Visualisation Interactive de Modeles Complexes avec les Cartes Graphiques Programmables

Rodrigo Toledo

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Visualisation Interactive de Modèles Complexes avec les Cartes Graphiques Programmables

THÈSE

présentée et soutenue publiquement le 12 octobre 2007

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Rodrigo Toledo

Composition du jury

Président :	Jean-Paul Haton
Rapporteurs :	Luiz Velho Jean-Michel Dischler
Examinateurs :	Jean-Claude Paul (Directeur) Bruno Levy (Codirecteur) Xavier Décoret Luciano Reis (Membre Industriel)

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Remerciements

Mes toutes premières pensées sont pour ma femme Danielle qui a du se déraciner afin de me suivre pour ce périple de plusieurs années dans un pays étranger ainsi que pour ma famille.

Je remercie mes directeurs de thèse, Jean Claude Paul et Bruno Levy, pour leur invitation à venir travailler en France dans leur groupe de recherche.

Un grand merci en particulier aux personnes de l'équipe ISA qui m'ont grandement aidé à réaliser cette thèse : Fabien Boutantin, Cédric Lamathe, David Ledez, Ben Li, Bruno Vallet et Isabelle Herlish. J'adresse aussi mes remerciements à Bin Wang pour sa collaboration sur la conversion des *Geometry Textures*. Sans oublier Hazel Everett et Sylvain Petitjean pour leur conseils avisés sur la rédaction.

Mes autres collègues du LORIA sont aussi dans mes pensées : Yves Rangoni, Joseph Razik, Pavel Kral, Augusto, Judson et les autres personnes de leur bureau, ainsi que Sergio Bezerra de l'Institut Élie Cartan.

Je souhaite vivement remercier les personnes de Tecgraf à PUC : l'équipe d'informaticiens : Alberto, Paulo, Mauricio, Gustavo et Aurélio ; l'équipe de designers ; Carolina Alfaro pour les corrections du texte en anglais.

Je souhaite aussi remercier dans le désordre : mes anciens directeurs de master : Luiz Velho et Marcelo Gattass ; les personnes de PETROBRAS : Luciano Reis, Ismael Santos, Enio Russo, Mathieu Moriss, Anelise Lara et Alvaro Maia.

Mes remerciements vont bien entendu à toutes les personnes, collègues, amis et famille (en particulier ma mère Isabel), qui m'ont permis d'achever ce travail, m'entourant de leurs conseils et de leur soutien en toutes situations.

Enfin, je remercie les institutions suivantes qui ont permis à cette thèse de voir le jour: Tecgraf, PETROBRAS, CAPES, UHP et LORIA.

Je dédie cette thèse à ma femme Danielle
et à ma fille Caroline.

Sommaire

List of Figures

Chapter 1

Visualisation Interactive de Modèles Complexes

1.1 Le challenge

La visualisation d'objets et de scènes de grande taille est un des défis de la visualisation par ordinateur, particulièrement dans le cadre d'applications interactives. Celles-ci requièrent deux qualités: une vitesse de l'ordre de 30 images par seconde dans le but de donner une impression de temps-réel et la qualité. En effet, des images réalistes sont de plus en plus considérées comme une condition essentielle, notamment en ce qui concerne les simulateurs.

Ainsi est-il communément reconnu que l'industrie pétrolière est pourvoyeuse de données de grande taille. Ces données sont également caractérisées par la grande diversitée des objets et des scènes utilisés: terrains, données géologiques volumiques (sismique 3D), données SIG (Systèmes d'Information Géographique), modèles d'ingénierie complexes (plateformes, raffineries), puits de forages, oléoducs et gazoducs. En outre, des informations additionnelles sont souvent attachées à tous ces éléments: annotations, identifications, descriptions techniques, et conditions instantanées.

Parmi tous ces objets, deux grandes familles peuvent être distingués: les objets naturels et les objets manufacturés. Malgré cette classification, tous les objets sont communément représentes par des maillages triangulés. De nombreuses raisons expliquent ce choix du triangle comme base de la représentation des primitives géométriques: tout objet peut être discrétisé en un maillage triangulaire; tout polygone peut se décomposer en un ensemble de triangles; les points d'un triangle sont toujours coplanaires; les triangles sont toujours convexes; les coordonnées barycentriques peuvent toujours être utilisées comme règle non-ambiguë d'interpolation de valeurs attachées au point dans le domaine du triangle; de plus, les cartes graphiques sont spécialisées dans la 'rasterisation' des triangles. Cependant, le choix d'un maillage triangulé ne s'avère pas toujours judicieux, à la fois pour des questions de performance et de qualité de rendu des images.

Objets Manufacturés

Afin d'illustrer notre propos, nous concentrons nos efforts sur la visualisation des plateformes pétrolières, qui sont à la fois sujettes aux problèmes de grande taille et de qualité de visualisation. En effet, les données les représentant sont particulièrement grandes, englobant des millions de petits objets fréquemment représentés comme des ensembles de triangles. Les modèles industriels sont principalement composés de longs enchevêtrements de tubes, câbles et autres réseaux techniques, eux-même composés de combinaisons de primitives simples (cylindres, cônes et morceaux de tores). Plusieurs

problèmes se présentent lorsqu'on utilise des maillages triangulés pour représenter ces formes simples: (i) la représentation n'est qu'une approximation, donc jamais parfaite; (ii) plus on utilise de triangles, plus on affine l'approximation, mais plus il faut de traitement (et de mémoire) pour calculer la rasterisation; (iii) bien que l'application d'un ombrage approprié puisse embellir le rendu, il persiste une limitation au niveau de la silhouette (spécialement après un zoom).

Dans le but de répondre point par point à chacun de ces problèmes, nous avons développé une solution en deux temps.:

(i) Premièrement, les données d'entrée sont principalement disponibles sous forme de maillages triangulés. Pour cette raison, une première étape détecte au sein des ensembles de triangles les primitives de plus haut degré (cylindres, cônes et tores). Elle exploite notamment la nature topologique et géométrique des données (i.e.: les grandes structures tubulaires). Cet algorithme se révèle trés efficace.

(ii) Une fois les primitives trouvées, leur affichage se fait non plus via une approximation polygonale mais en utilisant leur équation exacte. Cette technique est basée sur le 'lancer de rayon' implémenté directement dans la carte graphique. Il en résulte des images d'une très grande qualité (sans problème de silhouette, et avec des intersections précises au niveau du pixel).

Objets Naturels

Plusieurs méthodes existent afin d'obtenir une modélisation des objets naturels: le scanner 3D, l'acquisition stéréoscopique, etc. En général, après plusieurs traitements, les données sont converties en maillages triangulés. Gu et al. [GGH02] ont proposé les 'Géometry Images', une représentation alternative et bien appropriée pour les objets naturels. L'objet est décomposé en un ensemble de grilles régulières. Chaque grille est simplement un tableau 2D de points 3D, le 'Geometry Image' (pour plus de détails, voir les explications dans la Section 4.4). Cependant, bien que plus compacte que le maillage original, cette représentation utilise encore plus de ressource pour le rendu des images. Afin d'améliorer les performances de rendu, nous avons proposé une nouvelle représentation que nous appelons *Geometry Textures*.

Une 'geometry texture' est composée d'une boîte englobante, d'une carte des hauteurs et d'une carte des normales. L'algorithme ne dessine pas les surfaces avec des triangles, mais utilise plutôt une méthode de lancé de rayon avec gestion de la profondeur. L'algorithme développé convertit un modèle complexe en un ensemble de 'geometry textures'. Les modèles représentés par un assemblage de 'geometry textures' sont rendus de manière efficace et sont compatibles avec les primitives standards des GPU. En allégeant la charge du vertex pipeline, nous obtenons de meilleures performances qu'un rendu traditionnel.

Nous avons exploré en détail les deux types d'objets et proposé des nouvelles solutions basées sur des représentations alternatives. Nous avons exploré les possibilités des cartes graphiques programmables pour réaliser le rendu interactif des modèles massifs. Dans la section suivante nous décrirons quelques concepts de visualisation, et pour finir nous exposerons les contributions de cette thèse.

1.2 Images informatiques et Visualisation

Nos travaux appartiennent au domaine de l'imagerie informatique et, plus précisément, de la visualisation, dont le but principal est la transformation de données en images. Cette description correspond à la définition donnée par l'ISO (International Standards Organization) de l'ensemble du domaine de l'imagerie informatique. Nous préférons donc parler du domaine de l'imagerie informatique dans un sens plus vaste, en incluant

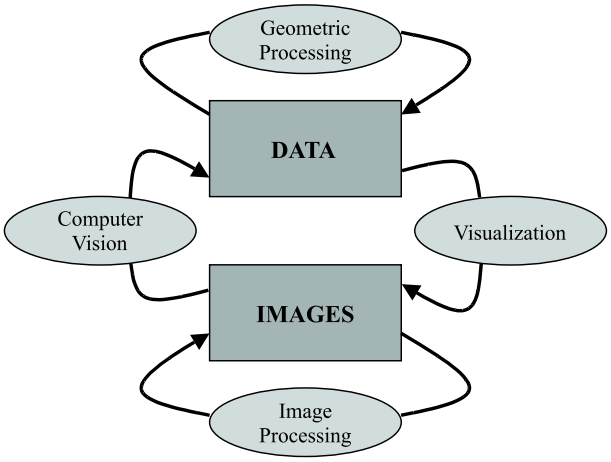


Figure 1.1: Disciplines de l'imagerie informatique (voir [GV97]).

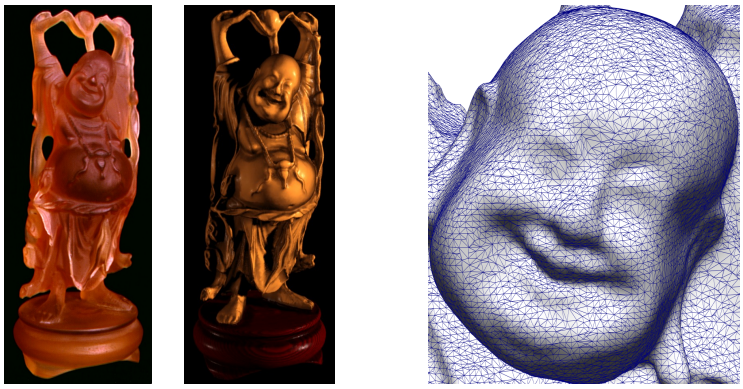
d'autres domaines comme le *traitement d'images* et la *vision par ordinateur*. Nous utilisons pour cela la classification de Gomes et Velho [GV97] basée sur la nature des entrées et sorties de chaque discipline (soit objets géométriques, soit images) et qui distingue quatre disciplines. **(Données ⇒ Données) Le traitement géométrique**, aussi appelé *traitement de données* ou *modélisation géométrique*, détermine la facon dont les données seront représentées et mémorisées par l'ordinateur. De nouvelles structures, attributs et propriétés peuvent être calculées grâce aux informations du modèle. Citons quelques exemples: calcul de la courbure, remaillage, structures hiérarchiques, paramétrisation, calcul d'enveloppes convexes, etc. Les traitements géométriques n'ont pas pour but principal la visualisation. Cependant, elle est un outil essentiel permettant la vérification des résultats obtenus par le traitement.

(Données ⇒ Images) La visualisation est le principal but de l'imagerie informatique. À partir de données mémorisées, l'algorithme de visualisation produit des images visibles grâce aux dispositifs d'affichage.

(Images ⇒ Images) Le **traitement d'image** prend des images en entrée et fournit d'autres images en sortie. Tous les type de traitements possibles sur des images sont contenus dans cette discipline, par exemple: flou, augmentation de détails, réduction de bruit, etc. Parfois, le traitement d'image est utilisé comme un post-traitement pour la visualisation ou comme un pré-traitement pour la 'vision par ordinateur'.

(Image ⇒ Données) La **vision par ordinateur** a pour but l'automatisation de l'extraction d'informations à partir d'images. Les applications sont nombreuses: robotique, reconnaissance de formes, reconnaissance de texte, et aide à l'analyse médicale.

La visualisation peut se diviser en deux axes majeurs: la visualisation en elle-même et la représentation des données. C'est pourquoi, dans un premier temps, nous introduisons la représentation des données et montrons en quoi il s'agit d'un problème important. Puis, dans une deuxième Section 1.2.2, nous décrivons succictement l'algorithme de suppression des surfaces cachées (HSR: Hidden Surface Removal), sur lequel nous reviendrons plus en détail dans la Section3.1, responsable des tests de visibilité. Finalement la visualisation étant un sujet très vaste, nous nous intéressons plus spécialement à la visualisation interactive de scènes massives et d'objets complexes.



(a) (b)

Figure 1.2: (a) Deux objets distincts avec des apparences différentes, mais la même géométrie (b).

1.2.1 Représentation d'objets

En visualisation 3D, nous voulons reproduire, à travers une image à deux dimensions, des objets et des scènes tels que l'oeil humain peut les voir (même s'ils n'existent pas dans le monde réel). Dans cette perspective, plusieurs éléments sont nécessaires comme, par exemple, une manière de représenter, mathématiquement et informatiquement, ces objets. Cette représentation doit décrire la géométrie de l'objet à reproduire. Mais, décrire la géométrie d'un objet n'est pas suffisant afin de le représenter complètement car il est aussi nécessaire d'avoir des informations sur son apparence. Par exemple, deux objets peuvent avoir exactement la même géométrie, mais avoir de couleurs diffèrentes ou composées de matériaux différents (e.g. caoutchouc et métal), voir la Figure 1.2(a).

Généralement, pour décrire leur géométrie dans $\mathbb{R}^3$, les objets sont décomposés en éléments de base surfaciques, ou volumiques, appelés *primitives*. Les primitives sont analytiquement définies par des paramètres de haut niveau (e.g., une primitive sphérique est définie par un centre C et un rayon r). La plus courante est le triangle défini par ses trois sommets. Les *maillages triangulés* (ou leur forme généralisée, les *maillages polygonaux*) sont très pratiques et flexibles pour représenter la géométrie de nombreuses classes d'objets. Voir la Figues 1.2(b).

Cependant, il existe nombre d'attributs autres que géométriques tout aussi essentiel pour une représentation réaliste d'un objet. Ainsi la nature du matériau constituant un objet joue-t-elle un rôle prépondérant dans les calculs d'éclairage et de gestion des ombres. Pour bien comprendre quelles sont les propriétés des matériaux significatives pour le stockage, il est important de bien comprendre les modèles d'éclairage. Ils peuvent varier énormément en complexité (cf. Section 4.2.3), mais pour les modèles simples, quelques propriétés suffisent à décrire l'apparence d'un objet (e.g., les propriétés diffuses et spéculaires). À noter les modèles d'éclairage plus sophistiqués prenant en compte des informations relatives à l'aspect imparfait des surfaces. En effet, la lumière réfléchie est intimement liée à la micro-géométrie des surfaces. Mais il serait en pratique impossible de le simuler et le représenter. C'est pourquoi, la représentation des objets va rarement aussi loin (que les détails microscopiques). La prochaine étape est le rendu réaliste de la scène.

1.2.2 Visualisation et rendu

Une des premières tâches en visualisation consiste à déterminer quels sont les objets visibles dans une scène. Les algorithmes en charge de ce travail sont appelés algorithmes de '*suppression de surface cachées*' (HSR: Hidden Surface Removal). Certains auteurs les

appellent aussi algorithmes de *détermination de surface visibles* (Visible-surface Determination) [FvDFH90]. Un objet (ou tout du moins une partie d'un objet) est considéré visible s'il se trouve dans le champ de vision et qu'aucun autre objet ne situe entre lui et l'observateur. Les algorithmes d'HSR les plus populaires sont le tampon des données de profondeur (z-buffer [Cat74]) et le lancer de rayon [App68]. Le tampon des données de profondeur (ou z-buffer) est implémenté dans toutes les cartes graphiques en combinaison avec l'algorithme de tramage des triangles. Le lancé de rayon est, quant à lui, utilisé quand la qualité est plus importante que la vitesse (pour des applications hors-ligne), par exemple pour des images fixes de haute qualité et les films d'animation. Nous discuterons de la suppression des surfaces cachées dans la section 3.1. Cette distinction entre les applications graphiques hors-ligne et en-ligne divise le domaine de la visualisation en deux domaines: le *rendu hors-ligne* et la *visualisation interactive*. Nos travaux rentrent dans le cadre de la visualisation interactive. La détermination des parties visibles et cachées de chaque objet ne constitue cependant qu'une partie du travail. Un autre aspect consiste à calculer correctement l'apparence de la surface en fonction de son environnement lumineux et des propriétés de sa matière (comme décrites dans la section précédente). Nous avons vu qu'il existe de nombreux modèles d'ombrage pour le rendu graphique. Par exemple, un modèle simple d'ombrage peut ne considérer que les *lois de Lambert* stipulant que, étant données une surface diffuse idéale et une source d'énergie directionelle, la surface réfléchit la totalité de l'énergie reçue dans toutes les directions avec une intensité qui dépend de l'angle θ formé par la normale de la surface et la direction de la source. Plus récemment, Phong [Pho75] a proposé d'ajouter une composante de réflexion spéculaire au modèle afin de simuler les surfaces brillantes (comme la peau d'une pomme). Un autre exemple d'ombrage, encore plus complexe, est basé sur les fonctions BRDF (fonctions de distribution de réflexion bi-directionnelles) [NRH$^+$77] qui décrit la quantité de lumière réfléchie en fonction de la direction de la lumière incidente et de l'angle de vision. Cependant, comme mentionné précédemment, la lumière réfléchie est le résultat de phénomènes qui surviennent au niveau microscopique et, en général, les modèles d'ombrage n'en sont que des approximations. Outre les problèmes standards de représentation et de visualisation, notre travail veut prendre en compte deux contraintes supplémentaires importantes: l'interactivité et un grand nombre de données en entrée.

Visualisation interactive

Le mot "interactive" signifie qu'il existe une relation entre le système de visualisation et les actions de l'utilisateur. Les images produites par l'algorithme de visualisation sont en effet le résultat des ordres donnés par l'utilisateur. Ce dernier interagit avec le logiciel par le moyen de dispositifs de commande (par exemple la souris et le clavier), espérant une réponse visuelle immédiate. La latence de cette réponse doit être inférieure à un temps maximum, afin que l'utilisateur ait l'impression de travailler avec un système *temps-réel*. La limite du temps de réponse est subjective et il n'existe pas de valeur reconnue par tous dans la littérature. La vitesse minimum d'affichage, acceptée comme interactive, varie entre 10ips et 30ips (*Images Par Seconde*). Ainsi, tout système graphique fonctionnant à une vitesse supérieure à 30ips pour être qualifié d'interactif. Avec l'aide des cartes graphiques actuelles, il est facile d'atteindre ces vitesses sur des scènes simples (jeux vidéos). Cependant, la visualisation de quantités importantes de données reste un défi, et n'est possible qu'en combinant plusieurs solutions spécialement adaptées.

Visualisation de quantités importantes de données et multi-échelles

La visualisation interactive des données de grande taille représente encore aujourd'hui un point d'intense recherche dans l'industrie. En effet, le nombre de données et leur complexité ne cessent de croître avec l'évolution des technologies. De plus, l'utilisateur devient de plus en plus exigant quant à la qualité des données à traiter et la qualité du

rendu. C'est pourquoi, tandis que l'utilisateur parcourt le scène virtuelle, le système doit correctement résoudre la visibilité et l'ombrage pour chaque pixel de l'image à l'écran, et ce avec une grande vitesse de rafraîchissement (e.g. au moins 30ips). En général, le volume de données traitable augment à la vitesse, voir plus vite, que la loi de Moore sur les processeurs. [1]. Pour cette raison, beaucoup de travaux ont été dédiés à l'accélération des algorithmes de visualisation afin de conserver les performance avec des images de qualité.

Dans une première partie, les différents algorithmes de visualisation et les systèmes d'accélération sont évoqués en fonction de l'échelle à laquelle ils sont appliqués: niveau de la scène, macroscopique (niveau de l'objet), mésoscopique et microscopique. Pour chaque échelle sont énumerés les principaux problèmes pour une visualisation correcte, les solutions apportées et comment accélérer les procédures:

Au niveau de la scène: À cette échelle, la question la plus importante est celle de la visibilité. Connaissant les paramètres de la caméra (position, direction et champ de vision) et les objets de la scène, lesquels sont visibles? Répondre à cette question est le but des algorithmes HSR décrits dans la section 3.1. Afin d'accélérer les procédures, plusieurs algorithmes ont été mis au point, à la fois pour le lancer de rayons et pour la 'rasterisation'. Dans les sections 3.2 et 3.3, nous passons en revue ces différents algorithmes d'accélération.

Niveau macroscopique (objets): La macro-géométrie d'un objet peut être représentée de multiples façons (e.g. surface implicites, maillage triangulé). Cependant, l'application des algorithmes classiques d'HSR pose problème. Quelle est la meilleure représentation pour une visualisation rapide et de qualité? Un nouveau concept est alors introduit, la *multirésolution*: un objet peut-être représenté à plusieurs résolutions (chacune d'entre elles étant une approximation de la macro-géométrie avec plus ou moins de détails). Il est à noter que la 'multi-échelle' et la multi-résolution sont deux notions totalement différentes. Ce dernier ne fait que modifier le niveau de détails de la géométrie macroscopique. La macroscopie est explorée plus en détails dans le chapitre 4.

Niveau mésoscopique: L'échelle mésoscopique se focalise sur un moyen de représenter (et de rendre) les détails des objets sans une extraction complète et exacte des micro-informations, et c'est pourquoi elle se trouve à un niveau intermédiaire (*meso*). C'est à ce niveau que le concept de texture apparaît. La texture est d'une certaine manière en relation avec la sensation tactile. Lorsque quelqu'un touche un objet avec ses mains, il peut sentir sa texture. Mais la texture est également à l'origine des informations visuelles d'une fàçon telle que nous pouvons nous imaginer ces sensations rien qu'en regardant un objet. Le placage de texture (Texture mapping [Cat74]), de relief (bump mapping [Bli78]), et d'application de textures bidirectionnelles (Bidirectional Texture Mapping, BTF [DvGNK99]) sont des exemples de techniques qui reproduisent les détails d'une surface sans polygones supplémentaires. Nous évoqueront ces méthodes parmi d'autres méthodes de mesostructures dans le chapitre 5.

Niveau microscopique (photons): Au niveau microscopique, les structures et les phénomènes sont plus difficiles à représenter et à simuler. L'étude à l'échelle du photon revient à regarder les détails d'une surface avec un microscope, en ralentissant un faisceau de photons pour comprendre comment la lumière est altérée par les propriétés

[1] Le 19 avril 1965, *Electronics Magazine* a publié un papier de Gordon Moore dans lequel if fit une prédiction à propos de l'entreprise des semi-conducteurs qui est devenu légendaire. Il dit que le nombre de transistors que l'industrie serait capable de placer dans un processeur informatique doublerait tous les ans (source: http://www.intel.com). Dans les dernières années, le rythme a baisé un peu, mais la densité des données a doublé tous les 18 mois, et il s'agit de la définition actuelle de la loi de Moore, qu'il a lui même 'bénie' [DCM04].

physiques et géométriques. A ce niveau de détail, plusieurs phénomènes se produisent simultanément: réflexion, refraction et absorption. La somme de tous ces phénomènes conduit au comportement de la lumière quittant la surface, et qui finalement arrive dans l'oeil de l'observateur.

Considérer les différentes échelles d'application est très important pour accélérer la visualisation étant donné que les problèmes sont différents pour chaque niveau. Par exemple, la plupart des techniques pour accélérer la vision au niveau de la scène ne peuvent être utilisés au niveau mésoscopique et vice-versa. C'est pour cette raison que nous avons décidé de regrouper les méthodes par niveau d'application.

1.3 Nos Contributions:

1.3.1 Combinaison de Rasterization et lancer de rayons

Dans la section 3.1, les algorithmes de surfaces cachées (HSR) les plus connus sont décrits: le lancer de rayons (ray-tracing ou ray-casting) et la rasterisation avec z-buffer. La première permet de déterminer pour chaque pixel quel objet est visible, tandis que la seconde détermine pour chaque primitive quels pixels doivent être rendus. Bien qu'elles semblent avoir des 'implémentations' orthogonales, des recherches ont été faites afin de combiner ces solutions en une seule regroupant certains de leurs avantages (cf. Section 3.1.4).

Nous proposons nous-mêmes une combinaison des algorithmes de rasterisation et de lancer de rayon pour le rendu d'une scène, mais d'une façon originale. Avec notre approche, certains objets sont rasterisés tandis que d'autres sont dessinés par lancer de rayons, indépendemment d'un quelconque ordonnancement. Ceci est rendu possible par l'utilisation du z-buffer lors de la résolution de la visibilité entre objets (cf. Section 7.1). Le but est d'utiliser l'algorithme de rendu (lancer de rayons ou rasterisation) le plus approprié à la représentation de chaque objet.

Comparativement aux systèmes de visualisation classiques, qui utilisent essentiellement la rasterisation, nous obtenons une bien meilleure qualité et ce avec de meilleures performances pour les objets massifs (cf. Chapitre 11). Notre solution utilise la programmation GPU (Graphics Processing Unit) pour effectuer le rendu des primitives par lancer de rayon et aussi pour exécuter la rasterisation classique.

1.3.2 Le lancer de rayons par GPU

Notre proposition utilise les extensions des cartes graphiques programmables actuelles afin de leur faire effectuer le rendu des primitives par lancer de rayon. Ces primitives devrait être particulièrement efficaces afin d'assurer une inter-activité convenable et ce, même pour des jeux de données très gros. Elles doivent également être compatibles entre elles, mais aussi avec les triangles rasterisés (qui sont les primitives de bases des graphiques).

Notre technique de visualisation se décompose en deux étapes exécutées dans le GPU en une seule passe sans changer le pipeline: (1) Les pixels possibles sont pré-sélectionnés par la rasterisation d'une primitive classique dont la projection englobe celle de nos primitives. (2) Pour chacun de ces pixels, un fragment programme calcule l'intersection et l'éclairage, par la méthode du lancer de rayon sur une représentation analytique de nos primitives. De plus, il change la valeur dans le z-buffer en fonction de l'équation de nos primitives.

Dans cette thèse, nous présentons cette procédure qui est applicable pour différentes classes de primitives de lancer de rayons qui respectent les contraintes données précédem-

ment (cf. Section 7.1). Ces primitives peuvent être n'importe quelle surface (ou volume) dont l'algorithme de lancer de rayon est connu, et dont un objet convexe englobant (ou plus simplement une boîte de circonscription) est donné.

Aussi avons-nous implémenté quelques surfaces implicites avec différents ordres de complexité (quadriques, cubiques, quartiques) et des surfaces représentées par des cartes de hauteurs.

1.3.3 Implémentations dans le GPU des primitives implicites

Dans un premier temps, notre implémentation s'est attachée à traiter des surface implicites d'ordre deux, les quadriques. Nous avons développé un algorithme générique (cf. Section 10.1) mais aussi des algorithmes dédiés à des primitives plus spécifiques: sphères (Section 10.2), ellipsoïdes (Section 10.3), cylindres (Section 10.4) et cônes (Section 10.5). Ces primitives sont particulièrement rapides à dessiner grâce à un algorithme d'intersection entre des rayons et des quadriques directement implémenté dans les GPUs programmables.

Dans la section 10.6, nous avons étendus cette procédure pour les surfaces cubiques. Enfin, nous avons développé une méthode spécialement adaptée aux tores (Sections 10.7 et 10.8). Les tores ont en effet une représentation implicite quartique, ce qui implique la résolution d'une équation de degré quatre. Nous avons essayé différentes solutions pour implémenter directement des méthodes numériques dans la carte graphique (Sturm, double derivative bisection, sphere tracing, Newton-Rampson) et nous avons obtenu les meilleurs résultats avec la méthode de Newton.

Le développement de techniques de visualisation spécifiques à ces primitives quadriques, cubiques et quartiques a été largement inspiré par la perspective de leur utilisation pour la visualisation d'environnements industriels (cf. Chapitre 11), comme les plateformes pétrolières.

1.3.4 Ingénierie inversé

La plupart des objets géométriques des infrastructures industrielles sont des combinaisons de quadriques communes et de sections de tores (ces derniers étant utilisés pour les jointures tubulaires en forme de coudes). Cependant, les données sont habituellement prétraitées, c'est-à-dire discrétisées, sous la forme de maillages triangulés. Sans les données originales, nous ne pouvons utiliser nos primitives spéciales pour la visualisation. Pour cette raison, nous avons développé un procédé afin de retrouver les primitives originales à partir des données triangulées. L'algorithme parcourt le maillage pour retrouver la structure originale des données (cf. Chapitre 9).

1.3.5 Implémentation des cartes de hauteur dans le GPU

Également basé sur les primitives rendues par le GPU, nous avons implémenté un algorithme de visualisation des surfaces représentées par des cartes de hauteur. Avec notre technique, ces primitives sont visualisées avec leur propre ombrage, une silhouette correcte et auto-occlusion. Ils peuvent être utilisés pour le rendu de 'mesostructures' (Chapitre 5) et aussi comme nouvelle représentation d'objets naturels complexes.

1.3.6 'Geometry Textures' et algorithme de conversion

Les surfaces complexes scannées sont usuellement représentées par un nuage de point ou bien un maillage triangulé. Nous proposons une nouvelle représentation basée sur un jeu de cartes de hauteur (les *Geometry textures*, cf. Chapitre 7). Cette nouvelle structure est visualisée grâce à la technique décrite dans la section précédente. Pour arriver à ce

résultat, la première étape consiste à convertir un maillage triangulé en un ensemble de 'geometry texture', comme expliqué dans la Section 8.1.

1.3.7 Etude de l'état de l'art en visualisation basé au niveau de échelle

Une de nos contributions est l'étude sur la visualisation (Chapitre 3, 4, 5 et 6). Nous avons pris la décision de grouper les techniques en fonction de leur taille: scène, macro, meso, et micro. Nous revoyons ici les algorithmes de visualisation, des plus basiques, aux plus récentes méthodes utilisant les cartes graphiques programmables, faisant ainsi des comparaisons appropriées.

1.4 La Thèse

Dans le domaine de la modélisation 3D, il existe plusieurs approches différentes pour représenter des objets, chacune d'entre elles étant bien adaptée à un domaine d'applications précis. La modélisation volumique représente les objets comme des combinaisons booléennes de solides, alors que la modélisation surfacique représente les objets par leurs bords. Les surfaces manipulées peuvent être représentées par des fonctions (quadriques, splines) ou bien discrétisées sous la forme d'ensembles de polygones. La construction et l'édition de ces modèles requièrent le développement de techniques de visualisation au sein du logiciel. L'objectif est la mise au point de solutions de visualisation de grande qualité et facilement modulables pour des applications variées dans l'industrie automobile, aéronautique, la médecine, la géologie, la botanique, la dynamique moléculaire ou la physique. Avec les avancées des techniques d'acquisition, de stockage et des capacités de calcul des ordinateurs, la taille des données à traiter augmente plus vite que la loi de Moore et représente une nouvelle aventure technique et scientifique. Par exemple, l'avènement récent des nouvelles générations de cartes graphiques ouvre de nouvelles perspectives pour une visualisation temps-réel de très grands modèles.

Cette thèse se propose de faire un point sur ces nombreuses techniques de visualisation et de contribuer à étendre certains aspects de ces méthodes. Nous avons organisé ce document en trois grandes parties:

- *L'État de l'Art - Partie I* (Chapitres 3, 4, 5 et 6). En premier lieu, nous décrivons un ensemble de travaux existants dans le domaine de la visualisation. Ils sont regroupés selon l'échelle à laquelle ils sont appliqués, pour fournir une comparaison directe de techniques similaires. Suivant une profondeur décroissante, chaque niveau d'échelle est exploré dans un chapitre différent : niveau scénique, niveau macroscopique, niveau mésoscopique et niveau microscopique.

- *Modèles Naturels - Partie II* (Chaptires 7 et 8). En opposition aux modèles manufacturés, nous considérons ici tous les objets naturels, mais également ceux fabriqués à la main par l'homme. Par exemple: objets géologiques, sculptures et données volumétriques.

- *Modèles Industriels - Partie III* (Chapitres 9, 10 et 11). Dans cette famille de modèles nous incluons tous les objets créés par l'industrie, particulièrement ceux qui ont été conçus par ordinateur (modèles de CAO). Généralement ils sont créés par des ingénieurs ou des techniciens et sont fondés sur des équations et des formules mathématiques, ayant pour résultat des formes exactes. Par exemples: moyens de transport, produits manufacturés et usines industrielles.

Partie I: L'état de l'art en visualisation

1.4.1 Niveau scénique

Au niveau de la scène, la visibilité des différents éléments est le premier problème à prendre en compte. Dans la Section 3.1 nous passons en revue les principaux algorithmes de visibilité : l'algorithme du peintre, le lancer de rayons et la rasterization. Pour optimiser la visualisation à ce niveau, il est essentiel d'organiser les objets dans une structure spatiale. C'est le sujet de la Section 3.2 (*subdivision spatiale de scène*). Dans les systèmes de lancer de rayons, la structure spatiale diminue la compléxité de l'algorithme de calcul d'intersection de rayon à $O(\log N)$ pour les N objets dans la scène [Wal04]. Pour les systèmes de rasterization, les structures sont utiles pour les algorithmes de *culling*, section 3.3. L'algorithme de *culling* peut éliminer de nombreux objets dans une scène complexe [BWPP04], réduisant ainsi le nombre d'objets rendus.

Proposition de thèse: Combinaison de la Rasterization et du Lancer de Rayons
Nous proposons un algorithme de visualisation qui combines des objets rasterizés et des objects dessinés par lancer de rayons dans la même scène.
Comme nous verrons dans la Section 4.2, il y a une classe d'objets pour lesquels le lancer de rayons est plus approprié que la rasterization. En employant l'algorithme de rendu le plus approprié pour chaque objet, nous réalisons une exécution plus rapide, avec une meilleure qualité et moins de consommation mémoire (cf. 7.1).

1.4.2 Niveau Macroscopique

La macro-géométrie d'un objet peut être représentée de multiples manières: par des surfaces implicites, des surfaces paramétriques ou par une représentation explicite (une maillage triangulaire par exemple). Les Sections 4.1 et 4.2 discutent sur les types de représentation.

La Section 4.3 explore les sujets de simplification et de multi-résolution de la géométrie pour accélérer le rendu. Dans la Section 4.4 nous montrons les méthodes qui remplacent la géométrie par des images pour la visualisation. Dans la Section 4.5 nous discutons des méthodes qui convertissent des surfaces explicites en volumes pour accélérer la visualisation.

Proposition de thèse: Méthode de visualisation adaptée au type primitif
Nous proposons un algorithme de visualisation qui utilise soit la rasterization soit le lancer de rayons selon le type primitif.
Pour les représentations explicites (maillage triangulaire) le système emploie la rasterization et pour les représentations implicites (par exemple, sphères, quadriques et tores) le système emploie le lancer de rayons. Nous avons obtenu des taux d'affichage interactifs en implémentant le système dans les GPU programmables. Nous testons cela en scènes complexes, avec de nombreux objets implicites, résultant en une vitesse et une qualité supérieure. (cf. Chapter III pour des objets et des scènes manufacturés).

1.4.3 Niveau mésoscopique

Au niveau mésoscopique nous nous sommes intéressés à l'aspect d'objets.

L'aspect d'un objet est le résultat de la microstructure de sa surface. Un modèle complet qui tient compte de la microstructure et de tous événements physiques se produisant à ce niveau serait impraticable et non interactif. Pour cette raison beaucoup d'effort a été consacré à la représentation de l'aspect à un niveau mésoscopique, entre les niveaux micro et macro. La structure surfacique que nous devons représenter dans ce niveau est appelée mesostructure. Les résultats visuels dépendent de la représentation de mesostructure mais dépendent également du modèle d'éclairage. Les modèles d'éclairage utilisés dans la macro-échelle (cf. Section 4.2.3) peuvent être appliqués au niveau mésoscopique, mais normalement un modèle plus sophistiqué est nécessaire.

Texture

Lorsque quelqu'un touche un objet avec ses mains, il peut sentir sa texture. Cette information tactile est un résultat de la *texture surfacique*. Mais la texture est également à l'origine des informations visuelles d'une façon telle que nous pouvons nous imaginer ces sensations rien qu'en regardant un objet. La texture est un résultat de la mesostructure de l'objet, elle peut être simulée de plusieurs manières dans des systèmes de visualisation.

Sa représentation la plus simple est la *carte des couleurs* (Section 5.1), qui est une image 2D simple appliquée sur un objet virtuel. Des représentations plus complexes de mesostructure sont la *carte des normal* (Section 5.2), la *carte des hauteurs* (Section 5.3) et le *volume* (Section 5.4). En conclusion, dans la Section 5.5, nous présentons des méthodes qui utilisent la fonction d'ombrage plutôt que de représenter la mesostructure.

Contribution de thèse: Lancer de Rayons de Mesostructures
En utilisant notre algorithme de lancer de rayons implémenté dans le GPU, il est possible de reproduire la mesostructure des surfaces representés au niveau macroscopique.

Technique [Reference]	Visual Effects					Characteristics				
	Detailed Lighting	Detailed Silhouette	Self-occlusion	Self-shadow	Interreflection	Extra Memory	Extra Polygons Factor	Input Difficulty	Preprocessing	Rendering Speed
Color										
Texture Mapping [Cat74]						3	1	L		↑↑
Normal										
Bump Mapping [Bli78]	X					3	1	M		↑
Height Map										
Displacement Mapping [Coo84]	X	X	X			1	10^2	M		—
VDM [WWT+03]	X	X	X	X		2^{12}	1	M	X	↑
Ray-casting [TLP03, POC05a]	X	X	X	X		1	1	M		—
Prism Ray-casting [HEGD04]	X	X	X			1	4	M		—
Volumetric Mesostructure										
Semi-Transparent [LDS02]	X	X	X			2^7	3	H		—
GDM [WTL+04]	X	X	X	X		2^{14}	4	H	X	—
Shell Texture Function [CTW+04]	X	X	X	X	X	10	NA	VH	X	↓↓
Shading Function										
BTF [DvGNK99]	X		X	X	X	2^9	1	VH	X	↓
PTM [MGW01]	X			X	X	5	1	H	X	↑
TensorTexture [VT04]	X		X	X	X	2^{11}	1	H	X	↓↓

Table 1.1: Comparaison entre différentes visualisations interactives de mesostructure (cf. Section 5.6 pour les explications détaillés.

Nous résumons les effets et les caractéristiques visuels des principales techniques de visualisation de mesostructure dans le Tableau 1.1. Certains des nombres présentés sont des approximations, mais ils sont signicatifs pour comparer les différentes techniques.

1.4.4 Niveau Microscopique

Dans ce sujet, nous discutons des petits détails dans l'aspect visuel. Le niveau microscopique est un sujet important et il clôture bien l'état de l'art sur la visualisation.

Modèle d'Éclairage

Nous décrivons à la Section 4.2.3 les modèles simples d'éclairage, appropriés à la macrogéométrie. Ces modèles ne tiennent pas compte de certains effets visuels, telles que l'interreflection, l'absorption de la lumière et la dispersion.

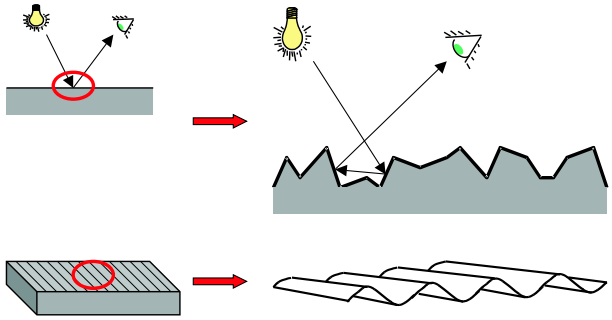


Figure 1.3: *Top:* Zoom on the surface microstructure: the microfacets interfere in the optical system. *Bottom:* Zoom on the mesostructure of an anisotropic surface.

Pour mieux comprendre ce qui se produit avec la lumière réfléchie des surfaces, nous discutons d'un modèle d'éclairage plus générique que le système diffuse/spéculaires, le BRDF (*Bidirectional Reflection Distribution Function*), cf. Section 6.1. Ensuite, nous décrivons une extension de BRDF, BSSRDF (*Bidirectional Subsurface Scattering Reflection Distribution Function*) dans la Section 6.2. Puis, nous expliquons deux techniques de visualisation basées sur des phénomènes de lumière à micro-échelle: *Photon Mapping* (Section 6.3), utilisé généralement dans des applications *off-line*; et *Precomputed Radiance Transfer Function* (Section 6.4) approprié au rendu en temps réel.

Partie II: Modèles Naturels

Nous avons découpé cette partie sur les modèles naturels en deux chapitres.

Le Chapitre 7 débute en présentant les principes des primitives GPU (Section 7.1), et explique les détails sur la primitive GPU de la carte d'hauteur (Section 7.2). Dans la Section 7.3, nous présentons *geometry texture* notre primitive de base pour visualiser les surfaces naturelles.

Dans le Chapitre 8, nous expliquons la conversion des objets naturels représentés par des triangles dans nos *geometry texture* (Section 8.1). Nous concluons la Partie II en montrant les résultats de rendu avec notre technique (Section 8.2).

1.4.5 Principes des Primitives GPU

Les cartes graphiques sont optimisés pour rasterizer et résoudre la visibilité des points, des lignes et des triangles. Nous proposons une implémentation basée sur le GPU de nouvelles primitives graphiques (par exemple sphères, cylindres, ellipsoïdes, carte de hauteurs), prolongeant le *pipeline* graphique (voir l'Annexe A). Nos primitives GPU étendues sont visualisées par un algorithme de lancer de rayons éxecuté à l'intérieur du GPU. Les calculs principaux sont faits au niveau des pixels dans le *pipeline*.

Pour déclencher l'algorithme par-pixel, il est encore nécessaire de rasterizer un ensemble de primitives standards (par example, des triangles). L'aire projetée de celles-ci doit couvrir l'aire projetée de la primitive GPU étendue. Nous appelons la première *RCA* (*Ray Casting Area*). Cette restriction assure que chaque pixel, d'où un rayon lancé intersecte la primitive étendue, exécutera l'algorithme (voir la figure 1.4).

En se basant sur la restriction ci-dessus nous pouvons énoncer quel type de primitive de GPU nous pouvons créer :

Si une surface ou un objet a une boîte englobante connue ou une enveloppe convexe (bidimensionnelle ou tridimensionnelle) pour lesquelles il y a un algorithme de lancer de rayons, alors il est possible d'en faire un primitive GPU.

Ces surfaces ou objets peuvent être implicitement décrits (e.g. les quadriques, cubiques et surfaces quartiques) ou explicitement décrits (e.g. carte des hauteurs et vol-

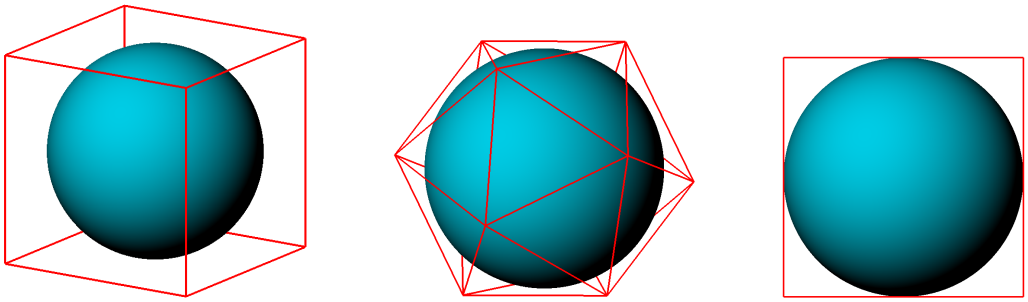


Figure 1.4: Exemple de différents ensembles de primitives standards (en formation d'un cube, d'un icosahedron et d'un carré) qui couvrent l'aire projetée de la sphère GPU (cf. Section 10.2).

umes). Il y a toujours une restriction puisque les cartes graphiques ne possèdent ni une mémoire ni une précision de calcul infinies, mais la flexibilité des GPUs modernes atteint presque la même capacité que celle des CPUs.

Un autre point important pour nos primitives GPU est que nous voulons les garder compatibles avec la méthode de rasterization. Par conséquent, des objets composés par des maillages triangulaires, et d'autres par des primitives GPU, doivent être visualisés dans la même scène. Pour cette raison, nous proposons une approche *hybride*, dans laquelle les objets dessinés par lancer de rayons et ceux rasterizés sont montrés ensemble en combinant les deux algorithmes de HSR les plus populaires (cf. Section 3.1).

La question de la visibilité entre les objets est résolue par le *z-buffer*, mis à jour par la rasterization mais aussi, par le lancer de rayons. Ainsi, dans la Section 7.1.1, nous commençons par expliquer notre algorithme d'HSR hybride et après, nous expliquons comment l'algorithme de visualisation de primitives GPU est partagée entre le *vertex* et *pixel shaders*.

RCA (*Ray Casting Area*)

Le choix d'un bon *RCA* pour une primitive est basé sur trois critères ((i), (ii) et (iii) ci-dessous).

L'ensemble des pixels dans une fenêtre d'écran est défini comme

$$W = \{\ (x,y)\ |\ x \in [1, width],\ y \in [1, height],\ x, y \in \mathbb{N}\ \}$$

Une primitive 3D dessinée dans l'écran W utilise également un ensemble de pixels définis par

$$S_{prim} = \mathcal{C}\big(\mathcal{P}(primitive), W\big)$$

$$\begin{aligned} \text{où} \quad & \mathcal{P} : \mathbb{R}^3 \mapsto \mathbb{R}^2 && \text{(projection)} \\ & \mathcal{C}(X, W) = X \cap W && \text{(clipping)} \end{aligned}$$

Chaque *RCA* est un sous-ensemble de W, définissant les pixels d'où les rayons sont lancés pour une primitive donnée. Pour visualiser correctement une primitive à l'intérieur du *RCA*, S_{prim} doit être dans l'aire du *RCA*

$$S_{prim} \ \subseteq\ RCA \ \subseteq\ W \qquad \text{(i)}$$

Le *RCA* est construit en assemblant les aires projetés des primitives standards habituellement affiche par le GPU : points, segments et polygones, avec leurs coordonnées définies dans $\mathbb{R}^3$.

$$RCA = \bigcup_i\ \mathcal{C}\big(\mathcal{P}(p_i), W\big) \qquad \text{(ii)}$$

$$\text{où} \quad p_i \in \{point, segment, polygon\}$$

En pratique, les primitives (p_i) sont celles qui seront passées à la carte graphique pour qu'elles soient rasterizées.

Nous définissons également deux fonctions que nous nous sommes intéressés à minimiser

$$V(RCA) = \sum_i \text{ sommets du } p_i$$
$$Waste(RCA, S_{prim}) = \frac{\mathcal{A}_{RCA} - \mathcal{A}_{S_{prim}}}{\mathcal{A}_{RCA}}$$
$$\text{où} \quad \mathcal{A}_X = \text{l'aire du } X = \#\text{ des pixels}$$

La fonction *Waste* mesure le gaspillage des pixels où l'algorithme de lancer de rayons est executé, mais il n'y a pas d'intersection avec la primitive à visualiser. La fonction V compte combien de sommet sont employés pour construire le *RCA*.

Le troisième critère est énoncé par

$$\text{minimizer } (\alpha_1 Waste(RCA, S_{prim}) + \alpha_2 V(RCA)) \qquad \text{(iii)}$$

Un exemple simple, mais valide, de *RCA* serait un rectangle avec la dimension de l'écran ($RCA \equiv W$), comme decrit en [CHH02], mais c'est le *RCA* avec le plus grand *Waste*. L'équilibre entre α_1 et α_2 de critère (iii) se reflète directement dans l'équilibre entre le *vertex pipeline* et le *pixel pipeline*. Le meilleur choix du α_1 et α_2 est dépendant de l'application, et peut être adapté pour décharger le pipeline quand un goulot d'étranglement est identifié.

La liste suivante classifie des *RCA* basés sur le genre de primitives basiques employées pour le construire (cf. Section 7.1.4 pour les détails):

- les faces d'un polyèdre;
- polygones;
- points et segments.

1.4.6 *Geometry Textures*

Pour des objets fortement triangulés (par exemple, le David de Michel-Ange), les triangles sont très petits une fois comparés à l'objet entier. Dans ce cas, nous pouvons dire que les triangles représentent en même temps les macro-structures et les méso-structures de l'objet. L'idée principale dans notre technique est d'employer un algorithme de visualisation approprié aux méso-structures mais appliqué à l'objet complet. Nous reconstruisons les détails géométriques au-dessus d'une approximation simple du modèle original.

Le problème central dans notre travail est différent de la visualisation de meso-structures. Nous nous sommes intéressés à reconstruire les détails d'un objet entier, qui est un modèle non-répétitif, puisqu'il change sur le domaine du modèle. Dans notre cas, l'entrée est un ensemble de carte des hauteurs converties du modèle original pour représenter l'information géométrique complète. La consommation de mémoire est une question importante dans cette situation, ainsi les techniques multidimensionnelles (cf. Section 5.5) sont prohibitives.

La *Geometry texture* est une représentation géométrique pour des surfaces. Son domaine est un hexahedron parallèle et dans son intérieur une carte des hauteurs représente la surface (Figure 1.5).

En fait, les *geometry textures* sont exactement notre primitive GPU de carte des hauteurs (Section 7.2), mais limités à un *RCA* parallélépipèdique. Le domaine n'est pas limité à un rectangle parfait, ainsi des échantillons vides sont prévus dans chaque carte des hauteurs.

Les cartes des hauteurs sont très compactes pour représenter la géométrie et, c'est la raison pour laquelle nous nous sommes intéressés à les employer. La construction d'un

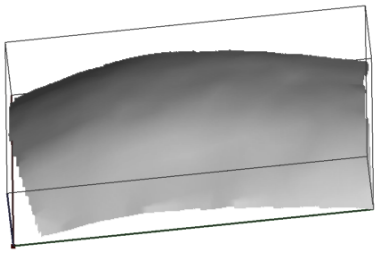


Figure 1.5: Example d'une geometry texture. Une carte des hauteurs est défini à l'intérieur du domaine du parallélépipède .

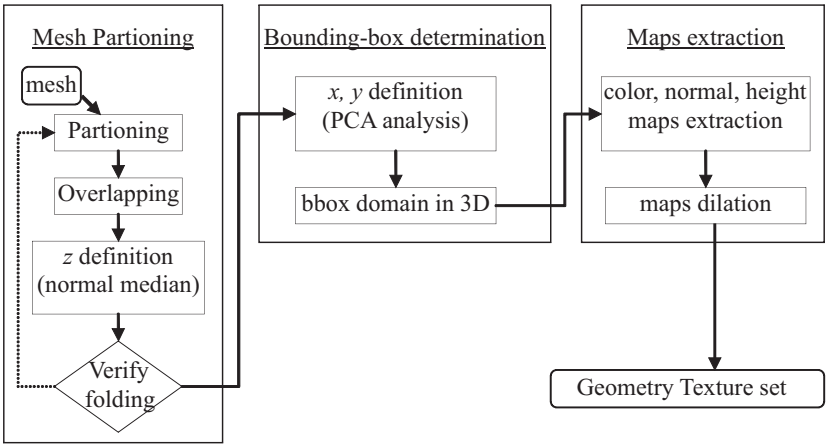


Figure 1.6: Procedure complète de convertion de maillages triangulaires en *geometry textures*.

ensemble de *geometry textures* à partir d'un modèle complexe, comme expliqué dans la Section 8.1, n'a aucune restriction dans la direction z (la direction des hauteurs) et chaque *geometry texture* est bien adaptée autour de l'extérieur de la surface (Section 8.2). En temps réel, les *geometry textures* emploient notre algorithme de lancer de rayons éxecutés par le *pixel shader* (Section 7.2). En conséquence, la géométrie est bien reconstruite, avec l'éclairage, l'auto-occlusion et la silhouette.

1.4.6.1 Convertion

Pour réaliser notre but, nous avons besoin d'une manière de représenter les objets naturels par des *geometry textures*. Nous divisons le domaine de surface d'objet en ensemble de morceaux, chacun lié à une *geometry texture*. Une fois que nous obtenons l'ensemble de *geometry textures* représentant l'objet complet, il est visualizer en utilisant notre algorithme de lancer de rayons en GPU.

L'entrée de notre procédé de conversion est un maillage polygonal (voir triangulaire) et la sortie est un ensemble de *geometry textures*.

La Figure 1.6 montre l'algorithme complet de convertion de maillages triangulaires en *geometry textures*.

Parmi les trois étapes de conversion, le *partitioning* est le plus difficile et critique pour le succès de l'algorithme. Nous avons testé deux stratégies différentes. Dans la Section 8.1.1 nous montrons les résultats de la division avec le PGP (*Periodical Global Parameterization* [RLL+05]) et dans la section 8.1.2 nous décrivons la méthode de division basée sur le VSA (Variational Shape Approximation [CSAD04]).

Les deux méthodes ont leurs propres avantages. Tandis que le PGP est meilleur pour réduire les espaces vides, le VSA est beaucoup plus efficace en évitant des morceaux

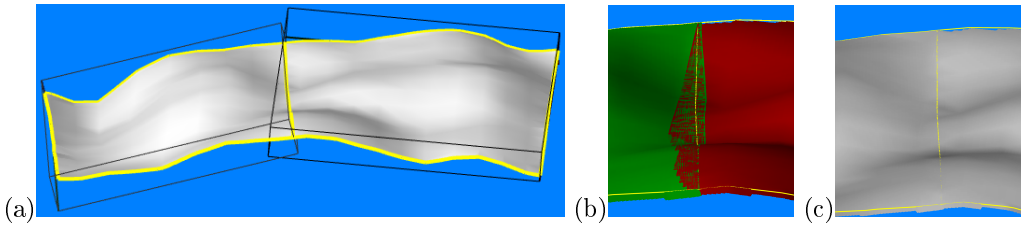


Figure 1.7: Rendu sans couture des *geometry textures* voisines.

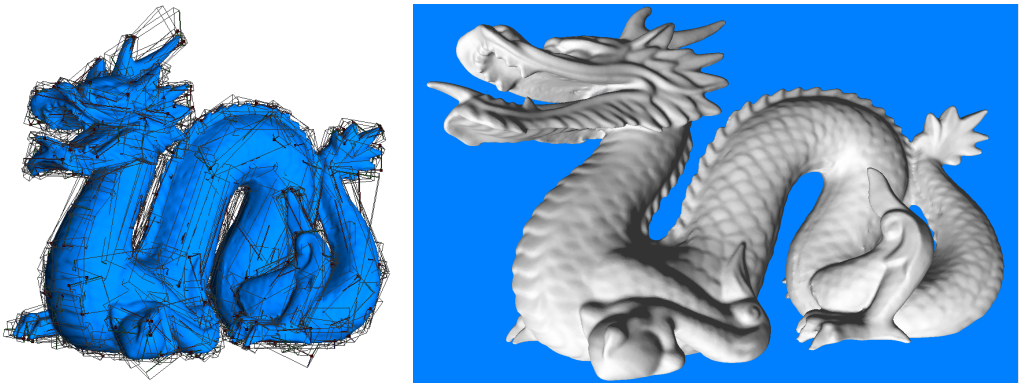


Figure 1.8: Rendu d'un modèle de dragon en utilisant des *geometry textures*.

auto-pliant. Comparant ces deux points, nous considérons le problème de l'auto-pliage plus important. Les morceaux auto-pliants ne peuvent pas être représentés par notre *geometry texture*, ce qui est une forte restriction. La réduction vide des espaces est juste une optimisation.

Basé sur ces faits, nous avons choisi VSA en tant que méthode de division.

1.4.6.2 Visualisation

Puisque nous visualizons un modèle comme composition de morceaux, une question importante est d'assurer la continuité de la reconstruction. Une jonction parfaite est rendue possible grace à deux détails dans notre technique.

Le premier a lieu au moment de la conversion : pour chaque élement un anneau supplémentaire des triangles des élements voisins est ajouté pour produire les *geometry textures*. Comme expliqué dans la Section 8.1, dans l'étape d'extraction des cartes (*maps extraction*), chaque *geometry texture* contiendra des informations de chevauchement sur ses voisins dans sa carte des hauteurs. La partie relative au chevauchement est nécessaire pour éviter des fissures et il est naturellement traité pendant l'affichage.

Le deuxième détail se rapporte à l'algorithme de visualisation qui génère l'information de z-buffer de manière correcte. Suivant les indications de la Figure 1.7, la continuité est également garantie par le calcul correct d'éclairage, basé sur les cartes des normales. L'information de profondeur par-pixel est également importante pour maintenir la compatibilité avec n'importe quel autre objet rendu dans la scène.

1.4.6.3 Performance

Nous avons employé le modèle du dragon pour les tests (Figure 1.8). À partir du modèle original composé de 871.414 triangles, nous avons produit trois ensembles de 453 *geometry textures*, changeant ainsi leur résolution maximum (128×128, 256×256 et 512×512). Les essais ont été effectués avec un Athlon XP 3800+ 2.4Ghz, 2GB RAM et une carte graphique GeForce 8800 GTX, 768MB.

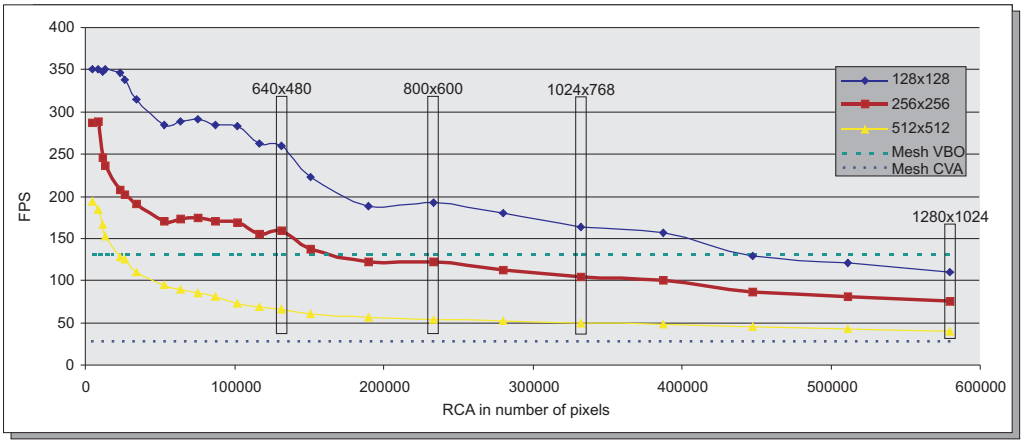


Figure 1.9: Performances du rendu du modèle de dragon avec des *geometry textures*.

Puisque les *geometry textures* on leur goulot d'étranglement localise a l'exécution de chaque pixel (cf. Section 7.3), nous avons changé le niveau de zoom pour les essais. La Figure 1.9 présente un graphique des résultats obtenus.

Part III: Modèles Industriels

De nos jours, les industries de pointe sont très consommatrices de ressources informatiques à toutes les étapes de la chaîne de production. Les transports, la santé et l'énergie sont les domaines industriels dans lesquels nous retrouvons les plus grands investissements en technologies telles que la simulation numérique, la visualisation et la réalité virtuelle. Il est fréquent que des réunions de personnel se déroulent dans des centres de réalité virtuelle où les équipes multidisciplinaires discutent des résultats de simulation ou modélisent de nouveaux produits par l'intermédiaire de dispositifs d'interaction tridimensionels.

Nous pouvons énumérer certains des domaines industriels qui utilisent la visualisation par ordinateur:

- l'industrie de transport ;
- l'industrie chimique ;
- l'électricité ;
- l'industrie du pétrole et du gaz.

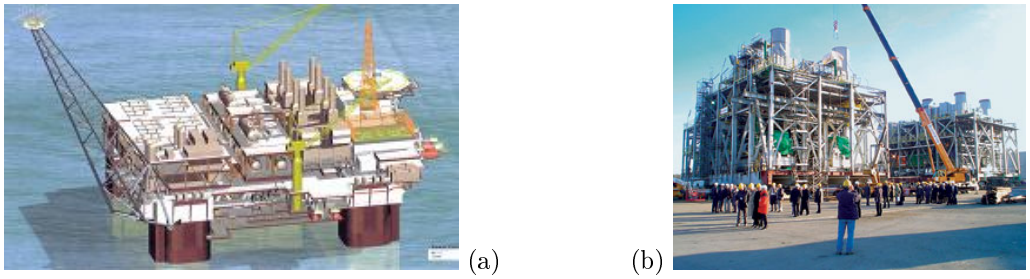
L'industrie du pétrole et du gaz et les sites industriels

Dans le domaine de l'industrie du pétrole, les activités principales de forage, d'exploration, de stockage, de transport, de transformation et de distribution emploient des équipements et des machineries complexes. Nous identifions trois classes : les machines complexes (par exemple les perforateurs et les *oilrigs*, cf. Figure 1.10a), les sites industriels (les raffineries, plateformes et bateaux, cf. Figure 1.10b) et les systèmes tubulaires (les oléoducs et les puits, cf. Figure 1.10c). Parmi ces trois classes, ce sont les sites industriels qui posent le plus grand défi pour la visualisation due aux données massives nécessaires pour représenter le modèle géométrique de l'infrastructure.

Par exemple, si nous considérons une base de données CAO (Conception Assistée par Ordinateur) d'un site industriel, la géométrie est décomposée en un ensemble de primitives géométriques. Ces primitives sont typiquement : des *primitives 2D* (carrés

(a) (b) (c)

Figure 1.10: Différents types d'objets manufacturés rencontrés dans le domaine du pétrole : (a) *oilrig*, (b) plateforme et (c) gazoducs. (Source d'images : `www.faculty.fairfield.edu`, `www.hoke.com` et `www.sufi.es`)



(a) (b)

Figure 1.11: La plateforme offshore P-52 de Petrobras.

et triangles) pour les murs, les planchers et les boîtes; des *cylindres* pour les systèmes tubulaires et les conduits; des *cônes* pour relier des tubes de rayon différent; des *coudes* (i.e., sections de tore) pour relier des tubes de direction différente. Un modèle typique contient environ un million de primitives.

Comme décrit dans la Section 4.1, le maillage triangulaire est une représentation géométriquement souple et son utilisation a été motivée durant les dernières années par les cartes de graphiques qui se basent essentiellement sur la rasterization de triangles. De ce fait, la géométrie des sites industriels est souvent maillée, résultant en un ensemble de plusieurs millions de triangles. Par conséquent, il est difficilement possible d'afficher interactivement tous ces triangles avec les cartes graphiques traditionnelles.

La construction des sites industriels implique beaucoup d'efforts et d'argent, son utilisation est stratégique pour beaucoup de compagnies internationales. Par exemple, Petrobras a récemment payé 900 millions de dollars pour la construction d'une nouvelle plateforme pétrolière, la P-52[2] (Figure 1.11). Il est prévu que cette plateforme produise environ 100.000 barrils par jour, l'équivalent de 150 millions de dollars par mois. La modélisation et la visualisation assistées par ordinateur font partie intégrante de cet effort.

Les modèles industriels (voir Figures 1.10b et 1.11) se composent principalement de plusieurs séquences de tuyaux, de lignes de câbles et d'autres réseaux techniques, qui sont fondamentalement des combinaisons des primitives simples (cylindres, cônes et portions de tore). Chaque logiciel a son propre format de données interne. Par exemple, un simple cylindre peut être stocké de différentes manières : comme deux points et un rayon, comme extrusion de cercle, comme révolution de segment ou même comme une

[2] Quelques détails à propos de la P-52 : 5000 emplois directs et 20.000 emplois indirects seront créés par sa construction. Elle fonctionnera dans des profondeurs d'eau d'approximativement 2000m dans le projet de développement du champ de Roncador *offshore*, Brésil. Plus d'information sur les sites suivants :
`www.bndes.gov.br/english/news/not930.asp`
`www2.petrobras.com.br/ri/ing/DestaquesOperacionais/ExploracaoProducao/Roncador.asp`

approximation polygonale. La manière la plus commune d'exporter des données est le maillage triangulaire (en fait, la seule commune à tous les logiciels). Une des raisons motivant l'usage des triangles comme format de données est le rendu puisque les cartes graphiques courantes sont optimisées pour l'affichage de triangles (cf. Section 4.2.2).

Cependant, la représentation des primitives simples par des approximations polygonales n'est clairement pas la meilleure solution. L'utilisation d'un maillage implique de faire un compromis entre d'une part la qualité et d'autre par la rapidité: augmenter la qualité demande plus de triangles et donc plus de calculs et inversement. Toutefois, un nombre excessif de triangles doit être évité pour de multiples raisons : consommation de mémoire pour le stockage, surcharge de transferts entre CPU et GPU et baisse de performances lors de l'exécution d'applications dont les limitations sont la géométrie.

Contribution de la thèse: Visualisation de Sites Industriels
L'objectif est de visualiser des modèles tubulaires massifs provenant dela modélisation CAO de sites industriels, avec une qualité de rendu élevée et ceci en temps réel sans approximation polygonale des primitives géométriques de base (Chapitre 11). La thèse apporte deux contributions principales:

- un algorithme pour récupérer des primitives géométriques implicites à partir d'une base de données maillée, avec une méthode d'*ingénierie inverse* (Chapitre 9).
- un algorithme de rendu qui emploie directement les équations des primitives géométriques implicites (ex: cylindres, cônes et tores), pour éviter leur triangulation (Chapitre 10);

Les avantages de notre méthode pour le rendu de structures tubulaires sont :

- **la qualité**, grâce à l'utilisation directe des équations des primitives, au lieu de l'approximation polygonale ;
- **la vitesse**, réalisée en codant notre algorithme directement sur les cartes graphiques programmables ;
- **la réduction significative de l'espace mémoire** pour la représentation des tuyaux. Il sera alors possible de gérer par conséquent de plus grandes bases de données par les cartes graphics et d'obtenir une visualisation en temps réel.

1.4.7 Récupération de Surface Implicite

Plusieurs domaines se servent de la *récupération de surface implicite*: reconstruction de surfaces, amélioration de géométrie, l'ingénierie inverse, reconnaissance basée sur le modèle et simplification de géométrie. Pour ces raisons, la récupération de surface implicite a été le sujet de plusieurs recherches [Pet02, WFAR99, YAH96, LMM97, WK05, OBA+03, OBA05]. Typiquement, l'entrée est composée d'un ensemble dense de points 3D obtenus après un balayage tridimensionnel de l'objet. Il existe aussi d'autres méthodes qui considèrent des maillages polygonaux comme information d'entrée comme dans notre cas.

Dans notre application, nous considérons les objets manufacturés. Habituellement, des objets composés par des pièces mécaniques peuvent être décrits par des plans, des cônes, des sphères, des cylindres et des surfaces toroïdales. Selon Nourse et al. [NHHM80] et Requicha et al. [RV82], 95% des objets industriels peuvent être décrits par ces pièces implicites. Ainsi, notre algorithme de récupération consiste à trouver les équations des surfaces implicites originales qui décrivent l'objet.

Petitjean [Pet02] énumère les quatre étapes principales de la récupération de surface:

- *estimation* : calcule la géométrie locale en utilisant des paramètres différentiels tels que la normale et la courbure ;
- *segmentation* : responsable de la division des données originales en sous-ensembles formant la plupart du temps une seule primitive géométrique;
- *classification* : cette étape décide dans quel type de surface un sous-ensemble devrait être inclus (cylindre, tore, cône ou autre);
- *reconstruction* : qui trouve les paramètres surfaciques correspondant aux données d'entrée.

1.4.7.1 Récupération de Surface – Méthode Numérique

Les modèles CAO industriels sont caractérisés par un faible nombre de points représentant la géométrie de chaque objet. Ceci cause un problème lors de l'étape de *classification* que nous nous attacherons à résoudre dans cette section.

Bien que quelques auteurs conseillent de faire une recherche pour chaque quadrique [FEF97, Pet02], nous avons préféré commencer par une recherche des 10 coefficients génériques des quadriques (cf. Équation 4.1 à la page 61) puis une classification de chaque surface selon son type (cylindre, cône, sphère, etc.).

Notre procédure de récupération des quadriques suit l'ordre suivant:

Etape I _ Ajustement à l'équation de la quadrique (méthode des moindres carrés)

Etape II _ classification de la quadrique (en évaluant les coefficients quadriques)

Etape III _ reconstruction (en extrayant des directions principales et en projetant les sommets du maillage d'origine)

Ajustement à l'équation de la quadrique

Le but est de trouver la surface quadrique Q paramétrisée par les 10 coefficients qui décrivent le mieux l'ensemble des points. Pour un ensemble de n points 3D $p_i = (x_i, y_i, z_i)$, nous voulons la solution du système d'équations linéaires:

$$f(x_i, y_i, z_i) = 0, \quad 0 > i \geq n \tag{1.1}$$

Pour les nuages denses de points, n est beaucoup plus grand que le nombre de variables (9 pour des quadriques) et le système ci-dessus est surdéterminé. Le système est résolu en utilisant la méthode des moindres carrés pour réduire au minimum la distance entre les points et la surface.

Nous pouvons écrire le système ci-dessus sous forme matricielle ($\mathrm{A}x = b$):

$$\begin{bmatrix} {x_1}^2 & 2x_1y_1 & 2x_1z_1 & 2x_1 & {y_1}^2 & 2y_1z_1 & 2y_1 & {z_1}^2 & 2z_1 \\ {x_2}^2 & 2x_2y_2 & 2x_2z_2 & 2x_2 & {y_2}^2 & 2y_2z_2 & 2y_2 & {z_2}^2 & 2z_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ {x_n}^2 & 2x_ny_n & 2x_nz_n & 2x_n & {y_n}^2 & 2y_nz_n & 2y_n & {z_n}^2 & 2z_n \end{bmatrix}_{n\times 9} \begin{bmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ I \end{bmatrix}_9 = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}_n$$

Nous voulons réduire au minimum l'erreur résiduelle $r = \mathrm{A}x - b$, ainsi nous résolvons x pour l'équation des moindres carrés $\left[\mathrm{A}^T\mathrm{A}\right]_{9\times 9}x = \left[\mathrm{A}^T b\right]_9$. Ceci peut être fait en utilisant la factorisation de Cholesky. (A noter que le 10^{me} coefficient quadrique (J) est

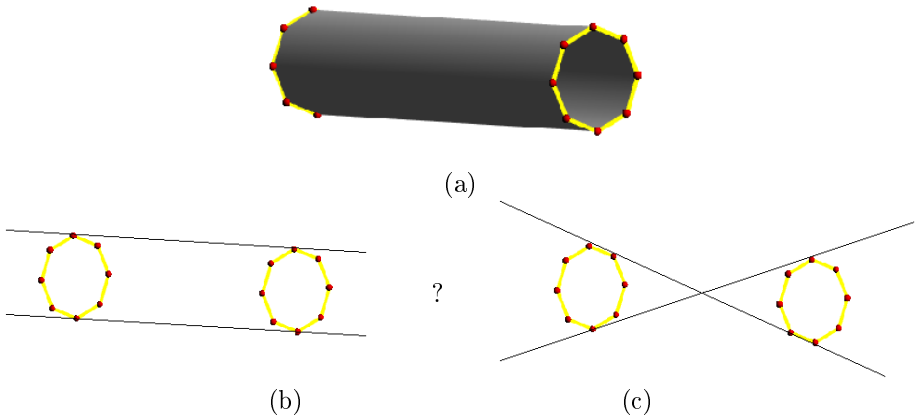


Figure 1.12:
(a) Les 16 sommets (en rouge) d'un cylindre d'un modèle CAO sont utilisés pour réaliser l'ajustement à l'équation du quadrique. Le faible nombre de points peut causer de l'imprécision dans la récupération surfacique. Ceci peut avoir comme conséquence une ambiguïté dans la détermination du type de quadrique. Par exemple, les sommets peuvent s'insérer dans un cylindre (b) ou dans un cône (c).

une constante. Ainsi, fixer que chaque équation soit égale à 1 est équivalent à prendre $J = -1$.)

Cependant, dans notre application le nuage de points n'est pas dense. Par exemple, un cylindre peut avoir 12 sommets dont certains sont alignés entre eux. Par conséquent, le système devient sous déterminé avec beaucoup de solutions possibles.

Dans la Figure 1.12 nous prouvons qu'un cylindre ou un cône peut s'adapter aux sommets d'un cylindre typique de CAO. Par conséquent, nous avons besoin de plus d'équations pour fournir plus d'informations indépendantes à notre système. Pour y arriver, nous avons ajouté des équations supplémentaires pour ajuster la normale de la quadrique à la normale $\vec{n_i}$ de chaque sommet.

Ainsi, pour chaque sommet, trois nouvelles équations (correspondant aux composants x, y et z de la normale) ont été insérées dans le système en respectant la règle suivante:

$$\vec{n} = \left[\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}\right] \tag{1.2}$$

Avec cette amélioration, le nombre d'équations est de $4n$ au lieu de n et nous pouvons mieux converger vers une solution unique (dans le sens des moindres carrés). Le nouveau système des équations devient :

$$\begin{bmatrix}
{x_1}^2 & 2x_1y_1 & 2x_1z_1 & 2x_1 & {y_1}^2 & 2y_1z_1 & 2y_1 & {z_1}^2 & 2z_1 \\
2x_1 & 2y_1 & 2z_1 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 2x_1 & 0 & 0 & 2y_1 & 2z_1 & 2 & 0 & 0 \\
0 & 0 & 2x_1 & 0 & 0 & 2y_1 & 0 & 2z_1 & 2 \\
{x_2}^2 & 2x_2y_2 & 2x_2z_2 & 2x_2 & {y_2}^2 & 2y_2z_2 & 2y_2 & {z_2}^2 & 2z_2 \\
2x_2 & 2y_2 & 2z_2 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 2x_2 & 0 & 0 & 2y_2 & 2z_2 & 2 & 0 & 0 \\
0 & 0 & 2x_2 & 0 & 0 & 2y_2 & 0 & 2z_2 & 2 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
{x_n}^2 & 2x_ny_n & 2x_nz_n & 2x_n & {y_n}^2 & 2y_nz_n & 2y_n & {z_n}^2 & 2z_n \\
2x_n & 2y_n & 2z_n & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 2x_n & 0 & 0 & 2y_n & 2z_n & 2 & 0 & 0 \\
0 & 0 & 2x_n & 0 & 0 & 2y_n & 0 & 2z_n & 2
\end{bmatrix}_{4n\times 9}
\begin{bmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ I \end{bmatrix}_9
=
\begin{bmatrix}
1 \\
\vec{n_1}.x \\
\vec{n_1}.y \\
\vec{n_1}.z \\
1 \\
\vec{n_2}.x \\
\vec{n_2}.y \\
\vec{n_2}.z \\
\vdots \\
1 \\
\vec{n_n}.x \\
\vec{n_n}.y \\
\vec{n_n}.z
\end{bmatrix}_{4n}$$

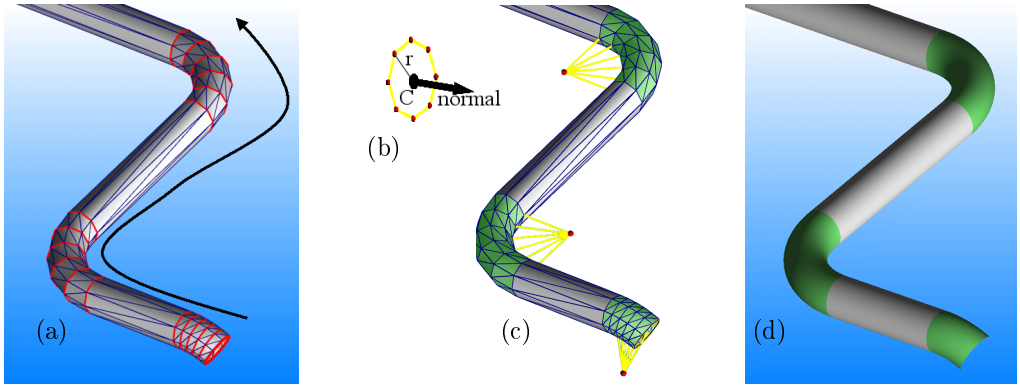


Figure 1.13: (a) Identification des anneaux en parcourant le maillage. (b) On estime les propriétés de chaque anneau: le centre C, le rayon r et le vecteur normal. (c) La segmentation et la classification sont réalisées en évaluant les anneaux consécutifs. (d) La reconstruction des primitives évoluées est obtenue en évaluant les propriétés de l'anneau.

Une fois de plus, nous voulons réduire au minimum l'erreur résiduelle du système d'équations ci-dessus. Nous pouvons résoudre $\left[\mathrm{A}^T\mathrm{A}\right]x = \left[\mathrm{A}^T b\right]$ avec la factorisation de Cholesky.

L'insertion des équations des normales a amélioré la procédure de récupération. Les résultats montrés à la Section 9.3 ont été obtenus exclusivement avec les maillages disjoints. Cependant, pour des données non segmentées, un problème lié à la normale peut apparaître (cf. Section 9.3).

Nous avons observé à la Section 9.2 que la régularité est une caractéristique intéressante pour ce type de données d'entrée. Pour cette raison, dans la prochaine sous-section nous expliquons l'algorithme adapté à la spécificité des données que nous avons conçu.

1.4.7.2 Récupération de Surface – Méthode Topologique

Notre algorithme est exclusivement destiné aux systèmes tubulaires, composés de cylindres, de portions de cônes et de coudes (les coudes correspondent aux portions de tore, comme montré en vert sur la Figure 1.13). L'idée est de rechercher toutes les sections circulaires régulières (les anneaux) en tenant compte de leurs connexions. En vérifiant les anneaux consécutifs nous pouvons trivialement déterminer s'ils correspondent à l'une de ces trois primitives évoluées. La vitesse d'exécution est l'un des avantages de cette approche en comparaison avec les approches numériques plus traditionnelles. Les approches numériques sont très appropriées pour des données brutes (obtenues par exemple par balayage). Dans notre cas, nous exploitons le fait que les données sont presque constamment régulières et peuvent être facilement parcourues en exécutant simplement un algorithme "couper-trouver".

Notre méthode est basée sur un parcours du maillage selon les étapes suivantes:

1. **détection de l'anneau** (cf. Figure 1.13(a)): les anneaux sont identifiés dans le maillage comme un ensemble de sommets coplanaires reliés en boucle, équidistants de leur centre.

2. **segmentation et classification des tuyaux** (cf. Figure 1.13(c)): les anneaux sont regroupés en primitives selon leur rayon et leur vecteur normal:

   - cylindres: correspondant à un couple d'anneaux de même normale et de même rayon;
   - tranches de cône: correspondant à un couple d'anneaux de même normale mais de rayon différent;

- coudes: correspondant à un ensemble d'anneaux de même rayon, et dont les co-normales convergent en un même point (voir les lignes jaunes sur la Figure 1.13(c)). Nous travaillerons avec des portions de tores jusqu'à 180 degrés, dans le cas contraire ils seront coupés en deux.

3. **reconstruction de primitives** (voir Figure 1.13(d)): une fois que l'ensemble des anneaux correspondant à chaque primitive a été individualisé, il devient facile de trouver les paramètres de la primitive à partir des centres et rayons des anneaux. Les centres des sections extrêmes de chaque primitive (P_1 et P_2) sont simplement les centres du premier et du dernier anneau. Les rayons sont aussi déterminés à partir des rayons des anneaux. Le centre C des coudes est l'intersection des co-normales. Chaque primitive est alors déterminée par un petit ensemble de paramètres:

   - cylindre: P_1, P_2, r
   - cône: P_1, P_2, r_1, r_2
   - coude: P_1, P_2, C, r

Après l'exécution de notre algorithme de récupération de surface, nous obtenons un ensemble de primitives implicites (cylindres, cônes, coudes) et un ensemble de polygones pour les autres objets.

