l'aide d'une fonction continue comme celle qui nous venons de présenter. La solution la plus simple pour faire prendre des valeurs binaires à cette fonction consiste à lui appliquer une fonction de seuil comme la fonction s suivante :

$$\begin{aligned} s : [0,1] &\longrightarrow \{0,1\} \\ \alpha &\longmapsto \begin{cases} 0 & \text{si} \quad \alpha < \frac{1}{2} \\ 1 & \text{sinon} \end{cases} \end{aligned}$$

La fonction $\tilde{f} = s \circ f$ n'étant plus linéaire, il faut utiliser une autre méthode d'optimisation.

Nous avons utilisé la méthode du recuit simulé [KGV83], qui fait partie des méthodes d'optimisation permettant de résoudre des problèmes pour lesquels aucune propriété particulière de la fonction à optimiser n'est connue, qui permette la résolution par une méthode classique. Cette méthode appartient à la classe de méthodes appelées *métaheuristiques* [DPÉTS03].

Cette méthode consiste à rechercher la solution optimale itérativement, où chaque nouvelle position à tester est déterminée en se déplaçant aléatoirement de la position précédente. Quand la nouvelle position s'avère plus optimale on la conserve. Dans le cas contraire on s'autorise à aller vers cette solution moins optimale mais seulement avec une probabilité, déterminée en fonction de la différence de valeur que produit cette nouvelle position avec celle que produisait la précédente, et d'une certaine énergie (ou température, par analogie avec la métallurgie, dont est inspirée cette méthode), décroissant à chaque itération effectuée. Faire fonctionner cet algorithme correctement demande de bien régler la loi de probabilité utilisée pour le parcours aléatoire et la loi de température. Il peut être avantageux aussi de disposer d'une estimation initiale de l'optimum.

Le seul élément *a priori* coûteux de l'algorithme est l'évaluation de la fonction que l'on cherche à minimiser, c'est-à-dire dans notre cas, l'échantillonnage du light-field paramétrique de notre représentation, pour des valeurs de paramètres variant à chaque itération, suivi du calcul de l'erreur associée, correspondant à une somme sur l'ensemble des rayons échantillonnés. Ceci peut s'implémenter de manière efficace sur le GPU, puisque les calculs à effectuer sur les rayons sont indépendants et donc parallélisables. D'autre part le calcul à effectuer pour un rayon est simple puisque il correspond au raycasting de la représentation en textures, que nous avons justement choisie car elle permet un rendu efficace sur GPU.

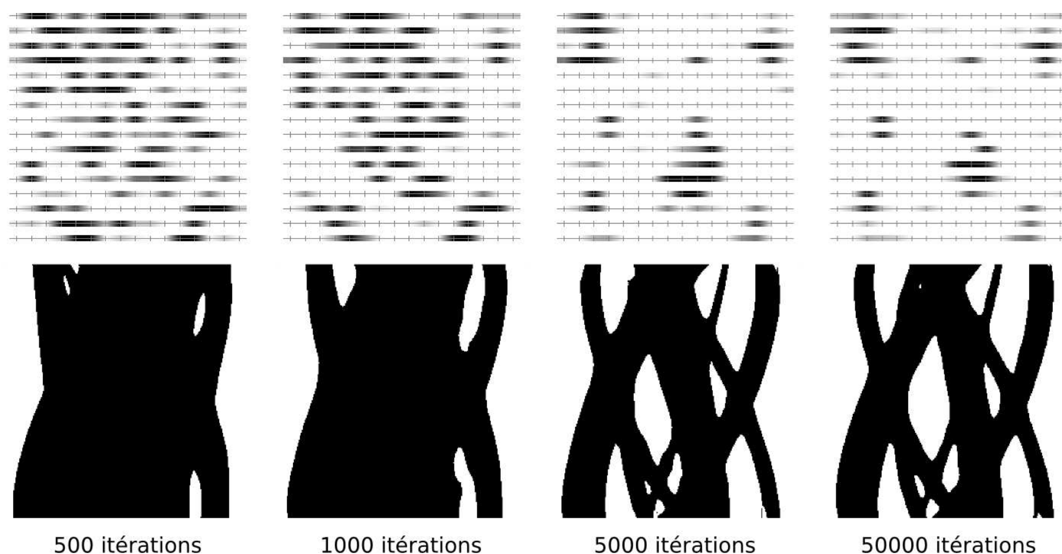


FIGURE 8.13 – Résultat de l'algorithme de recuit simulé pour l'optimisation de 16^2 texels, pour différents nombres d'itérations.

Résultats Les temps de calcul sont proportionnels au nombre d'itérations appliquées et dépendent relativement peu du nombre de texels de la représentation et du nombre de rayons lumineux utilisés, grâce à l'implémentation sur GPU. Notre implémentation est capable d'effectuer en moyenne un millier d'itérations par seconde. Cependant le nombre d'itérations nécessaires pour converger vers l'optimum recherché dépend fortement du nombre de paramètres de la représentation.

Expérimentalement, environ 50 000 itérations de recuit simulé sont nécessaires pour optimiser le canal d'opacité d'une représentation de 16×16 texels, c'est-à-dire une cinquantaine de secondes. Le réglage des paramètres de la méthode de recuit simulé (température initiale, loi de décroissance de la température, perturbation aléatoire de la solution courante) influe beaucoup sur la vitesse de convergence, il serait intéressant de disposer d'une méthode automatique pour les déterminer. D'autre part lorsque la taille de la représentation augmente (le nombre de texels), le nombre d'itérations nécessaires peut s'avérer très élevé, voire l'algorithme peut se retrouver bloqué dans un minimum local si la température a diminué trop vite. Ceci fait que nous n'avons pu appliquer cette technique que sur des représentations de taille relativement restreintes (16×16 texels).

Cependant le temps de calcul reste raisonnable si l'on ne considère que la composante d'opacité et un nombre de texels pas trop élevé. La figure 8.13 montre la représentation obtenue en fonction du nombre d'itérations.

Les résultats obtenus démontrent le potentiel de cette approche : on peut voir sur la figure 8.14 qu'il est possible avec relativement peu de texels d'approcher la composante d'opacité du light-field de l'objet original, de manière plus exacte qu'en appliquant un seuil après optimisation par moindres carrés avec la méthode précédente.

Nous appliquons ensuite la deuxième partie de l'algorithme proposé, consistant à optimiser les canaux RGB par la technique des moindres carrés, connaissant le canal d'opacité calculé par recuit simulé. Le temps de calcul total pour optimiser 16^2 texels pour un échantillonnage de 256^2 rayons lumineux est de 120 secondes. La

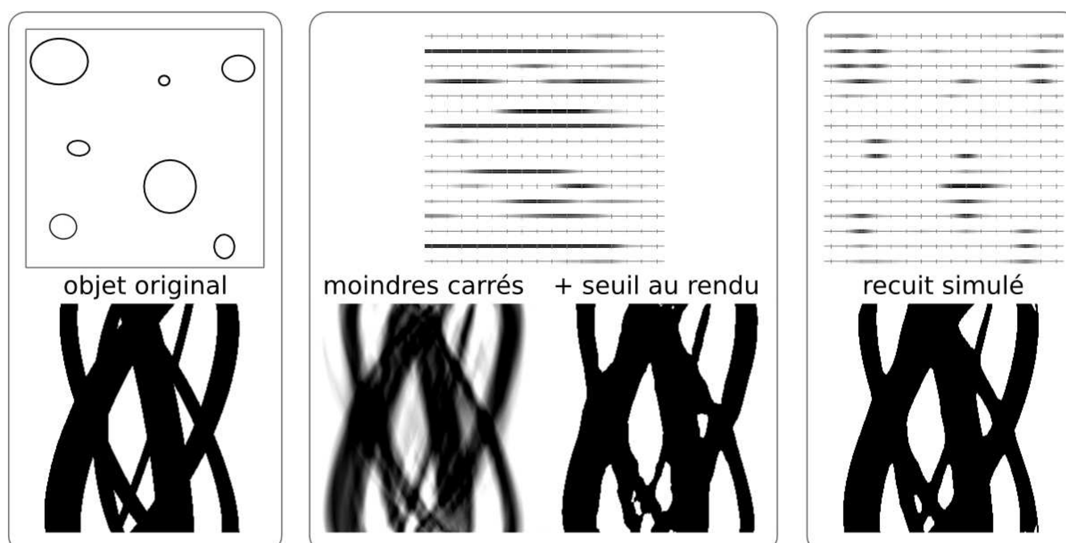


FIGURE 8.14 – Comparaison des méthodes d'optimisations sur la composante d'opacité : à gauche l'objet original et le light-field correspondant; au milieu la représentation optimisée par moindres carrés, son light-field et le light-field obtenu par seuillage (après optimisation); à droite la représentation optimisée par recuit simulé en tenant compte du seuillage lors de l'optimisation.

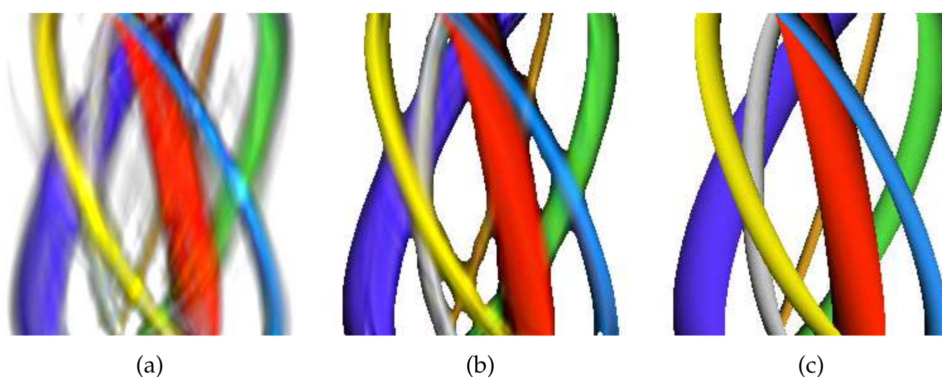


FIGURE 8.15 – Résultat de l'optimisation de 16^2 texels, (a) sans prise en compte de l'opacité, (b) avec l'algorithme complet de prise en compte de l'opacité; (c) le light-field original.

figure 8.15 montre la qualité du résultat obtenu, nettement supérieure à la méthode précédente : les silhouettes externes sont respectées, ce qui est logique puisque elles correspondent au canal d'opacité optimisé par recuit simulé, mais aussi les silhouettes internes, ce qui n'était pas évident *a priori*.

La figure 8.16 montre le résultat de l'algorithme sur un objet 3D complet, représenté par 24 textures de résolution 24×64. Certains problèmes restent aux silhouettes, dûs à un nombre trop faible d'itérations de recuit simulé, ce qui peut aussi se voir dans l'aspect bruité des textures calculées. Cependant le résultat obtenu est globalement bien meilleur que celui produit par l'optimisation sans prise en compte de l'opacité.

