# Real-time geometry synthesis

Matthias Holländer

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2013-ENST-0009

EDITE - ED 130

## Doctorat ParisTech

# T H È S E

**pour obtenir le grade de docteur délivré par**

## TELECOM ParisTech

### Spécialité « SIGNAL et IMAGES »

*présentée et soutenue publiquement par*

**Matthias HOLLÄNDER**

le 7 mars 2013

# Synthèse Géométrique Temps-Réel

Directeur de thèse : **Tamy BOUBEKEUR**

**Jury**

| | |
|---|---|
| **M. Mathias PAULIN**, Professeur, IRIT, Université Paul Sabatier Toulouse | Rapporteur |
| **M. Carsten DACHSBACHER**, Professeur, Karlsruher Institut für Technologie | Rapporteur |
| **M. Sylvain LEFEBVRE**, Chargé de Recherche, INRIA Nancy | Examinateur |
| **M. Henri MAÎTRE**, Professeur, Telecom ParisTech | Examinateur |
| **M. Tamy BOUBEKEUR**, Maître de conférence (HDR), CNRS-LTCI, Telecom ParisTech | Directeur |

**TELECOM ParisTech**
école de l'Institut Mines-Télécom - membre de ParisTech

T H È S E

To my family,
to the friends I gained,
and to those I lost on the way

# ABSTRACT

Real-time geometry synthesis is an emerging topic in computer graphics. Today's interactive 3D applications have to face a variety of challenges to fulfill the consumer's request for more realism and high quality images. Often, visual effects and quality known from offline-rendered feature films or special effects in movie productions are the ultimate goal but hard to achieve in real time. This thesis offers real-time solutions by exploiting the Graphics Processing Unit (GPU) and efficient geometry processing. In particular, a variety of topics related to classical fields in computer graphics such as subdivision surfaces, global illumination and anti-aliasing are discussed and new approaches and techniques are presented.

# SUMMARY

Today's real-time interactive 3D applications strive to simulate virtual worlds that are rich in geometric detail. When it comes to rendering these worlds, efficient techniques that can handle vast amounts of geometry and produce high quality images in fractions of a second are a necessity to ensure a good user experience.

The core idea of this thesis is to use geometry synthesis, processing and analysis to reach real-time performance in such demanding high-speed applications. We will exploit parallel algorithms running on current graphics hardware architectures to achieve this goal. In particular, we present classical topics in computer graphics, namely subdivision surfaces, global illumination algorithms and anti-aliasing techniques together with our contributions that are targeting real-time contexts. All presented techniques can be tuned to trade quality for speed by adapting the level-of-detail of the evaluation to a per polygon, per view or per pixel level.

Subdivision is a process that recursively refines the input line or mesh and converges, in the limit, to a smooth curve or surface. It is a popular means for creating high resolution smooth surfaces that can be controlled easily and intuitively by manipulating the underlying coarse representation and many content creation tools, such as 3D modeling packages, already support them. However, the recursive subdivision process is difficult to map to the parallel architecture of current consumer graphics hardware or slow when performed on the CPU. A straight forward implementation on the Graphics Processing Units (GPUs) using an iterative, i. e. multi-pass, approach might be tempting but requires either high amounts of memory or does not exploit the full capabilities of the graphics card. Instead, we suggest a precomputation step that is largely independent of the input geometry but which allows dynamic, flexible and fast upsampling based on a subdivision algorithm at runtime. Our approach is especially suitable for character animation as found in computer games or high quality rendering packages, can handle adaptive levels-of-detail and is easy to combine with existing pipelines and hardware/software tessellation units. We demonstrate our method for a variety of meshes and compare them to subdivision substitutes.

In the process of creating a synthetic image that the user perceives as realistic, lighting effects play an important role. Especially the simulation of global illumination, which goes beyond direct lighting effects, is a challenging task. Here, the light transport in the scene is simulated or approximated but non-local computations complicate a real-time evaluation. To accelerate visibility and/or (ir)radiance queries, the scene is organized hierarchically in a spatial tree comprised of bounding volume elements. Rasterization-based global illumination techniques often make use of a point sampled representation of the scene that is

organized in such a tree. We present a fast and parallel method to compute a multi-cut through this tree where each cut corresponds to a level-of-detail with respect to a certain point of view. The efficiency of our method is demonstrated for a variety of well-known algorithms and can be improved even further by making use of an incremental scheme or by amortizing the cost of the multi-cut computation across several frames. Further, we will discuss related memory issues and present solutions that still run in real time despite some additional overhead. Besides purely GPU-based implementations, we briefly investigate a hybrid, i. e. CPU/GPU, approach to harness the computational power of the full machine.

When rendering high quality images composed of millions of pixels in real time, an immense amount of surface points that are visible on screen need to be shaded. To determine and shade the set of visible surface points that are possibly influenced by a large number of light sources, real-time applications often employ a deferred shading technique, where visibility is solved first and subsequent shading is restricted to the visible portion on screen. To accomplish that, a geometry buffer that stores the attributes of the front-most surface is obtained in a first step which can be seen as an intermediate view-dependent scene representation. A single pixel within this buffer, however, corresponds only to a single surface sample. This leads to aliasing artifacts when the buffer is used for shading but can be improved using supersampling. Supersampling increases the resolution during rendering, thus, more samples per pixel are available that are later combined to a single value using a box filter. This largely reduces aliasing artifacts but increases the amount of samples that need to be shaded to determine the shading of the corresponding pixel. Instead of shading all samples in a brute-force approach, we suggest an adaptive scheme that is sensitive to geometric discontinuities in the geometry buffer and aware of possible shading discontinuities in the final image. This allows us to concentrate the workload where needed while keeping the number of samples to be shaded low for visually continuous areas. As a further improvement, we demonstrate how to integrate anisotropic texture filtering to produce images of even higher quality.

CHAPTER 0

# RÉSUMÉ

La géométrie numérique en temps réel est un domaîne de recherches émergent en informatique graphique. Pour pouvoir générer des images photo-réalistes de haute définition, beaucoup d'applications requièrent des méthodes souvent prohibitives financièrement et relativement lentes. Parmi ces applications, on peut citer la pré-visualisation d'architectures, la réalisation de films d'animation, la création de publicités ou d'effets spéciaux pour les films dits *réalistes*. Dans ces cas, il est souvent nécessaire d'utiliser conjointement beaucoup d'ordinateurs possédant eux-mêmes plusieurs unités graphiques (*"Graphics Processing Units"* – GPUs). Cependant, certaines applications dites *temps-réel* ne peuvent s'accomoder de telles techniques, car elles requièrent de pouvoir générer plus de 30 images par seconde pour offrir un confort d'utilisation et une intéraction avec des mondes virtuels 3D riches et réalistes.

L'idée principale de cette thèse est d'utiliser la synthèse de géométrie, la géométrie numérique et l'analyse géométrique pour répondre à des problèmes classiques en informatique graphique, telle que la génération de surfaces de subdivision, l'illumination globale ou encore l'anti-aliasing dans des contextes d'intéraction temps-réel. Nous présentons de nouveaux algorithmes adaptés aux architectures matérielles courantes pour atteindre ce but.

## 0.1 Introduction

Pour représenter une entité géométrique dans un monde 3D, on utilise souvent des surfaces (variétés de dimension 2) triangulaires. Une telle représentation peut être convertie facilement en une représentation pixellique qui peut être affichée directement sur un écran ou stockée dans une mémoire tampon (*"buffer"*), à l'aide d'un procédé appelé rastérisation qui est utilisé dans la plupart des applications temps-réel. Pour cela, la primitive géométrique (un point, une ligne, ou un polygone) est projectée sur le plan image d'une caméra virtuelle, et les pixels correspondants sont trouvées. Pour définir une couleur propre à chaque pixel, mais également, depuis peu, pour des applications générales en calcul géométrique, la machinerie graphique programmable proposée par des interfaces

de programmation (*"Application Programming Interfaces"* – APIs) telles que Direct3D ou OpenGL peut être utilisée.

La primitive géométrique est traitée à chaque étape du processus de manière différente, et peut même être stockée dans une mémoire tampon avant la rastérisation (*"stream-out"*). Par exemple, un triangle peut être traitée par le vertex shader, être ensuite tesselé et envoyé vers une mémoire tampon.

L'écriture de programmes GPU rapides et efficaces nécessite la compréhension du matériel graphique utilisé. Dans la plupart des calculs liés au rendu, les données d'entrée sont découpée ou ré-arrangée pour permettre une utilisation aussi compatible que possible avec le matériel graphique. Par exemple, elles peuvent être segmentées en différentes parties chacune traitée par un thread sur le processeur graphique. Des problèmes plus compliqués peuvent être résolus en utilisant des groupes de threads qui coopèrent et communiquent via la mémoire partagée ou des opérations atomiques.

## 0.2 Contribution

Les algorithmes présentés dans cette thèse utilisent des représentations géométriques différentes et adaptées au problème particulier considéré, et fournissent des représentations de niveaux-de-détail variés. Plus précisemment, nous faisons les contributions suivantes:

1. Une nouvelles méthode GPU pour la synthèse de surfaces lisses d'ordre de régularité élevés connues sous le nom de surfaces de subdivision, ainsi qu'un noyau GPU utilisant des tables de fonctions de bases précalculées compatible avec un large spectre de schémas interpolants ou approximants de subdivision pour les maillages animés composés de triangles ou de quadrangles. De plus, notre approche est compatible avec les méthodes de tessellation adaptatives et permet l'utilisation des unités de tessellation récentes (*"hardware tessellation units"*).

2. Un algorithme permettant le calcul de plusieurs coupes en parallèle à travers des structures de niveaux-de-détail (*"Level-of-Detail"* – LoD) hiérarchiques telles que des arbres de données. Cela est particulièrement utile pour l'accélération de techniques de calcul de l'illumination globale qui reposent sur ces structures pour les calculs dépendant de la visibilité, et de fait permet d'utiliser ces techniques en temps-réel.

3. Une nouvelle approche pour le rendu anti-aliasé dans les scenarios de rendu différé (*"deferred shading"*) basée sur une métrique utilisant une représentation géométrique intermédiaire de la scène en entrée et qui guide le rendu de l'image finale. Cette méthode permet d'obtenir un bon compromis entre la rapidité du rendu et la qualité de l'image finale.

## 0.3 Surfaces de subdivision en temps-réel

Les schémas de subdivision sont un moyen puissant de génération de surfaces lisses à partir de maillages ayant une topologie arbitraire [Cat+78; Doo+78]. Typiquement, ces maillages polygonaux (le domaîne, ou le maillage de base) sont manipulés à basse résolution, et les schémas de subdivision sont utilisés en tant que schémas de sur-échantillonage pour créer des maillages haute-résolution qui interpolent ou approximent les sommets du maillage de base. Pour les schémas populaires qui sont dérivés des surfaces de type spline, les surfaces créées ont une régularité d'ordre 2 ($C^2$) partout, à l'exception des sommets dits extraordinaires où la surface n'est que $C^1$. Par exemple, le schéma de subdivision de Loop [Loo87] pour les maillages triangulaires est basé sur des box-splines triangulaires quartiques. Pour évaluer un point sur la surface de subdivision, il est possible d'appliquer de manière récursive les masques de subdivision qui décrivent le schéma à l'aide d'opérations locales de raffinement du maillage et de filtrage. Ce procédé est en général réalisé sur le CPU car une implémentation directe sur la GPU requiert une large occupation mémoire ainsi que l'utilisation de multiples passes de rendu [Shi+05]. La surface peut également être évaluée de manière directe [Sta98], mais il est difficile de le faire sur la GPU. Une troisième option est d'utiliser des tables de fonctions de bases précalculées [Bol+02]. Dans la suite, on présente un bref résumé de l'état de l'art, puis l'approche développée qui combine de manière efficace les tables de fonctions de bases et les unités logicielles et matérielles de tessellation.

### 0.3.1 Surfaces de subdivision

Une surface de subdivision [Zor+00] est une surface paramétrique lisse définie par un maillage polygonal de base et d'un schéma de subdivision. Les schémas peuvent être caractérisés par le type de polygones utilisés dans le maillage de base (triangles, quadrangles ou les deux), les conventions de raffinement (scindement de face/sommet), le type de surface (schéma interpolant ou approximant les sommets du maillage de base), ainsi que la régularité de la surface de subdivision obtenue (la plupart des schémas sont au moins $C^1$). A chaque schéma correspond un ensemble de *masques* pour les différents cas topologiques pouvant intervenir, comme les sommets intérieurs, de frontières ou de coins, et souvent les cas permettant la création d'arêtes saillantes ou semi-saillantes dans la surface finale.

Dans la suite, nous utilisons les notations suivantes: Le maillage de base est noté $\mathbf{M}^0$ et $S$ dénote un opérateur local et compact de subdivision donné. $\mathbf{M}^k = S^k(\mathbf{M}^0)$ est le maillage de base modifié par $k$ étapes de subdivision. Chaque maillage $\mathbf{M}^i$ est composé d'un ensemble de sommets $V^i = \{v^i_j\}$ et d'un ensemble de faces $F^i = \{f^i_j\}$. Un point sur la surface peut être évalué à l'aide d'une combinaison linéaire des sommets $v^0_i$ en utilisant les fonctions de base comme poids de pondération [Bol+02]:

$$f(u,v) = \sum_i B^i(u,v) v^0_i$$

Les fonctions de base $B^i(u, v)$ ne dépendant pas des positions des sommets $v_i^0$, et peuvent de fait être précalculées. Cependant, elles dépendent du niveau de subdivision désiré et de la configuration topologique (intérieur, frontière) comme de la valence des sommets considérés. Une fois calculées, elles peuvent être réutilisées pour les faces ayant une configuration topologique similaire, par exemple pour différents maillages de la scène 3D.

### 0.3.2 Tessellation en temps-réel

Les processeurs graphiques (GPUs) d'aujourd'hui offrent une solution viable pour la tessellation adaptative en temps-réel. Pour cela, le maillage d'entrée est sur-échantilloné, ce qui veut dire que les polygones d'entrée sont remplacés par un grand nombre de polygones et les sommets du maillage raffiné sont déplacés selon une fonction définie sur le maillage de base. De manière optionnelle, le signal surfacique ainsi défini peut être déformé, en utilisant par exemple des cartes de déplacement (*"displacement mapping"*). Dans les applications temps-réel, le processus de synthèse et de déplacement est répété à chaque trame, et l'application n'a besoin d'intéragir qu'avec le maillage de base qui peut être dynamique. Il est alors possible de sur-échantilloner le maillage sur la GPU à l'aide des unités matérielles de tessellation ou via des fonctionnalités similaires (par exemple: *"instancing"*) [Bou+05a; Bou+08a]. Pour diminuer le coût d'évaluation du point sur la surface haute-résolution, des modèles alternatifs de création de surfaces courbes ont été développés. Ces alternatives aux surfaces de subdivision permettent d'obtenir une régularité visuelle sans requérir une évaluation récursive, mais donnent des surfaces moins régulières sur le plan strict de la géométrie. Par exemple, PN-Triangles [Vla+01] ou la tessellation de Phong [Bou+08b] sont des opérateurs de tessellation purement locaux (c'est-à-dire qu'ils ne dépendent que de l'information locale du triangle à sur-échantilloner, comme les positions et les normales de ses trois sommets). Pour le rendu, un champ de normales synthétique peut être généré en interpolant linéairement [Pho73] ou de manière quadratique [VO+97] les normales des sommets, indépendamment des valeurs différentielles réelles de la surface. De manière alternative, on peut utiliser des patches de Bézier bi-cubique [Loo+08] ou des patches de Gregory [Loo+09]. Un bon compromis est de combiner une étape de subdivision et l'usage de patches quadratiques pour approximer la surface de subdivision [Bou+07a]. Bien que toutes ces méthodes peuvent être utilisées pour produire des versions haute-résolution régulières du maillage d'entrée, elles nécessitent parfois d'être adaptées à l'environnement logiciel existant, mais surtout, elles ne permettent pas d'obtenir la qualité des surfaces de subdivision, comme nous le démontrerons plus tard dans le texte.