Les essais faits avec la méthode topologique ont démontré son efficacité. Les points positifs de la méthode topologique sont la simplicité et la vitesse d'exécution. L'implémentation est simple et robuste pour les maillages réguliers de CAO. Quelques minutes suffisent pour récupérer entièrement les primitives implicites à partir des données d'une centrale électrique (13 millions de triangles), sans compter le temps de chargement du modèle en mémoire.

il y a cependant quelques limitations dans l'approche topologique; cette méthode extrait seulement trois types de primitives implicites: cylindres, tranches de cônes et tores. Considérant que son application est intéressante pour les systèmes tubulaires, elle ne peut pas être appliquée à d'autres genres de structures que nous rencontrons également dans les modèles industriels de CAO. Par exemple, l'algorithme ne reconnait pas d'autres surfaces implicites, telles que des sphères et des cônes complets. Un autre problème lié à la reconnaissance de maillage de CAO est l'ambiguïté. Sans aucune information additionnelle, nous ne pouvons pas nous assurer qu'un objet maillé est une approximation d'une pièce arrondie ou une représentation exacte de l'objet original.

1.4.8 Visualisation de Primitives Implicites

Dans notre solution, nous visualisons des primitives graphiques spéciales en utilisant directement leurs équations pour l'affichage. Nous visualisons quelques surfaces implicites en implémentant le code directement dans la carte graphique. Pour cette raison nous les appelons *les primitives GPU*. Cette approche améliore la qualité, augmente la vitesse et de réduit l'utilisation de la mémoire lors de la visualisation des modèles industriels complexes.

Dans le Chapitre 10 nous expliquons les détails de l'exécution de chaque primitive GPU implicite (Sections 10.1 à 10.7). Dans le Chapitre 11, nous montrons les résultats pour la visualisation de sites industriels.

- *Quadriques Génériques* (Section 10.1) La primitive GPU pour des quadriques génériques emploie 8 sommets définissant la boîte 3D englobante autour de la surface quadrique (RCA_{Bbox}). Les 10 coefficients définissant l'équation quadrique (voir l'équation 4.1) sont passés comme paramètres au *vertex shader*. Les coordonnées locales des sommets sont $[\pm 1, \pm 1, \pm 1]$, cf. Figure 1.14(a). Nous avons pris la décision de fixer le repère local puisque les coefficients quadriques peuvent être ajustés pour inclure n'importe quelle opération linéaire (translation, rotation ou homothétie).

- *Les sphères* (Section 10.2) Nous avons examiné deux classes différentes de RCA pour le lancer de rayons des sphères: $RCA_{polydre}$ et $RCA_{billboard}$. Pour le $RCA_{polydre}$, nous avons utilisé un cube et un icosahedron pour réaliser quelques tests. Nous avons décidé d'employer le RCA_{point} en tant que notre *billboard*, puisqu'il a un nombre de sommet très bas comparé au polyèdre $RCA_{polydre}$.

- *Les ellipsoïdes* (Section 10.3)

- *Les cylindres* (Section 10.4) Pour les cylindres, un *billboard* à quatre faces est le meilleur choix de *RCA*. Il a un coût de calcul de sommets très faible, $V(RCA_{Billboard}) = 4$, et avec un bon agencement de calculs, il a une projection serrée englobant la projection du cylindre (voir Figure 1.14(c)), ce qui réduit la fonction de perte (voir Section 7.1.4).

  Le *vertex shader* calcule la position des sommets en se basant sur les dimensions du cylindre de telle sorte que les quatre sommets forment à chaque fois une enveloppe convexe parfaite (Figure 1.14(c)). Une fois la position des sommets calculée dans le repère du monde est calculée dans le *shader*, le CPU n'a plus besoin de passer les coordonnées réelles. De manière similaire, le système de coordonnées locales peut être déduit directement de la direction principale du cylindre et des directions des axes de l'enveloppe convexe calculée. Tout ceci réduit la quantité nécessaire d'information à envoyer par l'application C'est une sorte de *système de coordonnées dépendant de vue* où l'un des axes est fixé relativement par rapport au monde (l'axe principal du cylindre) et les autres axes dépendent de la direction de la vue. Au final, l'unique information par sommet nécessaire est la position relative de l'enveloppe convexe (devant/derrière et gauche/droite).

- *Les cônes* (Section 10.5) La *RCA* utilisée pour les cônes est un polyèdre dont les faces visibles sont rasterisées [[so the PS turns in]] autour de l'image finale. Les portions de cônes utilisent une boîte englobante avec un coût de sommet $V(RCA_{Bbox}) = 8$ (voir figure 1.14(d)). Pour un cône entier, nous avons choisi une pyramide comme polyèdre qui a un coût de sommet $V(RCA_{Pyramid}) = 5$ plus petit que les boîtes englobantes (voir Figure 1.14(e)).

- *Les cubiques* (Section 10.6) Nous avons étendu l'idée des primitives GPU qui ont été développées pour les quadriques pour les surfaces implicites cubiques. Il est possible de former des surfaces très intéressantes en utilisant les équations cubiques (voir 1.14(f)), mais ces objets ne sont pas aussi populaires que les quadriques classiques (ex : sphères, cylindres, cônes). Comparés aux quadriques, les calculs sont largement plus intensifs réduisant ainsi le nombre d'images par seconde. En fait, la solution consiste à essayer d'avoir des primitives GPU cubiques qui sont un peu plus proches du tore qui lui a une équation quartique.

- *Les tores* (Section 10.7) Trouver le zéro d'une équation quartique pour plusieurs pixels dans des temps interactifs n'est pas une tâche facile. Nous avons essayé quatre approches différentes et à partir des résultats, nous avons choisis un algorithme adéquat pour notre tore GPU.

  - Sturm
  - *Bissection par double dérivée* (**Contribution de la thèse:** Nous avons proposé et étudié un nouveau solveur de racine qui est une extension de l'algorithme de bisection simple.)
  - *Sphere Tracing*
  - La methóde de Newton-Raphson

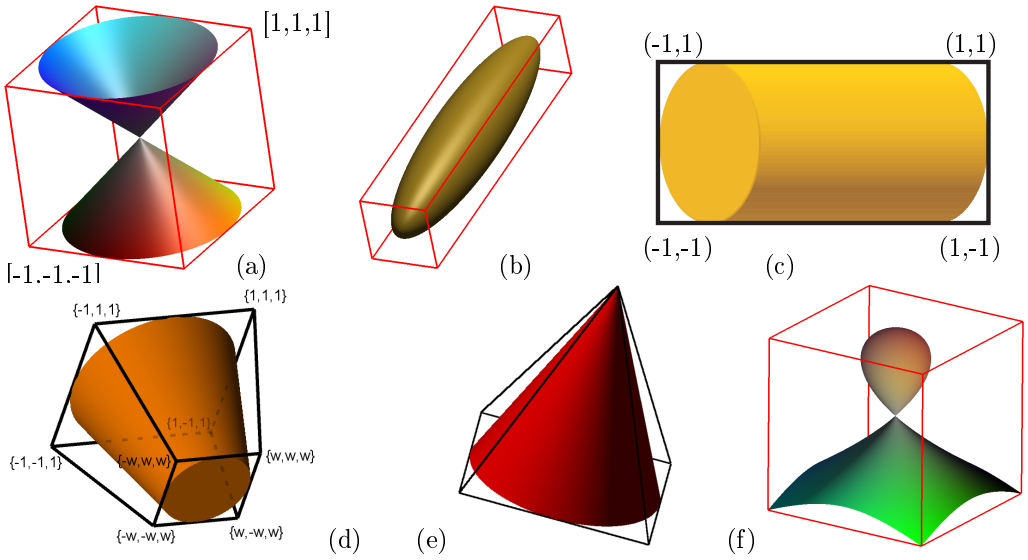


Figure 1.14: (a) Quadric generique, (b) ellipsoïdes, (c) cylindre, (d) tranche de cône, (e) cône complet et (f) cubique.

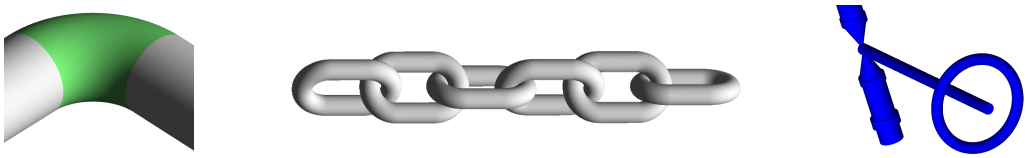


Figure 1.15: Examples de tores dans les sites industriels.

L'algorithme de Newton-Raphson a donné les meilleures performances pour notre Tore GPU. Nous avons utilisé un seuil pour stopper les itérations de telle sorte que l'erreur est assurée d'être inférieure à 0.1% du petit rayon r ($\epsilon(r) \leq 0.001$). Nous avons aussi testé $\epsilon(r) \leq 0.00003$. Les résultats sont présentés dans le Tableau 10.2 de la page 186

- *Portions de Tore* (Section 10.8) Dans les sites industriels, les tores sont présents dans les jonctions, les chaînes et quelques modèles CAO (voir figure 1.15). En fait, dans la plupart des cas, les objets contiennent uniquement une partie de tore. Bien que l'algorithme de lancer de rayon pour la portion de tore est le même que pour celui d'un tore complet (excepté pour un teste d'angle), nous suggérons un traitement spécial pour déterminer le *RCA* pour le cas de portion de tore (voir la Section 7.1.4 au sujet de *RCA*). En d'autres termes, le *pixel shader* pour les tores complets et pour les portions de tore sont fondamentalement identique, toutefois les *vertex shader* ont différentes implémentations.

1.4.9 Résultats pour la visualisation de sites industriels

Notre but est de représenter et d'afficher la plupart des surfaces d'un site industriel avec nos primitives GPU en ayant trois objectifs principaux: **qualité**, **performance** et **économie de mémoire**. Contrairement à la solution polygonale, les primitives GPU n'ont besoin d'aucune gestion de LOD puisqu'elles gardent leur qualité même lorsqu'elles sont vues avec un *zoom* important et nécessitent peu de calculs quand elles sont visualisées de loin. Cependant, il est important de noter qu'avec notre algorithme d'ingénierie inverse

nous récupérons environ 90% des objets dans ces grands modèles, tandis que les autres objets utilisent des techniques conventionnelles de visualisation (*rasterisation*).

Dans le Chapitre 11 nous discutons en détails des trois avantages mentionnés ci-dessus.

Amélioration de la qualité d'image

Nous listons ici cinq améliorations de la qualité d'image obtenues avec notre méthode.

Silhouette lisse. En maillant des surfaces incurvées, il n'est pas possible d'obtenir des silhouettes complètement lisses puisque nous sommes limités par la discrétisation des mailles. Les primitives GPU sont visualisées par pixel et ainsi les silhouettes sont toujours lisses, même après un agrandissement énorme à l'écran. Voir Figure 1.16(a).

Intersections (calcul de profondeur par pixel). Avec la rasterisation de triangles, la profondeur de chaque pixel (*z-value*) est le résultat de l'interpolation de la profondeur des sommets. D'autre part, les primitives GPU calculent la profondeur par pixel, qui est beaucoup plus précise. Quand deux primitives GPU ou plus s'intersectent, la frontière a une forme correcte due à ce calcul précis de visibilité. Voir la Figure 1.16(b) pour avoir une comparaison.

Continuité entre primitives d'un même tuyau. Dans les modèles de sites industriels, les tuyaux sont une longue séquence de primitives (cylindres, joints et cônes). Avec la solution polygonale, un risque de fissures entre les primitives peut apparaître. Pour éviter cela, les maillages devraient garder la même résolution et le même alignement pour deux primitives consécutives. Dans la Figure 1.16(c), les fissures apparaîtraient à cause d'un mauvais alignement. Ce genre de situation indésirable est également un problème pour des applications utilisant LOD pour visualiser des tuyaux. Dans ce cas, il est difficile de garantir la continuité (voir [KBO+99]). Avec les primitives GPU, celle-ci est garantie sans effort puisqu'il n'y a aucune discrétisation.

Per-pixel shading. Dans l'implémentation par défaut de la rasterisation de triangles, le GPU traite la couleur par sommet pour l'interpoler plus tard à l'intérieur du triangle projeté. Cette technique est appelée *Gouraud shading* [Gou71a, Gou71b]. Bien que rapides, les images finales ne sont pas d'aussi bonnes qualités, particulièrement autour des réflexions spéculaires (voir la Figure 4.6 à la page 69). Nos primitives calculent la nuance par pixel (*Phong shading*), sans aucune interpolation, améliorant la qualité d'image.

Contrôle de l'épaisseur des cylindres . Le contrôle de l'épaisseur, adopté dans le *vertex shader* pour les cylindres GPU, évite le rendu *pointillé* lorsqu'un cylindre haut et fin est visualisé de loin. De plus, quand ce contrôle est associé au calcul de couleur basé sur la moyenne normale, il réduit de manière significative l'effet de crénelage pour un groupe de cylindres parallèles (voir la Figure 10.14 à la page 175).

Espace mémoire

La récupération de surfaces topologiques convertit 90% des données originales de modèles industriels. L'espace mémoire des données récupérées est réduit à au plus 2% (98% de réduction). C'est une conséquence de la simplicité des informations implicites utilisées pour les primitives récupérées. Si nous considérons les 10% restant des triangles non récupérés, des modèles industriels tels que des plateformes pétrolières et des centrales électriques peuvent être stockés dans environ 15% de la taille originale des données.

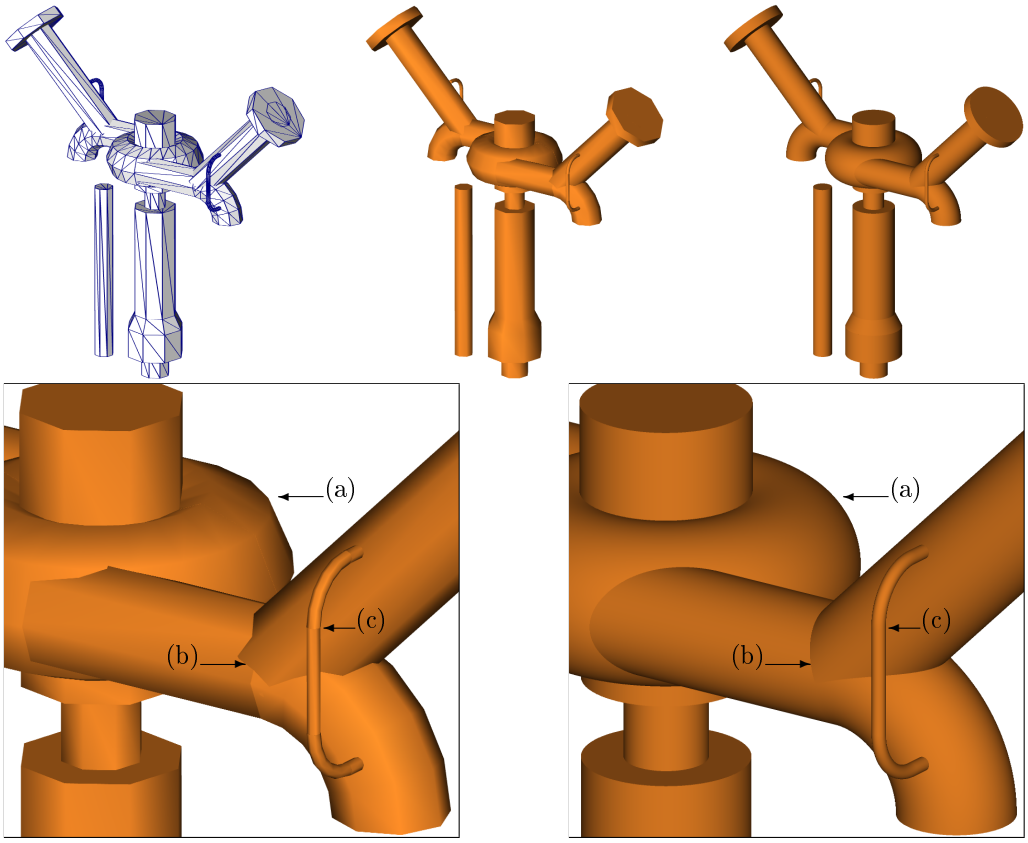


Figure 1.16: Une pièce industrielle d'une centrale électrique. *Top left:* Modèle original maillé. *Top center:* Rasterisation de triangles avec *Gouraud shading*. *Top right:* Visualisation avec les primitives GPU. *Bottom:* Triangles versus GPU primitives. **(a)** Lissage de la silhouette; **(b)** Précision d'intersection; **(c)** Continuité entre séquences de primitives.

Performance

Ce que nous voulons comparer est la meilleure solution obtenue avec notre méthode pour des primitives GPU et la meilleure solution obtenue par la rasterisation de triangles (deuxième et troisième colonnes dans le Tableau 11.2 à la page 196). En conclusion, les primitives GPU sont beaucoup plus rapides (vitesse presque doublée) comparées à la meilleure solution de rasterisation.

Dans le Tableau 11.3 à la page 197, nous avons une situation différente, où certains des triangles originaux (10%) n'ont pas été convertis en primitives implicites, nous les appelons UT (*Unrecovered Triangles*). Par conséquent, nous comparons le meilleur rendu de la rasterisation de VBO (troisième colonne) au rendu de la méthode hybride (des primitives GPU + UT, deuxième colonne). Comme décrit dans le Tableau 11.3, la solution hybride est entre 40% et 60% plus rapide que la rasterisation.

1.4.10 Conclusion

Le but de notre travail est d'accélérer les méthodes de visualisation pour obtenir un rendu interactif de modèles massifs.

Dans la Partie I, nous avons passé en revue la littérature sur les méthodes de visualisation selon leur échelle d'application.

Nous pouvons dire d'une façon simple que les algorithmes avancés pour les niveaux scénique et macroscopique ont comme objectif une exécution de haute-performance avec

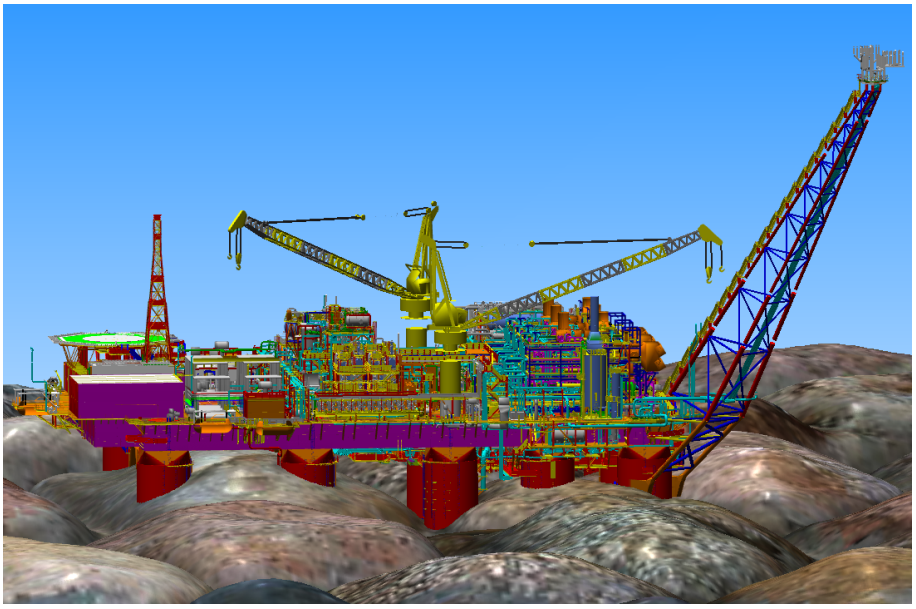


Figure 1.17: Visualisation Hybride

une faible perte en qualité. D'autre part, les algorithmes avancés pour les niveaux mésoscopique et microscopique ont pour but d'augmenter la qualité tout en essayant de réduire au minimum la vitesse perdue.

Nous avons concentré nos contributions au niveau macroscopique en présentant de nouvelles représentations surfaciques et de nouvelles primitives de visualisation (Chapitres 7 et 10). Nous avons prouvé que notre technique peut également être employée pour le niveau mésoscopique (voir la Section 5.3.3).

Dans notre approche, les modèles massifs sont classés dans deux catégories différentes: les modèles naturels et les modèles industriels. Pour chacun d'eux, nous proposons une nouvelle représentation en implémentant des algorithmes de conversion et en développant des primitives GPU.

Visualisation Hybride

Une de nos contributions importantes était d'établir un nouveau système de visualisation qui combine le lancer de rayons et la rasterisation des objets dans la même scène. Comme expliquée dans la Section 7.1, la clef pour une combinaison sans accroc de ces deux techniques consiste à prolonger un peu l'algorithme de lancer de rayons afin de respecter les règles de *z-buffer*.

Dans la Figure 1.17 nous montrons la combinaison de trois types de primitives. Les primitives implicites de lancer de rayons (cylindres, cônes et tores), un objet naturel de lancer de rayon (les cailloux) et les maillages rasterisées de triangles (pour d'autres objets de plateformes). En raison du rendement correct des valeur z par pixel, tous ces différents objets (avec différentes méthodes de rendu) sont parfaitement combinés pour la visualisation interactive.

Chapter 2

Introduction

2.1 The Challenge

Visualization of massive objects and scenes is a challenging topic in computer graphics, especially if we consider interactive applications. Two conditions are expected from these applications: speed (to provide a real-time feeling to the user, the images must be generated at a minimum rate, generally between 10 and 30 frames per second); and quality (realistic images are considered more and more as an essential condition, particularly in simulators).

Several applications in scientific visualization deal with massive models. The oil exploration industry, for example, is a domain in which the data sets are notoriously massive. Another challenge we find in this area is the heterogeneity of objects and scenes: terrain surfaces, seismic data, GIS (Geographic Information Systems) data, complex engineering models (e.g. platforms, refineries), oil wells with logs, oil and gas pipelines/ducts. Sometimes, extra information associated to each of these elements must be visualized: annotations, identifications, technical descriptions, and instantaneous conditions.

These various types of objects can be categorized into two main families: natural ones and manufactured ones. Despite this distinction, both types commonly have their surfaces represented by triangle meshes. There are some reasons for choosing triangles as the geometric primitives to be used: any object surface can be approximated to a triangle mesh; any polygon can be triangulated; triangle vertices are coplanar; triangles are convex; barycentric coordinates can be used as an unambiguous rule to interpolate vertex values in the triangle domain; and graphics cards are specialized in rasterizing triangles. However, representing every kind of object with triangle meshes might not be the most suitable choice, for reasons of efficiency and visualization quality.

Manufactured Objects

We consider as manufactured objects those projected digitally. In general, CAD (Computer Aided Design) models are composed by primitives described by simple math equations. Thus, their surface have precise boundary determination.

We concentrate our efforts in the visualization of industrial plants. Their data are especially massive, including millions of small objects, all of them frequently described by triangles. Industrial models, such as oil platforms, are mainly composed by large sequences of pipes, cable trays and other technical networks, which are geometric combinations of simple primitives (cylinders, cones and torus slices). Several problems appear when using triangle meshes for these kinds of primitives: (i) the representation is always an approximation, never perfect; (ii) the more we use the triangles, the more we enhance the approximation, but more computation (and memory space) is required by the rasterization algorithm; (iii) although a clever shading could beautify the rendering, there

is always a limitation on the silhouette of curved objects (especially after a zoom).

Our solution for manufactured objects is composed of two steps:

(i) As the data is commonly available as triangular meshes, a reverse engineering algorithm retrieves the higher-order primitives (cylinders, cones and tori) from the triangular mesh. Our algorithm is highly efficient because it explores the specific nature of the data (i.e. large number of tubular structures).

(ii) Once the high order primitives are recovered, we display them by using their exact equation, instead of a polygonal approximation. This is done by means of a ray-casting algorithm implemented directly in the graphics card. The results are images with very good quality (without silhouette problems and pixel-precise intersections).

Natural Objects

Natural objects are described by organic forms. In general they are not simple to describe mathematically. Since they were not projected digitally, it is necessary a method to acquire their information. There are many different ways to acquire natural objects: scanning, photographic stereo acquisition, seismic surveys. After aquisition we obtain a set of points that are processed to reconstruct the objects surfaces. Usually these surfaces are represented by triangle meshes. For rendering, this representation can explore graphics cards specialized in rasterizing triangles. However, mesh operations depends on random memory accesses. The use of regular grids to represent surfaces could allow efficient random access and other operations as composition and compression. Gu et al. [GGH02] proposed Geometry Images. The object is decomposed into a set of regular grids (2D arrays of 3D vertices). However, even if this representation is more compact than the initial mesh, it still consumes many resources at rendering time (see Section 4.4 for more about geometry images). To improve rendering performance, we propose a new representation, called *Geometry Textures*.

A geometry texture is composed by a bounding box, a height map and a normal map. The algorithm does not render the surface with triangles. Instead, a per-pixel ray-casting algorithm with the correct output depth is used. An algorithm is developed to convert a complex mesh model into a set of geometry textures. Models represented by patches of geometry textures are efficiently rendered seamlessly while keeping compatibility with standard GPU primitives. By uncharging the vertex pipeline, we obtain more efficiency than the traditional rendering method.

We have explored in detail the two types of objects (manufactured and natural) and designed new solutions, based on alternative representations. We have explored the programmable graphics cards to achieve interactive rendering for massive models. In the next section we briefly describe some visualization concepts followed by a section about the contributions of this thesis.

2.2 Computer Graphics and Visualization

Our research belongs to the *visualization* discipline. The main goal of *visualization* is transforming data into images. However, this description almost matches the ISO (International Standards Organization) definition for the whole computer graphics domain: "methods and techniques for converting data for a graphics device by computer". We prefer to classify the computer graphics domain in a wider sense, including other topics such as *image processing* and *computer vision*. We use Gomes and Velho's [GV97] classification based on the types of input and output of each discipline. There are four disciplines whose input and output are either data (geometry) or images (see Figure 2.1):

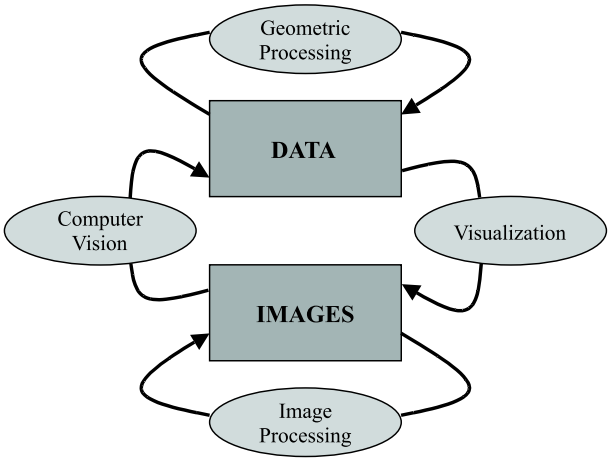


Figure 2.1: Computer Graphics disciplines (see [GV97]).

(Data ⇒ Data) Geometric Processing, also called *Data Processing* or *Geometric Modeling*, determines how model representation is stored and accessed in the computer. New structures, attributes and properties can be computed by processing model information. Examples of subjects in this domain are: curvature computation, remeshing, hierarchical structures, parameterization, convex hull computation, etc. The geometric processes do not necessarily have visualization as the main goal, but visualizing is an essential tool for verifying the results after processing.

(Data ⇒ Image) Visualization is the main goal in computer graphics. From stored data, visualization algorithms produce the images displayed in a graphics output device.

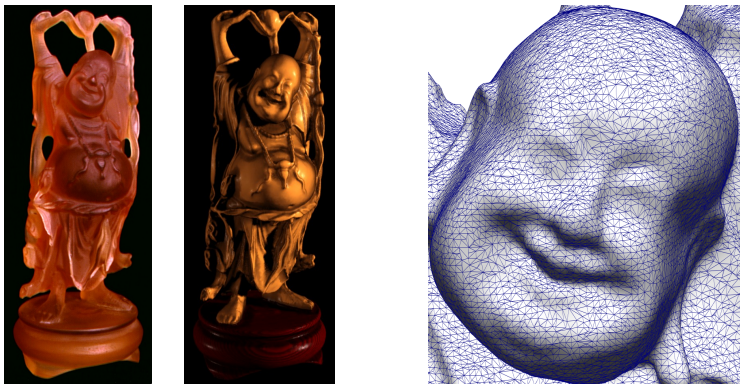
(Image ⇒ Image) In **Image Processing**, the input and the output are images. All kinds of operations that can be done on images are included in this discipline, such as blurring, sharpening, noise removal, etc. Sometimes image processing is used as a postprocessing for visualization or as a preprocessing for computer vision.

(Image ⇒ Data) The goal of **Computer Vision** is to automatically extract information from an input image. There are several applications such as robotics, OCR (Optical Caracter Recognition) and medical support.

In the following, we introduce some main points regarding visualization. Actually, we can say that visualization concerns two main topics: the visualization itself and the data *representation*. In Subsection 2.2.1, we discuss how to represent objects and why this is a complex and important issue. In Subsection 2.2.2, we briefly describe basic visualization and rendering algorithms (this topic will be deeply explored later in Section 3.1), responsible for the visibility tests. Visualization is actually a vast subject, but we are especially interested in *interactively* visualizing *massive scenes* and *complex objects*. For this reason we discuss more about them in the end of this section.

2.2.1 Object Representation

In 3D visualization we want to reproduce, through a two-dimensional image, objects and scenes as the human eye would see them (even if they do not exist in the real world). To achieve this goal many different tasks are necessary including a way to mathematically and computationally represent the objects. This representation must describe the geometry of the target object. However, describing the shape is not enough to entirely represent an object, since additional information about its appearance is also necessary. For example, two objects can have exactly the same geometry and topologie but different colors or even completely different materials (e.g., rubber and metal), see Figure 2.2(a).



(a) (b)

Figure 2.2: (a) Two distinct objects with different appearances; (b) although the same geometry.

Generally, to describe the geometry in $\mathbb{R}^3$, the objects are decomposed into surfacic or volumetric elements called *primitives*. Primitives are analytically defined by high-level parameters (e.g., a sphere primitive is defined by a center C and a radius r). A very common primitive is the triangle, defined by its three vertices. *Triangle meshes* (or their generalized form, *polygon meshes*) are flexible enough to represent the geometry of many different object classes. See Figure 2.2(b).

Object materials play an important role in the illumination and shading computation. Before classifying which material properties are significant, it is important to understand the shading models. They can vary significantly in complexity (see Section 4.2.3), but for simple models a few material properties are enough to describe an object appearance (e.g., diffuse and specular properties). However, more sophisticated shading models consider other information that exists in imperfect surfaces. Actually, surface microgeometry is responsible for what happens with the reflected light, but it would be practically impossible to simulate and represent it. Object representation, therefore, rarely reaches this level (microscale details).

Once digitally represented, the scene must be rendered, which is the topic of the next subsection.

2.2.2 Visualization and Rendering

A primary task in visualization is determining which parts of which objects are visible in a scene. The algorithms that perform this task are called *Hidden Surface Removal* (HSR). Some authors also call them *Visible-surface Determination* [FvDFH90]. An object (or part of it) is considered to be visible if it is in the viewing field and there is no other object between it and the viewer. The most popular HSR algorithms are z-buffer [Cat74] and ray tracing [App68]. Z-buffer is implemented in all current graphics cards, combined with the triangle rasterization algorithm. Ray tracing is mostly used when quality is more relevant than speed (for off-line applications), and it is frequently used for producing computer animations, including film scenes. We will discuss Hidden Surface Removal in Section 3.1. Online and off-line classification divides the visualization domain into two: *off-line rendering* and *interactive visualization*. We are interested in the second.

But finding the visible parts of each object is just part of the work. Another task is to correctly compute surface appearance based on the lighting environment and the material properties (as mentioned in the previous section). There are numerous shading models proposed for graphics rendering. For example, a simple shading model can consider only *Lambert's Law*. Lambert's Law states that, given an ideal diffuse surface and a directional source of energy, the surface reflects equal energy in any direction and its intensity depends on the angle θ between the surface normal and the source direction.

Later, Phong [Pho75] proposed a specular reflection component in the shading model to simulate shiny surfaces (such as an apple peel). A more complex shading example is the one based on the BRDF function (bidirectional reflection distribution function) [NRH+77], which describes the reflected light depending on the incoming light and the viewing (reflected) directions. But, as previously mentioned, the reflected light is the result of lighting phenomena happening in microscale level and, in general, the shading models are approximations of these phenomena.

In our work we have two extra constraints for visualization: interactivity and massive data input.

Interactive Visualization

The word "interactive" means that there is a relation between the visualization system and user actions. The images produced by the visualization algorithm are a result of orders given by users. Users interact with the software by means of an input device (for example, the mouse), and expect an immediate visual response. Latency should be restricted, so that the user feels as is he/she was working in a *real-time* system. The response-time limit is subjective and there is no fixed number found in the literature. The minimum speed accepted as interactive varies between 10fps and 30fps (*frames per second*). Thus, any computer graphics system working at a speed larger or equal to 30fps is unanimously considered an interactive visualization system. With the help of current graphics cards, such speed is not a problem when working with simple scenes. However, massive data visualization is still a challenge, and it is only possible to achieve this speed through the combination of special solutions.

Massive Data Visualization and Multiscale

Massive models are a challenge for interactive visualization systems. While the user walks through a virtual scenario, the system must correctly solve visibility and shading for each pixel in the screen, all this in a high frame rate ($\geq$ 30Hz). To visualize a large amount of data (complex scenes and/or complex objects) the work becomes very challenging. The visualization of massive data (complex objects and scenes) has always been an important issue. Generally, data increases in volume at the same rate or even faster than processors Moore's law [3]. For this reason, much work has been dedicated to accelerate the visualization algorithms, in order to keep efficiency while outputting quality images.

In this thesis we review visualization algorithms and acceleration systems found in the literature. We classify them in different levels of scale: scene-scale, macroscale (object-scale), mesoscale and microscale. For each scale we list what the main problems for correct visualization are, how traditional algorithms deal with them and how to accelerate the procedures.

The list below summarizes the contents of each scale level:

Scene-scale: In the scene level the main question is visibility. Given the camera parameters (position, direction and field-of-view) and objects in a scene, which of them are visible? How to answer this question is the goal of HSR algorithms described in Section 3.1. With the purpose of accelerating the visibility tests, several algorithms were created for both ray tracing and rasterization. In Sections 3.2 and 3.3, we explain these visibility-based accelerating algorithms.

[3] On April 19, 1965 Electronics Magazine published a paper by Gordon Moore in which he made a prediction about the semiconductor industry that has become the stuff of legend. He predicted that the number of transistors the industry would be able to place on a computer chip would double every year (source: http://www.intel.com). In subsequent years, the pace slowed down a bit, but data density has doubled approximately every 18 months, and this is the current definition of Moore's Law, which Moore himself has blessed [DCM04].

Macroscale (object-scale): The macro-geometry of an object can be represented in multiple ways (e.g., implicit surfaces, triangle meshes). How to apply the HSR algorithms on them is one issue in this level. Which representation is the best one for visual quality and speed? A concept that appears in this scale is that of *multiresolution*: an object can be represented in different resolutions (each of them is a different approximation of the macro-geometry with more or fewer details). Note that multiscale and multiresolution are different concepts. The latter usually modifies the level-of-detail of the macroscale geometry. Macroscale is deeply discussed in Chapter 4.

Mesoscale: The mesoscale focuses on a way to represent (and render) object details without the complete and exact micro-information, and that is why it is a middle (*meso*) level. The concept of texture belongs to the mesoscale level. Texture is somehow related to the tactile sensation. When someone touches an object with his hands he can feel its texture. But the texture is also responsible for visual information in such a way that we can imagine the sensation by just observing an object. Texture mapping ([Cat74]), bump mapping [Bli78] and Bidirectional Texture Mapping (BTF [DvGNK99]) are some examples of techniques that reproduce surface details without extra polygons. We will discuss these methods among other mesostructure methods in Chapter 5.

Microscale (photon-scale): In the microscale level the structures and phenomena are harder to represent and simulate. Research in photon-scale is like using a microscope to see the surface details, slowing-down the photons of a beam to understand how light intensity is altered by geometric and physical properties. In this level of detail, many relevant events occur at the same time: reflections, refractions and absorptions. The sum of all these phenomena results on how the light will leave the surface and finally reach the human eye. In chapter 6 we discuss some methods related to this scale: BRDF, *photon mapping* [Jen96] and *global illumination*.

Considering the different scales is important for accelerating the visualization, since the issues are different in each level. For example, most techniques to accelerate displaying in scene-scale cannot be applied in mesoscale and vice-versa. For this the reason we decided to group the methods by scale level.

2.3 Our Contributions:

Although listed sequentialy, the following contributions have strong relation between each other.

2.3.1 Combining Rasterization and Ray casting

In Section 3.1 we describe the main Hidden Surface Removal (HSR) algorithms. Among them, ray casting (or ray tracing) and rasterization/z-buffer are the most used. While the first one determines for each pixel which object is visible, the second one determines for each object primitive which pixels must be rendered. Although they seem to have orthogonal implementations, some research has been done to combine them, benefiting from both (see Section 3.1.4).

In our work we also propose the combination of ray casting and rasterization for rendering a scene, but in an original way. In our approach, some objects are rasterized while others are ray-casted, in a free sequence order. The seamless combination is possible due to the use of the z-buffer to solve the visibility problem between objects (see Section 7.1). The goal is to use the rendering algorithm (ray casting or rasterization) that is more appropriate to a given object representation.

Compared to common interactive visualization systems, which are usually based on hardware rasterization, we achieve better quality results and better performance for massive models (see Chapter 11). Our implementation explores the programmable GPUs (Graphics Processing Units) to implement ray-casted primitives and to execute the default rasterization.

2.3.2 GPU Ray-casted Primitives Framework

Our proposition includes the extension of current programmable graphics cards to include ray-casted primitives in their rendering pipeline. These primitives should be highly efficient to provide interactivity, even for large data volumes. They should also be compatible among one another and compatible with the rasterized triangles (which are the GPU default primitive).

Our visualization technique consists of two steps, both executed on the GPU in one-pass rendering, without breaking the pipeline:

(1) candidate pixels are pre-selected by rasterizing a standard primitive whose projection encloses the projection of our primitives;

(2) for each candidate pixel, a fragment program computes the intersection and lighting, in a ray-casting way, based on an analytic representation of our primitives. In addition, it replaces the z-buffer value of the fragment according to the equation of our primitives.

In this thesis, we present this framework that is suitable for different classes of ray-casted primitives that respect the above constraints (see Section 7.1). The primitives can be any surface (or volume) whose ray-casting algorithm is known and whose convex-hull (or bounding-box) is given.

We have implemented some implicit surfaces with different orders of complexity (quadrics, cubics and quartics) and surfaces represented by height maps. The first group is useful for manufactured objects, while the second one is more adequate for natural objects.

2.3.3 GPU Implementation of Implicit Primitives

We have started our implementation by second-order implicit surfaces, the quadrics. We have developed a generic quadric algorithm (see Section 10.1) and some specific quadric surfaces: spheres (Section 10.2), ellipsoids (Section 10.3), cylinders (Section 10.4) and cones (Section 10.5). These primitives are very fast to render due to a ray/quadric intersection algorithm that we have developed and implemented directly in programmable GPUs.

In Section 10.6, we extend the procedure for cubic surfaces. Finally, we give special attention to the torus surfaces (Sections 10.7 and 10.8). Tori have a quartic implicit representation, so the intersection computation needs to solve a forth-order equation. We have tested different solutions to implement numerical methods directly in the graphics card (Sturm, double derivative bisection, sphere tracing, Newton-Rampson) and we obtained the best results with Newton's root-find approach.

We are especially interested in industrial environment visualization, such as oil platforms (see Chapter 11), where tori, cylinders and cones are dominant.

2.3.4 Reverse Engineering

Most object geometry in industrial infrastructures is a combination of common quadrics and tori sections (the latter are used for "elbow"-shaped tubular joints). However, the model is usually presented by triangle meshes that discretize the pieces. Without the original information we cannot employ our special GPU primitives for visualization. For this reason, we have implemented a novel procedure for higher-order primitive recovery

from triangle meshes. The algorithm traverses the meshes to recover the original implicit primitives of data (see Chapter 9).

2.3.5 Height-maps GPU Implementation

Also based on the GPU ray-casted primitives framework, we have implemented the visualization of surfaces represented by height maps. In our technique, these primitives are visualized with self-shadowing, correct silhouette and self-occlusion. They can be used for mesostructure rendering (Chapter 5) and also as a new representation for complex natural objects (see Chapter 8).

2.3.6 Geometry Textures and the Conversion Algorithm

Complex scanned surfaces are usually represented by either a point set or a triangle mesh. We propose a new representation by means of a set of height maps (the *Geometry textures*, see Chapter 7). This novel structure is visualized by using the technique we have described in the previous subsection. To achieve the results, the first step consists in converting a triangle mesh representation in a set of geometry textures, which is explained in Section 8.1.

2.3.7 Survey on Visualization based on Scale Level

Finally, one of our contributions is a survey about visualization (Chapters 3, 4, 5 and 6). We made the decision to group the techniques by their scale level: scene, macro, meso and microscales. This scale grouping is an original approach for visualization techniques survey. We review basic visualization algorithms up to recent methods that use programmable graphics cards, making relevant comparisons.

2.4 Thesis Organization

We have grouped the chapters of this thesis in three parts as follows:

- Part I: State of the Art in Visualization
  - Chapter 2: Scene-scale
  - Chapter 3: Macroscale
  - Chapter 4: Mesoscale
  - Chapter 5: Microscale
- Part II: Natural models
  - Chapter 6: Geometry Textures
  - Chapter 7: Rendering Natural models with Geometry Textures
- Part III: Manufactured Objects and Infrastructures
  - Chapter 8: Structure Recovery
  - Chapter 9: Higher-Order Primitives Visualization
  - Chapter 10: Results for Manufactured Model Visualization

Part I

State of the Art in Visualization

In this part, we describe previous work about visualization. They are grouped according to their scale-level, providing a direct comparison of techniques with similar purpose. Each scale-level is explored in a different chapter following a top-down sequence.

- **Scene-scale**, Chapter 3. We start by explaining in details some of the HSR algorithms. Then, we describe *Spatial Scene Subdivision* methods and *Visibility Culling* algorithms.
- **Macroscale** or *object-scale*, Chapter 4. We start by explaining the different ways to represent the macro-geometry of an object (Section 4.1). Then we present the object-based acceleration methods: *Geometry simplification* (Section 4.3), *Replacing geometry by images* (Section 4.4) and *Replacing geometry by volumes* (Section 4.5).
- **Mesoscale**, Chapter 5. In this scale level we focus on how to simulate the appearance of objects. Many algorithms were created in this subject targeting interactive applications. We describe how to represent objects meso-structure and how to visualize them interactively.
- **Microscale**, Chapter 6. In this chapter we expose some sophisticated lighting models.

Chapter 3

Scene-scale

At scene-level, the visibility of different objects is the primary task to be solved. In Section 3.1 we review the main visibility algorithms: painter's algorithm, ray tracing and rasterization. To optimize the visualization at this level, it is essential to organize the objects in a spatially coherent structure. This is the subject of Section 3.2 (*Spatial Scene Subdivision*). In ray-tracing systems, the spatial structure reduces the ray-intersection algorithm to $O(\log N)$ for N objects in the scene [Wal04]. In rasterization systems, the structures are useful for the culling algorithms, Section 3.3. Culling algorithms can eliminate numerous objects in a complex scene [BWPP04], reducing the total number of rasterized objects and thus displaying them faster.

3.1 Hidden Surface Removal Algorithms

The goal of any *Hidden Surface Removal* (HSR) algorithm is to determine which parts of which objects are visible. Objects (or part of them) in the viewing field that are not occluded by any other object are considered visible. Several algorithms were proposed in computer graphics history and we discuss the most popular of them in this section.

A large number of HSR algorithms was developed since the beginning of computer graphics; in the next subsection (Subsection 3.1.1) we discuss one important class of them, the List-Priority Algorithms. Subsequently, we describe the two most used of all HSR algorithms: z-buffer/rasterization (Subsection 3.1.2) and ray tracing (Subsection 3.1.3).

3.1.1 List-Priority Algorithms

The basic idea of *List-Priority Algorithms* is to find a visibility ordering for objects so that, if the objects are rendered in this order, the final image is correct. For example, if all objects are ordered in a back-to-front direction (*depth-sort*) and we draw all of them in this order, the closer objects will correctly overwrite the more distant ones. This algorithm is often known as the *painter's algorithm*, in reference to the way a painter executes his work, see Figure 3.1. A typical problem for the painter's algorithm is the situation of objects cyclically overlapping one another; in this case, it is necessary to split some of them (see Figure 3.2).

List-priority algorithms differ in how they determine the sorted order. Newell, Newell and Sancha [NNS72] developed a complete depth-sort algorithm, while other algorithms were developed exclusively for displaying octree-encoded objects [DT81, GWW86]. But the most known sorting algorithm (used with the painter's algorithm) is the binary space-partitioning (BSP) tree developed by Fuchs, Kedem and Naylor [FKN80a, FAG83].

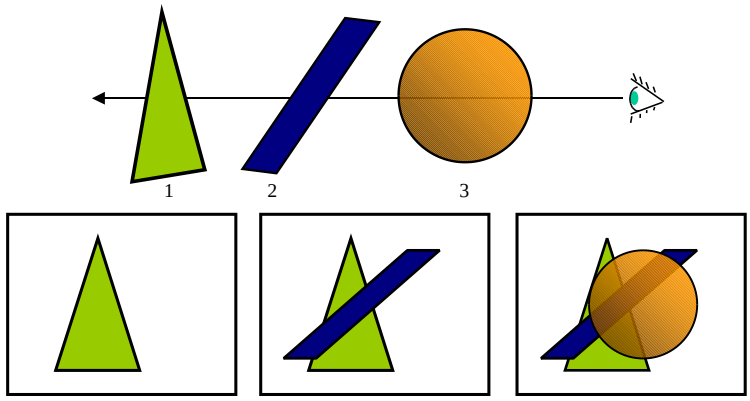


Figure 3.1: *Top:* In *List-Priority Algorithms*, the objects are ordered by their distance to the viewer. *Bottom:* The painter's algorithm renders them from back to front; closer objects cover the farther ones resulting in a correct *Hidden Surface Removal*.

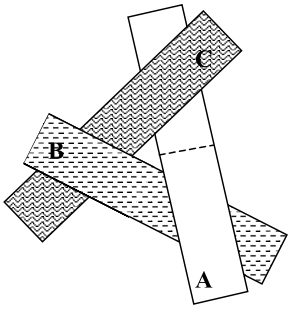


Figure 3.2: Objects disposed in a special configuration, resulting in cyclical overlapping. It is impossible to have a depth-sort of them, except if one of the objects is cut in two. In this figure the object A will be split in two.

Although restricted to a static scene, BSP is extremely efficient for real-time display from any viewpoint (see [dBvKOS97]).

The BSP tree recursively subdivides the 3D space forming a hierarchical structure with each node representing a convex subspace. In its construction step, generally done offline, planes are chosen to subdivide the space (or subspace) in two, which is done recursively until each subspace contains only one object (or just a few). During rendering, the painter's algorithm traverses the BSP structure considering the position of the camera relative to each cutting plane (front or back). This way, the objects' depth-sort is achieved linearly, accelerating the display.

The BSP tree was made popular by the *DOOM* PC video game. Using this data structure, real-time graphics could be achieved on a 33 MHz PC with a few megabytes of RAM. At that time, in the beginning of the 1990s, there was neither 3D GPU (Graphics Processor Unit) nor 3D API (OpenGL or Direct3D) on consumer PCs. So Doom was revolutionary in its ability to provide a fast texture-mapped 3D environment. It is a typical example of an algorithmic solution pushing the limits of hardware.

3.1.2 Z-Buffer and Polygon Rasterization

The z-buffer algorithm is conceptually the simplest HSR, and it can be efficiently implemented in hardware. Initially developed by Catmull [Cat74], it is used by all the current commercial graphics cards processors (*GPU - Graphics Processor Unit*). Z-buffer is generally used in combination with polygon rasterization (more precisely, triangle raster-

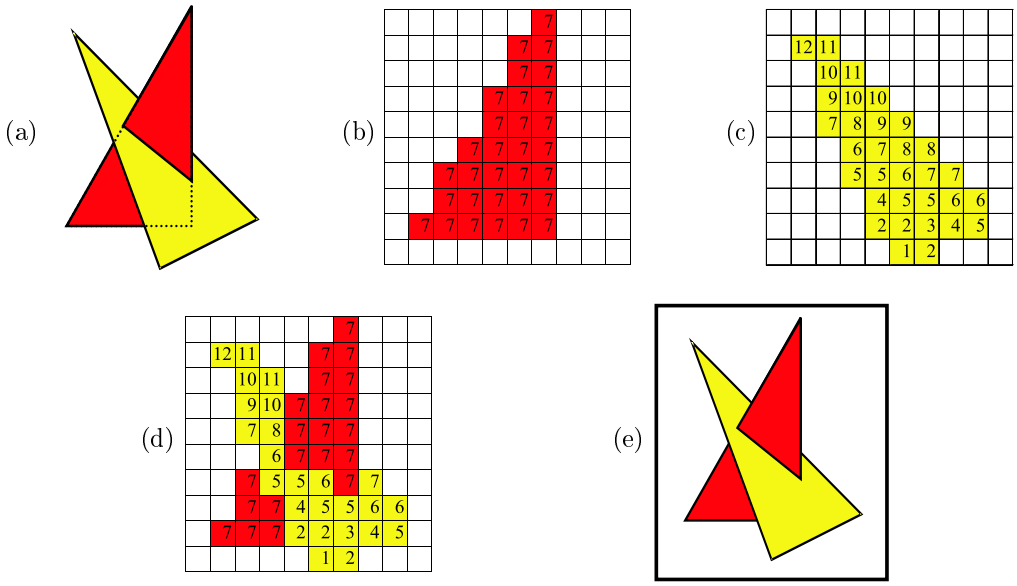


Figure 3.3: These pictures show how z-buffer solves the visibility between primitives. (a) The two primitives to be drawn; (b) the red triangle is perpendicular to the viewing direction, so the z-values are the same for every pixel; (c) the yellow triangle z-values; (d) when both are rasterized the z-values are tested and just the smallest values remain (in both color and z-buffers); (e) the final color buffer.

ization in hardware implementations). Rasterization consists in projecting the polygon vertices on the screen and filling the pixels inside the projected polygons (interpolating vertex color, texture coordinates and other values).

In the z-buffer algorithm, all the polygons are rasterized in any order. For each pixel, depth information (the distance to the viewer) is registered in a buffer, the z-buffer. The z-buffer has the same resolution of as the color-buffer and is initialized with the maximum depth. Before changing the color-buffer, each pixel has its depth compared with the one already registered in the z-buffer. If the current pixel is closer than the previous depth, then both buffers (color and z) are updated with the new information. At the end, the color-buffer only contains information about visible surfaces. See example in Figure 3.3.

For more than ten years, the z-buffer has been the standard implementation on GPU for solving 3D visualization. This popularity was firstly based on its very fast capabilities for processing polygons (measured in millions of triangles per second). Secondly, it became very easy for the user to display entire scenes, given that there is no restriction to the sequence order. With the availability of the 3D graphics libraries OpenGL and Direct3D, the access to this technology was made even easier.

3.1.2.1 Rasterization Acceleration Methods

The z-buffer rasterization method has become popular with its hardware implementation, which is available in any common graphics 3D processor nowadays. For this reason, this method is the reference in interactive visualization. However, for massive data, composed by several million triangles, rendering suffers significant loss in speed, thus preventing interactivity. For this reason, a lot of effort was done to accelerate the method with both hardware and software solutions.

In the hardware side, GPU (Graphics Processor Unit) has experienced an impressive evolution, doubling speed each 12 months, faster than Moore's law for CPU's (see footnote on page 35). But the concept of "massive data" has also been changing, and data

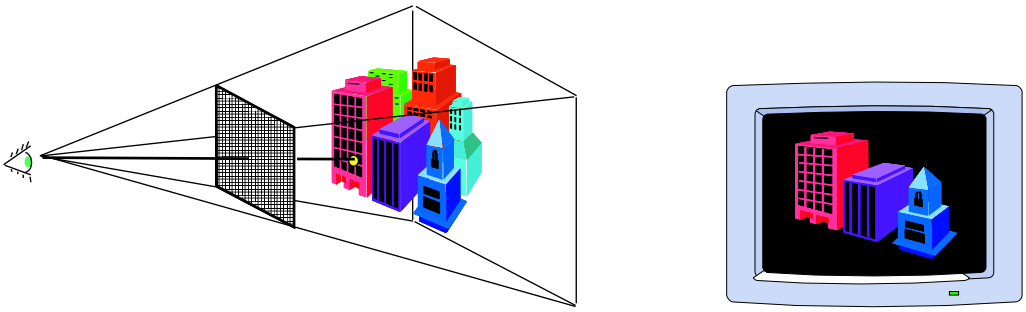


Figure 3.4: Given a center of projection O, an $N \times M$ pixels screen and the scene, the ray-tracing algorithm finds the first intersection between each viewing ray ($\vec{v} = (\text{Pixel}_{i,j} - O) \mid 0 \leq i, j < N, M$) and the objects in the scene.

sizes increase at least at the same rate. As a result, the visualization of very complex scenes always suffers from performance bottlenecks due either to vertex processing or to limited bandwidth transference between CPU and GPU.

In the software side, much work has been done in reducing the number of triangles passed to the graphics card in order to accelerate the rendering process. Techniques using pre-visibility tests [Ebe01, JC98, ZKEH97, KMGL97, HMC+97, CT96, GKM93, ZMHKEH97, LG95], multiresolution [Hop96, Hop97, GH97, Hop99, YSGM04, TGV01] and impostors [MS95, Sch97, SDB97, MD98, DDSD03] are the subject of intensive research.

We shall discuss each of these topics in different points of the text:

- Pre-visibility tests are discussed in Section 3.3.
- Multiresolution is the subject of Section 4.3, when we will discuss about macroscale techniques.
- Impostors are discussed in Section 4.4, also related to macroscale techniques.

3.1.3 Ray tracing

Historically, ray tracing was the first HSR algorithm. It was first developed by Appel [App68] and extended by Goldstein and Nagel [MAG68, GN71] and by Whitted [Whi80]. The ray-tracing algorithm traces imaginary rays of light from the viewer to the objects in the scene in order to determine visibility of their surfaces (see Foley et al. [FvDFH90]). The viewing-rays are fired from the center of projection through each pixel of the screen into the scene (see Figure 3.4). The object with the closest intersection determines the final color of the pixel. This final color depends on the object material and the illumination model.

The above explanation is just the basic implementation of ray tracing. The algorithm can be further extended to handle shadows, reflection and refraction using a *recursive ray tracing*. To compute shadows, an additional ray is traced from the intersection point to the light source. If the shadow-ray intersects any object, then the point is in a shadow. If the object intersected by the primary ray has a reflective material, then a new ray is computed by reflecting the original one to the surface normal. This new ray is then recursively fired as the primary ray (its origin is the last intersection). Similarly, refracted rays are created if there are transparent intersected objects. Since reflected/refracted rays can generate other rays, a depth limit is applied to the algorithm to avoid infinite recursiveness.

Ray tracing is known for the ability to produce photo-realistic high quality images. Nevertheless, to achieve those results, some extra efforts are necessary. Anti-aliasing,

special object material and radiosity are some of the features used to achieve such quality. However, these features impact the execution time so much that a single image can take hours to be produced in a conventional system.

The quality of the images is not the only reason for the popularity of ray tracing. There are some classes of objects that can be very easily implemented in ray-tracing systems. Quadrics (Section 4.2.1), for example, are very useful surfaces and they are almost always implemented in ray-tracing systems. In general, implicit surfaces are very suitable for ray-tracing algorithms.

We shall discuss three different topics related to ray tracing: in Subsection 3.1.3.1 we discuss a nomenclature problem, in Subsection 3.1.3.2 we expose some techniques about accelerating the ray tracing by software, and in Subsection 3.1.3.3 we show ways to accelerate the ray tracing by hardware.

3.1.3.1 Ray tracing or ray casting?

In the literature, there is no consensus about the difference in meaning between "ray casting" and "ray tracing". In the book of Foley et al. [FvDFH90], one of the most important references in computer graphics, both terms are treated as synonyms: "*Ray tracing*, also known as *ray casting* (...)" (extracted from Section 15.10). However, most authors prefer to use the two words (tracing and casting) to distinguish different algorithms.

We can usually find two ways to distinguish ray tracing from ray casting:

(i) One possibility is to distinguish them according to the type of rendered object. If the data is a volumetric representation of the object, then it is ray casting; if it is a surface representantion, it is ray tracing.

(ii) Another way to distinguish them is through the implementation of recursiveness. If there is recursiveness, it is ray tracing, otherwise it is ray casting.

The second distinction seems to be more accepted and, from our point of view, more appropriate. Using the recursiveness as the criterion, volume rendering is automatically classified as ray casting. Nonetheless, when rendering surfaces, if the algorithm only uses the primary rays, then it should be considered ray casting as well. If the algorithm treats reflection and refraction, then it is considered ray tracing.

For example, how should we classify an algorithm that casts shadows over surfaces besides the primary rays, but without any reflection/refraction rays? Using the criterion based on recursiveness, its classification is clear: it is a ray-casting algorithm.

Using Classification (ii), the primitives we developed in our work are ray-casted (see Section 7.1). The new aspect in our work is that the ray-casting algorithm for these primitives runs in the GPU.

3.1.3.2 Ray-Tracing Acceleration Methods

The ray-tracing algorithm can be accelerated in many different ways, usually exploring some kind of *coherence* (e.g. temporal coherence, image space coherence, object coherence). Among the diverse techniques developed in the last twenty years, we highlight three categories:

1. algorithms that **reduce the number of primary rays** (by subsampling [Gla88], tracing rays towards triangle vertices [UBBS01], or by caching previous frame samples [WDP99, WDG02]);

2. algorithms that **reduce the number of secondary rays** (e.g. reflection mapping and shadow maps [MH99a]);

3. techniques that **accelerate the ray-scene intersection** (by doing faster intersection tests [Woo90] or through spatial scene subdivision [JW89, KK86, Gig88, Sam89, Coh94, Sub90]).

Among all these techniques, spatial subdivision methods are the most important, since they have been proven to reduce the computational complexity of ray-scene intersection from $O(N)$ to $O(\log N)$ [Wal04]. We explore spatial subdivision in Section 3, which deals with algorithms related to scene-scale level.

Another (less algorithmic) category of research for accelerating ray tracing is the one based on *powerful hardware exploration* [WSS05, CHH02, PBMH02, WPS+03, KW03a]. An interesting point of using hardware acceleration is its orthogonality to the other acceleration methods, so hardware techniques have a significant potential for accelerating ray tracers.

In the next subsection we present hardware-based ray tracing techniques.

3.1.3.3 Hardware-Based Ray tracing

Ray tracing, traditionally run in common CPU, has been considered not fast enough to be used interactively. Nevertheless, recent technological evolutions, associated with acceleration methods (discussed in the previous section), have encouraged new brand of research that aims to produce real-time ray tracing (RTRT) systems. Regarding hardware, we can find three different kinds of platforms:

- CPU implementations: very similar to conventional algorithms, they differ in focusing on speed rather than on quality. Some applications also use parallelism to accelerate the algorithms.
- Dedicated Hardware: chips manufactured exclusively for targeting maximum performance in real-time ray tracing.
- Programmable GPU: Benefiting from programmable GPU, these implementations use the high computation capabilities of current graphics processors.

Real-Time Ray Tracing on CPU

RTRT (real-time ray tracing) on CPU differs from traditional algorithms regarding their priority since RTRT priors efficiency instead of high-quality. High-performance CPU implementations [WPS+03] provide satisfactory results only for simple scenes and lower resolutions. Even the performance of 3GHz processors is not fast enough for complex scenes. Maybe the reason is that ray tracing uses only a small part of the CPU power (only a small part of the CPU instruction set is used). An alternative for RTRT CPU implementation is the use of PC clusters [Wal04], but it is much more expensive and involves complex implementation details.

Considering these restrictions, simpler hardware with greater parallelism and containing the instructions required for ray tracing seems to be a more appropriate solution.

Dedicated hardware

In recent years, some special-purpose hardware architectures were developed for accelerating parts or even the whole ray-tracing algorithm. Some of these architectures accelerate only some ray-tracing procedures, suffering from bandwidth limitation between the different steps [Gre91]. Woop et al. work [WSS05], *RPU* (Ray Processing Unit), is the first programmable RTRT dedicated hardware. Before this, specialized hardware was restricted to a fixed functionality. The authors uses a hardware implementation for kD-tree traversal. Their results are acceptable despite the low speed of their prototype processor (66 MHz).

Ray tracing on programmable GPU

The ray-tracing algorithm has a natural parallelism behavior that has been explored more recently in graphics cards [CHH02, PBMH02, WPS+03, KW03a]. The usual z-buffer algorithm is not used to determine visible surfaces, since ray tracing is able to discover, for each pixel, which object has the closest intersection with the viewing ray.

In *The Ray Engine* [CHH02], the ray tracer is limited to render only triangles. For each one of them, a screen-size quadrilateral is passed to the engine, containing in its attributes the triangle description, while two textures contain the origin and direction of the rays. Therefore, each pixel can test the intersection using that information. The z-test is used to maintain the closest parametric distance of each ray if it touches any triangle. With some optimizations, almost 200M intersection computations per second are achieved, even with some data transference from the GPU back to the CPU (a very slow operation in the AGP bus).

Purcell et al. [PBMH02] have shown a very interesting way to use GPU for ray tracing. They broke the algorithm into kernels that access the entire scene data, which is mapped in textures. Some of the kernels need multiple passes, but they eventually succeed in eliminating the GPU-CPU communication bottleneck.

Krueger et al. [KW03a] have developed volume ray casting using a multi-pass approach. The first two passes generate the ray information that will be used by each pixel. Then a sequence of passes is done, alternating ray-traversal and acceleration steps. They have implemented two types of acceleration tests: empty space skipping and threshold opacity achieving. Using the early z-test of their ATI graphics card, each ray-traversal pass could be skipped if the acceleration pass that had just been done passes the test. These accelerations were their main goal, producing some good results, although below their own expectations.

The use of programmable GPU for interactive ray tracing is a very promising approach, especially when associated with the aforementioned structures, as in Foley et al. [FS05].

3.1.4 Using both Rasterization and Ray casting

Rasterization and ray casting have been used together in previous work. We can clearly distinguish two different uses of this combination:

- using rasterization for accelerating ray tracing; and
- using ray casting for enhancing image quality in rasterization systems.

Exploring fast GPUs, researchers have used rasterization as an initial step to accelerate ray tracing. Visibility information of the primary rays can be obtained from the rasterization result (using colors to encode objects id's) [WS01]. The same could be done for the shadow request, by previously rasterizing the scene from the light source point of view.

In Stamminger et al. [SHSS00], *corrective texturing* is used to add ray-tracing details over an already rasterized image. This is not done in real time. Actually, when the user stops moving, a progressive image quality enhancement begins.

Thesis proposition: Combining Rasterization and Ray Casting
We propose a visualization algorithm that combines rasterized objects and ray-traced ones in the same rendering.
As we will see in Section 4.2, there is a class of objects for which ray casting is more suitable than rasterization. By using the most suitable rendering algorithm for each object, we achieve better performance, better quality and less memory consumption (see Section 7.1).

3.2 Spatial Scene Subdivision

The goal of *Spatial Scene Subdivision* algorithms is to create an object hierarchy based on the spatial position of the objects. The output is always a tree whose leaves are the objects. Binary trees (or any of their variations) are the optimal ones.

A simple bottom-up subdivision procedure is the *bounding volume hierarchy* [KK86]. This procedure creates the hierarchy by assembling close objects and computing their bounding volume. To obtain a binary tree the objects and the bounding volumes are assembled in pairs. Although simple to implement, this technique allows different parts of the hierarchy to overlap in the same regions in space, which is a drawback since it can lead to inefficient traversal.

The other spatial scene subdivision methods always subdivide the space into non-overlapping set of subspaces (commonly called *voxels*). The most common spatial subdivisions are octrees, BSP trees and kD-trees.

Octree is a regular subdivision structure for 3D space (its 2D equivalent is the *quadtree*). Each subspace node is divided into eight nodes (the cutting point is half the length in each direction (x, y, z)). Usually a user-defined integer defines the tree size. See [Sam89].

BSP tree is explained in Section 3.1.1, where we show its association with the *painter's algorithm* for visualization purposes [FKN80a, FAG83]. In the ray-tracing context, the BSP reduces the number of intersection tests per ray in two ways: by eliminating objects to be checked, only traversing the *voxels* intersecting the ray; and also by performing an early ray termination when an object is found, since the objects are ordered by distance.

kD-tree has similarities with octrees and BSP trees. K-d trees generalize octrees by allowing splitting planes at variable positions and in one dimension at a time. A K-d tree can also be seen as a BSP tree but restricted to axis-aligned cutting planes. Thus, the K-d tree integrates the flexibility of adapting the subdivision according to the spatial density and the simplicity of respecting axis-aligned partition (each voxel is always an oriented parallelepiped). The kD-tree is a very common choice because it has equal or better performance than any other technique [Wal04]. For more about K-d trees, see [Sub90].

Spatial subdivision techniques are the most important algorithmic solution for ray-tracing acceleration. For rasterization these subdivision structures are also useful for visibility-culling methods, which is the subject of our next section. We will discuss in detail the *frustum culling*, *occlusion culling*, and *portal culling* methods, which are essentially scene-level scale acceleration techniques. Subsequently, we will also mention the *Back-Face Culling*, which actually works in object-scale (macroscale) level. The reason why we describe it in the scene-scale section is to keep all the visibility culling methods together.

3.3 Visibility Culling

Visibility is a very important issue in computer graphics. The algorithms for hidden surface removal (HSR) are completely based on the determination of the visible objects or part of them (Section 3.1). Since the beginning of computer graphics, many HSR algorithms have been developed, among which the most currently used are Z-buffer/rasterization and ray tracing. Nowadays, visibility computation for correct rendering is considered to be a problem already solved.

Visibility culling is a technique for optimizing the rendering of 3D scenes ([Ebe01, JC98, ZKEH97, KMGL97, HMC+97, CT96, GKM93, ZMHKEH97, LG95]); it is not intended as a replacement for HSR algorithms. The main goal in visibility culling is to

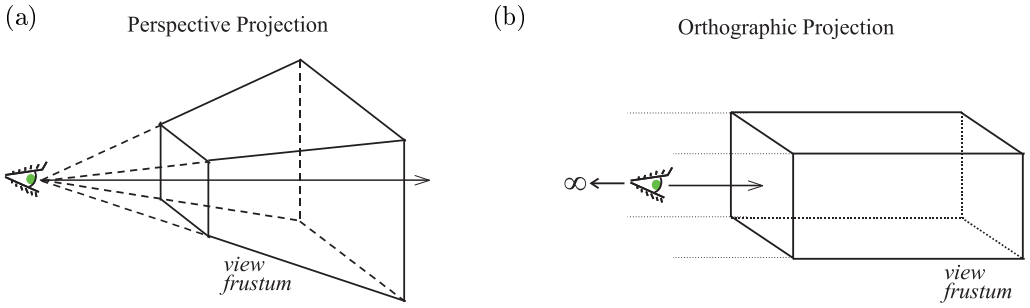


Figure 3.5: View frustum volume: (a) in perspective projection and (b) in orthographic projection.

uncharge the graphics pipeline by culling, as soon as possible, the primitives that, for some reason, are invisible in the current frame. In general these methods are conservative, eliminating just the primitives that are surely invisible. As a result, there is no loss of quality in the final images. The conservative approach does not cause errors, since invisible surfaces (falsely classified as visible) are nonetheless culled by the HSR algorithm. However, the HSR algorithm only solves the visibility at the end of the pipeline (in Z-Buffer it happens at the end of the raster stage).

The remainder of this section presents a survey of the different culling algorithms.

3.3.1 Frustum Culling

The frustum is the viewing volume in a visualization system. When using an orthographic projection the frustum is simply a rectangular box (See Figure 3.5b). In perspective projection the viewing volume is actually a truncated pyramid with a rectangular base (See Figure 3.5a). The frustum culling algorithm [Cla76] marks the objects as invisible if they are outside the frustum, and as visible if they are completely or partially inside. This method is conservative, since only objects totally outside of the frustum are culled away. Moreover, objects that are marked as visible are not necessarily visible because of further occlusions.

Usually, the visibility is done by testing the objects *bounding volumes* intersection with the frustum. The most common bounding volumes are spheres and boxes, but other simple volumes can be used, such as cylinders, capsules, lozenges or ellipsoids [Ebe01, Hof02]. The choice of which bounding volume to be used is normally based on three criteria: object fitting, intersection test speed and implementation difficulty.

The advantage of using bounding volumes in the visibility test is their computation time. It is much faster to test the intersection of the bounding volume with the frustum walls than to test each primitive of a complex object. Indeed, a bounding volume can include more than one object or even other bounding volumes. There are some hierarchical view-frustum culling in the literature (see [Cla76]) based on the scene graph. In a pre-order traversal, nodes with bounding volume outside the frustum will not be processed. With the hierarchy an entire group of objects can be discarded with only one test.

It is also important to mention that bounding volumes are not the only technique used to produce frustum culling. There is also the BSP (Binary Space Partition) approach (see Section 3.1.1), which divides the space using planes. When a plane does not intersect the frustum, the objects belonging to the opposite side of the frustum are culled. When associated with the *painter's algorithm* the BSP becomes a HSR algorithm. See [dBvKOS97, FKN80b] for further discussion about them.

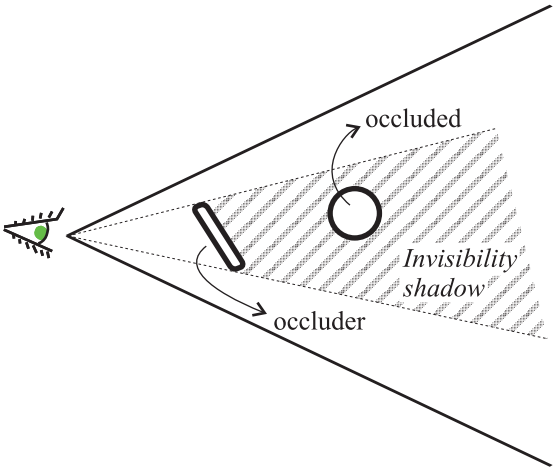


Figure 3.6: Visible objects occlude other objects if they are in their "invisibility shadow".

Figure 3.7: Occlusion culling applications: (a) industrial plants [BWPP04]; (b) urban environment [BWPP04]; and (c) architectural walkthrough [LG95].

3.3.2 Occlusion Culling

Even after frustum culling, there are still some objects (or part of them) that are invisible due to occlusion. Imagining a light located at the viewpoint, the visible objects project an "invisibility shadow" behind them (Figure 3.6). An object is considered occluded when it is completely inside the occluder "shadow".

Some 3D scenarios are especially interesting for applying the occlusion culling algorithm: for example, walkthrough in complex industrial environments (Figure 3.7(a)), in urban scenes (Figure 3.7(b)) or in any indoor scene (Figure 3.7(c)). In these situations, the observer can see only a fraction of the total primitives in each frame.

Occlusion tests are computationally expensive operations (much more demanding than frustum-culling tests). Thus, they are only interesting if used to cull groups of objects. That is why in occlusion algorithms, as we will see, it is especially interesting to use a hierarchical data structure containing the objects. This hierarchy is traversed in top-down order while the bounding volume of each node is used for the culling test.

Hudson et al. [HMC+97] maintain a set of potentially good occluders extracted from the list of objects in the scene. At runtime, a subset of occluders (based on the viewpoint) is used to create a collection of invisibility shadows called the *shadow frustum*. Using the shadow frusta (each one described by limiting planes), the bounding volume hierarchy is traversed and the subtrees completely obscured by the shadow are discarded.

Coorg and Teller [CT96] also begin with a selected set of occluders. They have inverted the idea of the shadow frustum, actually computing the space from which the viewer cannot see an object (or its bounding volume). For each potential occluded/occluder pair, the planes defining the blind region (from which the occluded

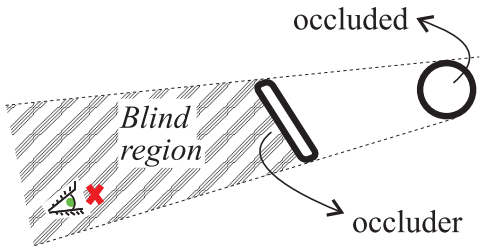


Figure 3.8: In an inverted way to describe occlusion, a potentially occluded object and an occluder define a "blind-region"; if the viewer is in the region then the object is certainly hidden.

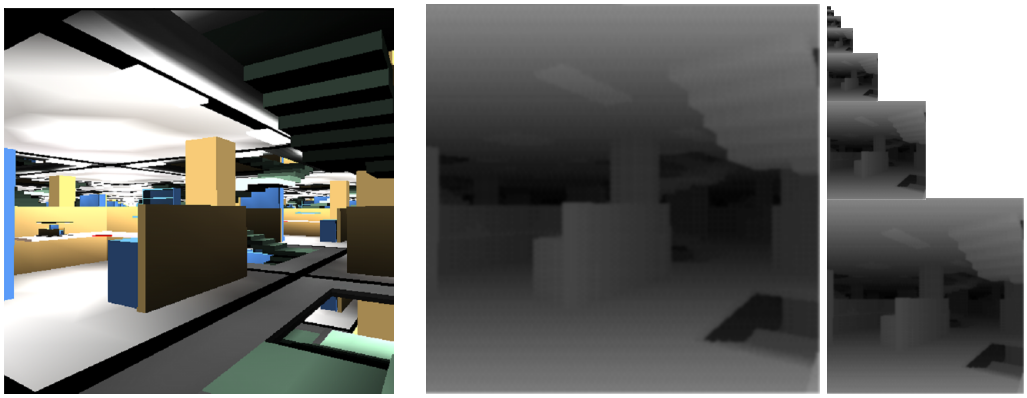


Figure 3.9: Z-pyramid: (a) the rendered scene, (b) the original Z-buffer and (c) the other images composing the Z-pyramid [GKM93] (which includes the original Z-buffer). *(Images originally from Ned Greene/Apple Computer.)*

objects are hidden) are computed (see Figure 3.8). During rendering, visibility events are tracked as the viewpoint moves (such as the viewpoint passing through one of the planes) and the occluded/occluder pairs of interest are updated.

The *Hierarchical Z-buffer Visibility* algorithm [GKM93] is an extension of the Z-buffer algorithm (see Section 3.1.2). The octree employed for object hierarchy allows an almost exact front-to-back order for the culling test and rendering. The procedure tests the sides of each node against the Z-Buffer. If they are all farther than the Z-buffer data, then the node is culled. Otherwise, the node is recursively processed or, in case of a leaf-node, its primitives are rendered. This algorithm maintains the Z-buffer as an image pyramid called Z-pyramid (see Figure 3.9), where the finest level is exactly the known Z-buffer, and each coarser level keeps, for each element, the farthest Z-value of the corresponding 2x2 elements of the finer level. For speed purposes, the node-culling test is performed in all Z-pyramid levels, starting with the coarsest ones and going up to the finest ones. It stops if the node is already classified as occluded.

Zhang et al. [ZMHKEH97] have created the *Hierarchical Occlusion Maps* (HOM) technique, which is similar to the *Hierarchical Z-Buffer Visibility* technique. HOM, however, decouples the visibility test into two steps: overlap and depth tests. As in the work by Hudson et al., in each frame HOM uses a subset of the potentially good occluders, which defines the occlusion and depth maps (used in the tests). The overlap test also uses an image pyramid like the Z-pyramid, but only for overlapping (they are called occlusion maps). The objects overlapping the occluders are then tested against the depth map. The depth map can be considered a special implementation of Z-buffer. The difference is that, for efficiency purposes, the resolution is fixed (64x64 and no pyramid structure) and the depth is marked with the farthest occluder point (the conventional Z-

buffer keeps the nearest point in each pixel). The depth verification process compares the nearest vertex depth of the testing object with those in the depth map, over a bounding rectangle (the rectangle containing the object in screen space). The objects are culled if they pass both tests: overlap and depth.

Using Graphics Hardware for Occlusion Culling

In recent GPUs, there is a useful implementation using the idea of occlusion query. The information is available to the developer as a simple query. After drawing some objects (for example, potential occluders), the CPU can ask the GPU whether an object (or its bounding box) is visible. The GPU returns the number of potentially visible pixels, testing a set of polygons against the current Z-buffer. Optionally, the visibility request and the rendering can be done at the same time, which is useful to test if an object previously marked as visible is still so. (For implementation details see [Reg02] and all about the OpenGL extension `NV_occlusion_query` [Len03].)

Visibility persistence is largely used in hardware-assisted occlusion culling applications [COCSD02]. It is based on temporal/spatial coherence between consecutive frames. Objects previously visible will probably appear in the current frame. On the other hand, one can test the bounding box of invisible objects and render them in the next frame if they appeared (showing with 1 delayed frame). Moreover, some algorithms keep the visible state of an object during some frames without retesting (in conservative applications this is done only for visible objects).

An issue in this hardware-assisted technique is the degree of CPU/GPU parallelism, which is partially broken each time the CPU waits for a GPU result. Grouping the queries can be one solution to this problem (see [Sek04]). Alternatively, Bittner et al. [BWPP04] have created a sophisticated queue of queries to avoid CPU stalls and GPU starvation: while waiting for the query results, the CPU continues sending to the GPU objects already classified as visible.

When the current Z-buffer is used to test occlusion, a pre-selection of occluders is not mandatory. The objects already rendered can play the role of occluders. But it is especially important to draw, as much as possible, the objects in a front-to-back order. For this, a spatial scene partition, such as an octree or a kD-tree, can help sort groups of objects by distance (and the walls defining the node can be used as the bounding box for visibility tests).

Non-conservative Culling

Some of the occlusion culling algorithms thus far described can relax the *conservative* aspect usually presented in culling methods. In other words, some visible objects may be skipped, provided that it does not change the final image too much (or during too much time, i.e. just during few frames). Relaxing this constraint makes it possible to use more efficient algorithms. This may be compared to non-conservative image compression. In non-conservative image compression, higher compression rates are obtained, at the cost of a lost in quality.

For example, in HOM[ZMHKEH97], Zhang et al. consider the possibility of a threshold for the opacity test for their occlusion maps. Applications using `NV_occlusion_query` implemented in hardware are also capable of applying the non-conservative idea. Once the GPU returns the number of "visible pixels", a threshold can be used, limiting the minimum number of pixels necessary for an object to be considered visible (see [BWPP04]).

Another general non-conservative procedure is to keep the state of an object marked as invisible for some frames without testing it. Notice that this is different from not retesting visible objects for some frames, which is a conservative practice.

The advantage of the non-conservative approach is the time-saving in each frame (see results in [Gre01]). This gain is a result of testing fewer objects or of not drawing

objects that are almost invisible. The drawback is the inaccuracy of images that may not contain visible parts of the scene; pop-up effects may also result (when objects are lately classified as visible).

Occlusion Culling Summary

As formerly described, the occlusion test is an expensive operation, but with very promising gains. When correctly implemented, enhanced performance is almost guaranteed for complex scenes. To achieve satisfactory gains, the implementation must include some important procedures:

(i) *group the objects* to reduce the number of occlusion queries;

(ii) execute *frustum culling* before the occlusion test, since it is much faster;

(iii) traverse the objects in a roughly *front-to-back* order;

(iv) when it is the case, choose very well the *occluders* (this is a hard task); and

(v) optionally implement a *non-conservative* approximation.