Ces résultats valident le principe de l'algorithme d'optimisation en deux temps, optimisant l'opacité d'abord, les couleurs ensuite. Malgré son caractère sous-optimal, cette approche rend le problème réalisable tout en produisant de bons

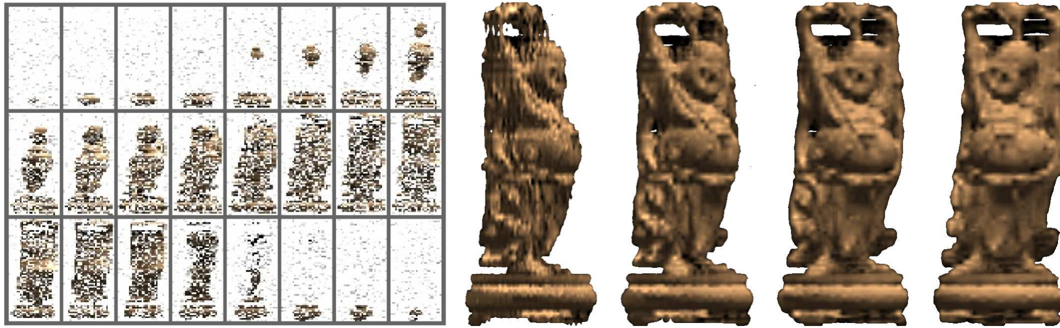


FIGURE 8.16 – Représentation optimisée de 24 textures de résolution 16×64, calculées par l'algorithme avec prise en compte de l'opacité (à gauche), et rendus pour différents angles de vue (à droite).

résultats, c'est donc une voie prometteuse. Il resterait alors à réfléchir à une méthode d'optimisation de l'information d'opacité plus efficace que la technique de recuit simulé, plus adaptée à ce problème en particulier. D'autre part l'utilisation de méthodes d'optimisaton non linéaires comme celle du recuit simulé, rend possible l'optimisation des nombreux paramètres que nous avons supposé fixés, comme la position des plans par exemple.

8.3.4 Bilan

En conclusion, nous venons de montrer comment il est possible de calculer par optimisation un ensemble de plans texturés pour représenter au mieux l'apparence d'un objet. Les résultats obtenus montrent que cette démarche est prometteuse, bien qu'il resterait à étudier des algorithmes d'optimisation plus efficaces pour pouvoir en tirer des applications utiles.

8.4 Possibilités offertes

L'approche générale consistant à optimiser une représentation paramétrique en s'appuyant sur le light-field de l'objet que l'on cherche à représenter offre de nombreuses possibilités. Nous en avons investigué une partie, nous décrivons maintenant celles qui nous semblent les plus prometteuses.

Meilleure représentation La première piste à explorer consisterait à appliquer la méthode présentée sur des représentations plus expressives. Déja sur la représentation à base de plans texturés, il serait intéressant de pouvoir déterminer la position optimale des plans en même temps que leurs textures.

De manière plus générale il serait possible aussi d'utiliser d'autres surfaces que des plans, notamment les surfaces qui peuvent être rendues efficacement par raycasting comme les quadriques ou les champs de hauteur.

Une autre possibilité serait d'affecter à chaque texel une autre information que la simple information de couleurs, par exemple une fonction de couleur dépendant

de l'angle de vue (par exemple par discrétisation de la sphère des directions, puis interpolation entre plusieurs échantillons de couleurs stockés par texel).

Il serait intéressant aussi d'appliquer cette approche sur une représentation volumique, très similaire de la représentation par plans parallèles texturés. Cette représentation a l'intérêt d'être relativement bien adaptée aux problèmes de filtrage (*antialiasing*) et des algorithme de rendu efficaces existent.

Meilleur échantillonnage du light-field La deuxième voie à suivre serait d'étudier les différentes manières d'échantillonner le light-field. L'échantillonnage régulier que nous avons appliqué n'est peut-être pas le plus approprié, et il faudrait pouvoir déterminer le nombre de rayons nécessaire pour l'optimisation d'une représentation donnée.

De plus il est possible de s'affranchir d'un échantillonnage lorsque le light-field peut être exprimé de manière continue : à condition de disposer d'une expression continue du produit scalaire entre light-fields, le problème d'optimisation s'exprime de la même manière.

Ensuite il faudrait pouvoir effectuer l'optimisation de l'ensemble 4D des rayons intersectant l'objet, pour pouvoir ensuite rendre l'objet sous n'importe quel angle de vue, sans la contrainte d'horizontalité. Au vu des temps de calcul dans le cas 2D, il serait nécessaire pour cela de disposer d'un algorithme d'optimisation beaucoup plus efficace que ceux que nous avons présenté.

Enfin l'utilisation de cônes lumineux plutôt que de simples rayons permettrait d'intégrer à la méthode d'optimisation les aspects de filtrage : cela nécessite d'une part d'ajouter une dimension à l'échantillonnage du light-field, l'ouverture des cônes lumineux (variant en fonction de la distance sous lequel l'objet est rendu), d'autre part de disposer d'une représentation paramétrique dont l'intégration par un cône lumineux puisse être exprimée simplement. C'est le cas par exemple d'une texture avec *mip-maps*, pour laquelle l'intersection par un cône lumineux peut être approchée par une simple interpolation trilinéaire. Ainsi si l'on ajoute les niveaux de mip-map aux textures de la représentation que nous avons traitée, il est possible de l'optimiser de sorte qu'au moment du rendu le filtrage (l'antialiasing) soit le meilleur qui soit.

Algorithme d'optimisation adapté Le troisième chemin à poursuivre serait de chercher un algorithme d'optimisation plus efficace, notamment en se servant de propriétés connues de notre problème précis. Par exemple dans le cas des représentations par textures il pourrait être intéressant d'appliquer une approche par raffinement successif, en calculant d'abord des versions à basse résolution puis utiliser cette information comme point de départ pour accélérer le calcul de plus hautes résolutions.

Il est possible aussi d'envisager des représentations paramétiques dont le coût de rendu ou le coût mémoire dépendent des paramètres (par exemple une texture dont la résolution est un paramètre). Prendre en compte cela dans l'optimisation nécessite d'ajouter des pénalités à la fonction de coût que l'on cherche à minimiser. Il faudrait alors dans ce cas là disposer d'un algorithme d'optimisation capable de prendre en compte ces termes de pénalités supplémentaires.

Meilleur critère de distance entre light-fields Enfin la quatrième direction qu'il faudrait envisager porte sur le choix d'un meilleur critère de comparaison entre light-fields. Nous avons utilisé une simple distance euclidienne sur l'espace RGBA, pour les bonnes propriétés que cela représente au moment de l'optimisation. Cependant une telle distance n'est pas correcte du point de vue perceptuel. Des fonctions de distance perceptuelles entre images, basées sur des propriétés connues de la vision humaine existent, il faudrait chercher à les étendre au cas des light-fields. Notamment il semble important de prendre en compte dans une fonction de distance les dérivées partielles (spatiales et directionnelles) du light-field, auquel la perception humaine est plus sensible qu'aux intensités elles-mêmes.

8.5 Conclusion

Nous avons présenté dans ce chapitre un moyen de transformer un échantillonnage dense du light-field sous la forme d'une représentation paramétrique permettant un rendu efficace. L'approche proposée fournit un moyen de savoir ce que l'on peut obtenir comme meilleure reproduction de l'apparence d'un objet avec une représentation paramétrique donnée, ainsi que les paramètres nécessaires.

Chapitre 9

Conclusion

Nous avons présenté dans cette thèse plusieurs stratégies pour représenter efficacement le détail visuel d'une scène virtuelle, d'abord dans le cas du détail géométrique, puis dans le cas de détail non surfacique.

Premièrement, nous avons montré que le détail géométrique peut être représenté efficacement à l'aide de reliefs, encodés dans des cartes de hauteur. La rapidité et la robustesse des algorithmes que nous avons proposé ouvrent l'utilisation du relief à la représentation de détails à l'échelle mésoscopique voire macroscopique, alors que son usage était précédemment généralement restreint à la représentation de micro-détails. Nous avons défini une notion de distance élargie, que nous exploitons avantageusement pour accélérer le rendu de relief tout en maintenant son exactitude, et dont l'utilisation pourrait être étendue à toute autre situation où l'on cherche à effectuer le raycasting d'une surface statique.

Deuxièmement, nous avons étudié le cas des représentations non géométriques, necessaires dans de nombreuses situations où les représentations à base de surfaces sont inadaptées. L'échantillonnage d'un ensemble de vues d'un objet avec sa parallaxe est une représentation à base d'images qui autorise un rendu fidèle et efficace. Nous avons enfin proposé une méthode permettant d'optimiser les paramètres d'une représentation construite de manière à permettre un rendu efficace, l'optimisation étant guidée par la seule donnée de l'apparence de l'objet, son light-field.

De nombreuses pistes de recherche peuvent être envisagées à partir de ces résultats. Les algorithmes de rendu de relief que nous avons proposé peuvent encore être rendus plus efficaces, notamment en étudiant plus en détail les propriétés de la distance élargie pour en déduire une information précalculée encore plus forte. De plus le problème du rendu de relief avec antialiassage est un problème difficile encore ouvert. Reste aussi le problème de la la détermination d'une transformation appliquable aux reliefs, suffisamment flexible pour permettre la segmentation d'une surface animée mais suffisamment contrainte pour permettre un rendu exact et rapide.

Pour la représentation par échantillonnage de vues avec parallaxe, la question du choix optimal de ces vues reste ouverte, et la redondance présente dans un tel ensemble de vues doit pouvoir être encore mieux exploitée. Enfin la méthode du

light-field paramétrique offre de nombreuses possibilités intéressantes, notamment l'utilisation d'une représentation plus flexible permettant la représentation d'objets animés et éclairables dynamiquement, le développement d'algorithmes d'optimisation suffisamment efficaces pour permettre la prise en compte des quatre dimensions du light-field et ainsi lever la contrainte d'horizontalité du point de vue, ou encore la prise en compte dans la procédure d'optimisation du mécanisme de filtrage des textures (mip-mapping) pour obtenir une représentation permettant un rendu filtré de manière optimale en fonction de la taille qu'occupe l'objet à l'écran.