### 0.3.3 Notre noyau GPU de subdivision

Dans la suite nous résumons brièvement notre approche pour produire des maillages de subdivision dynamiques à la volée (Fig. 0.1). Notre noyau est compatible avec tous les schémas de subdivision classiques (c'est-à-dire stationnaires, locaux

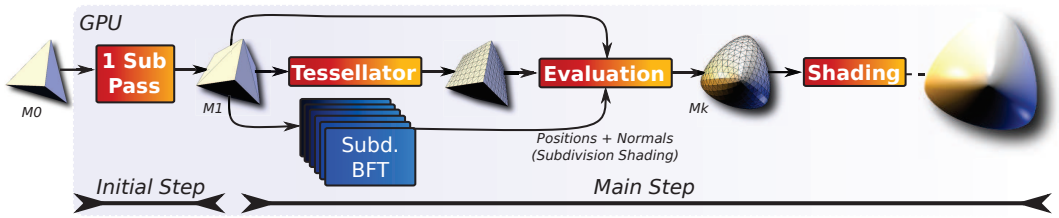



**Figure 0.1:** Notre processus de subdivision GPU de maillages.

et à support compact) pour lesquels des tables de fonctions de bases peuvent être générées. Il permet d'obtenir des performances temps-réel et est compatible avec les données d'entrée dynamiques (Fig. 0.2) telles que les données générées par un système d'animation de personnages ou un système de simulations physiques. Toute l'étape de sur-échantillonage est réalisée sur la GPU, et de fait le CPU est libéré de ce poids et peut être utilisé pour d'autres tâches.

Notre noyau utilise un nombre fixe de passes, qui exploitent toutes le parallélisme de la GPU, et qui requièrent uniquement les tables de fonctions de bases (*"Basis Function Tables"* – BFTs) et une unité de tessellation logicielle ou matérielle.

Nos trois composantes principales sont:

- Une première étape de subdivision sur la GPU pour isoler les sommets extraordinaires, après laquelle $\mathbf{M}^1$ est sur la mémoire de la GPU et chaque triangle de $\mathbf{M}^1$ contient au plus un sommet extraordinaire. Les normales de $\mathbf{M}^1$ sont également évaluées à cette étape, pour la génération postérieure d'un champ de normales régulier à l'aide du rendu de subdivision (*"Subdivision Shading"*). Il est à noter que les normales des sommets d'un triangle de $\mathbf{M}^1$ font partie de la même hémisphère, ce qui améliore la qualité du rendu lorsque le rendu de subdivision est utilisé.

- Une étape de tessellation (uniforme ou adaptative) est réalisée sur $\mathbf{M}^1$, pour créer directement une géométrie de la surface de subdivision à un niveau $k$. Cette étape peut utiliser soit des implémentations GPU, soit les unités matérielles de tessellation modernes.

- Finalement, un ensemble de tables de fonctions de bases $\{B^i\}$ est stocké sur la GPU. Au moment du rendu, les positions et normales des sommets de $\mathbf{M}^k$ sont calculées comme combinaisons linéaires des sommets de $\mathbf{M}^1$ (créé à la première étape), en utilisant les poids stockés dans $\{B^i\}$. Les normales sont calculées à l'aide d'une combinaison sur la sphère de Gauss est normalisée après [Ale+08].

On utilise des tables de fonctions de bases calculéees pour des configurations topologiques ne possédant au plus qu'un seul sommet extraordinaire, ce qui réduit

**Figure 0.2:** Gauche: maillage basse-résolution d'entrée. Milieu et droite: surface de subdivision adaptative créée en temps-réel avec notre approche.

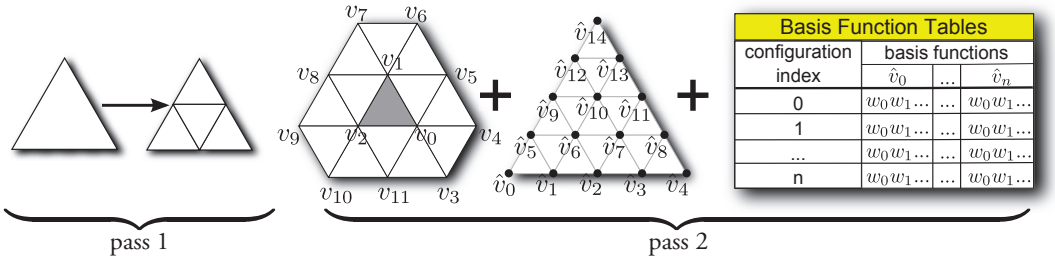



**Figure 0.3:** Un triangle d'entrée $t^0$ est subdivisé une fois sur la GPU, produisant quatre triangles $t_i^1$ (première passe). Chaque triangle $t_i^1$ est tessellé (deuxième passe, milieu) – par exemple à l'aide d'une unité matérielle de tessellation ou par instanciation d'un schéma de raffinement – et les fonctions de bases correspondantes sont choisies. Finalement, la position de chaque sommet $\hat{v}_i$ du maillage final est calculée comme une combinaison linéaire des sommets $v_i$ de $\mathbf{M}^1$.

le nombre de configurations à créer, donc aussi la mémoire. Ces tables sont compatibles avec les maillages ayant leurs sommets extraordinaires séparés, ce qui est toujours le cas après une étape de subdivision et qui motive notre stratégie.

Les étapes de précalcul mentionnées précédemment sont facilement intégrables dans un système de rendu donné et sont les composantes clé de nos noyaux de synthèse géométrique utilisés à la génération de chaque image. Premièrement, les tables de fonctions de bases [Bol+02] sont chargées en mémoire d'accès aléatoire (*"random-access resources"*) pour permettre leur utilisation dans un *shader*. La mémoire nécessaire au stockage de $\mathbf{M}^1$ est alors allouée sur la GPU pour permettre l'éxécution de la première étape de subdivision du maillage d'entrée. Le voisinage

de $\mathbf{M}^1$ nécessaire à l'étape principale est ensuite encodée. On impose que $\mathbf{M}^1$ soit une tessellation uniforme de $\mathbf{M}^0$ pour éviter toute manipulation de la mémoire GPU pendant l'éxécution du programme. Cela évite la modification de la répartition de la mémoire et permet d'animer $\mathbf{M}^0$ simplement sur la GPU. Lorsque la connectivité du maillage d'entrée ne change pas (si $F^1$ est statique), ces données n'ont pas besoin d'être recalculées.

Pendant l'étape initiale, on encode les masques de subdivision comme une fonction linéaire, ce qui est adapté aux architectures matérielles modernes en utilisant les poids et indices des sommets:

```
struct CompEntry {
   float m_fWeights[ n ];
   uint  m_uiIndices[ n ];
};
```

De manière similaire, on encode l'ensemble des triangles adjacents à un sommet donné de $\mathbf{M}^1$ pour permettre le calcul des normales de celui-ci. Les cas de frontières sont gérées à l'aide d'une simple valeur boolean indiquant si la normale à calculer se situe sur une frontière.

```
struct CompNormalEntry {
   bool  m_bIsBoundary;
   uint  m_uiIndices[ n ];
};
```

Nous avons choisi de recalculer les normales des sommets, bien qu'il soit possible d'utiliser les mêmes poids que pour les positions de ceux-ci, d'après l'idée de *"subdivision shading"* [Ale+08]. Cependant, ce procédé produit en général des résultats visuels trop lisses à notre goût.

Pour ce qui est du matériel graphique supportant les shaders de calcul ( *"compute shaders"*), chaque entrée peut être traîtée de manière indépendante par un thread sur la GPU. Il est possible de simuler ce comportement sur les cartes graphiques plus anciennes en instanciant une primitive de point. Le résultat est procedé et enregistré par le *"transform feedback"* ou avec l'étape de rasterization (*"rasterization stage"*). Cette dernière étape est nécessaire uniquement si les resources d'accès mémoire aléatoire sont réservées aux textures.

Dans la passe suivante, on complète $\mathbf{M}^1$ en mémoire vidéo avec le recalcul des normales. Différents schémas peuvent être appliqués lors de cette étape [Jin+05] et des attributs supplémentaires des sommets tels que des coordonnées de textures [DeR+98; He+10] peuvent être gérés de manière similaire.

Notre étape principale de subdivision est alors effectuée pour sur-échantilloner directement le maillage au niveau $i$ en utilisant les tables de fonctions de base (Fig. 0.3). Cette étape peut être intégrée directement dans les *pipelines* de rendu existants avant l'étape de rastérisation, et ne requiert que d'associer les configura-

tions des patches (c'est-à-dire les triangles et leurs triangles adjacents) aux entrées correspondantes dans les tables de fonctions de base à l'aide d'un simple index. Chaque patch est encodé dans notre pipeline comme

```
struct BFTPatch {
   uint m_uiPoints[ n ];
   uint m_uiIndexBFT;
   uint m_uiOneRingSize;
};
```

A l'aide d'une unité de tessellation, nous pouvons exporter les coordonnées 2D des points sur les triangles $f_i^1$ pour générer la triangulation raffinée. Les sommets de cette triangulation sont évalués simplement comme une combinaison linéaire des positions des sommets de $\mathbf{M}^1$ à l'aide des tables précalculées:

$$f(u,v) = \sum_i B^i(u,v) v_i^1$$

Nous générons également un champ de normales lisse en utilisant les mêmes poids que pour les positions [Ale+08]. Des tables précalculées correspondant aux masques des tangentes pourraient être utilisées. Cependant, l'utilisation des normales correspondant précisemment à la géométrie générée crée généralement un rendu visuel dégradé autour des sommets extraordinaires [Ale+08].

Notre approche est également compatible avec un raffinement adaptatif pendant l'étape de sur-échantillonage. Les sommets de schema adaptatif générés par l'unité de tessellation [Bou+08a] sont evaluées avec les BFTs et les valeur pour les sommets *inexistants* sont ignorées. Il est préférable que les sommets introduits aient des coordonnées dyadiques finies pour faciliter l'assocation de ces coordonnées paramétriques avec des indices dans les tables de fonctions de base.

### 0.3.4 Résultats

Pour évaluer notre méthode, nous avons testé les performances sur une machine équipée d'une carte graphique NVIDIA GeForce GTX 295 avec 1.8 GB de mémoire vidéo et un processeur Intel Core i7 2.67 GHz utilisant OpenGL sous Windows. Nous avons utilisé le noyau de raffinement adaptatif [Bou+08a] en tant qu'émulateur d'unité de tessellation, mesuré les performances pour différents maillages d'entrée et à différents niveaux de résolution de sortie, et comparé les résultats avec différentes techniques alternatives (Tab. 0.1). Bien que notre méthode ne soit pas sans conséquence sur les performances, elle demande à peu près deux fois plus de temps d'exécution qu'une simple subdivision linéaire et améliore significativement la qualité des maillages, puisque les surfaces réelles de subdivision sont générées en temps-réel (Fig. 0.4).

De plus, le champ de normales généré par le rendu de subdivision est suffisamment lisse pour éviter les discontinuités dans le rendu, même en présence de lumières fortement spéculaires (Fig. 0.5).

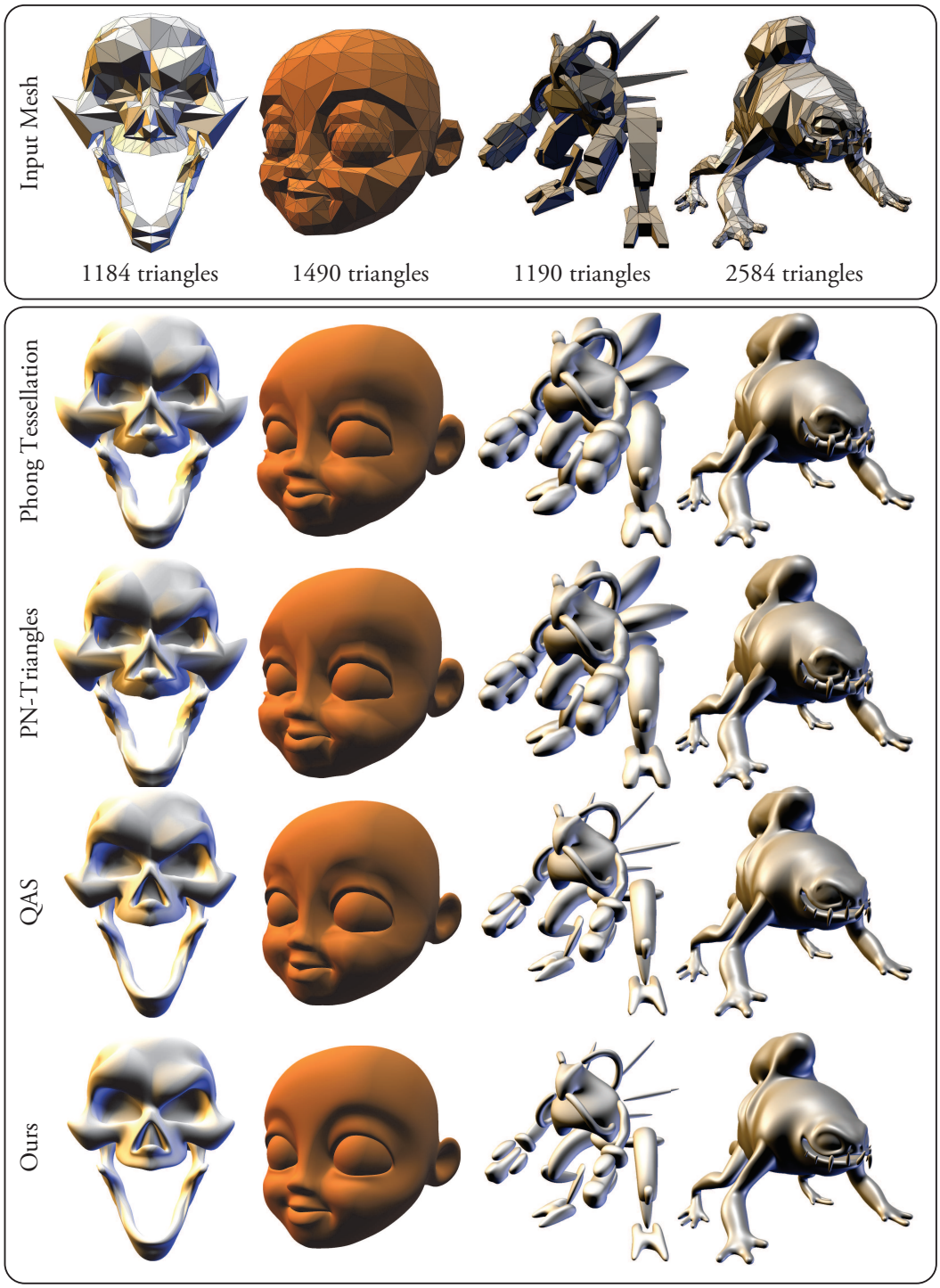



**Figure 0.4:** Raffinement sur la GPU: des alternatives à la subdivision comparées à nos maillages de subdivision ($\mathbf{M}^5$).

| Modèle | Head | Guy | MonsterFrog | BigGuy |
|---|---|---|---|---|
| #Triangles entrée | 524 | 1168 | 2584 | 2900 |
| *Tessellation Niveau 2 (4×4 split)* | | | | |
| #Triangles sortie | 8384 | 18688 | 41344 | 46400 |
| *Images par sec.* | | | | |
| Notre méthode | 641 | 495 | 334 | 291 |
| *Tessellation Niveau 4 (16×16 split)* | | | | |
| #Triangles sortie | 134 k | 299 k | 661 k | 742 k |
| *Images par sec.* | | | | |
| Notre méthode | 312 | 176 | 85 | 76 |
| *Tessellation Niveau 5 (32×32 split)* | | | | |
| #Triangles sortie | 536 k | 1196 k | 2646 k | 2969 k |
| *Images par sec.* | | | | |
| Notre méthode | 103 | 50 | 24 | 21 |
| QAS [Bou+07a] | 80 | 53 | 44 | 23 |
| PN-Tri. [Vla+01] | 109 | 69 | 41 | 35 |
| PT [Bou+08b] | 120 | 77 | 48 | 40 |
| Simple (linéaire) | 137 | 85 | 52 | 45 |

**Table 0.1:** Performance de notre algorithme à différents niveaux de résolution de sortie.

### 0.3.5 Conclusion

Nous avons présenté notre noyau de synthèse géométrique pour la création dynamique de maillages de subdivision en temps-réel avec comme sortie des maillages raffinés de manière adaptative. Cela a été réalisé grâce à une combinaison d'étapes impliquant une étape préliminaire de subdivision sur la GPU pour séparer les sommets extraordinaires, une étape de sur-échantillonage haute résolution à l'aide d'une unité matérielle/logicielle de tessellation, et une étape finale d'évaluation de la position des sommets du maillage raffiné à l'aide de tables de fonctions de base précalculées. Ce noyau ne requiert aucun précalcul spécifique au schéma de subdivision employé, à l'exception près de l'encodage des masques [Zor99] pour l'étape initiale de subdivision. Des travaux récents ont utilisé une approche similaire [Nie+12] et permettent la subdivision adaptative à la première étape ainsi que la préservation de géométrie présentant des arêtes saillantes ou semi-saillantes.