Space subdivision is especially important for items (i) and (iii) above. For example, a simple octree can help to render in an approximate front-to-back order, while grouping the objects. This hierarchical division also assists the frustum-culling test (item (ii)).

There are only few methods that support dynamic scenes. This is because it is especially hard to maintain the space partition and occluders classification. A recent work by Aila and Miettinen [AM04] considers the dynamism of objects; they took the decision of developing the algorithm completely by software (to avoid the latency of hardware implementations).

It is also interesting to mention the *early-z* implementation in current graphics hardware [Ceb05]. In the course of rasterizing the pixels a depth comparison against the z-buffer is done pixel-by-pixel, just before running the pixel shading. This method can prevent wasting time if the pixel is discarded, and it is especially interesting for expensive shading algorithms (for example, with many texture accesses). This technique cannot be considered a substitute for occlusion culling because it takes place almost at the end of the graphics pipeline.

Several authors classify occlusion methods into two categories: *from-point* and *from-region*. All algorithms related in this subsection would be classified as *from-point*, except the Coorg and Teller work [CT96], which contains the idea of blind-regions (see Figure 3.8). In contrast, the next subsection describes *Portal Culling*, with the majority of the algorithms computing region-to-region visibility. Indeed, portal culling can be seen as a subclassification of occlusion culling.

The bibliography about occlusion culling is very vast; some good surveys can be found in [BW03, Sek04, MH99b, COCSD02].

3.3.3 Portal Culling

In complex indoor scenario (for example a building floor), some rooms are never seen from other rooms. Imagining an observer in a room, only the rooms that are visible through doors and windows are not hidden. By calling rooms *cells* and doors and windows *portals*, the *portal culling* technique [AFPB, Tel92, LG95] is based on this observation. It lists which cells are visible from other cells. Then the application only renders the objects inside the cells which are classified as visible according to the observer position.

The basic idea in *Portal Culling* is related to that of occlusion culling. The main difference is that the search is done based on what can be seen through the portals, instead of eliminating what cannot be seen due to the walls (the occluders). Another

point of distinction is that almost all algorithms are from-region based (as opposed to occlusion culling techniques, for which the majority of algorithms are from-point based).

Work on this area has been developed since the early nineties [AFPB, Tel92, LG95]. All solutions have in common a preprocessing step for cell and portal classification. A data structure is created to keep the list of objects of each cell and their portals. By using this structure it is easy to retrieve the adjacent cells, making use of the portal information. So, for each cell, it is possible to record the *Potentially Visible Set* (PVS) of primitives that will be rendered in rendering time.

The first methods proposed start by constructing a BSP-tree of the scene, with the partition planes aligned with the walls. Within this structure the cells and portals are identified. In a posterior computation, cell-to-cell visibility is determined. Different algorithms were suggested for this step: point sampling [AFPB], shadow volumes [AFPB] and straight-line tracing [Tel92]. This last algorithm searches for straight lines directly linking two points, each of them in different cells. By doing so, each pair of cell can be classified as mutually visible. This type of procedures, although interesting, requires considerable preprocessing, which can take several hours depending on the examples.

The Luebke and George paper [LG95] is probably the most important reference in portal culling. It was, to our knowledge, the first one to introduce a real-time visibility solver. Their technique requires much less preprocessing than the predecessors and it can be used in interactive applications. A from-point based approach is achieved by executing the algorithm described in Figure 3.10.

Luebke and George approach has some relation to frustum culling technique, since the cull-box is used to cull out the objects and cells of the scene. Another interesting point in their procedure is the possibility of using mirrors as portals (as the example in the center of Figure 3.10(c)).

3.3.4 Back-Face Culling

The other culling methods described up to this point (*frustum*, *occlusion* and *portal*) are scene-level acceleration methods, since their algorithm traverses the scene to cull objects (or group of objects) that for some reason are not visible in the current frame. *Back-face culling*, although also a culling method, should not be classified in scene-scale level but in object-scale level (*macroscale*). This method applies the culling in portions of each object; there is no relation with the scene hierarchy. The reason for keeping the description of back-face culling in this section about scene-scale acceleration methods is to keep it close to the other culling methods, which are classified as scene-scale level.

In the real world, the parts of objects facing away from the viewer are not visible (Figure 3.11). The concept of "facing away" is completely related to the normal along the object surface. The zones where the normal and the viewing direction form an acute angle are not seen (for example, Point b in Figure 3.11).

For polygon-mesh rendering, if all the polygons are planar, each one can be classified as back-faced or front-faced. Those marked as back-faced will not be rendered. A practical way to check which face points toward the viewer relies on the polygon orientation. The vertex enumeration of a polygon can be clock-wise (CW) or counter-clock-wise (CCW). If the polygon orientation changes after projection (from CCW to CW or vice-versa), it means that the polygon is back-faced and can thus be culled. By convention, the CCW orientation characterizes front polygons. So, polygons whose projection has a CW ordering can be culled. Indeed, as pointed out by Blinn [GS93], orientation checking is geometrically the same as verifying if the angle between the normal and the viewing direction is acute (the computation is almost the same, except that it is done in different coordinate systems).

Nowadays, graphics cards have a hardware implementation of back-face culling. At the end of the geometry stage of the graphics pipeline (see Section A.1), the GPU processes the test right after the triangle projection. This process is very fast, since it is done

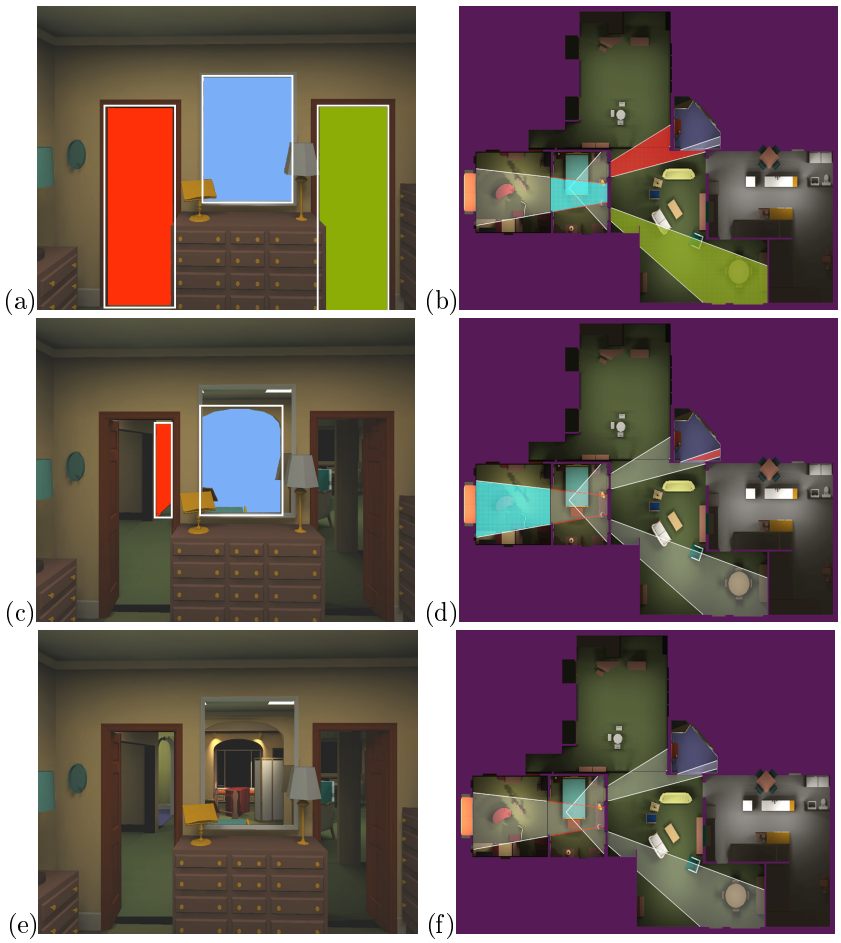


Figure 3.10: Detailed description of Luebke and George portal culling algorithm [LG95]. (a) In a first step, the algorithm renders the objects in the cell where the observer lies. Next, the portals are projected to the screen and their 2D bounding-boxes are found (the three white frames). The authors call them *cull-boxes*. Actually, the screen is also considered a cull-box in their recursive procedure. (b,c) For each cull-box not clipped out by the previous cull-box (the whole screen in the first recursive level), the adjacent cell will be rendered recursively as in (a). (d,e,f) Recursively, the objects of the adjacent cells are rendered (always clipped by the cull-box).

in parallel by the hardware, and it will save processing in the rest of the pipeline (the CW triangles are not rasterized). However, it could be interesting to cull the unwanted triangles earlier in the pipeline. This is the goal of *clustered culling algorithms*, using the CPU to cull a whole set of polygons in a single test.

Some schemes for clustered back-face culling were developed in the late nineties (see [JC98][ZKEH97][KMGL97]). The general idea is to classify the polygons according to their normal direction. An entire group of back-face polygons can be then culled in block with just one instruction. Zhang et al. [ZKEH97] suggest an approach in which the clustering is not done explicitly, but registered as extra information (a two byte mask) for each polygon. Actually, the two bytes code a normal quantization for each polygon. During the rendering, each polygon is individually tested by a logical *AND* operation. The advantage of this approach is a simpler integration into existing graphics systems, which usually have their own hierarchical arrangement of the objects.

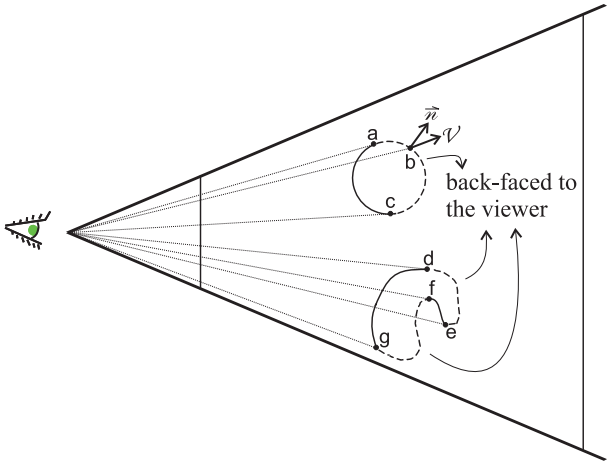


Figure 3.11: The dashed parts of the objects are back-faced to the camera. These back-faced zones are limited by the following pairs of points: $[a, b]$, $[b, c]$, $[d, e]$ and $[f, g]$. All along the dashed zone the angle between the viewing vector and the normal vector is acute (for example, see point b). Remark that the zone $[e, f]$ is front-faced, even if it is invisible as a result of an occlusion (see Section 3.3.2 for occlusion culling).

Chapter 4

Macroscale

The macro-geometry of an object can be represented in multiple ways: by implicit surfaces, parametric surfaces or by an explicit representation (a triangle mesh for example). This is the subject of Section 4.1. In the following sections we review how to accelerate visualization in this macroscale level. Section 4.3 surveys the geometry simplification and multiresolution subjects for displaying acceleration purposes. In Section 4.4 we show methods that replace geometry by images for visualization. Section 4.5 shows methods converting explicit surface representation in volumes to accelerate the visualization.

4.1 Representing Objects Geometry

In 3D computer graphics we want to reproduce, by a two-dimensional image, objects and scenes as the human eye could see them (even if they do not exist in the real world). To achieve this goal, many different tasks are necessary including a way to, mathematically and computationally, represent the objects. In this section we discuss about the geometry and shape description in a *macroscale* level. Describe the geometry is not enough to entirely represent an object since it misses important information about the material, but material properties are the scope of Chapter 5 about *mesoscale* level.

We are interested in describing the geometry of objects in $\mathbb{R}^3$. In general, these objects are decomposed into primitives defined by high-level parameters. We classify the primitives by their type (volumetric or surface) and by their mathematical definition (parametric, implicit or explicit). We finish the section by exposing the conversion problems between the primitives used for modeling and those used for rendering.

4.1.1 Volume and Surface Representations

Volume approach

Objects in real world can be considered as volumes once they have a concrete form, filling a finite part of the space. The volume is a set of points in $\mathbb{R}^3$ and so, to represent an object, we need a way to specify which points belong to it. In practice, complex objects are decomposed into volumetric primitives (the sphere is a basic example of volumetric primitive).

Surface approach

From the point of view that objects are solids, the use of volumetric primitives to represent them is a natural choice. However, to describe the shape of objects, it is enough to understand their boundary. Thus, following this approach, we are not interested anymore

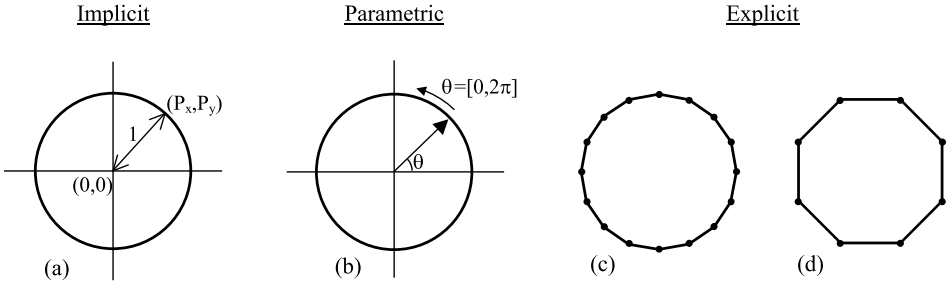


Figure 4.1: The unit circle. (a) Represented implicitly, (b) parametric representation and (c) explicitly represented by 16 points and (d) by 8 points.

in the points inside the object, but in the thin surface that separates the interior points from the exterior points. That is what we call *representation by surfaces.*

A *surface* is a two-dimensional submanifold of the three-dimensional Euclidean space. Actually, the word "surface" can be used to denote an $(n-1)$-dimensional submanifold of an n-dimensional manifold [Wei05], but we consider the restricted meaning for the 2D submanifold case (2-manifold).

To represent complex objects surface, it is necessary to decompose them in multiple surface primitives. The simplest example of surface primitive is a triangle. Note that with the surface representation, we are not restricted to represent only solid objects. For example, a surface composed by triangles can cover a half sphere without its interior (just the peel).

Besides the volumetric/surface classification, the primitives can also be distinguished by its mathematical definition as described in the next section. However, as we concentrate the discussion in objects represented by surfaces, volumetric representation is out of the scope of this thesis.

4.1.2 Primitive Types

We describe three types of primitives: **implicit, parametric** and **explicit**. In the first case, points belonging to the primitives are indirectly specified through a function, defining a containment criterion. In the parametric form, points belonging to the object are given directly by a collection of mapping or parameterizations. Finally, the explicit form is the simpler one: the primitive coordinates are explicitly given without any mathematical function.

For a better understanding of the difference between these types, we explore a simple 2D example: the unit circle in the plane (see Figure 4.1).

Implicit Circle

An implicit equation of the type $F(x,y) = 0$ can describe the unit circle as:

$$F(x,y) = x^2 + y^2 - 1, \quad x,y \in \mathbb{R}$$

The circle is defined by all the points (x,y) that satisfy $F(x,y) = 0$.

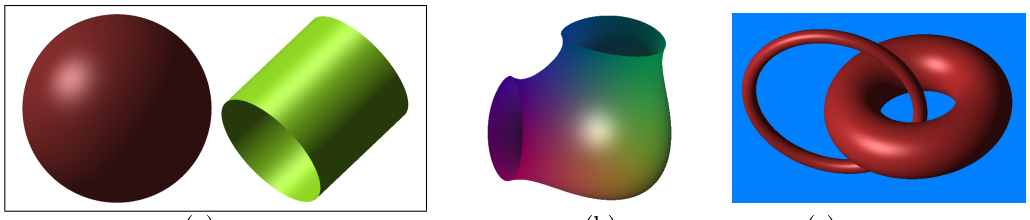
Parameterized Circle

The circle can also be described by a parametric equation $(x,y) = f(\theta)$, where:

$$f(\theta) = (cos\theta, sin\theta), \quad \theta \in [0, 2\pi]$$

The equation maps the interval $[0, 2\pi]$ onto the unit circle.

Explicit Circle



(a) (b) (c)

Figure 4.2: (a) Quadric surfaces: sphere and cylinder; (b) cubic surface; (c) two tori (quartic surfaces).

In the explicit form, a sequential list of N points, with their coordinates explicitly defined, represents the unit circle. We can use the parameterization form written above to help defining the sequence of points P_i as:

$$P_i = (cos(\frac{2\pi}{N} * i), sin(\frac{2\pi}{N} * i)), \quad i \in [0, N-1] \quad \text{and} \quad i, N \in \mathbb{N}$$

It is clear that the explicit representation of the unit circle is just an approximation. The fewer the number of points, the coarser representation we get (see Figure 4.1 c and d). For the circle case, the explicit representation is definitely not a good choice. But, as explained in the end of the section, there are complex objects in which the explicit primitives are the best choice (and sometimes the unique one).

Implicit Primitives

The *Implicit Surface* can be understood as the thin layer between the interior and the exterior of the implicit volume. So the surface is described by an equation of the type $f(x, y, z) = 0$, whose interior volume are all the points respecting $f(x, y, z) < 0$.

In general, the implicit surfaces are defined in polynomial form and they are classified by the polynomial order. A specific surface of degree p (in the family of surfaces of order p) is parametetized by S coefficients, where $S = \binom{p+3}{p}$.

The quadrics (see Figure 4.2(a)), second-order surfaces, are a very important class of surfaces, defining common forms as spheres, cylinders, ellipsoids and saddles (hyperbolic paraboloid). The different quadrics can be represented by their ten coefficients $\{A, B, C, D, E, F, G, H, I, J\}$ in the following equation:

$$f(x, y, z) = Ax^2 + 2Bxy + 2Cxz + 2Dx + Ey^2 + 2Fyz + 2Gy + Hz^2 + 2Iz + J = 0 \quad (4.1)$$

Other higher degrees can be used for defining implicit surfaces. Third degree equations are used to specify the cubics (see Figure 4.2(b)). The number of coefficients parameterizing the cubics is twenty ($S = 20$) and the polynomial equation is:

$$\begin{aligned} C(x, y, z) \quad = \quad & Ax^3 + By^3 + Cz^3 + \\ & Dx^2y + Ex^2z + Fxy^2 + Gy^2z + Hxz^2 + Iyz^2 + Jxyz + \\ & Kx^2 + Ly^2 + Mz^2 + Nxy + Oxz + Pyz + \\ & Qx + Ry + Sz + T = 0 \end{aligned}$$

Another very common implicit surface is the Torus (Figure 4.2(c)) whose equation has degree four, and so, it is classified as a quartic. Considering a torus oriented in direction z, zero centered and with the bigger radius R and the smaller one r, its equation is:

$$T(x, y, z) = (x^2 + y^2 + z^2 - (r^2 + R^2))^2 - 4R^2(r^2 - z^2) = 0 \quad (4.2)$$

As we have already mentioned, the surfaces could be open and could even have an infinity domain. For this reason, sometimes a restriction is imposed for the domain. For example, to enclose the surface inside an imaginary parallelepiped (as the cylinder in Figure 4.2(a)).

There are some advantages in using implicit representation:

- It is very simple to create level sets of these implicit surfaces (especially if compared with the other primitive types described later).
- Some kinds of requests are quickly answered when using implicit surfaces, such as intersections (i.e. with a line or with another implicit surface).
- Another interesting point of the implicit surfaces is the easy evaluation of the normal vector in any point $p = (x, y, z)$ (for surfaces defined by continuous and differentiable functions):

$$\vec{n} = \left[\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}\right] \tag{4.3}$$

Parametric Primitives

Parametric surfaces contain an explicit relation between the 2D and 3D spaces. The parameters are defined in 2D and a parametric equation gives the related points in 3D. This relation is known as *mapping*. For example, any three-dimensional point on a parametric surface is specified by an (u, v) ordered pair:

$$\begin{aligned} f: \mathbb{R}^2 &\rightarrow \mathbb{R}^3 \\ (u, v) &\rightarrow (x(u, v), y(u, v), z(u, v)) \end{aligned} \tag{4.4}$$

where $x(u, v)$, $y(u, v)$, $z(u, v)$ are functions of type $\mathbb{R}^2 \rightarrow \mathbb{R}$. In general, these functions are polynomials of degree three. The *cubic polynomials* are most often used because lower-degree polynomials would not have enough flexibility, while higher-degree polynomials require more computation. The most known parametric surfaces are the *Hermit* surfaces [Bar83], the *Bézier* surfaces [Bez70] and the *B-spline* surfaces [BBB87].

One can easily extracts a polygon mesh (see Section 4.2.2) from a parametric surface by uniformly sampling the parameters to create a set of vertices and connecting them to form the polygons. This operation is often called *tessellation*.

The polynomial parametric surfaces are traditionally more popular than the implicit surfaces in computer graphics. One of the reasons is the continuity control between neighboring surfaces when they are composing a more complex object. There are also some geometric operations for which the parametric surfaces are more convenient. They are easier to tessellate, subdivide and bound, and to perform any operation requiring a knowledge of "where" on the surface [Roc89].

Explicit Primitives

As explained in the beginning of this section, the explicit primitives are expressed directly by their coordinates. Any polygon defined by the coordinates of its vertices in the $\mathbb{R}^3$ is an explicit surface primitive. Generally, only planar polygons are expected (when all the vertices are coplanar), although, in some recent research, non-planar quadrilaterals are used as a primitive [HT04, Flo03]. The assembling of polygons to construct a complex object is known as *polygonal mesh*.

A polygonal mesh is formed by a set of vertices, edges and polygons. This set must respect some rules to be considered a 2-manifold surface. The polygon boundaries are three or more edges and the edges always connect two vertices. A vertex must be shared by at least two edges and an edge must be shared by two adjacent polygons (except when it is in the mesh border, where it is used by only one polygon).

A special polygonal mesh is the *triangle mesh*. Some nice properties appear when using only triangles: they are always convex and planar and the values defined in the vertices are perfectly interpolated inside the triangle (by the barycentric coordinates). Another point is that the current graphics cards are specialized in triangle rasterization, efficiently producing images from triangulated mesh surfaces. Finally, any polygonal mesh can be easily transformed into a triangle mesh, since any polygon (convex or not) can be triangulated [BGR88].

Height Map is a regular explicit surface representation. It is the discretization of a function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, $((x, y) \rightarrow z)$, stored in a 2D regular grid with a height information for each sample. It is not possible to have more than one height (z) information for the same (x, y), that is why this kind of representation can be classified as a two-and-a-half dimension (2,5D). Height maps (or *height fields*) are especially used in terrain representations, but the 2,5D prevents the reproduction of some geological features such as caves. In spite of this limitation, height maps are interesting due to their compactness.

4.2 Macroscale Geometry Visualization

The two most popular visualization algorithms (ray-tracing and z-buffer rasterization) offer different conditions depending on the primitive type.

4.2.1 Ray-tracing/ray-casting the primitives

For the ray-tracing/ray-casting algorithm it is essential to have a ray-primitive intersection algorithm. This procedure will be probably executed thousands of times for each frame, so it should be as efficient as possible. For each pixel in the screen, a ray that contains the camera position and the pixel is shot and tested against the objects in the scene. A ray R with the origin o (which can be the pixel itself) and normalized direction $\vec{v}$ (ex: $\vec{v} = \text{pixel} - \text{camera focus point}$) is described in the following parametric function

$$R: \quad (x, y, z) = o + t\vec{v}\,, \quad t \in \mathbb{R} \tag{4.5}$$

The intersection algorithm running for each pixel must answer the three basic questions:

- Is there an intersection between the ray and the object?
- Where is the closest intersection located?
- What is the surface normal at this point?

For different kinds of objects, different types of intersectors are designed to answer the questions above. In the following text we describe how they work for each primitive type: implicit, parametric and explicit.

Implicit Ray Intersection

To find the intersection between an implicit surface and a ray, we need to substitute x, y and z of Equation 4.5

$$x = x_0 + \vec{v}_x \quad y = y_0 + \vec{v}_y \quad z = z_0 + \vec{v}_z \quad \text{, where } o = [x_0, y_0, z_0],$$

in the implicit function equation, solving the final equation for t.

So, the t equation has the form:

$$a_0 t^p + a_1 t^{p-1} + \ldots + a_p = 0$$

where p is the implicit surface degree.

There are three possible situations: no intersection, one intersection or multiple intersections; depending on the number of roots found for t. For the last case, the smaller positive t is taken to compute the intersection. Using the ray equation (Equations 4.5), the intersection P is computed by:

$$P = o + t\vec{v} \tag{4.6}$$

For quadrics, the intersection computation is simple since it just involves the solution of a quadric equation. For higher order implicit primitives, the intersection computation is harder and sometimes susceptible to numerical problems. Many methods have been proposed for the high-degree algebraic surfaces, such as Newton Raphson's process [GW89], Sturm sequences [vW85], binary search using interval analysis and sphere tracing [Har96]. Those methods are much slower when compared to the direct solution for quadrics, but at least it is possible to ray-casted objects like those in Figure 10.19(b). Note that there are non-polynomial implicit surfaces that cannot be ray traced at all because of its complexity, for example, fractal surfaces.

Parametric Ray Intersection

Among the three primitive types, the parametric one is by far the most difficult type to be ray-casted. Finding the intersection between a ray and a bicubic patch also requires finding the solution of sixth-degree polynomials [CPC84], treating the parametric ray as the intersection of two planes [Hec88]. Numerical techniques were proposed, for example, using two-dimensional Newton iteration. Toth [Tot85] describes a method for finding an interval in which Newton iteration converges. A small survey about the subject can be found in [Han89].

Explicit Ray Intersection

Firstly, let us reduce the polygon-mesh ray-casting problem to a triangle-mesh problem, since any polygon can be decomposed into triangles. Ray casting a triangle is an easy problem divided into two quick steps [Bad90]. The first one is the intersection computation between the ray and the plan. With the intersection point we can do the second step: to verify if the point is inside the triangle (some simple dot and cross products can be executed to realize this test). The computation of triangle-ray intersection is so fast that some RTRT (real-time ray-tracing) systems prefer to use only triangles to describe their scene [CHH02, PBMH02].

For *height map*, the ray-casting algorithms were introduced to display terrains. In his book, LaMothe [LaM95] describes an incremental algorithm where the computation for one pixel helps the next one in the same column. This incremental procedure, lately improved to accept the render of objects over the terrain [SGC97], accelerates the terrain visualization.

4.2.2 Primitives Rasterization

As we have seen in Section 3.1.2, the use of rasterization with a z-buffer is the most popular solution for interactive visualization. The current GPUs are specialized in rasterizing triangles, being able to render millions of them by second. Thus, converting parametric and implicit primitives to polygon meshes is a common practice, known as *tessellation*. In this subsection we explain the rasterization of explicit primitives (*Explicit Rasterization*), we investigate some alternatives for rasterizing implicit primitives (*Implicit Rasterization*) and we expose some tessellation methods for parametric (*Parametric Tessellation*) and implicit surfaces (*Implicit Tessellation*).

Explicit Rasterization

Let us reduce again the polygon meshes to triangle meshes in the rasterization problem. In the next paragraph we shortly describe how the standard triangle rasterization is executed by the GPUs. Note that for *height map* representations, a regular triangular mesh is previously computed.

Inside the modern GPUs, there is a complex *graphics pipeline* needed in a raster-display system. A simplified version, presented in Figure 4.3, describes the pipeline in four stages. In the **Transformation** stage, the vertices are transformed from the object space into the world-coordinate system (additionally, some normal vectors may need to be transformed). The **Lighting** stage depends on the illumination model in use (flat, Gouraud or Phong). The evaluation occurs once per triangle in the flat model, once per vertex for Gouraud shading and once per pixel for Phong shading (in this last case it happens further in the pipeline, during the rasterization stage). The Gouraud shading is the most popular in real-time visualization. In the **Projection** stage the coordinates are finally transformed into the normalized projection coordinates. Together, the projection transformation and the *viewport transformation* determine where the vertices fall in the screen, also computing its z-value used for the z-buffer test. During the **Rasterization**

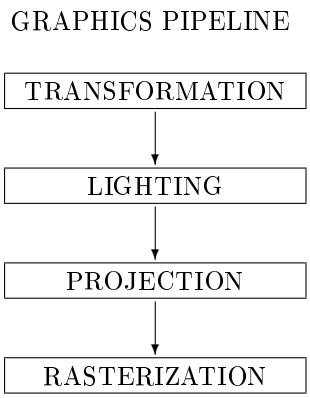


Figure 4.3: A simplified version of the 3D graphics pipeline.

stage, three subtasks are executed: *scan conversion* (determining the pixels covered by the triangle), *shading* (determining which color to assign to each covered pixel), and *z-test* (determining if the pixels are visible or not). In the standard GPU implementation, the pixel color and the z-value are interpolated by the scan-conversion from the vertices values.

Implicit Rasterization

In the literature we can find some essays to directly raster implicit surfaces, and more specifically, quadrics. To achieve this goal, dedicated hardware were proposed, some of them using a quadric interpolation circuit. We shortly describe two different systems, the Pixel Planes [FPE+89] and the one developed by the LIFL group [LNFC95, Fro96, LAP96]. The first one suggests the use of spheres and cylinders as primitives. For the scan conversion step (subtask that determines the pixels covered by the primitive), Pixel Planes use a combination of circles and straight line to define the boundary of the primitive projection. In their system, there is no *square root* implemented in hardware, so, spheres are approximate to parabolics and the cylinders by parabolic cylinders during the rasterization, see Figure 4.4. These approximations result in incorrect intersections and may be the reason why the project was discontinued. The LIFL group tried a more generic approach, accepting any type of quadrics as primitives due to the implementation of square root in hardware. The quadrics could be limited by planes. Actually, they deal with three types of primitives [LAP96]: volumetric quadrics, quadric patches and convex polyhedrons (a polyhedron is defined by a collection of planes without a quadric). The scan conversion was based on the conic contour (formed by the projection of the quadric silhouette). Their main problem is the primitive type restriction: using only (or almost only) quadrics is not reliable for modeling complex scenes. To overcome this limitation, they have worked in quadrics approximation of more complex surfaces [Fro96], but there is still no final solution.

Parametric Tessellation

The parametric surfaces have a natural bidimensional parameters (u, v) mapping points in $\mathbb{R}^3$. A simple procedure for tessellating parametric surfaces is done by iterative evaluation of the bicubic polynomials [FvDFH90]. For a given constant δ the surface is evaluated in $1/\delta^2$ points (for $\delta = 0.1$, it would be 100 points), defining a regular grid with $(1/\delta - 1)^2$ quadrilaterals. To obtain a triangle mesh the quadrilaterals must be decomposed into two or four triangles. In a more sophisticated procedure, it is possible to produce an adaptive tessellation of parametric surfaces. In this case the surface is recursively cut into four parts until a flatness threshold is achieved. Special care is necessary for objects

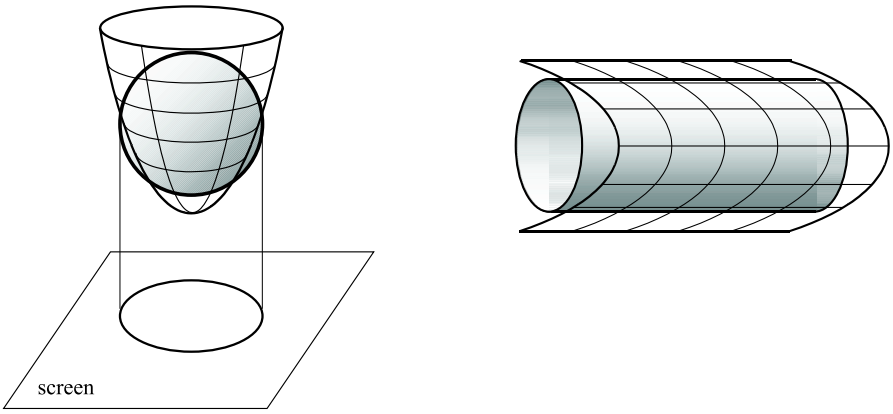


Figure 4.4: Sphere and cylinder approximations in Pixel Planes approach.

composed by a network of parametric patches: neighbor patches must have the same resolution in the border to avoid cracks.

Implicit Tessellation

The tessellation goal is to find a polygonal approximation to implicit primitive surfaces. There are several synonyms for the tessellation operation: *tilling, polygonization, triangulation, tracing* and *meshing*. Unfortunately, the tessellation for implicit form is not as straightforward as for parametric patches. For quadrics and some other implicit surfaces (as the torus), it is simpler to convert them to their parametric equivalent, but it is difficult or impossible to do so for higher-order functions. A generic approach for implicit tessellation starts by executing a *spatial partitioning*: a three-dimensional volume with semidisjoint cells that completely enclose the surface. Then, only the transverse cells are taken, that means, only cells intersecting the implicit surface. The intersections of the cell edges with the implicit surface are the vertices of the target polygon mesh. To connect these vertices the *Marching Cube* algorithm [LC87], or any of its variants, can be used. However, all these steps must respect some mathematical rules to work (see details in Velho et al. book [VdFG98], Chapter 10). An interesting short survey about the tessellation algorithms can be found in Bloomenthal's book [BW97] Chapter 4.

4.2.3 Macroscale Geometry Shading

In any visualization system we are interested in the color of each pixel in the final image. As we already discussed in Section 3.1, the HSR algorithms determine which object is the visible one for each pixel. However, the *lighting models* are responsible for obtaining the color in a surface point, given the surfaces characteristics (position, orientation and reflection properties) and the light sources illuminating them.

Diffuse and Specular Reflections

The diffuse shading model asserts that a point P over a surface sends a constant quantity of light to all directions. As we will see later, this is an approximation and it does not consider anisotropic surfaces. The intensity of diffusing light is measured by using the following equation:

$$I_d = I_l K_d(\vec{l} \cdot \vec{n}), \tag{4.7}$$

where I_l is the intensity of one light source, $K_d \in [0,1]$ is the material diffuse coefficient, $\vec{l}$ denotes a normalized vector from the point P to the light position and $\vec{n}$ is the normalized vector of the surface normal at point P (see Figure 4.5).

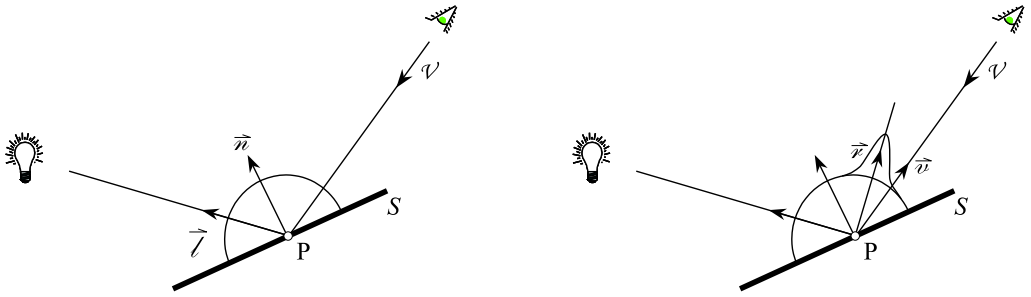


Figure 4.5: Diffuse Model: a point P emits the same quantity of energy in all directions (left picture). The intensity depends on the light source and also on the angle between the vectors $\vec{n}$ (the surface normal at point P) and $\vec{l}$ (the light direction). (*right picture:*) The specular component is very intense when it is close to the reflected light direction. In a given direction $\vec{v}$ the intensity of the specular component depends on the angle between $\vec{v}$ and $\vec{r}$.

Besides the diffuse component, objects can also have a reflection property responsible for the specular halo, which is commonly seen in metallic surfaces, for example. This component depends on the viewing angle. The resulted specular light intensity I_s in one direction is computed by:

$$I_s = I_l K_s (\vec{r} \cdot \vec{v})^n, \tag{4.8}$$

where $K_s \in [0,1]$ is the material specular coefficient, $\vec{r}$ is the reflected normalized vector, $\vec{v}$ is the unitary vector opposite to the viewing direction $\mathcal{V}$ and n is the specular-reflection parameter (values of n typically vary from 1 to several hundred [FvDFH90]).

Finally, an additional *ambient coefficient* is commonly used to give a minimal level of brightness for a scene. Therefore, parts of objects not exposed directly to a light will still be visible. Using Equations 4.7 and 4.8, the final light intensity I_f of a point P on the surface S through the direction $\mathcal{V}$ (check again Figure 4.5) can be computed by the equation:

$$I_f = I_a K_a + I_l K_d (\vec{l} \cdot \vec{n}) + I_l K_s (\vec{r} \cdot \vec{v})^n, \tag{4.9}$$

where I_a is the intensity of the ambient light and K_a is the material ambient coefficient.

Equation 4.9 is an approximation that works for monochromatic light. For systems where the color is represented by multiple isolated components, for example the *RGB* system (red, green and blue), the equation must be applied separately for each one so that we obtain the final color.

Polygon Mesh Shading

In a 3D rendering application, we try to reproduce in the screen the result of the light coming from the objects in the viewing direction under some illumination conditions. There are three shading methods used for rendering polygon meshes: flat shading; Gouraud shading [Gou71a, Gou71b] and Phong shading [Pho75]. All these shadings based their computations on the diffuse/specular model explained above (Equation 4.9).

As we have seen, the shading result depends on some surfaces properties (as the material diffuse coefficient) and the surface normal, which can vary along the surface domain. However, once the surface is discretized into a polygon mesh, the properties and normal values are stored either by vertex or by face. For simplification, let us consider that the surface properties are constant and only the normal varies. In *flat shading*, the normals are stored by face and the diffuse/specular model resulting in a unique flat color for each face. The Gouraud and Phong methods for polygon mesh shading assume that the normals are stored by vertices (for example using the weighted average normal of the adjacent faces). In the Gouraud shading technique, the diffuse/specular model

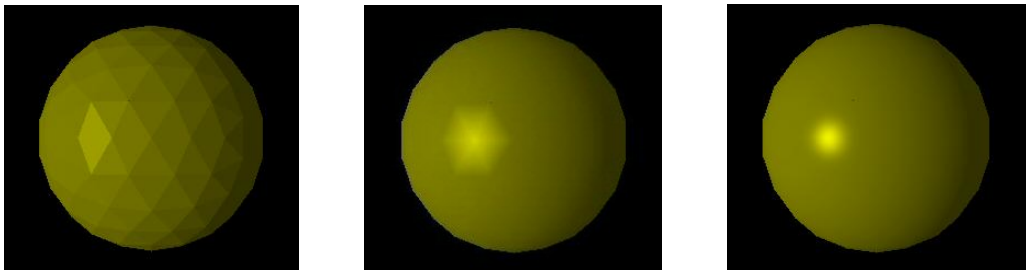


Figure 4.6: Polygon-mesh sphere with three different shading models: flat, Gouraud and Phong.

is computed by vertex and then the result is linearly interpolated inside the polygons. Considering the three methods mentioned above, the Phong method is the most accurate. This method linearly interpolates the normal vector inside the polygon domain. Then, for each pixel, the diffuse/specular model is applied. See comparative images in Figure 4.6.

When using only the geometry, we are limited to approximations since it is impossible to specify details inside each polygon. To increase results we need either a mesostructure representation, discussed in Section 5, or more sophisticated lighting models, discussed in Section 6.

4.2.4 Which one is the best representation?

The choice from one of the three formerly mentioned primitive types (implicit, explicit and parametric) is related to several different factors. It depends on the nature of the object itself, the data input method, the modeling procedure, the software used to manipulate the object and finally the technique chosen for rendering.

Depending on the object nature

The nature of the object is determinant for the quality of its representation, as we saw with the unit circle example in the beginning of this section. Spheres have the same problem of the unit circle when represented explicitly. Actually, all objects with smooth faces cannot be perfectly represented by a finite collection of polygons with explicit coordinates. So, the explicit sphere problem would also happen with almost all quadrics, cubics and smooth Bezier patches. However, the explicit representation is well suited for another kind of objects, for example, scanned ones.

When scanning an object for a three-dimensional reconstruction, the system captures a collection of points from the real model. This collection is used for creating a polygon mesh. Since the original information is explicitly acquired, the explicit representation is the natural and simplest choice. Some of the techniques for digitizing objects can yield millions (or even billions) of point locations. Building a polygon mesh representation of the object from this enormous collection of points is a non-trivial task and it has been focus of study in previous work [HDD+94].

Whereas, the lacking of a mathematical description is a limitation for explicit representations. There is a set of operations that cannot be directly applied over the polygon meshes (e.g., smooth deformation and level sets). That is the reason why a lot of research was dedicated to transform polygonal meshes into either implicit representations [WK05] or parametric ones [RLL+05].

Depending on the visualization algorithm

As we have shown in Sections 4.2.1 and 4.2.2, rasterizing and ray-casting have different behavior depending on the primitive type (implicit, parameterized or explicit). Table 4.1

shows the summary.

Primitive	Visualization System	
	Ray-casting or **Ray-tracing**	**Rasterization** and **Z-buffer**
Explicit	easy to implement fast rendering	easy to implement very fast rendering
Implicit *	easy to implement fast rendering	very hard to implement directly solution: tessellation
Parametric	hard to implement slow rendering	never implemented solution: tessellation

Table 4.1: This table presents how the two most popular visualization algorithms deal with each kind of surface primitive. (*) Considering low-order implicit surfaces, such as quadrics.

Nowadays, rasterization is the first choice for interactive visualization systems. The rasterization of explicit primitives is implemented in hardware and it is present in almost any commodity graphics card. However, implicit and parametric surfaces are not suitable for this kind of visualization. For rasterizing these surfaces it is mandatory a conversion to an explicit representation by means of a tessellation algorithm. The conversion permits fast displaying since every surface uses the hardware-implemented triangle rasterization. However it also brings some quality problems. It is impossible to perfectly reproduce smooth surfaces with explicit representation using triangle or polygon meshes.

Let us take the sphere as an example. The most appropriate representation for spheres is their implicit representation, based on the object nature. However, most of interactive visualization systems convert the sphere to a polygonal explicit representation (a triangle mesh, for example). This representation is not perfect since it is an approximation (see Figure 4.7). To compensate this problem, one can refine the tessellation. The more triangles are used the more quality is obtained in this approximation. However the rendering can suffer a slow-down due to an excessive number of triangles.

For implicit surfaces, there is also another visualization method that is fast and does not need any conversion: the ray-tracing/ray-casting algorithm. This is not true for parametric surfaces. For this reason, one of our contribution is about implicit surfaces:

Thesis proposition: Visualization method adapted to primitive nature
We propose a visualization system combining ray-casting and rasterization algorithms depending on the primitive type. For explicit representations (i.e., triangle meshes) the system uses rasterization and for implicit representations (e.g., spheres, quadrics and tori) the system uses ray-casting. We have obtained interactive frame rates by implementing this system in current programmable GPU. We prove that for complex scenes, with numerous implicit objects, the gains in speed and quality are substantials (see Chapter III about manufactured objects and scenes).

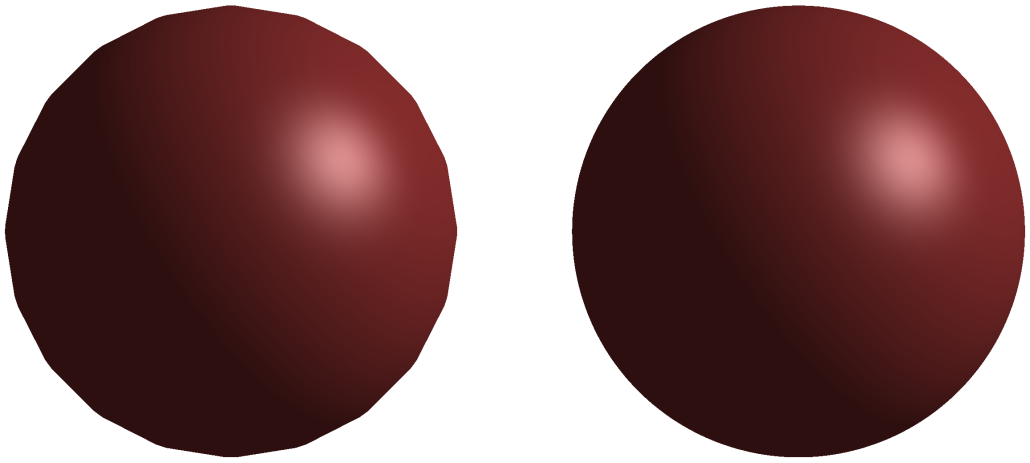


Figure 4.7: *Left:* Even with the Phong model, which best approximates the physics of a real shading, the silhouette is still a problem when representing the sphere by a polygon mesh (320 triangles in this example). *Right:* In this picture the sphere is visualized by ray-casting directly from its implicit representation.

4.3 Geometry Simplification

In the geometry simplification approach, objects represented by meshes are replaced by simpler meshes (i.e., with fewer primitives). The main idea is based on the observation that models with too many primitives (triangles for instance) sometimes are incompatible with the screen-resolution. For example, an object far from the observer, in a perspective projection, will fill just a small part of the screen (for example 10x10 pixels), even though its mesh can contain thousands of triangles. An excessive number of primitives, sometimes with many of them projecting an area much smaller than one pixel, should clearly be avoided. For that purpose, the simplification methods reduce the number of primitives to represent the same original objects. Before explaining each method, some general concepts are listed below.

Mesh Resolution is directly proportional to the number of primitives or, more usually, to the number of vertices (actually the number of vertices and faces are directly proportional due to the *Euler characteristic*[4]). By convention, a mesh M^i is finer than M^j if $(i > j)$. Coarser meshes are just an approximation of the original object, so they always contain an error which, in general, is inversely proportional to the resolution. The acceleration algorithms based on mesh simplification can extract a finite number of meshes M^i which are a simplification of the original mesh M^n $(i < n)$. This collection of meshes is known as *Levels of Detail* or LOD and the structure that contains the LOD is called a *multiresolution structure* (see [MH99a]). A very well known multiresolution structure is the *Progressive Mesh* [Hop96], which has a continuous representation for the different mesh resolutions. We discuss more about the progressive meshes later in the text (see page 77).

Multiresolution and Multiscale are different concepts. Multiscale has a wider domain; we separate the visualization problem in four different scale levels: scene-scale, macroscale (or object-scale), mesoscale and microscale (or photon-scale). Multiresolution concerns only the macroscale level. This concept is only related to the macro-geometry LOD.

Regular and Irregular Meshes differ in their topology. While in the first one the topology is the same all over the domain, in the second one it can vary from vertex to

[4] *Euler characteristic*: $V + F = E + 2$, where V is the number of vertices on a manifold polygonal mesh, F is the number of faces and E is the number of edges.

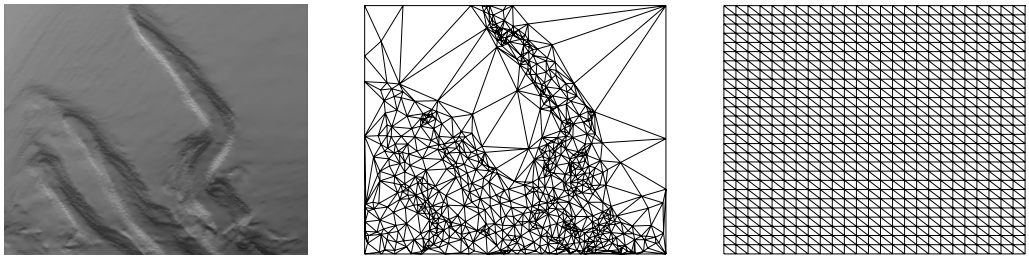


Figure 4.8: (a) A terrain data: (b) represented by an irregular mesh and (c) by a regular one, both with about 1500 triangles. The advantage of irregular meshes is their best adaptation to geometric features. On the other hand, the advantage of regular meshes is their natural parametric structure.

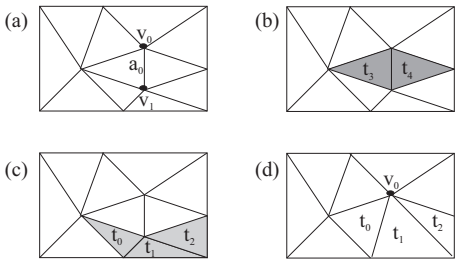


Figure 4.9: Edge collapse. (a) The edge a_0 is collapsed, as well as the vertex v_1, while the vertex v_0 does not change. (b) The triangles t_3 and t_4 are collapsed. (c) The triangles t_0, t_1 and t_2 will be directly connected with v_0 and no longer to v_1. (d) The collapse result.

vertex (see Figure 4.8), with no restriction on vertex valence. The first advantage of the regularity is its natural parametric structure. In the simplification domain, another advantage is the possibility of global operations for decimating the polygons. The drawback of regular meshes is the lack of flexibility due to their constrained structure. The irregular meshes can better adapt to complex geometric features at many scales. Complex scanned objects (as the *Happy Budha*, see Figure 2.2 on page 34) are normally represented by irregular meshes because of the unorganized position of the captured samples. Irregular meshes exclusively composed of triangles, called TIN (Triangular Irregular Network), are often used as input in the techniques later described in this section.

Decimation Operation is the operation recursively executed by the decimation algorithm. In a first step the operation is done over the original mesh M^n to obtain a mesh M^{n-1}. Each following operation i will be applied over the previous mesh, obtaining M^{n-i} from M^{n-i+1}. The decimation operations can be classified as global or local operations. The global ones are more often associated with regular meshes and they reduce drastically the mesh resolution (for example dividing the number of vertices by two). The local operations are finer, excluding only one or two vertices each time. A classical example is the *edge-collapse* operation shown in Figure 4.9.

Mesh decimation algorithms (task 1 in Figure 4.10)

The decimation algorithms are responsible for generating the multiresolution structure. Based on a pre-defined metric, the original mesh M^n is decimated to meshes with lower resolution representing the same object. The result is recorded in a *multiresolution structure* from which the meshes can be extracted at rendering time.

The decimation methods can vary in several ways:

- *Global or local decimation operation.*

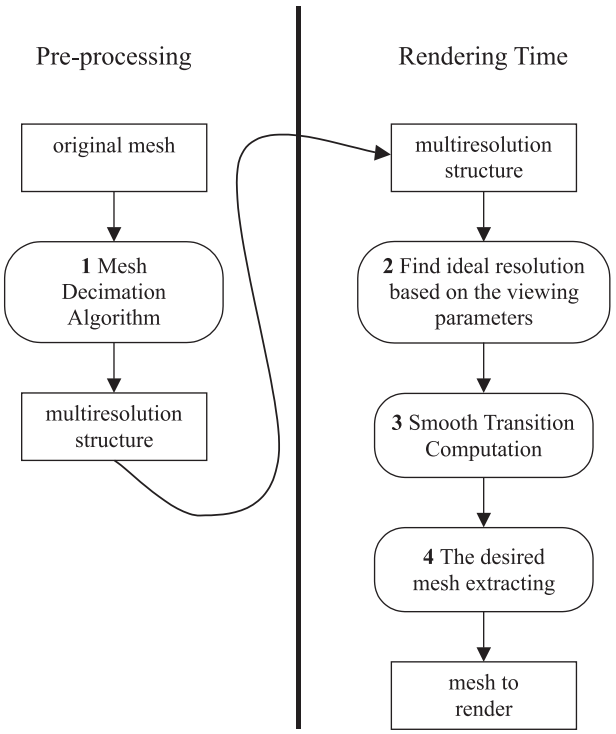


Figure 4.10: The four main tasks in applications for multiresolution meshes. Each task is detailed in the text.

- *Global or local error metric.* In the global approach the total difference between the original mesh and the coarser one is computed as the error. The local-error metric is normally associated with local decimation algorithms.
- *Geometric errors.* Many different geometric errors can be considered. Some of them are more appropriate only for global or local error metrics, but in general they can be associated with both. The following list contains the most common geometric errors considered during the *mesh decimation algorithms*:
  - surface-surface distances
  - vertex-vertex distances
  - point-surface distances
  - local curvature
  - volume preservation
  - edge norm (can be used as an error for edge-collapsing)
- *Vertex placement.* In some algorithms, the vertices of a simpler mesh always form a subset of the original vertices, while other algorithms prefer a repositioning strategy in order to minimize the error. The advantages of *subset placement* are less memory consumption and a simpler multiresolution structure for further extraction, while *optimal placement* tends to result in better approximations.
- *Attribute error consideration (optional).* Some authors also consider other attributes than the geometry. For example, one can consider the error related to the disparity of colors, normals or texture coordinates between the original object and its simpler representation. Some important work was developed on this subject, using appearance preservation as the criterion to simplify the polygonal models [COM98, GH98, Hop99]. The Garland and Heckbert work [GH98], as well as Hoppe work [Hop99], were based on their previous work about the *quadric-error metric* [GH97].

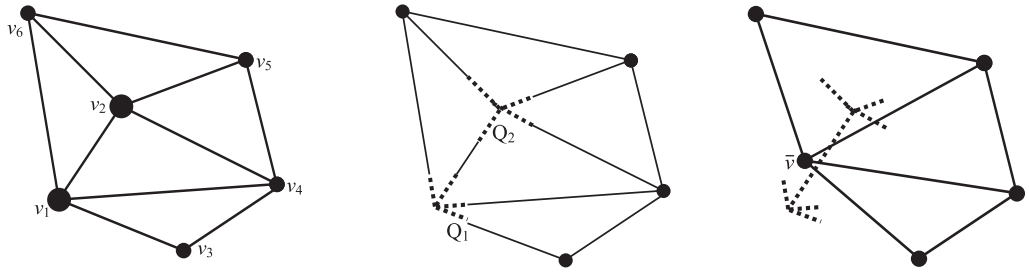


Figure 4.11: An example of a *Quadric Error Metric* pair contraction [GH97]. *Left*: The original mesh, the vertices v_1 and v_2 are about to be contracted. *Center*: The quadric error matrices Q_1 and Q_2 of each vertex encode the quadric error associated to the planes passing by the vertices. $Q_n = \sum_{p \in planes(v_n)} K_p$, where $planes(v_n)$ are the planes of the triangles containing vertex v_n. *Right*: The new quadric error after contraction is computed by addition ($\bar{Q} = Q_1 + Q_2$), the position of $\bar{v}$ is computed to minimize the error $\Delta(\bar{v}) = [\bar{v}\ 1]^\mathsf{T} \bar{Q} [\bar{v}\ 1]$. Note that there are two planes that counts twice in $\bar{Q}$ (the planes containing $\{v_1, v_2, v_3\}$ and $\{v_1, v_2, v_6\}$). However, Garland and Heckbert point out that although it may introduce some imprecision, a plane can be added at most 3 three times during the decimation process (one for each of its vertices) [GH97].

In the initial work about quadric-error metric (*QEM* [GH97]), Garland and Heckbert use a local operation for their simplification named *pair contraction*, which is actually a generalization of *edge collapse*, accepting any arbitrary pair of vertices. They choose an *optimal placement* strategy; therefore, when two vertices are contracted, they place the new vertex where it minimizes the error. An initial error quadric Q (a four by four matrix) is associated with each vertex of the original model. Indeed, Q encodes the sum of squared Euclidean distance to a set of planes for any point v in space, so the error is $\Delta(v) = [v\ 1]^\mathsf{T} Q [v\ 1]$. To understand Q, let us take first an isolated plane $p = [a\ b\ c\ d]^\mathsf{T}$ defining the equation $ax + by + cz + d = 0$. The associated *fundamental error quadric* $K_p = pp^\mathsf{T}$ can be used to compute the squared Euclidean distance from this plane to any point v using $[v\ 1]^\mathsf{T} K_p [v\ 1]$. The initial Q associated to each vertex is the sum of K_p for all the planes of incident triangles. Note that the initial error is 0, since each vertex lies exactly in the triangle planes. After a contraction $(v_1, v_2) \rightarrow \bar{v}$, an additive rule is used to compute the new matrix $\bar{Q}$ which approximates the error at $\bar{v}$: $\bar{Q} = Q_1 + Q_2$. Finally, to find the ideal space position for $\bar{v}$ they solve a very simple linear system to minimize $\Delta(\bar{v}) = [\bar{v}\ 1]^\mathsf{T} \bar{Q} [\bar{v}\ 1]$. In Figure 4.11 we explain the pair-contraction steps. In their next work [GH98], Garland and Heckbert extended the geometric concept to also consider attribute errors (e.g., colors, texture coordinates and normals). Each vertex is treated as a vector $v \in \mathbb{R}^{3+m}$, the first 3 components of v are the spatial coordinates and the remaining m components are the property values. In this condition, the error to be minimized also contains attribute information. Subsequently, Hoppe [Hop99] improved the algorithm, requiring less storage, with quicker evaluation and more accurate simplified meshes, capturing attribute discontinuities efficiently.

Clustering is another way to compute a simpler geometric approximation for a complex input mesh. A trivial approach is an *uniform vertex clustering*, in which a group of vertices spatially closed are substitute by only one vertex (see comparisons of this method and the QEM in [GH97]). In general, clustering or partitioning methods disregard the original topology of the input surface, which is interesting for drastic simplifications [CSAD04]. Interesting results were recently achieved by Cohen-Steiner et. al. [CSAD04], who introduced the notions of variational partitions, shape proxies and the $\mathcal{L}^{1,2}$ metric (based on $\mathcal{L}^2$ measure of the normal field). In their work, the proxies are a set of planes, each one approximating a region of the original data. Given an error metric E, a number k of proxies, and an input geometry S, they can find a partitioning $\mathcal{R}$ of S

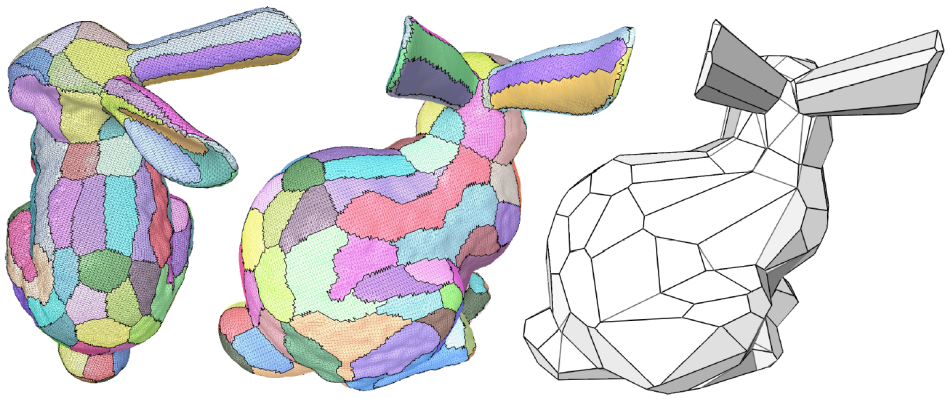


Figure 4.12: Variational Shape Approximation [CSAD04] applied on the bunny mesh.

in k regions and its respective set P of optimal proxies that minimizes $E(\mathcal{R}, P)$. The obtained partitioning (see Figure 4.12) is a *draft* for remeshing. Then the adjacency graph defines the connectivity used to find the vertices and edges forming the final approximation mesh. In [WK05], Wu and Kobbelt extended *Variational Shape Approximation* to also accept spheres, cylinders and rolling-ball blend patches as proxies (in the original work only planes could be proxies [CSAD04]). Their results are impressive, especially when applied in CAD models.

Similarities to our work:
In both cited work about Variational Shape Approximation [CSAD04, WK05] the original primitives are grouped and replaced by a higher-level representation: planes, spheres and cylinders. In this sense, our method about manufactured objects and scenes (see Chapters 9, 10 and 11) is comparable to their work. Our proposition includes the substitution of primitive groups (triangles, for instance) by a higher-level representation: cylinders, torus slices, spheres and cones. After a *reverse engineering* step (Chapter 9), these new representations are used for rendering, resulting in speed-up, more quality and less memory space than the original representation.

Tarini et al. [TCS03] introduce a set of measures to rate the performance of attribute preserving between a simplified mesh M_L and a high resolution mesh M_H. Given M_H and M_L it is necessary to use a function $\mathcal{F}$ for mapping any point p' of M_L in a point p of M_H.

$$p_i = \mathcal{F}(p'_i);\text{, where } p_i \in M_H \text{ and } p'_i \in M_L$$

With $\mathcal{F}$ it is possible to recover the details (e.g. colors, normals and texture coordinates) of M_H in M_L. They developed two view-dependent measurement tools (one in object space and the other in image space) to compare different mapping functions. They do the comparison of three mappings: proximity based; ray-casting along normal direction; and visibility based (this last one is also one of their novel contributions). These general measures can be very interesting to compare the appearance preservation of the extensive number of simplification methods.

Find ideal resolution based on viewing parameters (task 2 in Figure 4.10)

This rendering-time task decides which LOD should be used, based on the camera parameters. Distance is the simplest criterion for this task. The application chooses a coarse mesh when the camera is far and a finer one when the camera is close. Another approach

is the *screen-space geometric error*, probably the most popular one. Many formulas were introduced in the literature for the screen-space geometric error. The simplest formulas just project into the screen the object-space geometric error used in the preprocessing step, obtaining its real size in the final image. Actually, the different criteria itemized below can be combined:

- distance from the observer;
- screen-space geometric error;
- silhouette preserving;
- relative speed: if the observer is moving quickly a coarser resolution can be used since the human eye will not be able to capture the details;
- distance from the center of projection: users have the natural intuition to fix the focus on the center of the image, so this can also be a criterion.

Some methods described later use a view-dependent approach to choose the ideal resolution. This means that the resolution should vary along the mesh domain. An *adaptive-function* is determined based on some of the criteria listed above. The view-dependent methods are more complicated to implement but they are a very important solution for large size objects.

Smooth transition computation (task 3 in Figure 4.10)

If the mesh resolution changes abruptly in consecutive frames, the user will probably notice a *popping* in the image. To avoid this undesirable perception, a smooth transition should be executed along some frames. A simple method is the use of transparency. More complex solutions control the evolution of vertices and their properties between different LOD. According to the kind of complexity, this task and the previous task 2 are coded together.

Extract desired mesh (task 4 in Figure 4.10)

This last task of the pipeline extracts the mesh from the multiresolution structure based on the results of the previous task. The algorithm is completely related to how the structure is represented.

Much work has been done in this area and the following text describes the most important algorithms in an increasing order of complexity. We start by two view-independent simplification classes (*Discrete LOD* and *Progressive Meshes*), in sequence we revise the *View-dependent LOD* and we end with the *Nested Progressive Meshes*, which are also view-dependent.

4.3.1 Discrete LOD

This is the simplest LOD (levels of detail) technique. Given a polygonal model as input, the simplification algorithm will generate a sequence of polygonal approximations with different resolutions (see Figure 4.13). Any decimation algorithm can be used to create the objects levels of detail.

At rendering time, the most appropriate resolution according to the distance of the camera will be used. For example, the user can associate to each resolution the maximum and minimum distances at which the LOD should be used.

The biggest disadvantage of this static LOD is the risk of popping when changing the resolution of a visible object. When this happens the user suddenly observes a significant difference in the model details. A way to avoid popping is the use of blending between different resolutions. During some frames both resolutions (the current and

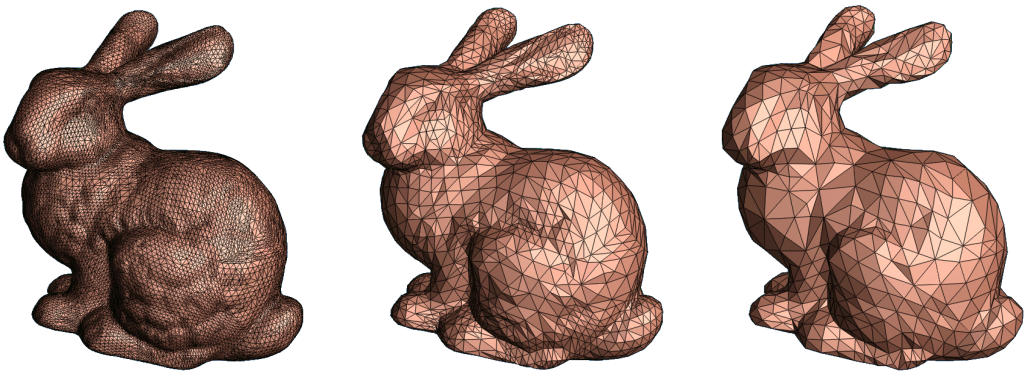


Figure 4.13: The same model in three different resolutions. The first one is the higher LOD and the last one the coarser.

(a) $\begin{pmatrix}\text{Original}\\ \text{Mesh}\end{pmatrix}$ $M^n \xrightarrow{\phi^1} M^{n-1} \xrightarrow{\phi^2} M^{n-2} \xrightarrow{\phi^3} \dots \xrightarrow{\phi^n} M^0$ $\begin{pmatrix}\text{Coarsest}\\ \text{Mesh}\end{pmatrix}$

(b) $\begin{pmatrix}\text{Coarsest}\\ \text{Mesh}\end{pmatrix}$ $M^0 \xrightarrow{\varphi^1} M^1 \xrightarrow{\varphi^2} M^2 \xrightarrow{\varphi^3} \dots \xrightarrow{\varphi^n} M^n$ $\begin{pmatrix}\text{Original}\\ \text{Mesh}\end{pmatrix}$

Figure 4.14: Progressive Meshes. (a) Sequence of meshes M^i in a progressive mesh structure and the associated n simplification operations ϕ^i, ordered as in their creation (M^n is the original mesh and M^0 is the coarsest mesh). (b) The same sequence in the inverse order (the inverse operation φ^i is called *vertex split*). Inverse ordering is advantageous for less memory consumption. It is only necessary to store the coarsest mesh (which is smaller than the original one) and the *vertex split* operations. Another advantage is that the time to extract a mesh M^i is proportional to its resolution.

the previous ones) are rendered with some transparency by the α-opacity control, which varies throughout the transition. [5]

4.3.2 Progressive meshes

First introduced in 1996 by Hugues Hoppe [Hop96], this kind of mesh can be constructed by using a group of simplification operations with local modifications. The preprocessing starts from the original mesh, executing an *edge collapse* (see Figure 4.9 on page 72) at each step, until the desired coarsest mesh is achieved. Instead of edge collapse operations, some implementations could consider another very similar operation, *vertex elimination*, followed by a local retriangulation (see [GH95]). Keeping the set of operations in order, one is able to extract a mesh in real-time and in any resolution between the coarsest and the finest ones (as opposed to the previous *discrete LOD*, which just offers some fixed resolutions).

This sequential construction brings an advantage for storing the data. The progressive mesh can be recorded as the original mesh with the sequence of decimation operations. Indeed, Hoppe [Hop96] only stores the coarse mesh and the inverse operations of edge collapse (see Figure 4.14).

Hoppe also introduces the *geomorphing* concept, used to avoid popping between different mesh resolutions. When the visualization system decides to change the mesh

[5] The simple use of transparency, even with just one fixed object representation, is also considered an acceleration method, named *alpha LOD* [MH99a]. As the distance between object and camera increases, its transparency also increases (α gets lower). The user defines the distance at which the transparency should start and the distance at which it is far enough so the object can completely disappear (α is zero) and does not need to be rendered anymore. The advantages of this method are the implementation simplicity and the continuity in visualization (no popping is expected).

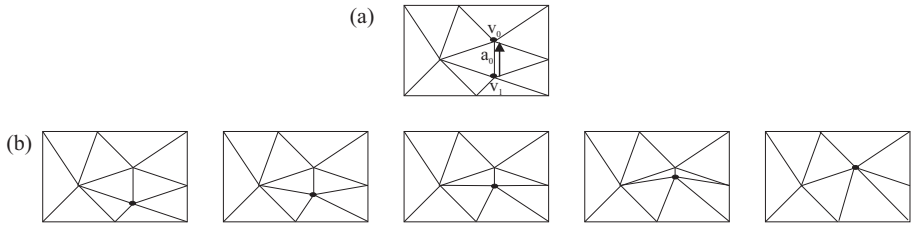


Figure 4.15: (a) An edge collapse operation will be done to reduce the mesh resolution. The vertex V_1 will be collapsed with the edge a_0. (b) The operation will be executed in 5 consecutive frames by smoothly displacing V_1 position. This is called *geomorphing*.

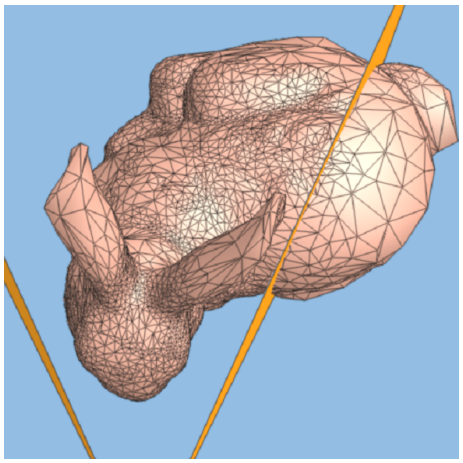


Figure 4.16: The bunny mesh with variable resolution adapted to the viewer position (from Hoppe's work [Hop97]).

resolution, from mesh M^i to mesh M^j for $(i > j)$ (i.e., reducing the resolution), all the collapsed vertices will be smoothly displaced until their final position after the operation (see Figure 4.15). The same happens in vertex splitting when $i < j$ (increasing resolution). This displacement can take an amount of time defined by the user or a fixed number of frames.

4.3.3 View-dependent LOD

The original *progressive mesh* is a great technique for fast extraction of meshes of different resolutions, but it is limited. It is based on pre-defining the order of local operations. At rendering time it is impossible to extract view-dependent meshes, since the order must be respected. In *View-dependent Refinement of Progressive Meshes* [Hop97], Hoppe describes a way to break the pre-defined operations order by controlling their dependencies. A DAG (Directed Acyclic Graph), created during preprocessing, specifies the order in which operations can be applied (in the original progressive meshes the graph would simply be a sequential list like in Figure 4.14). The ideal resolution is based on an *adaptive function*, which depends on some real-time criteria: view frustum, surface orientation, and screen-space geometric error.

Xia et al. [XESV97] developed an approach similar to that of Hoppe, which was subsequently applied to terrains [Hop98]. The application can extract a terrain mesh with a variable resolution. The popping between frames is also avoided by means of geomorphing.

Another view-dependent method, introduced by Luebke and Erikson [LE97], uses a vertex tree to control the order of vertex merging operations. The tree is computed in

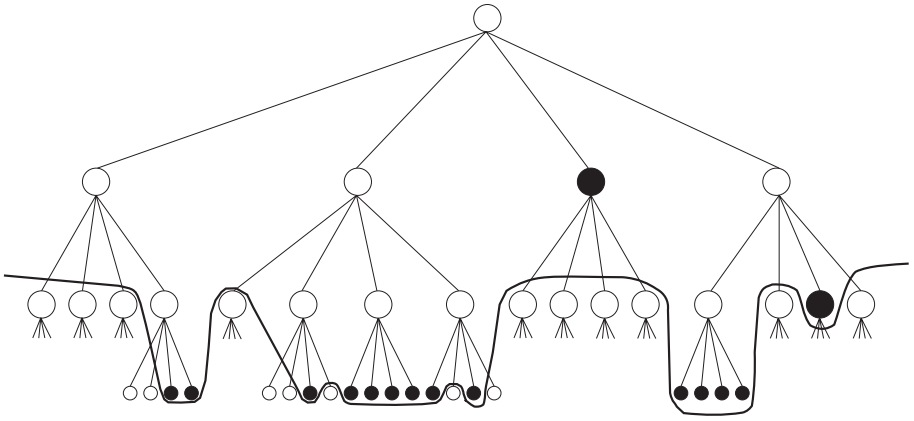


Figure 4.17: A cut in the hierarchical structure. The leaf nodes of the cut tree must be rendered (nodes in black). For each node there is a corresponding PM (progressive mesh). The cut nodes (under the cutting line) were eliminated either by frustum culling or because the parent had already been chosen (so its domain is covered).

a preprocessing step using any kind of simplification method. In real-time, the vertices are labeled as *active/inactive* according to their fold or unfold state. The method uses a screen-error threshold and silhouette preservation criteria to decide the refinement.

4.3.4 Nested progressive meshes

The *view-dependent progressive meshes* broke the limitations of PMs (*progressive meshes*) by making it possible to vary the resolution along the domain. However, it is harder to implement due to the dependency control over the local operations, which may reduce one of the biggest advantages of PMs, the simplicity.

For that reason some authors [YSGM04, TGV01] proposed the use of a set of progressive meshes to represent an object or a scene. By controlling the resolution of each PM, it is possible to have a global mesh with variable resolution to produce view-dependent rendering. However, the application must control the discontinuity between PM neighbors to avoid cracks. The PM set is hierarchically organized, so offering a first-step of refinement (the second-step is the PM itself). The first step produces a cut in the hierarchical structure to assign the nodes that must be rendered (see Figure 4.17). The next step defines the level of detail of each PM associated to each node.

In our previous work [TGV01, Tol00] terrains are visualized in real-time using a quadtree with one PM for each node. The quadtree as a hierarchy structure is a natural choice for terrain visualization, since the domain is 2D. For compatibility between neighboring PMs, boundary constraints are imposed during their construction. This means that the coarsest mesh (M^0 in Figure 4.14 on page 77) always contains the same boundary as the finest one (M^n in Figure 4.14), only the interior is simplified. Our tests have been done over very large terrains and the results show that it is possible to render them in real-time using commodity PCs (see Figure 4.18).

The Quick-VDR application [YSGM04] can visualize general massive models in real-time. Their hierarchical structure is a binary tree of clusters, which are spatially localized PM regions. They take advantage of their hierarchy to also produce occlusion culling and out-of-core pre-selection. They were able to render massive CAD, isosurface and scanned models, consisting of tens of millions of triangles at 10-35 fps on a desktop PC.

In both work (Yoon et al. [YSGM04] and Toledo et al. [TGV01]) the smooth transition when changing resolution of a single PM is assured by geomorphing. However, the changing resolution between different hierarchical levels can only be continuous if the following rule is respected during construction: the highest resolution mesh (M^n) of any

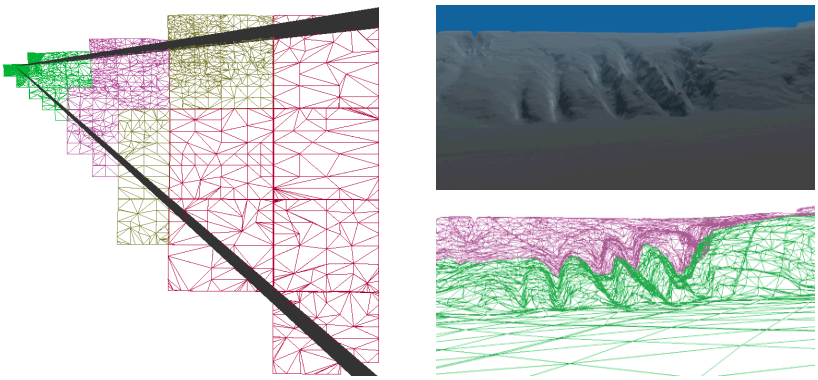


Figure 4.18: In our previous work [TGV01, Tol00] a terrain is visualized in real-time using the QuadLOD structure (which stands from Quadtree and Level of Detail). This structure is actually a cluster of PMs (progressive meshes) organized in a quadtree. Their view-dependent visualization system achieves 40 fps as an average frame rate.

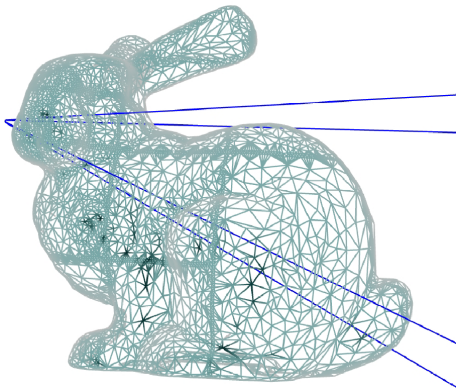


Figure 4.19: A demonstration of HLOD [EMWVB01] generated with partitioning approximate view-dependent simplification.

parent node is exactly the union of the base meshes (M^0) of each son. In this case, global refinement and coarsening operations introduce no popping artifacts.

Erickson et al. [EMWVB01, EM98] proposed a new structure called HLOD (Hierarchical Levels of Detail). The structure is a tree where each node is an object of the virtual scene. Each node keeps the different resolutions of its object, and the coarsest meshes can be grouped to form another node in the tree. They execute a partitioning of spatially large objects (as terrains) and treat each part as a disjoint object. They can also produce view-dependent simplification (see Figure 4.19).

4.3.5 Geometry Simplification Conclusion

Geometry simplification for real-time visualization reduces the frame rendering latency by simplifying the meshes in a scene. This is an important issue since real-time GPUs (Graphics Process Unit) use polygon meshes as input. For this reason, diverse methods have been developed in the last ten years. The following list contains some of the main characteristics that distinguish these methods:

- which decimation method is used in the preprocessing;
- the amount of extra-memory consumption;
- view-dependent or not;

- smooth transition (geomorphing or alpha-blending);
- how much extra-processing for the CPU when rendering;
- implementation complexity.

The *Geometry Simplification for acceleration purposes* area, reviewed in this subsection, has been subject of research in the last 10 years. Actually, the countless papers and methods in this area show the importance of the topic. Some interesting surveys can be found in [Gar99, HGar].

4.4 Replacing Geometry by Images - Impostors

We have previously seen the *Geometry Simplification* methods, which investigate how to reduce the number of polygons in explicit geometry representation to increase displaying performance. In this section we explain the *impostors* and other methods that replace geometry by images. In general, the goal of indirectly representing geometry by images is to accelerate visualization. Note that this subject is still in the macroscale level although some techniques in mesoscale level also use images to represent mesostructures (see Chapter 5).

The basic idea is to completely substitute objects by images. The objective is to relieve the geometry stage from computing the transformation of numerous vertices by using an image with a kind of object "photo" projected over a very simple proxy, for example, a rectangle. These techniques are commonly called "impostors" because they fake the geometry with an image-based solution. The applications must be careful to replace objects only when it is sure that the speed to produce and render the image is faster than displaying the object itself (otherwise the application will lose time processing instead of accelerating it).

In the following text we list the techniques in a complexity increasing order:

Sprite

The concept of sprites comes from 2D computer graphics: any image that moves around on the screen can be considered a sprite (for example, the mouse cursor). There is no need to have a rectangular shape since the use of transparent pixels. However, the sprite idea must be extended so that it can have a practical use in 3D scenes.

A natural requirement for the 3D sprites is the concept of depth. Only with depth information, it is possible to correctly sort the sprites with the other objects in scene [LS97]. In a more elaborate case, the depth is a per-pixel information, which can be used to produce correct intersections and for computing parallax effects (see, for example, *Relief Texture Mapping* [OBM00] later in this section).

Billboard

Billboard follows the same idea presented in sprites (an image with transparency to substitute the real object). However, in this case, the image is texture mapped over a 3D planar polygon in the scene (typically a rectangle). During rendering, a transformation is applied to keep the billboard facing the viewer (this action is known as billboarding [MH99a]). The billboard can be classified by their orientation policy as follows:

- *Axial billboard* keeps one of its axes aligned with the world coordinates. Let us take a tree as an example, as in Figure 4.20 (maybe the most typical billboard example [MD98]). A tree is an object with a roughly cylindrical symmetry, so the billboard should keep the top oriented in world top direction, while it rotates the frame to be as most as possible faced to the viewer. If the observer is moving over the terrain, any time he looks to the billboard direction, he will see the tree faced to him. However, if a flying view is allowed, then the billboard will not be able to correctly simulate the tree.
- *Screen aligned billboard* always keeps the face turned to the viewer. Useful examples are *lens flares*, clouds and *text labels*. Some of them need to keep the up vector aligned with the screen up vector. This is the case of text labels, but for isotropic forms (as the flares) this extra care is meaningless.
- *Full-screen billboard* is useful, for example, to simulate an environment behind everything in a scene. A common application is the sky simulation in outdoor virtual scenario.