Quatrième partie

Annexes

Annexe A

Shaders des algorithmes de rendu de relief

A.1 Vertex shader appliqué à la boîte englobante du relief

```
1  uniform mat4 W2N;      // repère monde −−(W2N)−−> repère normalisé
2  uniform vec4 worldEye; // position de l'oeil dans le repère monde
3  varying vec4 eye;      // position de l'oeil dans le repère normalisé
4  varying vec3 p0;       // point d'entrée du rayon lumineux dans la boîte, en coordonnées normalisées
5
6  void main() {
7      eye = W2N * worldEye;
8      gl_Position = ftransform();      // les sommets sont exprimés en repère monde
9      p0 = gl_MultiTexCoord0;          // coordonnées du sommet dans le repère normalisé
10     }
```

A.2 Fragment shader pour le rendu sans précalcul

```
1  uniform sampler2D heightMap;   // filtrage = GL_LINEAR
2  uniform sampler2D colorMap;
3  uniform sampler2D normalMap;
4  uniform vec2 resolution;       // résolution de la carte de hauteur
5  varying vec4 eye;              // position de l'oeil dans le repère normalisé
6  varying vec3 p0;               // position d'entrée du rayon lumineux dans la boîte, en coordonnées normalisées
7  uniform vec3 lightDir;
8  uniform mat4 N2W;         // repère normalisé −−(N2W)−−> repère monde
9
10 void main() {
11     const vec2 dPix = 1.0f / resolution;     // les dimensions d'un texel
12     vec3 d = eye.w * v0 − eye.xyz;           // la direction du rayon lumineux
13
14     // on règle la norme de d pour que le rayon lumineux sorte de la boîte englobante en (p0 + d) :
15     const vec3 dir = step(0.0, d),
16               scale = (dir − p0) / d;
17     d *= min(scale.x, min(scale.y, scale.z));
18
19     const vec2 dt = dPix / abs(d.xy),
20               t0 = abs(dir.xy − fract(0.5 + v0.xy * resolution));
21
22     vec4 D = vec4(d, 1),
23          P0 = vec4(p0, 0),
24          dU = D * dt.x,
```

```
25              dV = D * dt.y,
26              U  = P0 + dU * t0.x,
27              V  = P0 + dV * t0.y;
28
29      float zTex = −1;
30      vec4 P = vec4(0, 0, 2, 0);        // (x,y) quelconque, z=+infini, w=0
31
32      while (P.z > zTex && P.w <= 1) {
33          if (U.w < V.w) {
34              P = U;
35              U += dU;
36              }
37          else {
38              P = V;
39              V += dV;
40              }
41          zTex = texture2D(heightMap, P.xy).r;
42          }
43      if (P.z > zTex) discard;
44
45      // calcul du point d'intersection exact :
46      if (U.w − dU.w > V.w − dV.w) U −= dU; else V −= dV;
47      vec4 P1 = (U.w − dU.w > V.w − dV.w) ? U − dU : V − dV; // avant dernier point
48      vec4 P2 = P;    // dernier point
49      float zTex1 = texture2D(heightMap, P1.xy).r,
50            dz2 = zTex  − P2.z,
51            dz1 = zTex1 − P1.z;
52      vec3 pix = mix(P1.xyz, P2.xyz, dz1 / (dz1 − dz2));
53
54      // calcul de l'éclairement :
55      vec3 normal = 2 * texture2D(normalMap, pix).xyz − 1;
56      float diffuseCoef = max(0.0, dot( lightDir , normal));
57      vec4 color = texture2D(colorMap, pix);
58      if (color.a == 0) discard;
59      gl_FragColor = vec4(diffuseCoef * color.rgb, color.a);
60
61      // calcul du z effectif :
62      vec4 pixEye = gl_ModelViewProjectionMatrix * N2W * vec4(pix, 1);
63      gl_FragDepth = (1 + pixEye.z / pixEye.w) / 2;
64      }
```

A.3 Fragment shader pour le rendu utilisant le précalcul du rayon de sûreté

```
1   uniform sampler2D heightMap;   // filtrage : GL_LINEAR
2   uniform sampler2D radiusMap;   // filtrage : GL_NEAREST
3   uniform sampler2D colorMap;
4   uniform sampler2D normalMap;
5   uniform vec2 resolution;
6   varying vec4 eye;
7   varying vec3 p0;
8   uniform vec3 lightDir;
9
10  const int binary_search_steps = 10;
11
12  void main() {
13      const vec2 dPix = 1.0f / resolution;     // les dimensions d'un texel
14      vec3 d = eye.w * p0 − eye.xyz;           // la direction du rayon lumineux
15
16      // on règle la norme de d pour que le rayon lumineux sorte de la boîte englobante en (p0 + d) :
17      const vec3 dir = step(0.0, d),
18                 scale = (dir − p0) / d;
19      d *= min(scale.x, min(scale.y, scale.z));
20
```

```
21      const vec2 dt = dPix / abs(d.xy),
22                 t0 = abs(dir.xy − fract(0.5 + p0.xy * resolution));
23
24      const vec4 D  = vec4(d, 1),          // (x,y,z,t)
25                 P0 = vec4(p0, 0),
26                 dP = D * min(dt.x, dt.y);
27      vec4 P = P0 + dP * ((dt.x < dt.y) ? t0.x : t0.y);
28
29      float zTex = texture2D(heightMap, P.xy).r;
30      vec4 dPnew = dP;
31
32      // parcours par sauts de rayons variables :
33      while (P.z > zTex && P.w <= 1.0) {
34          dPnew = texture2D(radiusMap, P.xy).r * dP;
35          P += dPnew;
36          zTex = texture2D(heightMap, P.xy).r;
37          }
38      if (P.z > zTex) discard;
39
40      // recherche binaire :
41      vec4 deltaP = dPnew;
42      for (int i=0; i<binary_search_steps; i++) {
43          float zTex = texture2D(heightMap, P.xy).r;
44          if (P.z <= zTex) P −= deltaP;
45          // invariant : P est au dessus du champ de hauteurs
46          deltaP *= 0.5;
47          P += deltaP;
48          }
49
50      vec2 pix = P.xy;
51      if (any(lessThan(pix, 0.0)) || any(greaterThan(pix, 1.0))) discard;
52
53      // calcul de l'éclairement :
54      vec3 normal = 2 * texture2D(normalMap, pix).xyz − 1;
55      float diffuseCoef = max(0.0, dot(lightDir, normal));
56      vec4 color = texture2D(colorMap, pix);
57      if (color.a == 0) discard;
58      gl_FragColor = vec4(diffuseCoef * color.rgb, color.a);
59      }
```

Annexe B

Algorithme de précalcul du rayon de sûreté

```
1  class Vec2 {
2      public:
3          float x, y;
4
5          Vec2(float x=0.0f, float y=0.0f) : x(x), y(y) {}
6
7          friend float vecz(const Vec2 &a, const Vec2 &b) { return a.x * b.y − a.y * b.x; }
8
9          friend Vec2 mix(const Vec2 &a, const Vec2 &b, float alpha) {
10             return Vec2(
11                 (1 − alpha) * a.x + alpha * b.x),
12                 (1 − alpha) * a.y + alpha * b.y)
13                 );
14             }
15     };
```

```
1  class Image2DFloat {
2      public:
3          Image2DFloat(int w, int h) :
4              w(w), h(h),
5              data(new float[w * h]) {}
6
7          int width()   const { return w; }
8          int height()  const { return h; }
9          bool contains(int i, int j)      const { return (i >= 0) && (i < w) && (j >= 0) && (j < h); }
10         bool contains(float x, float y)  const { return (x >= 0) && (x < w) && (y >= 0) && (y < h); }
11
12         const float texel(int i, int j) const { return data[i + j*w]; }    // précondition : this−>contains(i,j)
13         float& operator()(int i, int j)       { return data[i + j*w]; }    // précondition : this−>contains(i,j)
14
15         // (i,j) quelconque, échantillonnage défini par <wrapMode>
16         float sample(int i, int j) const {
17             switch (wrapMode) {
18                 case CLAMP_TO_BORDER : return contains(i, j) ? texel(i,j) : borderColor;
19                 case REPEAT :          return texel(modulo(i, w), modulo(j, h));
20                 case MIRRORED_REPEAT : return texel(mirror(i,w), mirror(j,h));
21                 case CLAMP_TO_EDGE :
22                 default :              return texel(clamp(i, 0, w−1), clamp(j, 0, h−1));
23                 };
24             }
25
26         // interpolation bilinéaire : image mappée sur [0,w[*[0,h[
27         float interp(float x, float y) const {
28             int i0 = (int)( floorf (x − 0.5)),
29                 j0 = (int)( floorf (y − 0.5));
30             float a = x − 0.5 − i0,
31                   b = y − 0.5 − j0;
32             return (1−a) * (1−b) * sample(i0, j0   ) + a * (1−b) * sample(i0+1, j0   )
33                  + (1−a) *    b  * sample(i0, j0+1) + a *    b  * sample(i0+1, j0+1);
34             }
35
36         enum WrapMode { CLAMP_TO_EDGE, CLAMP_TO_BORDER, REPEAT, MIRRORED_REPEAT };
37
38     private:
39         int w, h;
40         float *data;
41         WrapMode wrapMode;
42         float borderColor;
43     };
```

```
class MapCoords {
    public:
        MapCoords(const Image2DFloat& map) : map(map) {}
        virtual ~MapCoords() {}

        void setCenter(int i, int j) {
            i0 = i;
            j0 = j;
            x0 = i0 + 0.5;
            y0 = j0 + 0.5;
            }

        virtual float texel(int u, int v)      const = 0;
        virtual float sample(float u, float v) const = 0;
        virtual Vec2 point(float u, float v)   const = 0;

    protected:
        const Image2DFloat& map;
        int   i0, j0;  // coordonnées du texel origine
        float x0, y0;  // centre du texel origine
    };

#define DEF_MAP_COORDS(name, dx, dy, imin, jmin, imax, jmax) \
    class name : public MapCoords { \
        public: \
            name(const Image2DFloat &map) : MapCoords(map) {}                          \
            virtual float texel(int u, int v)      const { return map.sample(i0+dx, j0+dy).r; }  \
            virtual float sample(float u, float v) const { return map.interp(x0+dx, y0+dy).r; } \
            virtual Vec2 point(float u, float v)   const { return Vec2(x0+dx, y0+dy); }         \
        };

DEF_MAP_COORDS(MapXPos, +u, +v);
DEF_MAP_COORDS(MapXNeg, -u, -v);
DEF_MAP_COORDS(MapYPos, +v, +u);
DEF_MAP_COORDS(MapYNeg, -v, -u);
```

```
class VizTexel {
    public:
        MapCoords *map;
        Vec2 p0, p1, p2;
        float epsilon;

    VizTexel(MapCoords *map, float epsilon) : map(map), epsilon(epsilon) {}

    void setCenter(int i, int j) {
        map->setCenter(i,j);
        p0 = slicePoint (0, -0.5);
        p1 = slicePoint (0,  0.0);
        p2 = slicePoint (0,  0.5);
        }

    bool testViz(float u, float alpha, Vec2 q1, Vec2 q2) const {
        return testTetra(u, alpha, p0, p1, q1, q2)
            && testTetra(u, alpha, p1, p2, q1, q2);
        }

    bool testTetra(float u, float alpha, Vec2 a1, Vec2 a2, Vec2 b1, Vec2 b2) const {
        const Vec2 q1 = mix(a1, b1, alpha);
        const Vec2 q2 = vecz(a2-a1, b2-b1) > 0 ? mix(a2, b1, alpha) : mix(a1, b2, alpha);
        const Vec2 q3 = mix(a2, b2, alpha);
        return testSegment(u, q1, q2) && testSegment(u, q2, q3);
        }

    bool testSegment(float u, Vec2 a, Vec2 b) const {          // composantes : a = (va, za), b = (vb, zb)
        bool res = testPoint(u, a.x, a.y) && testPoint(u, b.x, b.y);
        if (res && floorf(a.x) != floorf(b.x)) {
            const float
                v = fmaxf(floorf(a.x), floorf(b.x)),
                c = (v - a.x) / (b.x - a.x),
                z = (1-c) * a.y + c * b.y;
            res = res && testPoint(u, v, z);
            }
        return res;
        }

    bool testPoint(float u, float v, float z) const {
        return (map->sample(u, v) <= z + epsilon);
        }

    Vec2 slicePoint(float u, float v) const {
        return Vec2(v, map->sample(u,v));
        }
    };
```

```
class Precomputation {
    public:
        Image2DFloat heightMap;   // la carte de hauteur
        float epsilon;            // erreur autorisée
        int rMax;                 // rayon de sûreté maximum recherché