Après avoir présenté nos travaux la synthèse de géométrie haute résolution, nous allons maintenant montrer comment obtenir un rendu haute qualité de celle-ci à l'aide d'effets d'illumination globale. Encore une fois nous allons présenter des approches offrant un niveau de détail adaptatif, mais à la différence de ce qui a été fait précédemment nous n'allons pas nous concentrer sur une vue unique de

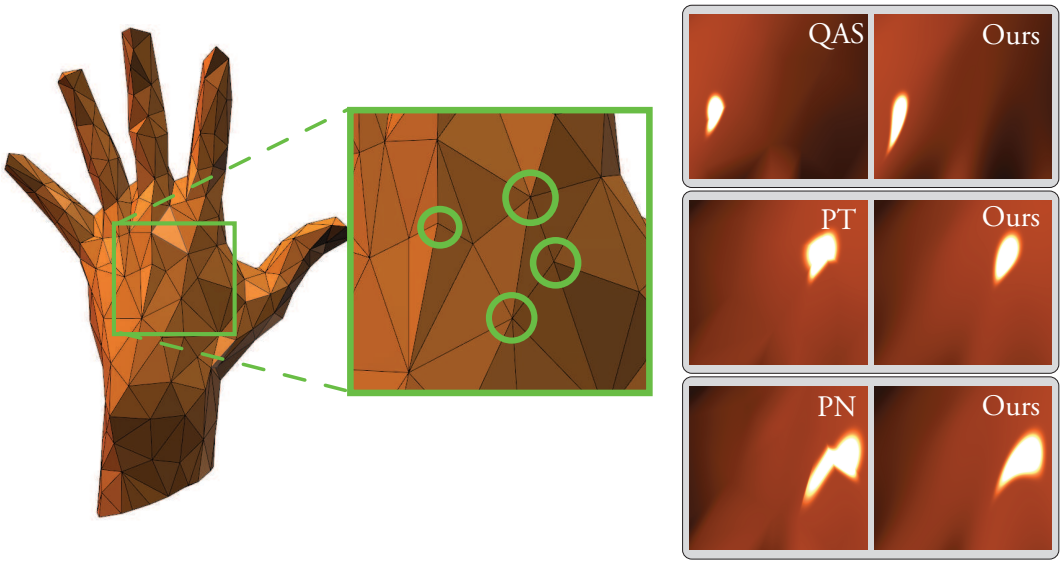



**Figure 0.5:** Qualité de la surface autour des sommets extraordinaires: comparaison entre quadratique approximation (QAS), Tessellation de Phong (PT), PN-Triangles (PN) et notre méthode de subdivision. Pour chaque schéma la lumière est placée de manière à produire des reflets fortement spéculaires près des sommets extraordinaires.

la géométrie mais sur la création d'un niveau de détail de la géométrie adapté à un grand ensemble de vues à la fois. Chaque vue sera de basse-résolution, mais requiert néanmoins un niveau de détail flexible de la géométrie pour permettre le rendu de la scène en temps-réel.

## 0.4 Illumination globale en temps-réel à l'aide de points

Dans cette section nous voyons comment calculer efficacement les effets d'illumination directe et indirecte, souvent appelés illumination globale (*"Global Illumination"* – GI). C'est une tâche relativement compliquée pour les applications temps-réel, car ce calcul implique la simulation du transport de la lumière dans la scène et doit être effectué avec un budget très mince pour chaque image rendue. Cependant, cela peut être facilité par l'utilisation d'une multitude de caches qui capturent chacun des informations supplémentaires sur la scène 3D. Ces caches sont traditionnellement remplis par un procédé de rastérisation dans une mémoire tampon basse résolution, et simplifient les calculs qui sont de plus dépendant de la visibilité locale. Pour accélérer ce procédé, la scène est souvent représentée par une structure hiérachique de primitives englobantes qui permet d'extrairent des niveaux de détail variés de celle-ci. Bien que le coût de rendu dans chacun de ces caches basse résolution est relativement faible, du au faible nombre de pixels impliqués dans l'opération, le grand nombre de ces caches nécessaire pour obtenir des résultats visuellement convaincants pose un problème géométrique car le coût de la sélection du niveau de détail approprié pour chaque vue est lui relativement

important. Cette dernière tâche est précisemment ce que notre structure parallèle de sélection de niveau de détail offre à bas coût. Nous commençons par introduire les concepts mathématiques nécessaires à la compréhension du problème, puis décrivons un ensemble de techniques connues dont l'étude est ici pertinente. Nous présentons ensuite notre algorithme de sélection parallèle pour l'accélération d'une variété de ces techniques ainsi que des extensions possibles de cet algorithme.

### 0.4.1 L'illumination Globale

L'équation du rendu de Kajiya [Kaj86] décrit la luminance énergétique réfléchie (*"output radiance"*) $L_o$ à un point $\mathbf{p}$ d'une surface dans la direction $\omega$ comme suit:

$$L_o(\mathbf{p},\omega) = L_e(\mathbf{p},\omega) + \int_{\Omega_+} f_r(\mathbf{p},\omega,\omega') L_i(\mathbf{p},\omega') \cos(\mathbf{n_p},\omega') \, \mathrm{d}\omega'$$

avec

1. $L_e$ est un terme non-nul décrivant la luminance énergétique incidente au point $\mathbf{p}$ dans la direction $\omega$.
2. $\Omega_+$ est l'hémisphère au point $\mathbf{p}$ centrée autour de la normale à la surface $\mathbf{n_p}$.
3. $f_r$ est la fonction de distribution de réflectance bidirectionnelle (*"Bidirectional Reflectance Distribution Function"* – BRDF) qui caractérise les propriétés du matériau en décrivant l'énergie réfléchie depuis la direction $\omega'$ et dans la direction $\omega$ au point $\mathbf{p}$.
4. $L_i$ luminance énergétique incidente au point $\mathbf{p}$ depuis la direction $\omega'$.
5. Le terme en cosinus étant le flux surfacique et décrivant le fait que le nombre de photons par élément de surface décroit avec l'angle entre $\mathbf{n_p}$ et $\omega'$.

Cependant, l'équation du rendu dépend d'elle-même, puisque la luminance énergétique incidente en $\mathbf{p}$ comprend non seulement la lumière venant directement d'une source de lumière mais également la lumière indirecte via des rebonds additionels dans la scène. Il est à noter que l'intégrale surfacique sur l'hémisphère ne peut être résolue de manière analytique que dans un ensemble très petit de cas.

Néanmoins, plusieurs solutions pratiques de rendu existent, et nous résumons maintenant celles qui sont le plus en rapport avec notre technique.

**Radiosité instantanée (*"Instant Radiosity"*)** La méthode de radiosité instantanée [Kel97] (*"Instant Radiosity"* – IR), consiste à lancer un certain nombre de photons depuis les sources de lumière, et de créer des sources de luimères secondaires aux points d'intersection avec la scène 3D. Ces sources de lumière virtuelles (*"Virtual Point Lights"* – VPLs) contiennent le flux radiant du point de la scène correspondant. Chaque point de la surface des objets de la scène qui doivent être rendus dans l'image finale est alors influencé par les sources de lumière

principales et virtuelles (VPLs). Dans les deux cas, la visibilité doit être prise en compte à l'aide de techniques de type raycasting ou à l'aide de cartes d'ombres (*"shadow maps"*).

Pour faire fonctionner cette approche en temps-réel, le placement des VPLs peut être accéléré en étendant la définition des cartes d'ombres pour capturer non seulement la profondeur depuis le point de vue mais également la normale et le flux. Ces cartes sont appelées cartes d'ombres réfléchissantes [Dac+05] (*"reflective shadow maps"*). Les requêtes de visibilité peuvent être accélérées significativement en calculant une carte d'ombres basse résolution pour chaque VPL, en approximant la scène 3D par un ensemble de points et en allouant un budget fixe par VPL. Les trous laissés dans les caches peuvent être comblés en utilisant une approche de type *"push-pull"*. Des améliorations peuvent être obtenues en organisant les VPLs hiérarchiquement [Wal+05], en utilisant un schéma complexe de sélection de niveau de détail [Rit+11] ou des méthodes de segmentation (*"clustering"*) [Rit+11; Pru+12] et en exploitant la cohérence temporelle des animations rendues [Lai+07].

**Les Caches d'Intensité d'Irradiation (*"Irradiance Caching"*)**
*"Irradiance Caching"* [War+88] est une technique basée sur l'idée que la lumière indirecte diffuse ne varie que de manière lisse. Il est donc suffisant de calculer les effets de lumière indirecte à un ensemble réduit de points et de diffuser les valeurs obtenues [Tab+04] à tous les autres points pour lesquels le rendu doit être effectué. Cette stratégie est séquentielle et est effectuée à la demande: lors du rendu d'un point de la surface, si les caches déjà calculés ne sont pas suffisant au rendu du point considéré, un nouveau cache est créé et pourra être utilisé plus tard pour le rendu de points autour de lui. Pour les contextes temps-réel, le nombre maximal de caches est en général fixé.

**Les Caches d'Intensité de Radiation (*"Radiance Caching"*)**
*"Radiance Caching"* [Kri+05] étend l'idée des caches d'intensité d'irradiation, en gardant des informations supplémentaires par cache, pour permettre l'interpolation de BRDFs ne contenant pas de hautes fréquences. Pour cela, des informations directionelles sont gardées en mémoire sous la forme d'harmoniques (hémi-)sphériques et d'un repère local et deux gradients de translations Dans un contexte temps-réel, ces caches peuvent être interpolés rapidement en utilisant des techniques basées sur l'idée mip-mapping de textures [Sch+12].

**Illumination globale à l'aide de représentations de points** L'illumination globale de scènes 3D représentées par un ensemble dense de points (*"Point-Based Global Illumination"* – PBGI) [Chr08] permet de calculer une approximation non-bruitée de l'occlusion ambiante (*"Ambiant Occlusion"* – AO) et du *bavement* de couleur (*"Colour Bleeding"*). Chaque point garde en mémoire la lumière réfléchie via des fonctions sphériques, et la totalité des points est organisée dans une structure spatiale en arbre. Cet arbre est traversé pour chaque point récepteur pour lequel la lumière indirecte doit être calculée via un procédé de rastérisation (essentiellement) de cette représentation pour obtenir une version basse résolution de la scène vue depuis

le point récepteur. Il est possible d'obtenir des performances intéractives voir temps-réel en utilisant, encore une fois, un faible nombre de récepteurs comme caches [Rit+09] et un opérateur de filtrage bilatéral [Slo+07] pour sur-échantilloner de manière propre le calcul de la lumière indirecte réalisé à la position de ces caches.

Dans la suite nous résumons notre algorithme parallèle d'extraction de *coupes* d'arbre correspondant à une sélection d'un niveau de détail (*"Level of Detail"* – LoD) de la scène pour l'accélération des techniques présentées.

### 0.4.2 ManyLoDs: Sélection parallèle de multiples niveaux de détail pour l'illumination globale temps-réel

Comme résumé précédemment, un large pannel de techniques reposent sur la création de caches qui capturent des informations supplémentaires de la scène 3D, et pour lesquelles le nombre de caches nécessaires peut facilement atteindre plusieurs centaines voir plusieurs milliers. Pour chaque cache, une image basse résolution est rastérisée pour un coût relativement faible par cache, alors que le coût de sélection du niveau de détail approprié par cache reste important. L'idée principale de notre algorithme est de calculer les coupes dans l'arbre progressivement en utilisant un noyau de synthèse géométrique qui exploite pleinement les capacités de traitement parallèle de la GPU. Il est également possible d'amortir le coût de calcul sur plusieurs images successives. Nous avons choisi de représenter la scène à l'aide d'une structure hiérachique de sphères englobantes (*"Bounding Sphere Hierarchy"* – BSH) et une simple distribution dense de points étant les feuilles de l'arbre. Nous démontrerons dans la suite les bénéfices de notre approche pour de multiples techniques, telles que l'Instant Radiosity, le PBGI et le calcul de réflexions/réfractions. De manière plus générale, notre approche offre:

- une sélection d'un LoD pour un grand nombre de vues en parallèle
- l'adaptation de schémas incrémentaux et de mises à jour partielles (*"lazy update"*) aux problèmes de type multi-vues

Nous utilisons une structure de subdivision spatiale [Ben75; Jac+80; Fuc+80] – un arbre – qui peut être parcouru efficacement pour obtenir des informations sur la scène 3D. Cet arbre peut être construit de manière *"top-down"*, en séparant récursivement les volumes englobants, ou de manière *"bottom-up"*, en fusionnant des éléments voisins pour créer des noeuds parents de l'arbre.

**Terminologie:** Une structure BVH est un ensemble hiérarchique de noeuds organisé dans une structure d'arbre: BVH $:= \{N_i\}$. Une paire *noeud-vue* $(N_i, V_j)$ correspond à un noeud du BVH et à une vue donnée. Nous définissons une *coupe* $C$ à travers un BVH selon un critère $c$, qui est une fonction scalaire de paires noeud-vue, ayant la propriété suivante:

$$\forall N_i, N_j, V_k : c(N_i, V_k) \leq c(N_j, V_k) \Leftrightarrow \text{level}(N_i) < \text{level}(N_j),$$

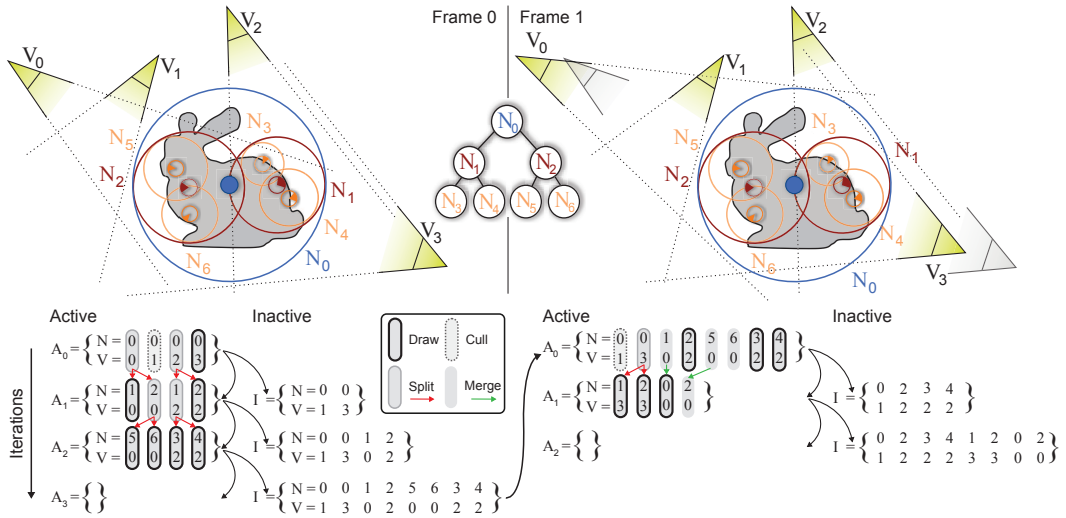



**Figure 0.6:** Principe de notre algorithme. Pour deux trames consécutives (gauche et droite) la géométrie de la scène (Bunny) représentée par une structure BVH (sphères) est rendue pour de multiples vues (de V0 à V3). Gauche: l'arbre est traversé itérativement (flèche verticale) en parallèle pour toutes les paires noeud-vue. La traversée commence par la liste dite *active* $A_0$ contenant le noeud racine $\mathrm{N}_0$ pour chaque vue. A chaque itération, les paires noeud-vue sont soit ignorées (*"culling"*), mergées (flèches vertes), coupées (flèches rouges), ou dessinées. La liste active restante liée au calcul de l'image est utilisée comme entrée pour le calcul de la trame suivante. Les noeuds ignorés ne sont pas enlevés de la liste pour pouvoir les utiliser comme points de départ pour les trames suivantes. Après toutes les itérations la liste active $A_3$ est vide et la liste dite *inactive* $I$ contient la coupe pour toutes les vues. Droite: Le résultat de la trame précédente est utilisée comme entrée pour le rendu de la trame suivante et pour des positions de caméra différentes. Ce procédé permet de réduire le nombre d'itérations avant convergence.

ce qui signifie que $c$ décroit de manière monotonique lorsque le niveau du BVH augmente. Dans notre cas, $c$ est défini comme étant la taille en pixels du noeud $N_i$ depuis la vue $V_j$ ou bien 0 si le noeud est ignoré. La coupe $C$ est l'ensemble $C := \{(N_i, V_j) \mid c(N_i, V_j) < \epsilon \;\&\; c(\text{parent}(N_i), V_j) > \epsilon\}$, où $\text{parent}(N_i)$ est le noeud parent de $N_i$ et $\epsilon$ est une valeur de tolérance donnée par l'utilisateur. On dit que la paire noeud-vue est *valide* si et seulement si $c(N_i, V_j) < \epsilon$.