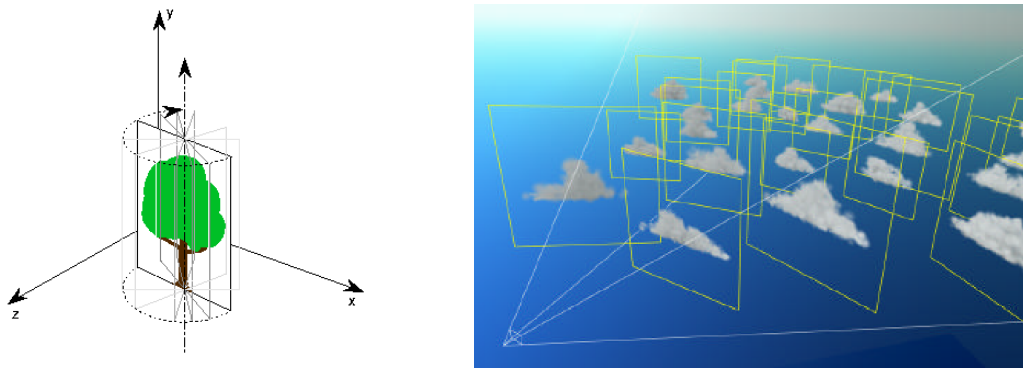


Figure 4.20: *Left:* Axial billboard. The frame containing the tree image turns according to the viewer position. Image from Siggraph Course Notes 1998: *Advanced Graphics Programming Techniques Using OpenGL* [MD98]. *Right:* Screen-aligned billboard. Image from Harris work about *Real-Time Cloud Rendering* [HL01]; in his technique an additional effect of scattering is added to the billboards.

Textured Clusters

The word *impostor* was probably first introduced by Maciel and Shirley [MS95], who used the word not only for image-based geometry approximation but also for LOD simplifications. However, the main innovation in their paper, as announced in the title, was the *Textured Clusters*, which are a typical image-based technique, and, maybe for this reason, "impostor" is often used to name all the techniques in this subject.

The *Textured Clusters* technique idea has a preprocessing that consists in projecting a cluster of objects to the walls of their bounding box. Then, each face, containing a textured cluster, can be used as an impostor in rendering time when the camera is far from the object (for close views the original model is used). This method has a lot of limitations since no depth information is attached to the textures in the box faces. For example, there is no parallax effect when the viewer moves to the sides.

Nailboard

Nailboard, introduced by Schaufler [Sch97], differs from the common billboard by the additional information of per-texel depth. The texture containing the image mimicking the complex object is described by the usual color components RGB and an extra α component for depth. In rasterization time, the depth computation must be done by pixel (rather than standard interpolation from the vertices). However, when Schaufler proposed the nailboards, there was no hardware capable to do per-pixel depth computation. For this reason, his prototype implementation was in software.

With the new programmable GPU, per-pixel depth computation becomes possible. The first implementation of nailboard in these GPUs was called *Depth Sprite*, promoted by Nvidia and explained in their CG tutorial [FK03], for rendering spheres. A texture-based sphere can be rendered with fidelity and correct intersections with other sprites or geometries. The depth texture contains a regular grid with the depths of a half-sphere (see Figure 4.21). This approach was extended also for cylinders in molecule visualization [BDST04]. In both work, a rectangle is used as the frame support for the nailboard. However, it is not possible to directly extend this texture-based approach for more complex objects as a torus (not even for other quadric objects, as ellipsoids or cones). Another limitation is the texture resolution, which is restricted to discretizations in u and v directions and in the depth (usually discretized in 255 values).

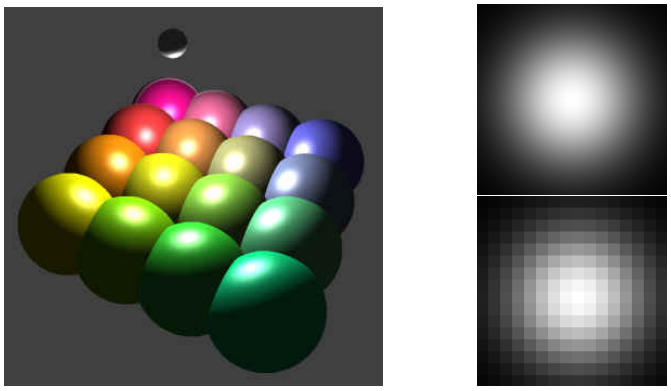


Figure 4.21: Sphere depth sprite. It is a kind of nailboard implemented in hardware [FK03]. Each sphere is rendered using a rectangle plus a depth texture. *Right top:* Example of sphere depth texture. *Right bottom:* Another example but in lower resolution. Actually, the quality of the depth sprite results depends on the depth-texture resolution.

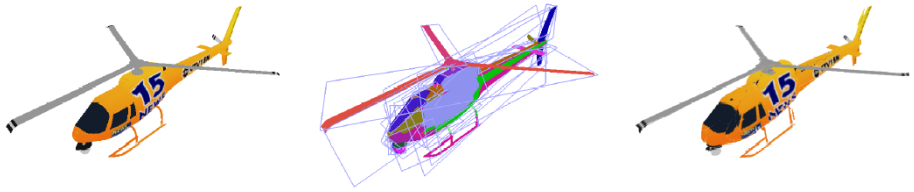


Figure 4.22: The billboard clouds [DDSD03] technique for model simplification. The original model, the set of billboards and the final visualization with 32 billboards.

Similarities to our approach:
We also implement spheres and cylinders with a simple rectangle as a proxy (actually, for spheres just a GL_POINT with scalable side is enough). But in our case the primitives are ray-casted, breaking through the texture limitations. Our primitives have practically no limitation in resolution (the limitation is the floating point precision achieved after several hundred zooming) and there is no texture access in the pixel shader. We also implemented other quadrics primitives (e.g., ellipsoids and cones) and other more complex implicit surfaces: cubics and tori.

Billboard Clouds

Billboard clouds [DDSD03] is a technique for extreme model simplification. The idea is substitute 3D models by a set of planes (billboards) with texture and transparency maps, see Figure 4.22. Each billboard has the image projection of a set of valid faces from the original model (a face is valid if its vertices are in a maximum Euclidean distance ε to the billboard plane). To define the set of billboards, they start by doing a uniform discretization of the plane space into bins $\beta_{\theta_i\phi_j\rho_k}$ (θ and ϕ are spherical coordinates, and ρ is the distance to the origin). Each bin has a computed density, obtained by summing the coverage of valid faces projection, and discounting the penalties (faces that are not valid because their distance, although they are not so far, for example, their distance is between ε and 2ε). Finally, they use a greedy optimization algorithm, based on bins density, to iteratively select planes that can collapse the maximum number of faces. The advantage of this approach, when compared to classical billboard, is the possibility of seeing the object representation from any viewing angle.

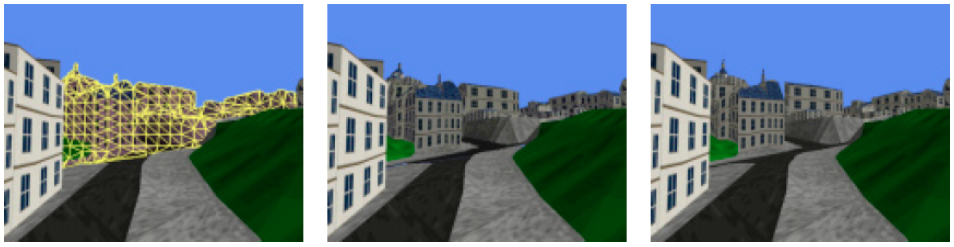


Figure 4.23: Results of Sillion et al. work [SDB97]. *Left:* impostor mesh; *center:* the composed image at about 15 fps; *Right:* the original image at 2 fps.

Relief Texture Mapping

The Relief Texture Mapping technique of Oliveira et al. work [OBM00] is a very interesting warping-based solution for the depth textures inspired by the McMillan and Bishop previous work [MB95]. Before applying the texture they do a two-phase warping (fast 1-D image operations), which depends on the viewing-angle and on each texel depth. As a result, the objects have perfect parallax and correctly respect the silhouettes. However, their work does not deal with self-shadowing and shading. Unfortunately, their warping technique is impossible to be executed in one unique pass in current programmable graphics card, and still needs to be done in software. The reason is that their warping operation is done by texel, depending on the camera position, before projecting the object to the screen.

Relief Texture Mapping could also be classified as a mesostructure technique since it can also be used to apply details over a macro-geometry (for example, simulating details on a bricked wall). But we prefer to classify it as impostor because their results target entire complex objects as buildings or statues.

Texture Depth Meshes

The idea of *Texture Depth Meshes* (TDM) is based on the generation of a 3D mesh from a texture that contains per-pixel depth information. The simplification pipeline follows the sequence: meshes → depth texture → mesh.

To reduce the parallax problem, Sillion et al. [SDB97] introduce the idea of impostors augmented with *three-dimensional* information. Their work for *outdoor* scene visualization has two types of rendering objects: local neighborhood, rendered with the original geometry and conventional textures; and the distant landscape, represented by impostors. The generation of impostors is either off-line or on demand. The impostor creation algorithm is:

(i) from the distant scenery, create color and depth images;

(ii) extract the external contour of the images (for the mesh boundary);

(iii) execute a constrained triangulation of the impostor considering depth disparity (note that the depth image is a kind of height map which is the input for the triangulation).

The 3D mesh obtained in the above algorithm can then be rendered with the color image as the texture, substituting the original model. With the depth information, the same mesh can be used for some consecutive frames while the viewer respect a space-temporal coherence, without losing too much information. With this technique they could improve their implementation from 2 fps to about 15 fps [SDB97]. See Figure 4.23.

In the MMR system [ACW+99] (Massive Model Rendering) another concept of TDM was implemented for indoor applications. Initially, the space is partitioned into *virtual*

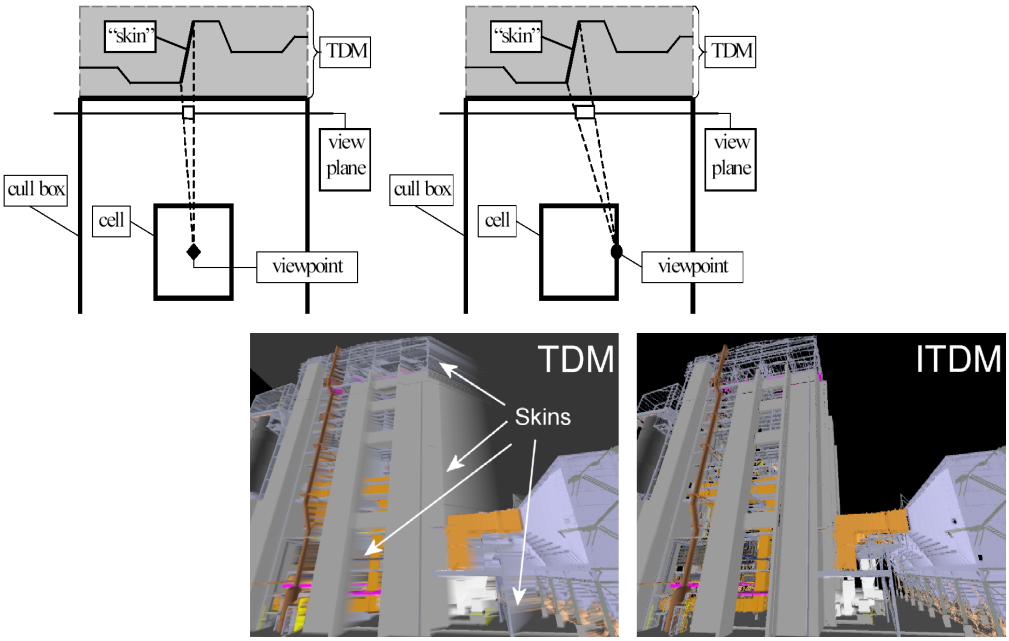


Figure 4.24: On the top images, the explanation of *skin* problem when using TDM. The first image on the bottom shows the skin problem when using the TDM (from MMR system [ACW+99]) and the second one shows ITDM solution without skins [WM03]. Images from Aliaga et al. [ACW+99] and Wilson et al. [WM03].

cells, each one surrounded by a *cull box*. Each face of the cull box contains a TDM with the information outside the box, see Figure 4.24. While *Textured Clusters* [MS95] is an outside-looking-in approach, the cull box faces in MMR are used for inside-looking-out viewing. So, when the user is inside a cell, the geometric primitives outside the cull box are culled away. All the TDM are computed in a pre process, similarly to steps (i) and (iii) in the algorithm explained above in the text. Once more, the depth information produces the parallax and perspective effects. However, there are some regions where the mesh stretches into *skins* to cover missing information (see Figure 4.24). Another issue is the storage space for the TDM, occupying about 10GB for the power plant, whose original data has 600MB.

Incremental Texture Depth Meshes

The main goal of Wilson et al. work [Wil02, WM03] is to avoid the skin problem presented in traditional TDM approach (as in MMR system [ACW+99], see Figure 4.24). *Incremental Texture Depth Meshes* (ITDM) uses the same idea of cells and *cull boxes* presented in MMR, but the main difference is the multiple viewpoint used in preprocessing for computing the ITDM auxiliary structure. They developed a greedy optimization algorithm for incrementally placing sample points in the environment to capture the most about the outside information. As a result, ITDM increases quality and reduces memory auxiliary data (1.4GB for the power plant) but it leads to a higher polygon counts if compared to TDM. The total memory space is reduced because, thanks to quality, ITDM application uses fewer cells (with TDM there is a more significant number of cells to avoid the skin problem). The last image in Figure 4.24 shows an ITDM result.

Geometry Images

In *geometry images* (GIM) [GGH02], the models geometry is also represented by images. But Gu et al. propose GIM method for remeshing a 3D model, not for displaying

it. It converts a surface into a topological disk by using a network of cutting paths and parameterizing the resulting chart onto a square. Sander et al. [SWG+03] extend the method to *multi-chart geometry images*, which splits a model to multi charts and parameterizes every chart.

GIM cannot be considered an acceleration method for displaying. To render the geometry images in a usual way, it is necessary to reconstruct many triangles from the images, and this will bring a heavy burden to the vertex shader process, what should be avoided. The standard GPU techniques to render a GIM use exhaustively the vertex pipeline (through vertex arrays or vertex textures). To reduce the impact over the vertex stage a LOD strategy is mandatory. However, when using multiple charts [SWG+03], the LOD rendering is a difficult task to implement. It is especially hard when neighbors patches have different LOD. This configuration requires special dynamic remeshing schemes to avoid cracks.

Similarities to our work:
The multi-chart geometry images representation looks similar to our *Geometry Textures* method explained in Chapter 7. Although both have resembling kind of representation, they differ a lot in how objects are rendered. In our work, the geometry is no longer represented by triangles, transferring the main rendering effort to the pixel pipeline and alleviating the vertex pipeline, resulting in efficiency gains.

In contrast to Geometry Images, in our approach the image representing the geometry is passed to the GPU as a texture (the *geometry texture*), which will be used by the pixel shader. There is no need to reconstruct any triangle and almost all the effort is done by pixel. Note that our pixel shader renders the geometry with the correct z-buffer output information. It means that the geometry textures are compatible with the standard GPU primitives (and also compatible between them selves).

The construction of geometry texture patches from a complex model has no restriction for the z direction (the height direction) and the patches are well fitted around the surface contour. In rendering time the geometry textures use a *ray-casting* algorithm implemented in a pixel-shader. As a result the geometric detail is reconstructed with the correct shading, self-occlusion and silhouette.

4.5 Replacing Geometry by Volume

In *volume-based geometry representation*, complex polygonal models are substituted by a volume representation. The volume-rendering algorithm visualizes the isosurface by reconstructing the original object surface.

Volume data is commonly represented by a 3D regular grid of voxels (*volume element*), which are the smallest volumetric information, normally containing one scalar. The values inside the volume can be associated to different colors and transparency levels. However, we are interested in the opaque isosurface visualization, without considering semi-transparent voxels.

For visualizing an isosurface inside a volume data there are typically three approaches:

- explicit extraction of the surface converted to a triangle, i.e. using the Marching Cube algorithm [LC87];
- ray casting inside the volume until an intersection with the isosurface is found for each pixel [Lev90];
- rasterizing view-aligned slices with hardware accelerated trilinear interpolation [WE98], only possible after the support of 3D textures [CCF94] in the graphics cards.

Another issue in the techniques described below in this section is the conversion from surface to volume. In general this is possible by computing distance-maps to get the volumetric representation. After choosing the grid resolution, each voxel receives a discrete value of the distance-to-closest-surface function (for example, Euclidean distance). The zero-set of this function encodes the original surface. A reference on how to construct distance maps can be found in [HYFK98].

Decoupling Polygon Rendering from Geometry

In [WSE99], Westermann et al. introduce a complex algorithm to transform surfaces in volumetric representations for rendering purposes. After an extensive preprocessing, they use hardware rasterization of 3D textures using slices that are perpendicular to the viewing direction. The preprocessing steps are:

1. Creation of a set of axis-aligned bounding boxes. The goal of this step is to reduce the empty space inside each 3D texture. They use the OBBtree [GLM96] algorithm, discarding empty boxes and not considering the hierarchy.
2. Distance volumes determination. Each box contains its own volume information that implicitly describes the surface (the zero-set isosurface). The process iteractively increases the volume resolution until a maximum error is achieved, based on one-sided Hausdorff distance. The distance is computed between the original mesh and the isosurface mesh extracted from the volume with the Marching Cubes algorithm [LC87].
3. Inner and outer meshes extraction. These meshes are important in rendering-time to restrict the slicing only close to the surface. They extract the meshes from the distance volume using small positive and negative isovalues and applying the Marching Cubes.
4. Tessellation of the boundary region between the approximating meshes that were resulted from the previous step. In preprocess this is accomplished by approximating the region by 3D primitives, e.g. cubes or tetrahedra. But this step can also be done in rendering time by tessellating the contours that result from clipping both meshes with the slicing planes.

Step number 2 has two important restrictions: the resolution between adjacent distance volumes must differ by a factor of two; and the boxes must have an overlap of at least one voxel.

They obtained interesting results, overcoming some hardware restrictions such as the lack of per-pixel shading (something that they propose as an OpenGL extension, not present at that time, and that can be currently found in graphics cards).

Far Voxels

Gobbetti and Marton work [GM05] for interactive rendering is a very complete system that implements several different acceleration techniques, including out-of-core data management, LOD and visibility culling. The culling is a mixing of potentially visible set (PVS) and occlusion culling. They can support massive models of different classes (isosurfaces, CAD or scanned models). Their weakness is a very expensive preprocessing, for example, one of the tested models consumed 7 hours of preprocessing in a cluster of 16 bi-processor PCs.

The *Far-Voxels* approach is based on the idea of scene partitioning, hierarchically grouped by volumetric clusters. The partition is done with an axis-aligned BSP tree: leaf nodes keep the original triangles data and inner nodes have a volumetric grid representation. In rendering time, the voxels composing each grid are splatted in the final image. The hierarchy level is chosen by respecting a maximal screen area for each voxel (i.e. about 1 pixel).

Voxels Offline Computation. For each inner node, there is a minimum distance d_{min} from which the voxels projection have the correct screen size. So, the preprocessing ray casts the model at distance d_{min} from a large number of different views, collecting visibility and shading information. Empty voxels or invisible voxels are excluded from the volume information. Note that the exclusion by visibility is a kind of PVS algorithm (see Section 3.3.3) that results in a significant elimination of primitives in rendering time, over 40% in their tests. Finally, the remaining voxels are classified in three different types according to the most appropriate shading (depending on the collected ray-casting information). The types are listed below in increasing complex order:

- Flat shader *a*: it is parameterized by a normal, found by averaging all sampled normals, and front/back colors also found by averaging.
- Flat shader *b*: the same as above, but the normal is obtained by computing the principal component analysis.
- Smooth shader: 6 reflectance and 6 normal control points (each one for each main viewing direction).

Their LOD based on voxels is not the only resource used for accelerating the rendering. Besides LOD and PVS, the other techniques used for interactive rendering are:

- frustum culling;
- asynchronous data fetching for out-of-core information request;
- hardware occlusion culling, possible due to a front-to-back rendering order;
- spatial/memory coherence; the primitives close in space are also close in disk, increasing cache performance in several levels.

Real-Time Ray-Casting of Discrete Isosurfaces

Hadwiger et al. [HSS+05] developed a sophisticated GPU implementation for isosurface ray casting. Using multiple passes, they are able to render high-quality images in interactive frame rate. The volume is subdivided into two regular grid levels: a finer one, more adapted to the surface and, consequently, with less empty space; and a coarse level

to manage graphics memory limitations. The input volume is also a distance map, as in Westermann et al. work [WSE99]. Ray-casting is solved after three rendering steps and the surface normal is obtained by computing the gradient $\mathbf{g} = \bigtriangledown f$ in the intersection point.

In this work, some special shadings were developed for *curvature mapping*, *ridges and valleys stressing* and *curvature flow visualization*. To compute principal curvature magnitudes (k_1, k_2) it is necessary to start by computing the Hessian $\mathbf{H} = \bigtriangledown \mathbf{g}$, which is done in 6 rendering passes. At the end, their visualization is done after 14 rendering passes: 3 for ray-casting plus 3 for gradient plus 6 for Hessian plus 1 for curvature plus 1 for final shading. Whereas, their performance is between 10 and 30 fps.

Similarities to our approach:
As in our *Geometry Textures*, see Chapter 7, complex surfaces are subdivided and implicitly represented inside a bounding box. In Hadwiger et al. [HSS+05], the surface is the zero-set of a distance function $f : \mathbb{R}^3 \longrightarrow \mathbb{R}$, while in our approach the surface is the zero-set of a height function $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$.

We can point out some drawbacks in their approach compared to ours:

- 3D volume data requires more space to store as also to load in graphics memory. The height-map can be mapped as a 2D gray-scale image. That is why they have to pay special attention to memory limitations.
- In their approach multiple-passes are done to execute the rendering. In our visualization algorithm just one pass is enough.
- Hadwiger et al. do not mention the possibility of rendering their surfaces with other objects in the scene. Our approach outputs a correct z-value, keeping compatibility with any other object in scene.

In spite of all that, it is important to stress the high-quality images produced in their results.

Chapter 5

Mesoscale

In previous chapters we have explored the scene-scale level (Chapter 3), which includes *Spatial Scene Subdivision* solutions and *Visibility Culling* algorithms, and the macroscale level (Chapter 4), which concerns the objects macro-geometry. In the mesoscale level we are interested in the objects appearance.

The appearance of an object is a result of the microstructure in its surface. A complete model that takes into account the microstructure and all the physical events occurring in this scale would be impractical and far from interactive rates. For this reason a lot of effort was dedicated in representing the appearance in a mesoscale level, between the micro and macro scales. The surface structure we need to represent in this level is named *mesostructure*. The visual results depend on the mesostructure representation but it also depend on the lighting model. The light models used in macroscale (see Section 4.2.3) can be applied to the mesoscale, but sometimes a more sophisticated model is necessary.

Texture

When touching an object someone can feel its texture. This tactile information is a result of the *surface texture*. The texture also gives visual information in such a way that we can imagine the sensation by just watching an object. The texture is a result of the object mesostructure and it can be simulated in several ways in visualization systems. Its simplest representation is the *color map* (Section 5.1), which is a simple 2D image applied over a virtual object. More complex mesostructure representations are the *normal map* (Section 5.2), the *height map* (Section 5.3) and the *volume* (Section 5.4). Finally, in Section 5.5, we present methods that store the whole shading function rather than representing the mesostructure.

5.1 Color Map

Color mapping technique, also known as *2D texture mapping* [Cat74, BN76] is the simplest mesostructure simulation method. Instead of representing object details by fine geometric information, an image covering the object surface (with a simple geometry) partially adds visual information.

In the macroscale shading model, which we have described in Subsection 4.2.3, it was not possible to specify appearance details inside each polygon. For example, in the macroscale geometry, if the material diffuse coefficient K_d varies along the surface it is discretized by assigning individual values to each polygon or to each vertex. As a result, the coefficient is either constantly or linearly interpolated in the interior of the polygons. To overcome this limitation, an image is mapped over the surface.

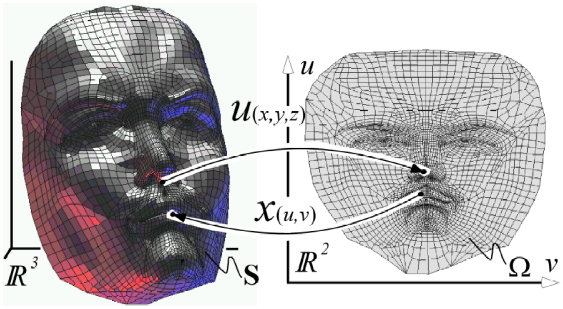


Figure 5.1: The projection function $\mathcal{U}(P)$ defines the one-to-one correspondence between the 3D surface $\mathcal{S}$ and the 2D set Ω.

The image, or the *texture*, is previously create (by synthesis or digitalization) determining the 2D regular table of *texels*[6], each of them containing a color representing the local diffuse coefficient. In rendering time, a way to obtain the correspondent 2D coordinates for any 3D surface point is necessary.

For a given surface $S \in \mathbb{R}^3$, a one-to-one correspondence with a 2D set Ω is assigned. This correspondence is defined by the projection function:

$$\mathcal{U}(P) = \begin{bmatrix} \mathcal{U}_u(x,y,z) \\ \mathcal{U}_v(x,y,z) \end{bmatrix}, \qquad f : \mathbb{R}^3 \longrightarrow \mathbb{R}^2$$

In some classes of surfaces, the projection function is an implicit part of surface formation. That is the case of parametric surfaces (see Section 4.1.2), which have a natural set of (u,v) values as part of their definition. Actually, for these classes of surfaces, the one-to-one correspondence is done in the opposite way, by a parameterization function $\mathcal{X}$:

$$\mathcal{X}(u,v) = \mathcal{U}^{-1}(u,v) = \begin{bmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{bmatrix}$$

For surfaces described by triangular meshes, a pair (u,v) is stored for each vertex. For a point P inside a triangle T, the correspondent (u,v) coordinates is obtained by an average of the coordinates in the three vertices (P_1, P_2, P_3) of T. The homogeneous barycentric coordinates of P weights the average computation of (u,v) as follows:

$$u(P) = \frac{u(P_1)\,t_1 + u(P_2)\,t_2 + u(P_3)\,t_3}{(\vec{P_1P_2} \times \vec{P_2P_3})},$$

$$v(P) = \frac{v(P_1)\,t_1 + v(P_2)\,t_2 + v(P_3)\,t_3}{(\vec{P_1P_2} \times \vec{P_2P_3})},$$

where

$$\begin{aligned} t_1 &= \vec{P_2P} \times \vec{P_2P_3} = \text{ barycentric coordinate of } P \text{ related to } P_1, \\ t_2 &= \vec{P_1P} \times \vec{P_1P_3} = \text{ barycentric coordinate of } P \text{ related to } P_2, \\ t_3 &= \vec{P_1P} \times \vec{P_1P_2} = \text{ barycentric coordinate of } P \text{ related to } P_3, \end{aligned}$$

and $(\vec{P_1P_2} \times \vec{P_2P_3})$ is the normalization factor so: $\dfrac{t_1 + t_2 + t_3}{\vec{P_1P_2} \times \vec{P_2P_3}} = 1.$

The use of barycentric assures a linear interpolation of the (u,v) coordinates inside each triangle.

[6] *texel* = texture element

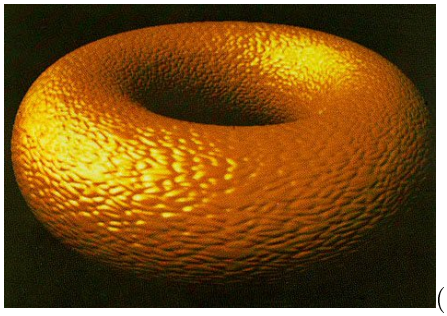
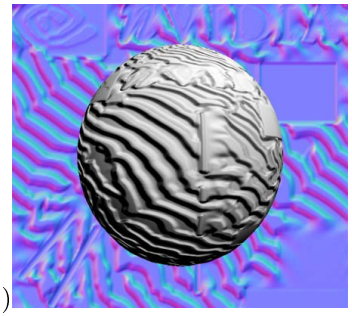

(a)

(b)

Figure 5.2: (a) The torus is rendered with an orange-fruit bump mapping (original image from Blinn's work [Bli78]). (b) The smooth macro-geometry of a sphere is perturbed by the bump mapping to simulate a special mesostructure; the background is the normal map stored in a RGB image.

Although largely used, probably due to its simplicity, texture mapping left a gap of important visual effects such as correct lighting and silhouettes. The next sections are about other methods that improve the mesostructure representation and shading.

5.2 Normal Map

Blinn extended the texture mapping with the bump mapping technique [Bli78], which gives visual effects of bumps and depressions by disturbing surface normals according to a bump map. The *mapping* over the surface is done in the same way as explained above for the textures. However, instead of associating a color to a surface point, the bump mapping associates a normal (or a normal perturbation). When computing the diffuse component (Equation 4.7) the normal vector $\vec{n}$ is obtained from the bump mapping. Bump mapping also affects the specular component (Equation 4.8) since the vector $\vec{r}$ depends on the vector $\vec{n}$. Note that texture mapping can be additionally applied in combination with the bump mapping.

In general, the normal map is defined using the local coordinate system (or *texture coordinate system*). The local coordinate system is formed by the u and v parameterization axes and the local normal respects the macro-geometry surface. These 3 axes form a 3×3 matrix known as the BTN matrix (Binormal, Tangent and Normal). So, typically, when computing the shading model, the other vectors (light vector $\vec{l}$ and viewing vector $\vec{v}$) are transformed to the local system using the BTN matrix.

In Figure 5.2 we can see some results of applying the bump mapping to some simple surfaces. Since the normal is a 3D vector, its components can be discretized and stored in a RGB color table (see background on Figure 5.2(b)).

The use of normal maps to represent the mesostructure brings significant visual impact in visualization results. However there are some other expected effects when we try to reproduce the mesostructure over a surface: the silhouette should contain the same details; mesostructure self-occlusion and self-shadows.

5.3 Height Map

As we have seen in Subsection 4.1.2, the height maps are a compact way to represent surfaces usually applied for terrains. In this section, we describe some mesoscale methods that use height maps to represent the mesostructure over a macro surface. The height map domain is no longer a plan, but it is adjusted to the geometry curvature (as in texture mapping).

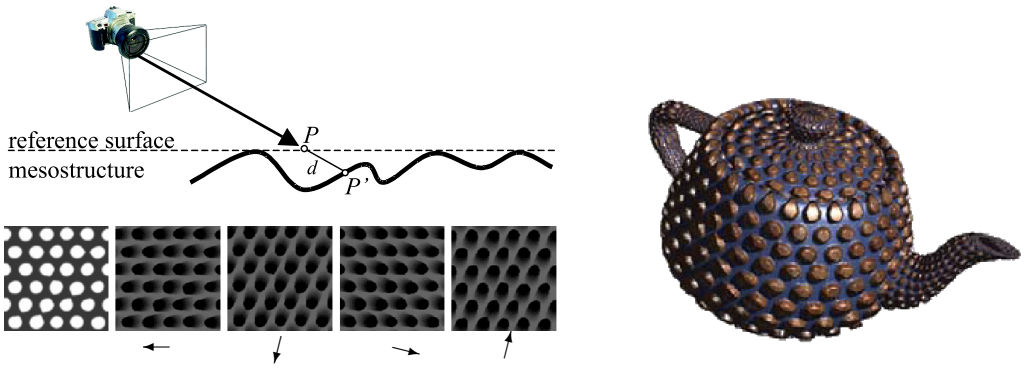


Figure 5.3: View-dependent Displacement Mapping (VDM) [WWT+03]. *Top left:* VDM computation scheme, for each point P on the reference surface, the preprocessing algorithm computes the distance d to the mesostructure intersection point P'. *Bottom left:* the result of preprocessing the height-map (most left image) in different viewing directions (small vectors). *Right:* the result of VDM applied in the teapot mesh.

5.3.1 Displacement Mapping

In displacement mapping technique [Coo84, CCC87] the macro geometry is subdivided into a large number of small polygons whose vertices are displaced in the normal direction according to the associated displacement map. Compared to simple bump mapping, this technique presents two additional visual effects: detailed silhouettes and self-occlusion. However, the excessive number of polygons is a problem for interactive visualization. Two approaches were taken to speed-up the displacement mapping: adaptive remeshing methods [DH00] and displaced-triangles generation in hardware [Kry05, AH05]. The hardware triangle generation is possible for recent graphics cards with the *vertex texture* feature. Nevertheless, the number of generated triangles is still too large, overcharging the vertex stage in the rendering pipeline.

5.3.2 View-dependent Displacement Mapping – VDM

In Wang et al. work [WWT+03], an object with fine-scale geometric features can be represented and rendered by a simple mesh and a view-dependent displacement mapping, correctly treating occlusion, silhouette and self-shadowing.

To achieve these results, they start by implementing a preprocessing step using a height-field as input, which represents the mesostructure. By means of a ray-casting algorithm, for each point P in reference surface, they record the distance d to the height-map intersection point P' (see Figure 5.3), for different viewing angles (32×8 viewing directions on hemisphere space) and difference surface curvatures. So, from a 128×128 height field (16KB), with 32×8 viewing directions and 16 different curvatures, they create the 64MB VDM data. All this data is accessible through a 5D function:

$$d = d_{VDM}(x, y, \theta, \phi, c),$$

where x, y are the texture coordinates, θ, ϕ are the spherical angles of the viewing direction and c is the reference surface curvature along the viewing direction.

During rendering, a simple mesh of the entire object is rendered with a dedicated *fragment program*. The fragment program accesses the VDM structure to execute silhouette detection, self-shadows and shading. The VDM representation is accessed twice: first using the camera-viewing angles and then using the light-viewing angles for shadow determination (the other three variables are the u, v texture coordinates and the surface curvature).

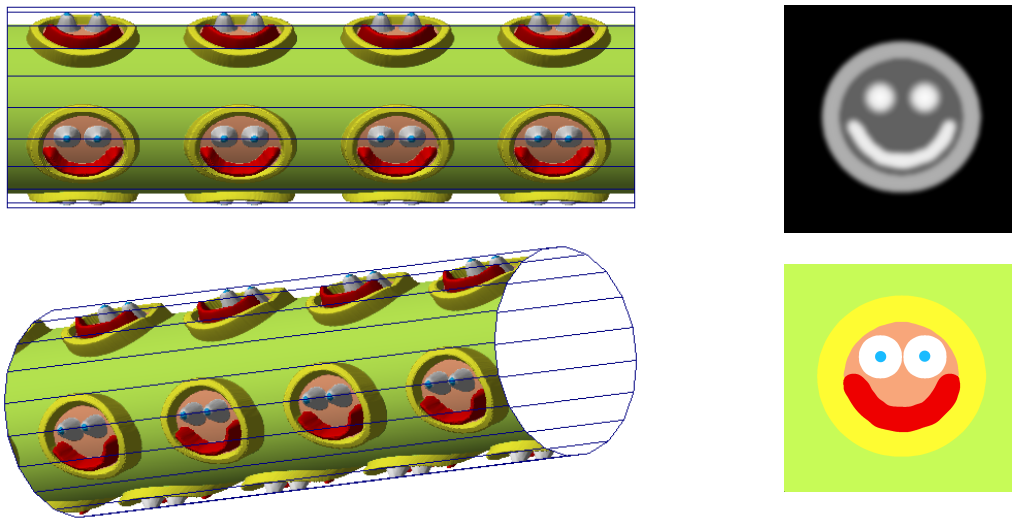


Figure 5.4: These pictures were produced in the beginning of our research project. The height-map mesostructure described in a gray-scale image (*top right*) is applied over the cylinder. The colors are specified by the texture (*bottom right*). The wire-frame is visible here to show the macro-geometry where the mesostructure is applied.

The results are impressive (see right image in Figure 5.3). However, their work has some limitations. First of all, their VDM generation uses orthographic projection, even if the camera is in perspective projection (the positional lights are also treated as directional lights). Another drawback is the imprecise information, since the VDM structure was constructed by using discretized variation of the variables, when in practice they are continuous. Actually, the main issue is memory space. Although they propose a *singular-value decomposition* compression to reduce the 64MB to 4MB (which brings even more imprecision), the VDM space prevents a lot of possible extensions to their work:

(i) the use of higher quality height maps is not viable (for example a 512×512 height map would need 1GB for the uncompressed VDM);

(ii) the use of multiple mesostructures in the same application is limited, for instance their application only shows a repeated mesostructure pattern;

(iii) since they use a preprocessing step to construct VDM, they cannot deal with dynamic changing in their geometry.

5.3.3 Interactive Mesostructure Ray Casting

Thesis Contribution: Mesostructure ray casting
Using our height-map ray casting implemented on GPU, it is possible to reproduce mesostructure over macro-geometry surfaces. See Figure 5.4.

Although our goal is the use of height-map ray casting for displaying whole objects (see *Geometry Textures* in Chapter 7), this technique is completely suitable for mesostructure visualization. The images in Figure 5.4 show the results obtained in the beginning of our research, also reported in an internal document [TLP03] and presented at IMPA's seminar [Tol04].

Some other researchers have done a parallel work exploring the same idea. In Policarpo et al. [POC05a, POC05b] we find an approach that is the most similar approach to ours. The ray casting is done in two steps: a linear one followed by a binary one. At

that time they were limited to no silhouette detail for non-planar geometry. Lately, they overcome the silhouette problem as explained in their technical report [OP05]. When comparing our implementation to their work, there are two improvements, both for efficiency: we compute a balance between the linear and the binary search depending on the viewing angle; and we use a double-resolution representation of the mesostructure, dividing the linear steps in two (the first one with the coarsest approximation and the second one with the original data). For details about our implementation, see Section 7.2.

In both implementations, Policarpo et al. and ours, the memory is no longer a problem as it is in the VDM solution. VDM produces 64MB of information from a 16KB height map, to precompute the ray casting from different points of view. The interactive mesostructure ray-casting accesses directly the original data, without any precomputation.

Hirche et al. work [HEGD04] is also centered in a height-map ray-casting per-pixel implementation. However, they extrude some prisms from the base mesh (their macrogeometry), and the ray casting is done by using the prism walls, quadrupling the number of faces, which is unnecessary in our method.

An interesting point in all these *mesostructure ray casting* methods is the breakthrough of limitations presented in bump mapping (as silhouette and self-occlusion) and those presented in traditional displacement mapping (as self-shadowing), see Table 5.1 on page 100. However there are still some restrictions due to the height-map nature, which prevents *overlapping*. Some useful mesostructure cannot be correctly represented with this limitation, for example *fur*. The next subsection is about methods that use a volume representation for mesostructure, avoiding this problem.

5.4 Volumetric Mesostructure

The use of volumetric mesostructure brings a more flexible way to represent details [KK89, MN98]. The mesostructures found in real world are better approximated by volumes, which have no overlapping restriction [WTL+04]. Another advantage is the possibility to add opacity per-voxel information, which can be used for simulating more complex lighting phenomena as refraction and subsurface scattering [CTW+04, PBFJ05].

Most of the rendering applications use prism extrusion from the base mesh to create a 3D parameterized space. This can lead to self-intersections near concave features, which is solved by some methods [PKZ04, PBFJ05], but not mentioned in others.

Interactive Semi-Transparent Volumetric Texture

In Lensch et al. work [LDS02] volumetric textures are applied over arbitrary macrogeometry mesh. In a previous step they span prisms from each triangle of the mesh, using the vertices normals, resulting in 6 vertices, all of them with 3D texture coordinates. They use back to front slicing technique, drawing the polygons inside each prism formed by the intersection of the orthogonal viewing slices. To produce a fast intersection computation, each *normal edge* (the vertical edges coinciding with the normals) is previously classified in relation to the plane as either above, below or intersected, mounting a table. So, the intersection plane inside the prism is drawn by visiting the quadrilaterals in the same order as the respective edges in the original mesh. The number of intersections in a quadrilateral (0, 1 or 2) can be rapidly discovered using the table of *normal edges* classification. They have created 3 different implementations: in software, in hardware, and a hybrid one, which is the fastest.

Generalized Displacement Maps (GDM)

Wang et al. work [WTL+04] is a generalization of VDM [WWT+03] (explained in Section 5.3.2) that accepts volume mesostructure definition. Their main structure, named GDM, is also a five-dimensional function accessed in real-time, but as opposed to VDM they use prisms, computing the ray exit point and evaluating the intersection in each prism domain. They correct some texture distortion problems on silhouette rendering presented in VDM and they can optionally visualize the objects with global illumination. In *Shell Texture Function* (STF [CTW+04]) Chen et al. compute subsurface scattering, using photon tracing in preprocessing and ray tracing for final visualization, but the rendering becomes no longer interactive (images are generated in about 100 seconds). In their subsequent work, SRTF (*Shell Radiance Texture Functions*) [SCT+05], the visualization achieves interactive frame rates. However it is not possible to reproduce detailed silhouettes since SRTF is rendered without ray tracing.

We can note that there are basically three ways to render the volumetric mesostructures found in previous work:

- slicing planes rendering [MN98, LDS02, PKZ04];
- non-interactive ray tracing [CTW+04, PBFJ05]; and
- solid-distance function application [WWT+03].

Indeed, this last one is a hybrid approach between ray casting and shading functions (described in next subsection), since it breaks the ray in segments and each segment accesses the pre-computed table information, the GDM structure.

From our knowledge, there is no method that uses ray tracing (or ray casting) for volumetric mesostructure interactive visualization. This subject should be a point for **future work** in mesostructure visualization. In our point of view, the ray-casting procedure could be done in the same way as described for height-maps, Subsection 5.3.3. Donnelly [Don05] uses distance function discretized in a 3D texture to accelerate height-map ray casting. Maybe Donnelly's work is the closest one to this open topic: "*volumetric mesostructure interactive ray casting*".

5.5 Representations of the Shading Function

Shading models are a very complex and important subject in visualization domain. Its complexity comes from the fact that these models try to represent and simulate the lighting phenomena happening in microscale level to obtain results in macro and mesoscale levels. In Section 4.2.3 we introduced the Phong's model [Pho75] based on the diffuse/specular components emitted by objects in the viewer direction. When applying this model in macroscale objects the results are not good enough since the details are missing. For this reason we investigate the mesoscale representation and shading of objects to recover the details.

In previous mesoscale sections we revised the mesoscale level, following a complexity order in the mesostructure representation. With texture mapping (Section 5.1) and bump mapping (Section 5.2), it is possible to store the parameters (diffuse coefficient and normals) along the object domain, varying them inside each macro polygon. To also consider the visibility effects in the mesostructure, as self-occlusion and detailed silhouette, more complex representations were proposed, using height maps (Section 5.3) and volumes (Section 5.4). In all these methods, the shading model in use is the diffuse/specular one, which is only an approximation of real lighting effects. For more accurate shading, a *bidirectional reflection distribution function* (BRDF) is more appropriate to categorize how light is reflected from a point on the surface. The two directions considered in this

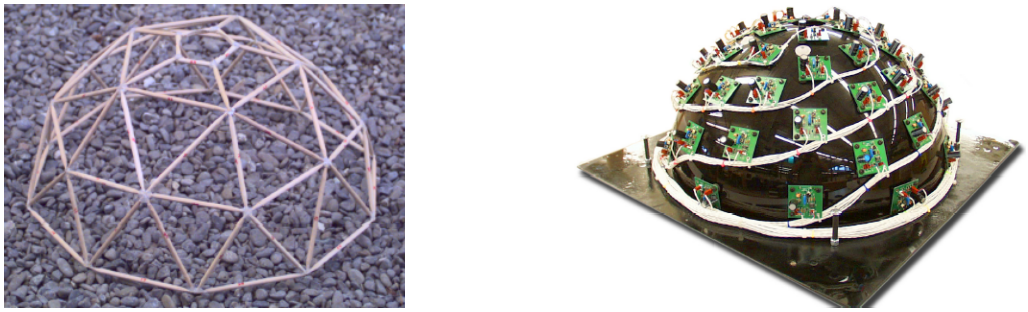


Figure 5.5: Two different devices used for capturing pictures around a hemisphere, images from PTM work [MGW01].

function are the incoming light and the viewing (reflected) direction. Although we explained BRDF in details only in microscale chapter (Chapter 6), this section covers the methods based on its principles. This scale difference is also remarked by Dana et al. [DNGK97] who relate BRDF to subpixel scale (microscale in our classification) and BTF to texture scale (mesoscale in our classification).

Most of the methods described in this current section use a set of images as the input information about the mesostructure shading function. This image collection is the result of capturing real pictures of the desired materials from different points of view, uniformly distributed in a hemisphere, see Figure 5.5. In some methods [LYS01, VT04] this is done synthetically, see Figure 5.6.

Bidirectional Texture Function (BTF)

Dana et al. [DvGNK99] introduced BTF in an experimental work based on the BRDF principles, explained in Section 6.1. They captured a set of images, using different material samples and varying the light and viewing directions (a regular discretization of the hemisphere possible angles). The images are stored in a publicly available database. So, the BTF (bidirectional texture function) is a 6D function, which has the same 4 BRDF variables (see Equation 6.1) plus 2 for position on the image:

$$BTF = f(\theta_{in}, \phi_{in}, \theta_{out}, \phi_{out}, u, v) \tag{5.1}$$

For each material a different BTF is specified. Some methods have extended the BTF approach. Liu et al. [LYS01] address two related problems: the synthesization of a continuous BTF from the discrete set of images to automatically generate a texture for any given light/viewing setting, and the specification of new synthesized BTF's without the captured images. To achieve these results, they use a height field as the basic specification of the material mesostructure. In [TZL+02], Tong et al. succeeded in rendering arbitrary surfaces with BRDF material although in low speed (about 1 frame per second).

Applying BTF on surfaces for real-time rendering is not easy; the lookup value at (u, v) is a hard task since BTF original information consists of a large amount of images. *Polynomial Texture Maps* [MGW01], described below, is a BTF simplification since it removes the viewing direction from the equation, reducing the number of images taken for each surface.

Polynomial Texture Maps (PTM)

In PTM [MGW01], Malzbender et al. also use a set of images as input. However they take the decision to vary only the lighting position, capturing all images from the same point of view. Once the camera is fixed, they avoid problems related to camera calibration.

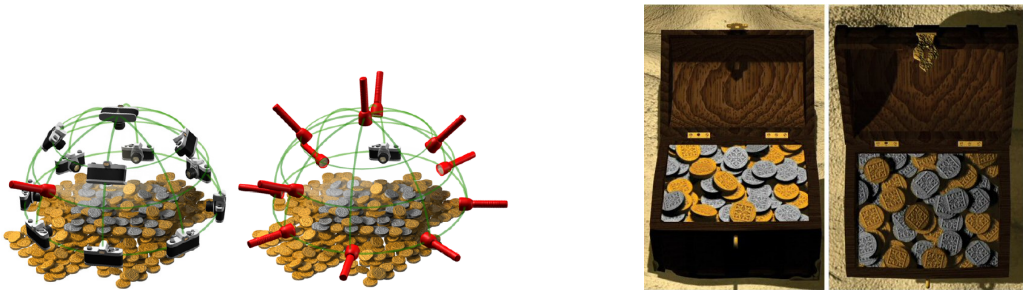


Figure 5.6: TensorTextures technique [VT04]. *Left:* The mesostructure is a composition of sample images captured by the preprocessing with different viewing and lighting directions. *Right:* The result is a 3D relief perception although TensorTexture is mapped onto a planar surface.

Their approach is explicitly a simplification of BTF (Equation 5.1) holding two of these dimensions constant (the output direction):

$$PTM_{r,g,b}(\theta_{in}, \phi_{in}, u, v) \tag{5.2}$$

Instead of recording and accessing all the pictures, which would be very costly, they do two convenient approximations to store per-texel information:

- Split chromaticity and luminance. Once the chromaticity for each (u, v) coordinate in images is fairly constant, they store an unscaled color per texel, which is modulated by the luminance depending on the input angle (θ_{in}, ϕ_{in}).
- They also explore the smooth dependence of luminance on light direction. So, they approximate per-texel luminance as a biquadratic with five coefficients (obtained by solving an equation system):

$$L(u, v; l_u, l_v) = a_0(u, v)l_u^2 + a_1(u, v)l_v^2 + a_2(u, v)l_u l_v + a_3(u, v)l_u + a_4(u, v)l_v + a_5(u, v)$$

As a result, PTM produces high quality photorealistic renderings of textured surfaces, although only five extra per-texel coefficients are needed. Especially with some proposed extensions, it reproduces several lighting effects: Fresnel, anisotropic, off-specular and depth of focus. Lately a GPU implementation was proposed in [KBR04], including the RGB PTM, which splits the polynomial function to each color channel, and the BRDF PTM, used for spatially homogenous materials with anysotropic characteristics. In BRDF PTM the light direction (l_u, l_v) is still used for independent variables while the view direction (v_u, v_v) is used as the 2D texture coordinate.

TensorTexture

The TensorTexture [VT04] has a similar approach to BTF [DvGNK99, LYS01]. The preprocessing begins by taking pictures of the synthetic mesostructure, varying the viewing and lightning angles (see Figure 5.6). The first different point is the use of an ensemble of textured surfaces as mesostructure. A second point is the way TensorTextures[7] are stored. Since the volume of information generated by the preprocessing is large (about 100MB in their examples), as in VDM, it is very important to compress the data. TensorTexture applies a dimensionality reduction after a multilinear analysis of the image data tensors. The results are good since there is just very small perceptual degradation even after 85% of compression.

[7] *tensors* are multilinear mappings over a set of vector spaces

Compared to VDM, TensorTexture has some advantages: no restriction about overlapping in their mesostructure (which is a restriction on VDM since it uses height maps); and inter-reflection effect (computed offline). However, this approach had proves to be too slow for interactive rendering. The CPU implementation, shown in Vasilescu et al. paper, has an average time of 1.6 seconds per image.

5.6 Mesoscale Conclusion

Technique [Reference]	Visual Effects					Characteristics				
	Detailed Lighting	Detailed Silhouette	Self-occlusion	Self-shadow	Interreflection	Extra Memory	Extra Polygons Factor	Input Difficulty	Preprocessing	Rendering Speed
Color										
Texture Mapping [Cat74]						3	1	L		↑↑
Normal										
Bump Mapping [Bli78]	X					3	1	M		↑
Height Map										
Displacement Mapping [Coo84]	X	X	X			1	10^2	M		—
VDM [WWT+03]	X	X	X	X		2^{12}	1	M	X	↑
Ray-casting [TLP03, POC05a]	X	X	X	X		1	1	M		—
Prism Ray-casting [HEGD04]	X	X	X			1	4	M		—
Volumetric Mesostructure										
Semi-Transparent [LDS02]	X	X	X			2^7	3	H		—
GDM [WTL+04]	X	X	X	X		2^{14}	4	H	X	—
Shell Texture Function [CTW+04]	X	X	X	X	X	10	NA	VH	X	↓↓
Shading Function										
BTF [DvGNK99]	X		X	X	X	2^9	1	VH	X	↓
PTM [MGW01]	X			X	X	5	1	H	X	↑
TensorTexture [VT04]	X		X	X	X	2^{11}	1	H	X	↓↓

Table 5.1: Comparison between different interactive mesostructure visualization.

We summarize the visual effects and characteristics of some of the main mesostructure visualization techniques in Table 5.1. Some of the presented numbers are approximations, but they are meaningful to compare different techniques. In the following text we pick up some relevant points in this table for deeper discussion:

Detailed Lighting This visual effect is observed in all techniques except for simple texture applying. When the unique applied technique is texture mapping, changing light direction has no direct effect over the mesostructure.

Detailed Silhouette This effect is predominantly seen in techniques using either height-map or volumetric mesostructure representations. On the other hand, none of the *Shading Function* methods are capable to reproduce silhouettes.

Interreflection The methods marked in this column on table are those that consider interreflection between microfacets in the mesostructure.

Extra Memory In this characteristic we have evaluated how much per-texel information is needed to represent the mesostructure (computing the uncompressed data). So, for example, RGB texture mapping uses 3 channels of color per texel and for simple height maps 1 extra channel is enough. For volumetric representations, we count how

many voxels there are in direction w. So, a mesostructure volume with $128 \times 128 \times 32$ in its (u, v, w) directions, represent a factor of 32 in *per-texel* information. Shell Texture Function use very thin volumes, with just 10 voxels in w direction. It is noticeable the huge numbers presented in VDM and GDM techniques. But in practice, they reduce memory consumption by doing a SVD (Single Value Decomposition) based compression, which can downsize the mesostructure representation to few megabytes. In VDM, they use 32×8 viewing directions and 16 curvatures to sample the mesostructure, producing 2^{12} data per texel. In GDM, there are 32×16 viewing directions and up to 32 voxels in w direction, resulting in 2^{14} extra information. The original BTF data collected by Dana et al. [DvGNK99] consist of 205 images for each material with RGB information, so we approximate the extra memory in table by 2^9 ($205 \times 3 = 615 \approx 2^9$). In the same way, TensorTexture use 37 viewing and 21 lighting directions, which we approximate by 2^{11} ($37 \times 21 \times 3 \approx 2^{11}$). But, after their multilinear analysis, the data can be compressed in 90% without perceptual loss.

Extra Polygons Factor For Displacement Mapping, we consider that the number of texels inside each polygon is about one hundred. In Shell Texture Function there are no polygon mesh anymore, but only volumetric representation used in their ray-tracing algorithm. Techniques based on prisms have 3 or 4 extra-polygons factor, never 5 because the prism base is not used. In [LDS02] the factor is 3 since only the prism quadrilaterals are useful, but in practice the number of rasterized polygons depends on the number of slices in their slice-based volume rendering.

Input Difficulty In this characteristic we classify the techniques as **L**ow, **M**oderate, **H**igh or **VH** for Very High, depending on the difficulty to create mesostructure input.

Preprocessing VDM and GDM techniques preprocessing are basically about data compression, except that the last one also does *global illumination* pre-calculations.

Rendering Speed Shell Texture Function (STF) and TensorTexture have bad remarks here and they should not even be considered interactive (each image takes about one minute to be rendered). Note that the STF extension called SRTF [SCT+05] have better performance but no detailed silhouette. No special technique is as fast as texture mapping, but bump mapping, VDM and PTM have significant performance since their per-pixel shading are simple.

Another global observation we can extract from Table 5.1 is the absence of an interactive method that contains both interreflection and detailed silhouette, showing that there is still a lack in mesostructure real-time rendering.

In this chapter we have seen the methods for mesoscale visualization. In most of the cases, the results are obtained after applying a simulation technique that reproduces what might happen in a microscale level. In next chapter (the last one in our visualization survey), we discuss the physical phenomena occuring in such scale level.

Chapter 6

Microscale

In microscale subject we discuss about the small details in the visual aspect. This scale is an important topic, closing our visualization survey.

Lighting Models

We have seen in Section 4.2.3 simple lighting models appropriate for macro-geometry rendering, but they do not take into account other visual effects, such as interreflection, light absorption and scattering.

To better understand what happens with reflected light from surfaces, we discuss, in Section 6.1, the *Bidirectional Reflection Distribution Function* (BRDF), a shading model more generic then the simple diffuse/specular one. Later, we describe an extension for BRDF, the *Bidirectional Subsurface Scattering Reflection Distribution Function* (BSSRDF) in Section 6.2. Then, we explain two visualization techniques based on microscale light phenomena: *Photon Mapping* (Section 6.3, which is commonly used in offline applications; and *Precomputed Radiance Transfer Function* (Section 6.4) suitable for real-time rendering.

6.1 Bidirectional Reflection Distribution Function

In surface visualization, the reflected light coming from the objects in the viewing direction determines the colors in the final image. The way in which light is reflected from a point on a surface varies from object to object depending on its material properties. A particular material reflection can be defined by a bidirectional reflection distribution

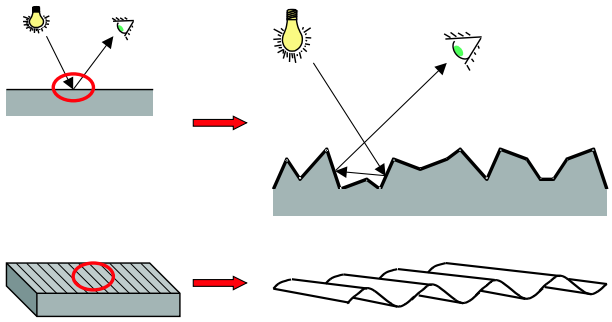


Figure 6.1: *Top:* Zoom on the surface microstructure: the microfacets interfere in the optical system. *Bottom:* Zoom on the mesostructure of an anisotropic surface.

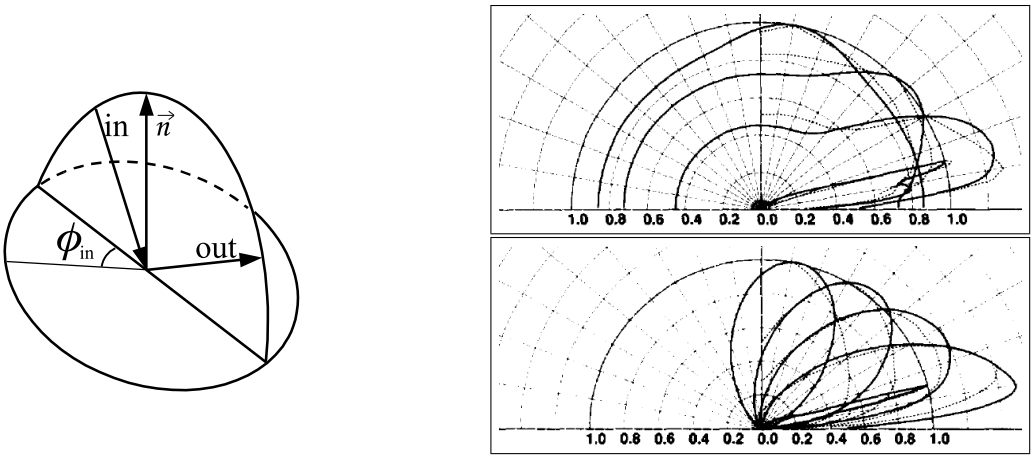


Figure 6.2: BRDF variables are the incident angle (θ_{in}, ϕ_{in}) and the reflected angle $(\theta_{out}, \phi_{out})$. *Left:* The plane containing *in*, *out* and normal is in evidence. *Right:* BRDF cross-section for different materials with a light wavelength $\lambda = 0.5\mu m$ and varying incident angle θ_{in} from He et al. work [HTSG91b]. *Top right:* magnesium oxide with incident angles of $\theta_{in} = 10°, 45°, 60°, 75°$; *Bottom right:* aluminum with incident angles of $\theta_{in} = 10°, 30°, 45°, 60°, 75°$.

function (BRDF) [NRH+77]. This function depends on the incident light direction and on the reflected direction:

$$BRDF = f(\theta_{in}, \phi_{in}, \theta_{out}, \phi_{out}) \tag{6.1}$$

So, based on the BRDF model, to compute the outgoing light L_o in a given direction $\vec{\omega}_o$ we need to integrate the incoming light in all directions:

$$L_o(\vec{\omega}_o) = \int_\Omega BRDF(\vec{\omega}_i, \vec{\omega}_o) L_i(\vec{\omega}_i) cos(\theta_i) d\vec{\omega}_i \tag{6.2}$$

where Ω represents the hemisphere space since the light beams coming from the back are not considered. $\vec{\omega}$ describes the spherical coordinate angles (θ, ϕ) of a given direction vector.

Different surface materials have different associated BRDF; actually, this function can also vary depending on the light source wavelength [HTSG91a]. For multiple incoming lights, the total reflected light would be obtained by integrating the separate BRDF [Wat00]. Note that anisotropic surfaces are naturally considered by the BRDF function (See Figure 6.2). For isotropic surfaces, BRDF reduces to three variables since rotations about the surface normal become meaningless [MPBM03].

The diversity of BRDF is a result of the surfaces nature, see Figure 6.3. Except for perfect ones, such as glass or still water, surfaces have a microgeometry that plays a very important role in the optical system (see Figure 6.1). Once it would be very hard to simulate the optical events in the microgeometry all over an object surface, some research has been done in reproducing the shading result in a mesoscale level (see Section 5.5).

6.2 BSSRDF

BRDF is a general lighting model applicable to different materials, however, it cannot simulate some light effects such as subsurface scattering. For this reason, Jensen et al. [JMLH01] developed the BSSRDF (SS for *subsurface scattering*). In BRDF model, the light enters and leaves a surface at the same position, which is not true for some materials

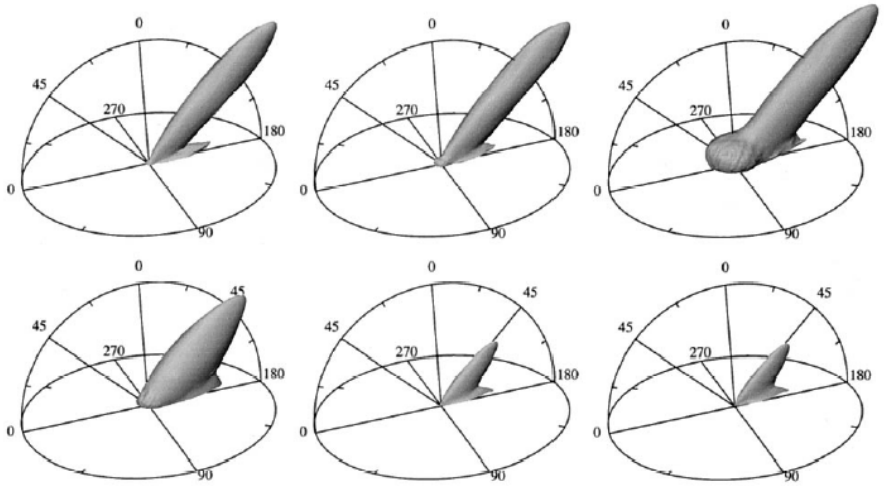


Figure 6.3: BRDF measurement over two different surfaces (top and bottom rows) for each RGB component with fixed incident angle (45°). Images from Marschner et al. work [MWLT00] about a simple image-based process for measuring a surface BRDF.

(for example, translucent objects). BSSRDF makes possible to consider materials with light transport below the surface.

With the BSSRDF model, the outgoing light computation is an extension of BRDF (see Equation 6.2) including integration over an area A around a point x_0 in the surface:

$$L_o(x_0, \vec{\omega}_o) = \int_A \int_\Omega S(x_i, \vec{\omega}_i; x_o, \vec{\omega}_o) L_i(x_i, \vec{\omega}_i) cos(\theta_i) d\vec{\omega}_i dA(x_i) \tag{6.3}$$

where S is the BSSRDF function relating the incoming beam in point x_i and direction $\vec{\omega}_i$ with the outgoing light in point x_o with direction $\vec{\omega}_o$.

$$BSSRDF = S(x_i, \vec{\omega}_i; x_o, \vec{\omega}_o) \tag{6.4}$$

In [JMLH01], the authors show some results for translucent objects and also for human skin rendering that were not possible with original BRDF. Although much faster than *photon mapping* (see next section), the algorithm based on BSSRDF is not appropriate for interactive visualization.

6.3 Photon Mapping

The most popular algorithms for rendering (rasterization/z-buffer and basic ray-tracing, see Section 3.1) do not consider indirect illumination in their standard implementations. Some important effects are not computed, such as caustics and light reflection from object to object. The *photon mapping* method [Jen96] was created to compute *global illumination*. The technique is based on the phenomena happening with a light beam after leaving its source. The algorithm has two steps, in the first one the photon maps are constructed and, in the second one, the scene is rendered.

Step 1. For each light source in the scene, a large number of photons are casted through the scene. When the photon reaches a surface two actions are triggered: the information is stored within the photon map; and, according to the surface BRDF, new photons are generated. The map is recorded using a spatial structure, for example a kD-tree (see Section 3.2).

Step 2. The rendering step is done with an extension for the ray-tracing algorithm. First, for each pixel, the algorithm finds the intersection point x between the ray $\vec{\omega}_o$ (passing through the pixel) and a surface. Then the radiance L_o leaving point x is computed based on the N photons with the shortest distance to x. So, Equation 6.2 is approximated by:

$$L_o(x, \vec{\omega}_o) \approx \sum_{p=1}^{N} BRDF(x, \vec{\omega}_{i,p}, \vec{\omega}_o) \frac{\Delta\Phi_p}{\pi r^2} \tag{6.5}$$

where each photon p was originated from direction $\vec{\omega}_{i,p}$ with flux $\Delta\Phi_p$; and r is the radius of the smallest sphere centered in x containing all the N photons.

In [Jen96], Jensen has used two extra types of photons: caustic and shadow photons. The caustic photons are created by densely casting them towards the specular objects. They are recorded in the *caustic photon map*, when they hit a diffuse surface. The shadow photons are created by extending the photon ray after its first intersection, recording the further intersections in the *shadow photon map*. Then, this map can be used for accelerating the shadow test in rendering time.

A GPU implementation for photon mapping has been done by Purcell et al. [PDC+03], obtaining good results. Besides photon mapping, there are other algorithms for global illumination rendering. Radiosity [GTGB84] and Monte Carlo ray tracing [Kaj86] are also interesting techniques for the same purpose.

6.4 PRT function

The techniques previously discussed in this chapter are not able to be reproduced interactively. This is a consequence of the large amount of computation needed to simulate what happens in the microscale. However, Sloan et al. [SKS02] have introduced a real-time rendering method based on the *Precomputed Radiance Transfer* (PRT) function. In preprocessing, they create a set of functions over the object surface representing the transferred radiance from the object onto itself. In real time, the ambient light distribution modulates the transfer functions coefficients to obtain the final result.

The transfer functions are represented in a spherical harmonic basis (which is an orthonormal basis, such as the Fourrier series, but appropriate for the surface of a sphere). The spherical harmonic (SH) can be efficiently used to store low-frequency lighting environment, requiring few coefficients (9-25). So, the preprocess computes the SH coefficients of several points over the surface (for example, the vertices of a mesh), considering self-shadows, interreflection and scattering, by means of a global illumination calculation. The same computation is done for the environment light, so both can be used for the real-time rendering. Note that the object can be freely rotated in the environment because SH functions are rotationally invariant. If the environment changes, only their coefficients must be recomputed.

To compute the coefficient c_i, it is necessary to integrate the light function $light(\theta, \phi)$ in spherical coordinates with the correspondent SH function y_i:

$$c_i = \int_{2\pi} \int_{\pi} light(\theta, \phi) y_i(\theta, \phi) sin(\theta) d\theta d\phi \tag{6.6}$$

where $sin(\theta)$ appears because of the spherical coordinates nature, where the "equator" and the "poles" have different weight.

In Figure 6.4 we can see how coefficients are used in real-time to obtain the final image (computations are done on the GPU).

Although based on the microscale light properties, the PRT results enhance the macro visualization of objects, due to the focus on low-frequency lighting in Sloan et al. work. Later, they have done an extension, titled *Bi-scale radiance transfer* [SLSS03] that also

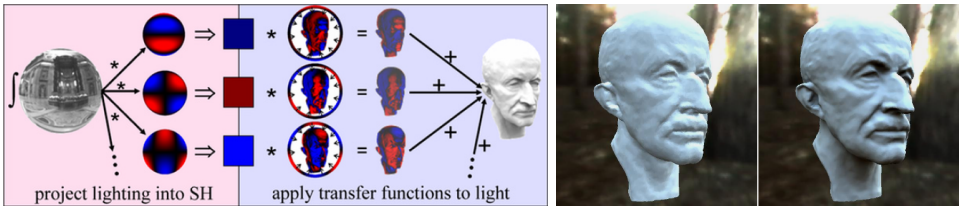


Figure 6.4: *Left:* The SH coefficients of the ambient light are multiplied by the SH coefficients of the transfer functions over the object. *Right:* The two images compare the result between simple unshadowed irradiance versus PRT based rendering. Note the difference in the neck, under the nose and in the ear (images from Sloan et al. work [SKS02]).

considers the mesoscale lighting. Finally, in 2004, they proposed the *all-frequency PRT* [LSSS04] that use a larger number of coefficients. To make it work in real-time with so many coefficients, they apply some compression and clusterization based on Haar wavelet and principal component analysis.

Part II

Natural models

Natural models introduction

We have classified models and objects in two families:

- *Manufactured models* (Chapters 9, 10 and 11). In this family we include all objects that are created by industry, especially those that were designed in a computer (CAD models). In general, they are created by engineers or technicians based on equations and formulas, resulting in exact shapes and forms. Some examples are: transport vehicles, manufactured products and industrial plants.

- *Natural models* (Chapters 7 and 8). In opposition to manufactured models, we consider here all objects found in the nature, but also those that are hand made by humans. Some examples are: geological features, sculptures and volumetric data.

Most real-time visualization methods consider only models described by triangle meshes (independently of the model family). However, for some objects, triangle discretization is not the best representation. In this thesis we propose an alternative representation for natural and manufactured models more adapted to their characteristics. The rendering algorithm uses this new representation for interactive visualization.

Our rendering is based on ray-casting special primitives in the graphics card. We call them *GPU primitives*. We have implemented specific primitives for natural objects (based on an explicit representation) and for manufactured objects (using implicit surfaces).

With primitives more adapted to the models (compared to triangle meshes), we aim to achieve the following benefits:

- *Visual quality enhancement*. More adequate primitives may result in better images.

- *Performance*. Especially in massive models, the number of triangles often exceeds the limit to achieve interactivity (about a few million triangles nowadays). We reduce the total number of primitives by using some sophisticated ones.

- *Reduction of memory consumption*. The use of parametric information can drastically reduce the use of memory.

We have split this part about natural models into two chapters.

Chapter 7 starts by introducing the principles of GPU primitives (Section 7.1), and explains the details about the GPU height-map primitive (Section 7.2). Finally, in Section 7.3, we present *geometry texture*, our basic primitive to render natural surfaces.

In Chapter 8, we explain the conversion of natural objects represented by triangles into our geometry textures (Section 8.1). We conclude this part by showing the rendering results of our technique (see Section 8.2).

Chapter 7

Geometry Textures

In this chapter we introduce geometry textures. A set of geometry textures can be used to represent complex surfaces as an alternative for triangle mesh representation. Geometry texture visualization is implemented in GPU, using our framework (see Section 7.1). In Section 7.2 we present a general GPU height-map primitive. Then, in Section 7.3 we explain details about geometry textures. Geometry texture applications can be found in Chapter 8.

7.1 GPU primitive principles

Graphics hardware is optimized for rasterizing and solving the visibility of points, lines and triangles. We propose a GPU implementation of new graphics primitives (e.g. spheres, cylinders, ellipsoids, height-maps), extending the graphics pipeline (see Appendix A). Our GPU primitives are visualized through a ray-casting algorithm implemented inside the GPU. The main computations are done in the pixel stage of the pipeline.

To trigger the per-pixel algorithm, it is still necessary to raster a set of standard primitives (i.e. triangles). The projected area of the standard primitives must cover the projected area of the extended GPU primitive. This restriction assures that every pixel from where a casting ray intersects the extended primitive will execute the algorithm (see Figure 7.1).

Based on the above restriction we can now state which type of GPU primitive we can implement:

If a surface or an object has a known bounding-box or convex hull (two-dimensional or three-dimensional) for which there is a ray-casting algorithm, then it is a GPU primitive

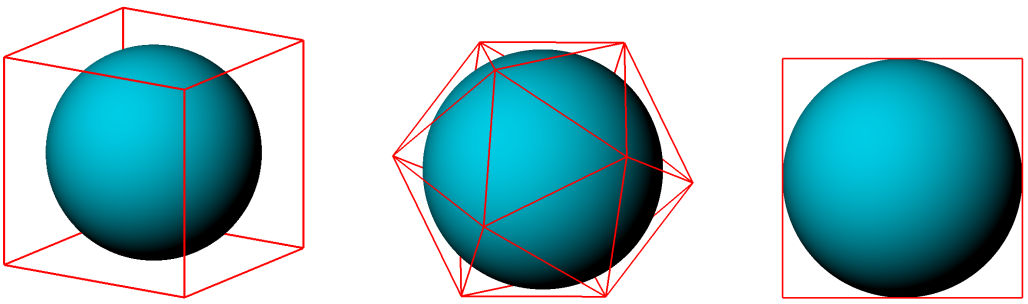


Figure 7.1: Some examples of different sets of standard primitives (forming a cube, an icosahedron and a square) covering the projected area of the sphere GPU primitive. See Section 10.2.

candidate.

These surfaces or objects can be either implicitly described (e.g. quadrics, cubics and quartics surfaces) or explicitly described (e.g. height-maps and volumes). There is still a hardware restriction since graphics cards do not have infinite memory and precision, but the flexibility of modern GPU is almost reaching the same capacity as CPUs.

Another important point for our GPU primitives is that we want to keep their compatibility with the rasterization method. Therefore, objects composed by triangle meshes can also be visualized in the same scene. For this reason we propose a *hybrid* approach, in which ray-casted objects and rasterized ones are displayed together, combining the two most popular HSR algorithms (see Section 3.1). The visibility issue between objects is solved by the z-buffer updated by both rasterization and ray-casting methods. So, in this section we start by explaining our hybrid HSR algorithm and, next, we explain how the GPU primitive visualization tasks are split between the vertex and the pixel shaders[8].

7.1.1 Hybrid HSR

As previously described, the use of rasterization with z-buffer is a well-known procedure that is implemented in all current graphics cards. This hardware implementation of the rasterization algorithm is what ensures high performance and we do want to keep this advantage. To insert ray-casted objects in this context, we have implemented the algorithm in programmable GPU. However, some extra computation is necessary. The ray-caster must use the same image buffer and z-buffer as the rasterized objects. When casting a ray through a pixel and finding an intersection with an object, the algorithm should respect the z-buffer rules:

(i) The depth of the intersection (i.e., the distance from the viewer in direction z) must be computed by using the same model-view-projection matrix used by the rasterized objects.

(ii) Before changing the image buffer, the computed depth should be compared to the current correspondent entry in the z-buffer. If it is closer, then the image buffer can be changed with the calculated color (this color is normally computed in the ray-casting algorithm by using an illumination model).

(iii) If the depth-test is OK, the z-buffer must be updated with the computed depth.

Following these z-buffer rules, the ray-casting algorithm can be freely combined with the rasterization algorithm. It does not matter which objects (the rasterized or the ray-casted ones) are rendered first. Indeed, the ray-casting algorithm can be executed independently for each object, since the z-buffer controls which one is closer to the viewer for each pixel.

Everything is executed in the programmable GPU. In case of objects for which the best rendering algorithm is rasterization, we use the default GPU implementation. For objects that the best rendering algorithm is ray casting, we load the dedicated shaders (pixel and vertex shaders) changing the GPU behavior to compute object-ray intersection instead of standard rasterization.

We call the ray-casted objects *GPU primitives*. All the GPU primitives implemented in this work respect the compatibility criterion and can be freely used in any scene graph. They are all rendered in a single pass and in any order (among themselves or mixed with standard triangles).

In this work we explore the GPU primitives for rendering both implicit and explicit surfaces. For each different surface we have developed a different visualization primitive

[8] Vertex and pixel shaders are the two programmable stages in current graphics cards; see Appendix A for more details about GPU programming.

with different pairs of vertex and pixel shaders. Although each primitive has a distinct pair of shaders, the vertex and pixel algorithms have some common responsibilities, as explained below.

7.1.2 Vertex shader tasks

Although the pixel shader is the final responsible for the ray-casting algorithm, the vertex shader has a key role for our GPU primitives. As explained in Appendix A, any vertex shader has two main tasks: it computes the final vertex position in the eye-coordinate system[9], and transmits the necessary information to the pixel shader (this information is interpolated inside the rasterized polygon).

To optimize the rendering of each primitive, only a subset of the pixels in the screen executes the pixel shader. We call this subset *Ray-Casting Area* or simply *RCA* . We explain it later in this section. For example, a well fitted bounding box (RCA_{Bbox}) can be used to trigger the pixel algorithm only for pixels that are close to the aimed primitive. So, the rasterization of the bounding box front faces[10] defines the set of pixels that will execute the ray-casting algorithm for this primitive. However, as explained later, the 3D bounding box is not the only solution to rasterize pixels that are close to the final image.

The vertex shaders are also responsible for passing information to the pixel shaders needed for a correct ray-casting. The viewing and light direction vectors are captured by the vertex shader and transformed into the *local coordinate system*. In addition, each pixel requires its origin in the local coordinate system. Thus, the vertex shader outputs this information which is correctly received by the pixels due to the interpolation done in the rasterization stage. The use of a local coordinate system enables a faster ray-casting execution. For example, the ray-casting can be done over a canonical representation of the surface.

Another piece of information passed to the pixel shader is related to the z-value. The final z-value is computed individually in the pixel shader (note that this value cannot be interpolated in rasterization as in the case of simple triangles, since it does not have a linear variation). Although it could be completely computed by the pixel algorithm, we use the vertex code to pre-compute part of the z-value equation. The exact computation is explained right after the discussion about *pixel shader tasks*, which is the next topic.

Figure 7.2 lists the vertex shader input/output.

7.1.3 Pixel shader responsibilities

The first task for each pixel is to find an intersection between the ray (defined by origin and viewing direction received from the vertex shader) and the implicit surface. If there is no intersection, the pixel must be discarded. If there are multiple intersections in the domain, the one closest to the origin is taken. The surface normal is given by computing the gradient at the intersection point. Using the normal, the final color is computed by applying a chosen illumination model. Occasionally, the pixel shader can apply extra shading effects such as textures and self-shadow tests. The shadow test is done by repeating the intersection test but with a ray whose origin is the intersection point and the direction is the lighting direction.

Thus, the main responsibilities of the GPU primitive pixel shaders are listed below (optional tasks are in brackets):

- Find intersection

[9] Actually, in the clip-coordinate system, since the output value is computed after multiplying the world coordinate by both model-view and projection matrix. But we refer to it as eye-coordinate for a better understanding.

[10] It is important to notice that there is no need to render all the faces, just the front ones. If all of them are used, it will double the computations, since, for each pixel, two fragments are produced. The pixel shader would then be called twice to solve the same intersection.

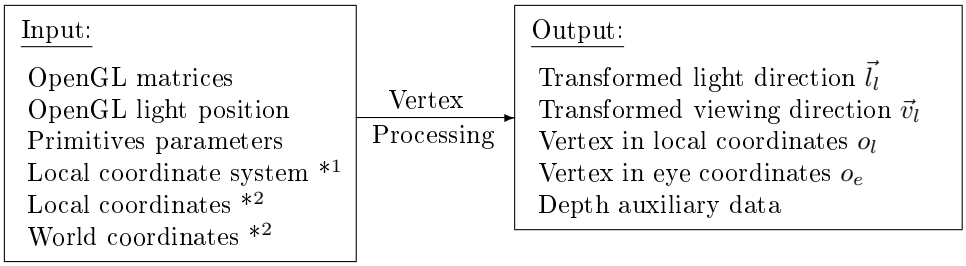


Figure 7.2: Vertex shader common input and output for GPU primitives. (*1) The local coordinate system (or primitive coordinate system) is an input for just a few implemented primitives. There are two situations that needs this input: when the pixel algorithm executes the ray-casting by using global coordinate system; or when the coordinate system is computed in the vertex shader based on the primitive parameters (that is the case of cylinders). (*2) Local and world coordinates are per-vertex information; other inputs are either per-primitive or per-frame information. The world coordinates can be optionally omitted and computed by transforming the local coordinates back to the world coordinate system. *Output:* The "$_l$" index in $\vec{l}_l$, $\vec{v}_l$ and o_l indicates that they are in the local coordinate system and "$_e$" in o_e indicates eye coordinate system. o_e is not used in pixel shader, but the graphics pipeline uses this information to compute the vertex screen position.