```

```
7     Precomputation(Image2DFloat heightMap,
8                 float epsilon = 0.0f,
9                 int rMax = max(heightMap.width(), heigthMap.height()))
10      : heightMap(heightMap), epsilon(epsilon), rMax(rMax) {}
11
12    Image2DFloat computeRadiusMap() {
13      MapCoords* maps[4];
14      maps[0] = new MapXPos(heightMap);
15      maps[1] = new MapXNeg(heightMap);
16      maps[2] = new MapYPos(heightMap);
17      maps[3] = new MapYNeg(heightMap);
18      VizTexel* viz [4];
19      for (int d=0; d<4; d++)
20        viz = new VizTexel(maps[d], epsilon);
21
22      const int w = heightMap.width();
23      const int h = heightMap.height();
24
25      Image2DFloat radiusMap(w,h);
26
27      for (int i=0; i<w; i++)
28        for (int j=0; j<h; j++) {
29          for (int d=0; d<4; d++) viz[d]->setCenter(i, j);
30
31          int r = 1;
32          bool ok = true;
33          while (ok && r < rMax) {
34            const float a = float(r) / float(r + 1);
35
36            const int kMin = -r - 2;
37            const int kMax = r + 1;
38            for (int k=kMin; ok && k<=kMax; k++) {
39              const float d0 = k + (k==kMin ? 0.5 : 0);
40              const float d1 = k + (k==kMax ? 0.5 : 1);
41
42              for (int d=0; ok && d<4; d++) {
43                const Vec2 q1 = viz[d]->slicePoint(r+1, d0);
44                const Vec2 q2 = viz[d]->slicePoint(r+1, d1);
45                ok = ok && viz[d]->testViz(r, a, q1, q2);
46                }
47              }
48
49            if (ok) r++;
50            }
51
52          radiusMap(i,j) = r;
53          }
54
55      for (int d=0; d<4; d++) {
56        delete maps[d];
57        delete viz[d];
58        }
59
60      return radiusMap;
61      }
62    }
```

Bibliographie

[AB91] Edward H. Adelson et James R. Bergen. The Plenoptic Function and the Elements of Early Vision. Dans Michael S. Landy et Anthony J. Movshon, éditeurs, *Computational Models of Visual Processing*, p. 3-0. MIT Press, Cambridge, MA, 1991. (Cité p. 140)

[ABB+07] C. Andújar, J. Boo, P. Brunet, M. Fairén, I. Navazo, P. Vázquez et À. Vinacua. « Omnidirectional Relief Impostors ». *Computer Graphics Forum*, 26(3):553-60, septembre 2007. (Cité p. 116 et 153)

[AK89] James Arvo et David Kirk. « *A survey of ray tracing acceleration techniques* », p. 201-62. Academic Press Ltd., London, UK, UK, 1989. (Cité p. 37)

[App68] Arthur Appel. « Some techniques for shading machine renderings of solids ». Dans *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, p. 37-5, New York, NY, USA, 1968. ACM. <http://doi.acm.org/10.1145/1468075.1468082>. (Cité p. 36)

[Arv86] James Arvo. « Backward Ray Tracing ». Dans *Developments in Ray Tracing (SIGGRAPH '86 Course Notes)*, août 1986. (Cité p. 25, 123 et 129)

[AW87] John Amanatides et Andrew Woo. « A Fast Voxel Traversal Algorithm for Ray Tracing ». Dans *Eurographics '87*, p. 3-0, 1987. (Cité p. 79)

[Bab05] Lionel Baboud. « Représentations Alternatives pour l'Affichage d'Objets Lointains ». Master's thesis, Artis/GRAVIR/IMAG-INRIA, 2005. (Cité p. 151, 154 et 155)

[BD06] Lionel Baboud et Xavier Décoret. « Rendering Geometry with Relief Textures ». Dans *Graphics Interface '06*, 2006. <http://artis.imag.fr/Publications/2006/BD06>. (Cité p. 87, 91 et 113)

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston et Pat Hanrahan. « Brook for GPUs: stream computing on graphics hardware ». *ACM Trans. Graph.*, 23(3):777-86, 2004. <http://doi.acm.org/10.1145/1015706.1015800>. (Cité p. 36)

[BHGS06] Tamy Boubekeur, Wolfgang Heidrich, Xavier Granier et Christophe Schlick. « Volume-Surface Trees ». *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2006)*, 25(3):399-06, 2006. <http://iparla.labri.fr/publications/2006/BHGS06>. (Cité p. 114 et 115)

[Bli78] James F. Blinn. « Simulation of wrinkled surfaces ». *SIGGRAPH Comput. Graph.*, 12(3):286-92, 1978. <http://doi.acm.org/10.1145/965139.507101>. (Cité p. 64)

[BN76] James F. Blinn et Martin E. Newell. « Texture and reflection in computer generated images ». *Commun. ACM*, 19(10):542-47, 1976. <http://doi.acm.org/10.1145/360349.360353>. (Cité p. 30 et 122)

[BNM+08] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton et Cyril Crassin. « Interactive multiple anisotropic scattering in clouds ». Dans *ACM Symposium on Interactive 3D Graphics and Games (I3D)*, 2008. <http://www-evasion.imag.fr/Publications/2008/BNMBC08>. (Cité p. 49)

[Bre73] Richard P. Brent. *Algorithms for minimization without derivatives*. Dover Publications, Englewood Cliffs, N.J, 1973. (Cité p. 69)

[BSK04] Mario Botsch, Michael Spernat et Leif Kobbelt. « Phong Splatting ». Dans *Symposium on Point - Based Graphics 2004*, juin 2004. (Cité p. 27, 28 et 38)

[BW03] Jiří Bittner et Peter Wonka. « Visibility in Computer Graphics ». Rapport Technique TR-186-2-03-03, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, mars 2003. human contact: technical-report@cg.tuwien.ac.at. (Cité p. 29)

[CBCG02] Wei-Chao Chen, Jean-Yves Bouguet, Michael H. Chu et Radek Grzeszczuk. « Light field mapping: efficient representation and hardware rendering of surface light fields ». Dans *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, p. 447-56, New York, NY, USA, 2002. ACM. <http://doi.acm.org/10.1145/566570.566601>. (Cité p. 162)

[CHCH06] Nathan A. Carr, Jared Hoberock, Keenan Crane et John C. Hart. « Fast GPU ray tracing of dynamic meshes using geometry images ». Dans *GI '06: Proceedings of Graphics Interface 2006*, p. 203-09, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society. (Cité p. 37)

[CHH02] Nathan A. Carr, Jesse D. Hall et John C. Hart. « The ray engine ». Dans *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, p. 37-6, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. (Cité p. 37)

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre et Elmar Eisemann. « GigaVoxels : Ray-Guided Streaming for Efficient and Detailed Voxel Rendering ». Dans *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press. <http://artis.imag.fr/Publications/2009/CNLE09>, to appear. (Cité p. 38)

[Com92] Philippe Comar. « *La perspective en jeu* », Chapitre Des yeux en plus, p. 65-78. Gallimard, 1992. (Cité p. 49 et 50)

[Coo84] Robert L. Cook. « Shade trees ». Dans *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, p. 223-31, New York, NY, USA, 1984. ACM. <http://doi.acm.org/10.1145/800031.808602>. (Cité p. 64)

[CPD07] Jiawen Chen, Sylvain Paris et Frédo Durand. « Real-time edge-aware image processing with the bilateral grid ». Dans *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 103, New York, NY, USA, 2007. ACM. <http://doi.acm.org/10.1145/1275808.1276506>. (Cité p. 129)

[CSAD04] David Cohen-Steiner, Pierre Alliez et Mathieu Desbrun. « Variational shape approximation ». Dans *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, p. 905-14, New York, NY, USA, 2004. ACM. <http://doi.acm.org/10.1145/1186562.1015817>. (Cité p. 114)

[CTCS00] Jin-Xiang Chai, Xin Tong, Shing-Chow Chan et Heung-Yeung Shum. « Plenoptic sampling ». Dans *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, p. 307-18, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. <http://doi.acm.org/10.1145/344779.344932>. (Cité p. 141)

[CW93] Shenchang Eric Chen et Lance Williams. « View interpolation for image synthesis ». Dans *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, p. 279-88, New York, NY, USA, 1993. ACM. <http://doi.acm.org/10.1145/166117.166153>. (Cité p. 141)

[CZRG06] Chuo-Ling Chang, Xiaoqing Zhu, P. Ramanathan et B. Girod. « Light field compression using disparity-compensated lifting and shape adaptation ». *Image Processing, IEEE Transactions on*, 15(4):793-806, April 2006. <10.1109/TIP.2005.863954>. (Cité p. 161)

[Dav06] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006. (Cité p. 169)

[DCSD02] Oliver Deussen, Carsten Colditz, Marc Stamminger et George Drettakis. « Interactive visualization of complex plant ecosystems ». Dans *VIS '02: Proceedings of the conference on Visualization '02*, p. 219-26, Washington, DC, USA, 2002. IEEE Computer Society. (Cité p. 27)

[DD02] Frédo Durand et Julie Dorsey. « Fast bilateral filtering for the display of high-dynamic-range images ». *ACM Transactions on Graphics (Proc. of SIGGRAPH 2002)*, 21(3):257-66, juillet 2002. (Cité p. 129)

[DDSD03] Xavier Décoret, Frédo Durand, François Sillion et Julie Dorsey. « Billboard Clouds for Extreme Model Simplification ». Dans *Proceedings of the ACM Siggraph*. ACM Press, 2003. <http://artis.imag.fr/Publications/2003/DDSD03>. (Cité p. 158, 162 et 163)