Pour une machine parallèle qui peut ajouter des éléments à une liste à l'aide d'une stratégie de *"prefix scan"* [Ble89] ou d'une stratégie similaire, nous offrons trois variantes de notre approche: une approche basique, une approche incrémentale pour diminuer le coût d'éxecution, et une approche dite *paresseuse* avec un coût amorti sur plusieurs trames.

### Approche basique

Pour trouver toutes les coupes au travers d'un arbre avec arité $n$ et de profondeur maximum $h$ pour $m$ vues en parallèle, nous gérons deux listes de paires noeud-vue: $I$ et $A$. La liste $I$ contient toutes les paires noeud-vue dites *inactives*, c'est-à-dire

l'ensemble des noeud-vue qui sont dans la coupe et qui ne requièrent pas de calcul supplémentaire. Les paires noeud-vue de la liste $A$ sont considérées comme *actives* et ont besoin d'être traîtées (Fig. 0.6 partie gauche).

Au départ, nous initialisons la liste $I = \emptyset$ comme étant vide et la liste $A_0 = \{(N_0, V_0), \ldots, (N_0, V_{m-1})\}$ avec le noeud racine de l'arbre couplé à chaque vue. Pour trouver le LoD approprié pour toutes les vues, nous effectuons $h$ étapes: A l'étape $k = 1 \ldots h$, un noyau parallèle est éxécuté sur toutes les paires noeud-vue $a \in A_{k-1}$ de la liste active à l'étape précédente. Le noyau ajoute la paire $a := (N_a, V_a)$ à $I$ si elle est valide. Sinon, de nouvelles paires noeud-vue résultant du scindement d'un noeud sont ajoutées à la liste $A_k$, une paire pour chaque noeud fils de $N_a$ couplé avec la vue $V_a$. Après $h$ étapes, $I$ contient la coupe *multi-vues* et peut être utilisée pour des applications de rendu.

**Approche incrémentale**

Au lieu d'initialiser la liste $I = \emptyset$ comme étant vide et la liste $A_0 = \{(N_0, V_0), \ldots, (N_0, V_{m-1})\}$, nous démarrons avec le résultat de la coupe obtenue à la trame précédente [Xia+96]. Cela signifie que la liste $I$ de la trame précédente est utilisée comme liste active à l'étape $0$ $A_0$, en partant du principe que deux coupes réalisées à des images successives ont de fortes chances d'être similaires (Fig. 0.6, partie droite). Cela requiert des modifications de notre algorithme. Premièrement, les paires noeud-vue doivent être fusionnés lorsque $(\text{parent}(N_i), V_j)$ devient valide, par exemple lorsque la caméra s'éloigne des noeuds considérés. Deuxièmement, les paires noeud-vue qui sont ignorées ne doivent pas être simplement oubliées mais mises de côté pour les utiliser comme point de départ pour le calcul des coupes des trames suivantes. Cela peut se produire par exemple lorsque une paire noeud-vue sort du champ de la caméra puit y rentre à nouveau. Cependant, on peut fusionner des noeuds ignorés lorsque le noeud parent est ignoré. Le nouveau noyau réalise donc l'une de ces trois actions sur une paire noeud-vue $a \in A_{k-1}$:

1. Si $a$ est valide mais que son parent $a'$ ne l'est pas: $a$ est ajouté à $I$.

2. Si $a$ est valide, que son parent $a'$ l'est aussi et que $a$ est le premier fils de $a'$: $a'$ est ajouté à $A_k$. Aucun autre fils de $a'$ n'est mis dans $I$. Cela évite d'avoir $n$ copies de la même paire noeud-vue dans $I$.

3. Si $a$ et son parent $a'$ sont invalides, tous les fils de $a$ sont ajoutés à $A_k$.

Si la scène est cohérente au cours du temps, c'est-à-dire la géométrie et les vues varient de manière continue comme dans nos applications, le nombre d'opérations requises pour produire $I$ est diminué d'au moins un ordre de grandeur par rapport à l'approche basique précédemment présentée.

**Approche *paresseuse* de mises à jour partielles**

Lorsque des coupes approximatives sont suffisantes, on peut limiter le nombre d'itérations de $A$ à un nombre $q < h$. Par exemple, si $q = 1$ une seule passe est réalisée sur toutes les paires noeud-vue actives à chaque trame, ces paires devenant immédiatement inactives. De fait, une paire est soit laissée telle quelle, ignorée, fusionnée avec ses noeuds frères ou bien divisée, mais le processus n'est pas répété. La précision des coupes peut alors être diminuée car $I$ peut contenir des paires noeud-vue invalides. Cependant, l'algorithme est simplifié de manière drastique en limitant le nombre d'arêtes de l'arbre traversées à $q$ par paire noeud-vue.

Bien que notre algorithme est suffisamment général pour supporter une grande variété d'arbres, pour tous les exemples présentés dans cette thèse nous avons appliqué les étapes suivantes de précalcul.

**Précalcul**

On commence par distribuer uniformément $n = 2^d$ points sur les surfaces de la scène, créant un ensemble $P$ représentant la scène. Pour supporter les scènes dynamiques, nous transférons les déformations nécessaires au BVH [Rit+08], en exprimant chaque point $p \in P$ à l'aide de coordonnées locales barycentriques $p = (s, t, i_{\text{tri}})$ dans un triangle $i_{\text{tri}}$.

Deuxièmement, on construit la topologie d'un arbre binaire complet de profondeur $d + 1$ depuis $P$ en $k$, pendant lesquelles on sépare l'ensemble des noeuds en ré-ordonnant les éléments dans $P$. Pour éviter de ré-ordonner les éléments à chaque itération, on utilise l'approche de Wald et Havran [Wal+07]. Tous les noeuds sont ordonnés une fois dans chacune des trois dimension, puis une approche diviser-et-conquérir est réalisée sur les trois listes ordonnées pour construire l'arbre avec complexité $\mathcal{O}(N \log N)$. L'étape $k \in [0 \dots d + 1]$ considère $2^k$ sous-séquences $Q_j$ de taille $2^{d-k}$. Pour chaque $Q_j$, on considère les trois divisions en deux sous-ensembles de même taille réalisées dans les directions $x$, $y$ et $z$. La division qui est choisie est celle qui minimise la somme des volumes des deux séquences créées, qui deviennent deux noeuds dans l'étape suivante. Chaque noeud $N_0 \dots N_{2^{d+1}-1}$ stocke une sphère englobant les positions des points, et un cône englobant les normales.

**Phase d'exécution**

**Mise à jour** Avant de calculer les multi-coupes en parallèle à chaque trame, on met à jour la géométrie du BVH de manière ascendante en parallèle. On démarre un thread pour chaque point de la scène pour mettre à jour les feuilles de l'arbre en premier. Chaque thread met à jour la géométrie du point en utilisant ses coordonnées barycentriques $(s, t)$ dans le triangle $i_{\text{tri}}$ approprié. Pour mettre à jour les volumes englobants des autre noeuds, on procède de manière ascendante et démarre des threads pour chaque noeud du niveau de l'arbre considéré, chaque thread fusionnant les volumes englobants de ses fils. Il est à noter que cette étape

```
tant que( !estVide( A_k ) ) {
 pour chaque a de A_k, faire en parallèle {
    if( !estValide( parent( a ) ) {
       if( estValide( a ) ) {
          dessine( a );
          ajouteA( I, a );
       } else {
          ajouteA( A_{k+1}, tousLesFilsDe( a ) );
       }
    } else if( estValide( a ) & estPremierFils( a ) ) {
       ajouteA( A_{k+1}, parent( a ) );
}}}
```

**Listing 1:** Pseudo code de notre algorithme incrémental dans un shader géométrique utilisant des capacités de flux sortant (*"stream-out capabilities"*).

n'est réalisée qu'une seule fois à chaque trame, indépendamment du nombre et de la position des vues.

**Calcul de multi-coupes** Nous avons utilisé les capacités de flux sortant de notre carte graphique supportant les Modèles de Shaders 5.0 via le *"transform feedback"* pour ajouter des éléments aux flux sortants individuels tout en maintenant un grand nombre de threads. $I$ et $A_k$ sont associés à des *"vertex streams"* et traîtée dans un shader géométrique (Lst. 1). A chaque itération, nous avons les options suivantes pour une paire noeud-vue $a$: séparer (créer $n$ fils suivant l'arité de $a$ et ajouter les fils à la liste active), dessiner (dessiner le noeud dans la vue correspondante et l'ajouter à la liste inactive), ignorer (ajouter $a$ à la liste inactive, pour le considérer comme point de départ pour les images suivantes). Chaque appel de fonction du shader ajoute un petit ensemble de valeurs dans une liste. Une amplification des données de cet ordre est considéré comme étant un scénario optimal pour un shader géométrique.

Pour le rendu, nous dessinons simplement un point (GL_POINT) avec un rayon approprié à la sphère englobante correspondante, (que le lecteur se souvienne que nous utilisons une structure de sphères englobantes comme hiérarchie). Nous associons chaque vue à un carreau dans la texture de destination, pour éviter de faire le rendu de nos $m$ coupes dans $m$ textures différentes (Fig. 0.10 partie droite).

### 0.4.3 Applications et résultats

Nous allons maintenant présenter de multiples applications tirant profit de notre approche, pour améliorer la qualité et/ou les performances de plusieurs effets de rendu d'illumination globale utilisant des distributions de points. Nous analysons ensuite les performances de notre algorithme en général.

**Instant Radiosity** Pour simuler l'illumination indirecte, *"Instant Radiosity"* [Kel97] place un ensemble de sources secondaires de lumière – lumières ponctuelles virtuelles (*"virtual point lights"* – VPLs) – au point d'intersection des photons émis

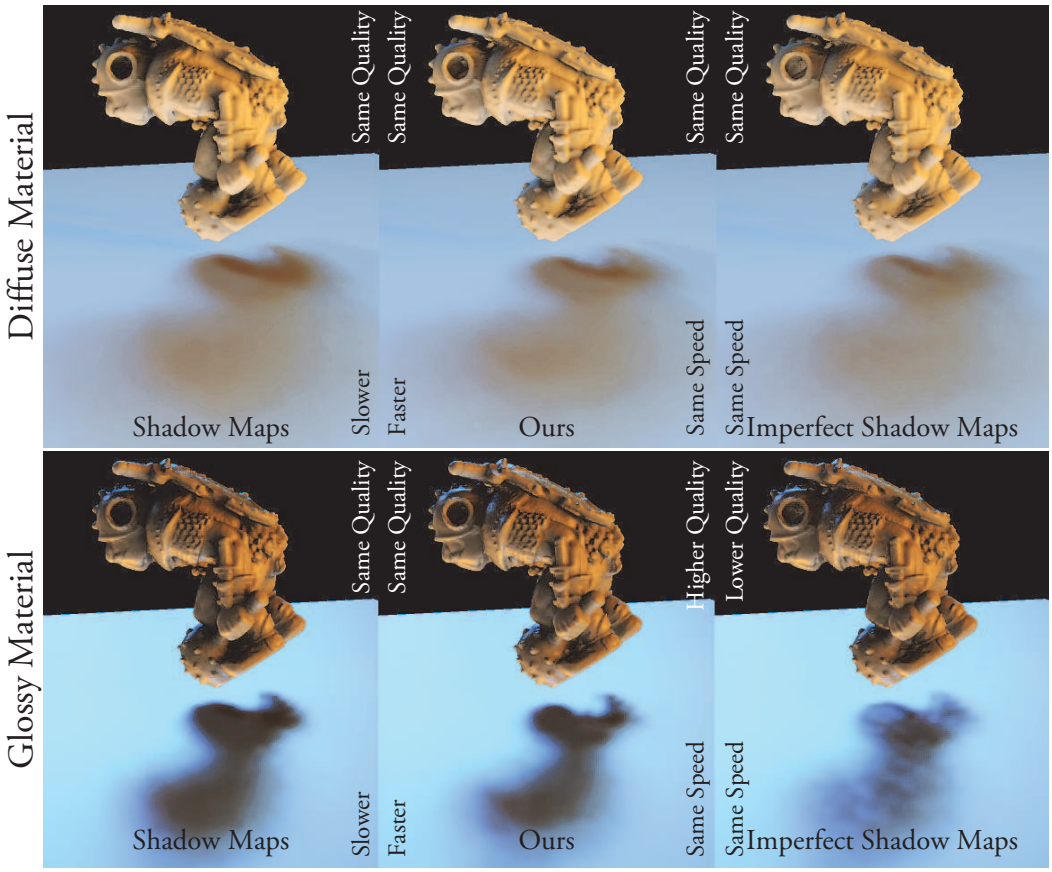



**Figure 0.7:** Un personnage dynamique (500 k polygones, 500 k points) sous lumière naturelle, utilisant des matériaux diffus (gauche) et spéculaires (droite). Lumière naturelle utilisant 10244 lumières ponctuelles, et comparaison avec les classiques cartes d'ombres (haut à gauche, bas à gauche), notre approche multi-vues pour rastériser les cartes d'ombres (haut au centre, bas au centre) et ISM (haut à droite, bas à droite). Notre approche et ISM ont été ajustées pour obtenir le même temps d'exécution de 4 ms. Pour des surfaces diffuses, notre approche et ISM produisent des résultats similaires. Pour des surfaces brillantes, les imperfections de ISM sont visibles, alors que la qualité de notre approche est préservée.

depuis la source de lumière primaire. Pour chaque VPL, une simple carte d'ombres basse-résolution est calculée, mais un grand nombre de VPLs est nécessaire pour obtenir un résultat réaliste. Nous calculons les coupes de la scène pour tous les VPLs en parallèle et améliorons la qualité par rapport aux approches précédentes [Rit+08] pour des performances comparables, comme démontré en figure 0.7 et en figure 0.8 pour des matériaux brillants. Pour un budget de temps d'exécution fixé, notre méthode s'adapte mieux à la densité de points et peut éviter de créer des trous, alors que l'algorithme classique de cartes d'ombres imparfaites (*"Imperfect Shadow Maps"* – ISM) les fait apparaitre.

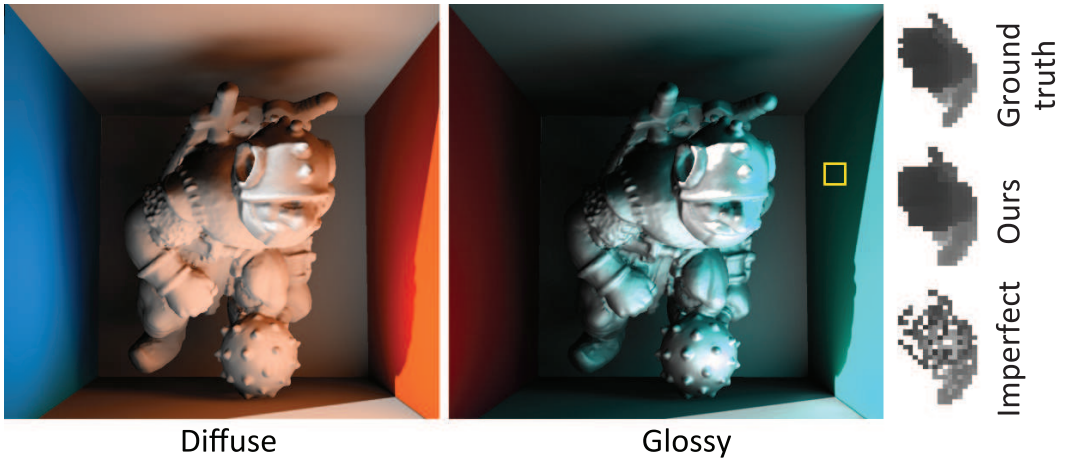



**Figure 0.8:** Instant Radiosity utilisant 1024 VPLs pour le maillage Grog (500 k triangles, 500 k points). Temps de rendu de 2 ms pour un matériau diffus (en haut) et brillant (en bas) à une résolution de 1024×1024. Pour un budget de temps fixé de 2 ms, notre algorithme est plus proche de la vérité terrain (cartes d'ombres) que ISM. ISM contient des trous pour les occludeurs proches, comme montré dans la région indiquée.