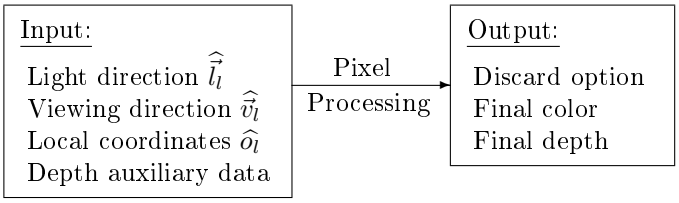


Figure 7.3: The pixel shader input matches the vertex shader output described in Figure 7.2 whose values were interpolated during rasterization (the *hat* in $\widehat{\vec{l}_l}$, $\widehat{\vec{v}_l}$ and $\widehat{o_l}$ indicates interpolated value). During the process, all computations are done in the local coordinate system so there is no transformation, which avoids extra expensive computation. Remark that the GPU primitives use all three possible outputs for a pixel shader (see Appendix A).

- Compute normal
- Compute diffuse color
- [Texture mapping]
- [Specular + ambient]
- [Ray cast self-shadow]
- Compute z-value

Figure 7.3 lists the pixel shader input/output.

7.1.4 Ray-Casting Area

The choice of a good *RCA* for a primitive is based on three criteria ((i), (ii) and (iii)) described as follows.

The set of pixels in a screen window is defined as

$$W = \{\ (x,y)\ |\ x \in [1, width],\ y \in [1, height],\ x,y \in \mathbb{N}\ \}$$

A 3D primitive drawn in the screen W uses also a set of pixels defined by

$$S_{prim} = \mathcal{C}\big(\mathcal{P}(primitive), W\big)$$

$$\begin{aligned} \text{where} \quad & \mathcal{P} : \mathbb{R}^3 \mapsto \mathbb{R}^2 && \text{(projection)} \\ & \mathcal{C}(X, W) = X \cap W && \text{(clipping)} \end{aligned}$$

Each RCA is a subset of W, defining the pixels from where the rays are casted for a given primitive. To correctly render a primitive inside a RCA, S_{prim} must be within the RCA set

$$S_{prim} \subseteq RCA \subseteq W \tag{i}$$

The RCA is constructed by joining areas of projected standard primitives. These standard primitives are the ones usually rendered by the standard GPU pipeline: points, segments and polygons, with their coordinates defined in $\mathbb{R}^3$.

$$RCA = \bigcup_i \mathcal{C}\big(\mathcal{P}(p_i), W\big) \tag{ii}$$

$$\text{where} \quad p_i \in \{point, segment, polygon\}$$

In practice, the RCA primitives (p_i) are the ones passed to the graphics pipeline to be rasterized.

We also define two functions we are interested in minimizing:

$$\begin{aligned} V(RCA) &= \textstyle\sum_i \text{ vertices in } p_i && \text{(vertex cost)} \\ Waste(RCA, S_{prim}) &= \frac{\mathcal{A}_{RCA} - \mathcal{A}_{S_{prim}}}{\mathcal{A}_{RCA}} && \text{(pixel cost)} \end{aligned}$$

$$\text{where} \quad \mathcal{A}_X = \text{area of } X = \#\text{ of pixels}$$

Function $Waste$ measures the wasting of pixels where the ray-casting algorithm runs, but does not result in an intersection with the primitive. Function V counts how many vertices are used to construct the RCA.

The third criterion is stated by

$$\text{minimize } (\alpha_1 Waste(RCA, S_{prim}) + \alpha_2 V(RCA)) \tag{iii}$$

A simple valid RCA would be a square with screen dimension ($RCA \equiv W$), as addressed in [CHH02], but this is the RCA with the greatest $Waste$. The ray-cast pixel shader is executed for each pixel of the rasterized standard primitives. The balance between α_1 and α_2 of Criterion (iii) reflects directly in the balance between vertex and pixel pipelines. The best choice of α_1 and α_2 is application-dependent, and can be adapted for unloading the pipeline when a bottle-neck is identified.

The following list classifies the RCA based on the kind of primitives used to construct it:

- **Polyhedron faces** The faces of a bounding volume are good candidates for constructing a RCA. The simplest example is the bounding box (RCA_{Bbox}), which has vertex cost $V(RCA_{Bbox}) = 8$. However, for some primitives, a simple bounding box results in a high pixel cost ($Waste$ function) if the parallelepiped does not properly wrap the primitive. In the case that the bottle-neck of the application is the pixel pipeline, better fitting polyhedrons can be proposed (see Section 10.8 for an example) even if that increases the vertex cost, giving priority to α_1. On the other hand, if the rendered primitive is repeated many times in the application scene, a high vertex cost might impact the vertex pipeline, and this suggests increasing α_2.

- **Polygons** Another valid RCA is formed by only one planar polygon. In this case, the interior of the polygon works as a window through which it is possible to see the ray-casted primitive. A natural strategy would be to use the polygon RCA with a billboard behavior (for more about billboard, see Section 4.4). If the chosen polygon is a rectangle (probably the most common choice), then the vertex cost is $V(RCA_{rect}) = 4$.

- **Points and Segments** The point primitive implemented in the graphics card can actually play a more important role than just a single point. This happens because its size can be customized. Indeed, we can say that the graphics card point is a square with scalable side size. We are interested in the use of points as squares by scaling their size to much larger than a point. The advantage of these scalable points is that they can be used as billboards without any extra effort in keeping the billboard screen faced (it works as a gratuitous billboard). There is only one simple extra task: establishing the point size. Besides that, the RCA_{point} has the lowest possible vertex cost $V(RCA_{point}) = 1$. A very interesting primitive to ray cast using the RCA_{point} is the sphere (see Section 10.2).

We finish this section about GPU primitive principles by explaining the z-value computation used in all implementations.

7.1.5 Computing the z-value

In pixel programming, it is optionally possible to output the z-value, or simply *depth*. It must be used when the depth inside a polygon is not a linear combination of the values in the vertices. That is exactly our case.

The equation that computes the z-value of a point $P_w = [x, y, z, 1]$ in world coordinates is:

$$\text{z-value} = \frac{z_e}{w_e} 0.5 + 0.5 \tag{7.1}$$

where z_e and w_e are components of $P_e = [x_e, y_e, z_e, w_e]$, which is the point P_w in eye coordinates. P_e is obtained by applying the *model-view projection* matrix M in P_w:

$$P_e = M \cdot P_w$$

Equation 7.1 is applied by default in common vertex shaders and further interpolated by pixel. One straight way to compute the depth by pixel would be by doing this same computation but inside the pixel code. This means that, to transform a point to the eye coordinate system, we would need its world coordinate available in the pixel shader. However, as formerly explained, our GPU primitives ray casting is done in the local coordinate system. Therefore, the intersection point P should be previously transformed to world coordinates, P_w, then transformed to eye coordinates, P_e, to finally apply Equation 7.1. Transformation is an expensive operation and should be avoided. For this reason we have elaborated a more efficient way to compute the depth without multiple transformations.

The intersection point P is computed by the ray-casting algorithm ($P = \widehat{o_l} + t\widehat{\vec{v}_l}$). (Remember that *hat* denotes interpolated value from vertex to pixel.) If $\vec{v}_l$ is a linear transformation of $\vec{v}_e$ we can assert that the eye coordinates of P is:

$$P_e = \widehat{o_e} + t\widehat{\vec{v}_e}$$

where $\vec{v}_e$ can be computed by applying M to $\vec{v}_w$ ($\vec{v}_e = M \cdot \vec{v}_w$). Note that the homogeneous coordinate of $\vec{v}_w$ is zero, since it is a vector and not a point in space.

Since depth computation only uses the z and w components, the pixel code can compute it with just four auxiliary scalars: $o_e.z$, $o_e.w$, $\vec{v}_e.z$ and $\vec{v}_e.w$. The z-value is then computed by:

$$\text{z-value} = \frac{\widehat{o_e.z} + t\widehat{\vec{v}_e.z}}{\widehat{o_e.w} + t\widehat{\vec{v}_e.w}} 0.5 + 0.5 \tag{7.2}$$

In the vertex shader, $o_e.z$ and $o_e.w$ are already computed for outputting the vertex in eye coordinates. So, it is necessary to compute $\vec{v}_e.z$ and $\vec{v}_e.w$, which can be done by

simple 3D dot product between $\vec{v}_w = [\vec{v}_w.x, \vec{v}_w.y, \vec{v}_w.z]$ and the first three components of the third and fourth rows of matrix M:

$$\vec{v}_e.z = [\vec{v}_w.x, \vec{v}_w.y, \vec{v}_w.z] \cdot [M_{(3,1)}, M_{(3,2)}, M_{(3,3)}]$$
$$\vec{v}_e.w = [\vec{v}_w.x, \vec{v}_w.y, \vec{v}_w.z] \cdot [M_{(4,1)}, M_{(4,2)}, M_{(4,3)}]$$

where $M_{(i,j)}$ is the jth element of the ith row in matrix M.

As discussed in the Appendix A about GPU programming, it is a clever strategy to concentrate the effort in the vertex shader more than in the pixel shader. In Equation 7.2, besides the scalar computations done in the vertex shader, the only operation we can pre-compute is the multiplication by 0.5. However, in *orthographic projection* there is a special situation because $\vec{v}_e.w$ is zero. The depth equation becomes:

$$\text{z-value} = \frac{\widehat{o_e.z} + t\widehat{\vec{v}_e.z}}{\widehat{o_e.w}} 0.5 + 0.5$$

which can be rewritten as:

$$\begin{aligned} \text{z-value} &= \widehat{\sigma} t + \widehat{\rho} \\ \text{where} & \\ \sigma &= \frac{0.5\ \vec{v}_e.z}{o_e.w} \\ \rho &= \frac{0.5\ o_e.z}{o_e.w} + 0.5 \end{aligned}$$

Thus, σ and ρ can be computed in the vertex shader, while the pixel shader just uses a single assembly instruction (MAD - Multiply and Add) to find the z-value.

The z-value computation explained up to this point is applied in all the GPU primitives. For this reason, we will not mention the z-computation any more when describing each primitive implementation.

7.1.6 General conclusion about GPU primitives

In this section we have introduced a framework to implement GPU primitives. They are based on per-pixel ray-casting algorithm, totally implemented in the graphics cards. They are a seamless extension for the standard primitives found in conventional GPU (point, line and triangles). Since these new primitives respect the Z-buffer rules, special effect techniques, which are usually applied to triangles, can also be applied to our primitives. For example: occlusion queries, explained in Subsection 3.3.2; and shadow maps [Wil78, BS01], which use a two-pass rendering by first recording the z-buffer of the scene from the light point-of-view.

With the implementation of implicit GPU primitives (Chapter 10), the results show better quality and better performance when rendering thousands of them.

With the GPU height-map primitive (see next section) it is possible to render sophisticated impostors, mesostructure features over surfaces, and complex surfaces represented by geometry textures (see Section 7.3).

7.2 GPU Height-map primitive

7.2.1 Definition and Rendering

A height map (also called height field, relief map or depth texture) is a rectangular regular table where a height is specified for each entry. One way of representing a height

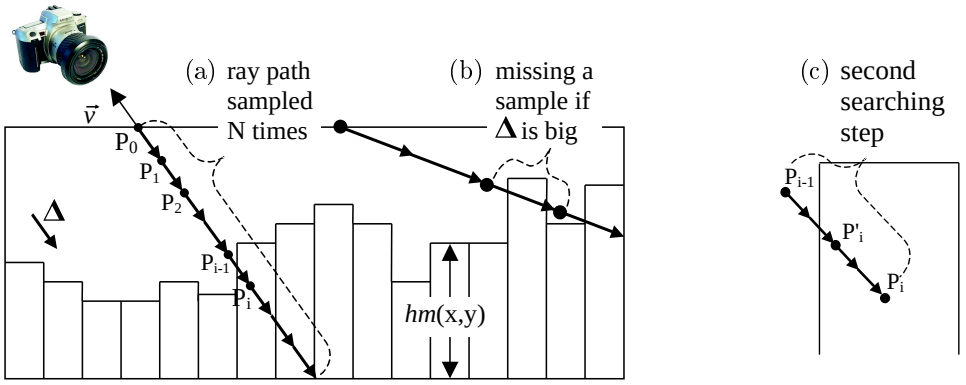


Figure 7.4: Height-map cut view. (a) Searching the intersection; (b) large Δ could result in missing information; (c) a second searching step is done to refine the intersection point.

map is a gray scale image with brightness representing height (see Figure 7.5(a) on page 121). Terrains are an example of surface well suited to be represented by height maps.

This type of discretization, where there are always a minimum and a maximum value, can be represented by a function $hm(x, y) : [0, 1]^2 \rightarrow [0, 1]$. We use a Boolean function $z(P)$, for $(P.x, P.y, P.z) \in [0, 1]^3$, which returns true if $P.z$ is lower or equal than $hm(P.x, P.y)$ and false otherwise.

$$hm(x, y)\ , \quad \text{where} \quad (x, y) \in [0, 1]^2 \quad \text{and} \quad hm(x, y) \in [0, 1]$$

$$z(P) \Leftarrow (\ P.z \leq hm(P.x, P.y)\ ) \quad ? \quad \text{true} : \text{false}$$

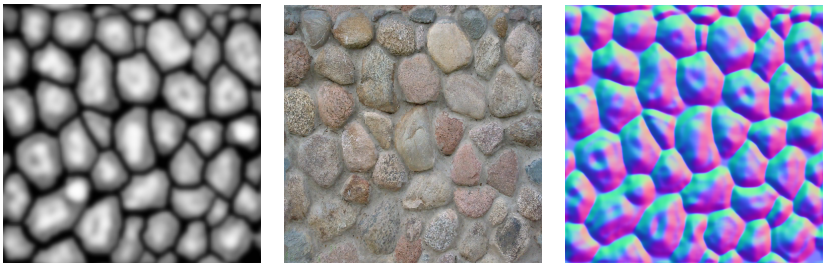
For instance, we can consider a parallelepiped as the polyhedron *RCA* for the height map. The coordinates of its vertices are between 0 and 1 (as in a canonical cube), defining the local coordinate space. The height map data is passed to the GPU through a gray-scale texture. The x and y axes of the parallelepiped are totally coincident with the u and v coordinates of the texture, while z coincides with the height data represented by the gray-scale colors of the height map texture.

As explained in Section 7.1.2, the *RCA* transmits to pixel shading the ray direction already in local space. So, each pixel knows the path of the ray inside the canonical cube, and can easily compute the exit point of the polyhedron using the origin and the viewing direction. At this point, the task is to answer the two main questions of ray-casting: is there an intersection between the ray and the height map? Where is the first touched point?

One strategy is to uniformly sample this ray path in N points P_i (see Figure 7.4(a)) and use function $z(P)$ to find the surface. This could be implemented by a conditional loop varying P_i, from the entering to the exit point, and stopping when $z(P_i)$ is false. In each step a 3D point P, which started with the ray's origin, is translated by Δ, which is the vector from the ray's origin and the exit point divided by N.

```
RayCasting()
{
   P: Point3D
   P ⇐ ray_origin
   Δ ⇐ (exit_point - ray_origin) / N
   while ( P ≠ exit_point  and  !z(P))
     P ⇐ P + Δ
}
```

When a ray does not intersect any height sample, the pixel should be discarded without any contribution to the frame buffer or to the z-buffer. For pixels not discarded, when the intersection point is finally found, the true color can be retrieved from a color



(a) (b) (c)

Figure 7.5: From left to right: the height map, the color map, and the normal map (each map has 512×512 texels). Figure 7.6 shows the rendering results.

texture (see Figure 7.5(b)), and the normal can be either computed directly from the height map or passed as a third texture (see Figure 7.5(c)) to be used for the shading. The coordinates used in the texture lookup are $(P.x, P.y)$ right after the iterations. Finally, the depth is computed to register the correct z-value for the z-buffer.

The decision about the value of N is a trade off between result quality and speed. The height map should also influence the decision about N, since high frequency data could be missed if a large Δ (which is inversely proportional to N) is used (see Figure 7.4(b)).

We propose the following improvements for the height-map GPU visualization:

(i) **Two-steps searching.** Even if Δ is short enough not to miss a sample, the point where the ray touches the object is not always correctly taken. So we propose a second searching step, but this time, the ray is defined to be just between points P_{i-1} and P_i (see Figure 7.4(c)). This search can use the same ray-casting function with N' steps, but it could also be optimized to a binary search, dividing the search domain by two in each step. Therefore, with M steps in the binary search we can multiply the precision quality by a 2^M factor. Notice that this second searching procedure is done in the same pixel code, it is not a multi-pass algorithm.

(ii) **Balancing between steps.** Binary search is much more efficient than the linear search. However, it cannot be the only searching procedure because, along the ray path, there could be several intersections. In other words, the $z(P)$ function may switch different times along the parametric ray for each pixel. On the other hand, when the ray is totally perpendicular to the height map (in a perfect top view), $z(P)$ will change sign at most once. In this case, the binary search can perfectly and efficiently compute the intersection without the previous linear search. Based on this observation, we have implemented a balance between the linear and the binary steps according to the viewing slant. When in a vertical view, we reduce the linear search iterations while increasing the binary search iterations. For an almost horizontal viewing direction we do the opposite, prioritizing the linear search. In some cases the performance doubles if compared to fixed number of iterations (see Figure 7.6).

(iii) **Silhouette.** A pixel whose ray does not result in an intersection should be discarded without any contribution to the final rendering. This is possible due to a `KILL` instruction implemented in pixel shaders (see Appendix A). As a result, it is possible to see a correct silhouette, according to the height map. See Figure 7.6 for an example.

(iv) **Non-rectangular height map domain.** A height sample with value equal to 0 can be considered either as the surface base (its "ground", see Figure 7.7) or as a representation of empty space (see Figure 7.8). The possibility of representing empty space is essential for geometry textures (see Section 7.3).

(v) **Self-shadowing.** Finally, shadow casting can be implemented to consider self-shadowing. We implement it by using the same ray-casting algorithm (without the second search step, which is unnecessary), but the starting point is the intersection, and the direction (which determines Δ) is the light direction. See Figure 7.6 and 7.7.

7.2.2 Parallel Work

Another approach for ray-casting height maps in GPU was proposed by Badoud and Décoret [BD06], focusing on impostors. Instead of combining linear and binary searches, they suggest a *rasterization-based intersection*, which can be accelerated with a *pre-computed robust binary search*. In the following items we briefly explain both techniques:

- The linear search cannot guarantee that a hit will not be missed (see Figure 7.4b). For this reason, Badoud and Décoret have implemented a method that never misses a hit. The idea is based on 2D line rasterization algorithm (such as the Bresenham algorithm [FvDFH90]). The height function $hm(x, y)$ is only evaluated in the *center-lines* of the height map. The center-lines are the lines connecting neighboring texels through their middle point. At each step, they increment in either x or y for the next fetch, without missing any center line. This solution, although more precise, is slower than the linear/binary search combination. The next item is about their improvement for speed purposes.

- The problem with rasterization-based intersection is that, in each iteration, the step is always a small one, retarding the algorithm. With the *pre-computed robust binary search* it is possible to take bigger steps in some situations. In pre-computation, for each discreet point (x, y) in the height map domain, they compute the *safety radius* (based on the *extreme ray* computation). Thus, for a given (x, y) position in the search, the size of the next step can be the safety radius of this point, which is bigger than or equal to the rasterization step. As a result, the next point is either inside the height map (but with at most one intersection between this point and the previous one), or outside of it, but much closer to the hit point. In the first case, they launch the binary search to find the intersection point. In preprocessing, they heavily sample the directions of rays for each texel, but fortunately it takes only 10s in their GPU implementation for a 128×128 height map.

Another different point in their work is how they capture the height map. Rather than using an orthogonal projection (the first natural choice), they use a perspective one. As a result, features that are almost perpendicular to the viewer are better captured. We discuss more about Badoud and Décoret's work in Section 7.3.1.

Policarpo et al. [POC05a, POC05b] use height-map ray-casting for both impostors and mesostructure (see Section 5). They also implement a combination of linear and binary searches, however without balancing. See Figure 7.6 for comparison.

Our balancing method has a key role for ray-casting the height maps with quality without reducing performance.

Height map with curvature

Policarpo and Oliveira have extended their work [OP05] to produce silhouettes and also to work on surfaces with arbitrary curvature. They compute the local curvature by fitting a simplified quadric (with just two coefficients, $z = Ax^2 + By^2$) to each vertex and its neighbors. Although simple, two coefficients are enough to capture positive, or negative, or null curvature in the two main local directions. During the ray traversal, its distance to the ground is computed based on the curvature equation.

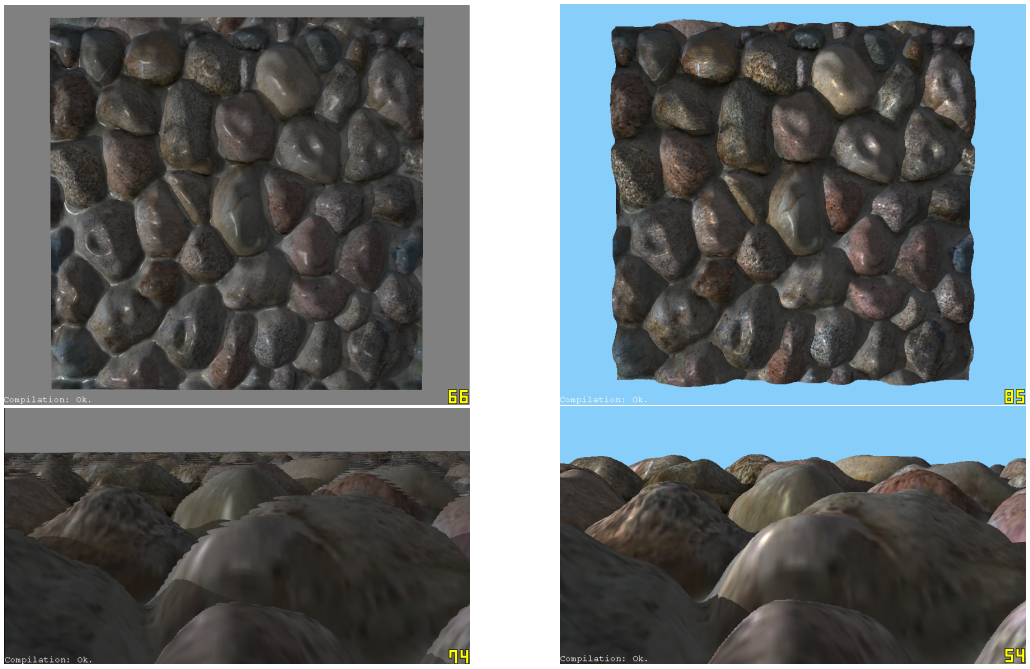


Figure 7.6: Comparison between Policarpo et al. method (*left column*) and ours (*right column*), both implemented in the *Shader Designer* software (©Typhoon Labs). We have implemented a balancing between linear and binary searches. This results in better performance when seen from above (*top right* image), and better quality when seen in a profile view (*bottom right* image). The maps (height, color and normal) used in this example are shown in Figure 7.5. Note: the silhouette is not implemented in their code available with the software.

They have obtained good image results with this algebraic method. However, at each iteration the computation becomes much more complex. At the end of the next subsection, we will show a geometric solution to height map ray-casting on curved surfaces.

7.2.3 GPU height map in different RCAs

In this subsection we show different applications for the GPU height map. The most appropriate *RCA* type to be used (see Section 7.1.4 about different *RCA*) depends on the application.

7.2.3.1 GPU height map in $\mathbf{RCA_{rectangle}}$

Rectangles have a low vertex cost $V(RCA_{rectangle}) = 4$. They are used as a canvas to render the height map. In Figure 7.7 we can see the height map rendered from different perspectives with self-shadowing.

An interesting application for this kind of primitive is to use them as impostors. For example, they could be used for bricked walls or to simulate rocks in a floor (Figure 7.6). For some of these impostors, it is essential to have empty samples in the height map, for cases in which the domain is not an exact rectangle.

The main limitation of simple $RCA_{polygon}$ refers to the silhouette in the case of oblique views. For this reason, the $RCA_{polyhedron}$ is also an interesting choice.

7.2.3.2 GPU height map in $\mathbf{RCA_{polyhedron}}$

A parallelepiped is the first option for a polyhedron *RCA*. Parallelepipes define a 3D canonical space: $\{x, y, z\} \rightarrow [0, 1]^3$. The top face works as the $RCA_{polygon}$ (explained in

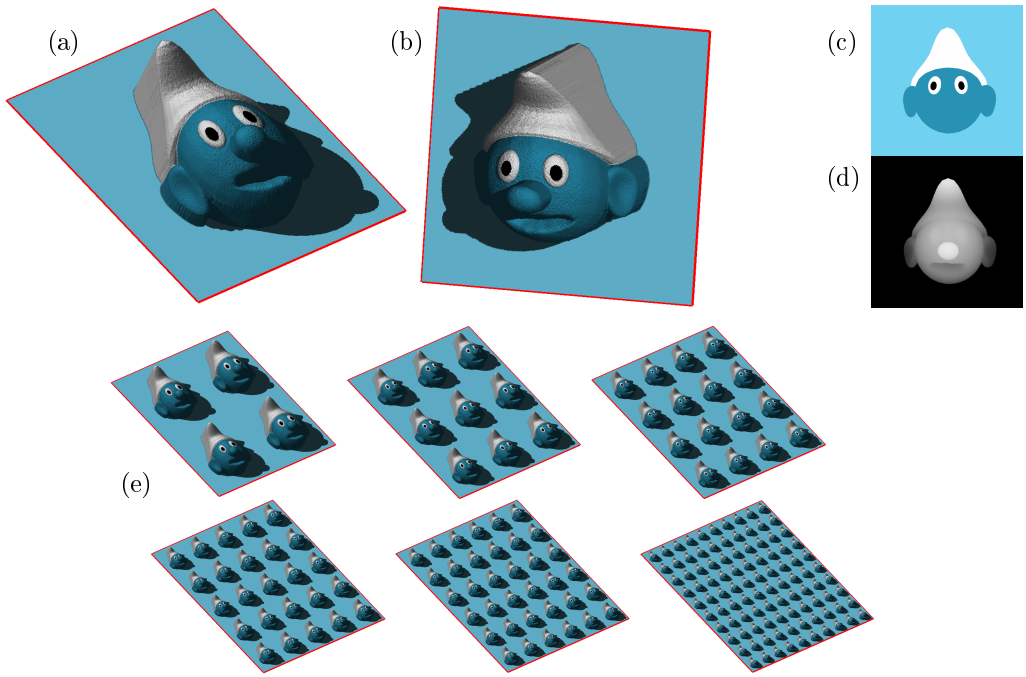


Figure 7.7: (a) and (b) Examples of $RCA_{rectangle}$ for height-map GPU primitive. (c) and (d) Color and height maps used in this example (256×256 resolution). (e) Without any extra computation, the height map can be freely repeated over the plane by changing the local coordinates on the vertices.

the previous topic) from where rays are casted for each rasterized pixel starting in point $P_0 = \{x, y, 1\}$. The lateral faces are similar to the top face, but the starting point's z coordinate varies between 0 and 1 (from bottom to top). In the bottom face, the rays starting point is $P_0 = \{x, y, 0\}$. The bottom has two different behaviors depending on whether the zero value is considered ground or empty space. In the first case, $z(P_0)$ always returns true, so the search is never executed. In the second case, when $hm(P_0.x, P_0.y)$ is zero, the search is executed in the same way as it is done on the other faces.

Actually, $RCA_{polyhedron}$ is not restricted to a parallelepiped. Any polyhedron containing the height map can be used. See Figure 7.8(e) for an example. An advantage of an adapted polyhedron is the possible reduction in the number of pixels being unnecessarily rasterized (those that are discarded). However, the vertex cost is much bigger than using a parallelepiped. See Section 7.1.4 about vertex and pixel costs.

7.2.3.3 GPU height map on surfaces with curvature

The goal of applying the GPU height map over non-planar surfaces is the capability to simulate mesostructures (Section 5). We consider only surfaces described by a triangle mesh. So, for each face, the local coordinate system (the one used for the height map) is set with direction $\vec{z}$ according to the face normal, and $(\vec{x}, \vec{y})$ coincides with the texture coordinate system $(\vec{u}, \vec{v})$ projected to the normal plane.

We have tried two different implementations. In the first one, we simply apply the algorithm explained in Section 7.2.1 for each surface, using the varying local coordinate system. We have obtained reasonable results (see Figure 7.9), but two problems arise. The first one is about the silhouette: how to determine which pixels that should be killed. The second one is how to avoid abrupt changes between neighboring faces.

To solve the first problem we considered a threshold distance for the ray. If the search passes this limit, the pixel is discarded (see Figure 7.9(c)). However, some significant

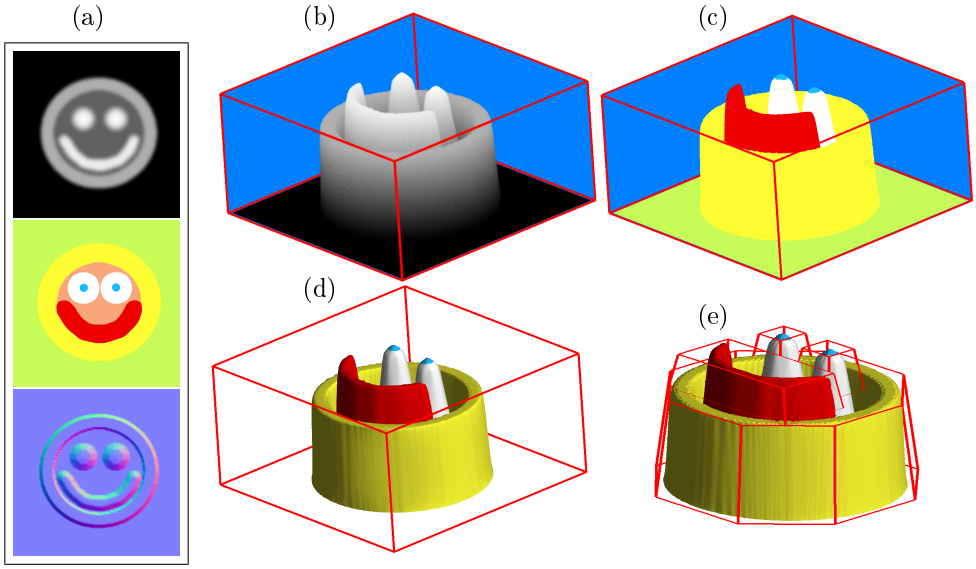


Figure 7.8: (a) The three maps used in this example (height, color and normal) in 1024×1024 textures. (b) The faces of the cube are rasterized and the algorithm is executed for each pixel, ray-casting the height map. Pixels in blue are discarded. (c) The pixel shader fetches the color map according to the hit point. (d) The final shading is obtained using the normal map. (e) The same per-pixel algorithm is used over the rasterized pixels of a more complex polyhedron. The advantage of this polyhedron is a reduction of the pixel cost, since fewer pixels without any contribution are being processed. However, the vertex cost is higher.

artifacts appear (Figure 7.9(f)), which prevents the adoption of this solution. A simple alternative could be done without silhouette implementation (Figure 7.9(d)).

To solve both problems, silhouette and discontinuity, we proposed a geometric solution explained below.

7.2.3.4 Geometric solution for surfaces with curvature

The curvature of a surface can be locally defined by their *principal curvatures*, k_1 and k_2, associated with their *principal directions*, $\vec{e_1}$ and $\vec{e_2}$. With the normal vector $\vec{n}$, they form an orthogonal system in $\mathbb{R}^3$. See [dC76] for more information about this topic. In each principal direction the curvature can be convex, concave or planar, resulting in different combinations. Following, we show the geometric solution for one specific curvature combination and how it could be extended for all kinds of curvatures.

We have chosen a cylinder, which has a convex/planar curvature all over the surface. To map the height-map space into the cylinder space, we follow the principal directions as demonstrated in Figure 7.12, distorting the height-map space (Figure 7.10).

The *RCA* is a simple cylinder mesh. The highest points of the height map touch the mesh. The ground of the height map defines an internal cylinder that will play the role of the cylinder base in the final view.

In Figure 7.11 we show two different situations for a ray traversing the height map. Rays in *Situation I* never touch the ground, and rays in *Situation II* may touch the ground. For *Situation I*, rays can exit the cylinder without touching the height map and the corresponding pixel should be discarded. In the bottom of each picture in Figure 7.11, we show what happens in height-map space for both situations. The rays that are straight in the cylinder space become curved. Thus, our proposition is to search for intersections using the distorted ray instead of curving the height map.

In practice, for our linear search, we need to sample the curved ray in regular spaces. The computation of correct samples on this ray would be expensive, since it would require

Figure 7.9: (a) We want to apply a mesostructure pattern over a discretized torus. The mesostructure is represented by the height and color maps in Figure 7.5. (b) Each torus face runs the height map GPU ray-casting, using a local coordinate system. (c) In detail the mesostructure is applied with self-occlusion and silhouette (self-shadowing is disabled in this example). (d) The same as previous one, but without silhouette. (e) The same torus using another height map as mesostructure (see Figure 7.8). (f) The silhouette based on ray distance may result in errors (pointed by the arrows). Note: These examples run in about 60fps when filling a 1024^2 viewport in a GeForce 7900 graphics card.

an extra procedure for each iteration. Therefore, we approximate the curve in *Situation I* by two consecutive straight rays and the curve in *Situation II* by one straight ray. See Figure 7.11.

To do the approximation, we must find the height-map coordinates of the points P_m

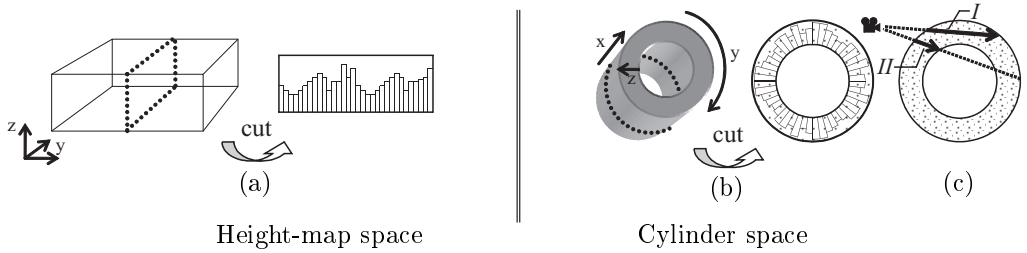


Figure 7.10: (a) The height map will be applied on the cylinder space. (b) In the cylinder space the height map is distorted. (c) In *Situation I* some of the rays may traverse the cylinder without touching any height map sample. The corresponding pixels of these rays may be discarded. In *Situation II* the rays are never discarded, since they always hit something (at least the base ground).

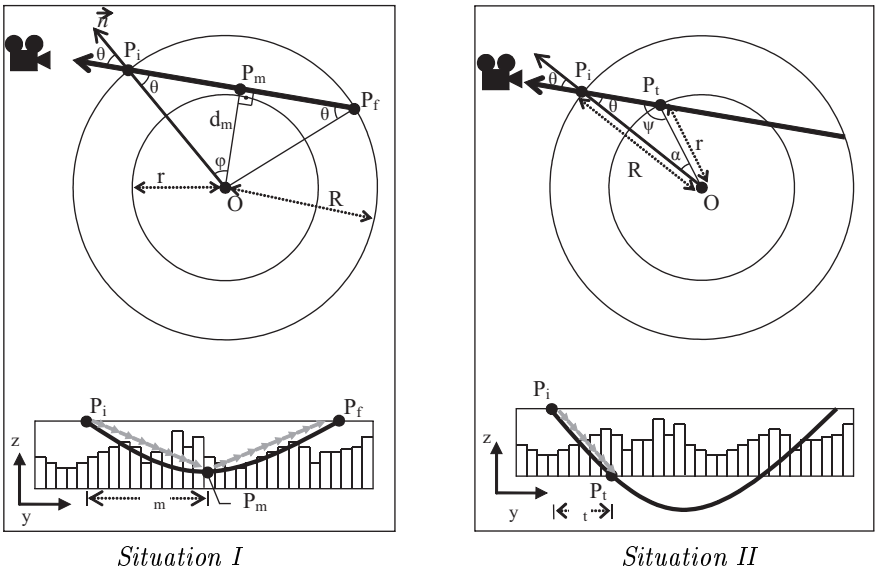


Figure 7.11: The pictures show the geometric interpretation of the rays in both situations I and II, and in both spaces: cylinder space above and height-map space below. Our algorithm computes the points P_i, P_m, P_f and P_t to obtain an approximation of the ray path (sequence of arrows in gray).

for the rays in *Situation I*, and P_t for the rays in *Situation II* (P_i is where the ray starts, and P_f can be calculated from P_i and P_m). Our variables are:

- P_i , which is the local coordinate of the pixel on the cylinder. $P_i.z$ is always 1 and $P_i.xy$ varies over the surface and can be stored as texture coordinates.
- $\vec{v}$, which is the viewing vector, pointing towards the camera.
- r, R are the internal and external radii.
- τ that indicates how many times the height map is repeated over the cylinder in direction y.
- $\vec{n}$ is the normal in point P_i.
- θ is the angle between $\vec{n}$ and $\vec{v}$ in the principal curvature direction: $\theta = \arccos(\vec{n} \cdot \frac{\vec{v_{xy}}}{|\vec{v_{xy}}|})$

Thus, for *Situation I*, the (x, y, z) coordinates of P_m are:

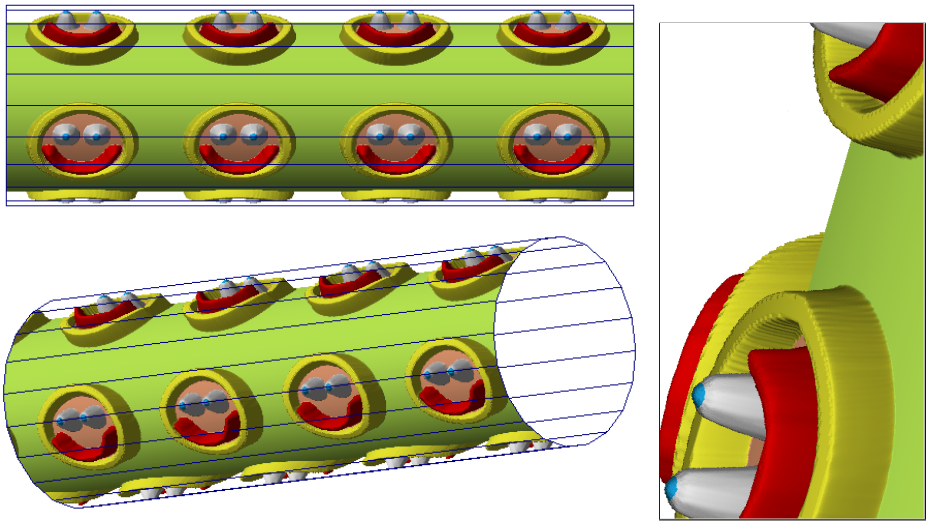


Figure 7.12: We have chosen the cylinder as the curved surface to be used as an example in our geometric algorithm (the silhouette in detail on the right). The solution can be extended to more general curvatures.

$$
\begin{aligned}
P_m.x &= P_i.x + (\Delta_m \frac{\vec{v.x}}{\vec{v.y}}) \\
P_m.y &= P_i.y + \Delta_m, \\
&\quad \text{where } \Delta_m = \tau\phi \text{ , and } \phi = \pi - 2\theta \\
P_m.z &= \frac{(d_m - r)}{R - r}, \\
&\quad \text{where } d_m = R\sin(\theta), \text{ so} \\
P_m.z &= \frac{\sin(\theta) - \frac{r}{R}}{1 - \frac{r}{R}}
\end{aligned}
$$

In *Situation II*, the (x, y, z) coordinates of P_t are:

$$
\begin{aligned}
P_t.x &= P_i.x + (\Delta_t \frac{\vec{v.x}}{\vec{v.y}}), \\
&\quad \text{where } \Delta_t = \tau\alpha \text{ , and} \\
&\quad \alpha = \pi - (\theta + \psi) \text{ , and } \psi = \arcsin(\frac{R}{r}\sin(\theta)) \\
P_t.y &= P_i.y + \Delta_t \\
P_t.z &= 0
\end{aligned}
$$

In Figure 7.12 we show the results of our geometric solution for rendering GPU height-map primitive over curved surfaces.

To generalize the idea, it is necessary to calculate the curvature for each vertex of the mesh that defines the *RCA*. One possible approach is the computation of *tensor of curvatures*, as explained by Cohen-Steiner and Morvan [CSM03]. Once we have the principal curvatures and directions, we can interpolate this information to find the curvature in the viewing ray direction.

This research about height-map ray-casting over curved surfaces was discontinued, but it is a suggestion for future work (see Chapter 12). We have changed the focus of our research, targeting *Geometry Textures* (see next section).

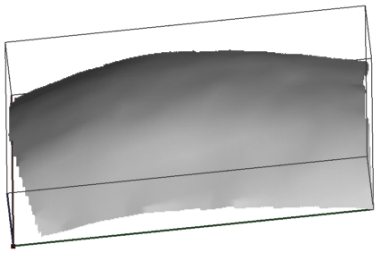


Figure 7.13: Example of a geometry texture patch. A height map is defined inside the parallelepiped's domain.

7.3 The Geometry Textures

For over-tessellated objects (e.g. Michelangelo's David), the triangles are very small when compared to the entire object. In this case, we can say that triangles represent macro and mesostructures of the object. The main idea in our technique is to use a visualization algorithm appropriated to mesostructure but applied for the complete object. We reconstruct the fine-scale geometric details over a simple proxy of the original model.

The problem setting in our work is different from standard mesostructure rendering. We are interested in reconstructing the details of an entire object, which is a non-repetitive pattern, since it changes over the model domain. In our case, the input is a set of height maps converted from the original model to represent the complete geometrical information. Memory consumption is an important issue in this situation, so multi-dimensional techniques (see Section 5.5) are prohibitive.

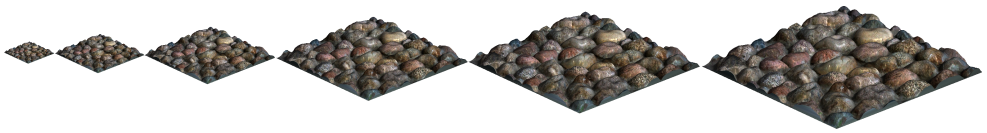
Geometry texture is a geometric representation for surfaces. Its domain is a parallel hexahedron and in its interior a height-map represents the surface (see Figure 7.13). Actually, geometry textures are exactly our GPU height-map primitive (Section 7.2), but restricted to a parallelepiped *RCA*. The domain is not restricted to a perfect rectangle, so empty samples are expected in each geometry texture. Height maps are a very compact way to represent geometry, and this is the reason why we are interested in using them. The construction of geometry texture patches from a complex model, as explained in Section 8.1, has no restriction in direction z (the height direction) and the patches are well fitted around the surface contour (see Section 8.2). In rendering time, geometry textures use our ray-casting algorithm implemented in a pixel-shader (Section 7.2). As a result, the geometry is reconstructed with correct shading, self-occlusion and silhouette.

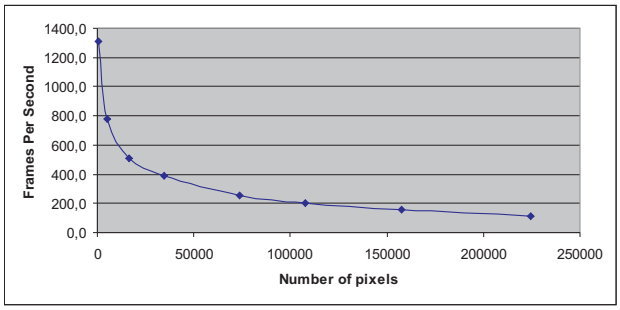
7.3.1 Previous work in multiple height maps

Relief Texture Mapping [OBM00] is the first work on object visualization based on multiple height maps. As explained in Section 4.4, the authors start by capturing the depth of an object from six points of view (up, down, left, right, front and back). In rendering time they use a warping-based technique in the faces of the bounding box to reconstruct the original model.

Baboud and Décoret [BD06] also use multiple height maps to represent and render complex objects. They store the height map of the six orthogonal points of view using an adapted perspective frustum. The use of perspective when capturing the depth increases the detail information about features that are almost perpendicular to the view point. In rendering time, they use a clipped frustum to reduce pixel cost.

Porquet et al. [PDG05] have developed *Real-time high-quality View-Dependent Texture Mapping using per-pixel visibility.* In their work they render a complex surface by using a rough geometric approximation and applying on it colors and normals according to previous captured information, including depth map (or height map). In preprocess-





pixels	FPS
570	1306,5
5247	778,5
16206	506,5
34503	391,0
73668	254,5
107715	203,0
157470	155,5
224382	113,0

Figure 7.14: *Performance test.* We have fixed one viewing direction, varying the zoom over a $RCA_{rectangle}$ with the example of Figure 7.6). The results show that rendering GPU height-map primitives has a LOD behavior. The fewer pixels it must render, the more we gain in performance. The tests have been done with a GeForce 7900 graphics card.

ing, they capture the three maps (color, normal and height) from several points of view. In rendering time, the three closest views to the camera projection are used by the pixel shader that runs for each fragment P on the simplified surface. Based on the height map, the pixel shader reconstructs the equivalent information captured from each point of view (P_1, P_2, P_3). Based on Euclidean distance, the algorithm finds which one among them is the nearest sample to P. The closest one defines which of the three color and normal information should be used in rendering. The main drawback in this application is the lack of silhouette information.

The use of multiple height maps for representation of a complex model is an interesting idea due to the compactness. For rendering, the use of GPU height-map primitives is promising. The drastic reduction of vertices in the scene (the usual bottleneck for complex object visualization) may result in better performance. On the other hand, the GPU height-map primitives visualization converges the effort in the pixel shader. For this reason they have a natural LOD behavior. When far from the viewer, fewer pixels are used, increasing performance. See Figure 7.14 for an example.

In the next chapter we will discuss about how to convert triangle mesh models into a set of geometry textures (see Section 8.1). In Section 8.2 we show the rendering results.

Chapter 8

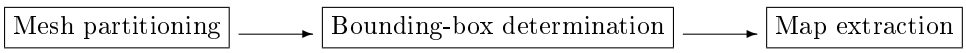
Rendering Natural Models with Geometry Textures

To achieve our goal, we need a way to represent natural objects through geometry textures. To do so, we divide the object-surface domain into a set of charts. Each chart is mapped to a geometry texture. Once we have obtained the set of geometry textures representing the entire object, it can be rendered using our GPU ray-casting algorithm.

In Section 8.1 we explain the conversion algorithm, which extracts a geometry texture set from a polygon-mesh surface. Section 8.2 shows the results obtained with some examples.

8.1 Conversion

The input for our conversion procedure [11] is a polygon (or triangle) mesh and the output is a set of geometry textures. Figure 8.1 shows the complete algorithm for converting triangle meshes into geometry textures. The algorithm is basically executed in the following sequence:



Among the three conversion steps, *mesh partitioning* is the most difficult and criticial for the success of the algorithm. Some conditions are expected when partitioning the input model mesh into charts:

(i) Since height maps can have only one height value for a given (x, y) coordinate in the domain, the chart should not contain a folding situation.

(ii) The chart domain should be as square as possible to reduce empty spaces (this item is also related to bounding-box determination).

(iii) Ideally, the charts should have similar sizes, so the maps would have more uniform resolution along the represented surface (this issue can be overcome with adaptive map size).

(iv) For visualization purposes, neighboring charts must share their boundary, resulting in some overlapping between them. Right after partitioning we add an extra triangle ring around each chart (see Figure 8.2b).

[11] Part of the work described in Section 8.1 has been done in cooperation with postdoctoral researcher Bin Wang.

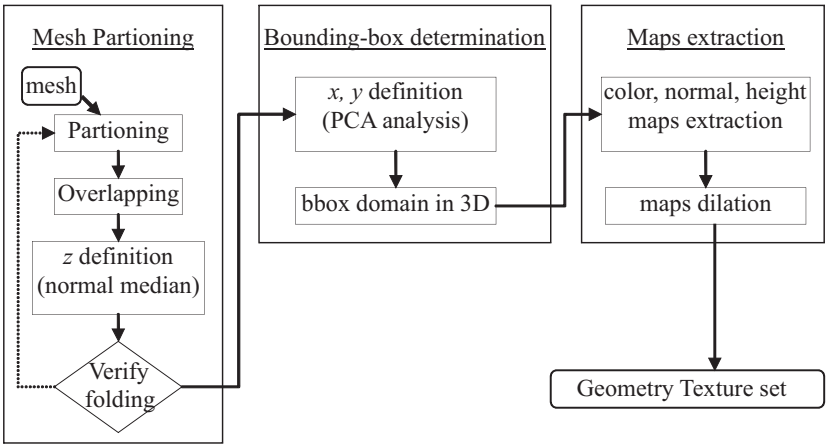


Figure 8.1: This figure presents the complete procedure for converting polygon meshes into a set of geometry textures. Note that the *verify folding* step sends a chart back to the *partitioning* step in case of a folding situation. After this conditional step, the conversion flow is executed individually for each chart.

Bounding-box determination includes finding the best orthonormal coordinate system and domain for each one of the charts. Direction $\vec{z}$ is taken as the median normal of all triangles in one chart. Then, $\vec{x}$ and $\vec{y}$ must be chosen based on the minimization of empty spaces in the domain (item (ii) in the list above). We have used PCA (Principal Component Analysis) algorithm to determine directions $\vec{x}$ and $\vec{y}$. Finally, we project the coordinates of all triangles to obtain the exact domain in each direction. $(\vec{x}, \vec{y}, \vec{z})$ forms the local coordinate system of the chart.

In the *map extraction* phase, there are three maps we want to obtain: height, color and normal maps. We have developed a GPU solution to extract the three maps from one chart. Based on the local coordinate system of each chart, we render its triangles in an orthogonal view projection perpendicularly to direction $\vec{z}$, filling out the viewport with a user-defined resolution (for example, 256×256). We repeat the procedure three times capturing the image buffer:

- To obtain the *color map* we render the colors without illumination.
- To generate the *normal map* we assign (x, y, z) global coordinates of normal $\vec{n}_i$ of each vertex v_i to (r, g, b). During rasterization, within each triangle, the normals are interpolated and then normalized again.
- To obtain the *height map* we start by finding in CPU the maximum and minimum coordinates (h_{max} and h_{min}) in direction $\vec{z}$. We assign a value $h_i \in [0, 1]$, computed as $h_i = \frac{v_i.z - h_{min}}{h_{max} - h_{min}}$, to each vertex v_i. This value is interpolated inside the triangles and outputted as luminance. Actually, normal and height maps can be rendered together, using the α channel for height and keeping (r, g, b) for normal.

Note that for space-reduction purposes, we do not use floating-point precision in any of the maps. The image buffer is captured using 8-bit precision.

The rendering algorithm loads the maps as textures. However, before that, we apply a texel-dilation procedure to prevent sampling problems, which are especially common if using mip-mapping. Our dilation algorithm fills empty texels around map domain with interpolated values. This procedure is especially important for normal and color maps. See Figure 8.2.

In the following subsections we discuss two different strategies we have implemented for partitioning. In Section 8.1.1 we show the results of partitioning with PGP (Periodical

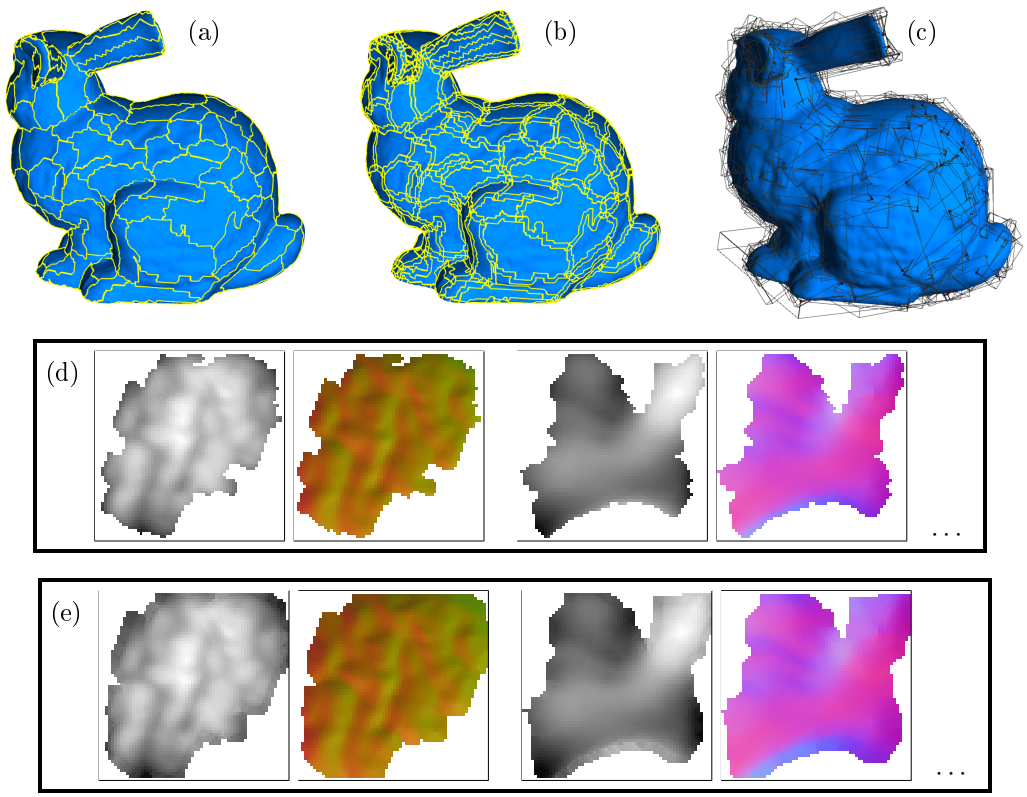


Figure 8.2: (a) Mesh partitioning using VSA (see Section 8.1.2). (b) Overlapping procedure, each chart is increased by one ring of triangles. (c) Bounding-box determination. (d) Map extraction (height and normal maps of two different charts). (e) Map dilation (the same maps on (d) with a two-pixel dilation).

Global Parameterization [RLL+05]) and in Section 8.1.2 we describe the partitioning method based on VSA (Variational Shape Approximation [CSAD04]).

8.1.1 Partitioning with PGP

Periodic Global Parameterization is a globally smooth parameterization method for surfaces. Mesh parameterization (a way to map triangular meshes in 3D space to a 2D domain) has been the object of several studies [BSNJ02, Flo97]. However, PGP can be applied to meshes with arbitrary topology, which is a restriction in other parameterization methods [RLL+05]. Moreover, we can extract a quadrilateral chart layout from this parameterization, which can be used to guide our partitioning.

PGP parameterization is based on two orthogonal piecewise linear vector fields defined over the input mesh. This orthogonality is especially interesting for our partitioning method to reduce empty spaces in the domain (item (ii) of the list presented on page 131). These vector fields can be obtained by computing the principal curvature directions [CSM03], resulting in a parameterization that follows the natural shape of the surface (see Figure 8.3).

PGP has yilded good results in avoiding too many empty spaces (see Section 8.1.3). However, we have found that charts obtained with PGP partitioning present some folding problems (see Table 8.2 on page 135). For this reason we have decided to investigate VSA as an alternative partitioning method, as will be seen in the next section.

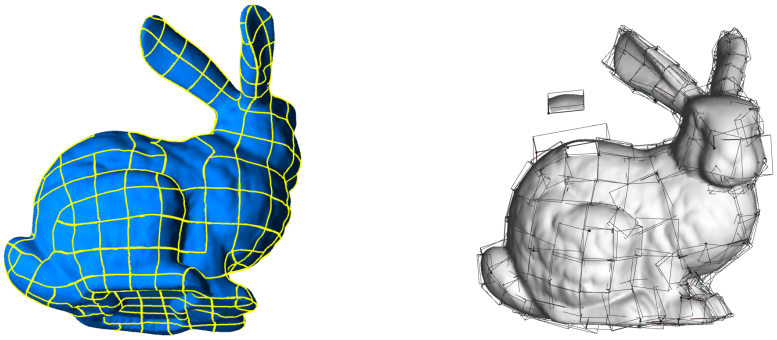


Figure 8.3: Partitioning result based on PGP.

Model	Method	Total texels	Empty texels	%
Bunny	PGP	1008128	240357	23.84%
Bunny	VSA	461312	179194	38.84%
Buffle	PGP	2498048	563652	22.56%
Buffle	VSA	979456	453069	46.25%

Table 8.1: Considering empty spaces, PGP is the best partitioning method.

8.1.2 Partitioning with VSA

Variational Shape Approximation is a clustering algorithm for polygonal meshes that can be used for geometry simplification (as explained in Section 4.3). Our partitioning algorithm based on VSA uses each cluster as an initial chart, which is further increased to overlap neighboring domains (see Figures 8.2a and 8.2b).

The main advantage of using VSA compared to PGP is that it reduces the number of folding cases. VSA uses $\mathcal{L}^{1,2}$ metric, which is based on the $\mathcal{L}^2$ measure of the normal field. This means that it takes into account the normal direction of the vertices to cluster them. A folding situation occurs only if vertices in a chart have a sharp difference in their normal direction (over 90 degrees). With VSA, each chart only contains vertices with similar normal directions. In Figure 8.2a we can notice that VSA partitioning splits the neck and body of the bunny model (as well as its body and tail) because of the sharp difference in normal directions.

In the next section we describe some experiments to compare the PGP and the VSA methods.

8.1.3 Comparing PGP and VSA

In our tests we have partitioned with both PGP and VSA two different models: bunny (see Figures 8.2 and 8.3) and buffle (see Figure 8.4).

Our first test compares how good PGP and VSA are to avoid empty spaces. We have counted for each chart the number of used and empty texels. This was done in the four cases (PGP bunny, VSA bunny, PGP buffle and VSA buffle). For simplicity, the charts were extracted without overlapping and the maps without dilation. The maximum map size was 128×128. A map could be smaller than that (adapted to the chart size), but respecting the power-of-two restriction. Table 8.1 shows the results, indicating that PGP has shown a better performance in this aspect.

In our second test we have counted how many charts have folding problems, considering both models with both partitioning methods. A way to test if a chart has a folding situation is to verify if there is any *inverted triangle* when rasterizing all triangles in local direction $\vec{z}$. Given that all the triangles in the mesh respect a counterclockwise rotation, if at least one of them is clockwise rasterized then the mesh folds itself in direction

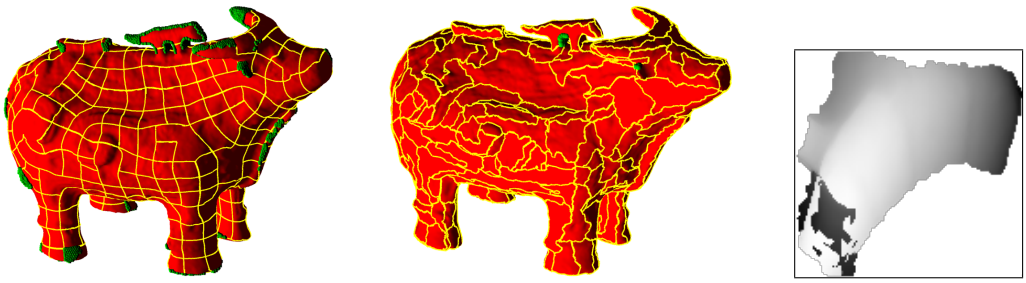


Figure 8.4: *Left:* Buffle partitioning with PGP, triangles with folding problem have their vertices marked in green. *Center:* Buffle partitioning with VSA, triangles with folding problem have their vertices marked in green. *Right:* Example of height map extracted from a chart with a folding problem.

Model	Partitioning Method	Initial Triangles	Initial charts	Inverted triangles	Folding charts
Bunny	PGP	69,451	224	817	45
Bunny	VSA	69,451	222	3	3
Buffle	PGP	117,468	403	2819	62
Buffle	VSA	117,468	397	34	6

Table 8.2: This table shows the number of self-folding charts. VSA significantly reduces the number of folding situations (identified by inverted triangles).

Model (charts)	Map size	VSA	Bbox	Maps	Total	Memory
Bunny (222)	128×128	168.9s	6.5s	8.8s	3min4.2s	1.76MB
Buffle (220)	128×128	184.4s	20.5s	14.6s	3min39.5s	3.73MB
Dragon (355)	128×128	17,402.0s	145.3s	22.6s	4h52min49s	4.07MB
Dragon (453)	128×128	26,796.0s	214.2s	24.9s	7h30min35s	4.21MB
Dragon (453)	256×256			31.7s	7h30min42s	20.0MB
Dragon (453)	512×512			49.5s	7h31min00s	80.6MB

Table 8.3: Conversion times for several models using VSA partitioning. Most of the effort is in the partitioning step. Our VSA implementation was not as optimized as that presented by Steiner et al. [CSAD04], who report at most 10 minutes to partition big models. The last column shows the memory size used by the geometry textures (we have not considered color textures).

$\vec{z}$. Figure 8.4 shows the buffle model after partitioning with inverted triangles marked in green. In Table 8.2 we present the number of folding charts and the total inverted triangles found. Clearly, VSA is more appropriate to avoid folding problems.

Both methods have their own advantages. While PGP is better for reducing empty spaces, VSA is much more efficient in avoiding self-folding charts. Comparing these two points, we have considered the folding problem the most important issue. Self-folding charts cannot be represented by our geometry texture, which is a strong restriction (Figure 8.4(*right*) shows what happens with a height map of a folding chart). On the other hand, empty-space reduction is merely an optimization.

Based on these facts, we have chosen VSA as our partitioning method (see the total conversion time for several objects in Table 8.3). The results presented in the next section have been achieved using VSA.

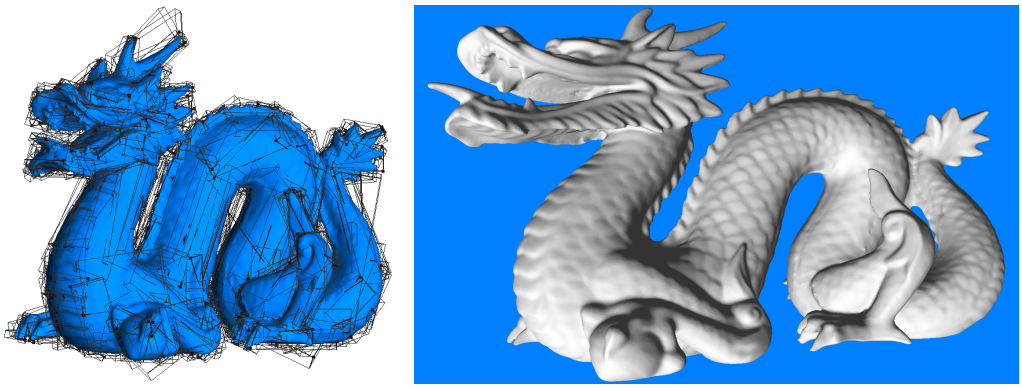


Figure 8.5: *Left:* Geometry-texture bounding boxes of dragon model. *Right:* The same model rendered with 453 geometry textures with maximum resolution 256×256.

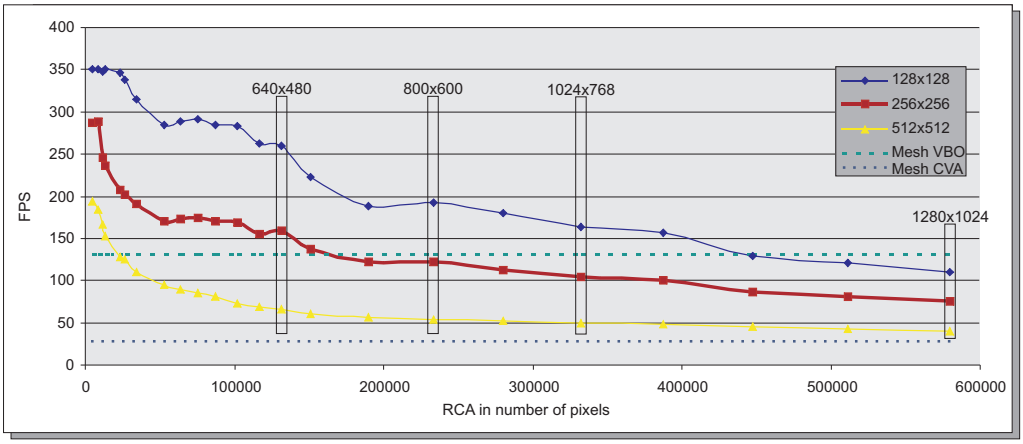


Figure 8.6: We have compared three different geometry texture resolutions (128^2, 256^2, 512^2) and the original mesh rendered with VBO (Vertex Buffer Object) and CVA (Compiled Vertex Array) of the dragon model, in a GeForce 8800 graphics card.

8.2 Rendering and Results

Once we have obtained a set of geometry textures, we will render them using the GPU height-map ray-casting algorithm, as explained in Sections 7.2 and 7.3. In the following sections we discuss performance (Section 8.2.1), rendering quality (Section 8.2.2) and memory use (Section 8.2.3).

We have used the *dragon* model[12] for tests (see Figure 8.5). From its original model with 871,414 triangles we have generated three sets of 453 geometry textures, varying their maximum resolution (128×128, 256×256 and 512×512). The tests were done in an Athlon XP 3800+ 2.4Ghz, 2GB RAM, with a GeForce 8800 GTX, 768MB graphics card.

8.2.1 Performance

Since geometry textures have their performance bottlenecked per pixel (resulting in LOD behavior – see Section 7.3), we have varied the zoom level for the tests. We have plotted the results in Figure 8.6.

[12] http://graphics.stanford.edu/data/3Dscanrep/

To compare performance between geometry textures and common triangle-mesh visualization, we have measured the rendering speed of the original dragon model in two special situations: with *Compiled Vertex Array* (CVA) and with *Vertex Buffer Object* (VBO). Both methods are advanced features in OpenGL and without them the speed would be less than 1 FPS for such model size. CVA guarantees that each vertex is processed only once and not for each triangle containing it. VBO extends CVA and assures that the complete set of vertices is in the graphics card's memory. The VBO feature is present only in recent graphics cards. As a result, for the dragon model with 871K triangles, we have obtained 27 FPS with CVA and 135 FPS with VBO (both plotted as dashed lines in Figure 8.6).

The abscissa axis in Figure 8.6 represents the number of pixels of the dragon's RCA, which is formed by the rasterization of the geometry-textures bounding-box faces (without recounting overlapping fragments). A few RCA pixels mean zoom out, while more RCA pixels represent a bigger model on the screen. For a practical idea, we have highlighted some window sizes (640×480, 800×600, 1024×768, 1280×1024) which the model would fulfill.

The ray-casting algorithm is the same independently of resolution. However, we have verified that the lowest maximum resolution (128×128) was the fastest one. This maybe a result of less graphics card memory in use (4.21MB), optimizing caching and fetching. As we will see in the next section, for quality purposes, 256×256 would be a good choice for maximum resolution for the dragon's 453 geometry textures used in test. However, its performance is slower than the original mesh rendered with VBO for resolutions larger than 800×600.

8.2.2 Quality

We have compared image quality by rendering the three sets of geometry textures for the dragon model and its original mesh. Most visual differences among the four situations appear in the silhouette. For this reason, in Figure 8.7, we focus on the dragon tail. Geometry textures with 256^2 and 512^2 are very close to the original mesh rendering. Based on this observation (which is the same for the whole model), there is no reason in using resolution 512^2 on this case, which would multiply by four the memory size compared to 256^2. However, geometry textures with resolution 128^2 produce an imprecision that can be noticed all over the model (see Figure 8.7(d)).

Seamless Rendering

Since we are rendering a model as a composition of patches, an important issue is to assure the continuity of the reconstruction. A perfect junction is possible due to two details in our technique, as has been previously described.

The first one is in conversion time: for each patch an extra ring of triangles from the neighboring patches is used to generate the geometry textures (in the dragon example, we have used four extra rings). As explained in Section 8.1, in the map-extraction stage, each geometry texture will contain overlap information about the neighbors in its height-map. The overlap area is necessary to avoid cracks and it is naturally treated during rendering.

The second detail refers to the rendering algorithm, and it is based on the fact that the correct z-buffer information is generated. As shown in Figure 8.8, continuity is also guaranteed by the correct computation of the final shading over the overlapping parts, based on the normal maps. The per-pixel depth information is also important to maintain compatibility with any other object rendered in the scene.

Figure 8.7: (a)Original dragon model with 800K triangles. (b)Dragon rendering with 512×512 geometry textures. (c)Dragon rendering with 256×256 geometry textures. (d)Dragon rendering with 128×128 geometry textures.

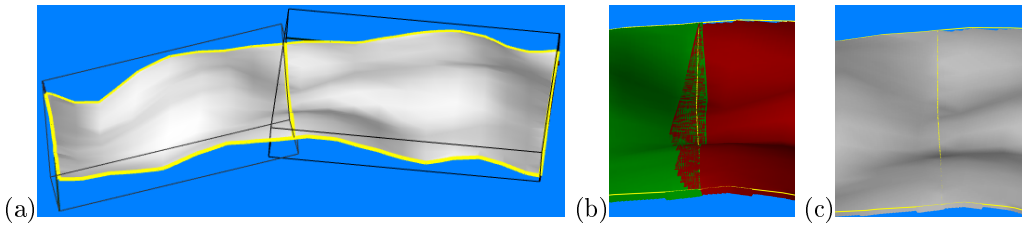


Figure 8.8: Seamless rendering of neighboring patches. Even if continuous pixels are not rendered by the same geometry texture (in Figure (b), green and red colors were respectively used for left and right patches), the final image has no seams (c). We can see that there is a *z-fighting* in the overlapping part, but the fragments are fighting for the same final pixel color.

8.2.3 Memory

As shown in Table 8.3, the memory usage is quadratically proportional to the maximum geometry-texture resolution size: ($128^2 \rightarrow$ 4.12MB, $256^2 \rightarrow$ 20.0MB, $512^2 \rightarrow$ 80.6MB). The dragon's original mesh uses 13.3MB. We have seen in previous sections that the recommended resolution in our tests was 256^2, which has a compatible size to the original model. If memory is a problem, then 128^2 could be used with the drawback of some precision loss (see Section 8.2.2).

Part III

Manufactured Objects and Infrastructures

Manufactured objects and infrastructures introduction

Nowadays, high technology industries make large use of computer resources in all working processes. Transports, health and energy are some of the industries with the largest investments in digital technologies such as numerical simulation, visualization and virtual reality. It is common for the industries to have staff meetings happening in *reality centers* where multidisciplinary teams discuss simulation results or new ideas in products manufacturing making use of special virtual reality devices.

In the **transport industry**, the new trend in the design and life cycle management for complex products, such as planes, cars and trains, is to develop common databases, shared by all the activities involved in the development of the product (e.g., numerical simulation, mechanical engineering and design). Data visualization is a common point in these activities. The rendering software must deal with individual requirements showing the images in interactive frame rates.

It is well known that 3D technology has had an important impact in the domain of health in the last years. More recently, the **chemical industry** has benefit from advances in virtual reality especially related to the design of new drugs. When visualization is coupled to a molecular dynamic simulation, it allows to understand and analyze the mechanisms occurring in the interaction and to define the factors determining its specificity.

Electricity production and transportation requires huge infrastructures. Simulating the global behavior of these structures is an especially difficult task. The principal difficulty comes from the multi-scale nature of the problems: for instance, simulating the behavior of a nuclear reactor requires the combination of mathematical models capturing the microscale (particle escape) behavior with mathematical models operating at a coarser scale (e.g., to simulate the thermodynamic exchanges in the reactor). For visualization, the challenge would be how to render interactively the simulations realized in such different scales.

In the **Oil & Gas industry**, the problems are similar (energy production and transportation). But in this case the problem is even larger and it may reach an international scale. For example, Petrobras (The Brazilian oil company) completed in December 1998 the Bolivia-Brazil gasline, with 3,150 kilometers [13]. During the gasline design, numerous simulations were realized to define the best path considering costs, ambient impacts and geographical limitations. The global management of oil production involves important computational resources, to simulate mechanical infrastructures, to assist oil wells production and exploration and to support the search of new resources.

Visualization plays a key role in all the high technology activities of the abovementioned industries. However, we concentrate our attention on the Oil & Gas industry.

Oil & Gas industry

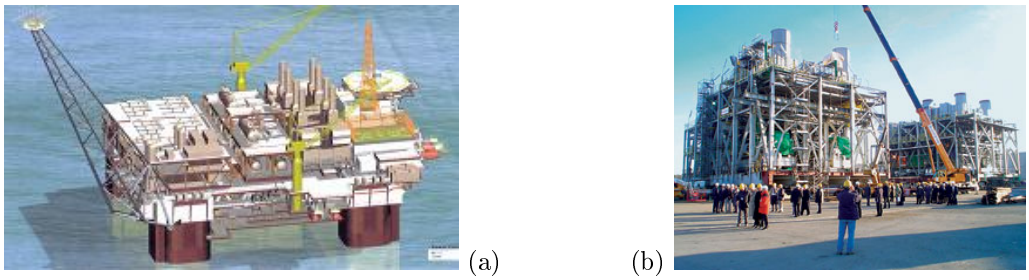
In the oil-industry domain, complex equipments and machineries are routinely used in the main activities of drilling, oil exploration, storing, transportation, transformation and distribution. We identify three different classes of these manufactured objects: complex machines (e.g. drillers and oilrigs, see Figure 8.9a), industrial plants (refineries, platforms and ships, see Figure 8.9b) and ducts (oil-ducts and wells, see Figure 8.9c). Among these three classes, the industrial plants are the most challenging for visualization due to the massive data needed for representing the infrastructure.

For instance, if we consider a CAD (Computer Aided Design) database of an industrial plant, the geometry is decomposed into a set of geometric primitives. Those primitives are typically: *planar primitives* (squares and triangles) for the walls, floors and boxes; *cylinders* for the tubes, pipes and ducts; *cones* for connecting pipes of different radius;

[13] `www2.petrobras.com.br/AtuacaoInternacional/unidades/bolivia/ingles/Main.htm`

(a) (b) (c)

Figure 8.9: Different types of manufactured objects found in the oil-industry domain: (a) oil rig, (b) offshore platform and (c) gas pipe. (Image sources: `www.faculty.fairfield.edu`, `www.hoke.com` and `www.sufi.es`)



(a) (b)

Figure 8.10: The P-52 Petrobras offshore platform. (a) The digital model, (b) the platform is still under construction.

elbows (i.e., torus sections) for connecting pipes in different directions. A typical database contains about one million of those primitives.

As already mentioned in Section 4.1, triangle mesh is a versatile surface representation and its use has been motivated in the last decades due to the visualization algorithm implemented in graphics cards, which is based on triangles rasterization. As a consequence, industrial plants database are often tessellated, yielding over 100 million triangles. As a result, it is impossible to interactively display the massive set of triangles in conventional hardware.

In this part we will focus on the *industrial plants* data, which is an important challenge for interactive visualization.

Industrial Plants

The construction of industrial plants involves a lot of effort and money and it is strategic for many international companies. For example, Petrobras has recently paid 900 millions of dollars for the construction of a new oil platform, the P-52[14] (Figure 8.10). Eventually, this platform will produce about 100,000 barrels per day, what represents up to 150 millions of dollars per month. Computer assisted modeling and visualization, are a significant part of this effort.

Modeling and simulation of complex industrial environments are difficult tasks for the designers and engineers. There are several different domains in this context: oil industries (platforms and refineries), transport industries (planes and ships), factories architecture, and so forth. Even though involving similar kinds of tasks, each domain

[14] Other details about the P-52: 5,000 direct jobs and 20,000 indirect jobs will be generated for its construction. It will operate in approximately 6,000-foot water depths in the Roncador Field Development Project, offshore Brazil. More information can be found in:
`www.bndes.gov.br/english/news/not930.asp`
`www2.petrobras.com.br/ri/ing/DestaquesOperacionais/ExploracaoProducao/Roncador.asp`

is quite different, and as a consequence, there is a considerable multitude of software applications. All the software used for modeling or virtual simulation of those industrial systems absolutely needs to visualize them in real-time.

Modeling such environments involves many professionals (sometimes from different fields), who need to interact. Usually, the companies are equipped with virtual rooms where those professionals can join their results to change and edit the models under construction [RRPB05]. These rooms are also used for simulations and training. Once the models are ready, workers can learn what their work would be in the future installations and even practice what to do in emergency situations. In all those activities, real-time visualization is an essential task.

Industrial models (see Figures 8.9b and 8.10) are mainly composed of multiple sequences of pipes, cable trays and other technical networks, which are basically combinations of simple primitives (cylinders, cones and torus slices). Each software has its own internal data format and different ways to manipulate the primitives. For instance, even a simple cylinder can be stored in different ways: as two points and a radius, as a circle extrusion, as a segment revolution or even as a tessellated approximation. The most common way to export data is the triangular mesh (actually, the only one found in all software). One of the reasons for using triangles as a data format is the rendering, since the current graphics cards are optimized for triangle rendering (which has been the situation for the last ten years, see Section 4.2.2).

However, representing simple primitives by tessellated approximations is clearly not the best solution for many reasons:

(i) it causes much higher memory consumption: a simple cylinder can be stored with 7 floating-point numbers (i.e., the 3D coordinates of two points and the radius); when it is tessellated with 10 vertices in both extremities, it consumes about 400 bytes (to store topological information, vertices position and normal), see Section 9.4;

(ii) even using a very high resolution, the tessellation is always an approximation (it means that, during visualization, there will always be a detailed zoom from which one can see a faceted silhouette instead of a smooth one), see Chapter 10;

(iii) another issue is an ambiguity problem, for example, an 8-sided cylinder could be a representation of a real circular cylinder or a pipe with really 8 faces (as explained in Figure 9.14 on page 161), see Section 9.4.

Some of the above-mentioned problems can be diminished by changing the mesh resolution. If we want more precision in the tessellated approximation we should increase the number of polygons. On the other hand, if memory is a problem, then fewer polygons should be used.

Tessellation presents a clear tradeoff between quality and simplicity. The more quality is demanded, the largest number of triangles is necessary for the representation. However, the excessive number of triangles should be avoided for multiple reasons: memory consumption for storage, CPU/GPU transference overloading and performance downgrading for geometry-limited applications.

Thesis contribution: Industrial plants visualization
Our goal is to display massive tubular CAD models, such as industrial plants, with high quality and in real time without any tessellation of simple geometric primitives (Chapter 11). We have two main contributions:

- an algorithm to retrieve higher-order geometric primitives from the tessellated database in a *reverse-engineering* method (Chapter 9).
- a rendering algorithm that directly uses the equations of higher-order geometric primitives (e.g. cylinders, cones and tori), which avoids their tessellation (Chapter 10);

The advantages of our method for tubular structures rendering are:

- **Quality**, achieved thanks to the direct use of the primitives equations, instead of a tessellated approximation;
- **Speed**, achieved by coding our algorithm directly in the programmable graphics card processor; and
- Significant **memory reduction** when representing pipes. As a consequence, larger databases can be uploaded in the graphics board and displayed in real time.

The remainder of this part is structured as follows: Chapter 9 explains how to recover basic geometric primitives from standard tessellated models; in Chapter 10 we detail the implementation of each primitive, showing the results for industrial plants in Chapter 11.