[DN04] Philippe Decaudin et Fabrice Neyret. « Rendering Forest Scenes in Real-Time ». Dans *Rendering Techniques (Eurographics Symposium on Rendering - EGSR)*, p. 93-02, june 2004. <http://www-evasion.imag.fr/Publications/2004/DN04>. (Cité p. 165)

[Don05] William Donnelly. « *GPU Gems 2* », Chapitre Per-Pixel Displacement Mapping with Distance Functions, p. 123-36. Addison-Wesley, 2005. (Cité p. 64, 72 et 84)

[DPÉTS03] Johann Dréo, Alain Petrowski, Éric Taillard et Patrick Siarry. *Métaheuristiques pour l'optimisation difficile*. Eyrolles, 2003. (Cité p. 179)

[DS06] Carsten Dachsbacher et Marc Stamminger. « Splatting indirect illumination ». Dans *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, p. 93-00, New York, NY, USA, 2006. ACM Press. <http://doi.acm.org/10.1145/1111411.1111428>. (Cité p. 122)

[DSDD02] Xavier Décoret, François Sillion, Frédo Durand et Julie Dorsey. « Billboard Clouds ». Rapport Technique RR-4485, INRIA, Grenoble, June 2002. (Cité p. 158, 162 et 163)

[dTL04] Rodrigo de Toledo et Bruno Levy. « Extending the graphic pipeline with new GPU-accelerated primitives ». Dans *International gOcad Meeting, Nancy, France*, 2004. <http://www.tecgraf.puc-rio.br/~rtoledo/publications>, Also presented in Visgraf Seminar 2004, IMPA, Rio de Janeiro, Brazil. (Cité p. 38)

[dTL08] Rodrigo de Toledo et Bruno Lévy. « Visualization of Industrial Structures with Implicit GPU Primitives ». Dans *ISVC, International Symposium on Visual Computing - Lecture Notes in Computer Science, to appear*, 2008. (Cité p. 38)

[dTLP07] Rodrigo de Toledo, Bruno Levy et Jean-Claude Paul. « Iterative Methods for Visualization of Implicit Surfaces on GPU ». Dans *ISVC, International Symposium on Visual Computing*, Lecture Notes in Computer Science, p. 598-609, Lake Tahoe, Nevada/California, November 2007. Springer. (Cité p. 38)

[DTM96] Paul E. Debevec, Camillo J. Taylor et Jitendra Malik. « Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach ». Dans *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, p. 11-0, New York, NY, USA, 1996. ACM. <http://doi.acm.org/10.1145/237170.237191>. (Cité p. 141)

[dTWL07] Rodrigo de Toledo, Bin Wang et Bruno Levy. « Geometry Textures ». Dans *SIBGRAPI '07: Proceedings of the XX Brazilian Symposium on Computer Graphics and Image Processing*, p. 79-6, Washington, DC, USA, 2007. IEEE Computer Society. <http://dx.doi.org/10.1109/SIBGRAPI.2007.20>. (Cité p. 64, 114, 115 et 117)

[dTWL08] Rodrigo de Toledo, Bin Wang et Bruno Lévy. « Geometry Textures and Applications ». *Computer Graphics Forum*, 27(8):2053-065, décembre 2008. (Cité p. 64, 72, 114, 115 et 117)

[DW07] Scott T Davis et Chris Wyman. « Interactive refractions with total internal reflection ». Dans *GI '07: Proceedings of Graphics Interface 2007*, p. 185-90, New York, NY, USA, 2007. ACM. <http://doi.acm.org/10.1145/1268517.1268548>. (Cité p. 122)

[DWS+97] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich et M.B. Mineev-Weinstein. « ROAMing terrain: Real-time Optimally Adapting Meshes ». *Visualization '97., Proceedings*, 0:81-88, Oct. 1997. <10.1109/VISUAL.1997.663860>. (Cité p. 64)

[EAMJ05] Manfred Ernst, Tomas Akenine-Möller et Henrik Wann Jensen. « Interactive rendering of caustics using interpolated warped volumes ». Dans *Graphics Interface 2005*, p. 87-6, 2005. (Cité p. 122)

[EE95] Bruno Ernst et M.C. Escher. *The Magic Mirror of M.C. Escher*. Taschen, Köln, 1995. (Cité p. 51)

[EMF02] Douglas Enright, Stephen Marschner et Ronald Fedkiw. « Animation and rendering of complex water surfaces ». *ACM Transactions on Graphics (Proc. of SIGGRAPH 2002)*, 21(3):736-44, juillet 2002. (Cité p. 121)

[FB68] Albert Flocon et André Barre. *La perspective curviligne*. Flammarion, Paris, 1968. (Cité p. 49 et 51)

[FF01] Nick Foster et Ronald Fedkiw. « Practical animation of liquids ». Dans *SIGGRAPH 2001*, p. 23-0, 2001. <http://doi.acm.org/10.1145/383259.383261>. (Cité p. 121)

[FR86] Alain Fournier et William T. Reeves. « A simple model of ocean waves ». *Computer Graphics (Proc. of SIGGRAPH '86)*, 20(4):75-4, août 1986. (Cité p. 121)

[FS05] Tim Foley et Jeremy Sugerman. « KD-tree acceleration structures for a GPU raytracer ». Dans *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, p. 15-2, New York, NY, USA, 2005. ACM. <http://doi.acm.org/10.1145/1071866.1071869>. (Cité p. 37)

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner et John F. Hughes. « *Computer Graphics: Principles and Practice* », Chapitre Visible-Surface Determination, p. 649-20. Addison-Wesley, 2nd édition, 1990. (Cité p. 25)

[GD98] J. P. Grossman et William J. Dally. « Point sample rendering ». Dans *In Rendering Techniques 98*, p. 181-92. Springer, 1998. (Cité p. 27)

[GGH02] Xianfeng Gu, Steven J. Gortler et Hugues Hoppe. « Geometry images ». *ACM Trans. Graph.*, 21(3):355-61, 2002. <http://doi.acm.org/10.1145/566654.566589>. (Cité p. 154)

[GGSC96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski et Michael F. Cohen. « The lumigraph ». Dans *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, p. 43-4, New York, NY, USA, 1996. ACM. <http://doi.acm.org/10.1145/237170.237200>. (Cité p. 141 et 162)

[GH97] Michael Garland et Paul S. Heckbert. « Surface simplification using quadric error metrics ». Dans *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, p. 209-16, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. <http://doi.acm.org/10.1145/258734.258849>. (Cité p. 29)

[GH99] Stefan Gumhold et Tobias Hüttner. « Multiresolution rendering with displacement mapping ». Dans *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, p. 55-6, New York, NY, USA, 1999. ACM. <http://doi.acm.org/10.1145/311534.311578>. (Cité p. 64)

[GHFP08] Jean-Dominique Gascuel, Nicolas Holzschuch, Gabriel Fournier et Bernard Peroche. « Fast Non-Linear Projections using Graphics Hardware ». Dans *ACM Symposium on Interactive 3D Graphics and Games*, feb 2008. <http://artis.imag.fr/Publications/2008/GHFP08>. (Cité p. 51)

[GI83] D. Goldfarb et A. Idnani. « A numerically stable dual method for solving strictly convex quadratic programs ». *Mathematical Programming*, 27:1-3, 1983. (Cité p. 169)

[GLD06] Olivier Génevaux, Frédéric Larue et Jean-Michel Dischler. « Interactive refraction on complex static geometry using spherical harmonics ». Dans *Symposium on Interactive 3D Graphics and Games*, p. 145-52, 2006. <http://doi.acm.org/10.1145/1111411.1111438>. (Cité p. 122)

[GM02] Manuel N. Gamito et F. Kenton Musgrave. « An accurate model of wave refraction over shallow water. ». *Computers & Graphics*, 26(2):291-307, 2002. <http://dx.doi.org/10.1016/S0097-8493(01)00181-9>. (Cité p. 121)

[GMIG08] Enrico Gobbetti, Fabio Marton et José Antonio Iglesias Guitián. « A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets ». *Vis. Comput.*, 24(7):797-06, 2008. <http://dx.doi.org/10.1007/s00371-008-0261-9>. (Cité p. 38)

[Gom00] Miguel Gomez. « *Game Programming Gems* », Chapitre Interactive simulation of water surfaces, p. 187-94. Charles River Media, Inc., 2000. (Cité p. 121 et 122)

[GPSS07] Johannes Günther, Stefan Popov, Hans-Peter Seidel et Philipp Slusallek. « Realtime Ray Tracing on GPU with BVH-based Packet Traversal ». Dans Alexander Keller et Per Christensen, éditeurs, *IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, p. 113-18, Ulm, Germany, 2007. IEEE, Eurographics, IEEE. (Cité p. 37)

[Gre86] Ned Greene. « Environment mapping and other applications of world projections ». *IEEE Comput. Graph. Appl.*, 6(11):21-9, 1986. <http://dx.doi.org/10.1109/MCG.1986.276658>. (Cité p. 30 et 122)

[GSSK05] Ismael Garcia, Mateu Sbert et Laszlo Szirmay-Kalos. « Leaf Cluster Impostors for Tree Rendering with Parallax ». Dans *In Proceedings of EG Short Presentations 2005*, p. 69-2, 2005. (Cité p. 163)

[Gum03] Stefan Gumhold. « Splatting illuminated ellipsoids with depth correction ». Dans *Proceedings of 8th International Fall Workshop on Vision, Modelling and Visualization 2003*, p. 245-52, nov 2003. (Cité p. 38)

[GVL96] Gene H. Golub et Charles F. Van Loan. « *Matrix Computations* », Chapitre 5.2.1 Householder QR, p. 224-25. Johns Hopkins University Press, 3rd édition, 1996. (Cité p. 169)

[Har96] John C. Hart. « Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces ». *The Visual Computer*, 12:527-45, 1996. (Cité p. 73 et 83)

[HDE80] D.E. Hasselmann, M. Dunckel et J.A. Ewing. « Directional Wave Spectra Observed during JONSWAP 1973 ». *Journal of Physical Oceanography*, 10:1264-280, août 1980. (Cité p. 121)

[HEGD04] Johannes Hirche, Alexander Ehlert, Stefan Guthe et Michael Doggett. « Hardware accelerated per-pixel displacement mapping ». Dans *GI '04: Proceedings of Graphics Interface 2004*, p. 153-58, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society. (Cité p. 64 et 72)

[Hen07] Justin Hensley. « AMD CTM overview ». Dans *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 7, New York, NY, USA, 2007. ACM. <http://doi.acm.org/10.1145/1281500.1281648>. (Cité p. 36)