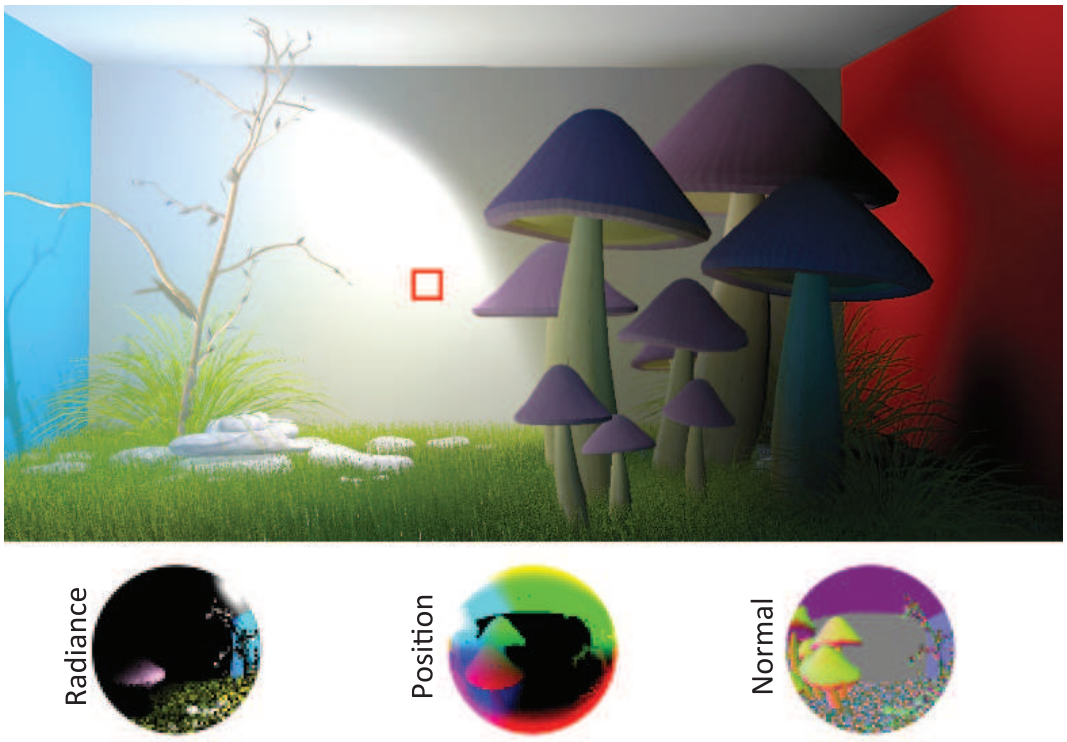



**Figure 0.9:** En haut: *"Point-Based Global Illumination"* dans une scène (700 k triangles, 1 M points). Rendu à une résolution de 1024×1024 avec 4096 vues (29 ms). En bas: Radiation, position et normal pour une vue (indiqué en rouge).

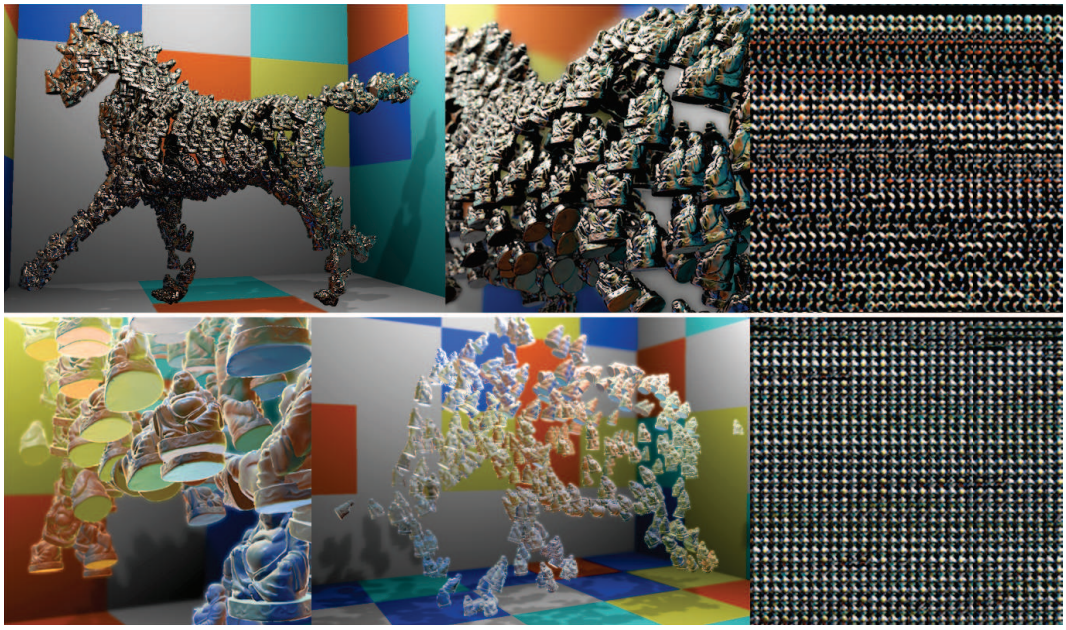


**Figure 0.10:** Rendu de 400 réflexions (haut) et réfractions (bas) d'objets (2 M polygones, 2 M points) animés sous la forme d'un animal animé à l'intérieur d'une boite de Cornell. Notre approche rastérise tous les objets dans la carte de réflexion de chaque autre (droite) en 40 ms.

**Illumination globale à l'aide de points (*"Point-Based Global Illumination"* – PBGI)** PBGI est utilisée pour simuler des effets d'illumination complexes [Bun05; Chr08; Rit+09]. Alors que la scène 3D est soit rastérisée soit rendue à l'aide de lancers de rayons (*"ray tracing"*) dans la vue primaire, les effets d'illumination indirecte sont simulés en plaçant des vues secondaires dans la scène, par exemple à la position 3D correspondante à chaque pixel, et en faisant un rendu basse-résolution pour ces vues. Finalement, une convolution avec la BRDF est effectuée de manière séquentielle [Chr08] ou parallèle pour toutes les vues [Bun05; Rit+09]. Au contraire, nous parallélisons tout le processus et trouvons toutes les coupes d'un BVH sous lumières pour tous les pixels visibles. Nous obtenons des résultats de haute qualité, comme obtenu par les travaux précédents [Bun05; Chr08; Rit+09], mais améliorons sensiblement les performances (Fig. 0.9).

**Réflexions** Les cartes d'environnement (*"environment maps"*) sont utiles pour simuler des effets visuels comme les réflexions [Bli+76; SK+05] et réfractions [Wym05]. Notre approche permet la création en temps-réel de ces cartes pour des centaines d'objets en parallèle (Fig. 0.10). Pour chaque objet, nous rastérisons une carte d'environnement à l'aide de la représentation de points de la scène et trouvons les coupes appropriées dans le BVH.

**Autres applications** Notre algorithme peut également accélérer des applications quand le rendu n'est effectué que dans une vue (avec $m = 1$). Pour la rastérisation

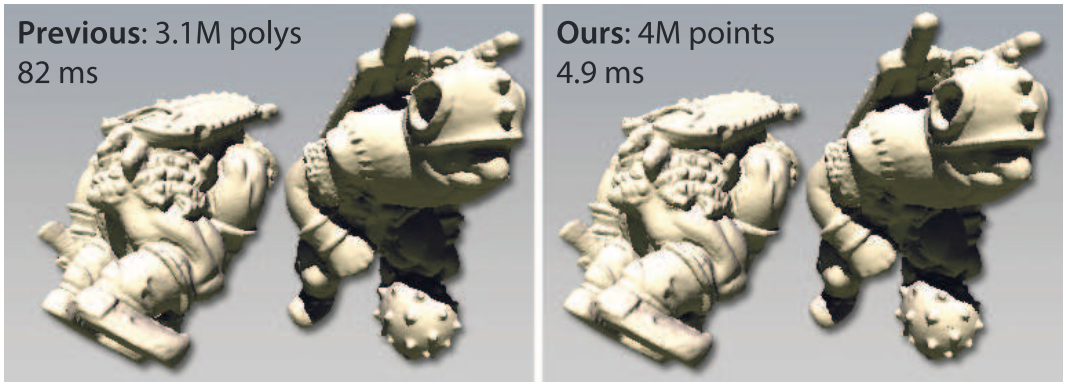



**Figure 0.11:** Temps d'exécution (coût de rendu non inclus) pour le modèle Grog (1.4 M sommets) utilisant OpenGL seul (gauche) et notre méthode (droite).

| Technique | Dragon 4 M | Grog 1.3 M |
|---|---|---|
| Polys | 3.2 s | 1.1 s |
| 8 k points (ISMs) | 114 ms | 120 ms |
| ManyLoDs, non-incrémental | 120 ms | 55 ms |
| ManyLoDs, incrémental | 8 ms | 5.3 ms |
| ManyLoDs, paresseuse | 4.9 ms | 2.9 ms |

**Table 0.2:** Performances de nos algorithmes versus autres approches. Nous utilisons la scène de la Fig. 0.7 avec 1024 VPLs et bougeons la source de lumière de $\approx$ 10 degrés par trame.

de maillages denses (par exemple QSplat [Rus+00]), nous pouvons paralléliser la coupe au travers de l'arbre (Fig. 0.11).

Nous avons mesuré les performances de notre algorithme pour une carte graphique NVIDIA GeForce GTX 480 (1.5 GB de mémoire vidéo) et une implémentation OpenGL 4.1. Les temps de calcul pour des maillages de taille différente sont donnés en table 0.2 et pour chaque composante individuelle de notre algorithme en table 0.3. Cela prend plus de temps de réaliser un recalcul complet du BVH car cela requiert d'ordonner tous les points et mettre à jour toutes les ressources GPU à chaque fois. Quand la scène inclut des déformations importantes de la scène (comme des explosions), la qualité du BVH se dégrade et nécessite un recalcul complet. Dans le cas d'animations de personnages, nous pouvons cependant réutiliser la structure de l'arbre déjà calculée, et ne mettre à jour que sa géométrie. Si la scène présente des incohérences temporelles partielles, comme dans le cas d'animations rapides ou de mouvements rapides de caméra, notre approche paresseuse de mises à jour partielles peut être utilisée à la place (Fig. 0.12).

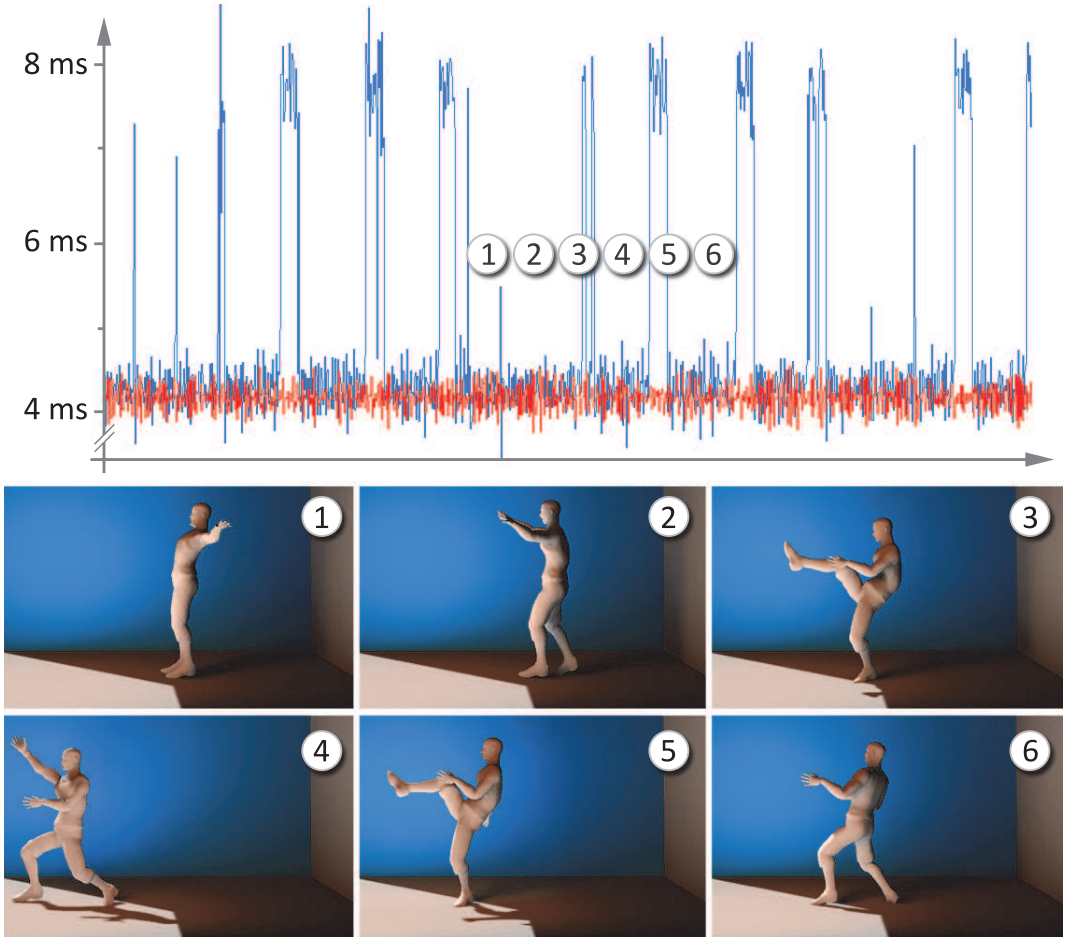



**Figure 0.12:** Courbes de temps de calcul pour chaque image (haut) pour l'animation *kicking* (bas). La courbe bleue montre le temps de calcul pour l'approche basique, la rouge montre le temps de calcul pour l'approche dite paresseuse. Le temps de calcul augment pour les mouvements rapides, c'est-à-dire, temporellement moins cohérents de l'animation (kicking en 3 et 5). L'utilisation de l'approche paresseuse permet d'éviter cela, avec une qualité indiscernable.

La différence perçue est insignifiante à cause de la nature de la plupart des effets d'illumination globale, qui ne contiennent que des basses fréquences.

### 0.4.4 Variantes de ManyLoDs

Bien que l'approche présentée de ManyLoDs permet d'obtenir de hautes performances, elle requiert également une grande occupation mémoire, à cause du pire cas envisageable pour la taille des listes $I$, $A$ et temporaire. La variante proposée dans cette section met l'accent sur l'approche basique de ManyLoDs, c'est-à-dire, à chaque trame la traversée de l'arbre commence à la racine de celui-ci et les paires noeud-vue ne peuvent être que dessinées, ignorées ou séparées.

| | Dosch<br>2.1 M ply<br>1.3 M pts | Grog<br>1.3 M ply<br>1.3 M pts | Sponza<br>72 k ply<br>1.3 M pts |
|---|---|---|---|
| Construction | 10 s | 9 s | 7 s |
| Mise à jour | 26 ms | 26 ms | 15 ms |
| Poly | 46 ms | 11 ms | 3.8 ms |
| $c = 1$ px | 3.0 ms | 2.7 ms | 3.4 ms |
| $c = 2$ px | 1.4 ms | 1.7 ms | 1.6 ms |
| $c = 4$ px | 0.7 ms | 0.8 ms | 3.8 ms |

**Table 0.3:** Performances de notre algorithme pour la création et la mise à jour du BVH et la coupe pour différents niveaux de qualité en 1024×1024. Notre condition de terminaison $c$ est la taille en pixels de la sphère englobante projetée sur la vue.