Chapter 9

Structure Recovery

In our approach, we absolutely need the original information about higher-order primitives (cylinders, torus slices, cones etc), since tessellated data sets will not successfully benefit from our rendering method. One of our examplary data sets, the Petrobras oil platform, partially includes this kind of information. However, this is often not the case, as can be observed in the PowerPlant model (the largest publicly available industrial data[15]). For this reason, we need a *reverse engineering* system to recover the basic implicit surfaces over regular tessellated data.

We start this chapter with some general concepts about surface recovery found in the literature (Section 9.1). Then, in Section 9.2, we describe some characteristics found in our input data that guided the development of our recovery algorithms. Finally, in Sections 9.3 and 9.4, we explain two approaches we have implemented for implicit structure recovery.

9.1 An Overview of Implicit Surface Recovery

Several applications make use of *Implicit Surface Recovery*: surface reconstruction, geometry enhancement, reverse engineering, model-based recognition and geometry simplification. For this reason, implicit surface recovery has been the subject of multiple research work in the last years [Pet02, WFAR99, YAH96, LMM97, WK05, OBA+03, OBA05]. Typically, the input consists of a dense 3D point set obtained after a three-dimensional scanning of an object. However, some methods consider polygon meshes as input information, which is our case.

In our application, we are only concerned with manufactured objects for our reverse engineering process. Usually, objects composed by mechanical pieces can be well described by patches of planes, cones, spheres, cylinders and toroidal surfaces. According to Nourse et al. [NHHM80] and Requicha et al. [RV82], 95% of industrial objects can be described by these implicit patches (and 85% if toroidal patches are not allowed). Thus, our recovery algorithm focuses on finding the equations of the original implicit surfaces used to describe the object.

Petitjean [Pet02] lists the four main tasks usually found in recovery algorithms:

- *estimation*: computes the local surface geometry using differential parameters such as normal and curvature;
- *segmentation*: responsible for dividing the original data into subsets, each one probably forming a unique geometric primitive;

[15] `http://www.cs.unc.edu/~walk/`

- *classification*: this step decides in which surface type a subset should be included (cylinder, torus, cone or some other); and

- *reconstruction*: finds the surface parameters to correctly fit the input data.

Among these tasks there are two that are closely related: *segmentation* and *classification*. In an ideal case, the classification step would clearly benefit from a correct segmentation output. In this case, when trying to find which surface best represents the *segment*, the algorithm would not suffer from disturbing neighboring data. But, unfortunately, both tasks are rarely executed in a perfect sequence. In general, they are executed simultaneously rather then sequentially.

The *estimation* step can be seen as a preprocessing stage for segmentation and classification. In this step, local properties are computed for the whole original data set. Using topological information from the input data, different methods are capable of computing these properties. In general, the properties we are interested in are: normal (n), principal curvature directions (e_1, e_2) and principal curvature values (k_1, k_2). Based on *differential geometry* principles [dC76], the estimation methods vary significantly, and there is no consensus on the most appropriate way to approximate geometric local properties [Pet02].

The *segmentation* step is commonly executed using region-based approaches. As an example, in Section 4.3 we have mentioned the work of Wu and Kobbelt [WK05] based on region clustering for partitioning an input mesh. But, besides *region clustering* there are two other region-based strategies found in the literature: *region growing* [LMM97] and *split-and-merge* [PM86]. In some of these strategies the segmentation is done in combination with the classification step (such as in Werghi et al.[WFAR99]).

Classification is the last difficult task in recovery algorithms, since *reconstruction* is almost a straightforward procedure, provided all the previous steps were well executed.

Following, we introduce two recovery strategies used in our application. The first one, described in Section 9.3, is more similar to conventional surface-recovery approaches. The second one, in Section 9.4, is more industrial-CAD data driven, and it is deeply based on specificities found in our input data. In order to understand the characteristics present in industrial-plant data, we start by enumerating the details and important points of this data type in Section 9.2.

9.2 Industrial-Plant Database Special Characteristics

Ideally, the tessellated representation should never be the only option to access industrial data sets. Especially in our case, we would like to have the original geometric information so we could use it in our visualization algorithm. But in practice this is not the case, and triangle meshes are frequently the only data representation available. One reason is that triangle meshes are currently the *lingua franca* in computer visualization and mesh representation (see Section 4.1). The multitude of software in the numerous CAD areas also contributes to the lack of another common representation or data type.

Reverse engineering typically uses scanned data as input. In this kind of data, there is a dense set of points laid down on object's surfaces. However, the data we are dealing with present a different situation, since they were generated without any scanning or capturing step. Although both data are composed by point sets (and an additional topology information[16]), we can list some different characteristics found in industrial tessellated software-generated data:

(i) They have sparser samples in the point set when compared to scanned data. For example, a cylinder patch can contain only 12 vertices in its border.

[16] Some traditional reverse engineering techniques use topology as input, besides the point set.

(ii) The vertices are regularly distributed. For instance, in most cases, the vertices of a tubular structure will be located on a set of circles (or *rings*), like a backbone and vertebrae. This regularity can also be observed in the topology; for example, in a cylinder, edges are aligned with the axis.

(iii) Besides regularity, the vertices also have a precise location. This means, for example, that if a set of vertices is expected to be lying on the same plane, then this will be the case, except for a very small threshold.

(iv) There is a partial segmentation, with some objects separated from one another. For instance, two different pipelines are separated objects in the data, although each one is composed by different patches (cylinders, cones and tori) that were not yet split.

This last characteristic is a significant advantage of our data set. However, characteristic (i) is a drawback that disturbs traditional recovery algorithms. In Section 9.3 we explain our *numerical method* for surface recovery that tries to overcome this problem. On the other hand, characteristics (ii) and (iii) are an advantage that we have explored in our *topological method* described in Section 9.4.

9.3 Numerical Method

In this subsection we focus on how to overcome the sparse characteristic of CAD industrial data to correctly process the *classification* step, which is the third step in conventional recovery algorithms (*estimation*, *segmentation*, *classification* and *reconstruction*).

Let us consider that the normal $\vec{n_i}$ of each vertex v_i was already computed in the *estimation* step and that the *segmentation* step was already successfully executed. Now we have a set of surface meshes, each one corresponding to a higher-order geometric primitive. In the *classification* step we want to verify which of these meshes correspond to quadric surfaces. As explained in [WFAR99] we can execute a least square fitting to find out the coefficients that determine the quadric equation.

Although some authors suggest a specific search for each quadric surface [FEF97, Pet02], we adopted the strategy of starting by fitting the 10 coefficients (9 in practice) of generic quadrics and then classifying each surface as one of the expected types (cylinder, cone, sphere, etc.).

Our quadric recovery procedure follows the sequence:

Step I _ quadric fitting (by least-square fitting)

Step II _ quadric classification (by evaluating the quadric coefficients)

Step III _ reconstruction (by extracting main directions and by projecting input vertices)

Quadric Fitting (Step I)

We want to find the quadric surface that best fits the input point-set already segmented. In standard quadric fitting, the measure of fitting adequacy is done by evaluating the minimum distance between the point set and the quadric. The goal is to find the quadric surface Q parameterized by 10 coefficients (see Equation 4.1 on page 61) that best describes the point set. For a point set with n 3D points $p_i = (x_i, y_i, z_i)$, we want the solution of the following system of linear equations:

$$f(x_i, y_i, z_i) = 0, \quad 0 > i \geq n \tag{9.1}$$

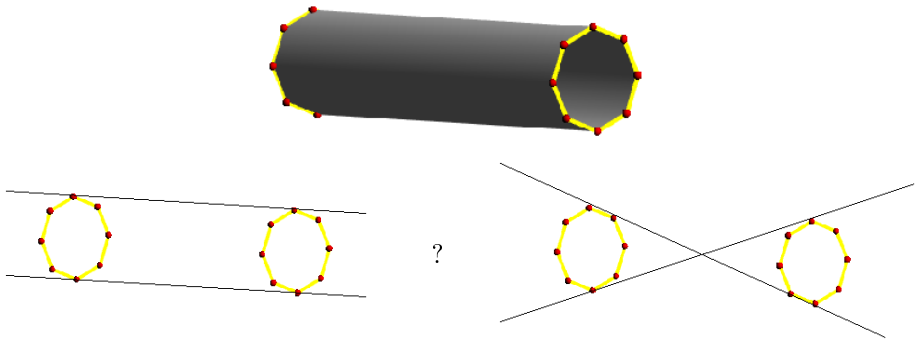


Figure 9.1: *Top:* The 16 vertices (in red) of a cylinder in a CAD model are used for fitting a quadric in the reverse engineering process. That is not much information, and may result in ambiguity. For example, the vertices can be fitted by a cylinder (*bottom left*) or by a cone (*bottom right*).

For dense point clouds, n is much larger than the number of degrees of freedom (9 for quadrics) and the above system is overdetermined. The system is solved by using least-squares fitting to globally minimize the distance between the points and the surface.

We can write the system above in matrix form ($\mathrm{A}x = b$):

$$\begin{bmatrix} {x_1}^2 & 2x_1y_1 & 2x_1z_1 & 2x_1 & {y_1}^2 & 2y_1z_1 & 2y_1 & {z_1}^2 & 2z_1 \\ {x_2}^2 & 2x_2y_2 & 2x_2z_2 & 2x_2 & {y_2}^2 & 2y_2z_2 & 2y_2 & {z_2}^2 & 2z_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ {x_n}^2 & 2x_ny_n & 2x_nz_n & 2x_n & {y_n}^2 & 2y_nz_n & 2y_n & {z_n}^2 & 2z_n \end{bmatrix}_{n\times 9} \begin{bmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ I \end{bmatrix}_9 = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}_n$$

We want to minimize the residual error $r = \mathrm{A}x - b$, so we solve x for the least-squares equation $\left[\mathrm{A}^T\mathrm{A}\right]_{9\times 9}x = \left[\mathrm{A}^T b\right]_9$. This can be done using the Cholesky factorization. (Note that the 10^{th} quadric coefficient (J) is actually a constant so setting the right hand to 1 is equivalent to fixing $J = -1$.)

However, in our application the point cloud is not dense. For example, a cylinder candidate segment can have about 12 vertices with multiple alignments, yielding an underdetermined system of equations with many possible solutions. In Figure 9.1 we show that either a cylinder or a cone can fit the vertices of a typical CAD cylinder. Therefore, we need more equations to provide our system with more independent information. To do so, we have added extra equations to fit the quadric surface normal to the normal $\vec{n_i}$ of each vertex. Thus, for each vertex, three new equations (corresponding to x, y and z components of the normal) were inserted in the system respecting the rule:

$$\vec{n} = \left[\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}\right] \tag{9.2}$$

With this improvement, the number of equations is $4n$ instead of n and we can converge to a single best solution (in least-squares sense). The new system of equations is:

$$\begin{bmatrix} {x_1}^2 & 2x_1y_1 & 2x_1z_1 & 2x_1 & {y_1}^2 & 2y_1z_1 & 2y_1 & {z_1}^2 & 2z_1 \\ 2x_1 & 2y_1 & 2z_1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2x_1 & 0 & 0 & 2y_1 & 2z_1 & 2 & 0 & 0 \\ 0 & 0 & 2x_1 & 0 & 0 & 2y_1 & 0 & 2z_1 & 2 \\ {x_2}^2 & 2x_2y_2 & 2x_2z_2 & 2x_2 & {y_2}^2 & 2y_2z_2 & 2y_2 & {z_2}^2 & 2z_2 \\ 2x_2 & 2y_2 & 2z_2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2x_2 & 0 & 0 & 2y_2 & 2z_2 & 2 & 0 & 0 \\ 0 & 0 & 2x_2 & 0 & 0 & 2y_2 & 0 & 2z_2 & 2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ {x_n}^2 & 2x_ny_n & 2x_nz_n & 2x_n & {y_n}^2 & 2y_nz_n & 2y_n & {z_n}^2 & 2z_n \\ 2x_n & 2y_n & 2z_n & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2x_n & 0 & 0 & 2y_n & 2z_n & 2 & 0 & 0 \\ 0 & 0 & 2x_n & 0 & 0 & 2y_n & 0 & 2z_n & 2 \end{bmatrix}_{4n\times 9} \begin{bmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ I \end{bmatrix}_9 = \begin{bmatrix} 1 \\ \vec{n_1}.x \\ \vec{n_1}.y \\ \vec{n_1}.z \\ 1 \\ \vec{n_2}.x \\ \vec{n_2}.y \\ \vec{n_2}.z \\ \vdots \\ 1 \\ \vec{n_n}.x \\ \vec{n_n}.y \\ \vec{n_n}.z \end{bmatrix}_{4n}$$

Once more, we want to minimize the residual error of the above system of equations. This is done by solving $\left[\mathrm{A}^T\mathrm{A}\right]x = \left[\mathrm{A}^T b\right]$ with the Cholesky factorization.

Classification (Step II)

Based on the quadric coefficients $[A, \cdots, J]$ computed in the previous step, we can find out what kind of quadric surface these coefficients determine. In our case, we are searching for simple surfaces such as cylinders, cones and spheres. Our classification is partially based on the quadric geometric invariants [YAH96]. For this purpose, we need to evaluate the upper-left three-by-three matrix R extracted from the quadric matrix:

$$R = \begin{bmatrix} A & B & C \\ B & E & F \\ C & F & H \end{bmatrix}_{3\times 3} \quad \text{, where } R_{i,j} = Q_{i,j},\ 1 < i, j \leq 3$$

The eigenvalues and the determinant of R are fundamental for quadric classification. The eigenvectors will be also explored in the next step, *reconstruction*.

Reconstruction (Step III)

The input of this step is a mesh representing a surface segment already detected as one of the quadrics we are interested in. The expected output are the exact parameters that can be used to reconstruct this surface segment.

Let us take the cylinder as an example. We are interested in a higher-level representation of the cylinder consisting of the two extreme 3D points P_1 and P_2 and the cylinder radius r (see Figure 9.2).

First of all, using the eigenvalues and eigenvectors it is possible to find the cylinder main direction $\vec{v}$. Assuming that the quadric coefficients describe a cylinder, the matrix $R_{3\times 3}$ contains two identical eigenvalues and one null eigenvalue. The eigenvector associated with the null eigenvalue indicates the cylinder main direction.

To completely describe the cylinder main axis we need one point on the axis besides its direction. To find this point we solve a very simple system of equations. Given the implicit function $c(P) = [P_xP_yP_z1]Q[P_xP_yP_z1]^T$, in which the zero-set defines a cylinder, the derivative of the function is null if and only if P is a point in the main cylinder axis:

$$c'(P) = [0, 0, 0], \text{ if and only if } P \in \text{cylinder axis.}$$

So we can set the following system of equations:

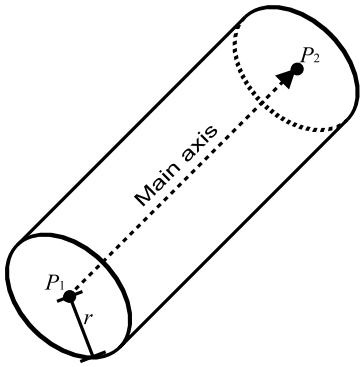


Figure 9.2: In our specification, the cylinder is defined by the two extreme points (P_1 and P_2) and the radius r.

$$\begin{aligned}\frac{dc}{dx} &= 2Ax + 2By + 2Cz + 2D = 0\\ \frac{dc}{dy} &= 2Bx + 2Ey + 2Fz + 2G = 0\\ \frac{dc}{dz} &= 2Cx + 2Fy + 2Hz + 2I = 0\end{aligned}$$

where $[A, \cdots, I]$ are the quadric parameters in Q.

Any point P_0 satisfying the above system is on the cylinder main axis.

So, up to this point, we have obtained P_0 and $\vec{v}$ defining the cylinder axis. But we still need to find out the extreme points P_1 and P_2 and the radius r.

By projecting all the vertices v_i of the original mesh on the main axis parametrically defined by $P_0 + t\vec{v}$, we can find the extreme points P_1 and P_2:

$$P_1 = P_0 + \min_{i=1}^{N}(\vec{v} \cdot (v_i - P_0))$$
$$P_2 = P_0 + \max_{i=1}^{N}(\vec{v} \cdot (v_i - P_0))$$

and the radius r can be computed by:

$$r = \sum_{i=0}^{N} \frac{\parallel (\vec{v} \times (v_i - P_0)) \parallel}{N \parallel \vec{v} \parallel}.$$

Results (Test 1.1 – Helicopter Landing on the P40 Oil Platform)

In the example shown in Figure 9.3, there are 446 separated meshes. Some of them define the interior of the sample model, which contains symbols ("P 40" and "H") and other geometric forms; and most of the meshes around the model are tessellated cylinders we are interested in.

With the numerical approach for reverse engineering described in this subsection we have successfully recovered the 436 cylinders in this model. The execution time was 6.11 seconds[17] for the entire recovery process, which includes *fitting*, *classification* and *reconstruction*.

[17] In an AMD Athlon(TM) XP 2100+ 1.73GHz processor with 1GB of RAM.

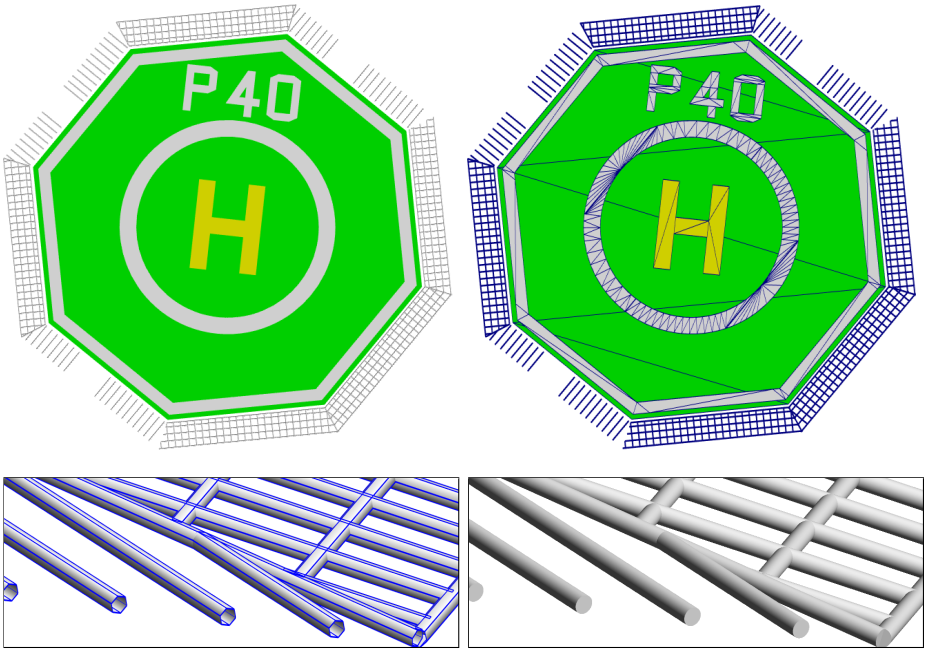


Figure 9.3: Part of an oil platform model (*top left*) made of 446 different meshes (*top right*). There are several tessellated cylinders around the model sample (*bottom left*). Our numerical algorithm has recovered the higher-order information about the cylinders, so we can visualize the cylinder primitives (*bottom right*) with our rendering algorithm explained in Section 10.4.

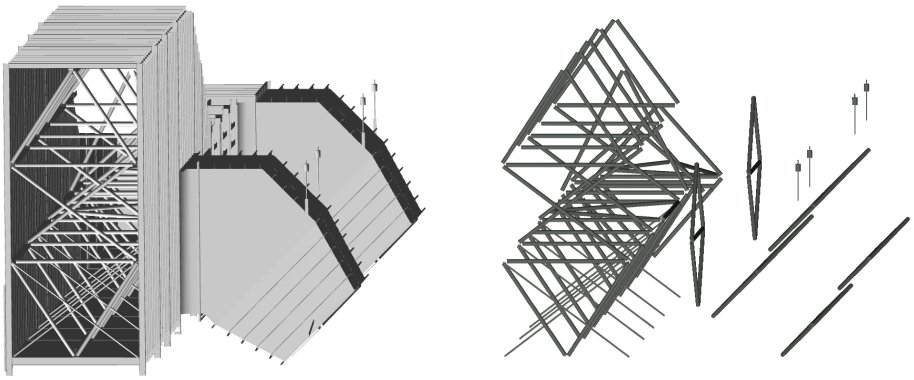


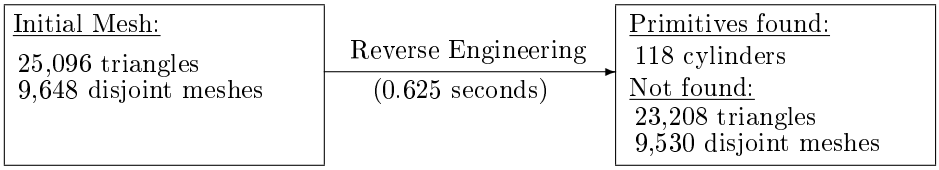
Figure 9.4: PowerPlant piece of data (Section 17, Part b, g1). On the left the original data, on the right the 118 cylinders found. In this piece, there are mainly structural cylinders instead of pipes. As a consequence, they are already segmented.

Results (Test 1.2 – PowerPlant, Section 17)

In Section 17 almost all the cylinders are disjointed meshes in the original data. They appear in the model more as structural objects than as pipes (see Figure 9.4). This is an important issue because our numerical algorithm does not segment the model.

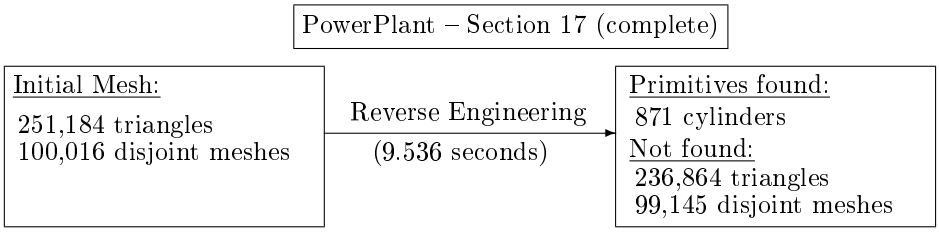
We have applied reverse engineering of the PowerPlant, section 17, part b, g1 file. The following diagram shows the numbers:

PowerPlant – Section 17 – Part b – g1.ply



The number of cylinders found is the same if we execute the topological algorithm (see Section 9.4). The original and found data are shown in Figure 9.4.

Executing the numerical approach on the entire Section 17 we have obtained the following result:



Compared with the topological method, the numerical approach misses some cylinders (there are 901 found with the topological method against 871 with the numerical approach). They are probably some not-segmented meshes that are undetectable by our numerical approach.

Results (Test 1.3 – PowerPlant, Section 18)

Section 18 of PowerPlant is exclusively composed by cylinders and tori. These data were completely recovered by the topological reverse engineering, as shown in detail in Table 9.2 on page 157. However, the numerical approach faced two problems: we had not implemented a torus detection and the not-segmented cylinders could not be found.

PowerPlant – Section 18 (167,012 triangles)		
	numerical	topological
cylinders found	2214	2674
tori found	0	858
recovery time	110.62s	1.43s
unrecognized triangles	129,540	0
effectiveness	22.43%	100%

Table 9.1: Comparing numerical and topological results for the reverse engineering of the PowerPlant Section 18.

As we can see in Table 9.1, the numerical approach could only find 83% of the cylinders and none of the tori. As a result, only 22.43% of the triangles were converted to the higher-order primitives. Another significant difference between both methods is efficiency. While the numerical method executes the reverse engineering in almost two minutes, the topological one takes 1 second and a half (both algorithms delete triangles found at the end).

See Section 9.4 for more details about the topological approach.

Numerical Approach Conclusion

The insertion of equations that fit the normals improved the recovery process of sparse vertices. The results shown here were successful exclusively over disjoint sets of meshes.

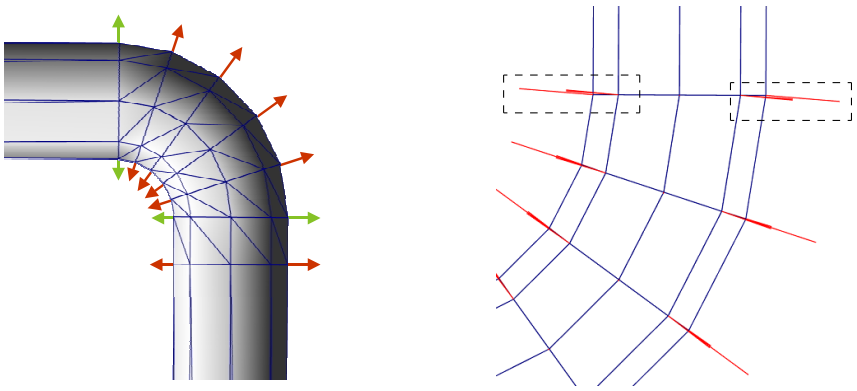


Figure 9.5: **Normal estimation problem.** *Left:* For a tubular mesh with fixed radius, one should expect the normals of vertices in rings (circular sections) to have a perpendicular direction to the ring normal. Even for rings separating two different primitives, such as a torus and a cylinder, the normals (in green) should respect a perpendicular direction. *Right:* However, the normal estimation of a vertex uses the average of neighboring faces, resulting in normals not perpendicular to the ring direction (in red), especially for rings in the frontier between tori and cylinders. (We have verified the same problem in the PowerPlant model, which already contains normal information in its publicly available data set.)

However, for non-segmented data, a problem related to the normal may appear.

The normal at each vertex is obtained by computing the average normal of neighbor faces during the *estimation* phase. Therefore, a vertex shared by a cylinder and a torus will not be perpendicular to the cylinder axis as we should expect (see Figure 9.5)[18]. Even if the normal information is already available in the original data (as in the PowerPlant data set), the normals are not precise (see Figure 9.5(*right*)), and they were also probably computed using the average of normal faces. This problem disturbs our *classification* algorithm and it would also disturb standard *segmentation* algorithms, which are heavily normal dependent [Pet02].

On the other hand, we have observed that regularity is a positive characteristic in this kind of input data. For this reason, in the next subsection we explain the algorithm we have designed, which is well adapted to the specificity of the data.

9.4 Topological Method

Drawbacks of the numerical approach (such as segmentation difficulties, heavy normal dependency [Pet02] and the complexity of tori fitting [LMM97]) motivated us to search for alternative ideas about how to extract higher-order primitive information. The regularity found in the input data was the motivation for our *topological* approach.

Our algorithm is exclusively designed to segment tubes composed of cylinders, truncated cones and "elbows" (the elbows correspond to torus slices, shown in green in Figure 9.6). The basic idea is to find all circular regular sections (the *rings*) and their connectivity. By verifying consecutive rings we can trivially determine if they fit one of these three higher-order primitives. Speed is one of the advantages of this method over traditional numerical approaches for reverse engineering. Numerical approaches are very appropriate for rough data input (for example, obtained by scanner). In our case, however, we exploit the fact that the data are almost constantly regular and can be easily traversed by executing a simple cut-and-find algorithm.

Our method is based on a mesh traversal consisting of the following steps:

[18] In other words, the normal vectors of vertices forming circular sections (*rings*) should lie in the plane containing the ring. See next section for more information about rings.

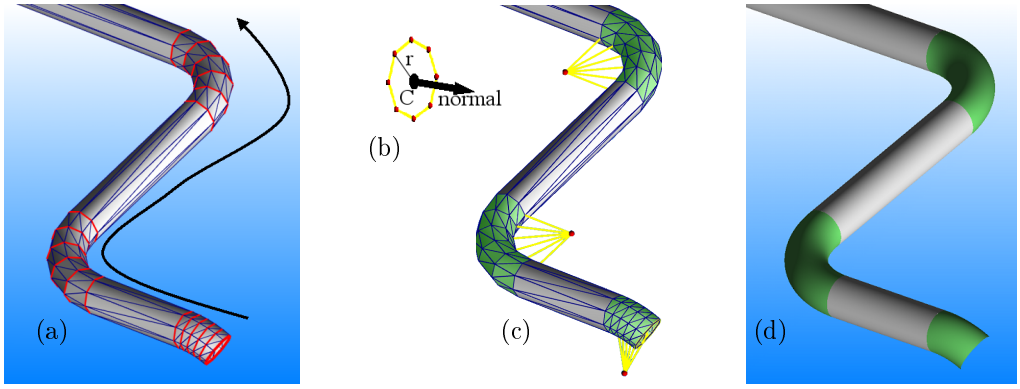


Figure 9.6: (a) Identification of coplanar loops (*rings*) by traversing the mesh. (b) We estimate the properties for each ring: center C, radius r and normal vector. (c) Segmentation and classification are done by evaluating consecutive rings. (d) Reconstruction of the higher-order primitives is obtained by evaluating ring properties.

1. **ring detection** (see Figure 9.6(a)): rings are identified in the mesh as coplanar loops of six vertices or more, equidistant to their center.

2. **tube segmentation and classification** (see Figure 9.6(c)): rings are grouped into primitives according to their radii and normal vector:

   - cylinders: correspond to a pair of rings with the same normal and radius;
   - cone sections: correspond to a pair of rings with the same normal and different radii;
   - elbows: correspond to a set of rings with the same radius, and with co-normals radiating from the same point (see yellow lines in Figure 9.6(c)). For a consecutive pair of rings, R_1 and R_2, we first verify if their centers and their normals are coplanar, if this is not the case, it is not an elbow. Then, we find the radiation point (see red points in the same figure) as the intersection point between three planes: the plane just tested, the plane containing R_1 and the plane containing R_2. This way, the co-normal of a ring is the vector linking the radiation point and the center. (Note that we accept torus slices up to 180 degrees, otherwise they will be cut into two.)

3. **primitive reconstruction** (see Figure 9.6(d)): once the set of rings corresponding to each primitive is determined, it is easy to find the parameters of the primitive from the loop centers and radii. The centers of the extreme sections (P_1 and P_2) of each primitive are simply the centers of the first and last corresponding rings. The radii are also determined from the loops. For elbows, the center C is the intersection of the co-normals (shown in yellow in Figure 9.6(b)). Each primitive is then completely determined by a small set of parameters:

   - cylinder: P_1, P_2, r
   - cone: P_1, P_2, r_1, r_2
   - elbow: P_1, P_2, C, r

After our reverse-engineering algorithm is applied to the database, the result is a set of higher-order primitives (cylinders, cones, elbows) and a set of polygons for the other objects.

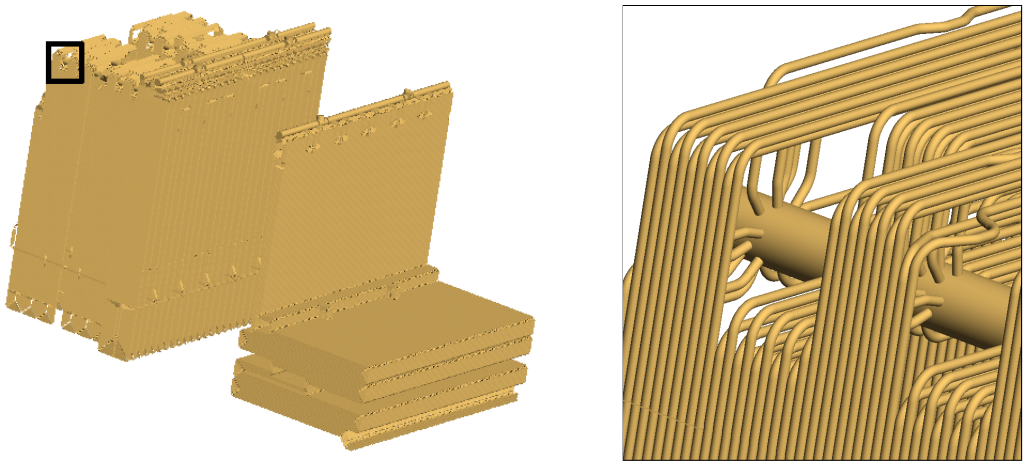


Figure 9.7: PowerPlant Section 1, with about 60,000 GPU-primitives (converted from 3,5 million triangles). We have achieved 70 fps against 8 fps in the tessellated version (see Section 11). It took 24 seconds to recover the primitives in an AMD Athlon(TM) XP 3800+ 2.41GHz, 2GB RAM.

Results (Test 2.1)

We have applied the topological algorithm, based on ring traversal, to the PowerPlant data. In this test we have used Section 1 of the PowerPlant (see Test 2.2 for the entire PowerPlant). The data is distributed in 21 sections and Section 1 is the largest, containing about 3.5 million triangles corresponding to 30% of the model. We have chosen this section in this test for two reasons: it is exclusively made of tubes and pipes; and it represents a significant part of the model.

The following scheme summarizes the numbers in the test:

PowerPlant – Section 1

Initial Mesh:	Reverse Engineering (24.03 seconds) →	Primitives found:
3,429,528 triangles 141,034 disjoint meshes (119MB of memory)		31,372 cylinders 26,546 tori 20 cones (1.9MB of memory)

Note that the number of disjoint meshes is bigger than the number of primitives found because there are some cylinder caps as separated meshes; but they are welded in our recovery algorithm.

A very positive side effect after converting meshes to higher-order primitives is an expressive reduction of memory consumption, as shown in the scheme above. The reduction can achieve 98% of the original data (from 119MB to 1.9MB). As we have already discussed, only a few scalars are enough to describe primitives such as cylinders or tori (the first one requires 7 scalars while the *elbow* torus-slice requires 10). On the other hand, in the tessellated model, these primitives use hundreds of bytes to store vertex positions, vertex normals, topology information, and, sometimes, extra information about the caps covering the extremities (in a higher-order representation, caps are just an optional Boolean). See Figure 9.8 for a comparison in the cylinder case.

The performance mentioned in Figure 9.7 (24.03 seconds to recover the primitives) was measured considering the model already in memory.

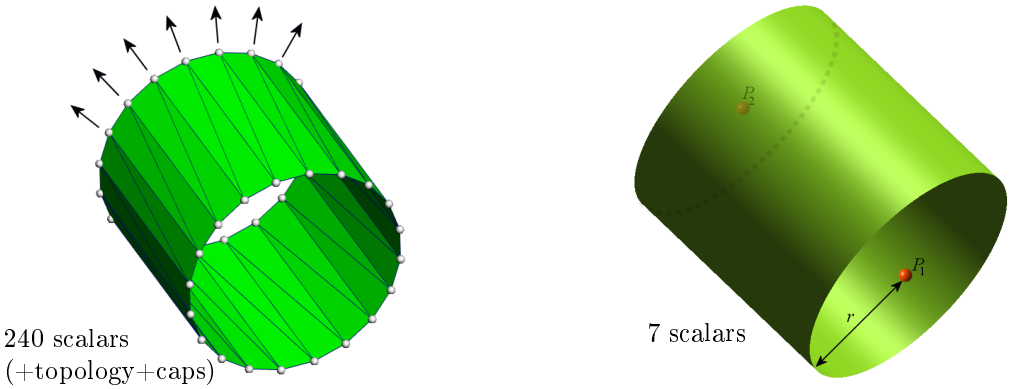


Figure 9.8: *Left:* A tessellated cylinder with n vertices is usually stored with $6n$ scalars in floating-point format (3 scalars for position and 3 for normal). So, a cylinder with 20 vertices in each border needs 240 floating-point numbers to store them and some more bytes to store the topological structure and information about the caps. *Right:* On the other hand, a higher-order cylinder representation uses only 7 scalars (two 3D points and the radius) and an optional Boolean for cap information.

Results (Test 2.2)

We have tested the reverse engineering method over the entire PowerPlant. Table 9.2 shows the numbers for each data section.

For each section we have measured:

- *Original Data*: the number of triangles; the number of separated meshes; and the size of a `ply` file that records the data.

- *Reverse Engineering*: in two situations, *deleting* the meshes of recovered primitives (so the unrecovered data can be separately saved as triangle meshes) and *just find*, also applying the reverse engineering without deleting anything.

- *Unrecovered Data*: the same kind of information as in *Original Data*, but computing the number of faces after joining coplanar triangles (done for optimization purposes). The unrecovered data are mainly box walls and floors and very few primitives not found (see *effectiveness* below).

- *Recovered Primitives*: the number of the three basic primitives (cylinder, cone and torus) and the memory space computed as: $4*(7*\text{cylinders}+8*(cones)+10*(tori))$.

There are some specific but relevant information about the sections:

- Sections 1, 2, 15, 18, 19 and 20 were exclusively composed by tubes. As a consequence, they were 100% recovered (actually, section 15 was 99.86%). See Figure 9.9.

- The number of triangles in Section 12 does not match the number available in the UNC homepage from where the PowerPlant was downloaded. There might be an error in our loading algorithm or there is a mistake in their homepage.

- The external part of the chimney in Section 16 was found after a manual cutting of its base. This is the only case of assisted recovering on the PowerPlant.

Section	Original Data			Reverse Engineering		Unrecovered Data			Recovered Primitives			
#	triangles	disjoint meshes	memory (KBytes)	deleting time (sec.)	just find time (sec.)	faces / triangles	disjoint meshes	memory (KBytes)	cylinders	cones	tori	memory (KBytes)
1	3429528	141034	119213	24,03	24,32	0 / 0	0	0	31372	20	26546	1895
2	161568	8317	5955	1,23	1,14	0 / 0	0	0	1795	0	1205	96
3	382062	23553	14406	3,75	3,32	23094 / 41836	933	1125	4061	629	2038	210
4	55178	11475	2675	0,70	0,56	13025 / 22136	7017	1076	771	12	154	27
5	222728	24898	9196	2,09	2,35	14279 / 28629	13016	1691	2589	593	1021	129
6	55300	6881	2410	0,78	0,51	10035 / 18945	3738	790	804	96	80	28
7	98592	33302	5519	1,26	1,14	29191 / 53075	29106	3485	1391	0	2	38
8	152586	12193	6013	1,40	1,23	4733 / 9270	3868	525	1664	30	972	84
9	121862	31108	6253	2,40	1,32	18853 / 35901	7658	1434	2773	0	6	76
10	197120	78437	11472	3,09	2,37	80860 / 161401	76406	9862	636	20	0	18
11	133398	59288	8692	1,95	1,64	58037 / 114352	57686	7185	674	0	0	18
12	360872	24919	13755	3,93	2,86	20313 / 38067	5818	1392	3678	493	2034	195
13	114334	8547	4360	1,15	0,84	4712 / 8616	327	239	1312	185	659	67
14	248432	105381	15748	3,73	3,10	107124 / 212635	103371	13127	507	2	18	14
15	1141240	64921	34777	9,00	8,61	984 / 1632	654	90	13656	0	8183	693
16	365970	149720	22739	5,34	4,26	148737 / 291864	145951	18271	970	2	68	29
17	251184	100016	15638	2,84	2,40	102825 / 203958	96238	12451	901	0	0	24
18	167012	17151	6465	1,43	1,39	0 / 0	0	0	2674	0	858	106
19	2650680	95688	89245	18,48	18,76	0 / 0	0	0	22433	6	19607	1379
20	2415976	111767	86556	17,51	17,26	0 / 0	0	0	22940	24	18908	1366
21	17356	5313	986	0,26	0,05	4417 / 8702	4413	58	300	0	0	8
Σ	12742978	1113909	482073	106,35	99,43	641219 / 1251019	556200	72801	117901	2112	82359	6507

Table 9.2: Reverse engineering over the power plant sections.

Effectiveness

From the sum row (Σ) in Table 9.2 we can observe that 90% of the PowerPlant was recovered as cylinders, cones and tori $\left(\frac{\text{(unrecovered triangles)}}{\text{(original triangles)}} \approx 10\%\right)$. In Figure 9.9

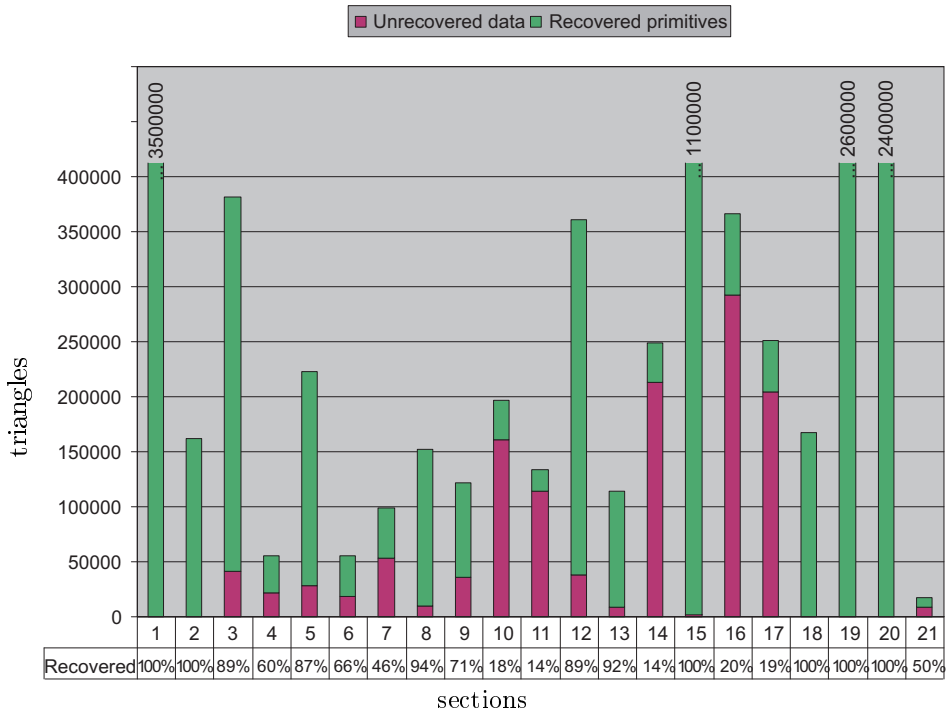
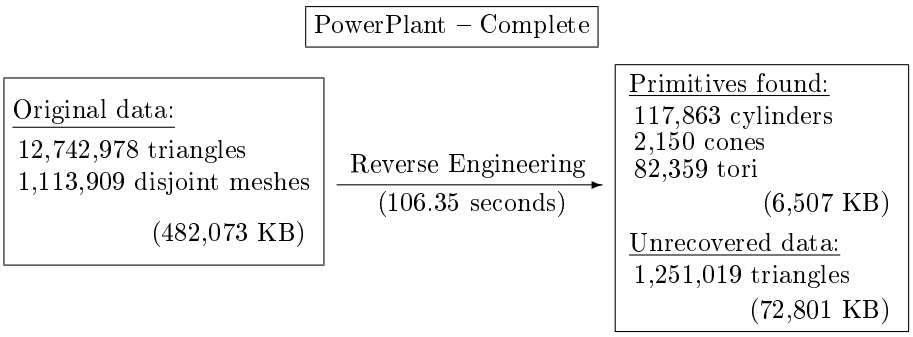


Figure 9.9: The chart displays the percentage of successfully recovered primitives. The objects not recognized by our algorithm are mainly unrecoverable primitives (except for a few primitives shown in Figure 9.10).

we list the recovery percentages for each of the 21 sections.

The following diagram summarizes the PowerPlant conversion:



Most of the unrecovered data were objects that cannot be directly described by implicit surfaces (for example, boxes and walls). However, our algorithm missed some surfaces that should be described implicitly. Figure 9.10 shows the most common cases. In our implementation, if a tube contains one of the objects in Figure 9.10 the whole pipe is not recovered.

Efficiency

The times shown in Table 9.2 in column *Reverse Engineering* were taken using an AMD Athlon(TM) XP 3800+ 2.41GHz with 2GB memory. Before executing the reverse engineering, the data was already in memory, thus we did not consider the time of loading and topology creation. We have measured the execution time in two different situations:

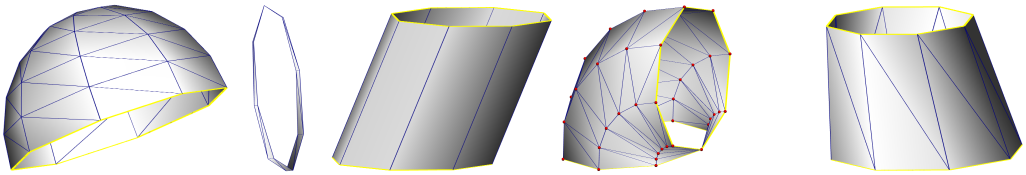


Figure 9.10: Some of the missing data in our recovery algorithm. *From left to right*: half-sphere; very thin torus section; sheared cylinder section; torus section with a singular vertex (this happens when the torus radii have the same value, it is called *horn torus*); and sheared cone.

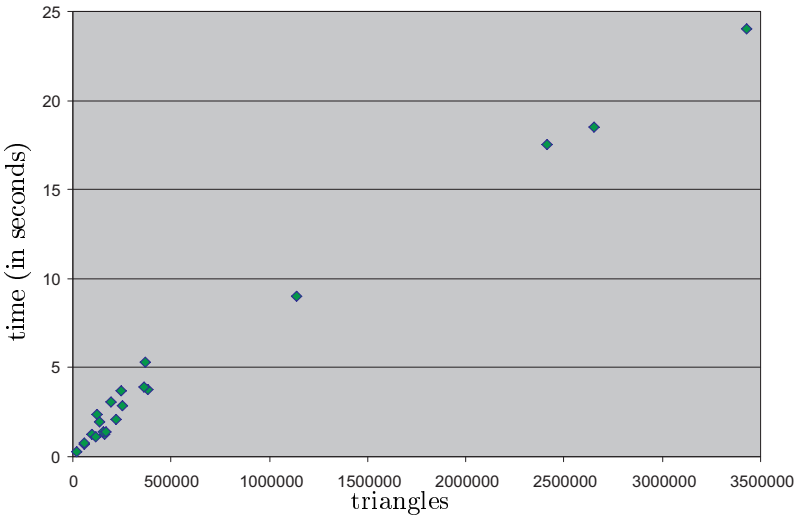


Figure 9.11: Performance for recovering the 21 sections of the PowerPlant. The algorithm is linearly proportional to the number of triangles.

deleting and not deleting the recovered meshes. By deleting them, we are able to save the unrecovered data after execution.

In Figure 9.11 we plot the relation (triangles×time) of all 21 sections (the *deleting* column in Table 9.2). As expected, the recovery algorithm is linearly proportional to the number of triangles.

We were able to process more than 100 thousands triangles per second:

$$\frac{12,742,978\triangle}{106.35s} \approx 120,000\triangle/s.$$

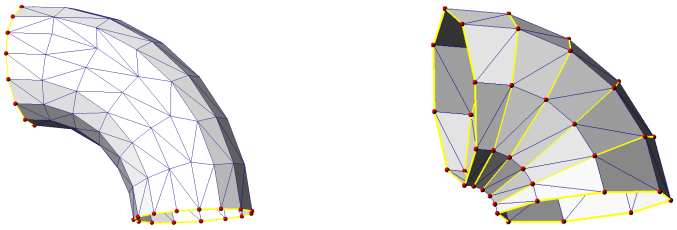
Memory Reduction

The meshes that were completely converted to primitives represent 11,491,959 triangles in 409,272 KB. After recovering them to higher-order primitives they were reduced to 6,507 KB. The reduction factor is 98.41%.

$$1 - \frac{409,272}{6,507} = 0.9841$$

However, if we consider that 1,251,019 triangles were not converted and they still use 72,801 KB of memory, the reduction factor becomes 83.54%.

$$1 - \frac{482,073}{79,308} = 0.8354$$

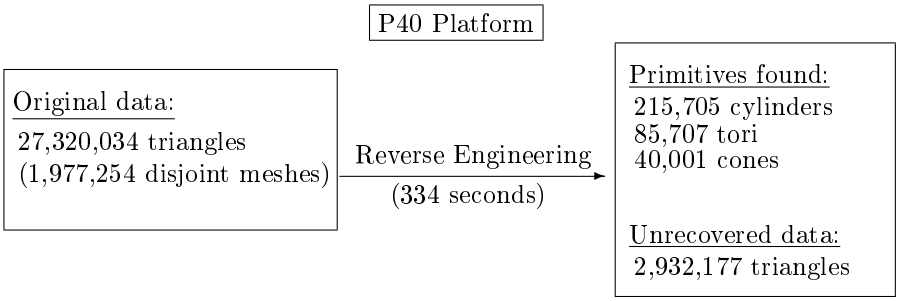


Petrobras P40 PowerPlant

Figure 9.12: *Left:* Meshes in Petrobras data are not necesseraly regular between consecutive rings. In this example only the vertices on the border lie on rings. *Right:* In PowerPlant data, there is always at least one edge connecting consecutive rings.

Results (Test 2.3)

In this test we have applied the reverse engineering algorithm to one of the Petrobras' platforms: P40. The results are summarized in the following scheme:



In this example, the recovery process was also done with our topological approach, which finds and classifies consecutive rings in the meshes. However, the Petrobras data was not as regular as in PowerPlant. Between each pair of rings, an irregular mesh can occur (see Figure 9.12), while in the PowerPlant data the rings were always in sequence without any topological gap. As a result, the recovering algorithm was slower for the Petrobras data.

We were able to process about 80 thousands of triangles per second:

$$\frac{27,320,034\triangle}{334s} \approx 82,000\triangle/s.$$

With PowerPlant data, this rate was $120,000\triangle/s$.

As in the PowerPlant case, we have recovered about 90% of the P40 triangles, identifying cylinders, cones and tori $\left(\frac{\text{(unrecovered triangles)}}{\text{(original triangles)}} \approx 10\%\right)$.

In Chapter 11, we will show the gain obtained in rendering time with such significant reduction of triangles.

Topological Approach Conclusion

The tests done with the topological method have demonstrated its efficiency. The positive points of topological method are the simplicity and the execution speed. The ring-traversal implementation is simple and robust for regular CAD meshes. Only a few minutes were necessary to entirely recover the high-order primitives in the PowerPlant data (13 millions of triangles), considering the model already loaded in memory.

However, there are some limitations in the topological approach. This method only extracts three types of higher-order primitives: cylinders, truncated cones and tori.

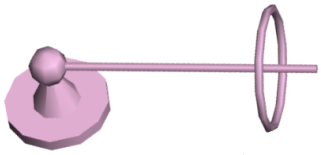


Figure 9.13: This model portion is composed by two cylinders, one complete torus, one complete cone and one sphere. The topological reverse engineering can find the cylinders and the torus (actually, two half tori since we search for torus sections with up to 180 degrees). In order to detect complete cones and spheres the numerical fitting approach is required.

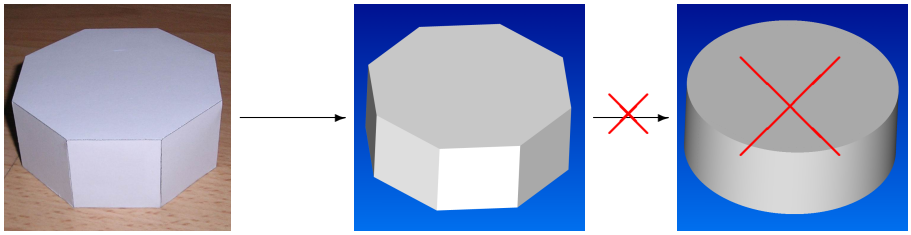


Figure 9.14: An octagonal piece (*left*) should be represented by an octagonal mesh (*center*). Ideally, the recovery algorithm should not substitute the tessellated model for a perfect cylinder (*right*), but it cannot distinguish the situation without extra input information. This is an intrinsic issue when recovering CAD data.

Whereas its application is interesting for tubular structures and pipes, it cannot be applied in other kinds of structures also found in industrial CAD models. For example, there are other implicit surfaces, such as spheres and complete cones (see Figure 9.13), although not as numerous as tubes and pipes.

Usually these extra primitives, as in Figure 9.13, are isolated meshes inside the industrial CAD model. Based on this evidence, we propose the use of the numerical approach explained in Section 9.3, which can recover the quadric surfaces already segmented inside the model.

Another problem related to CAD mesh recovery is ambiguity. Without any additional information, we cannot ensure that a faceted piece is an approximation of a round one or an exact representation of the original piece. For example, in Figure 9.14, the octagonal piece on the left should not be represented by a perfect cylinder, but by an octagonal mesh. This is an intrinsic issue when recovering CAD models. In a perfect scenario, round implicit surfaces, such as cylinders, should not have the tessellated representation as the unique information, but unfortunately this is often the case.

A very important point in recovering CAD data is how to separate the higher-order recovered primitives from the rest. As shown in our results, for models with mixed data (pipes and boxes, for example), we were able to split both types of primitives (implicit surfaces and triangle meshes) without losing information.

In the next section we show how to extend the OpenGL graphics pipeline to efficiently display higher-order primitives (plus the remaining triangles).

Chapter 10

Higher-Order Primitives Visualization

In our solution, we display special graphic primitives that make direct use of their equations. We visualize some implicit surfaces by implementing the code directly in the graphics cards. For this reason we call them *GPU primitives*. This approach improves quality, speed and memory reduction for complex industrial environment visualization.

In Section 7.1 of the previous part, we have presented the principles of the GPU primitives. For natural objects rendering we have used the geometry textures, which describes geometry by an explicit height-map representation. For the manufactured objects rendering, we use implicit surfaces implemented as GPU primitives.

In this chapter we explain the details about the implementation of each implicit GPU primitive (Sections 10.1 to 10.7). In Chapter 11, we show the results for industrial plants visualization.

10.1 Generic Quadrics

The GPU primitive for generic quadrics uses 8 vertices defining the 3D bounding box around the quadric surface (RCA_{Bbox}). The 10 coefficients defining the quadric equation (see Equation 4.1) are passed as parameters to the vertex shader. The local coordinates of the vertices are $[\pm 1, \pm 1, \pm 1]$, see Figure 10.1. We took the decision to fix the local coordinate system since the quadric coefficients can be adapted to include any linear operation (translation, rotation or scale). In our implementation, we consider that the world coordinates are also provided by the application.

The pixel shader must solve the quadric-ray intersection. If there is no intersection or the intersection occurs out of the domain $[-1, 1]^3$ the pixel is discarded.

We have seen that the implicit equation of a quadric (Equation 4.1) contains 10 coefficients. The implicit equation can be rewritten in a matrix form, where the 10 coefficients are placed in a 4×4 symmetric matrix $\mathbf{Q}$:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0 \tag{10.1}$$

Substituting Equation (4.5) into (4.1), the intersection of a ray and a quadric is

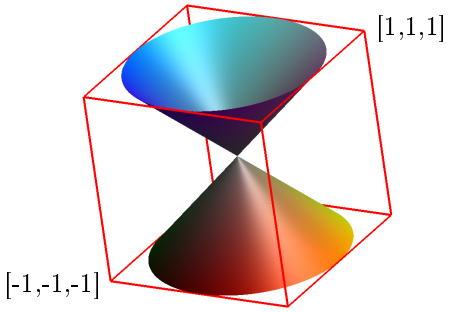


Figure 10.1: Bounding box around the quadric to trigger the pixel shader. The local coordinates domain is $[-1,1]^3$.

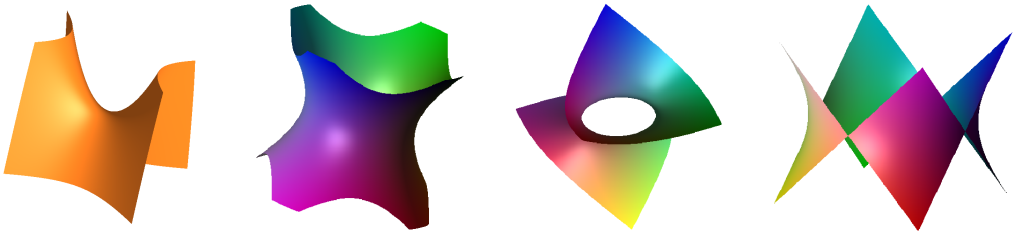


Figure 10.2: Some quadric surfaces rendered by using our GPU quadric primitive. Note that, in some of the quadrics, we apply a color according to the intersection coordinates (mapping (x,y,z) in (r,g,b)).

obtained by solving:

$$V\mathbf{Q}V\,t^2 \;+\; 2\,O\mathbf{Q}V\,t \;+\; O\mathbf{Q}O = 0 \tag{10.2}$$

where V is the vector: $[\vec{v}_x, \vec{v}_y, \vec{v}_z, 0]$ and
O is the vector: $[o_x, o_y, o_z, 1]$ and
$\mathbf{Q}$ is the quadric coefficient matrix

So, to compute the intersection, one just needs to solve a quadratic equation in t, which involves a square root:

$$t = \frac{-b \pm \sqrt{b^2 - ac}}{a} \tag{10.3}$$
$$\text{where} \quad a = V\mathbf{Q}V,\; b = O\mathbf{Q}V\,,\; c = O\mathbf{Q}O$$

(Remark that we use a reduced quadratic equation, by removing the "2" factor from the b parameter. So, $b = O\mathbf{Q}V$.)

The sign of Δ ($\Delta = b^2 - ac$) determines one of the three possibilities: no intersection ($\Delta < 0$), one intersection ($\Delta = 0$), or two intersections ($\Delta > 0$). In the last case, the smallest positive root is chosen. Using the ray and normal equations (Equations 4.5 and 4.3), the intersection P and the normal vector $\vec{n}$ are computed by:

$$P = o + t\vec{v} \tag{10.4}$$
$$\vec{n} = \mathbf{R}P \tag{10.5}$$
$$\text{where} \quad \mathbf{R} \quad \text{is the } [3 \times 3] \text{ upper-left submatrix of } \mathbf{Q}.$$

In the following lines, we present a piece of code to find the two intersections of a quadric and the normal at the closest one, using the CG language [MGAK03]. This

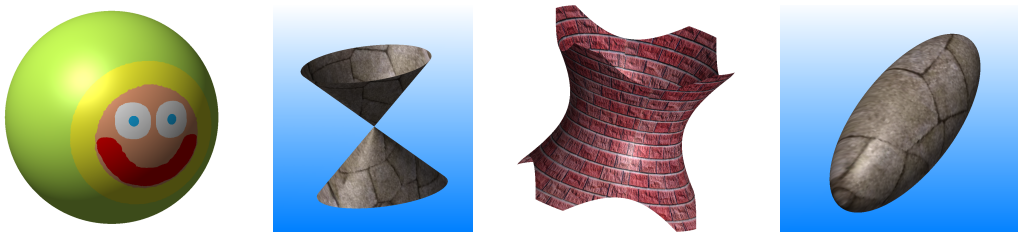


Figure 10.3: Texture mapping using cylindrical coordinates in GPU quadric primitives.

language has powerful instructions such as matrix-vector multiplication and vectors dot product and it can be compiled to run on the GPU.

```
float a, b, c, delta, t1, t2;
float4 o, v;
float4x4 Q;
a = dot (v, mul(Q,v));
b = dot (o, mul(Q,v));
c = dot (o, mul(Q,o));
delta = b*b - a*c;
if (delta<0)  discard;
delta = sqrt(delta);
t1 = ( -b - delta ) / a;
t2 = ( -b + delta ) / a;
t = (t1>=0) ? t1 : t2;
intersection = o + t*v;
normal = normalize(mul(Q,intersection));
```

Finally, using the normal and the lighting direction, we apply the Phong illumination model. Some quadric surfaces are shown in Figure 10.2.

Performance

We achieved 96 fps for rendering a single GPU quadric primitive fulfilling a 512×512 window in a NVidia Quadro FX 3000 graphics card. More results are presented in Chapter 11.

Texture mapping

Texture mapping on implicit surfaces is not as straightforward as on parametric surfaces because there is no direct relation between the 3D and 2D spaces ($[x, y, z] \rightarrow [u, v]$). Despite of that, we implemented some examples with texture using cylindrical mapping.

For the cylindrical mapping, we use the local coordinates $[x, y]$ in the domain $[-1, 1]^2$ as a directional vector whose polar coordinates θ maps the texture coordinate u. And the local coordinate z maps the texture coordinate v. Thus, we apply a transformation from $[0 \cdots 2\pi, -1 \cdots 1]$ to $[0 \cdots 1, 0 \cdots 1]$. The mapping is computed in the pixel shader, which is also responsible for fetching the texture value.

Figure 10.3 shows some results of texture mapping in our GPU quadric primitive.

Self shadow

Still in the same pixel code, it is possible to compute self-shadow for the found intersection point. The ray R' with origin in P and direction $\vec{l}$:

$$R' : (x, y, z) = P + t'\vec{l}$$

is the one used to cast the shadow. If there is any intersection in the direction of the light, the point is in shadow. Note that there is no need to compute the exact intersection point, just verify if $\Delta \geq 0$.

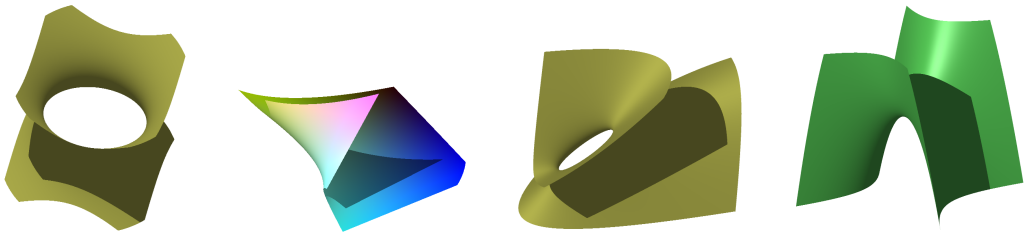


Figure 10.4: GPU quadric primitive with self-shadowing.

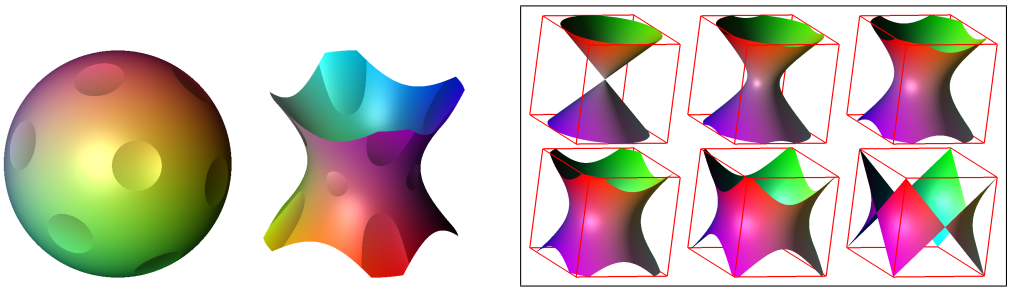


Figure 10.5: (*Left:*) Two GPU quadrics with normal perturbation effect. (*Right:*) Morphing between different quadrics executed in real time (110 FPS using *NVidia Quadro FX 3000*).

Figure 10.4 contains some quadrics with self-shadow test.

It is also possible to produce shadow between different primitives in the scene, by using the conventional *shadow map* technique [Wil78]. Shadow map is a z-buffer based algorithm. The idea is, in a first pass, to render the scene with respect to the light position and direction, saving the z-buffer contents (the shadow map). In the final pass, for each fragment, its location is compared to the shadow map. In case its distance is farther away from the light source than the correspondent value in the map, then it is in shadow. Our GPU primitives are completely compatible with shadow maps because we always update the z-buffer. It means that, with this technique, our primitives can produce shadows over themselves and over the basic triangle primitives (and vice-versa).

Special effects

Before finishing this section about generic quadrics, we want to show two other tests we have done with our quadrics.

The first one is a normal perturbation we have implemented inside the pixel code. As a result, the quadrics seems to have some bumps and depressions, see Figure 10.5(*left*). The goal of this test is to demonstrate that our GPU primitives are compatible with other special effects and that their implementation can be combined inside the same pixel shader, executed with just one pass.

The second test was an animation that executes a morphing between two different quadrics. In this case, the pixel shader needs an extra input about the second set of parameters (10 coefficients) defining a second quadric Q_2, plus a time variable. Using the time stamp to map a real value m between 0 and 1, a morphing quadric is defined by: $Q_m = mQ + (1-m)Q_2$. The ray casting is done over Q_m. An example of result is shown in Figure 10.5(*right*), which can be rendered in about 100 fps.

10.2 Sphere

We have tested two different classes of *RCA* to ray cast spheres: $RCA_{polyhedron}$ and $RCA_{billboard}$. For the $RCA_{polyhedron}$, we have used both a cube and an icosahedron to

RCA	Cube	Icosahedron	Point
V_{RCA}	8	12	1
min *Waste*	21.5%	13.3%	21.5%
max FPS	295	318	295
max *Waste*	39.6%	19.4%	21.5%
min FPS	190	297	295

Table 10.1: Results for different *RCA* used for spheres (see Figure 10.6). To compute these values we fixed $\mathcal{A}_{S_{obj}} = \pi r^2$ as the projection of a sphere with radius r. The minimal and maximal Point *Waste* and the minimal Cube *Waste* are computed using the square enclosing it ($\mathcal{A}_{square} = 4r^2$). The hexagonal projection area of a Cube is used for the maximal *Waste* computation. Finally, the Icosahedron *Waste* was computed by experimentation. In our measuring, we use a sphere, whose projection has 400 pixels diameter, centered in a 1024^2 viewport in a *NVidia Quadro FX 3000*.

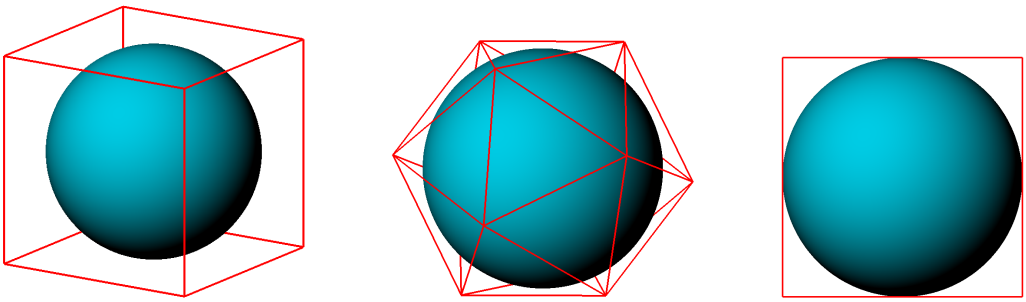


Figure 10.6: Different RCA used for spheres: cube, icosahedron and a scalable point.

perform some tests. We decided to use the RCA_{point} as our billboard, since it has a very low vertex cost if compared to the polyhedron RCA. Table 10.2 shows a comparison between the different RCA.

The spheres are perfectly enclosed in the polyhedrons or in the square resulting from the RCA_{point}.

The ray casting executed inside the polyhedron uses a local coordinate system (the vertex shader is responsible for transforming coordinates). So, in this system, the sphere is zero centered with unit radius. Our goal is to compute the intersection between a ray $R : (x, y, z) = o + t\vec{v}$ and a sphere S given by

$$S: \quad x^2 + y^2 + z^2 = 1 \tag{10.6}$$

Since $\vec{v}$ is normalized, solving for t yields

$$t^2 + 2(\vec{o} \cdot \vec{v})t + (\vec{o} \cdot \vec{o}) - 1 = 0 \tag{10.7}$$

where $\vec{o} = o - (0, 0, 0)$.

The only computations required to find t are

$$\begin{aligned} b &= (\vec{o} \cdot \vec{v}) \\ c &= (\vec{o} \cdot \vec{o}) - 1 \\ t &= -b - \sqrt{b^2 - c} \end{aligned}$$

If $(b^2 - c) < 0$, there is no intersection and the pixel is discarded. In case there is an intersection, we are only interested in the first one (that is why we only use $-\sqrt{b^2 - c}$

and do not consider $+\sqrt{b^2 - c}$). The intersection point can then be easily computed by $o + t\vec{v}$. Once the sphere has unit radius and it is zero-centered (in primitive coordinates), the intersection point coordinates has exactly the same value as the normalized normal.

Using the powerful GPU assembly instructions, as DP3 (Dot Product between 3D vectors) and MAD (Multiply and Add over 1 to 4 floats at once), we solve both the sphere intersection and shading with only 11 ARBFP instructions (see Appendix A for more information about GPU programming languages). The vectors `v` and `l` are the normalized representations of the viewing and light directions, while `o` is the origin of the ray.

```
DP3  oo, o, o;            #oo = dot (o,o)
DP3  ov, o, v;            #ov = dot (o,v)
SUB  oo, oo,1;
MAD  delta, ov, ov, -oo;
KIL  delta;               #discard if delta < 0
RSQ  delta, delta;
RCP  delta, delta;        #sqrt(delta)
ADD  t, -delta, -ov;      #find t of intersection
MAD  normal, t, v, o;     #normal=inters= o + t*v
DP3  diffuse, normal, l;  #diffuse = dot(normal,l)
MUL  out, diffuse, color; #SHADE without specular
```

Actually, for shading with specular and ambient components, four extra instructions are needed, and one instruction is also required for a correct z-buffer value output (totaling 16 assembly instructions for an orthographic projection).

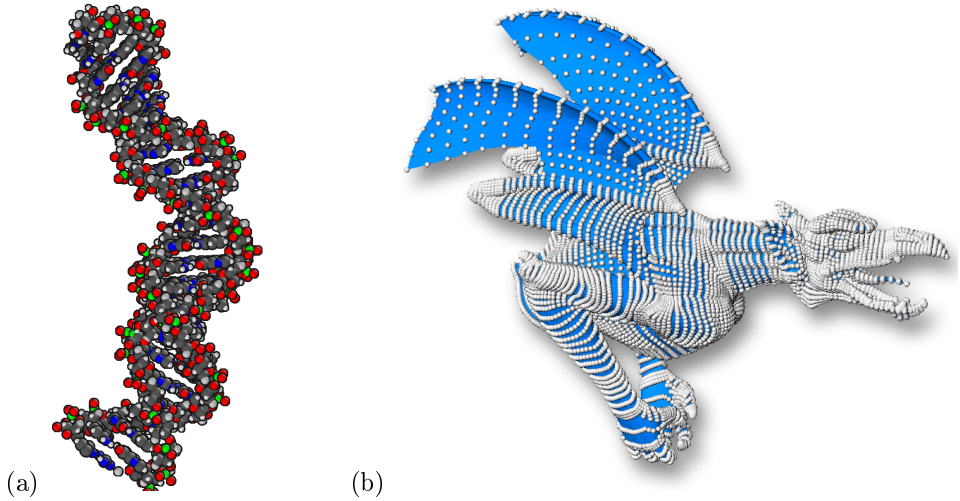
We are able to draw a sphere with a diameter of 1024 pixels, inscribed in a viewport of 1024^2 at 65 FPS, by using the cube as the RCA. All the one million pixels are running the sphere intersection algorithm. Smaller spheres increase the rate. For instance, a sphere with diameter of 512 pixels inside the same viewport (1024^2) is rendered at 220 FPS.

Sphere in $\mathbf{RCA_{point}}$. An efficient RCA for spheres is the GL_POINT (with OpenGL library), which actually covers a square on the screen. The size of this square can be specified either by using GL_POINT_SIZE or by changing the vertex point size in the vertex shader (by enabling GL_VERTEX_POINT_SIZE_ARB). Some optimizations can be done in the case of orthographic projections. All computations reduce to 2D operations in screen-space (including the update of the z-value). This approach is also useful for drawing glyphs (i.e., symbols with zoom-independent sizes) with correct z-buffer. See Figure 10.7(b).

10.3 Ellipsoid

We have also dedicated some effort in the implementation of a fast ellipsoid. We adopted the use of a RCA_{box} to enclose the ellipsoid. Inside the box, the primitive coordinates are set in a way to distort the ellipsoid that gets a sphere form. Thus, the pixel shader code is very close to the sphere one. The CPU delivers the following information for each ellipsoid:

- the ellipsoid center;
- three orthonormal vectors, defining the ellipsoid main axes;
- each vertex primitive coordinate $(\pm 1, \pm 1, \pm 1)$;
- the proportion between axes.



(a) (b)

Figure 10.7: (a) A DNA using sphere GPU primitive in a RCA_{point}, a special effect is used for enhancing the sphere borders. (b) Each vertex of the object is rendered as a small sphere (the glyphs) also using the RCA_{point}.

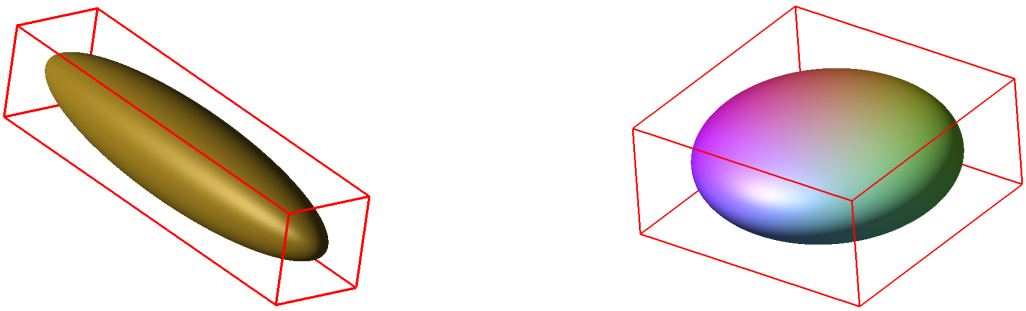


Figure 10.8: Ellipsoids inside their RCA_{box} are ray-casted as distorted spheres.

By using this information the vertices can compute their own world coordinates to correctly position themselves in the scene (see Figure 10.8).

Compared to the spheres, the ellipsoid pixel shader has little extra work in computing the surface normal (which was almost free in the sphere computation, see Section 10.2). The extra work consists in adjusting the normal according to the proportion between axes, followed by a normalization, totaling five extra ARBFP instructions.

In order to try some performance test, we developed a system for tensor of curvature visualization [CSM03] computed over the vertices of a triangular mesh. Each tensor is represented by an ellipsoid that uses a local coordinate system formed by the vertex normal direction and the main curvature directions. So, a thin ellipsoid indicate that the mean curvature in the main direction is much stronger than in the second main direction, while a round ellipsoid means similar curvature in all directions. The ellipsoid sizes have also a meaning, the size is to small where the mesh is almost flat, while bigger sizes are seen in significant curvature locations. We also use the colors to give some extra information. They represent the main curvature direction $(x, y, z) \rightarrow$(red,green,blue). See Figure 10.9 for the results.

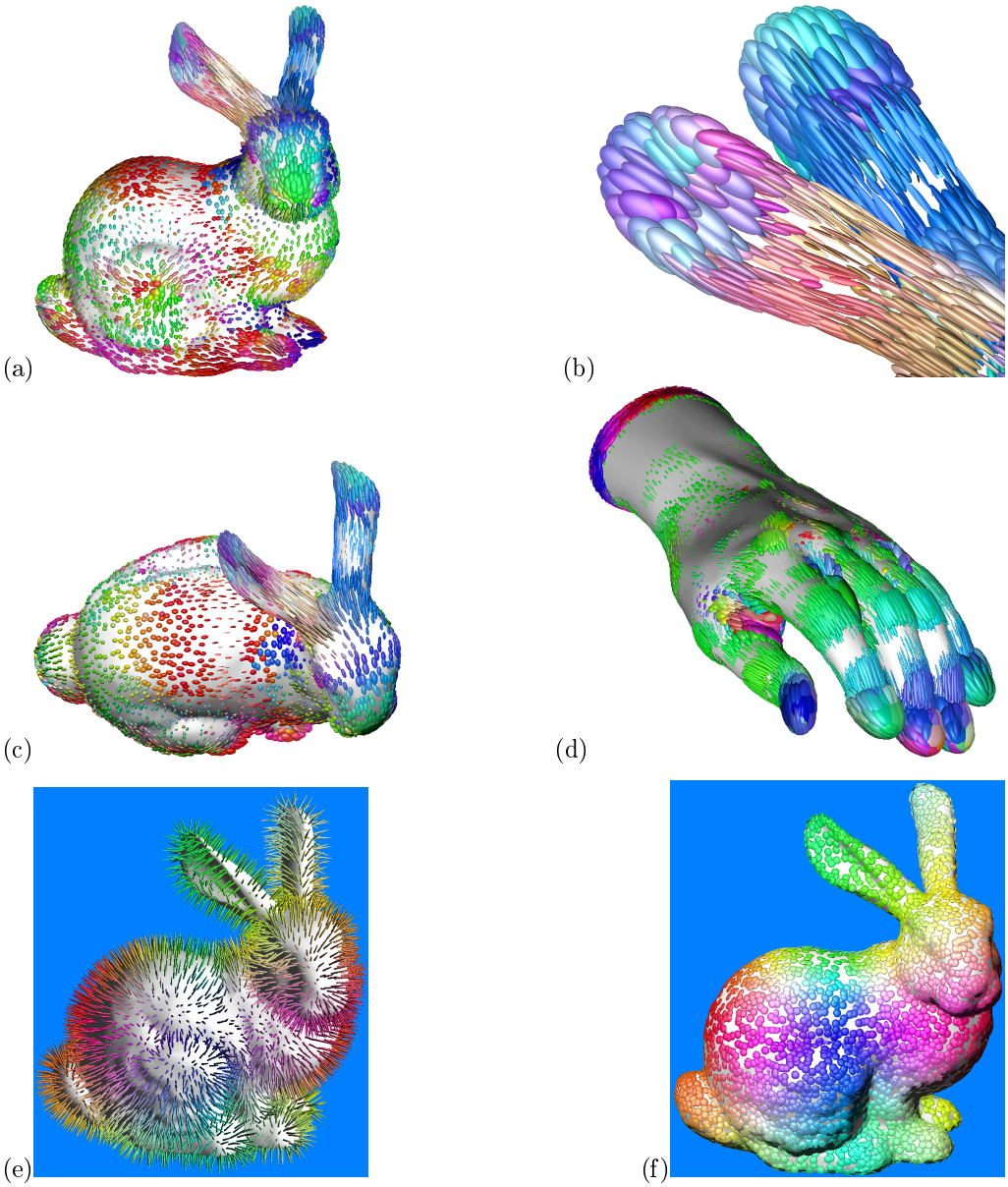


Figure 10.9: Tensor of curvature displayed using ellipsoids GPU primitives. The color represents the main curvature direction (red is x, green is y and blue is z). (b,c,d) 5000 ellipsoids GPU primitives, (d) 6,500 ellipsoids over the handle model, at 30 FPS in a 1024^2 viewport (using *NVIDIA Quadro FX 3000* graphics card). Notice that the ellipsoids are perfectly combined with the triangle mesh, which uses the standard rasterization of the graphics card. (e,f) In both images, there is an ellipsoid for each vertex of the bunny mesh, and one of the ellipsoid axes coincides with the normal direction. In (e) this axis is the longest one, resulting in this thorn appearance, in (f) it is the shortest one, resulting in a *M&M* appearance.

10.4 Cylinder

For the cylinders a four-sided billboard is the best RCA choice. It has a very low vertex cost, $V(RCA_{Billboard}) = 4$, and, with the correct computation, it has a very tight projection enclosing the cylinder projection (see Figure 10.10), what reduces the *Waste* function (see Section 7.1.4).

The vertex shader computes the vertices position based on the cylinder dimensions, in

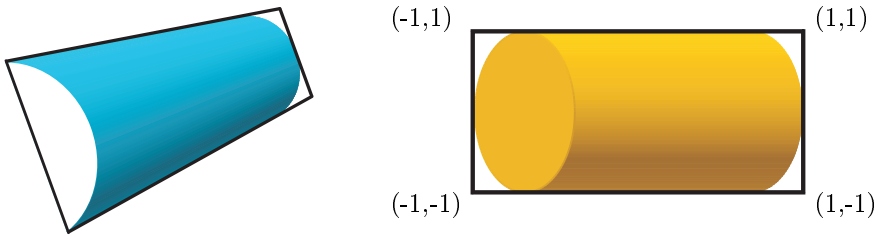


Figure 10.10: The GPU cylinder rendered with the rasterization of a billboard. (a) Rendered in perspective, (b) capped cylinder (rendered with the covers) in orthographic projection.

a way that the 4 vertices always form a perfect convex hull (Figure 10.10). Since the world position of the vertices is computed in the shader, the CPU does not need to pass the world coordinates. Similarly, the local coordinate system can be deduced directly from the cylinder's main direction and the computed convex hull axes directions, also reducing the information needed to be sent by the application. This is a kind of *view-dependent* coordinate system, where one of the axes is fixed relatively to world (the cylinder main axis) and the other axes depend on the viewing direction. Finally, the unique per-vertex information required is the relative position in the convex hull (front/back and left/right).