[HLCS99] Wolfgang Heidrich, Hendrik Lensch, Michael F. Cohen et Hans-Peter Seidel. « Light Field Techniques for Reflections and Refractions ». Dans *Rendering Techniques '99 (Proc. EG Workshop on Rendering)*, p. 187-96, juin 1999. (Cité p. 122)

[HNC02] Damien Hinsinger, Fabrice Neyret et Marie-Paule Cani. « Interactive animation of ocean waves ». Dans *Symposium on Computer Animation*, p. 161-66, 2002. <http://doi.acm.org/10.1145/545261.545288>. (Cité p. 121)

[Hop96] Hugues Hoppe. « Progressive meshes ». Dans *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, p. 99-08, New York, NY, USA, 1996. ACM. <http://doi.acm.org/10.1145/237170.237216>. (Cité p. 29)

[Hop99] Hugues Hoppe. « New quadric metric for simplifying meshes with appearance attributes ». Dans *VISUALIZATION '99: Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99)*, Washington, DC, USA, 1999. IEEE Computer Society. (Cité p. 162)

[HPP00] Vlastimil Havran, Jan Prikryl et Werner Purgathofer. « Statistical Comparison of Ray-Shooting Efficiency Schemes ». Rapport Technique TR-186-2-00-14, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, mai 2000. human contact: technical-report@cg.tuwien.ac.at. (Cité p. 37)

[HQ07] Wei Hu et Kaihuai Qin. « Interactive Approximate Rendering of Reflections, Refractions, and Caustics ». *IEEE Transactions on Visualization and Computer Graphics*, 13(1):46-7, 2007. <http://dx.doi.org/10.1109/TVCG.2007.14>. (Cité p. 122)

[HS98] Wolfgang Heidrich et Hans-Peter Seidel. « View-independent environment maps ». Dans *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, p. 39-f., New York, NY, USA, 1998. ACM. <http://doi.acm.org/10.1145/285305.285310>. (Cité p. 50)

[HS04] Christian Henning et Peter Stephenson. « Accelerating the ray tracing of height fields ». Dans *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, p. 254-58, New York, NY, USA, 2004. ACM. <http://doi.acm.org/10.1145/988834.988878>. (Cité p. 64 et 72)

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston et Pat Hanrahan. « Interactive k-d tree GPU raytracing ». Dans *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, p. 167-74, New York, NY, USA, 2007. ACM. <http://doi.acm.org/10.1145/1230100.1230129>. (Cité p. 37)

[HSS+05] Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler et Markus Gross. « Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces ». Dans *Proceedings of Eurographics 2005*, p. pp. 303-312, 2005. <http://www.vrvis.at/publications/PB-VRVis-2005-024>. (Cité p. 38)

[HVT+06] Yaohua Hu, Luiz Velho, Xin Tong, Baining Guo et Harry Shum. « Realistic, Real-Time Rendering of Ocean Waves ». *Computer Animation and Virtual Worlds*, 17(1):59 -67, 2006. <http://dx.doi.org/10.1002/cav.v17:1>, Special Issue on Game Technologies. (Cité p. 122)

[HW98] Andrew J. Hanson et Eric A. Wernert. « Image-based rendering with occlusions via cubist images ». Dans *VIS '98: Proceedings of the conference on Visualization '98*, p. 327-34, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. (Cité p. 107)

[IDN03] Kei Iwasaki, Yoshinori Dobashi et Tomoyuki Nishita. « A volume rendering approach for sea surfaces taking into account second order scattering using scattering maps ». Dans *2003 Workshop on Volume Graphics*, p. 129-36, 2003. <http://doi.acm.org/10.1145/827051.827071>. (Cité p. 122)

[Jen01] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., 2001. (Cité p. 122 et 128)

[JMW07] Stefan Jeschke, Stephan Mantler et Michael Wimmer. « Interactive Smooth and Curved Shell Mapping ». Dans Jan Kautz et Sumanta Pattanaik, éditeurs, *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering)*, p. 351-60. Eurographics, Eurographics Association, 6 2007. <http://www.cg.tuwien.ac.at/research/publications/2007/JESCHKE-2007-ISC/>. (Cité p. 111 et 117)

[Kaj86] James T. Kajiya. « The rendering equation ». Dans *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, p. 143-50, New York, NY, USA, 1986. ACM. <http://doi.acm.org/10.1145/15922.15902>. (Cité p. 36 et 37)

[KB04] Leif Kobbelt et Mario Botsch. « A survey of point-based techniques in computer graphics ». *Computers & Graphics*, 28:801-14, 2004. (Cité p. 27)

[KGV83] S. Kirkpatrick, C. D. Gelatt et M. P. Vecchi. « Optimization by Simulated Annealing ». *Science*, 220:671-80, 1983. (Cité p. 179)

[KHK+08] Aaron Knoll, Younis Hijazi, Andrew Kensler, Mathias Schott, Charles Hansen et Hans Hagen. « Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic ». *Computer Graphics Forum*, 2008. (Cité p. 38)

[KHW+07] Aaron Knoll, Younis Hijazi, Ingo Wald, Charles Hansen et Hans Hagen. « Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic ». Dans *Proceedings of the 2nd IEEE/EG Symposium on Interactive Ray Tracing*, p. 11-8, 2007. (Cité p. 38)

[KM90] Michael Kass et Gavin Miller. « Rapid, stable fluid dynamics for computer graphics ». *Computer Graphics (Proc. of SIGGRAPH '90)*, 24(4):49-7, août 1990. <http://doi.acm.org/10.1145/97879.97884>. (Cité p. 121)

[KOKK06] Takashi Kanai, Yutaka Ohtake, Hiroaki Kawata et Kiwamu Kase. « GPU-based rendering of sparse low-degree implicit surfaces ». Dans *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, p. 165-71, New York, NY, USA, 2006. ACM. <http://doi.acm.org/10.1145/1174429.1174455>. (Cité p. 38)

[KRS05] Andreas Kolb et Christof Rezk-Salama. « Efficient Empty Space Skipping for Per-Pixel Displacement Mapping ». Dans *Proc. Vision, Modeling and Visualization (VMV)*, p. 407-414, 2005. (Cité p. 64 et 73)

[KS00] Kiriakos N. Kutulakos et Steven M. Seitz. « A Theory of Shape by Space Carving ». *Int. J. Comput. Vision*, 38(3):199-18, 2000. <http://dx.doi.org/10.1023/A:1008191222954>. (Cité p. 161)

[KSN08] Yoshihiro Kanamori, Zoltan Szego et Tomoyuki Nishita. « GPU-based Fast Ray Casting for a Large Number of Metaballs ». *Computer Graphics Forum (Proc. of Eurographics 2008)*, 27(3):351-60, 2008. (Cité p. 38)

[KTI+01] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda et Susumu Tachi. « Detailed shape representation with parallax mapping ». Dans *Proceedings of the ICAT 2001*, p. 205-08, 2001. (Cité p. 64)

[KW03] J. Kruger et R. Westermann. « Acceleration Techniques for GPU-based Volume Rendering ». Dans *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society. <http://dx.doi.org/10.1109/VIS.2003.10001>. (Cité p. 38)

[LB06] Charles Loop et Jim Blinn. « Real-time GPU rendering of piecewise algebraic surfaces ». *ACM Trans. Graph.*, 25(3):664-70, 2006. <http://doi.acm.org/10.1145/1141911.1141939>. (Cité p. 38)

[LFY+06] Shengying Li, Zhe Fan, Xiaotian Yin, Klaus Mueller, Arie E. Kaufman et Xianfeng Gu. « Real-time Reflection using Ray Tracing with Geometry Field ». Dans *Eurographics 2006 Proceedings (Short Papers)*, p. 29-2, 2006. (Cité p. 122)

[LGQ+08] D. Brandon Lloyd, Naga K. Govindaraju, Cory Quammen, Steven E. Molnar et Dinesh Manocha. « Logarithmic perspective shadow maps ». *ACM Trans. Graph.*, 27(4):1-2, 2008. <http://doi.acm.org/10.1145/1409625.1409628>. (Cité p. 32)

[LH96] Marc Levoy et Pat Hanrahan. « Light field rendering ». Dans *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, p. 31-2, New York, NY, USA, 1996. ACM. <http://doi.acm.org/10.1145/237170.237199>. (Cité p. 141 et 162)

[LHN05] Sylvain Lefebvre, Samuel Hornus et Fabrice Neyret. « *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation* », Chapitre Octree Textures on the GPU, p. 595-13. Addison Wesley, 2005. (Cité p. 108)

[LP95] J. R. Logie et John W. Patterson. « Inverse Displacement Mapping in the General Case ». *Comput. Graph. Forum*, 14(5):261-273, 1995. (Cité p. 64 et 110)

[LT00] Peter Lindstrom et Greg Turk. « Image-driven simplification ». *ACM Trans. Graph.*, 19(3):204-41, 2000. <http://doi.acm.org/10.1145/353981.353995>. (Cité p. 162)

[LW85] Marc Levoy et Turner Whitted. « The Use of Points as a Display Primitive ». Technical Report 85-022, University of North Carolina, Computer Science Department, University of North Carolina at Chapel Hill, January 1985. (Cité p. 27)

[LWC+02] David Luebke, Benjamin Watson, Jonathan D. Cohen, Martin Reddy et Amitabh Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002. (Cité p. 29)

[MB95] Leonard McMillan et Gary Bishop. « Plenoptic modeling: an image-based rendering system ». Dans *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, p. 39-6, New York, NY, USA, 1995. ACM. <http://doi.acm.org/10.1145/218380.218398>. (Cité p. 65 et 141)

[MMB97] William R. Mark, Leonard McMillan et Gary Bishop. « Post-rendering 3D warping ». Dans *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, p. 7-f., New York, NY, USA, 1997. ACM. <http://doi.acm.org/10.1145/253284.253292>. (Cité p. 141, 144 et 149)

[MPSS05] Ovidio Mallo, Ronald Peikert, Christian Sigg et Filip Sadlo. « Illuminated Lines Revisited ». Dans *IEEE Visualization*, p. 19-26, 2005. (Cité p. 27)

[MRG03] M. Magnor, P. Ramanathan et B. Girod. « Multi-view coding for image-based rendering using 3-D scene geometry ». *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(11):1092-1106, Nov. 2003. <10.1109/TCSVT.2003.817630>. (Cité p. 162)

[MW08] Morgan McGuire et Kyle Whitson. « Indirection Mapping for Quasi-Conformal Relief Mapping ». Dans *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '08)*, February 2008. <http://graphics.cs.williams.edu/papers/IndirectionI3D08/>. (Cité p. 108)

[MWM87] Gary A. Mastin, Peter A. Watterberg et John F. Mareda. « Fourier synthesis of ocean scenes ». *IEEE Computer Graphics & Applications*, 7(3):16-3, 1987. (Cité p. 122)