Pour éviter de stocker des coupes temporaires et des exécutions sur le CPU, tous les calculs sont maintenant effectués sur la GPU. Cela inclut le calcul des coupes de l'arbre ainsi que la rastérisation des points. Nous le faisons en exploitant NVIDIA's CUDA. Cet environnement permet un scan général [Ble89] au sein d'un noyau et beaucoup d'implémentations GPU différentes [Har+07; Dot+08] et spécialisées [Bax11; Hwu11] existent. Un opérateur de scan peut être utilisé pour *"stream-compaction"* [Ble93], et nous permet donc d'organiser les paires noeud-vue qui requièrent soit d'être dessinées soit d'être traîtées. Les itérations sont réalisées dans un noyau à l'aide d'un boucle `while`, au lieu d'être réalisées sur le CPU. Avant la boucle (Lst. 2), on initialise une pile *todo* de paires noeud-vue en mémoire partagée contenant toutes les vues couplées à la racine de l'arbre. Pendant chaque itération dans la boucle, chaque thread traîte une paire noeud-vue en prenant les éléments de la pile et réalise l'action appropriée (dessiner, ignorer, séparer). Puis un scan prefix est réalisé en mémoire partagée pour organiser et compacter les données. Ensuite, tous les éléments qui doivent être dessinés peuvent l'être efficacement à l'aide d'une procédure spéciale de rastérisation [Liu+10].

Nous avons mesuré les performances de notre algorithme utilisant un scan général GPU et un scan spécialisé de type ballot [Bax11]. Ce dernier offre de meilleures performances qu'un scan général car il bénéficie de l'accélération matérielle (avec l'instruction `__ballot`). Il offre également le bénéfice d'un usage réduit de la mémoire partagée et d'une implémentation plus simple. Les temps de calcul des coupes et des coupes plus le rendu sont donnés en table 0.4. Il est à noter que pour les 204×153 vues mesurées, on requiert une texture d'une résolution de 6528×4896 car la taille du rendu des vues est de 32×32 pixels.

En exécutant à la fois le calcul des coupes et le rendu de celle-ci dans CUDA, nous perdons le bénéfice de l'accélération matérielle pour la rastérisation. Néanmoins, nous avons gagné la possibilité de traîter des groupes de vues, ce qui devrait ouvrir

```
void TraverseEtFaireRendu() {
   uint tid = ThreadIdx.x;
   __shared__ stack< NoeudVue > s_PileTodo;
   __shared__ stack< NoeudVue > s_TempFinis;
   uint tid = TreadIdx.x;
   s_PileTodo.push( NoeudVue( 0, tid ) );    // paire (racine,vue)
   while( !s_PileTodo.EstVide() && !tousThreadsTermines() ) {
      NoeudVue paireCourante = s_PileTodo.pop();
      // dessiner, séparer, ignorer:
      int iEvenement = DecideEvenement( paireCourante );
      uint uiTodoIdx = Scan( /* ... */ );
      uint uiFinisIdx = Scan( /* ... */ );
      switch( iEvenement ) {
         case SEPARER:
            // mettre tous les fils dans la pile à uiTodoIdx
         case DESSINER:
            // ajouter le noeud-vue à la pile finis à uiFinisIdx
      }
      FaireRendu( s_TempFinis );
}}
```

**Listing 2:** Pseudo-code pour ManyLoD avec le rendu en place (*"in-place shading"*).

| Vues | Technique | #*"Cutfinders"* | Coupe | Coupe +Rendu |
|---|---|---|---|---|
| 34×25 | Scan | 64 | 63.8 | 80.3 |
| | Ballot | 64 | 61.6 | 78.1 |
| | Scan | 128 | 65.4 | 84.3 |
| | Ballot | 128 | 64.4 | 82.9 |
| 68×51 | Scan | 64 | 234.4 | 293.5 |
| | Ballot | 64 | 225.6 | 285.4 |
| | Scan | 128 | 134.2 | 170.5 |
| | Ballot | 128 | 131.7 | 168.3 |
| 204×153 | Scan | 64 | 1829.3 | 2309.6 |
| | Ballot | 64 | 1766.5 | 2243.2 |
| | Scan | 128 | 1012.3 | 1307.9 |
| | Ballot | 128 | 995.1 | 1289.1 |

**Table 0.4:** Performances de ManyLoDs avec rendu en place pour un nombre varié de vues. Nous avons mesuré les performances pour le calcul des coupes et le calcul des coupes combiné avec le rendu du scan et de la version ballot du scan en utilisant 64 ou 128 threads pour trouver les coupes par bloc (*"cutfinders"*). Chaque micro vue a une résolution de 32×32 pixels. Tous les temps sont exprimés en millisecondes.

la recherche pour des travaux futurs sur la génération optimale de ces groupes (schémas de clustering), ainsi que sur l'initialisation de la pile à l'aide d'une coupe représentant un compromis pour l'ensemble du groupe. Bien que notre technique de rendu en place ne peut pas obtenir les performances d'une implémentation

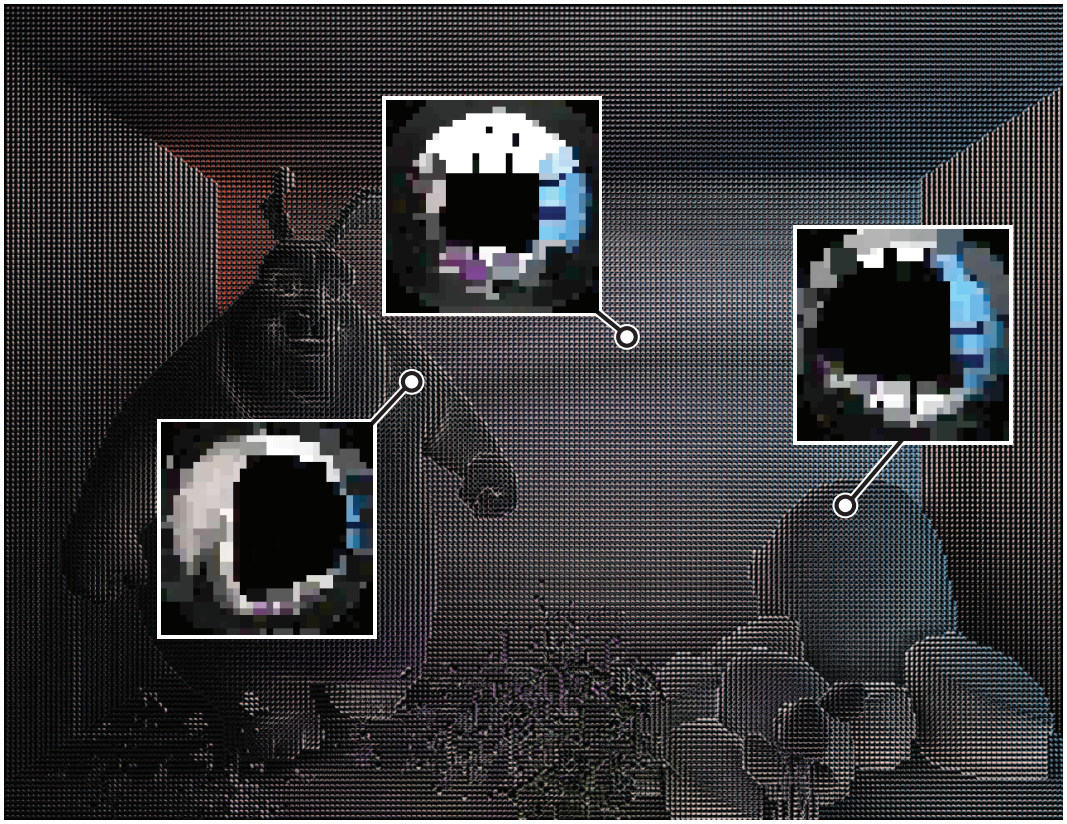


**Figure 0.13:** Rendu de 33280 vues (208×160), où chaque vue a une résolution de 32×32 pixels.

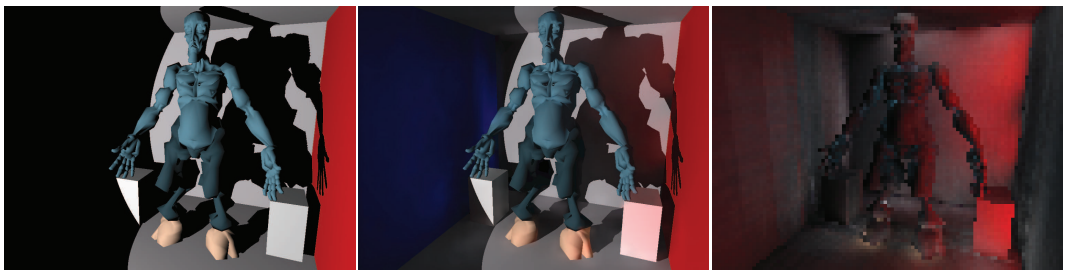



**Figure 0.14:** ManyLoDs avec rendu en place, l'illumination indirecte a été calculée avec une approche PBGI et 102×76 vues combiné avec un sur-échantillonage bilatéral. Chaque vue a une résolution de 64×64 pixels.

basée uniquement sur des shaders, nous obtenons des performances raisonnables et sommes capables de gérer plus de vues à la fois (Fig. 0.14), des vues de plus haute résolution ou des arbres de plus grande profondeur grâce à la réduction de la mémoire utilisée. Nous pensons que cette extension est utile pour des systèmes ayant des contraintes sur la mémoire disponible, comme la production de films où

le nombre de vues est très grand et le pire scénario d'utilisation excède largement celle-ci.

### 0.4.5 Conclusion

Nous avons présenté plusieurs méthodes pour traverser un BVH en parallèle pour calculer efficacement plusieurs coupes représentant différents LoDs. Notre algorithme peut accélérer diverses techniques d'illumination globale et exploiter des shaders, CUDA ou bien un environnement hybride CPU/GPU. En couplant les noeuds d'un arbre aux vues dans lesquelles on veut rendre la scène, nous pouvons exploiter le parallélisme et raffiner itérativement les coupes en utilisant la mémoire globale ou partagée.

Bien que nous n'utilisons que des arbres binaires comme structure pour les BVHs utilisés, d'autres arbres peuvent bénéficier de notre algorithme en adaptant les opérations de séparation des noeuds pour produire $n$ paires noeud-vue en fonction de l'arité du noeud considéré. Cela peut néanmoins nécessiter plus de calculs et possiblement une structure explicite de l'arbre. L'utilisation d'un arbre complet offre un bon compromis entre les performances de la mise à jour de l'arbre et une distribution de points suffisamment fine de la scène. Nous pensons que de futures architectures matérielles supportant le dessin de points et de triangles simultannément depuis le même shader permettraient de gérer des arbres ayant des triangles comme feuilles et des volumes englobants arbitraires stockés dans les noeuds internes.

La mémoire requise peut être réduite significativement en utilisant notre algorithme avec rendu en place et le *"stream compaction"* en CUDA. Bien que cette version est limitée à l'approche basique de ManyLoDs et ne peut exploiter la rastérisation matérielle, elle permet de traîter un groupe de vues sur un bloc tout en répartissant les calculs sur tous les threads lui appartenant. La mémoire ainsi économisée peut être utilisée pour traîter des arbres plus grands, plus de vues ou des vues de plus haute résolution.

Pour nos travaux futurs, nous voudrions considérer également la transparence, par exemple des media participatifs ainsi que le traîtement de données d'une manière *"out-of-core"*.

Dans la section suivante, nous définirons un LoD par pixel et analyserons une représentation intermédiaire de la scène utilisant de la synthèse géométrique pour supprimer les artifacts classiques d'aliasing dans un scénario de rendu différé (*"deferred shading"*).

## 0.5 Anti-Aliasing en rendu différé

Jusqu'à présent, nous avons utilisé des noyaux de synthèse géométrique pour créer des maillages lisses de haute résolution et calculer des effets d'illumination

réalistes. Cependant, le procédé qui consiste à appliquer des effets de lumière à une surface – appelé *"shading"* – dans un contexte temps-réel présente d'autres problèmes. Les procédures traditionnelles de rendu direct négligent souvent la visibilité finale dans l'image et appliquent le rendu à tous les fragments passant le test de profondeur. Les pipelines modernes de rendu différé [Dee+88] offrent un spectre plus large d'effets [Sai+90; Har+04] en reportant la phase de rendu dans une procédure travaillant en espace écran après que les problèmes de visibilité aient été résolus. Au lieu de faire le rendu de chaque fragment directement, un ensemble d'attributs est écrit dans une mémoire séparée. L'ensemble de ces espaces mémoire est appelé couramment un buffer géométrique (*"Geometry Buffer"* – G-Buffer) [Sai+90] contenant typiquement les informations de normales, profondeurs et albedo. Cependant, comme un pixel du G-Buffer correspond à un pixel dans le résultat final, des problèmes sévères d'aliasing peuvent apparaître. Une variété d'algorithmes existent pour résoudre ce problème [McC+99; Shi05; Yan+08; Kir+09; Thi09; Lau10; Sal+12; Cha+11; Lik+12], en utilisant des informations additionnelles sur la lumière [Eng08], ou en filtrant les valeurs de couleur [Lot09; Res09; And11].

Nous nous concentrerons sur les scénarios de sur-échantillonage (*"supersampling"*) et améliorerons les performances en concentrant le calcul effectué sur les zones contenant de l'aliasing, ce qui rend notre approche spécialement attractive pour les cas où le coût de rendu par pixel est important. Dans la suite, nous décrivons brièvement les moyens nécessaires pour combiner notre approche avec des techniques de filtrage anisotropiques efficaces.

### 0.5.1 Sur-échantillonage adaptatif pour anti-aliasing différé

On commence par remplir un G-Buffer sur-échantilloné de taille $(m \times w) \times (m \times h)$, où $w$ et $h$ sont la largeur et la hauteur de la fenêtre, et $m \in \mathbb{N}$ est le facteur multiplicatif de sur-échantillonage, en faisant le rendu en haute résolution. Chaque pixel de l'image finale correspond à une fenêtre d'échantillons de taille $m \times m$ dans le G-Buffer. On fait ensuite le rendu effectif (application de la BRDF) d'un seule pixel par fenêtre et stocke le résultat dans une mémoire séparée $\mathcal{S}$. Ensuite, on effectue une passe d'analyse de discontinuités qui prend en compte le contenu du G-Buffer ainsi que de $\mathcal{S}$ pour déterminer le nombres d'échantillons supplémentaires $k$ requis pour le rendu effectif final de chaque pixel. Finalement, nous faisons le rendu effectif du G-Buffer de manière adaptative, ce qui signifie que pour chaque pixel un nombre variable d'échantillons est considéré. La procédure complète est résumée en figure 0.15.

Pour chaque pixel, et une fenêtre d'échantillonage de taille $m$, notre passe d'analyse détermine le nombre additionnel d'échantillons requis en attribuant plus d'échantillons aux endroits présentant des discontinuités géométriques ou des discontinuités causées par le rendu. Pour la première, nous proposons les métriques suivantes.

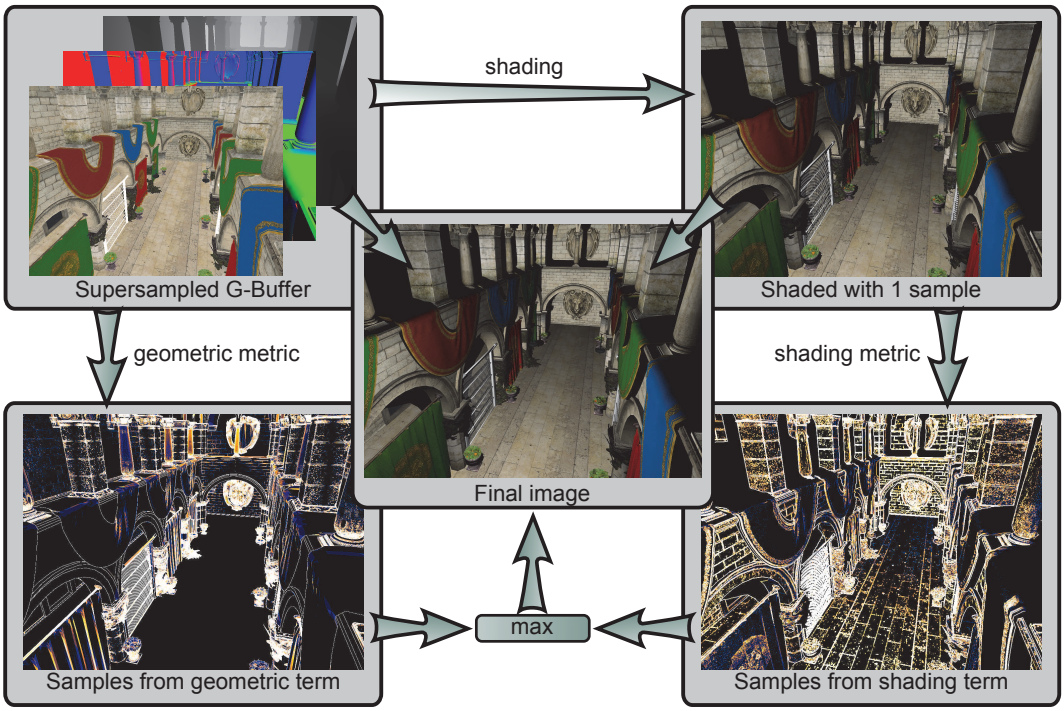



**Figure 0.15:** Aperçu de notre méthode: Notre passe de rendu différé effectue le rendu dans un G-Buffer agrandi, c'est-à-dire, chaque pixel de l'image finale correspond à une fenêtre dans ce buffer. Puis, le G-Buffer est évalué à l'aide de notre métrique géométrique. Le premier échantillon est rendu pour chaque pixel et notre métrique image est appliquée. Pour l'image finale, nous faisons le rendu de chaque pixel de manière adaptée, en se basant sur les résultats de nos métriques et de l'image précédemment rendue avec un échantillon par pixel.