For each cylinder the application only need to pass the following information to the GPU:

- cylinder main direction (3 floats),
- radius (1 float),
- beginning point (3 floats),
- four 2D vertices, each one is just a pair of integers containing -1 or 1 for front/back and left/right, {(-1,-1),(-1,1),(1,-1),(1,1)}

In practice, to accelerate rendering, our OpenGL implementation uses two texture coordinates to pass the 7 floats information and a `glRecti` with the 4 vertices. So, for each cylinder, the CPU-GPU communication is only 7 floats and 4 integers.

10.4.1 Orthographic Projection

In orthographic projection, often used for CAD modeling, it is easy to implement a *capped cylinder* primitive. A small adjust in the vertex shader corrects the position of the front vertices to also pack the cap, and the RCA is still a rectangle in the screen (see Figure10.10(b)), while in the pixel shader a simple test detects if the pixel is in the cap. So, in the same primitive we describe an object whose tessellated counterpart would contain three triangulations (one for the cylinder body, and two for the caps).

A cylinder can be defined by two points (C_0 and C_1) and a radius r.

The cylinder main axis links both points ($C_1 - C_0$).

Our cylinders are composed by a body, a top and a bottom. The two last ones are the circular caps closing the cylinder (Figure 10.11a). Let us assume that the bottom is always visible while the top is invisible (except when the cylinder is completely perpendicular to the viewing direction when both are invisible).

View-dependent Coordinate System Generally, in a cylinder local coordinate system, the $\vec{z}$ coincides with the cylinder main axis (see Figure 10.11b) while $\vec{x}$ and $\vec{y}$ are in

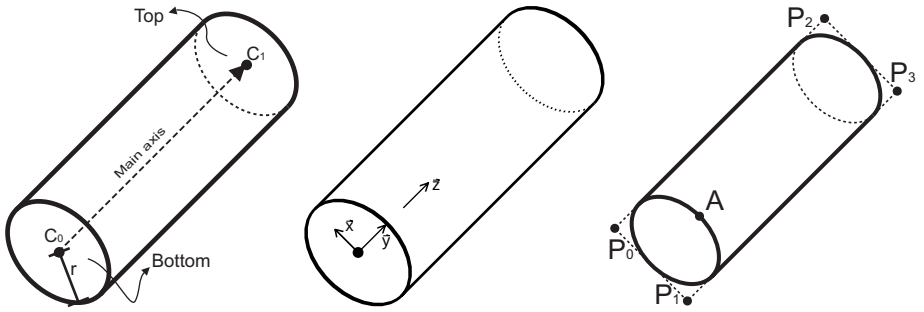


Figure 10.11: (a) Cylinder definition. (b) View-dependent Coordinate System. (c) Billboard points (P_0, P_1, P_2, P_3).

a plane perpendicular to $\vec{z}$, respecting the right-hand rule. In our *View-dependent Coordinate System*, we impose one more restriction, $\vec{x}$ must be perpendicular to the viewing direction $\vec{v}$.

$$\begin{aligned} \vec{z} &= \frac{(C_1 - C_0)}{|\,(C_1 - C_0)\,|} \\ \vec{x} &= \vec{v} \times \vec{z} \\ \vec{y} &= \vec{z} \times \vec{x} \end{aligned}$$

The vector $\vec{v}$ is a normalized vector pointing to the observer (as if it was perpendicular to the sheet of paper for the reader in Figure 10.11b). Note that $\vec{y}$ and $\vec{z}$ are aligned when projected to the screen, although perpendicular in $\mathbb{R}^3$.

Billboard Cylinder Using this coordinate system, we are able to find the four points (P_1, P_2, P_3, P_4) that will compose the billboard (see Figure 10.11c).

$$\begin{aligned} &P_0 = C_0 + \vec{x'} - \vec{y'} && P_1 = C_0 - \vec{x'} - \vec{y'} \\ &P_2 = C_1 + \vec{x'} + \vec{y'} && P_3 = C_1 - \vec{x'} + \vec{y'} \\ &\text{where } \vec{x'} = r \cdot \vec{x}, \quad \text{and} && \vec{y'} = r \cdot \vec{y} \end{aligned}$$

Those points would be a subset of the bounding box points constructed by the same coordinate system. Therefore, the rectangle formed by P_0, P_1, P_3, P_2 is a diagonal plane inside this virtual bounding box.

In Appendix B, we propose an alternative to ray casting in our pixel code for cylinders in orthographic projection. Basically, by dividing the interior of the billboard into regions, we can apply very simple instructions to exactly reproduce the cylinder in terms of shading and depth. As a result, the gpu-cylinder rendering is so optimized that it is even faster than the default pixel shader for triangles. In perspective these optimizations cannot be implemented but, as we will see in the next subsection, we also managed to achieve good performance for our GPU cylinder.

10.4.2 Perspective Projection

In perspective projection we also use a billboard whose four vertices are computed based on the view-dependent coordinate system (see Subsection 10.4.1). However, due to the

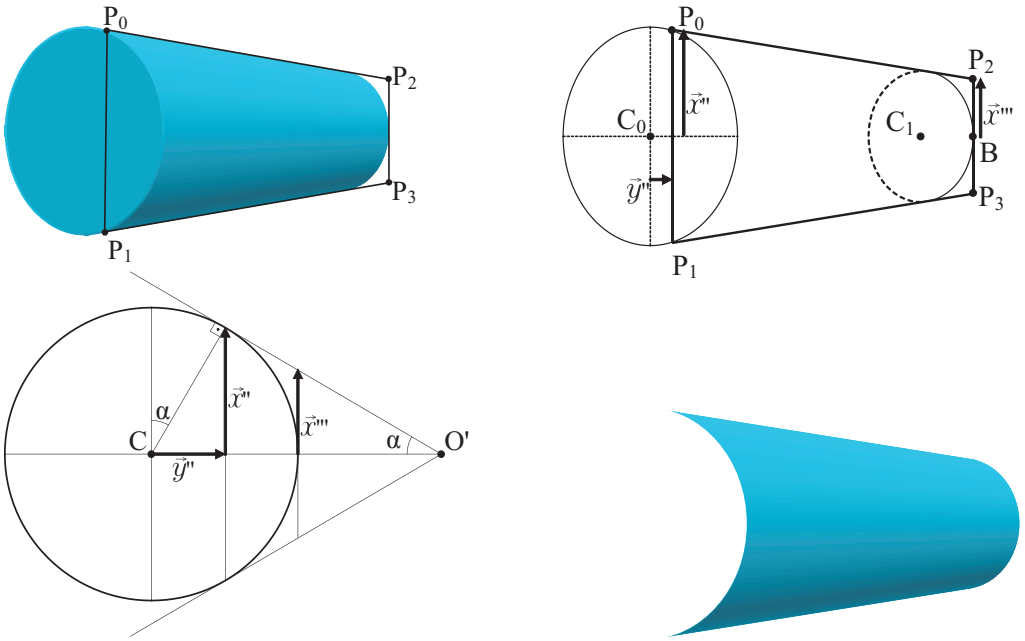


Figure 10.12: *Top:* Billboard in perspective completely covering the cylinder body. *Bottom left:* We reduce to $\mathbb{R}^2$ the problem of finding the vectors $\vec{x''}, \vec{y''}$ and $\vec{x'''}$. The cylinder and the observer position O are projected into the plane defined by the cylinder main axis (O' is the projected observer). The center points C_1 and C_2 are projected to the same point C. We can find the vertices position based on the cylinder radius r, the distance $|O'C|$ and the unit vectors $\vec{x}$ and $\vec{y}$ from the view-dependent coordinate system (see equations in text). *Bottom right:* The final image.

perspective distortion, the vectors $\vec{x'}$ and $\vec{y'}$ are slightly different. Another difference between perspective and orthographic projections is that the cylinder cap cannot be easily added to the cylinder body with the billboard solution.

To compute the final position of the vertices P_0, P_1, P_2, P_3, we must consider the viewer position. Based on Figure 10.12, we can deduce the computation of their positions as follows:

$$
\begin{aligned}
P_0 &= C_0 + \vec{y''} + \vec{x''} & P_1 &= C_0 + \vec{y''} - \vec{x''} \\
P_2 &= C_1 + \vec{y'} + \vec{x'''} & P_3 &= C_1 + \vec{y'} - \vec{x'''} \\
&\text{where} & \vec{x''} &= \vec{x} \cdot r \cdot \cos\alpha, \\
& & \vec{y''} &= \vec{y} \cdot r \cdot \sin\alpha, \\
& & \vec{x'''} &= \vec{x} \cdot (|O'C| - r) \cdot \tan\alpha, \text{ and} \\
& & \alpha &= \arcsin\left(\frac{r}{|O'C|}\right).
\end{aligned}
$$

Covering billboard cylinders in perspective

Putting caps to close cylinders is not an easy task for billboard cylinders in perspective. In orthographic projection (see Section 10.4.1) it is enough to extend the billboard in directions $\vec{P_2P_0}$ and $\vec{P_3P_1}$ (see Figure 10.11), which are actually parallel to each other. However, in perspective, these directions are divergent and extending them would result in a RCA with an expensive pixel cost (see Section 7.1.4). Besides that, it would be

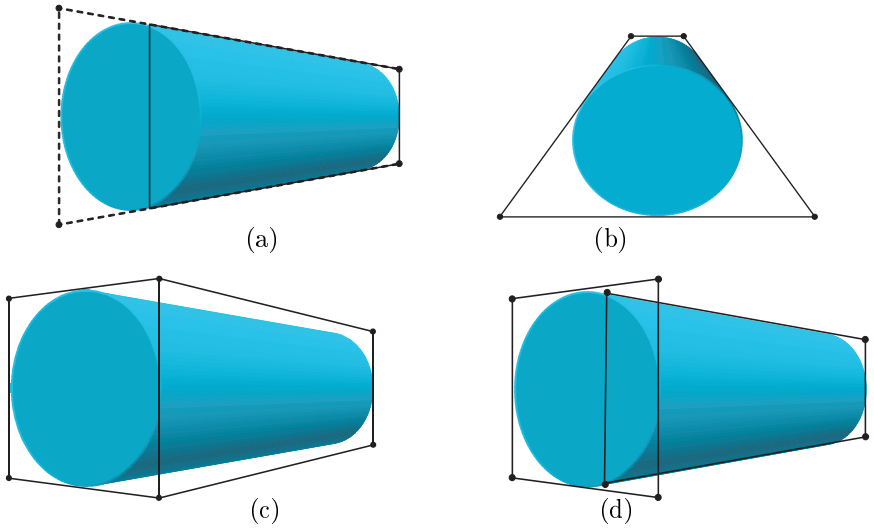


Figure 10.13: Three different ways to render the GPU cylinder with caps in perspective. (a) Extending the billboard used for the body, (b) however, it may result in a significant waste of pixels for some viewing angles. (c) Two faces and six vertices of a bounding box to produce cylinder with caps. (d) Two distinguished billboards, one for the cylinder body and another one for the cylinder cap.

hard to compute the vertices positions in some special situations, for example, when the cylinder main axis has almost the same direction of the viewing vector.

Besides the billboard extension, there are two possibilities for rendering the cylinder caps: either abandoning the billboard and returning to the 3D bounding box solution; or using an *independent* primitive for the caps. For the first case, it would be enough to use only two faces of the bounding-box with 6 vertices computing their position by using the view-dependent coordinate system (see Figure 10.13a). For the last case, we can use another billboard that show the cap turned to the viewer (see Figure 10.13b).

In our implementation we have chosen to use another billboard for rendering the caps (as in Figure 10.13b). The vertex shader of this GPU primitive uses the same parameters of the cylinder body (cylinder main direction, radius and beginning point). It also computes the cylinder view-dependent coordinate system for positioning the four vertices. Notice that only one primitive is used to render both caps. This is possible because the two caps are not visible together in the same frame. Based on the main-axis and viewing directions the cap is positioned either in the *bottom* or in the *top* of the cylinder.

The pixel shader of this GPU cap is very simple. The pixel is discarded if $(x^2+y^2) > 1$ (the domain inside this billbaord is $[-1,1]^2$). The shading uses the normal, which is either the main axis or its opposite, depending if it is the bottom or the top.

The advantage of using another billboard for the cylinder cap is the simplicity of its pixel shader. However, using the six-vertices bounding box, the vertex cost would be reduced: $V(cylinder_{Bbox}) = 6 < V(cylinder_{Bboards}) = 8$.

10.4.3 Thickness Control

A special feature developed for the GPU cylinder is the *thickness control*. In the vertex shader we impose a minimum of 1 pixel large for the billboard. In the vertex shader we guarantee that the vectors $\vec{x'}$ (for orthographic projection), $\vec{x''}$ and $\vec{x'''}$ (for perspective) have a minimum size of 1 pixel in the screen. This avoids the dashed line aspect of thin cylinders rendering, what may happen when one rasterizes the cylinder triangular mesh, if the viewer is far away.

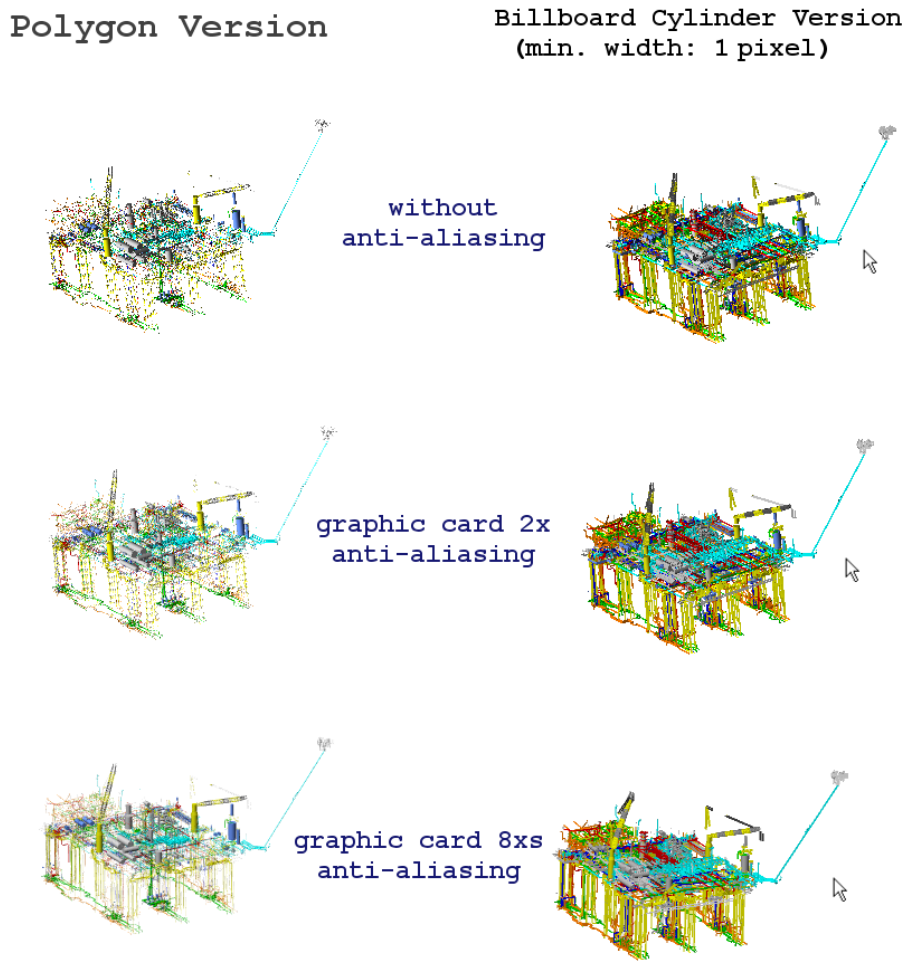


Figure 10.14: In the *left column* the cylinders are rendered as polygon meshes. In the right column, we use our cylinder GPU primitive on a $RCA_{billboard}$ with thickness control (minimum of 1 pixel). The mouse cursor gives an idea of zoom level. We compare the results with three different hardware anti-aliasing setup. On the top, no anti-aliasing. On the middle, simple $2x$ anti-aliasing. On the bottom, the most sophisticated option in our GeForce 6800 graphics card: $8xs$ anti-aliasing. The results show that our thickness control improves quality in all situations, but it is especially better when the anti-aliasing is turned off.

To uniform the color along the one-pixel thin billboard, the normal used in the illumination equation is the average one of the cylinder visible body. To produce a smooth transition, when the billboard width is between 1 and 5 pixels, the normal is computed as a smooth interpolation between the average normal and the normal computed at each pixel. The thickness control is the key for the zoom out quality enhancement for a uniform group of thin cylinders (see Figure 10.14), reducing aliasing problems.

10.5 Truncated and Complete Cones

The *RCA* used for cones is a polyhedron, whose front faces are rasterized, so the pixel shader turns in the pixels around the final image. The truncated cones use a bounding box with vertex cost $V(RCA_{Bbox}) = 8$ (see Figure 10.15(*left*)). For complete cones, a pyramid is the warping polyhedron, which has smaller vertex cost, $V(RCA_{Pyramid}) = 5$, than bounding boxes (see Figure 10.15(*right*)).

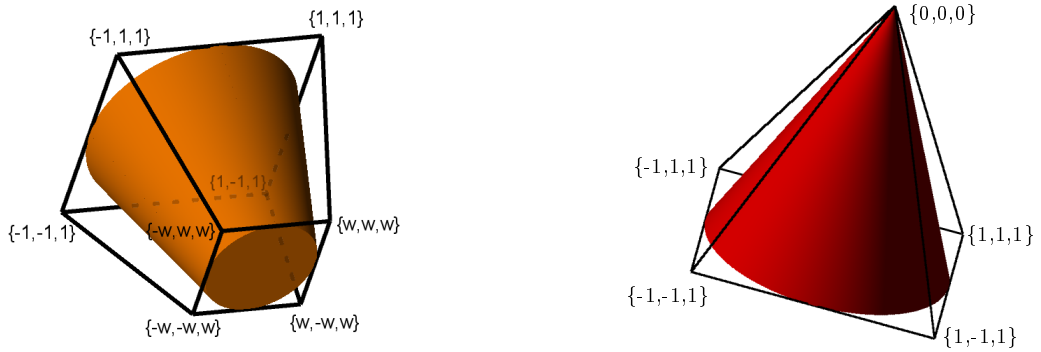


Figure 10.15: Truncated Cone. Primitive coordinate system of each vertex, where w is the proportion between the smaller radius and bigger one.

As shown in Figure 10.15, each vertex has a local coordinate system delivered to the pixel shaders. Thus, the ray casting algorithm searches the intersection with a canonical cone, whose apex is $\{0,0,0\}$, height is equal to 1 and base radius is also equal to 1. For truncated cones, the base radius is the bigger one, while the smaller one is w.

Although with different RCA, both types of cones use the same vertex and pixel shaders. The ray casting inside the pixel code is very close to the sphere one. The equations to solve are:

$$(\vec{v} \cdot \vec{v'})t^2 + 2(\vec{v} \cdot \vec{o'})t + (\vec{o} \cdot \vec{o'}) = 0 \tag{10.8}$$

$$t = \frac{-b \pm \sqrt{b^2 - ac}}{a} \tag{10.9}$$

where

$$a = \vec{v} \cdot \vec{v'},\ b = \vec{v} \cdot \vec{o'},\ c = \vec{o} \cdot \vec{o'},$$
$$\vec{v'} = [v_x, v_y, -v_z],$$
$$\vec{o'} = [o_x, o_y, -o_z]$$

Figure 10.16 describes how the information flows through shaders, starting by the application.

The ARBFP code contains only about 25 instructions.

Optimizations

We want to point out two interesting implementation details we have done for optimization.

- For complete cones, the pyramid used as a polyhedron can be passed to the graphics card as a triangle strip, see Figure 10.17. This kind of optimization is important for large number of primitives in the scene, which is the case of industrial plants visualization. The benefits of using triangle strip instead of triangle sequence are fewer vertex transformations and CPU-GPU communication reduction. Note that if the application bottleneck is in the pixel processing, no performance gain is expected from this optimization.

- In the vertex shader we use an eye-dependent coordinate system, as we have done with the cylinders (see Section 10.4). The objective is to reduce the *Waste* function, by keeping the RCA with a more appropriate set of pixels. See Figure 10.18.

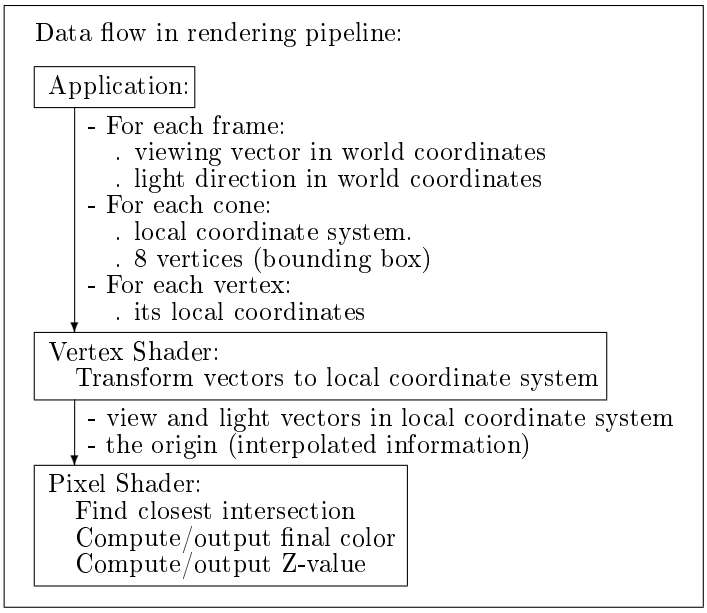


Figure 10.16: Pipeline data flow when rendering a *cone GPU primitive*.

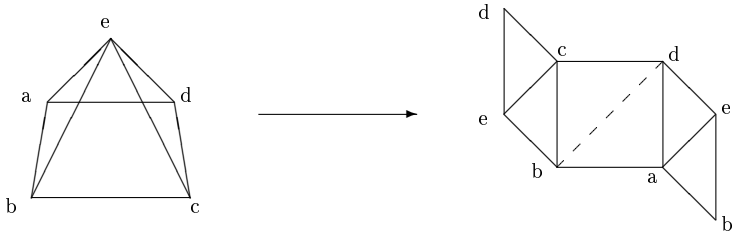


Figure 10.17: Striping the pyramid RCA for optimization, it is especially important for applications with large number of primitives, such as industrial plants visualization.

10.6 Cubic

We extended the idea of GPU primitives, which we had developed for quadrics, for cubic implicit surfaces. It is possible to form interesting surfaces by using cubic equations (see Figure 10.19), but these forms are not as popular as the classical quadrics (e.g., sphere, cylinders, cones). Comparing to quadrics, the computation is much more intense, resulting in fewer frames per second. Actually, the main point in trying cubic gpu primitives is to get a step closer to the torus, which has a quartic equation.

The difficult in implementing the cubic primitive is the intersection searching process. It includes a root find problem of third degree. There are many polynomial root find algorithms known in the literature [PTVF92] that could work in the cubic case. We choose to use a binary search based on the *Sturm theorem*. The binary search seems a good solution in our case, since we have a restricted domain (we use a RCA_{Bbox} with local coordinates in $[-1,1]^3$, as in the generic quadrics in Section 10.1).

Sturm Theorem

The theorem is based on a set of functions, known as *Sturm functions*, which are derivate from the base function $f(x)$:

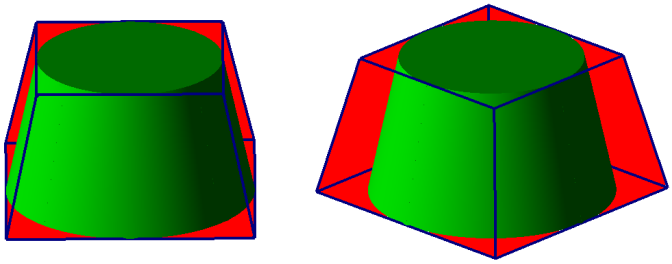


Figure 10.18: (*Left*) With the eye-dependent coordinate system, the bounding box is always perpendicular to the screen, (*Right*) while without it, it may happen an oblique projection. As a result, the number of wasting pixels (in red) is reduced in average with the eye-dependent coordinate system.

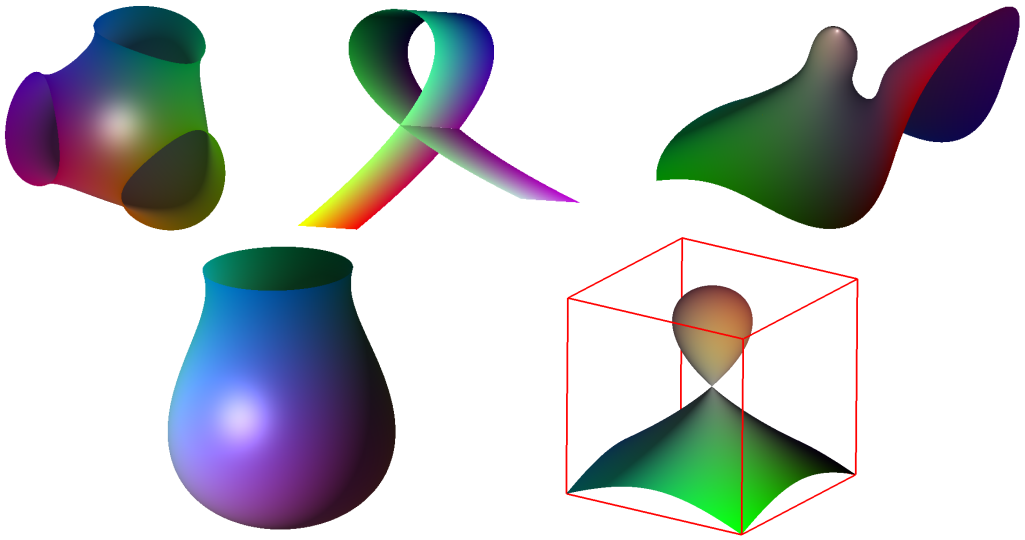


Figure 10.19: Cubic surface examples. They are all clipped outside the domain $[-1,1]^3$, which is represented by a cube in the last image.

$$\begin{aligned} f_0(x) &= f(x) \\ f_1(x) &= f'(x) \\ f_n(x) &= -\left\{ f_{n-2}(x) - f_{n-1}(x) \left[\frac{f_{n-2}(x)}{f_{n-1}(x)} \right] \right\}, \quad n \geq 2 \end{aligned}$$

So, for a cubic function, there are four Sturm functions we need to generate (the last one, f_3, is a constant).

With the Sturm functions it is possible to find the number of real roots of an algebraic equation over an interval. After evaluating the set of functions for the two points defining the interval, the difference in the number of sign changes between them is going to give the number of roots in the interval.

Algorithm

Based on Sturm theorem, we can do a binary search to find the first intersection between the ray $R : (x, y, z) = o + t\vec{v}$ and the cubic surface. The initial interval is between $t = 0$ and $t = \lambda$, where the point $P_2 = o + \lambda\vec{v}$ is the point where the ray leaves the domain

$[-1,1]^3$. At each iteration, the interval is divided into two. The following algorithm describes the procedure:

```
Compute the terms of functions f_0, f_1, f_2, f_3 for a given Cubic surface C
Find λ
t_1 = 0
t_2 = λ
n_1 = number_of_sign_changes (t_1)
n_2 = number_of_sign_changes (t_2)
if (abs(n_2 - n_1) < 1)
  discard
t_m = t_2 / 2
LOOP n times
{
  n_2 = number_of_sign_changes (t_m)
  if (abs(n_2 - n_1) < 1)
    t_2 = t_m
  else
    t_1 = t_m
  t_m = (t_1 + t_2) / 2
}
intersection = o + t_m v
```

Remark that, since we search only for the first intersection, we can use the number of sign changes (n_1) of the origin point ($t = 0$) in all iterations. So, we just recompute the number of sign changes for the searching point (t_m). This is an important issue, since the computation of sign changes is very expensive, involving the evaluation of four functions (actually three, since one of them is a constant).

Results

Sturm algorithm is not as high performance as quadric ray casting. We achieved 30 fps in a GeForce6800 NVidia graphics card. The quality of the results is very good (see Figure 10.19). However, in some special situations (e.g., close to singular points), the computation exceeds the precision of the GPU and some errors may appear, which can be especially evident after zooming.

10.7 Torus

For the torus GPU primitive we use a well fitted bounding box as our RCA_{Bbox}. As we have seen in Section 4.1.2, a torus can be described by a quartic implicit function:

$$T(x,y,z) = (x^2 + y^2 + z^2 - (r^2 + R^2))^2 - 4R^2(r^2 - z^2) = 0 \qquad (10.10)$$

The above equation defines a zero centered torus with main direction in z. This canonical position is the one used by our pixel shader to ray cast the torus (with one more definition: $R + r = 1$).

Given a ray $\vec{R} : (x,y,z) = o + t\vec{v}$, where $\vec{v} = [v_x, v_y, v_z]$ is a normalized vector, the ray-torus intersection equation [Han89] [19] is:

[19] The equation in Hanrahan document [Han89] was not totally perfect, Eric Haines lately suggested a correction (see: `http://steve.hollasch.net/cgindex/render/raytorus.html`).

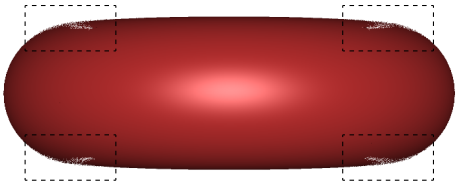


Figure 10.20: Due to floating-point imprecision, Sturm method can result in bad images for torus. The problem is more evident in the viewing-angle shown in this picture.

$$at^4 + bt^3 + ct^2 + dt + e = 0 \tag{10.11}$$

where
$$a = 1$$
$$b = 4(\vec{o} \cdot \vec{v})$$
$$c = 2((\vec{o} \cdot \vec{o}) - (R^2 + r^2) + 2(\vec{o} \cdot \vec{v})^2 + 2R^2(v_z)^2)$$
$$d = 4((\vec{o} \cdot \vec{v})((\vec{o} \cdot \vec{o}) - (R^2 + r^2)) + 2R^2 v_z o_z)$$
$$e = ((\vec{o} \cdot \vec{o}) - (R^2 + r^2))^2 - 4R^2(r^2 - o_z^2)$$

Finding the root of a quartic equation for several pixels in interactive rates is not an easy task. We have tried four different approaches, comparing their results to choose an adequate algorithm for our GPU torus. They are described in the following subsections.

We have given special attention to the GPU torus because it is a very common primitive in pipe sequences (a quarter of torus is typically used for junctions in an elbow format, see Section 10.8).

10.7.1 Sturm

We have extended the algorithm used in our cubic GPU primitive (see Section 10.6). Compared to ray-cubic intersection, ray-torus intersection has an extra computation since there is one more function to be evaluated in each iteration, totaling five functions. As a result, Sturm is not so fast for solving the torus-ray intersection (see Table 10.2). Another problem with the Sturm approach is the numerical precision. The complexity of terms on each Sturm function may overflow the floating-point capacity. This problem is viewing-angle dependent and in some cases may produce incorrect images (see Figure 10.20).

10.7.2 Double Derivative Bisection

Thesis contribution: Double Derivative Bisection
We have proposed and investigated a new root finder that is an extension of simple bisection algorithm.

This technique is also a binary search, as in Sturm technique. The idea is an extension of *Bisection method*, a root-find algorithm over a given interval where is known to have one root. To explain our double derivative bisection we start by reviewing the simple bisection and how to extend it for using derivatives.

Simple Bisection Given two points t_0 and t_1, where $f(t_0)$ and $f(t_1)$ have different signs, we can ensure that there is at least one root (where $f(x)$ is a continuous function). Using the interval's midpoint $t_m = 0.5(t_0 + t_1)$ we evaluate the function, $f(t_m)$. Based on its sign we reduce the interval to be between t_0 and t_m or to be between t_m and t_1, depending on the sign of $f(t_m)$ is the same as in $f(t_1)$ or not. After n iterations the interval is reduced to $\log(n)$ of its original size. The advantage of this method is

its simplicity. In fact, it is always successful. However, we need to start by getting two points with opposite signs at the beginning.

Derivative Bisection As we have just seen, the bisection algorithm depends on $f(t_0)$ and $f(t_1)$ having different signs. By using *derivative bisection*, we extend the bisection algorithm for working on two other special situations:

- both $f(t_0)$ and $f(t_1)$ with positive signs but with one and only one local minimum in the interval; and
- both $f(t_0)$ and $f(t_1)$ with negative signs but with one and only one local maximum in the interval.

In these circumstances we can guarantee that, if there is an intersection (actually, up to two intersections), the *Derivative Bisection* algorithm can find it (them).

The algorithm will do a binary search for the local minimum (maximum), but if the searching point (t_m) crosses the abscissa (in other words, $f(t_m)$ changes the sign if compared to the previous one), the algorithm switches for a classical bisection search. The following pseudocode does the derivative bisection for the first above mentioned circumstance ($f(t_0)$ and $f(t_1)$ are positives):

```
bool Is_there_a_root (f(x), f'(x), t_0, t_1, &first_root, &second_root)
{
  if (f'(t_0) >= 0)
    return false;   //no local minimum
  if (f'(t_1) <= 0)
    return false;   //no local minimum
  do forever
  {
    t_m = 0.5 (t_0 + t_1);
    if (f(t_m) < 0)
    {
      first_root = Bisection(f(x),t_0,t_m)
      second_root = Bisection(f(x),t_m,t_1)
      return true;
    }
    if (f(t_m) == 0)
    {
      if (f'(t_m) == 0)
      {
        //the local minimum is a root (the function only touches the abscissa)
        first_root = second_root = t_m
        return true;
      }
      if (f'(t_m) < 0)
      {
        first_root = t_m
        second_root = Bisection(f(x), t_m+Δ, t_1)
        return true;
      }
      else
      {
        second_root = t_m
        first_root = Bisection(f(x), t_0, t_m-Δ)
        return true;
      }
    }
    if (f'(t_m) == 0)
      return false; //we reach the local minimum and it is still positive
    if (f'(t_m) < 0)
      t_0 = t_m
    else
      t_1 = t_m
  }
}
```

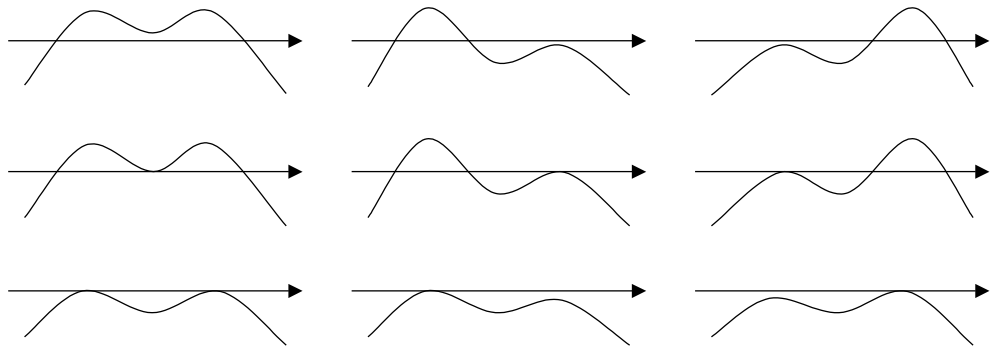
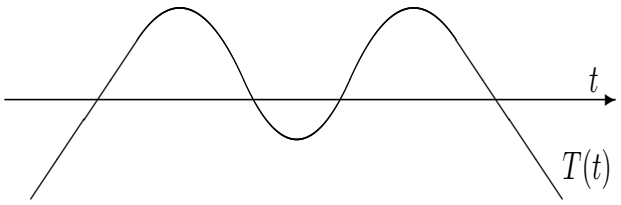


Figure 10.21: All possible plots for the function $T(t)$ where there is at least one root (or one ray-torus intersection), except the four roots case.

This algorithm must work if one of the two above-described circumstances is set up at the beginning. It is almost as robust as the traditional bisection algorithm, except that there is a comparison to zero, which needs a threshold due to numerical precision. In the following text, we show how the derivative bisection can be useful for the ray-torus intersection search.

Double Derivative Bisection for Torus We have seen that a torus can be described by a quartic implicit function (Equation 10.10) and the ray-torus intersection involves the solution of an equation of degree four (Equation 10.11), which yields a maximum of four possible intersections. One can easily imagine a ray traversing a torus and crossing its boundary four times. If we plot the evaluation of the torus intersection function $T()$ for this four-times crossing ray parameterized by t, we will obtain something close to the following form:



where T has at most two local maximum and at most one local minimum (see all the possible graphics for T in Figure 10.21).

The basic idea in the *Double Derivative Bisection* is to divide the problem into two, running the single derivative bisection twice, first on the portion before the local minimum and then on the portion right after (the local minimum is represented by $\triangle_2$ in Figure 10.22).

However, finding exactly the local minimum includes the solution of the derivative equation (of the original Equation 10.11), which is a third-degree equation:

$$T'(t) = 4at^3 + 3bt^2 + 2ct + d = 0 \tag{10.12}$$

This computation is expensive and we should avoid it. So, instead of solving the above equation, we approximate the local minimum location by a much simpler computation. We compute the third-derivative root, which is the average of the three first-derivative roots (based on Vieta's Formulas):

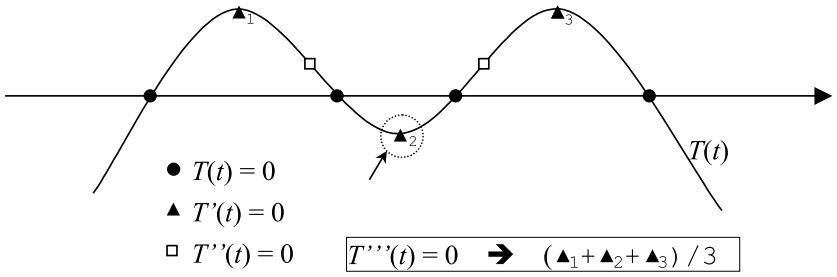


Figure 10.22: The round points mark the roots, the triangles mark the first derivative roots and the squares mark the second derivative root. The third derivative root is actually the average of the first derivative roots (see Equation 10.14). Its location must be somewhere around the second first-derivative root ($\triangle_2$).

$$\begin{aligned} T'(t) &= 4at^3 + 3bt^2 + 2ct + d = (t - \triangle_1)(t - \triangle_2)(t - \triangle_3) \\ & \quad \frac{3b}{4} = -(\triangle_1 + \triangle_2 + \triangle_3) \\ T'''(t) &= 4t + b \\ T'''(t_M) &= 0 \quad \Rightarrow \quad t_M = -\frac{b}{4} = \frac{\triangle_1 + \triangle_2 + \triangle_3}{3} \end{aligned} \tag{10.13}$$

The third-derivative root t_M is not exactly the local minimum we were searching for, but it is a good approximation and can be used as the "division point" in our algorithm (see Figure 10.22).

So, the computation to find the root of the third derivative ($T'''(t_M) = 0$) is very simple since uses only b, whose value is $4(\vec{o} \cdot \vec{v})$ (see Equation 10.11):

$$t_M = -(\vec{o} \cdot \vec{v})$$

Although the third derivative is only an approximation (see Figure 10.22), it is good enough to divide the root finding algorithm into two for applying the derivative bisection (that is why we call our technique: *double derivative bisection*). Notice that we are only searching for the first intersection (the first root). So, we would apply the derivative bisection for the second section only if no root was found in the first one.

With the double derivative bisection we have not obtained faster results than with the Sturm technique. See Table 10.2 for comparison. However, we have got perfect results (without the numerical problems found with Sturm, see Figure 10.20), guaranteeing an error $\epsilon(r) \leq 0.0014$ relative to the minor radius r.

10.7.3 Sphere Tracing

The sphere tracing was proposed by Hart in 1996 [Har96]. The idea is to find the ray-intersection by stepping closer and closer through the ray. Given the Euclidean distance function $d(x)$ to a surface, we can march along the ray from point x the distance $d(x)$ without penetrating the surface.

For a torus of major radius R and minor radius r, centered at the origin and main direction in z (the canonical torus previously mentioned), the distance function is:

$$d(x) = ||(||(x, y)|| - R, z)|| - r$$

Compared to other methods, sphere tracing for ray-torus intersection needs less computation for each iteration. In this case, there is no need to compute derivatives as in

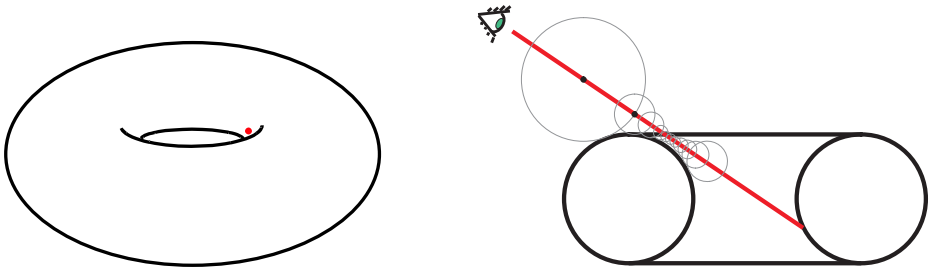


Figure 10.23: Example of critical situation in the sphere-tracing for ray-torus intersection. In this case, the ray almost touches the torus reducing the step size, which is based on the ray-torus distance. As a result, a bigger number of iterations are necessary to find the final intersection.

Sturm, Double-derivative bisection or Newton-Raphson (subsections 10.7.1, 10.7.2 and 10.7.4). However, there are some critical situations whose approximation is very slow, increasing the number of iterations (see Figure 10.23). When the ray comes very close to the surface without intersecting it, the steps become very small demanding multiple iterations to transpose this critical part to finally find the intersection in a further point.

To overcome these critical points, we proceed with two sphere tracing for each ray with different starting points. The first one uses the entering point in the torus bounding box, the second one uses the same "division point" of double derivative bisection: the second third-degree root (t_M). If after some iterations with the first sphere tracing $d(x)$ becomes greater than $d(x-1)$ then we proceed with the second sphere tracing.

With our two-rays implementation of sphere tracing we achieved better performance than the single-ray (see Table 10.2). However, the convergence of this algorithm is still slow. In the next subsection we present the implementation that resulted in the best performance among the four ones we have tried.

10.7.4 Newton-Raphson

Up to this point we have seen three methods of ray-torus intersection that we have implemented to process in the GPU. The *Sturm* implementation proved to be precise, however not fast enough for interactive rendering; *sphere tracing* and *double derivative bisection* are faster than *Sturm* but we decided to try a fourth method: the *Newton-Raphson* root finder. In this subsection we explain the Newton method, concluding that it is the fastest one (among the four ones we have tried).

The Newton-Raphson method (sometimes called the *Newton's method*) also uses the derivative evaluation in each iteration (as in Sturm and in double derivative bisection). The Newton's formula derives from the Taylor series, which is:

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \cdots \tag{10.14}$$

If δ is small enough, we can ignore the high-order terms so, for each iteration, we can move a δ step with:

$$\delta = -\frac{f(x)}{f'(x)}. \tag{10.15}$$

The geometric interpretation of the above equation is that we find the next position x_{i+1} by extending the tangent in $f(x_i)$ until it crosses zero.

Newton's performance. Newton-Raphson algorithm converges quadratically. This means that near a root, the algorithm doubles the significant digits after each step [PTVF92]. However, far from the root, it may have a bad behavior. For example, if

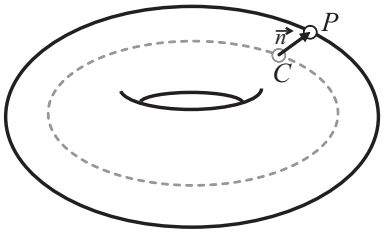


Figure 10.24: The normal of a point P on the torus surface can be computed by using the closest point C in the torus internal ring. Dividing vector $\vec{n}$ by the smaller radius we obtain the normalized normal.

the current position is too close to a local extreme, the derivative is almost zero and the next step will vanish to infinite. For this reason we took some extra cares in our GPU ray-torus intersection implementation.

Implementation. As explained at the beginning of this section, we use a well fit bounding box around the torus from where the ray-casting starts (it is the same bounding box in the four tested algorithms). Therefore, we are not so far from the root, but we can still push forward the starting point. Before applying the Newton iterations, we start by executing a ray-sphere intersection (the sphere radius is the sum of the torus radii). If there is no intersection, we can discard the pixel; if the intersection is negative (before the starting point), we ignore it; and if the intersection is positive, then we move the starting point to this position.

We know that between the starting and the intersection points it is still possible to have a local minimum (or local maximum), see Figure 10.21 on page 182. To avoid a vanishing situation, we use some bounds to guarantee that the step will not be bigger than λ. Empirically, we found that $\lambda = 0.15$ (relative to our canonical torus) efficiently avoids the vanishing cases without loosing performance.

The Newton-Raphson algorithm gave the best performance for our GPU-Torus. We use a threshold to stop the iterations that assures an error smaller than 0.1% of the minor radius r ($\epsilon(r) \leq 0.001$). We have also tested with $\epsilon(r) \leq 0.00003$. The results are shown in table 10.2.

10.7.5 Normal Computation

One possible way to compute the normal vector for a point lying in the torus surface is by taking the three partial derivatives of the torus function at this point. Since the equation is defined by a four-degree polynomial, the operation is quite expensive. Instead of an algebraic solution, we have implemented a geometric one.

Considering the canonical torus with the main direction in z (see Equation 10.10), the normalized normal $\vec{n}$ at the point P lying on the torus is:

$$\vec{n}_{torus} = \frac{P - C}{r} \text{ , where } \quad C = \begin{cases} C_{xy} = \frac{P_{xy}}{|P_{xy}|} \\ C_z = 0 \end{cases} \tag{10.16}$$

The geometric interpretation of Equation 10.16 is explained in Figure 10.24.

Note that we can use part of this computation to define a threshold to discard a pixel. The threshold is relative to the smaller radius r. The discard situation would be:

$$\text{if } (|\| P - C \| - r| > \text{threshold}), \text{ then discard}$$

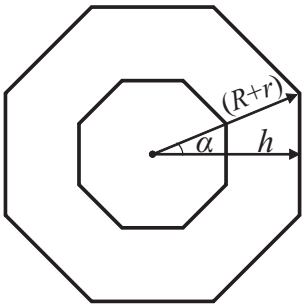
		Projection Angle						Average	Error	
		0°	18°	36°	54°	72°	90°		$\epsilon(r)$	$\epsilon(R+r)$
Pixels	RCA_{Bbox} pixels	524,176	516,986	495,086	430,740	324,622	209,000	416,768		
	Torus pixels (S_{torus})	312,552	305,704	285,932	256,132	203,532	146,672	251,754		
	$Waste(RCA_{Bbox}, S_{torus})$	40.4%	40.9%	42.2%	40.5%	37.3%	29.8%	38.5%		
		FPS								
GPU Primitive	Sturm	68	68	71	82	108	163	93	0.000025	0.000011
	Bisection	15	15	16	19	25	46	23	0.0014	0.0006
	SphereTracing	16	16	16	20	27	46	24	0.001	0.000429
	two-rays SphereTracing	66	66	65	80	100	153	88	0.001	0.000429
	Newton 0.00003	246	254	261	302	380	537	330	0.00003	0.000013
	Newton 0.001	267	283	291	324	410	590	361	0.001	0.000429
Polygonal	Polygon 32*32	2653	2649	2668	2704	2730	2744	2691	0.011236	0.004815
	Polygon 64*64	1238	1238	1238	1238	1238	1238	1238	0.002811	0.001205
	Polygon 128*128	369	369	369	369	369	369	369	0.000703	0.000301
	Polygon 256*256	96	96	96	96	96	96	96	0.000176	0.000075
	Polygon 512*512	24	24	24	24	24	24	24	0.000044	0.000019
		Megapixels/second							StdDev	$\frac{\text{StdDev}}{\text{Average}}$
GPU Primitive	Sturm	35.64	35.16	35.15	35.32	35.06	34.07	35.07	0.485	1.38%
	Bisection	7.86	7.75	7.92	8.18	8.12	9.61	8.24	0.631	7.65%
	SphereTracing	8.39	8.27	7.92	8.61	8.76	9.61	8.60	0.527	6.14%
	two-rays SphereTracing	34.60	34.12	32.18	34.46	32.46	31.98	33.30	1.111	3.34%
	Newton 0,00003	128.95	131.31	129.22	130.08	123.36	112.23	125.86	6.587	5.23%
	Newton 0,001	139.95	146.31	144.07	139.56	133.10	123.31	137.72	7.654	5.56%

Table 10.2: Comparison between several torus rendering techniques. The tests were done using a 1024×1024 viewport with a torus filling the window. We have used a GeForce 7900 graphics card with vsync disable. The rendered torus has radii $r = 0.3$ and $R = 0.7$.

10.7.6 GPU Torus Results

Rendering One Torus

We have done several tests measuring the performance of each one of the four GPU Torus methods: Sturm, Bisection, SphereTracing and Newton. We have considered two different implementations for SphereTracing (one-ray and two-rays) and two implementations for Newton (varying the threshold). The results are presented in Table 10.3, which also contains the performance of traditional polygonal rasterization method. In the following text we pick up some relevant points in this table for deeper discussion:



$$
\begin{aligned}
\epsilon(R+r) &= 1 - \frac{h}{R+r} \\
\epsilon(R+r) &= 1 - \cos(\alpha), \\
&\quad \text{where } \alpha = \frac{\pi}{N} \\
\epsilon(R+r) &= 1 - \cos\left(\frac{\pi}{N}\right)
\end{aligned}
$$

Figure 10.25: Geometrical error for a polygonal $N * N$ torus representation. On the left, an example of polygonal 8*8 torus. On the right, the computation used for filling out the *Error $(R+r)$* column for the polygonal tori in Table 10.2.

Projection Angle. The performance of our GPU Torus is limited by the pixel processing. The number of pixels of the *RCA* projection is determinant for the final frame rate and it varies according to the torus angle and distance. For this reason we have done tests with 6 different viewing-angles (including top and perpendicular viewing), see Figure 10.26.

RCA_{Bbox} **pixels.** Number of pixels produced by the RCA_{Bbox} projection. See first row in Figure 10.26.

Torus pixels. Number of pixels produced by torus projection. See second row in Figure 10.26.

$Waste(RCA_{Bbox}, S_{torus})$**.** Function that measures the pixel cost (see Section 7.1.4).

Polygon $N * N$**.** Polygonal version of torus with N rings and N sides (as *GlutTorus* function definition).

Error. Our GPU torus techniques use an error threshold measured relative to the smaller radius: $\epsilon(r)$. We computed the error of the polygonal tori relative to their total radius: $\epsilon(R+r)$. We can extract from one error the other one by using the radii proportion of our testing torus ($r = 0.3$ and $R = 0.7$). As explained in Figure 10.25, for a polygonal torus $N * N$ the error is: $\epsilon(R+r) = 1 - \cos\left(\frac{\pi}{N}\right)$

Note that the errors considered here are only geometrical ones. Actually, the shading of polygonal torus presents approximation errors that do not happen in the GPU torus methods, which use per-pixel shading computation.

Megapixels/second. It is the corresponding multiplication of FPS and RCA_{Bbox} pixels. This number indicates how many times the ray-casting algorithm was executed (in millions) per second.

StdDev and $\frac{\textbf{StdDev}}{\textbf{Average}}$**.** Some of our ray-casting techniques suffer different per-pixel performance depending on the viewing-angle. To identify this fact we computed the standard deviation of each technique. As a conclusion, Bisection and SphereTracing techniques had the most view-dependent performance. The last one is a consequence of critical situations as shown in Figure 10.23.

FPS. We have measured the frame rate for different angles for each different method, averaging them on the last column. Among all GPU Primitive methods, *Newton 0.001* have presented the best performance. As we will see in the Table 10.3, the performance of our GPU torus becomes interesting for multiple tori. However, we can see that even for individual torus, we can obtain competitive numbers. For an error $\epsilon(R+r) \approx 0.000350$, the polygonal version is slightly better. On the other hand, for an error $\epsilon(R+r) \approx 0.000015$, the GPU Torus is much faster. See this direct comparison in the following table extracted from Table 10.2:

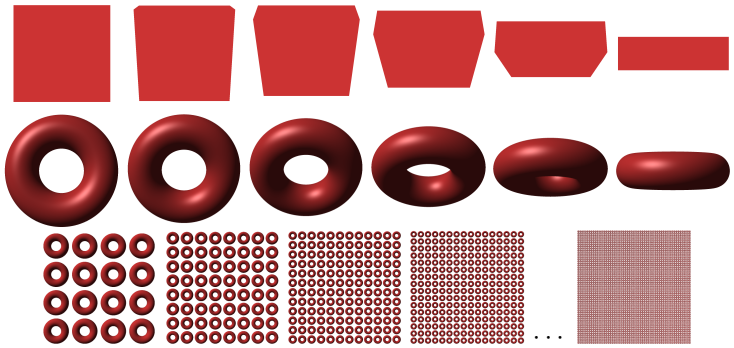


Figure 10.26: Several viewing-angles used for testing torus rendering performance (see Table 10.2). In the first row the RCA_{Bbox} used for our GPU primitive, in the second row the torus itself. The third row shows how we have multiply the number of tori for the results in Table 10.3.

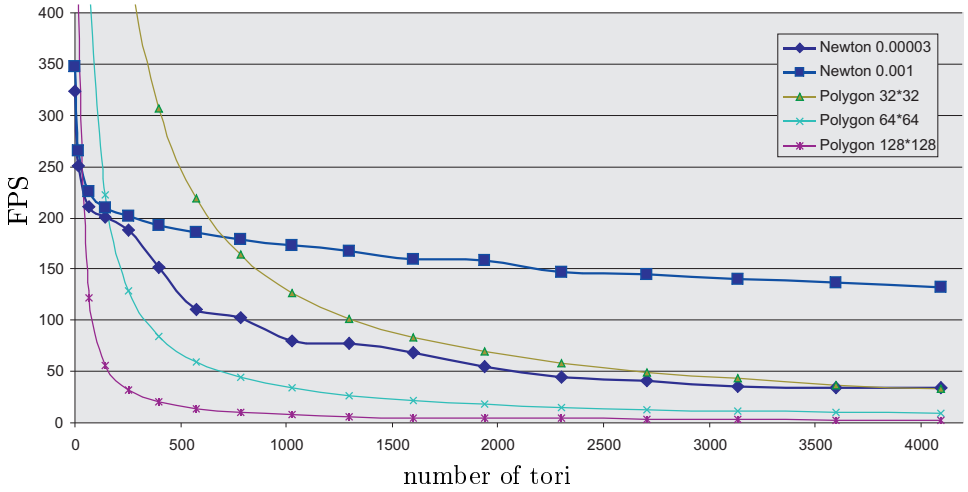


Figure 10.27: Performance of different techniques for multiple tori. This graphic plots the results shown in Table 10.3.

Method	Error $(R+r)$	Average FPS
GPU Newton 0.001	0.000429	361
Polygon 128*128	0.000301	369
GPU Newton 0.00003	0.000013	330
Polygon 512*512	0.000019	24

Rendering Multiple Tori

One advantage of our GPU torus is that the bottleneck is no longer on the vertex stage, but on pixel stage. It means that the performance will not be reduced as much as using the polygonal torus version when increasing the number of tori. In Table 10.3 we compare the GPU Newton technique in two different thresholds with five different resolutions of polygonal torus. We plot the result (except the two slowest polygonal versions) in Figure 10.27.

We can verify that *GPU Newton 0.001* is the fastest for more than 700 tori. Note that if compared to polygonal 128*128 (which has equivalent error as shown in Table 10.2) the Newton method is always faster. Another interesting point is that, for 16000

number of tori	Newton		Polygonal (within displaylists)				
	(0.001)	(0.00003)	(32*32)	(64*64)	(128*128)	(256*256)	(512*512)
1	348	324	▷ 2730	2716	2108	1279	468
16	266	251	▷ 2299	1304	432	131	34
64	226	211	▷ 1268	460	122	34	9
144	210	201	▷ 725	222	56	15	4
256	201	188	▷ 453	129	32	9	2
400	193	151	▷ 306	84	20	6	1
576	186	110	▷ 219	59	14	4	1
784	▷ 179	102	164	44	10	3	< 1
1024	▷ 173	80	127	34	8	2	< 1
1296	▷ 167	78	101	26	6	2	< 1
1600	▷ 160	68	83	22	5	1	< 1
1936	▷ 158	55	69	18	4	1	< 1
2304	▷ 147	45	58	15	4	< 1	< 1
2704	▷ 145	41	49	13	3	< 1	< 1
3136	▷ 140	35	43	11	3	< 1	< 1
3600	▷ 137	34	37	10	2	< 1	< 1
4096	▷ 133	34	33	9	2	< 1	< 1
16384	▷ 50	9	9	2	< 1	< 1	< 1

Table 10.3: Comparison between the techniques for multiple tori. We have varied the number of tori from 1 up to 4096 (plus an extra 16384 test), measuring their performance in *frames per second* (FPS). For each tori set we have applied a continuous rotation to measure the frame rate. The tori are rendered with a display list, always fitting the window screen (1024×1024). As presented in Figure 10.27, the Newton method with error $\epsilon(r) = 0.001$ is the one with best performance for a higher number of tori. The symbol " ▷ " indicates the best performance.

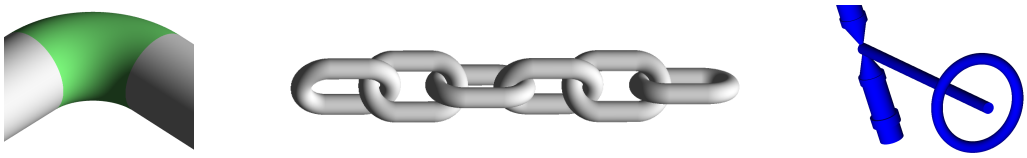


Figure 10.28: Examples of torus in manufactured models.

tori, Newton 0.001 is the only one that keeps an interactive frame rate (50 fps).

10.8 Torus Slice

The previous section about torus has shown the effort we have put for achieving a correct and fast rendering. In manufactured plants, torus appears in junctions, chains and CAD patterns (see Figure 10.28). Actually, in most of the cases, the objects contain only a *slice* of a torus. Although the algorithm for ray-casting the torus slice is the same as the one for a complete torus (except for an angle test control), we suggest a special treatment for determining the RCA for the slice case (see Section 7.1.4 about RCA). In other words, the pixel shader for both complete and partial torus are basically the same, however the vertex shaders have different implementations.

The torus slice uses a bounding box more adapted to its form instead of the parallelepiped one (see Figure 10.29). The choice for this polyhedron form was based on the idea of reducing the pixel wasting by better fitting the slice. At the same time, we try to

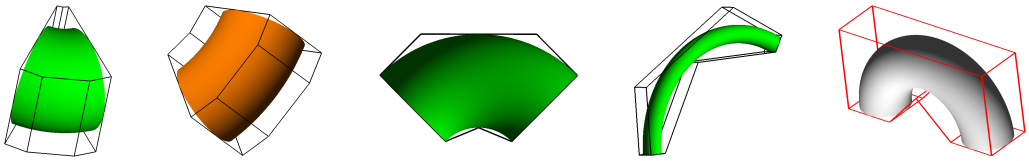


Figure 10.29: The torus slice RCA (Ray-Casting Area) is a bounding polyhedron with 14 vertices. It is well fitted around tori of different slice angles (from left to right): 20°, 45°, 90°, 120°, 180°.

avoid a significant vertex cost and we keep the possibility of surrounding torus slices from small angles up to 180 degrees. The vertex cost of our torus slice is $V(RCA_{Bpoly}) = 14$.

We have implemented a special vertex shader to automatically locate vertices around the torus slice based on its parameterization data.

Each vertex can be classified as either internal or external, top or base and by the relative angle (see Figure 10.30).

For each torus slice the application passes the following information to the GPU:

- center (3 scalars);
- revolution vector (3 scalars);
- center-to-begin vector (3 scalars);
- slice angle, i.e. 90° for a quarter of torus (1 scalar);
- big radius and small radius (2 scalars), the radii could be passed implicitly as the vectors norm; and
- the parameterized vertices (there should be at least 12 vertices, in our implementation we use 14)

The application sends each vertex by passing their parameterized coordinates. The x coordinate actually contains -1 for base and 1 for top, the y coordinate is -1 for internal and 1 for external and the z coordinate represents the angle in the interval $[0, 1]$. The vertex shader receives this unique classification and computes its world and local coordinates. (The local coordinate system is deduced from the two vectors: revolution and center-to-begin.)

As in the cylinder billboard case, these parameterized coordinate vertices are generic (can be used to any torus slice limited to 180 degrees), so they are pre-compiled in a unique display list. In rendering time, for each torus slice, the CPU sends to the GPU a group of 12 floats (the 12 scalars listed above) and the display list call.

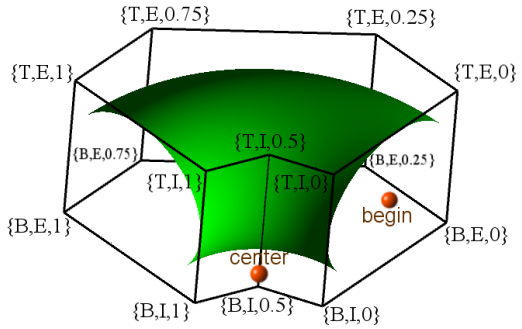


Figure 10.30: The adapted bounding polyhedron for torus slices with 14 vertices. Each vertex has a parameterized coordinate, where B/T means base/top, E/I means external/internal and the third coordinate lies in the interval $[0, 1]$ representing the partial angle (1 is the complete slice angle, 90 degrees in this case).

Chapter 11

Results for Manufactured Model Visualization

Our goal is to represent and render smooth surfaces of huge industrial plants with our GPU primitives in order to achieve three benefits: **quality**, **performance** and **less memory usage**. In contrast with the polygonal tessellated solution, GPU primitives do not require any LOD management since they have high quality even when viewed up close, and demand reduced computation when viewed from far away. However, it is important to notice that our primitives cover about 90% of the objects in these huge models, and the other objects do need conventional rendering techniques.

In the following subsections we discuss the advantages of our technique by grouping them according to the three above-mentioned benefits.

11.1 Enhancing Image Quality

We want to compare the image quality obtained in our results to the one obtained by the usual triangle rasterization on graphics cards. Both methods focus on interactive rendering and it would be out of context to compare our results with offline rendering. We have listed five image quality improvements obtained with our method.

Smooth silhouette. The tessellation of curved surfaces prevents the possibility of rendering a continuously smooth silhouette, since it is restricted by the mesh discretization. The GPU primitives are computed by pixel, therefore the silhouettes are always smooth, even after a huge zoom. See Figure 11.1(a).

Intersections (Per-pixel z-computation). In conventional triangle rasterization, the z-buffer depth of each pixel is the result of interpolation of per-vertex depth. On the other hand, the GPU primitives compute per-pixel depth, which is much more accurate. When two or more GPU primitives intersect, the boundary has a correct shape due to this correct visibility decision. See Figure 11.1(b) for a comparison.

Continuity between pipe primitives. In industrial plant models, the pipes are a long sequence of primitives (cylinders, joints and cones). With the tessellated solution, there is a risk of cracks appearing between primitives. To avoid them, the tessellation should keep the same resolution and the same alignment for both consecutive primitives. In Figure 11.1(c), a misalignment has caused cracks. This kind of undesirable situation is also a problem in applications applying LOD in pipes. In that case, it is hard to guarantee

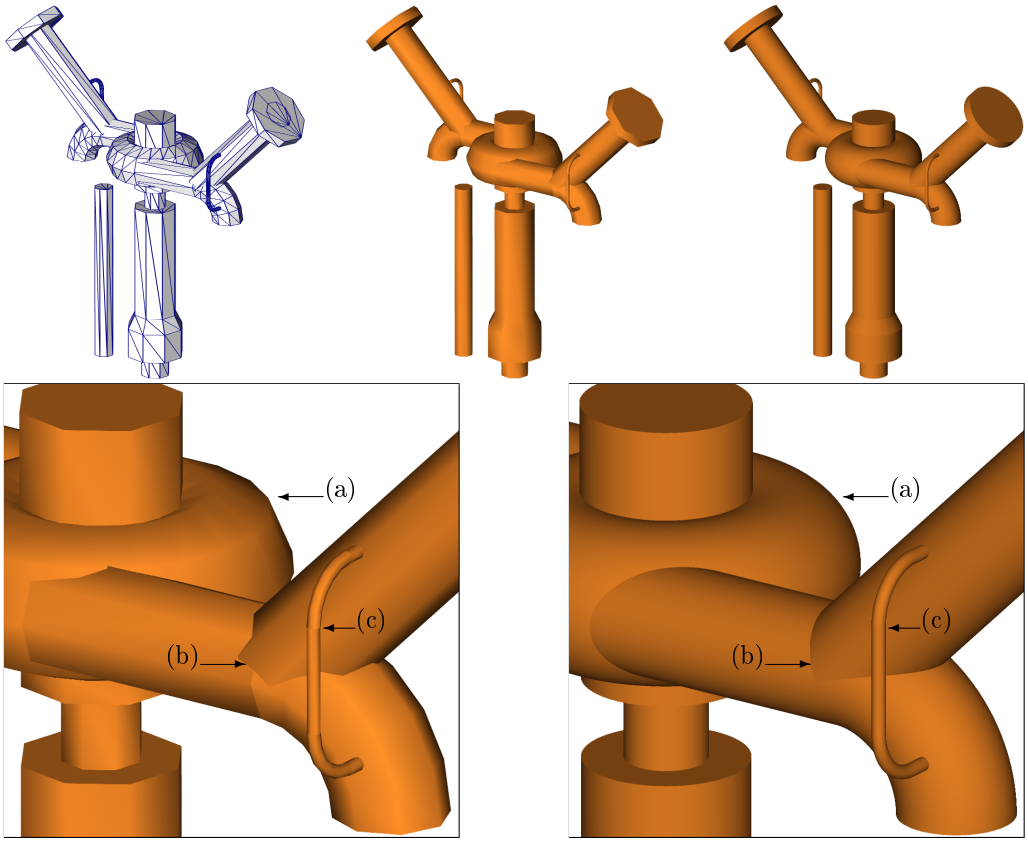


Figure 11.1: An industrial piece in Power plant Section 13. *Top left:* Original tessellated data. *Top center:* Usual triangle rasterization with Gouraud shading. *Top right:* Rendering with GPU primitives. *Bottom:* Triangles versus GPU primitives. **(a)** Silhouette roundness; **(b)** intersection precision; **(c)** continuity between consecutive primitives.

continuation (see [KBO+99]), whereas, with GPU primitives continuity is natural since there is no discretization.

Per-pixel shading. In the default implementation of triangle rasterization, the GPU processes the color by vertex for further interpolation inside the projected triangle. This technique is known as Gouraud shading [Gou71a, Gou71b]. Although fast, the final images are not so good and artifacts are visible, especially around specular reflections (see Figure 4.6 on page 69). Our primitives compute the shading by pixel (Phong shading), without any interpolation, enhancing image quality.

Cylinder thickness control. The thickness control adopted in the vertex shader of GPU cylinders avoids *dashed* rendering when zooming out a thin and high cylinder. Moreover, when this control is associated with color computation based on the normal average, it significantly reduces the aliasing pattern effect for a group of parallel cylinders (see Figure 10.14 on 175).

11.2 Memory space

In Table 11.1 we summarize the memory-space information in Tests 2.1, 2.2 and 2.3 (see Section 9.4).

Test number	2.1	2.2	2.3
Model	PowerPlant Sec. 1	PowerPlant	P40 oil platform
Initial triangles	3,429,528	12,742,978	27,320,034
Initial space	119MB	482MB	1033MB
Convertion rate	100%	90.18%	89.26%
Primitives space	1.9MB	6.5MB	10.25MB
Total space	1.9MB	79.3MB	121.19MB

Table 11.1: Memory space comparison.

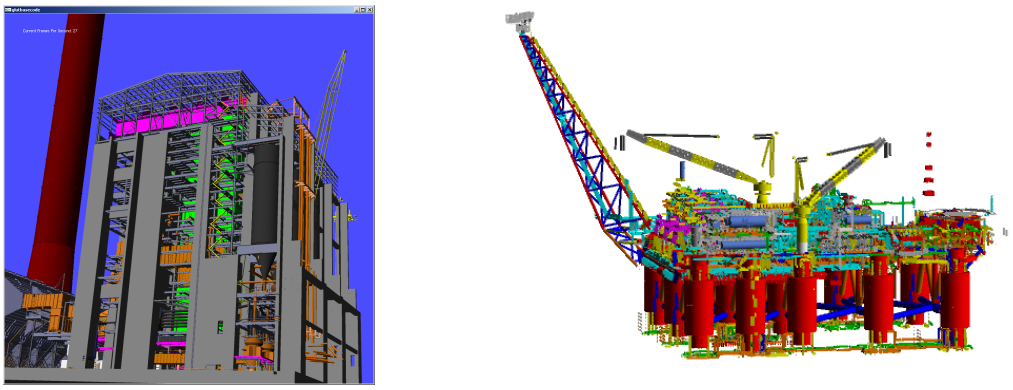


Figure 11.2: *Left:* PowerPlant model used on Test 3.1. *Right:* P40 oil-platform model used on Test 3.2.

The topological recovery procedure converts 90% of the original data of industrial models (Test 2.1 is an exception because it is only composed by tubes and pipes resulting in 100% of conversion rate). The memory-space of recovered data is reduced to at most 2% (98% of reduction). This is a consequence of compact implicit representation used for recovered primitives. If we consider the remaining 10% of unrecovered triangles, industrial models such as oil platforms and power plants can be stored in about 15% of its original data size.

This expressive reduction in storage also implies an important contribution for rendering performance. Most of the time, bandwidth limitation is the main obstacle to performance in massive model visualization. Transferring information from CPU memory to GPU memory and then to vertex processing are expensive operations. Alleviating these steps is one of the keys for better frame rates (see next section).

11.3 Performance

We have done two sets of performance measures in Test 3.1 and Test 3.2, varying data and visualization settings (see Figure 11.2). In both tests, the hardware parameters were:

- PC processor: AMD Athlon(TM) XP 3800+ 2.41GHz
- 2 GB memory
- Graphics Card: GeForce 7900 GTX512MB (anti-aliased and VSync turned off)

Results (Test 3.1)

In this test we have used some subsets of the PowerPlant model. The rendering was done in orthographic projection and the model was entirely in frustum view, fulfilling a

Section	FPS				Other information	
	GPU primitives		VBO			
	isolated	grouped	multiple	simple	triangles	primitives
1	48.5	93.0	51.0	2.9	3,429,528	57,938
15	138.5	268.5	177.5	165.5	1,141,240	21,839
19	62.5	128.0	73.5	3.8	2,650,680	42,046
20	66.5	130.5	66.9	4.1	2,415,976	41,872

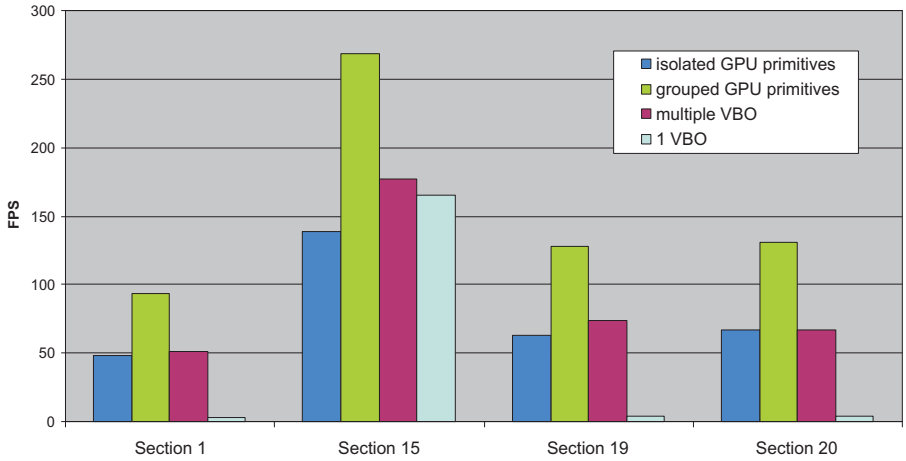


Table 11.2: Performance test for Sections 1, 15, 19 and 20 of PowerPlant. The frame-rate values (*FPS*) are plotted in the graphic on the bottom. *Other information* columns were extracted from Table 9.2.

1024×768 screen. In this scenario, the bottleneck is not the pixel shader since primitives are not so large on the screen.

PowerPlant sections 1, 15, 19 and 20, used in Table 11.2, were intentionally chosen because they were 100% recovered by our reverse engineering algorithm (see Table 9.2). This way, we can directly compare rasterization and GPU primitive techniques. We have created two situations for GPU primitives, standard (as explained in Chapter 10) and grouped (see Section 11.3.1). The latter is twice as fast as the standard solution. In the same way, we have compared two rasterization strategies based on VBO (Vertex Buffer Object): in one of them, the model is decomposed in multiple VBOs and in the other one, a unique VBO is used. Decomposing is a good strategy for large models (millions of triangles for instance). Note that for PowerPlant section 15 (the smallest among the 4 tested sections) multiple VBO is not so important.

What we want to compare is our best solution for GPU primitives and the best solution for triangle rasterization (second and third columns in Table 11.2). As a conclusion, GPU primitives are clearly faster (almost doubling speed) compared to the best rasterization solution.

In Table 11.3, we have a different situation, where some of the original triangles (10%) were not converted to implicit primitives. We call them UT (Unrecovered Triangles). As a consequence, we compare the best VBO rasterization rendering (third column) with the hybrid rendering (GPU primitives + UT, second column). As we can see in Table 11.3, the hybrid solution is between 40% and 60% faster than rasterization.

Note that all the rendering tests were executed without any culling technique, which would speed-up all the frame rates.

Section	FPS				Other information		
	GPU primitives		VBO				
	(only)	(+ UT)	(all)	(UT)	triangles	UT	primitives
12	911	719	524	2935	360,872	38,067	6,205
all	27.8	21	12.9	81	12,742,978	1,251,019	202,372

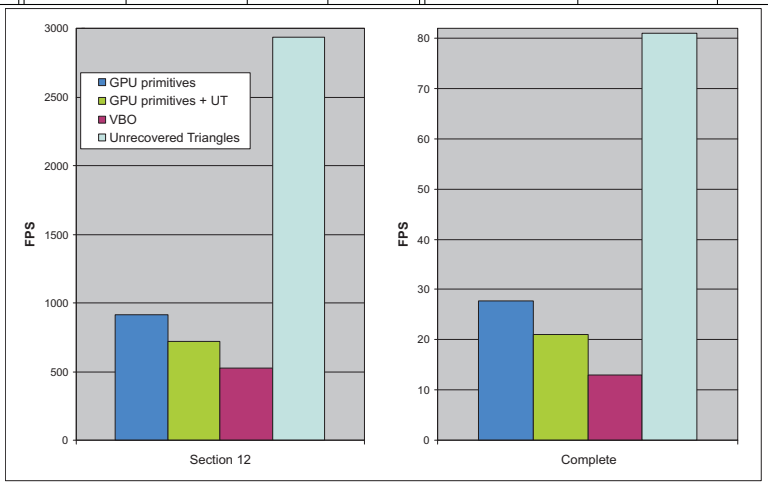


Table 11.3: Performance test for Section 12 and the entire PowerPlant. *UT* means unrecovered triangles, in other words, triangles that were not converted into primitives by the reverse engineering. The frame-rate values (*FPS*) are plotted in the graphic on the bottom. *Other information* columns were extracted from Table 9.2.

Method	FPS	Memory	Triangles
4-sided mesh	40fps	283MB	4,475,852
8-sided mesh	21fps	464MB	10,559,474
12-sided mesh	13fps	740MB	18,394,158
GPU primitives	22fps	89MB	(*)

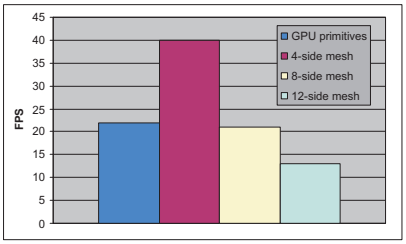


Table 11.4: Performance results for oil-platform P40. (*) 341,413 primitives: 215,705 cylinders, 40,001 cones and 85,707 tori.

Results (Test 3.2)

For this test, we have used an oil platform (Petrobras P40) in perspective. In this set of measures we have compared the visualization of GPU primitives with mesh rasterization with different levels of tessellation. The testes were done with 4, 8 and 12-sided meshes (see Figure 11.3). We were not able to load a 16-sided mesh version because of RAM memory restriction (see Table 11.4). Note that in this test we have ignored the *unrecovered triangles*.

Since we have not implemented any culling technique and the rendering is bottlenecked by the vertices, the frame rate is almost constant independently of camera position. For this reason we have decided to measure the frame rate from a fixed and similar point-of-view in all four cases (from abroad and viewing the entire platform fulfilling the screen).

As we can verify in Figure 11.3, the GPU primitives achieve a much better image quality than any tessellated solution. The performance of GPU primitives is comparable to the 8-sided solution (see Table 11.4). Finally, as shown in Table 11.4), the memory use is much smaller with our primitives than with the tessellated solution.

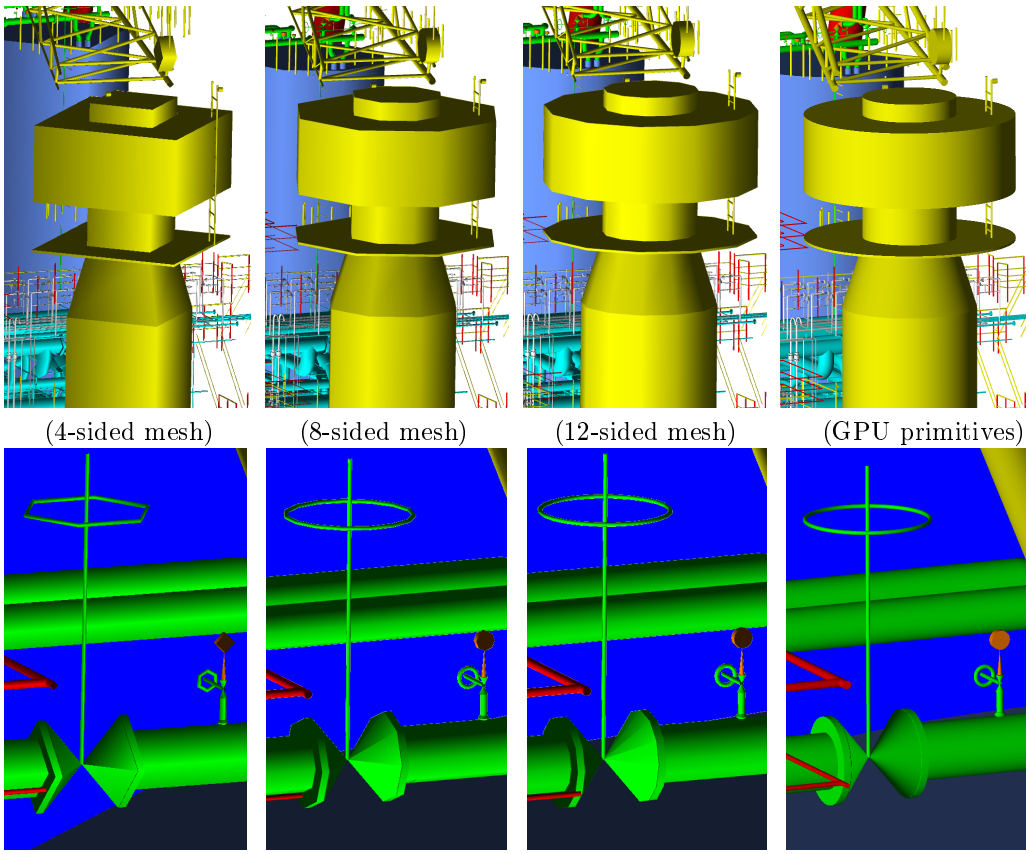


Figure 11.3: Image quality comparison for P40 rendering.