[NBGS08] John Nickolls, Ian Buck, Michael Garland et Kevin Skadron. « Scalable Parallel Programming with CUDA ». *Queue*, 6(2):40-3, 2008. <http://doi.acm.org/10.1145/1365490.1365500>. (Cité p. 36)

[Ney98] Fabrice Neyret. « Modeling Animating and Rendering Complex Scenes using Volumetric Textures ». *IEEE Transactions on Visualization and Computer Graphics*, 4(1):55-0, janvier–mars 1998. ISSN 1077-2626. (Cité p. 111)

[NJ08] Kyung-Gun Na et Moon-Ryul Jung. « Curved Ray-Casting for Displacement Mapping in the GPU ». Dans *Advances in Multimedia Modeling*, p. 348-357, 2008. (Cité p. 111)

[NN94] Tomoyuki Nishita et Eihachiro Nakamae. « Method of displaying optical effects within water using accumulation buffer ». Dans *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, p. 373-79, New York, NY, USA, 1994. ACM. <http://doi.acm.org/10.1145/192161.192261>. (Cité p. 122)

[OB07] Manuel M. Oliveira et Maicon Brauwers. « Real-time refraction through deformable objects ». Dans *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, p. 89-6, New York, NY, USA, 2007. ACM. <http://doi.acm.org/10.1145/1230100.1230116>. (Cité p. 122)

[OBM00] Manuel M. Oliveira, Gary Bishop et David McAllister. « Relief texture mapping ». Dans *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, p. 359-68, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. <http://doi.acm.org/10.1145/344779.344947>. (Cité p. 65, 66, 116, 141, 144 et 149)

[OKL06] Kyoungsu Oh, Hyunwoo Ki et Cheol-Hi Lee. « Pyramidal displacement mapping: a GPU based artifacts-free ray tracing through an image pyramid ». Dans *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology*, p. 75-2, New York, NY, USA, 2006. ACM. <http://doi.acm.org/10.1145/1180495.1180511>. (Cité p. 65 et 73)

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn et Timothy J. Purcell. « A Survey of General-Purpose Computation on Graphics Hardware ». *Computer Graphics Forum*, 26(1):80-13, 2007. <http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x>. (Cité p. 36)

[OP05] Manuel M. Oliveira et Fabio Policarpo. « An Efficient Representation for Surface Details ». Rapport Technique RP-351, UFRGS, 2005. (Cité p. 64, 72 et 110)

[PA00] Simon PREMOZE et Michael ASHIKHMIN. « Rendering natural waters ». Dans *Pacific Conference on Computer Graphics and Applications*, page 23, 2000. (Cité p. 122 et 126)

[Pag98] David W. PAGLIERONI. « The directional parameter plane transform of a height field ». *ACM Transactions on Graphics*, 17(1):50-0, 1998. <http://doi.acm.org/10.1145/269799.269802>. (Cité p. 73)

[Pan75] Erwin PANOFSKY. *La perspective comme forme symbolique*. Editions de Minuit, Paris, 1975. (Cité p. 50)

[PBFJ05] Serban D. PORUMBESCU, Brian BUDGE, Louis FENG et Kenneth I. JOY. « Shell maps ». *ACM Trans. Graph.*, 24(3):626-33, 2005. <http://doi.acm.org/10.1145/1073204.1073239>. (Cité p. 111)

[PBMH02] Timothy J. PURCELL, Ian BUCK, William R. MARK et Pat HANRAHAN. « Ray Tracing on Programmable Graphics Hardware ». *ACM Transactions on Graphics*, 21(3):703-12, July 2002. <http://graphics.stanford.edu/papers/rtongfx/>, ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002). (Cité p. 37)

[PD09] Sylvain PARIS et Frédo DURAND. « A Fast Approximation of the Bilateral Filter Using a Signal Processing Approach ». *Int. J. Comput. Vision*, 81(1):24-2, 2009. <http://dx.doi.org/10.1007/s11263-007-0110-8>. (Cité p. 129)

[PDC+03] Timothy J. PURCELL, Craig DONNER, Mike CAMMARANO, Henrik Wann JENSEN et Pat HANRAHAN. « Photon mapping on programmable graphics hardware ». Dans *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, p. 41-0, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. (Cité p. 37)

[Pea86] Darwyn R. PEACHEY. « Modeling waves and surf ». *Computer Graphics (Proc. of SIGGRAPH '86)*, 20(4):65-4, août 1986. <http://doi.acm.org/10.1145/15922.15893>. (Cité p. 121)

[PGSS07] Stefan POPOV, Johannes GÜNTHER, Hans-Peter SEIDEL et Philipp SLUSALLEK. « Stackless KD-Tree Traversal for High Performance GPU Ray Tracing ». *Computer Graphics Forum*, 26(3):415-24, septembre 2007. (Proceedings of Eurographics). (Cité p. 37)

[PHL91] John W. PATTERSON, Stuart G. HOGGAR et J. R. LOGIE. « Inverse Displacement Mapping ». *Comput. Graph. Forum*, 10(2):129-139, 1991. (Cité p. 64 et 110)

[PKZ04] Jianbo PENG, Daniel KRISTJANSSON et Denis ZORIN. « Interactive modeling of topologically complex geometric detail ». *ACM Trans. Graph.*, 23(3):635-43, 2004. <http://doi.acm.org/10.1145/1015706.1015773>. (Cité p. 111)

[PM64] Willard J. PIERSON et Lionel MOSKOWITZ. « A Proposed Spectral Form for Fully Developed Wind Seas Based on the Similarity Theory of S. A. Kitaigorodskii ». *Journal of Geophysical Research*, 69:5181-191, décembre 1964. (Cité p. 121)

[PM90] P. PERONA et J. MALIK. « Scale-Space and Edge Detection Using Anisotropic Diffusion ». *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(7):629-39, July 1990. <http://dx.doi.org/10.1109/34.56205>. (Cité p. 129)

[PMDS06] Voicu POPESCU, Chunhui MEI, Jordan DAUBLE et Elisha SACKS. « Reflected-Scene Impostors for Realistic Reflections at Interactive Rates ». *Computer Graphics Forum (Proceedings of Eurographics 2006)*, 25(3):313-22, 2006. (Cité p. 122)

[PN08] S. PATIDAR et P.J. NARAYANAN. « Ray Casting Deformable Models on the GPU ». Dans *Computer Vision, Graphics & Image Processing, 2008. ICVGIP '08. Sixth Indian Conference on*, p. 481-488, 2008. (Cité p. 37)

[PO06] Fabio POLICARPO et Manuel M. OLIVEIRA. « Relief mapping of non-height-field surface details ». Dans *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, p. 55-2, New York, NY, USA, 2006. ACM. <http://doi.acm.org/10.1145/1111411.1111422>. (Cité p. 64 et 115)

[PO07] Fabio POLICARPO et Manuel M. OLIVEIRA. « *GPU Gems 3* », Chapitre Chapter 18: Relaxed Cone Stepping for Relief Mapping, p. 409-28. Addison-Wesley Professional, 2007. (Cité p. 64, 102 et 103)

[POC05] Fabio Policarpo, Manuel M. Oliveira et Joao L. D. Comba. « Real-time relief mapping on arbitrary polygonal surfaces ». Dans *Symposium on Interactive 3D graphics and games*, p. 155-62, 2005. <http://doi.acm.org/10.1145/1053427.1053453>. (Cité p. 64, 65, 72, 102 et 103)

[PP94] David W. Paglieroni et Sidney M. Petersen. « Height distributional distance transform methods for height field ray tracing ». *ACM Trans. Graph.*, 13(4):376-99, 1994. <http://doi.acm.org/10.1145/195826.197312>. (Cité p. 73)

[PR06] Voicu Popescu et Paul Rosen. « Forward rasterization ». *ACM Trans. Graph.*, 25(2):375-11, 2006. <http://doi.acm.org/10.1145/1138450.1138460>. (Cité p. 28)

[PSS+06] H.-F. Pabst, J.P. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig et B. Froehlich. « Ray Casting of Trimmed NURBS Surfaces on the GPU ». Dans *Interactive Ray Tracing 2006, IEEE Symposium on*, 2006. (Cité p. 38)

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling et Brian P. Flannery. « *Numerical Recipes: The Art of Scientific Computing, Third Edition* », Chapitre Chapter 9: Root Finding and Nonlinear Sets of Equations, p. 442-86. Cambridge University Press, 2007. (Cité p. 67 et 69)

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar et Markus Gross. « Surfels: Surface Elements as Rendering Primitives ». Dans *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, p. 335-42, juillet 2000. (Cité p. 27)

[QQZ+03] Huamin Qu, Feng Qiu, Nan Zhang, Arie Kaufman et Ming Wan. « Ray Tracing Height Fields ». *Computer Graphics International Conference*, 0:202, 2003. <http://doi.ieeecomputersociety.org/10.1109/CGI.2003.1214467>. (Cité p. 64 et 72)

[RAH07] David Roger, Ulf Assarsson et Nicolas Holzschuch. « Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU ». Dans Jan Kautz et Sumanta Pattanaik, éditeurs, *Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering)*, p. 99-10. Eurographics and ACM/SIGGRAPH, the Eurographics Association, jun 2007. <http://artis.imag.fr/Publications/2007/RAH07>. (Cité p. 37)

[RH06] David Roger et Nicolas Holzschuch. « Accurate Specular Reflections in Real-Time ». *Computer Graphics Forum (Proceedings of Eurographics 2006)*, 25(3):293-02, sep 2006. <http://artis.imag.fr/Publications/2006/RH06>. (Cité p. 51 et 122)

[Ris07] Eric Risser. « Rendering 3D volumes using per-pixel displacement mapping ». Dans *Sandbox '07: Proceedings of the 2007 ACM SIGGRAPH symposium on Video games*, p. 81-7, New York, NY, USA, 2007. ACM. <http://doi.acm.org/10.1145/1274940.1274958>. (Cité p. 113)

[RMD04] Alex Reche, Ignacio Martin et George Drettakis. « Volumetric Reconstruction and Interactive Rendering of Trees from Photographs ». *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)*, 23(3):720-27, July 2004. <http://www-sop.inria.fr/reves/Basilic/2004/RMD04>. (Cité p. 162)

[RSP07] Eric Risser, Musawir Shah et Sumanta Pattanaik. « Faster Relief Mapping using the Secant Method ». *Journal of Graphics Tools*, 12(3):17-4, 2007. (Cité p. 64 et 72)

[SCD+06] Steven M. Seitz, Brian Curless, James Diebel, Daniel Scharstein et Richard Szeliski. « A Comparison and Evaluation of Multi-View Stereo Reconstruction Algorithms ». Dans *CVPR '06: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, p. 519-28, Washington, DC, USA, 2006. IEEE Computer Society. <http://dx.doi.org/10.1109/CVPR.2006.19>. (Cité p. 162)