**Profondeux MinMax (MM)** Le nombre d'échantillons requis pour cette métrique dépend des valeurs minimale et maximale de la profondeur ($d_{\text{min}}$ and $d_{\text{max}}$) présentes dans la fenêtre d'échantillonage:

$$k = c \times (d_{\text{max}} - d_{\text{min}}).$$

**Cône de normales (*"Normal Cone"* – NC)** Cette métrique se concentre sur les normales dans chaque fenêtre d'échantillonage. On calcule la moyenne normale $\bar{\mathbf{n}}$ de la fenêtre et compare toutes les normales $\mathbf{n}_{ij}$ à celle-ci, calculant le cône de normales centré autour de la normale moyenne. Le nombre d'échantillons donné par cette métrique est:

$$k = c \times \sqrt{1 - (\min_{ij}(\text{abs}(\text{dot}(\bar{\mathbf{n}}, \mathbf{n}_{ij}))))}.$$

| Sampling | | Reference | ND | MM | NC | DV | NV |
|---|---|---|---|---|---|---|---|
| 2×2 | temps réel | 3.80 | 3.29 | 3.01 | 3.36 | 3.22 | 3.27 |
| | ralenti | 18.56 | 11.52 | 10.09 | 10.72 | 10.06 | 11.18 |
| | MSE ($\times 10^{-4}$) | 0.00 | 8.26 | 12.88 | 9.03 | 13.07 | 8.60 |
| | Échantillons % | 100.00 | 38.10 | 31.20 | 32.50 | 30.90 | 34.60 |
| 3×3 | temps réel | 11.37 | 7.69 | 6.14 | 7.27 | 6.28 | 7.59 |
| | ralenti | 97.36 | 42.77 | 29.00 | 39.36 | 30.49 | 39.89 |
| | MSE ($\times 10^{-4}$) | 0.00 | 3.47 | 6.47 | 3.86 | 6.28 | 3.80 |
| | Échantillons % | 100.00 | 26.90 | 16.0 | 22.80 | 16.80 | 23.00 |
| 4×4 | temps réel | 21.48 | 15.68 | 10.90 | 13.66 | 11.54 | 15.33 |
| | ralenti | 174.76 | 89.35 | 54.45 | 61.02 | 61.03 | 81.87 |
| | MSE ($\times 10^{-4}$) | 0.00 | 13.55 | 25.71 | 14.28 | 24.17 | 14.60 |
| | Échantillons % | 100.00 | 23.20 | 9.90 | 20.00 | 11.20 | 18.30 |

**Table 0.5:** Performances et qualité de nos métriques géométriques. La référence évalue tous les échantillons pour chaque pixel. MSE indique l'erreur carrée moyenne (*"Mean Squared Error"* – MSE) par rapport à la référence. Le pourcentage d'échantillons indique le nombre total d'échantillons qui on participé au rendu de l'image finale. Tous les temps de calcul sont exprimés en ms. Notre métrique basée sur la profondeur minimale et maximale ( MM ) est la plus rapide, mais introduit une large erreur. La métrique avec l'erreur MSE la plus basse et le plus haut pourcentage d'échantillons est marqué en vert ( ND ).

**Variance de profondeur (*"Depth Variance"* – DV)** On calcule la variance des valeurs de profondeur $d_i$ dans la fenêtre pour le pixel correspondant, le nombre d'échantillons donné augmentant avec la variance:

$$k = c \times (\frac{\sum d_i^2}{n} - (\frac{\sum d_i}{n})^2).$$

**Variance de normales (*"Normal Variance"* – NV)** On calcule la normale moyenne $\bar{\mathbf{n}}$ de la fenêtre, et attribuons un nombre d'échantillons augmentant avec la variance des normales dans la fenêtre:

$$k = c \times \sum_i (1 - \text{dot}(\mathbf{n}_i, \bar{\mathbf{n}}))^2.$$

**Normales et profondeurs (*"Normals and Depth"* – ND)** Cette métrique combine les résultats de nos métriques NC et DV, et le nombre d'échantillons attribué est donné par

$$k = \max(k_{\text{NC}}, k_{\text{DV}}).$$

Une comparaison de la qualité et temps de rendu résultant de l'utilisation de ces différentes métriques (MM, NC, DV, NV et ND) est donnée en table 0.5. Notre métrique ND, prenant en compte les valeurs de normale *et* de profondeur, offre

d'après nos expériences l'erreur carrée moyenne la plus basse tout en ne nécessitant qu'un nombre bas d'échantillons. Nous utilisons cette métrique dans toute la suite. Cependant, les artefacts liés au rendu ne sont pas considérés dans aucune de ces métriques et ils requièrent une approche différente.

**Métrique image** Cette métrique utilise la texture rendue $\mathcal{S}$ comme entrée. Pour chaque pixel, l'intervalle $r$ de valeurs de luminosité est calculé en prenant en compte les quatre pixels voisins directs: $r = l_{\text{max}} - l_{\text{min}}$. Le nombre d'échantillons est alors défini comme:

$$k = \begin{cases} 0 & \text{si } r < \max(t_{minThreshold}, l_{\text{max}} \times t_{edgeThreshold}) \\ c \times r & \text{sinon} \end{cases}$$

Finalement, les résultats des métriques géométrique et image sont combinées en prenant le maximum des deux valeurs.

### 0.5.2 Echantillonage de la fenêtre

Le nombre d'échantillons requis pour chaque pixel ayant été calculé à l'étape précédente, on sélectionne les $k$ échantillons supplémentaires par pixel en échantillonant le G-Buffer dans l'ordre donné par les matrices de distribution ordonnée (*"ordered dithering matrices"*) [Bay73] (Fig. 0.16). Ces matrices sont construites de telle sorte que la distance moyenne entre deux nombres consécutifs est maximisée. Leur utilisation permet de garantir une distribution propre des échantillons qui sont pris dans le G-Buffer sur-échantilloné.

$$\begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix} \begin{pmatrix} 2 & 6 & 3 \\ 5 & 0 & 8 \\ 1 & 7 & 4 \end{pmatrix} \begin{pmatrix} 1 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 0 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}$$

**Figure 0.16:** Matrices de distribution pour les tailles $m = 2, 3, 4$ qui sont utilisées pour échantilloner le G-Buffer.

### 0.5.3 Filtrage anisotropique

Pour permettre le filtrage anisotropique nous devons différer l'accès des textures de la scène à la passe finale du rendu. Pour ce faire, nous stockons toutes les textures de la scène dans un tableau 3D et enrichissons notre G-Buffer avec les coordonnées de textures ainsi que l'index correspondant à la texture du pixel. Lors de la passe finale de rendu, nous utilisons les dérivées des coordonnées de texture (via les fonctionnalités `dFdx`/`dFdy` de GLSL) pour diriger le filtrage anisotropique (fonction `textureGrad`). Un tel traîtement résulte en de mauvais résultats à la frontière de deux matériaux différents, mais nous pouvons éviter ce cas en

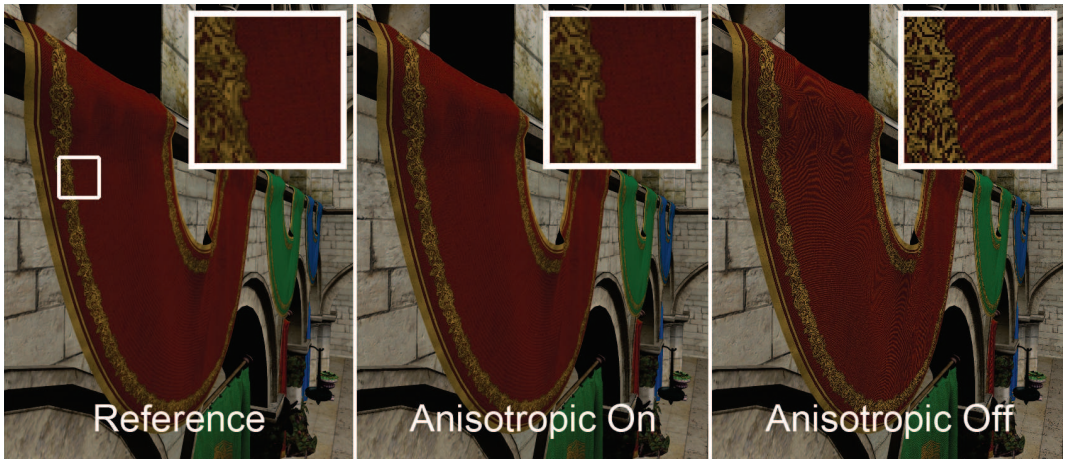



**Figure 0.17:** Gauche: référence utilisant un sur-échantillonage de taille 3×3 avec filtrage anisotropique, milieu: notre approche avec filtrage anisotropique, droite: notre approche sans filtrage anisotropique.

détectant les dérivées trop importantes dans la texture de coordonnées uv et la texture inidiquant l'index de texture dans la tableau 3D (Fig. 0.17).

### 0.5.4 Evaluation de plusieurs échantillons

L'évaluation de multiples échantillons peut être adaptée au matériel graphique utilisé. L'approche la plus simple est d'évaluer et accumuler les échantillons dans une simple boucle dans un shader.

### 0.5.5 Résultats

Nous avons comparé notre approche à l'approche *force brute* qui réalise l'évaluation de tous les échantillons avec le filtrage anisotropique activé. Les deux méthodes sont basées sur le sur-échantillonage en utilisant un grid ordonnée (*"ordered grid"*). L'erreur carrée moyenne (MSE), le ratio signal sur bruit (PNSR) et le pourcentage d'échantillons rendus sont donnés en table 0.6. Toutes les expériences ont été réalisées sur une machine équipée d'un processeur Intel Core i7 2.67 GHz et d'une carte graphique GeForce GTX 480 avec 1536 MB VRAM à une résolution de $1024 \times 768$ pour notre implémentation basée sur OpenGL. Nous avons utilisé la scène Sponza avec 280 K triangles et 194 K sommets pour nos tests.

#### Discussion et conclusion

Nous avons présenté une méthode pour améliorer les performances des méthodes d'anti-aliasing basées sur le sur-échantillonage (*"Supersampling Anti-Aliasing"*) dans le contexte de rendu différé. Nos résultats sont proches de la référence, et requièrent beaucoup moins de temps de calcul. Les métriques proposées offrent un bon compromis entre les performances et la précision avec une erreur controlable.

| Sur-échantillonage | Méthode | ms/image | ralenti | MSE ($\times 10^{-4}$) | PSNR dB | Échantillons % |
|---|---|---|---|---|---|---|
| 2×2 | Référence | 26.0 | 78.7 | – | – | 100 |
| | Notre | 22.9 | 72.2 | 5.0 | 91.14 | 57.8 |
| 3×3 | Référence | 34.9 | 158.36 | – | – | 100 |
| | Notre | 30.69 | 143.77 | 3.08 | 93.24 | 53.40 |
| 4×4 | Référence | 57.48 | 317.76 | – | – | 100 |
| | Notre | 51.68 | 268.35 | 3.79 | 92.34 | 53.14 |

**Table 0.6:** Performances de notre algorithme pour la scène Sponza. La technical référence evalue tous les échantillons pour chaque pixel d'une manière force brute. On donne le ratio signal sur bruit (*"Peak Signal-to-Noise Ratio"* – PSNR), l'erreur carrée moyenne (*"Mean Squared Error"* – MSE) ainsi que le pourcentage d'échantillons rendus par rapport à la référence. Tous les temps sont exprimés en ms.

Notre approche adaptative permet de concentrer la charge de travail aux endroits où plus d'échantillons sont requis, tout en réduisant le nombre total d'échantillons de moitié.

### 0.5.6 Conclusion/Travaux futurs

Dans cette thèse, plusieurs méthodes ont été présenté pour obtenir des résultats temps-réel de qualité similaires à ceux obtenus par les méthodes hors-ligne. Pour atteindre ce but, tous les algorithmes utilisent de manière intensive des noyaux GPU qui réalisent du traîtement géométrique à leur base et offrent une manière d'équilibrer entre la rapidité d'exécution et la qualité obtenue. Ils exploitent le matériel graphique moderne et utilisent des structures de niveau de détail pour l'abstraction de données au niveau face/maillage, vue ou pixel qui sont utiles pour les méthodes de nouvelle génération de surfaces de subdivision, illumination globale ou anti-aliasing dans un contexte temps-réel.

De plus, notre algorithme de subdivision ouvre des directions de recherche pour les algorithmes de subdivision temps-réel. Les cartes graphiques de génération à venir permettent une programmation GPU plus flexible, car les threads peuvent être créer dynamiquement sur la GPU sans appel au CPU. Cela devrait permettre plus de flexibilité par rapport aux schémas adaptatifs, dans l'esprit de [Nie+12]. La compression des tables de fonctions de base devient néanmoins rapidement un problème quand l'environnement a besoin de gérer plusieurs algorithmes de subdivision différents ainsi que des propriétés géométriques telles que des arêtes saillantes ou semi-saillantes.

En illumination globale, les caches sont souvent utilisés pour résoudre les problèmes de visibilité et déterminer l'irradiance à des endroits variés dans la scène. Notre algorithme ManyLoDs permet d'extraire les niveaux de détail variés – nécessaires pour un rendu efficace et adaptatif des caches – d'une représentation de la scène basée sur une hiérarchie de volumes englobants. Nous parallélisons

le calcul de la coupe en utilisant des noyaux géométriques vites qui peuvent facilement être exécutés par des centaines ou des milliers de threads sur la GPU. Les travaux futurs sur le sujet peuvent étendre cette idée en groupant des vues ayant des propriétés géométriques proches, par exemple la position 3D et/ou les normales. Des abstractions supplémentaires peuvent être utilisées, telles que la notion de représentant du groupe de vues. La coupe correspondant au représentatif devrait minimiser les coûts de calcul pour atteindre les coupes de l'ensemble des vues du groupe.

L'anti-aliasing en temps-réel pour le rendu différé est un sujet de recherches actif. Le calcul de plusieurs échantillons est requis pour réduire les artefacts visuels. Notre approche adaptative détermine le nombre d'échantillons approprié pour chaque pixel à l'aide de deux métriques différentes. Ceci est utile pour les approches utilisant le sur-échantillonage pour réduire le nombre total d'échantillons pour lesquels un rendu est effectué, spécialement si le coût de rendu par pixel est important, comme cela est courant en productions industrielles. Les travaux futurs sur le sujet devraient considérer d'introduire de la cohérence temporelle dans la définition de l'échantillonage et des méthodes de rendu plus rapides. Un rendu plus rapide pourrait être obtenu en analysant la contribution des échantillons à l'image finale et aux effets variés (tels que le *"subsurface scattering"* ou les effets de lumière spéculaire).

# Contents

CHAPTER 1

# INTRODUCTION

For computer-generated special effects and in offline-rendered feature films, image generation can take up to minutes or even hours, hence, time is often not a critical factor. However, *real-time* applications, such as computer games, augmented reality or medical applications, often strive to simulate worlds that the user can interact with. Usually, this implies various *actions* in the scene such as character animation, physical simulations, moving cameras, changing materials or even photo-realistic lighting. While handling a subset or all of these problems, many such real-time applications seek to generate 30 or even more images per second to ensure a fluent and comfortable user experience. Consequently, only a very limited time budget of fractions of a second is available for each image that needs to be generated. It gets even more difficult considering that today's real-time applications have to deal with a vast number of pixels due to increasing display resolution and wide-spread availability of multi-view stereoscopic displays which require multiple images to be generated.