11.3.1 Grouping numerous primitives.

In our rendering tests we have identified that the bottleneck lies in passing primitive information to the vertices that form the *RCA*. For this reason, we have decided to slightly change the way we pass individual primitive data from CPU to GPU, grouping all primitive information.

The grouping strategy is based on a special feature of recent graphics cards: per-vertex texture fetching. Before the first frame rendering, we include the information of all primitives in a floating-point texture, loading it into video memory. The vertices that define the *RCA* implicitly include the (u, v) texture coordinate to be accessed by the vertex shader. For example, in our standard solution, each cylinder is trigged by a `glRect(-1,-1,1,1)` command. In our grouping solution we use `glRect(-u,-v,u,v)`. Before doing its task, the vertex shader starts by reading implicit information from the texture. In the case of a cylinder (see Section 10.4), this information is a set of 7 scalars that can be read in two texture fetches (each texture fetch brings up to 4 floating-point values).

This strategy has resulted in doubling the speed of our primitives (see Test 3.1). In despite of that, the bottleneck is still in loading implicit information, and it is located in the texture fetching. This fact can be verified by increasing the number of texture accesses (adding an extra dummy instruction) which reduces frame rate. On the other hand, if we reduce texture accessing (partially setting unvarying information, e.g. same radius and direction for all cylinders), the frame rate proportionally climbs.

In Section 12.2 we discuss some possibilities to overcome the above limitation on state-of-the-art graphics card technology (NVidia's G80).

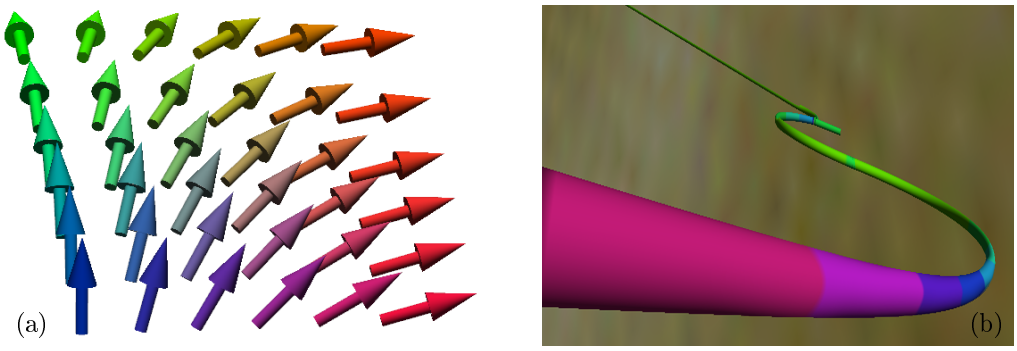


Figure 11.4: (a) Arrows formed by cones and cylinders. Generally, the colors of the glyphs can be used for extra information. In this simple example, the colors represent direction, mapping $[x, y, z]$ into $[r, g, b]$. (b) An oil riser with colors showing simulation result properties.

11.4 Other Applications in Scientific Visualization

The use of implicit primitives with high quality and performance has countless applications. There are three of them we want to emphasize:

Glyphs. Several applications make use of glyphs to show multiple properties in a specific domain. In Figure 10.9 on page 170 we have shown the use of ellipsoids for tensor of curvature visualization. In that example the domain was a surface: however, more sophisticated applications could display specific properties in $\mathbb{R}^3$. Ellipsoids are just an example of possible primitives for glyphs visualization. Cones, cylinders and spheres are also useful. For instance, the combination of a cylinder and a cone form an arrow that can be used to show vectorial properties (see Figure 11.4(a)).

Streamlines. Streamlines are 3D lines with a specific radius, being shaped into a cylindrical aspect. We can represent streamlines through a set of points in $\mathbb{R}^3$. Optimally, each pair of consecutive points should be connected as Bezier curves, but in practice, sequences of adjusted GPU cylinders can be used for rendering.

Streamlines are an appropriate solution to visualize the path of particles in time. Flow simulation is an example of this kind of application [SGS05]. In medicine, there are multiple applications which can make use of streamlines, for example visualization of white matter tracts [MSE+06] and heart fibrillation [KKW05]. In the oil industry, they can also be used in 3D simulations and for rendering specific real objects. For example, wells and risers can be rendered using GPU streamlines. See Figure 11.4(b).

Molecules. Another straightforward application of GPU implicit primitives is in chemistry. Molecule visualization can make use of spheres and cylinders (see Figure 10.7 on page 169). More complex GPU primitives can be used for Connolly surface visualization (see Figure 11.5).

Solvent-excluded surface (also known as Connolly surface) is formed by the region defined by a probe sphere center as it rolls over the atoms [Con85]. As a result, there are three basic primitives that compose the final surface:

- Spheres around the atoms position, with radius equal to atom radius plus probe radius. They are shown in yellow in Figure 11.5.
- Toroidal patches (they have the form of the interior of a torus) connecting two atoms with distance less or equal to probe diameter. They are shown in blue in Figure 11.5.

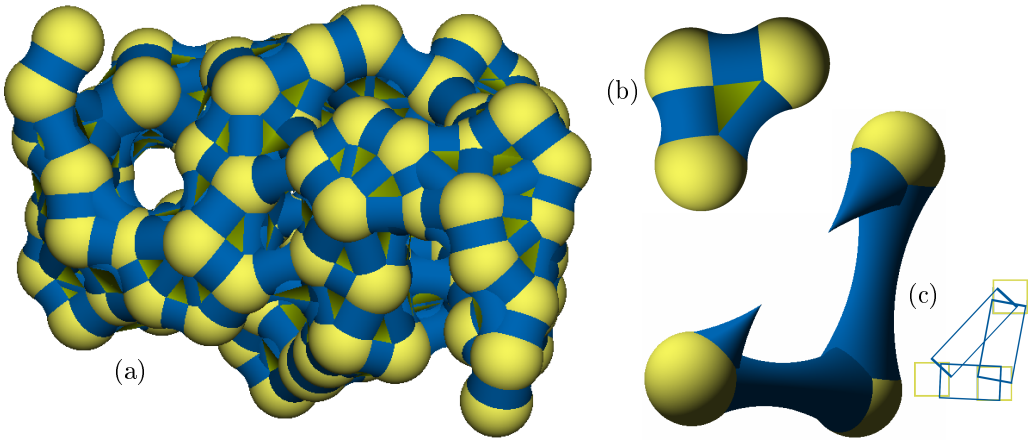


Figure 11.5: (a) The solvent-excluded surface of the Carbon-Alpha atoms. (b) The three types of primitives: spheres in yellow, toroidal patches in blue and spherical concave patch in green. (c) The toroidal patch can form special shape (with disconnected extremities) depending on the atom/probe radii configuration. Even in this exception, the toroidal patch is treated as a unique GPU primitive with $RCA_{Billboard}$ formed by a quadrilateral.

- Spherical concave patches shown in green in Figure 11.5. This patch connects three atoms that are already connected by three toroidal patches.

Figure 11.5 shows Carbon-Alpha solvent-excluded surface rendered with GPU spheres and two specific GPU primitives for toroidal patches and spherical concave patches.

Chapter 12

Conclusion

As was discussed throughout this thesis, interactive rendering is one of the major tasks in computer graphics. It is especially challenging for several applications, such as simulators, CAD/CAM, games, and so on. When the data in use have a significant size (millions of polygons for instance), applications suffer from speed limitations that may hinder interactivity. Massive models are usually composed either by numerous small objects (such as an oil platform) or by very detailed geometry information (which is the case of high-quality scanned models).

The goal of our work was to speed-up visualization methods in order to obtain interactive rendering of these massive models.

In Part I, we have reviewed the visualization literature from the scale-level point-of-view. We can roughly say that advanced algorithms for scene-scale and macroscale levels focus on increasing performance without losing quality. On the other hand, advanced algorithms for mesoscale and microscale levels focus on enhancing quality while trying to minimize performance downgrade.

At scene level, culling algorithms are the main key to increase frame rates (see Chapter 3). They have been deeply explored in the past years. At the macro level (Chapter 4), different techniques (from LOD to impostors) are used to improve rendering speed. Mesoscale level is characterized by several methods that introduce details in the final rendering (see Chapter 5). Finally, at the microscale level (Chapter 6) the inserted details are more precise than in mesoscale, based on microscopic physical phenomena that occur with the light in the scene.

We have focused our contributions on the macroscale level, introducing new surface representations and rendering primitives (see Chapters 7 and 10). We have shown that our technique can also be used for mesoscale purposes (see Section 5.3.3). In our approach, massive models are classified into two different categories: natural and manufactured models. For each one, we have proposed a new representation, implementing a conversion algorithm and developing GPU-based primitives.

The following sections summarize the main conclusions about our **contributions** for natural and manufactured models including representation, conversion and rendering.

12.1 Natural Models

In Section 7.1 we have introduced extended GPU primitive principles, which are based on a per-pixel ray-casting algorithm. Section 7.3 has presented *Geometry Texture*, which is a derivation of height-map GPU primitive (Section 7.2). In Chapter 8 we have proposed a new representation for natural models using a set of geometry textures.

Our results (Section 8.2) have shown that this new representation is suitable for natural models. The final rendered images have similar quality compared to traditional

polygonal rasterization methods.

The following items are some of the positive aspects of our technique:

- The rendering speed naturally follows a LOD behavior. This means that when the model is small on the screen (i.e., far from the camera), rendering is faster. This is a result of relieving the vertex-stage burden, transferring computation bottleneck to pixel stage.
- Depending on the chosen geometry texture resolution, our representation requires less memory than polygonal representation, without losing significant geometry information.
- Our technique is compatible with common rasterization methods, thus geometry-texture objects can be inserted in any virtual scene composed by polygonal objects.

Our potential drawbacks are:

- Recent graphics cards that implement VBO (Vertex Buffer Object) extension have a very high performance for polygonal rendering. As a consequence, when compared to VBO performance, our technique is faster only if the model is not so big on the screen. For example, the geometry textures for the dragon model are faster than the polygonal version if the model's screen size is below than 800×600 (see Section 8.2.1, Figure 8.6). In our tests we have considered one model rendering. However, if the scene contains multiple models, polygonal rendering would proportionally lose performance, while our solution would keep a stable frame rate.
- Another adverse point is the sophisticated conversion algorithm (Section 8.1). Our procedure is composed by multiple passes, one of them, the partitioning step, being considerably elaborate. All this complexity may prevent the practical use of our technique for natural models.

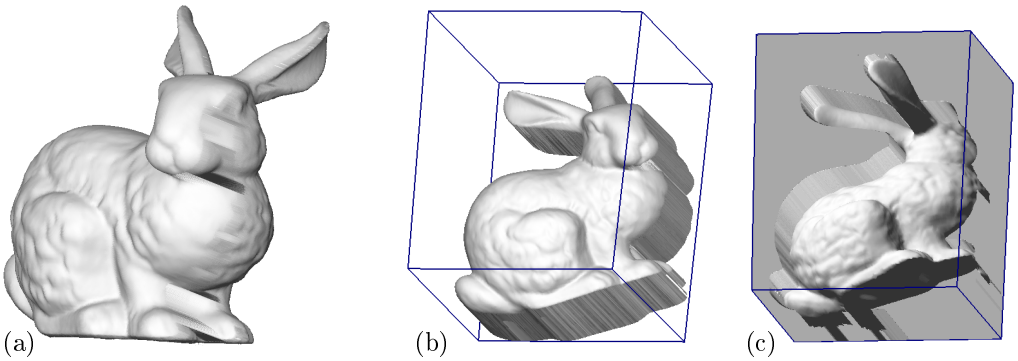
This last point can be overcome in one of our future work propositions (see below).

Future Work for Natural Model Visualization

GPU height map on surfaces with curvature. In Section 7.2.3.3, we have introduced this topic showing our solution for convex/planar curvatures. We have also explained how the technique could be extrapolated to other types of curvature. We suggest a continuation of this research since it is a promising solution for mesostructure visualization. We have not continued this study branch because it was out of our current scope, which focuses on the macroscale problem.

Image operations on geometry textures. Once we have a new geometry representation based on images (height maps), image operations can be applied on these maps to obtain new results. For example, one could use a low-band filter to smooth the geometry (or a high-band filter to highlight small geometric features). These image operations to transform geometry and their consequences can also be the focus of a new research.

Multilayer height-map. Multilayer height-map is an alternative to overcome the limitation in representing surfaces with folding. The idea is to have only one "geometry texture" to represent the entire natural model. We can start by determining the unique bounding-box and its height direction. After that, we can obtain multiple height-maps from the polygonal geometry. This can be done by successively taking the highest point for each discreet coordinate (u, v) in the domain. In rendering time, a multilayer height-map is rendered using per-pixel ray casting. The pixel algorithm treats the layers as



(a) (b) (c)

Figure 12.1: Multilayer height-map idea. The bunny is rendered (a) using two layers, one for the top (b) and one for the bottom (c). The rendering artifacts in (a) are a consequence of using only two layers in this example.

forming a CSG (Constructive Solid Geometry) model. Figure 12.1 shows an example with the bunny model and two height-map layers (indeed, this model would need more than two, but it is just a proof of concept).

This last proposition yields a new type of representation for natural models. Although similar to a geometry texture set, it may have some advantage in reducing implementation complexity. In preprocessing, the multilayer height-map would skip partitioning (which is the most difficult step) and overlapping steps. In rendering time, this technique would require fewer vertices and would avoid multiple fragments for the same pixel. On the other hand, pixel shader would become more complex.

12.2 Manufactured Models

In Part III, we have focused our work on industrial plant visualization. Among the manufactured models, industrial plants are the most challenging data for interactive rendering. In this kind of data there are numerous pipes and technical objects that are geometrically described by combining simple primitives (cylinders, cones and elbows). In common interactive visualization systems, these primitives are tessellated and rasterized in the final image. In this thesis we suggest replacing tessellated primitives with our GPU implicit primitives (Chapter 10).

In general, industrial plant models are described by a set of polygonal meshes, with all primitives already tessellated. For this reason, we have developed a reverse engineering algorithm (Chapter 9). The results have shown that this algorithm is highly efficient if compared to traditional recovery procedures, although specific to this type of data. More than that, we have seen that 90% of polygons can be replaced by implicit primitives, significantly reducing memory space.

Finally, in Chapter 11 we have detailed the positive points of our method. The benefits are classified in three categories: quality (e.g., perfect silhouette and per-pixel depth), memory and rendering efficiency. In this last point, we have shown that we can practically double speed if compared to traditional triangle rasterization. Grouping the primitives information has proven to be a good strategy for performance purposes.

Future Work for Manufactured Model Visualization

The use of GPU primitives for CAD and industrial models is very promising and now we point out some possible future work in the list below:

New primitives. We have developed GPU primitives for several implicit surfaces (see Chapter 10). However, there are some surfaces found in industrial plant that were not covered in our work, such as sheared cylinder, sheared cone and half sphere (reverse-engineering topological procedures should also be implemented). New primitives can be developed for other applications, as exposed in Section 11.4 (e.g., streamlines and Connolly surface primitives).

Optimizations with new graphics card generation (G80). In most recent graphics cards a new programmable stage, geometry shader, is available. This stage is located between vertex and pixel processes in the pipeline (see Appendix A) and it allows the creation of new vertices directly in GPU. Exploring this new trend can originate different research topics. A possible experiment would be to test polygon/rasterization solution with GPU LOD to compare with our results. On the other hand, our GPU primitives could be optimized in per-vertex primitive information fetching. We have seen that the performance bottleneck consists in passing per-primitive implicit information to the vertices (see Section 11.3.1). We suggest using a unique per-primitive vertex that receives the implicit information and creates the other vertices in their correct position to form the final *RCA*. In this situation, the number of texture fetching calls (in our grouping solution) would be reduced significantly, for example from 42 to 3 fetches for each torus slice. Note that if we use one vertex per primitive, maybe using VBO instead of a texture, it could achieve a better performance without repeating information in memory, as explained in Section 11.3.1. Eventually, this *primo* vertex could be charged with new responsibilities, such as a frustum-culling test.

CSG primitives. Since we can render implicit primitives based on their equations, the same idea can be extended to render CSG primitives. Based on our work, Velho et al. [RdFV06] have developed *Hardware-assisted Rendering of CSG Models*. In their method, a complex model is spatially subdivided in an octree in such a way that in each cell there are at most two primitives in one Boolean operation. Each individual cell is then visualized with a simple GPU ray-casting algorithm. We propose a different approach, mixing our work with the depth-peeling technique; it is possible to obtain similar results to those presented by Hable and Rossignac [HR05], but for implicitly defined primitives. As a consequence, there is no need to subdivide the model and it can be visualized by rendering the primitives in a bottom-up order (in the CSG hierarchy) based on the operations.

Mixing height-map and implicit primitives. Finally, another point for study would be to combine implicit and explicit representations for the same primitive. As will be demonstrated in the next section, both types of primitives can be rendered together without any restriction. But what we propose here is something else. The idea is to mix them as a new primitive. For example, we could produce some holes in natural models by subtracting some implicit spheres from the geometry texture primitive. Another example could be rendering implicit surface objects using our GPU primitives but applying height-map mesostructure information. For instance, Figure 7.9 in page 126 was created with a polygonal torus, but it could have been made with a GPU torus.

12.3 Hybrid Rendering

One of our important contributions was a new visualization system that combines ray-casted and rasterized objects in the same scene. As explained in Section 7.1, extending a little bit the ray-casting algorithm to respect z-buffer rules is the key for the seamless combination of the different techniques.

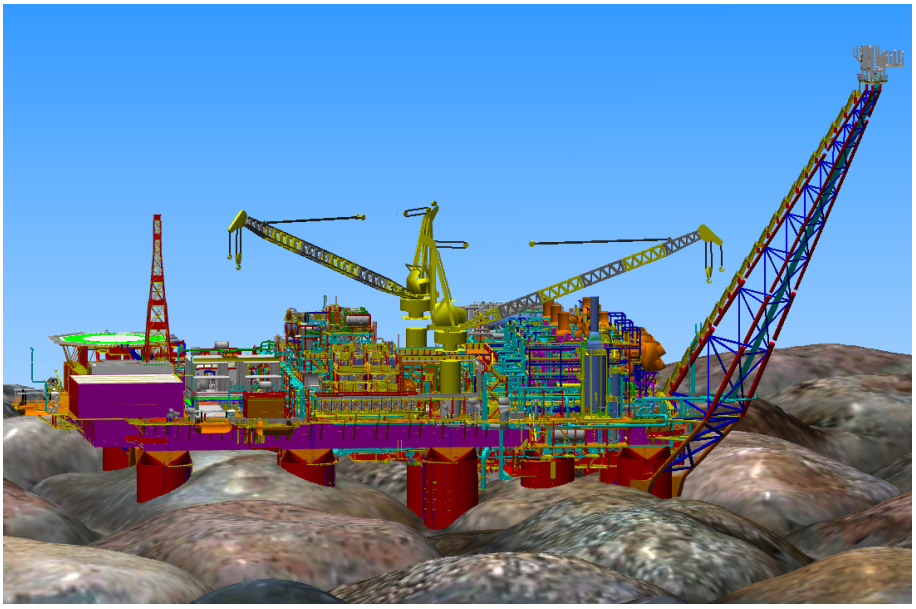


Figure 12.2: This image is a proof-of-concept of our hybrid rendering technique. It contains 2,932,177 of triangles (representing platform walls, floors, and hull), 341,413 implicit GPU primitives (representing pipes and other manufactured objects) and a geometry texture (representing the rocks). While triangles are rasterized, the other primitives are ray casted; the final image is possible because the z-buffer is updated by all of them. This scene was rendered at 14 fps with a NVidia GeForce 8800 graphics card.

In Figure 12.2 we show the combination of three types of primitives. Ray-casted implicit primitives (cylinders, cones and tori), ray-casted natural object (the terrain) and rasterized triangle meshes (for other platform objects). Due to correct per-pixel z-value output, all those different objects (with different rendering methods) are perfectly combined for interactive visualization.

Appendix A

Graphics Hardware

When the first video cards appeared, their main task was very simple: to take the final image produced by the CPU and display it on the screen. In the mid 1990's, a revolution started to take place. Graphics cards were updated with a processor specialized in 3D rendering called GPU (Graphics Processing Unit). With this new resource, the CPU was partially free from executing the rendering process, now performed by the GPU. See Figure A.1 to have an idea of this evolution.

GPU technology is dedicated to fastly rendering 3D triangles, mainly because of the hardware implementation of the algorithm to rasterize them (see Section 3.1.2).

There are some geometric reasons for choosing triangles as the geometric primitive to be used: triangle vertices are always coplanar; any object can be approximated by a triangular mesh; any polygon can be triangulated; triangles are always convex; and barycentric coordinates can be used as an unambiguous rule to interpolate vertex values in the triangle domain.

To process the triangles, the GPU has a 3D pipeline implemented in hardware. Thus, the 3D scenario (including the triangulated meshes) is part of the input and the output is the final 2D image. We briefly discuss the 3D pipeline in the following section.

A.1 3D Graphics Pipeline

Now that the GPU is responsible for the rendering, the CPU merely needs to pass some information to the graphics card and let it do the work. Such information consists of: the objects (or surfaces) described by triangles, their vertex attributes (positions, colors, texture coordinates, normals, etc.), optional texture images to be applied over the meshes, and information about the scenario (camera position, light, and so on), see Figure A.2 (*left*). The GPU receives the data and executes the 3D pipeline (Figure A.2 (*right*)), divided into the following main stages:

- *Transformation*: responsible for transforming the 3D world-space coordinates of the vertices into camera space, used for positioning the point of view, and for translating and rotating objects.
- *Lighting*: in this step the vertex colors are computed using light position and properties, object material properties, and the vertex normals.
- *Projection*: responsible for converting the 3D coordinates into 2D screen coordinates.
- *Rasterization*: responsible for drawing each pixel of the 2D converted primitives. It interpolates the vertex attributes, filling the pixels and applying the texture, if it is the case.

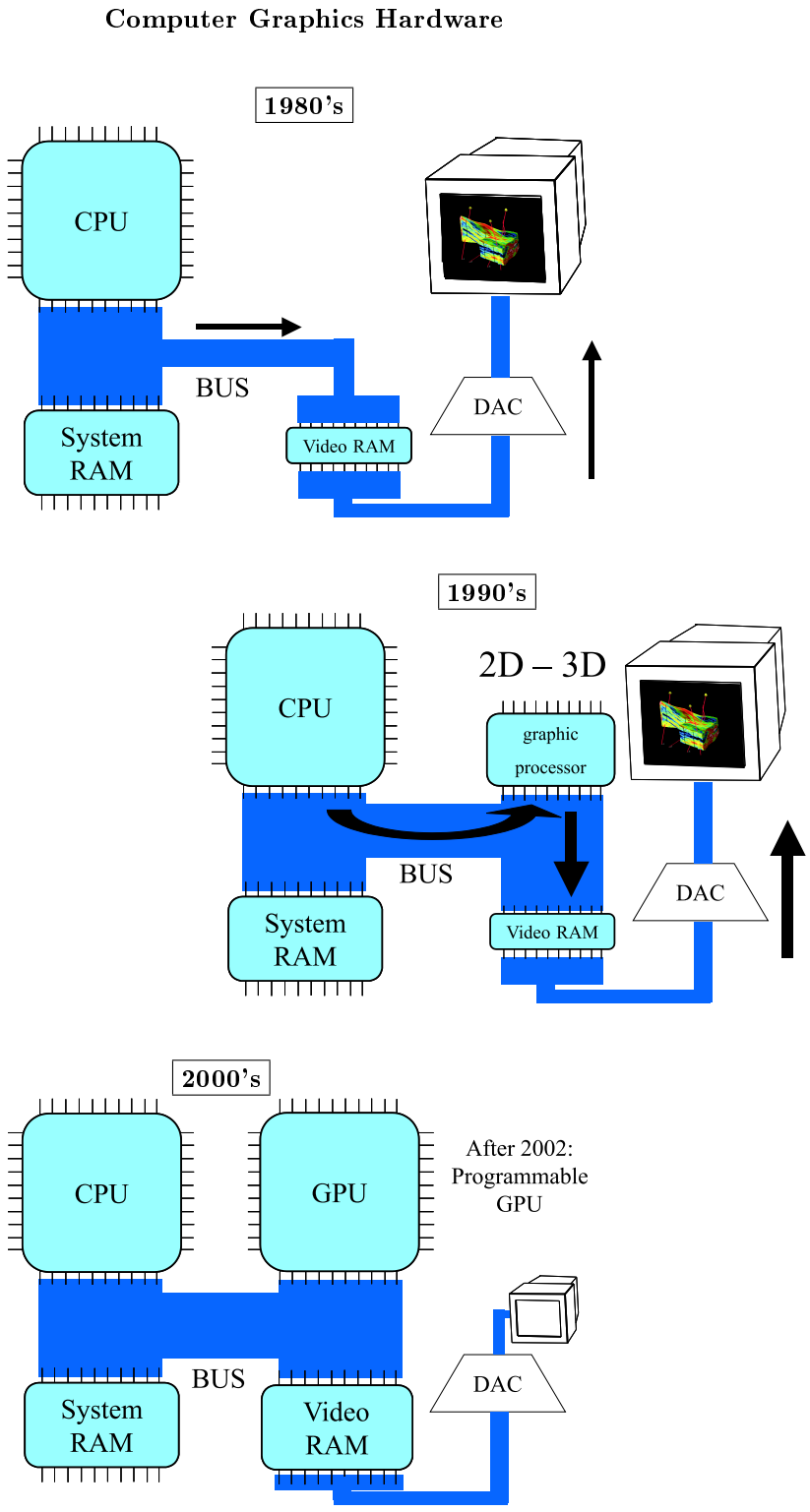


Figure A.1: Graphics hardware evolution.

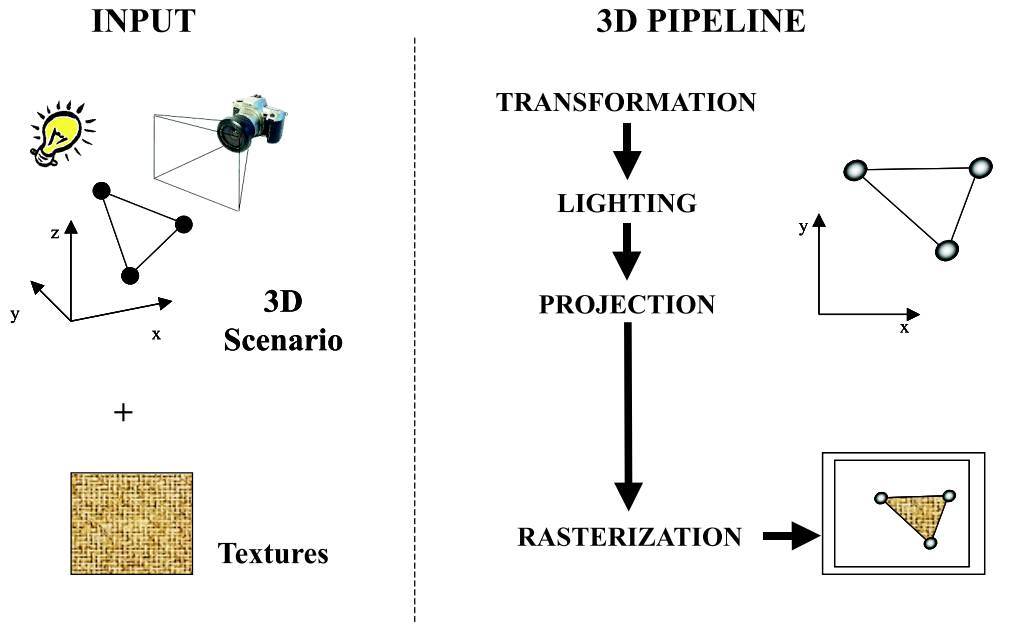


Figure A.2: GPU input and the 3D graphics pipeline.

Until recently, the 3D pipeline described had a fixed functionality. We could modify the parameters but we could not change the algorithms inside of it. Even with some improvements implemented in the GPU pipeline, such as multi-textures and 3D textures, the algorithms remained unchangeable. However, as we will see in the next section, current GPUs enable programmability in part of the pipeline. This way, the programmer can actually change the default behavior of the graphics cards.

A.2 GPU Evolution

Since GPUs appeared in the mid-1990's, they have been undergoing significant evolution. Their power has been doubling every 12 months, faster than Moore's law for CPUs (which double their power every 18 months). Besides the power, their features also improved significantly.

In the beginning, GPUs were able to render triangles computing their illumination and applying a 2D texture. New features appear in every new graphics cards generation, such as multi-textures, blending, multi-pass, occlusion queries and 3D textures. However, until recently, GPUs were restricted to behave like a black-box, whose input was the scene (textured objects + light + camera position) and output was the final image. Inside the processor, multiple stages were executed (the 3D pipeline, see previous section), with no possibility to change their behavior, except through parameters that could be added to the input.

In 2002/2003, new GPU models have become programmable. Now, users can completely change the default behavior of the processors. Some programs (using GPU instructions) can also be transmitted to the pipeline additionally to the usual meshes and textures (see Figure A.3).

This programmability provides the possibility of altering the two main stages of the pipeline: the geometry stage (transformation, lighting and projection), and the rasterization one. Basically, the GPU allows the substitution of a specific geometric manipulation program, called *vertex shader*, for the standard transformation, lighting and projection processes. The input and output of the vertex shader are the vertices and their parame-

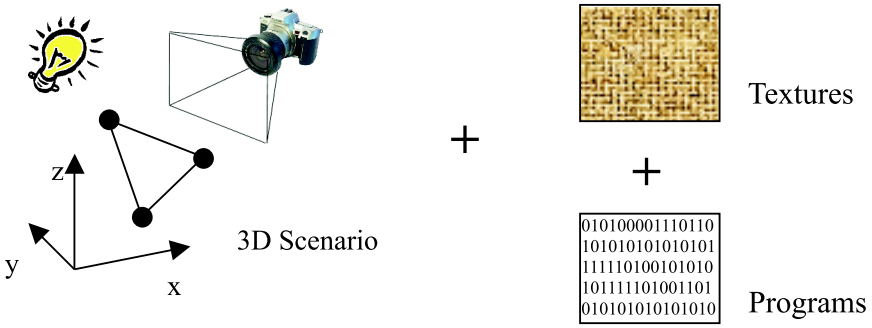


Figure A.3: New input for programmable GPUs.

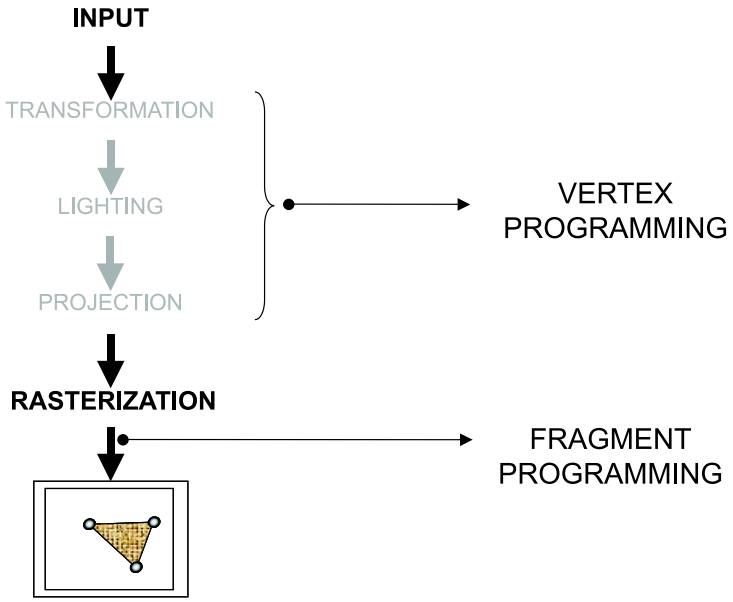


Figure A.4: Shaders are user defined programs to be executed by the GPU, replacing part of its natural pipeline stages.

ters. Further in the pipeline, a *fragment shader* loaded on the GPU can run a complex color-manipulation algorithm for each pixel that results from the rasterization algorithm. See Figure A.4.

Programmable GPUs were originally designed to motivate programmers to produce real-time applications with better quality. The same idea was implemented before for off-line graphics applications (see [EMP+02]). The *pixel shaders* can execute powerful techniques before outputting the final pixel color. Special effects, such as *bump-mapping*, Phong shading and environment mapping, can be easily programmed. Similarly, the *vertex shaders* can compute the final vertex position on the screen, based on the user code.

With the possibility to create programs executed directly by the graphics processors, the GPU has become capable of performing tasks that are different from specific graphics computations. Programmability has brought the possibility of applications that are different from interactive visualization. A new branch of research called GPGPU (*General-Purpose GPU*) started to include topics such as linear algebra [KW03b], Image Processing [MA03, GDH06] and multigrid solvers [BFGS03, GWL+03]. Even in the visualization domain, there are two topics using GPU that are also considered a general purpose approach, as they do not use the standard triangle rendering pipeline: global illumination

[CHH03, PDC+03, CHL04] and ray tracing [CHH02, PBMH02, WPS+03, KW03a]. See Section 3.1.3.3 for some references in ray-tracing implementation on the GPUs.

A.3 GPU Programmability Details

As explained in the previous section, GPU programmability is carried out by small programs that are executed in parallel in two different stages of the graphics card rendering pipeline: the vertex stage and the pixel. Consequently, there are now three levels of programming that are executed in the following order:

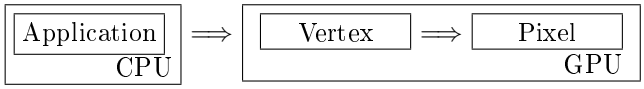


Figure A.5: Three different programming levels.

The vertex shaders are executed for each vertex sent to the graphics card, while the fragment shaders are executed for each pixel resulting from the triangle rasterization step. For example, imagine a scene with 50,000 vertices, filling half of a screen with 1000×1000 pixels, being rendered at 30 fps. The vertex shader would run 1,500,000 times per second (50000×30) and the pixel shader 15,000,000 times per second ($\frac{1000 \times 1000}{2} \times 30$).

Fortunately, the GPU stages are highly parallelized and shader codes are generally composed by few dozens of instructions. Anyway, optimizing shader algorithms is sometimes the key to obtain interactive frame rates, due to the high frequency of their execution.

There are some special conditions in GPU programming that are different from those in CPU programming. For example the basic variable/register in a GPU is a 4-tuple float that can be used to both coordinates (x,y,z,w) and colors (r,g,b,α). The variables can be used as a uniform set of values executing the operations in all values at the same time, or they can be accessed individually using suffixes as "`.x`" or "`.rgb`" (ex: "`var.rgb=1.0;`").

Another difference is the availability of very powerful operations in the GPU native set of instructions. Some examples are `MAD` (multiply and add), `DP3` and `DP4` (for dot product of 3D or 4D vectors), and `XPD` (cross product), which can be applied at once over the 4-tuple variables.

To deeply explore GPU programmability it is essential to understand its powerfulness and limitations. In the following subsections we describe both the vertex and fragment shaders' capabilities.

A.3.1 Vertex Shader

In Figure A.6, we list the essential inputs for vertex shaders and their expected output. The output list provide s a clear understanding about the shaders responsibilities. In the list below we point out some details about the output:

- there are four possible output colors (primary/secondary, front/back);
- the output coordinates are expected to be in *clip space* including w, thus (x, y, z) output coordinates will be divided by the homogeneous coordinate, later on the pipeline;
- *pointsize* is another property that can be changed in the vertex code (this is useful for our sphere GPU primitive, see Section 10.2);
- the texture coordinates are a special set of variables (at least eight 4-tuple floats) that can be used to pass parameters to the fragment shader. All information passed from the vertex to the fragment is interpolated inside the triangle (using barycentric coordinates).

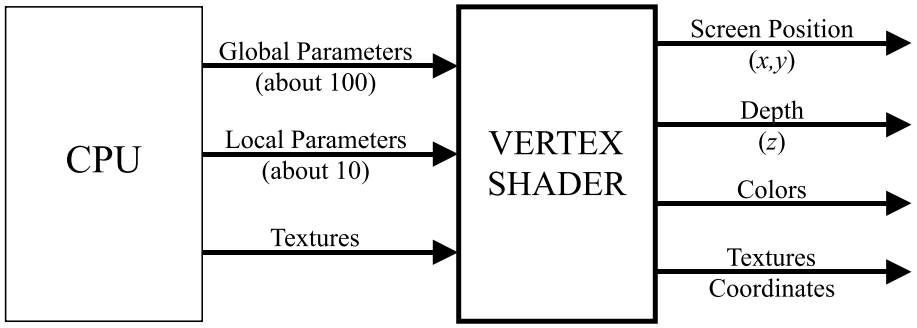


Figure A.6: Input and output for vertex shaders.

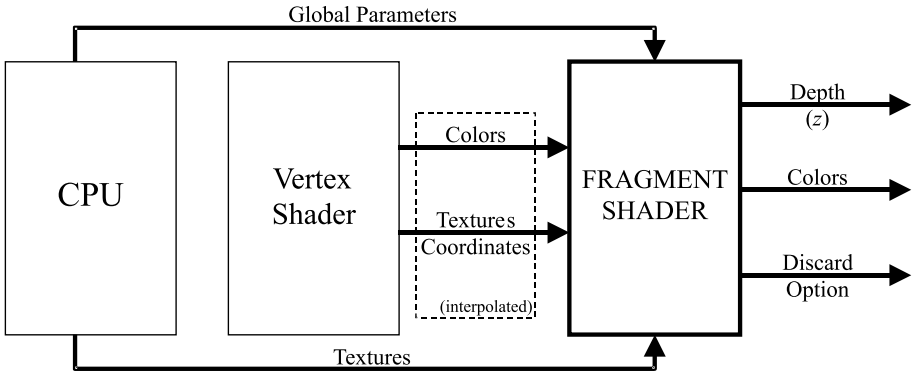


Figure A.7: Input and output for pixel shaders.

Concerning the input, also presented in Figure A.6, the difference between local and global parameters is that the former are used to pass information to a specific vertex shader, while the latter are common information to any vertex shader. Finally, the input textures have become recently available to vertex shaders (in the first programmable GPU they were only available to fragment shaders).

The conclusion is that the vertex shaders main responsibility is to compute the vertex screen position, depth and color (and pass parameters to the fragment shader).

A.3.2 Fragment Shader

Figure A.7 shows the fragment input and output. The output list provides an idea about what is possible to do with a fragment code.

In a default fragment shader, if color and depth are not changed, the output is merely the interpolated information passed by the vertices. The discard option is false by default. If it is used, it means that the current pixel will not affect the output buffers (color buffer, z-buffer, etc.) as if it had never existed.

For computation, the shader uses the input data received from the vertex and it can also access up to eight different textures (plus global parameters).

As for vertex shaders, the textures are also an input for the shader code. It is important to mention that these textures should not be seen simply as images but actually as tables of any kind of information. This is especially the case if we consider that current GPUs accept floating-point textures.

A.3.3 GPU programming languages

Programmable GPUs are a recent product, and so are the associated languages and compilers developed. They are still in progress, far from the mature C or C++ compilers. We can classify the available GPU languages in two classes: assembly languages and higher-level languages. GPU shader languages were deeply inspired in the Renderman language [EMP+02], which is also a shader language but for non-interactive applications.

The ARB [20] group has defined two standard assembly languages, one for each GPU programmable stage: ARBVP for vertex and ARBFP for fragment. The good point of this standardization is that all manufacturers must implement the defined instructions, although they could also offer their own assembly language. For example, NVidia has developed independent assembly languages NVVP and NVFP.

GPU higher-level languages have C-like syntax. The most known are: CG from NVidia, GLSL specified by the ARB group, and Microsoft HLSL. CG and HLSL have a compiler that can output different assembly languages, among them ARBVP and ARBFP. GLSL has a different concept, the compilation is done internally by the graphics card driver. This approach has two advantages: the drivers may verify any extension or restriction to the specific GPU; and when the driver is updated, the new version may contain an improvement. On the other hand, when compiling with CG to one of the ARB assembly languages, it is possible to edit the final code for optimization purposes. As we explained in the beginning of this section, optimization is an important issue for shaders.

A.3.4 Nomenclature: Pixel or Fragment Shader?

Up to here we have named the second programmable stage of the GPU *pixel shader*. However, some graphics developer groups prefer another nomenclature: *fragment shader*. For example the ARB group uses the word "fragment", while Microsoft prefers "pixel".

There is a technical reason for calling the shader "fragment". For the same pixel on the screen there could be several fragments of different triangles, and the shader will be launched for each fragment. The code can run several times for the same pixel, depending on the number of fragments. Therefore, "fragment shader" is technically the correct expression to use. However, pixel is a more intuitive notion. One can easily understand what the vertices and pixels of a graphics application are. For this last reason we have chosen *pixel shader* as the default nomenclature used in this thesis.

[20] The OpenGL Architecture Review Board (ARB), an independent standards consortium formed in 1992, governs the OpenGL API. Composed of many of the industry's leading graphics vendors, the ARB defines conformance tests and approves specifications for new OpenGL features and extensions, and advances the standard.

Appendix B

An alternative to ray casting for cylinders

Inside the billboard domain of a GPU cylinder (see Section 10.4), a ray casting could be executed to draw the cylinder. However, we have tried another approach for GPU cylinder in *orthographic* projection, considering a region classification of our billboard. We classify in four regions, each one treated differently (see Figure B.1a).

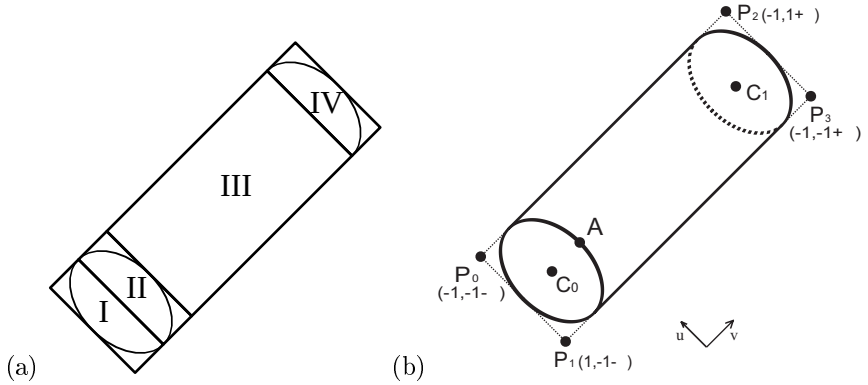


Figure B.1: (a) Billboard regions (b) Billboard 2D coordinate system

B.1 Billboard 2D coordinate system

Our billboard has a 2D coordinate system (see Figure B.1b) and the regions are defined along the $\vec{v}$ direction.

Let us define the vectors $\vec{y'} = A - C_0$ and $\vec{z'} = C_1 - C_0$. The vector $\vec{y'}$ has the same direction of vector $\vec{y}$ of our view-dependent coordinate system (see Section 10.4.1) and its norm is the cylinder radius r.

The size of each region (I, II, III and IV) depends on the norm of $\vec{y'}$ and $\vec{z'}$ when projected on the screen. Let τ denotes the ratio between those norms:

$$\tau = \frac{\mid M_p \cdot \vec{y'} \mid}{\mid M_p \cdot \vec{z'} \mid} \text{ , where } M_p \text{ is the projection matrix.}$$

Using v coordinates: $P_0.v = P_1.v = (-1 - \tau)$ and $P_2.v = P_3.v = (1 + \tau)$. The regions can be easily mapped as follows:

$$\begin{aligned} \text{region } I &\Rightarrow \text{from } (-1-\tau) \text{ to } -1 \\ \text{region } II &\Rightarrow \text{from } -1 \text{ to } (-1+\tau) \\ \text{region } III &\Rightarrow \text{from } (-1+\tau) \text{ to } 1 \\ \text{region } IV &\Rightarrow \text{from } 1 \text{ to } (1+\tau) \end{aligned}$$

For the u coordinate, we use $P_0.u = P_2.u = -1$ and $P_1.u = P_3.u = 1$ (see Figure B.1b). This coordinate is the key to define the cylinder body output color, and it will also be important for the discard, as we will see later.

B.2 Fragment computation

During the fragment stage, we have to answer to three questions for each fragment:

i. fragment discard?

ii. output color?

iii. output depth?

Each fragment has the information about its own coordinates (u_f, v_f) (in the 2D coordinate system) and the value τ.

B.2.1 Discard

The discard can only occur in the regions I and IV. This task can be transformed to a simple classification problem of inside/outside a circle.

if (region = I or IV)
  if $(u_f^2 + ((\mid v_f \mid -1) \cdot \tau)^2 > 1)$
    discard

B.2.2 Color

In our system, the final color c_f is computed based on the original color c_o, on the ambient light and the diffuse contribution of one light source.

$$\begin{aligned} c_f &= c_o \otimes (k_a + Ik_d) \text{ , where} \\ &\quad k_a \text{ is the ambient coefficient} \\ &\quad k_d \text{ is the diffuse coefficient} \\ &\quad I \text{ is the Illumination factor (source light contribution)} \end{aligned}$$

The bottom of the cylinder has a unique color (that can be pre-computed) while the body has a variation color.

if (region = I or II) and if $(u_f^2 + ((v_f + 1) \cdot \tau)^2 > 1)$
  output bottom color
else
  output body color

Bottom color :

In this case, the diffuse illumination factor I can be pre-computed (in the vertex shader).

$$I = \cos\,\theta \;=\; (-\vec{z}\cdot\vec{\ell}) \;\;,$$

where $\vec{\ell}$ is the normalized vector pointing to the light source.

Body color :

The illumination varies along the body depending on the angle of the normal at each point. The normal at any point of the body lies in the xy-plane (formed by the vectors $\vec{x}$ and $\vec{y}$ of the *view-dependent coordinate system*). The sine of the angle α between the normal and the vector $\vec{y}$ is directly mapped by the 2D coordinate u_f.

The brightest color on the body occurs in a line where the points have the normal most turned to the light direction (see Figure B.2). The illumination I_b of this line can be computed as:

$$I_b = \;\sin\,\theta$$

Let us take two points (B and C) in the body in the same xy-plane (see Figure B.2). Point B is over the brightest line of the body and C is the point where we are interested to obtain the illumination.

$$\begin{aligned} I_c &= \cos\,\gamma\; I_b \\ \cos\,\gamma &= \cos\,(\alpha-\beta) \;=\; \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta) \end{aligned}$$

Notice that some information can be pre-computed (in the vertex stage) and pass to the fragment: I_b, $\cos\,\beta$ and $\sin\,\beta$ (notice that $\sin\,\alpha = u_f$).

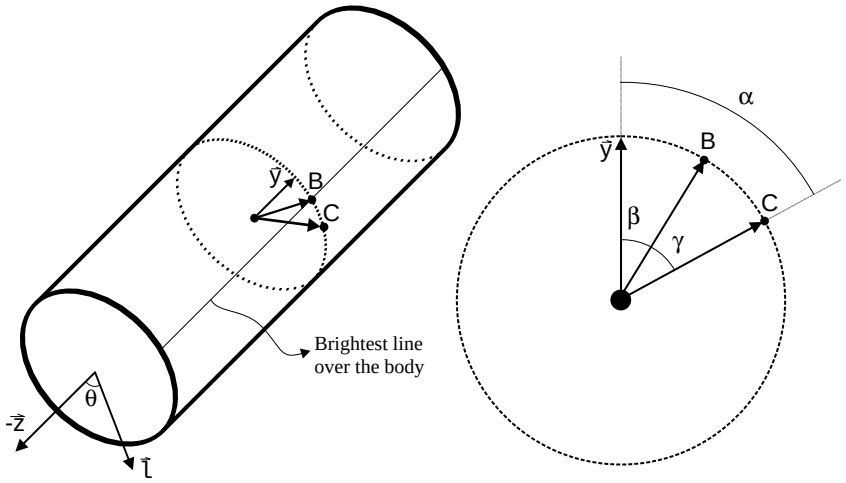


Figure B.2: The normal angles over the cylinder body

B.2.3 Depth

Let us call d_a the depth of the point A (see Figure B.1b), $d_{y'}$ the depth of the vector $\vec{y'}$ and $d_{z'}$ the depth of the vector $\vec{z'}$ (all of them are computed in the vertex stage).

$$d_{y'} = (M_p \cdot \vec{y'}) \cdot [0\,0\,1]^T$$
$$d_{z'} = (M_p \cdot \vec{z'}) \cdot [0\,0\,1]^T$$

The depth of the points along the bottom is given by:

$$d_{bottom} = d_a + d_{y'}(\frac{v_f + 1 - \tau}{\tau})$$

The depth of the points along the body is given by:

$$d_{body} = d_a + d_{y'}(1 - u_f) + d_{z'}(1 + v_f - (\tau u_f))$$

Bibliography

[ACW+99] Daniel Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Kenny Hoff, Tom Hudson, Wolfgang Stuerzlinger, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manocha. Mmr: an interactive massive model rendering system using geometric and image-based acceleration. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 199–206. ACM Press, 1999.

[AFPB] John Milligan Airey and Jr. Frederick P. Brooks. *Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations.* PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill.

[AH05] Arul Asirvatham and Hugues Hoppe. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Terrain Rendering Using GPU-Based Geometry Clipmaps, pages 27–45. Addison Wesley, 2005.

[AM04] Timo Aila and Ville Miettinen. dpvs: An occlusion culling system for massive dynamic environments. *IEEE Computer Graphics and Applications*, pages 86–97, 2004.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968.

[Bad90] Didier Badouel. An efficient ray-polygon intersection. pages 390–393, 1990.

[Bar83] Brian A. Barsky. A description and evaluation of various 3d models. In Tosiyasu L. Kunii, editor, *Computer Graphics Theory and Applications (Proceedings of InterGraphics '83)*, pages 75–95. Springer-Verlag, April 1983.

[BBB87] Richard Bartels, John Beatty, and Brian Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling.* Morgan Kaufmann Publishers, Palo Alto, CA, 1987.

[BD06] Lionel Baboud and Xavier Décoret. Rendering geometry with relief textures. In *GI '06: Proceedings of the 2006 conference on Graphics interface*, pages 195–201, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[BDST04] Chandrajit Bajaj, Peter Djeu, Vinay Siddavanahalli, and Anthony Thane. Texmol: Interactive visual exploration of large flexible multi-component molecular complexes. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 243–250, Washington, DC, USA, 2004. IEEE Computer Society.

[Bez70] Pierre E. Bezier. *Emploi des machines a commande numerique.* Masson et Cie., 1970.

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schroder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[BGR88] Brenda S. Baker, Eric Grosse, and Conor S. Rafferty. Nonobtuse triangulation of polygons. *Discrete Comput. Geom.*, 3(2):147–168, 1988.

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 286–292, August 1978.

[BN76] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19:542–546, 1976.

[BS01] Stefan Brabec and Hans-Peter Seidel. Hardware-accelerated rendering of antialiased shadows with shadow maps. In *CGI '01: Computer Graphics International 2001*, pages 209–214, Washington, DC, USA, 2001. IEEE Computer Society.

[BSNJ02] Lévy Bruno, Petitjean Sylvain, Ray Nicolas, and Maillot Jérome. Least squares conformal maps for automatic texture atlas generation. In ACM, editor, *SIGGRAPH 02, San-Antonio, Texas, USA*, Jul 2002.

[BW97] Jules Bloomenthal and Brian Wyvill, editors. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[BW03] Jiří Bittner and Peter Wonka. Visibility in computer graphics. Technical Report TR-186-2-03-03, Institut für Computergraphik und Algorithmen, 2003.

[BWPP04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum (Proceedings of Eurographics '04)*, (3), 2004.

[Cat74] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, 1974.

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 95–102, July 1987.

[CCF94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In Arie Kaufman and Wolfgang Krueger, editors, *1994 Symposium on Volume Visualization*, pages 91–98. ACM SIGGRAPH, October 1994. ISBN 0-89791-741-3.

[Ceb05] Cem Cebenoyan. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Graphics Pipeline Performance, pages 473–486. Addison Wesley, 2005.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46. Eurographics Association, 2002.

[CHH03] Nathan A. Carr, Jesse D. Hall, and John C. Hart. Gpu algorithms for radiosity and subsurface scattering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 51–59. Eurographics Association, 2003.

[CHL04] Greg Coombe, Mark J. Harris, and Anselmo Lastra. Radiosity on graphics hardware. In *Graphics Interface (to appeared)*. CIPS, Canadian Human-Computer Commnication Society, 2004.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[COCSD02] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Transaction on Visualization and Computer Graphics*, 2002.

[Coh94] Daniel Cohen. Voxel traversal along a 3D line. In Paul Heckbert, editor, *Graphics Gems IV*, pages 366–369. Academic Press, Boston, 1994.

[COM98] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-perserving simplification. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 115–122. ACM Press, 1998.

[Con85] M.L. Connolly. Molecular surface triangulation. *J. Appl. Crystallogr.*, 18:499–505, 1985.

[Coo84] Robert L. Cook. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231. ACM Press, 1984.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 137–145, New York, NY, USA, 1984. ACM Press.

[CSAD04] David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. Variational shape approximation. *ACM Transactions on Graphics. Special issue for SIGGRAPH conference*, pages 905–914, 2004.

[CSM03] David Cohen-Steiner and Jean-Marie Morvan. Restricted delaunay triangulations and normal cycle. In *SCG '03: Proceedings of the nineteenth annual symposium on Computational geometry*, pages 312–321, New York, NY, USA, 2003. ACM Press.

[CT96] Satyan Coorg and Seth Teller. Temporally coherent conservative visibility (extended abstract). In *SCG '96: Proceedings of the twelfth annual symposium on Computational geometry*, pages 78–87. ACM Press, 1996.

[CTW+04] Yanyun Chen, Xin Tong, Jiaping Wang, Stephen Lin, Baining Guo, and Heung-Yeung Shum. Shell texture functions. *ACM Trans. Graph.*, 23(3):343–353, 2004.

[dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational geometry: algorithms and applications.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[dC76] Manfredo do Carmo. *Differential Geometry of Curves and Surfaces.* Prentice Hall, 1976.

[DCM04] M. A. G. Darrin, B. G. Carkhuff, and T. S. Mehoke. Future trends in miniaturization for wireless applications. Technical report, October-December 2004.

[DDSD03] Xavier Décoret, Frédo Durand, François X. Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. *ACM Trans. Graph.*, 22(3):689–696, 2003.

[DH00] M. Doggett and J. Hirche. Adaptive view dependent tessellation of displacement maps. In *Proc. of Eurographics/SIGGRAPH workshop on graphics hardware 2000*, pages 59–66, 2000.

[DNGK97] Kristin J. Dana, Shree K. Nayar, Bram Van Ginneken, and Jan J. Koenderink. Reflectance and texture of real-world surfaces authors. In *CVPR '97: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, page 151, Washington, DC, USA, 1997. IEEE Computer Society.

[Don05] William Donnelly. *GPU Gems 2 - Per-Pixel Displacement Mapping with Distance Functions*, chapter Per-Pixel Displacement Mapping with Distance Functions, pages 123–135. Addison Wesley, 2005.

[DT81] L. J. Doctor and J. G. Torborg. Display techniques for octree-encoded objects. *IEEE Computer Graphics and Applications*, 1:29–38, July 1981.

[DvGNK99] Kristin J. Dana, Bram van Ginneken, Shree K. Nayar, and Jan J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics*, 18(1):1–34, 1999.

[Ebe01] David H. Eberly. *3D Game Engine Design*, chapter Standard 3D Objects. Morgan Kaufmann Publishers, 2001.

[EM98] Carl Erikson and Dinesh Manocha. Simplification culling of static and dynamic scene graphs. Technical report, 1998.

[EMP+02] David S. Ebert, Kenton F. Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing & Modeling: A Procedural Approach, Third Edition (The Morgan Kaufmann Series in Computer Graphics).* Morgan Kaufmann, December 2002.

[EMWVB01] Carl Erikson, Dinesh Manocha, and III William V. Baxter. Hlods for faster display of large static and dynamic environments. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 111–120. ACM Press, 2001.

[FAG83] H. Fuchs, G. D. Abram, and E. D. Grant. Near real-time shaded display of rigid objects. volume 17, pages 65–72, July 1983.

[FEF97] Andrew W. Fitzgibbon, David W. Eggert, and Robert B. Fisher. High-level CAD model acquisition from range images. *Computer-aided Design*, 29(4):321–330, 1997.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FKN80a] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. volume 14, pages 124–133, July 1980.

[FKN80b] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, New York, NY, USA, 1980. ACM Press.

[Flo97] Michael S. Floater. Parametrization and smooth approximation of surface triangulations. *Comput. Aided Geom. Des.*, 14(3):231–250, 1997.

[Flo03] Michael S. Floater. Mean value coordinates. *Comput. Aided Geom. Des.*, 20(1):19–27, 2003.

[FPE+89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. *SIGGRAPH Comput. Graph.*, 23(3):79–88, 1989.

[Fro96] Max Froumentin. *Modélisation à l'aide de surfaces quadriques pour la synthèse d'images*. PhD thesis, LIFL group, Université de Lille I, June 1996.

[FS05] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Gar99] M. Garland. Multiresolution modeling: Survey & future opportunities, 1999.

[GDH06] Stéphane Gobron, François Devillard, and Bernard Heit. Retina simulation using cellular automata and gpu programming. *Machine Vision and Applications Manuscript*, page to appear, 2006.

[GGH02] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361, New York, NY, USA, 2002. ACM Press.

[GH95] Michael Garland and Paul Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, September 1995.

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

[GH98] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 263–269. IEEE Computer Society Press, 1998.

[Gig88] Michael Gigante. Accelerated ray tracing using non-uniform grids. In *Proceedings of Ausgraph '90*, pages 157–163, 1988.

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238. ACM Press, 1993.

[Gla88] Andrew S. Glassner. Spacetime ray tracing for animation. *IEEE Comput. Graph. Appl.*, 8(2):60–70, 1988.

[GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 171–180. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[GM05] Enrico Gobbetti and Fabio Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph.*, 24(3):878–885, 2005.

[GN71] Robert A. Goldstein and Roger Nagel. 3-D visual simulation. *Simulation*, 16(1):25–31, January 1971. introduction to CSG, ray tracing, director's language.

[Gou71a] H. Gouraud. Computer display of curved surfaces. *IEEE Trans. Computers*, C-20(6):623–629, 1971.

[Gou71b] H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20(6):623–629, June 1971.

[Gre91] Stuart Green. *Parallel processing for computer graphics*. MIT Press, Cambridge, MA, USA, 1991.

[Gre01] Ned Greene. A quality knob for non-conservative culling with hierarchical z-buffering. In *Visibility (Course 30)*, ACM Siggraph Course Notes. ACM SIGGRAPH, 2001.

[GS93] IEEE Computer Graphics and Applications Staff. Backface culling snags. *IEEE Comput. Graph. Appl.*, 13(6):94–97, 1993.

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 213–222, New York, NY, USA, 1984. ACM Press.

[GV97] Jonas Gomes and Luiz Velho. *Image Processing for Computer Graphics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[GW89] Curtis F. Gerald and Patrick O. Wheatley. *Applied numerical analysis*. Addison Wesley, fourth edition, 1989.

[GWL+03] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 102–111. Eurographics Association, 2003.

[GWW86] Irene A. Gargantini, Timothy R. Walsh, and Oi-Lun Wu. Viewing transformations of voxel-based objects via linear octrees. *IEEE Computer Graphics and Applications*, 6(10):12–21, October 1986.

[Han89] Pat Hanrahan. *An introduction to ray tracing*, chapter A Survey of Ray - Surface Intersection Algorithms, pages 33–77. Academic Press Ltd., 1989.

[Har96] J. C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

[HDD+94] Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. Piecewise smooth surface reconstruction. *Computer Graphics*, 28(Annual Conference Series):295–302, 1994.

[Hec88] Paul S. Heckbert. Ray tracing jell-o brand gelatin. *Commun. ACM*, 31(2):131–134, 1988.

[HEGD04] Johannes Hirche, Alexander Ehlert, Stefan Guthe, and Michael Doggett. Hardware accelerated per-pixel displacement mapping. In *Graphics Interface*, 2004.

[HGar] Paul S. Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. Technical report, to appear.

[HL01] M. J. Harris and A. Lastra. Real-time cloud rendering. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001 Proceedings*, volume 20(3), pages 76–84. Blackwell Publishing, 2001.

[HMC+97] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 1–10. ACM Press, 1997.

[Hof02] Mauricio Hofmam. Tratamento eficiente de visibilidade através de Árvores de volumes envolventes. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro, February 2002.

[Hop96] Hugues Hoppe. Progressive meshes. *Proceedings of SIGGRAPH 96*, pages 99–108, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. *Proceedings of SIGGRAPH 97*, pages 189–198, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[Hop98] Hugues H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. *IEEE Visualization '98*, pages 35–42, October 1998. ISBN 0-8186-9176-X.

[Hop99] Hugues Hoppe. New quadric metric for simplifiying meshes with appearance attributes. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 59–66. IEEE Computer Society Press, 1999.

[HR05] John Hable and Jarek Rossignac. Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1024–1031, New York, NY, USA, 2005. ACM Press.

[HSS+05] Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler, and Markus Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. In Marc Alexa and Joe Marks, editors, *Proceedings of Eurographics '05*, volume 24, pages pp303–312. Blackwell Publishing, September 2005.

[HT04] Kai Hormann and Marco Tarini. A quadrilateral rendering primitive. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 7–14, New York, NY, USA, 2004. ACM Press.

[HTSG91a] Xiao D. He, Kenneth E. Torrance, Francois X. Sillion, and Donald P. Greenberg. A comprehensive physical model for light reflection. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 175–186, July 1991.

[HTSG91b] Xiao D. He, Kenneth E. Torrance, François X. Sillion, and Donald P. Greenberg. A comprehensive physical model for light reflection. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 175–186, New York, NY, USA, 1991. ACM Press.

[HYFK98] Jian Huang, Roni Yagel, Vassily Filippov, and Yair Kurzion. An accurate method for voxelizing polygon meshes. In *IEEE Symposium on Volume Visualization*, pages 119–126, 1998.

[JC98] Andreas Johannsen and Michael B. Carter. Clustered backface culling. *J. Graph. Tools*, 3(1):1–14, 1998.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag.

[JMLH01] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 511–518, New York, NY, USA, 2001. ACM Press.

[JW89] David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface '89*, pages 164–72, Toronto, Ontario, June 1989. Canadian Information Processing Society.

[Kaj86] James T. Kajiya. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, New York, NY, USA, 1986. ACM Press.

[KBO+99] M. Krus, P. Bourdot, A. Osorio, F. Guisnel, and G. Thibault. Adaptive tessellation of connected primitives for interactive walkthroughs in complex industrial virtual environments. In Michael Gervaut, Dieter Schmalstieg, and Axel Hildebrand, editors, *Virtual Environments '99. Proceedings of the Eurographics Workshop in Vienna, Austria*, pages 23–32, 1999.

[KBR04] J. KESSENICH, D. BALDWIN, and R. ROST. *OpenGL Shading Language*, chapter Polynomial Texture Mapping With BRDF Data, pages 252–261. Addison Wesley, 2004.

[KK86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 269–278. ACM Press, 1986.

[KK89] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 271–280, New York, NY, USA, 1989. ACM Press.

[KKW05] Polina Kondratieva, Jens Krüger, and Rüdiger Westermann. The application of gpu particle tracing to diffusion tensor field visualization. In *Proceedings IEEE Visualization 2005*, 2005.

[KMGL97] Subodh Kumar, Dinesh Manocha, William Garrett, and Ming Lin. Back-face computation of polygon clusters. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 487–488. ACM Press, 1997.

[Kry05] Yuri Kryachko. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Using Vertex Texture Displacement for Realistic Water Rendering, pages 283–294. Addison Wesley, 2005.

[KW03a] Jens Krueger and Ruediger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[KW03b] Jens Krueger and Ruediger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.

[LaM95] Andre LaMothe. *Black Art of 3d Programming*. FT Prentice Hall, 1995.

[LAP96] Hervé LAPORTE. *Etude Logicielle et Materielle d'un Système de Visualisation Temps-Réel Basé sur la Quadrique*. PhD thesis, LIFL group, Université de Lille I, June 1996.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[LDS02] Hendrik P. A. Lensch, Katja Daubert, and Hans-Peter Seidel. Interactive semi-transparent volumetric textures. In Günther Greiner, Heinrich Niemann, Thomas Ertl, Bernd Girod, and Hans-Peter Seidel, editors, *Proceedings of Vision, Modeling, and Visualization VMV 2002*, pages 505–512, Erlangen, Germany, 2002. Akademische Verlagsgesellschaft Aka GmbH.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 199–208. ACM Press/Addison-Wesley Publishing Co., 1997.

[Len03] Eric Lengyel. *The OpenGL Extensions Guide*. Delmar Thomson Learning, 07 2003.

[Lev90] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[LG95] David Luebke and Chris Georges. Portals and mirrors: simple, fast evaluation of potentially visible sets. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 105–ff. ACM Press, 1995.

[LMM97] G. Lukacs, A.D. Marshall, and R. R. Martin. Geometric least-squares fitting of spheres, cylinders, cones and tori, May 1997.

[LNFC95] H. Laporte, Eric Nyiri, Max Froumentin, and Christophe Chaillou. A graphics system based on quadrics. *Computers & Graphics*, 19(2):251–260, 1995.

[LS97] Jed Lengyel and John Snyder. Rendering with coherent layers. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 233–242, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[LSSS04] Xinguo Liu, Peter P. Sloan, Heung Y. Shum, and John Snyder. All-frequency precomputed radiance transfer for glossy objects. In Alexander Keller and Henrik W. Jensen, editors, *Eurographics Symposium on Rendering*, pages 337–344, Norrköping, Sweden, 2004. Eurographics Association.

[LYS01] Xinguo Liu, Yizhou Yu, and Heung-Yeung Shum. Synthesizing bidirectional texture functions for real-world surfaces. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 97–106, New York, NY, USA, 2001. ACM Press.

[MA03] Kenneth Moreland and Edward Angel. The fft on a gpu. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119. Eurographics Association, 2003.

[MAG68] MAGI. 3-D simulated graphics offered by service bureau. *Datamation*, 14:69, February 1968.

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

[MD98] T. McReynodls and D.Blythe. Advanced graphics programming techniques using opengl. ACM Siggraph Course Notes. ACM SIGGRAPH, 1998.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[MGW01] Tom Malzbender, Dan Gelb, and Hans Wolters. Polynomial texture maps. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 519–528, New York, NY, USA, 2001. ACM Press.

[MH99a] Tomas Müller and Eric Haines. *Real-time rendering*. A. K. Peters, Ltd., 1999.

[MH99b] Tomas Müller and Eric Haines. *Real-time rendering*, chapter Occlusion Culling. A. K. Peters, Ltd., 1999.

[MN98] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. In George Drettakis and Nelson Max, editors, *Eurographics Rendering Workshop 1998*, pages 157–168, New York City, NY, July 1998. Eurographics, Springer Wein. ISBN 3-211-83213-0.

[MPBM03] Wojciech Matusik, Hanspeter Pfister, Matthew Brand, and Leonard McMillan. Efficient isotropic brdf measurement. In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering*, pages 241–247, Aire-la-Ville, Switzerland, 2003. Eurographics Association.

[MS95] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *Symposium on Interactive 3D Graphics*, pages 95–102, 211, 1995.

[MSE+06] Dorit Merhof, Markus Sonntag, Frank Enders, Christopher Nimsky, and Guenther Greiner. Hybrid visualization for white matter tracts using triangle strips and point sprites. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1181–1188, 2006. Member-Peter Hastreiter.

[MWLT00] Stephen R. Marschner, Stephen H. Westin, Eric P. F. Lafortune, and Kenneth E. Torrance. Image-based bidirectional reflectance distribution function measurement. *Applied Optics-OT*, 39(16):2592–2600, June 2000.

[NHHM80] B. Nourse, D. G. Hakala, R. Hillyard, and P. Malraison. Natural quadrics in mechanical design. *Autofact West*, 1:363–378, 1980.

[NNS72] Martin E. Newell, R. G. Newell, and T. L. Sancha. A new approach to the shaded picture problem. In *Proc. ACM Nat. Conf.*, page 443. 1972.

[NRH+77] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Geometric considerations and nomenclature for reflectance. Monograph 161, National Bureau of Standards (US), October 1977.

[OBA+03] Yutaka Ohtake, Alexander Belyaev, Marc Alexa, Greg Turk, and Hans-Peter Seidel. Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3):463–470, 2003.

[OBA05] Yutaka Ohtake, Alexander Belyaev, and Marc Alexa. Sparse low-degree implicit surfaces with applications to high quality rendering, feature extraction, and smoothing. In *SGP '05: Proceedings of the 2005 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. Eurographics Association, 2005.

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 359–368, July 2000.

[OP05] Manuel M. Oliveira and Fabio Policarpo. An efficient representation for surface details, 2005.

[PBFJ05] Serban D. Porumbescu, Brian Budge, Louis Feng, and Kenneth I. Joy. Shell maps. *ACM Trans. Graph.*, 24(3):626–633, 2005.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[PDC+03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[PDG05] Damien Porquet, Jean-Michel Dischler, and Djamchid Ghazanfarpour. Real-time high-quality view-dependent texture mapping using per-pixel visibility. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 213–220, New York, NY, USA, 2005. ACM Press.

[Pet02] Sylvain Petitjean. A survey of methods for recovering quadrics in triangle meshes. *ACM Comput. Surv.*, 34(2):211–262, 2002.

[Pho75] Bui-T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[PKZ04] Jianbo Peng, Daniel Kristjansson, and Denis Zorin. Interactive modeling of topologically complex geometric detail. *ACM Trans. Graph.*, 23(3):635–643, 2004.

[PM86] B. Parvin and G. Medioni. Segmentation of range images into planar surfaces by split and merge. In *CVPR '86: In Proc. of International Conference on Computer Vision and Pattern Recognition (CVPR '86)*, pages 415–417, 1986.

[POC05a] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 155–162, New York, NY, USA, 2005. ACM Press.

[POC05b] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. *ACM Trans. Graph.*, 24(3):935–935, 2005.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing (2nd ed.)*. Cambridge University Press, Cambridge, 1992. ISBN 0-521-43108-5.

[RdFV06] Fabiano Romeiro, Luiz Henrique de Figueiredo, and Luiz Velho. Hardware-assisted rendering of csg models. In *Proceedings of SIBGRAPI 2006 - XIX Brazilian Symposium on Computer Graphics and Image Processing*, pages –, Manaus, October 2006. SBC - Sociedade Brasileira de Computacao, IEEE Press.

[Reg02] Ashu Rege. Occlusion (hp and nv extensions), 2002. GDC (Game Developers Conference). NVidia. http://developer.nvidia.com/object/gdc_occlusion.html.

[RLL+05] Nicolas Ray, Wan Chiu Li, Bruno Lévy, Alla Sheffer, and Pierre Alliez. Periodic global parameterization. In *Tech report*, January 2005.

[Roc89] A. P. Rockwood. The displacement method for implicit blending surfaces in solid models. *ACM Trans. Graph.*, 8(4):279–297, 1989.

[RRPB05] Luciano P. Reis, Alberto Barbosa Raposo, Jean-Claude Paul, and Fabien Bosquet. An architecture for collaborative geomodeling. In *CRIWG*, pages 121–136, 2005.

[RV82] A. A. G. Requicha and H. B. Voelcker. Solid modeling: a historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, 2:9–22, March 1982.

[Sam89] Hanan Samet. Implementing ray tracing with octrees and neighbor finding. *Computers and Graphics*, 13(4):445–60, 1989.

[Sch97] Gernot Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques '97*, pages 151–162, London, UK, 1997. Springer-Verlag.

[SCT+05] Ying Song, Yanyun Chen, Xin Tong, Stephen Lin, Jiaoying Shi, Baining Guo, and Heung-Yeung Shum. "hell radiance texture functions. In *PG '05: Proceedings of the 9th Pacific Conference on Computer Graphics and Applications*. IEEE Computer Society, 2005.

[SDB97] François Sillion, George Drettakis, and Benoit Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. In D. Fellner and L. Szirmay-Kalos, editors, *Computer Graphics Forum (Proc. of Eurographics '97)*, volume 16, pages 207–218, Sep 1997.

[Sek04] Dean Sekulic. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter Efficient Occlusion Culling. Pearson Higher Education, 2004.

[SGC97] Flávio Szenberg, Marcelo Gattass, and Paulo Cezar Carvalho. Um algoritmo para a visualização de terrenos com objetos. Master's thesis, Pontifícia Universidade Católica (PUC-Rio), Rio de Janeiro, Brasil, 1997.

[SGS05] Carsten Stoll, Stefan Gumhold, and Hans-Peter Seidel. Visualization with stylized line primitives. In Cláudio T. Silva, Eduard Gröller, and Holly Rushmeier, editors, *IEEE Visualization 2005 (VIS 2005)*, pages 695–702, Minneapolis, USA, 2005. IEEE.

[SHSS00] Marc Stamminger, Joerg Haber, Hartmut Schirmacher, and Hans-Peter Seidel. Walkthroughs with corrective texturing. In B. Peroche and H. Rushmeier, editors, *Rendering Techniques 2000 (Proceedings of the Eleventh Eurographics Workshop on Rendering)*, pages 377–388, New York, NY, 2000. Springer Wien.

[SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics. Special issue for SIGGRAPH conference*, 21(3):527–536, 2002.

[SLSS03] Peter-Pike Sloan, Xinguo Liu, Heung-Yeung Shum, and John Snyder. Bi-scale radiance transfer. *ACM Transactions on Graphics. Special issue for SIGGRAPH conference*, 22(3):370–375, 2003.

[Sub90] K. R. Subramanian. *Adapting Search Structures to Scene Characteristics for Ray Tracing*. Ph.d. thesis, Department of Computer Sciences, University of Texas at Austin, December 1990.

[SWG+03] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 146–155, Aire-la-Ville, Switzerland, 2003. Eurographics Association.

[TCS03] Marco Tarini, Paolo Cignoni, and Roberto Scopigno. Visibility based methods and assessment for detail-recovery. In *Proceedings of IEEE Visualization '03*. IEEE, 2003.

[Tel92] Seth Jared Teller. *Visibility computations in densely occluded polyhedral environments*. PhD thesis, 1992.

[TGV01] Rodrigo Toledo, Marcelo Gattass, and Luiz Velho. Qlod: A data structure for interactive terrain visualization. Technical report, VISGRAF Laboratory TR-2001-13, IMPA, 2001.

[TLP03] Rodrigo Toledo, Bruno Lévy, and Jean Claude Paul. Thesis progress report 2003, 2003.

[Tol00] Rodrigo Toledo. Quadlod: Uma estrutura para a visualização interativa de terrenos. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro, April 2000.

[Tol04] Rodrigo Toledo. Extending the graphic pipeline with new gpu-accelerated primitives. VISGRAF Seminar. IMPA, Rio de Janeiro, Brasil, May 2004.

[Tot85] Daniel L. Toth. On ray tracing parametric surfaces. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 171–179, New York, NY, USA, 1985. ACM Press.

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

[UBBS01] Thomas Ullmann, Daniel Beier, Beat Bruderlin, and Alexander Schmidt. Adaptive progressive vertex tracing in distributed environments. In *PG '01: Proceedings of the 9th Pacific Conference on Computer Graphics and Applications*, page 285, Washington, DC, USA, 2001. IEEE Computer Society.

[VdFG98] L. Velho, L. H. de Figueiredo, and J. Anthony Gomes. *Implicit Objects in Computer Graphics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[VT04] M. Alex O. Vasilescu and Demetri Terzopoulos. Tensortextures: multilinear image-based rendering. *ACM Trans. Graph.*, 23(3):336–342, 2004.

[vW85] Jarke J. van Wijk. Ray tracing objects defined by sweeping a sphere. *Computers & Graphics*, 9(3):283–290, 1985.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[Wat00] Alan Watt. *3D Computer Graphics*. Addison-Wesley, third edition, 2000.

[WDG02] Bruce Walter, George Drettakis, and Donald Greenberg. Enhancing and optimizing the render cache. In Paul Debevec and Simon Gibson, editors, *Thirteenth Eurographics Workshop on Rendering (2002)*. Eurographics, ACM Press, June 2002.

[WDP99] Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. In D. Lischinski and G.W. Larson, editors, *Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)*, volume 10, pages 235–246, New York, NY, Jun 1999. Springer-Verlag/Wien.

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 169–177, New York, NY, USA, 1998. ACM Press.

[Wei05] Eric W. Weisstein. Surface, (accessed in 01/2005). From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/Surface.html.

[WFAR99] N. Werghi, R. Fisher, A. Ashbrook, and C. Robertson. Faithful recovering of quadric surfaces from 3d range data, 1999.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 270–274, August 1978.

[Wil02] Andrew Thomas Wilson. *Spatially encoded image-space simplifications for interactive walkthrough*. PhD thesis, 2002. Director-Dinesh Manocha.

[WK05] Jianhua Wu and Leif Kobbelt. Structure recovery via hybrid variational surface approximation. *Computer Graphics Forum (Proceedings of Eurographics '04)*, (3):277–284, 2005.

[WM03] Andrew Wilson and Dinesh Manocha. Simplifying complex environments using incremental textured depth meshes. *ACM Trans. Graph.*, 22(3):678–688, 2003.

[Woo90] Andrew Woo. Fast ray-polygon intersection. page 394, 1990.

[WPS+03] Ingo Wald, Timothy J. Purcell, Joerg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.

[WS01] I. Wald and P. Slusallek. State-of-the-art in interactive raytracing, 2001.

[WSE99] Ruediger Westermann, Ove Sommer, and Thomas Ertl. Decoupling polygon rendering from geometry using rasterization hardware. In D. Lischinski and G. W. Larson, editors, *Rendering Techniques '99*, pages 45–56. Springer-Verlag, Wien, New York, 1999.

[WSS05] Sven Woop, Jörg Schmittler, and Philipp Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, 2005.

[WTL+04] Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. Generalized displacement maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques*, 2004.

[WWT+03] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Transactions on Graphics*, 22(3):334–339, July 2003.

[XESV97] Julie C. Xia, Jihad El-Sana, and Amitabh Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, 1997.

[YAH96] Cai Y.Y., Nee A.Y.C., and Loh H.T. Geometric feature detection for reverse engineering using range imaging. *Journal of Visual Communication and Image Representation*, 7:205–216, September 1996.

[YSGM04] Sung-Eui Yoon, Brian Salomon, Russell Gayle, and Dinesh Manocha. Quick-vdr: Interactive view-dependent rendering of massive models. In *VIS '04: Proceedings of the IEEE Visualization 2004 (VIS'04)*, pages 131–138. IEEE Computer Society, 2004.

[ZKEH97] Hansong Zhang and III Kenneth E. Hoff. Fast backface culling using normal masks. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 103–ff. ACM Press, 1997.

[ZMHKEH97] Hansong Zhang, Dinesh Manocha, Tom Hudson, and III Kenneth E. Hoff. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 77–88. ACM Press/Addison-Wesley Publishing Co., 1997.