[Sch93] Christophe Schlick. « A Customizable Reflectance Model for Everyday Rendering ». Dans *Fourth Eurographics Workshop on Rendering*, p. 73-4, 1993. (Cité p. 123)

[Sch98] Gernot Schaufler. « Image-based object representation by layered impostors ». Dans *VRST '98: Proceedings of the ACM symposium on Virtual reality software and technology*, p. 99-04, New York, NY, USA, 1998. ACM. <http://doi.acm.org/10.1145/293701.293714>. (Cité p. 107 et 116)

[SD99] Steven M. Seitz et Charles R. Dyer. « Photorealistic Scene Reconstruction by Voxel Coloring ». *Int. J. Computer Vision*, 35(2):151-73, 1999. (Cité p. 161)

[SD02] Marc Stamminger et George Drettakis. « Perspective shadow maps ». Dans *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, p. 557-62, New York, NY, USA, 2002. ACM. <http://doi.acm.org/10.1145/566570.566616>. (Cité p. 32)

[SdTG08] Paulo Santos, Rodrigo de Toledo et Marcelo Gattass. « Solid height-map sets: modeling and visualization ». Dans *SPM '08: Proceedings of the 2008 ACM symposium on Solid and physical modeling*, p. 359-65, New York, NY, USA, 2008. ACM. <http://doi.acm.org/10.1145/1364901.1364953>. (Cité p. 115)

[SG06] M. F. A. Schroders et R. v. Gulik. « Quadtree relief mapping ». Dans *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, p. 61-6, New York, NY, USA, 2006. ACM. <http://doi.acm.org/10.1145/1283900.1283910>. (Cité p. 73)

[SGHS98] Jonathan Shade, Steven Gortler, Li-wei He et Richard Szeliski. « Layered depth images ». Dans *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, p. 231-42, New York, NY, USA, 1998. ACM. <http://doi.acm.org/10.1145/280814.280882>. (Cité p. 141 et 144)

[SGS06] Carsten Stoll, Stefan Gumhold et Hans-Peter Seidel. « Incremental Raycasting of Piecewise Quadratic Surfaces on the GPU ». Dans Ingo Wald et Steven G. Parker, éditeurs, *IEEE Symposium on Interactive Raytracing 2006 Proceedings*, IEEE Symposium on Interactive Raytracing Proceedings, p. 141-50, Salt Lake City, USA, September 2006. IEEE, IEEE. (Cité p. 38)

[SKALP05] László Szirmay-Kalos, BarnabáÁs Aszódi, István Lazányi et Mátyás Premecz. « Approximate Ray-Tracing on the GPU with Distance Impostors ». *Computer Graphics Forum (Proc. of Eurographics 2005)*, 24(3):695-704, 2005. (Cité p. 64, 71, 114 et 122)

[SKP07] Musawir A. Shah, Jaakko Konttinen et Sumanta Pattanaik. « Caustics Mapping: An Image-Space Technique for Real-Time Caustics ». *IEEE Transactions on Visualization and Computer Graphics*, 13(2):272-280, 2007. <http://doi.ieeecomputersociety.org/10.1109/TVCG.2007.32>. (Cité p. 122)

[SKUP+09] László Szirmay-Kalos, Tamás Umenhoffer, Gustavo Patow, László Szécsi et Mateu Sbert. « Specular Effects on the GPU: State of the Art ». *Computer Graphics Forum*, 26(1):1-4, 2009. (Cité p. 122)

[SN07] Jag Mohan Singh et P. J. Narayanan. « Real-time ray-tracing of implicit surfaces on the GPU ». Rapport Technique, International Institute of Information Technology, Hyderabad, july 2007. (Cité p. 38)

[SP99] Gernot Schaufler et Markus Priglinger. « Efficient Displacement Mapping by Image Warping ». Dans *10th EUROGRAPHICS Workshop on Rendering*, p. 175-86, 1999. (Cité p. 65, 66, 113, 141, 144 et 149)

[Sta03] Jos Stam. « Real-time fluid dynamics for games ». Dans *Game Developer Conference*, mars 2003. (Cité p. 121)

[SWBG06] Christian Sigg, Tim Weyrich, Mario Botsch et Markus Gross. « GPU-Based Ray-Casting of Quadratic Surfaces ». Dans *IEEE/Eurographics Symposium on Point-Based Graphics 2006*, p. 59-6, juillet 2006. (Cité p. 38)

[SYGM03] J. Stewart, J. Yu, S. J. Gortler et L. McMillan. « A new reconstruction filter for undersampled light fields ». Dans *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering*, p. 150-56, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. (Cité p. 141)

[Szé06] L. Szécsi. « The Hierarchical Ray Engine ». Dans *WSCG*, 2006. (Cité p. 37)

[TA99] Seth Teller et John Alex. « Immediate-Mode Ray-Casting ». Rapport Technique, MIT Laboratory for Computer Science, Cambridge, MA, USA, 1999. (Cité p. 25)

[Tat06] Natalya Tatarchuk. « Dynamic parallax occlusion mapping with approximate soft shadows ». Dans *Symposium on Interactive 3D graphics and games*, p. 63-9, 2006. <http://doi.acm.org/10.1145/1111411.1111423>. (Cité p. 64, 65 et 72)

[TB87] Pauline Y. Ts'o et Brian A. Barsky. « Modeling and rendering waves: wave-tracing using beta-splines and reflective and refractive texture mapping. ». *ACM Transactions on Graphics*, 6(3):191-14, 1987. <http://doi.acm.org/10.1145/35068.35070>. (Cité p. 121)

[Tes99] Jerry Tessendorf. « Simulating ocean water ». Dans *Siggraph Course Notes*. ACM Press, 1999. (Cité p. 121 et 122)

[TIS08] Art Tevs, Ivo Ihrke et Hans-Peter Seidel. « Maximum Mipmaps for Fast, Accurate, and Scalable Dynamic Height Field Rendering ». Dans *Symposium on Interactive 3D Graphics and Games (i3D'08)*, p. 183-90, 2008. (Cité p. 65, 73, 102 et 103)

[TM98] C. Tomasi et R. Manduchi. « Bilateral Filtering for Gray and Color Images ». Dans *International Conference on Computer Vision*, page 839, 1998. (Cité p. 129)

[WAA+00] Daniel N. Wood, Daniel I. Azuma, Ken Aldinger, Brian Curless, Tom Duchamp, David H. Salesin et Werner Stuetzle. « Surface light fields for 3D photography ». Dans *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, p. 287-96, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. <http://doi.acm.org/10.1145/344779.344925>. (Cité p. 162)

[Wat90] Mark Watt. « Light-water interaction using backward beam tracing ». Dans *SIGGRAPH '90*, p. 377-85, 1990. <http://doi.acm.org/10.1145/97879.97920>. (Cité p. 122)

[WD06] Chris Wyman et Scott Davis. « Interactive image-space techniques for approximating caustics ». Dans *Symposium on Interactive 3D graphics and games*, p. 153-60, 2006. <http://doi.acm.org/10.1145/1111411.1111439>. (Cité p. 122)

[Wel04] Terry Welsh. « Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces ». Rapport Technique, Infiscape Corporation, 2004. (Cité p. 64)

[Whi80] Turner Whitted. « An improved illumination model for shaded display ». *Commun. ACM*, 23(6):343-49, 1980. <http://doi.acm.org/10.1145/358876.358882>. (Cité p. 36 et 37)

[Wil78] Lance Williams. « Casting curved shadows on curved surfaces ». Dans *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, p. 270-74, New York, NY, USA, 1978. ACM Press. <http://doi.acm.org/10.1145/800248.807402>. (Cité p. 52)

[WMG+07] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker et Peter Shirley. « State of the Art in Ray Tracing Animated Scenes ». Dans Dieter Schmalstieg et Jiří Bittner, éditeurs, *STAR Proceedings of Eurographics 2007*, p. 89-16. The Eurographics Association, septembre 2007. (Cité p. 37)

[WMM+04] Markus Weber, Marco Milch, Karol Myszkowski, Kirill Dmitriev, Przemyslaw Rokita et Hans-Peter Seidel. « Spatio-Temporal Photon Density Estimation Using Bilateral Filtering ». Dans *Computer Graphics International*, p. 120-127, 2004. <http://doi.ieeecomputersociety.org/10.1109/CGI.2004.1309200>. (Cité p. 129)

[WOL+05] K.-H. Wong, X. Ouyang, C.-W. Lim, T.-S. Tan et J. Nievergelt. « Rendering anti-aliased line segments ». Dans *CGI '05: Proceedings of the Computer Graphics International 2005*, p. 198-05, Washington, DC, USA, 2005. IEEE Computer Society. (Cité p. 27)

[WTL+04] Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo et Heung-Yeung Shum. « Generalized displacement maps ». Dans A. Keller et H. W. Jensen, éditeurs, *Eurographics Symposium on Rendering*, p. 227-34, 2004. (Cité p. 72, 82 et 111)

[WWT+03] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo et Heung-Yeung Shum. « View-dependent displacement mapping ». *ACM Trans. Graph.*, 22(3):334-39, 2003. <http://doi.acm.org/10.1145/882262.882272>. (Cité p. 72, 82 et 110)

[Wym05] Chris Wyman. « Interactive image-space refraction of nearby geometry ». Dans *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, p. 205-11, New York, NY, USA, 2005. ACM. <http://doi.acm.org/10.1145/1101389.1101431>. (Cité p. 122)

[Wym08] Chris Wyman. « Hierarchical caustic maps ». Dans *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, p. 163-71, New York, NY, USA, 2008. ACM. <http://doi.acm.org/10.1145/1342250.1342276>. (Cité p. 122)

[YYM05] Jingyi Yu, Jason Yang et Leonard McMillan. « Real-time reflection mapping with parallax ». Dans *Symposium on Interactive 3D graphics and games*, p. 133-38, 2005. <http://doi.acm.org/10.1145/1053427.1053449>. (Cité p. 122)

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang et Baining Guo. « Real-time KD-tree construction on graphics hardware ». Dans *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, p. 1-1, New York, NY, USA, 2008. ACM. <http://doi.acm.org/10.1145/1457515.1409079>. (Cité p. 37)

[ZPQS07] G. Zeng, S. Paris, L. Quan et F.X. Sillion. « Accurate and Scalable Surface Representation and Reconstruction from Images ». *IEEE transactions on pattern analysis and machine intelligence*, 29(1):141-158, January 2007. (Cité p. 162)

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar et Markus Gross. « Surface splatting ». Dans *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, p. 371-78, New York, NY, USA, 2001. ACM. <http://doi.acm.org/10.1145/383259.383300>. (Cité p. 27)

[ZT02] Eugene Zhang et Greg Turk. « Visibility-guided simplification ». Dans *VIS '02: Proceedings of the conference on Visualization '02*, p. 267-74, Washington, DC, USA, 2002. IEEE Computer Society. (Cité p. 162)