The core idea of this thesis is to use geometry synthesis, processing and analysis to reach real-time performance in such demanding high-speed applications based on the close relationship between rendering and the necessary geometric computations. We offer new algorithms running on current graphics hardware architectures to achieve this goal. In particular, we address classical topics in computer graphics, namely subdivision surfaces, global illumination algorithms and anti-aliasing techniques, and present our contributions that are targeting real-time contexts.

Before we dive into the topics and contributions of this thesis we will cover some general background that is common for all remaining chapters and after which the reader can either read all chapters in the prescribed order or look at them individually.

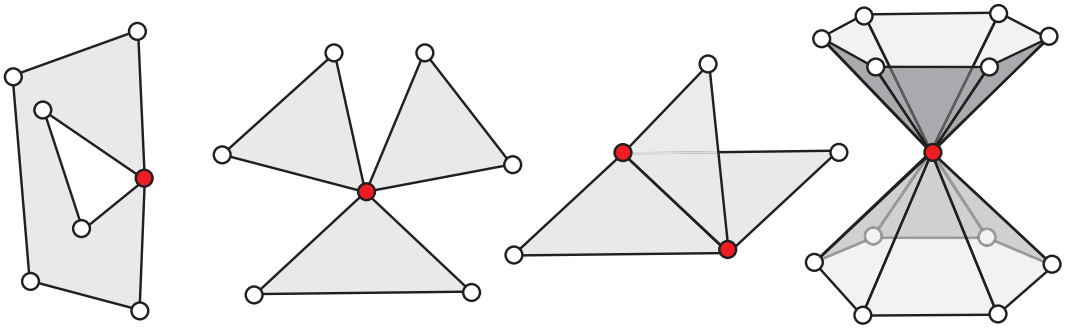


**Figure 1.1:** Non-manifold cases. Left to right: vertex repeated in the 8-gon, vertex shared by 6 boundary edges, edge shared by more than 2 faces, and upper and lower face-set is not connected through edge-adjacency.

## 1.1 Geometry for Shading

We will quickly introduce some basic principles for using geometry for shading that are valid for all chapters and we will see more complex examples of using geometry for rendering in chapter 2.

In computer graphics, geometry is either expressed implicitly or using a parametric representation. The former describes a surface $\mathcal{S}$ as a set of points and a function $f$ with $f : \mathbb{R}^3 \rightarrow \mathbb{R}$, so that $\mathcal{S} = \{\mathbf{x} \in \mathbb{R}^3 | f(\mathbf{x}) = 0\}$. Points belonging to the surface have a value of $0$ and others can be classified as inside or outside the shape in case of a signed function. Further discussion of implicit surfaces is out of the scope of this thesis and we refer the reader to [Bot+10]. In the following we will assume that our geometry is represented using a parametric, i. e. polygon-based, representation. We will call a piece of geometry a *mesh* $\mathbf{M}$, with $\mathbf{M} = \{V, F\}$, where $V$ is a set of vertices and $F$ a set of faces connecting elements of $V$ to form polygons. Further, we call $\mathbf{M}$ a 2-manifold if

1. every polygon $p \in \mathbf{M}$ connects exactly $n$ different vertices $v_1, \ldots, v_n$, i. e. $\forall v_i, v_j \in \{v_1, \ldots, v_n\}, i \neq j : v_i \neq v_j$
2. every edge $e$ is shared by exactly one or two polygons (i. e. $e$ is either *interior* or *boundary*)
3. every vertex $v \in V$ has exactly zero or two boundary edges
4. the set of all polygons $p_i$ sharing a vertex $v$ is connected regarding edge-adjacency between polygons.

For each condition a failure case is demonstrated in figure 1.1. For the remainder of this document we assume all meshes to be 2-manifold meshes consisting of triangles if not mentioned otherwise. To display a mesh on screen it needs to be rendered.

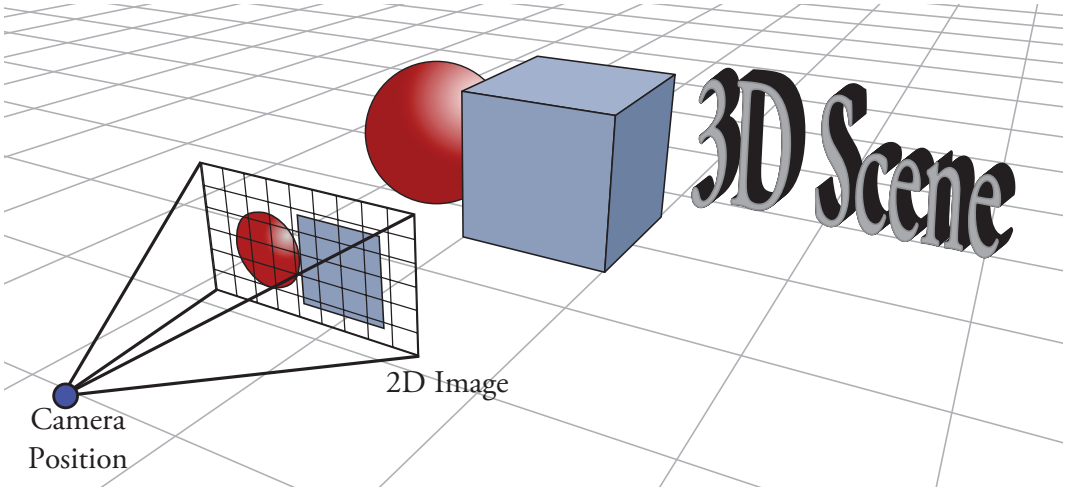



**Figure 1.2:** Rendering is the conversion of a 3D scene to a 2D image.

## 1.2 Rendering

The process of taking a scene description (geometry, materials, lights, textures) and a certain point of view, usually represented by a camera, and converting it to a synthetic image is generally called *rendering*. The camera is defined by its position in space and an image plane onto which the scene is discretized (Fig. 1.2). This simplified definition, called pinhole camera, assumes that all rays of light go through a common camera center. More sophisticated camera models exist that allow to render depth of field or lens flare effects but those are out of the scope of this thesis. For the image, a pixel-based, i. e. discrete, representation is common.

Many real-time algorithms use a *rasterization* based approach to render geometry. To do so, the input geometry is projected to an image plane and the coverage of pixels is determined. For order-independent results, e. g. to solve for visibility, this process can be coupled to a configurable depth-test in combination with an extra buffer called the *z-Buffer*. In the most common case, the $z$-Buffer is filled with the depth values of the rasterized primitive generated by interpolation of the primitive's vertices. Visibility can then be solved by querying the $z$-Buffer and only allowing colour output of subsequent rasterized fragments of another primitive if their corresponding $z$-value is smaller, i. e. closer to the camera.

Rasterization can be run efficiently in hardware which is, indeed, offered by most of today's graphics cards. Together with other complex tasks such as geometry processing and shading, it is made available through the programmable pipeline.

## 1.3 Programmable Pipeline

Today's most commonly used pipelines are the Khronos Group's OpenGL [Seg+12] and Microsoft's DirectX [Gee10] pipeline. As of writing this, the currently available

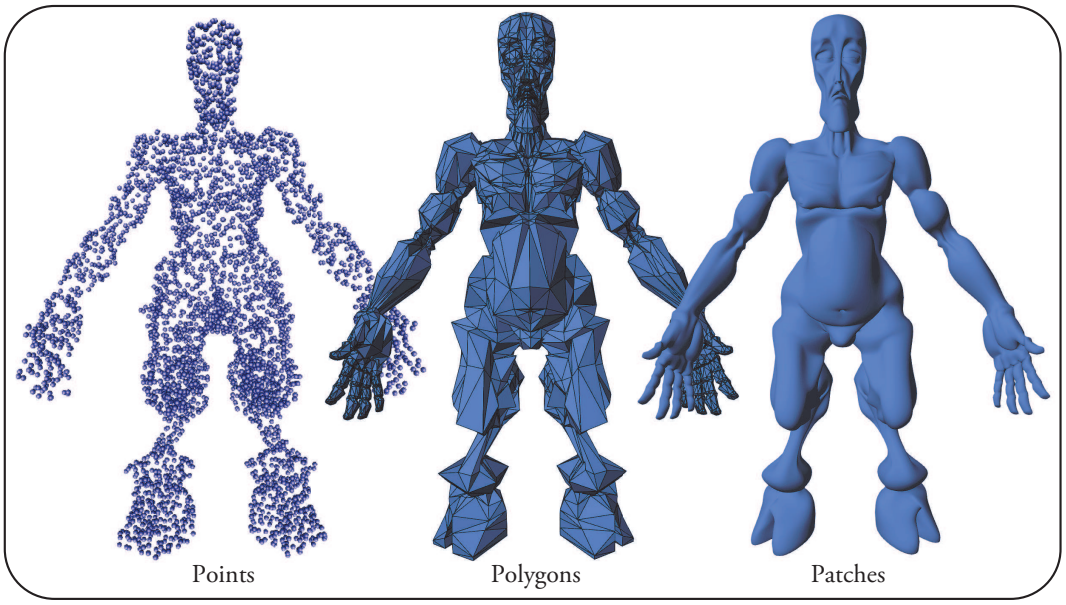



**Figure 1.3:** Different representations for geometry used in this thesis. Left to right: Point-sampled geometry, polygonal representation and higher order representation based on patches.

versions are OpenGL 4.3 and DirectX 11. These pipelines have evolved to be very flexible and highly programmable in contrast to older versions. Their specified behaviour is either implemented in software through library calls or processed directly on specialized graphics hardware controlled by a driver. Most recent changes to these programmable pipelines allow a wide variety of mesh processing, rendering, shading and general purpose computations to be executed on the Graphics Processing Unit (GPU) that were not possible before. The CPU on the other hand, as a more sequential processing unit, is then free for other tasks or only in charge of communicating with the GPU by dispatching the required draw and processing calls.

Rendering pipelines accept different input primitives and this thesis will focus on points, polygons (e. g. triangles, quads) and higher order surfaces (Fig. 1.3).

The primitives undergo different stages, some of which are user-programmable and some are of fixed functionality. The latter are often implemented directly in hardware for performance reasons and only partially customizable (Fig. 1.4). Often the input data undergoes certain transformations that lead to a screen representation (Fig. 1.5).

First, the vertices of the input primitives are transformed using a vertex shader. This commonly involves a change of basis, i. e. transforming the input into world-space, camera-space, etc. Additionally, during this stage the vertices can specify attributes that can be interpolated over the rasterized primitive. This turns out

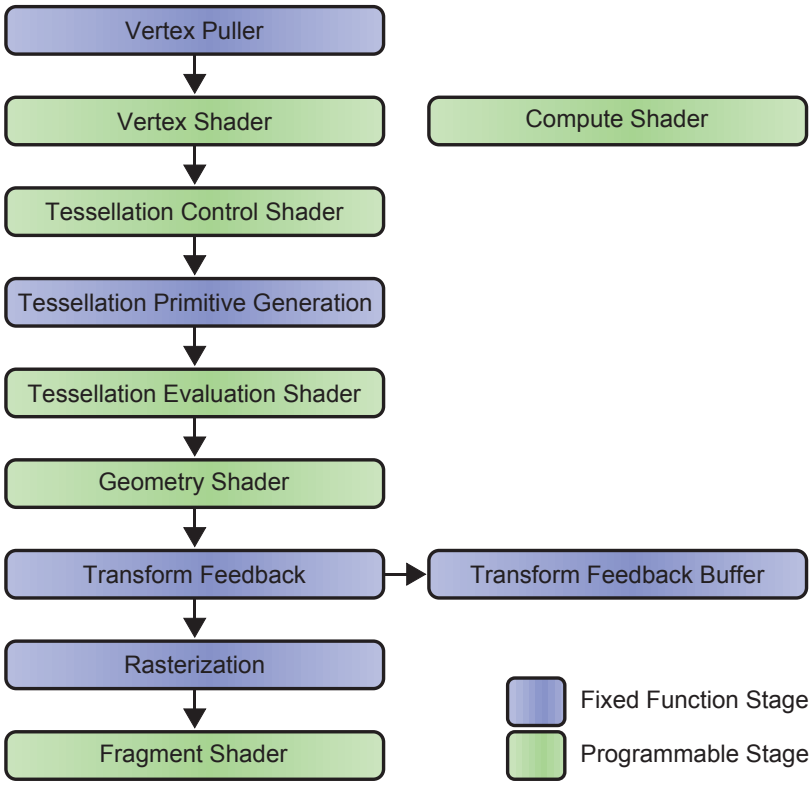



**Figure 1.4:** Programmable and fixed function stages of OpenGL 4.3.

to be a powerful tool and is, for instance, used for texture mapping, where the parametric coordinates need to be interpolated correctly to assure perspective correct texturing.

Then, polygons can be upsampled, i. e. tessellated, using the hardware tessellation stages composed of a programmable tessellation control stage, a fixed tessellation primitive generation stage and a programmable tessellation evaluation stage. The result of the tessellation control shader, user-specified splitting ratios per polygon and polygon edge, drive the primitive generation stage which takes care of creating the new set of vertices and the connectivity of the high-resolution polygon. For each newly created vertex the tessellation evaluation shader is executed. In contrast to the vertex shader, the user has access to all vertices of the low resolution polygon and each vertex generated by the primitive generator is assigned either parametric (for quads) or barycentric (for triangles) coordinates. The programmable tessellation stage is fairly new to the pipeline although many attempts have been made in the past on specialized hardware (e. g. ATI Radeon HD 2000–4000 and the Microsoft XBOX 360). This *delayed* upsampling has many advantages. Animation artists can work with easier-to-control meshes of low resolution and memory requirements as well as bandwidth are severely reduced. This is because the high resolution mesh is only generated when required and can be amplified directly on the GPU, e. g. using displacement maps. Further, many

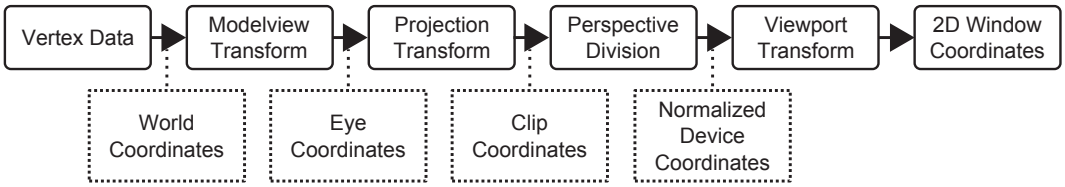



**Figure 1.5:** Common coordinate transformations for rendering.

aliasing artifacts can be reduced or even removed by using *as-small-as-necessary* polygons. For performance reasons tessellation can be done adaptively [Mun+08; Fis+09] using view-dependent metrics [Dyk+04; Bou10].

The primitives are then passed on to the geometry shader stage, where they can be processed in groups, deleted or modified. Even further refinement is possible but not recommended due to decreasing performance with increasing output. The result of this stage can be streamed out to a buffer and potentially be fed back into the pipeline. After the geometry shader stage, the primitives undergo frustum and backface culling with subsequent clipping if rasterization is requested. Rasterization will convert the primitives to a pixel representation (either on screen or to a texture), and for each fragment, i. e. pixel or subpixel of the rasterized primitive, a fragment shader is evaluated.

The notion of parallelism is important to notice here. Conceptually, each input primitive can be treated separately thus enabling parallel processing of all primitives. And also within stages processing is parallel, e. g. each pixel is shaded independently. For specialized implementations or to overcome pipeline limitations, parallel GPU computing environments such as CUDA or OpenCL can be used. Alternatively, especially on older hardware, pipeline stages were intentionally *misused* to exploit the GPU's processing power. Most recent Application Programming Interfaces (APIs), however, close this gap by adding a compute shader stage that is independent of all other stages. This allows for parallel processing of graphics resources such as buffers and textures within a common (graphics) framework, and switching to a compute environment such as CUDA or OpenCL can often be avoided.

Although most stages are optional, e. g. the tessellation stages can be disabled, this design dictates certain processing rules. For instance, recursive tessellation of polygons requires a loop involving the transform feedback stage, and the rasterization stage is not accessible directly from the compute shader.

For writing efficient GPU programs, it is essential to understand the underlying hardware architecture and its limitations.

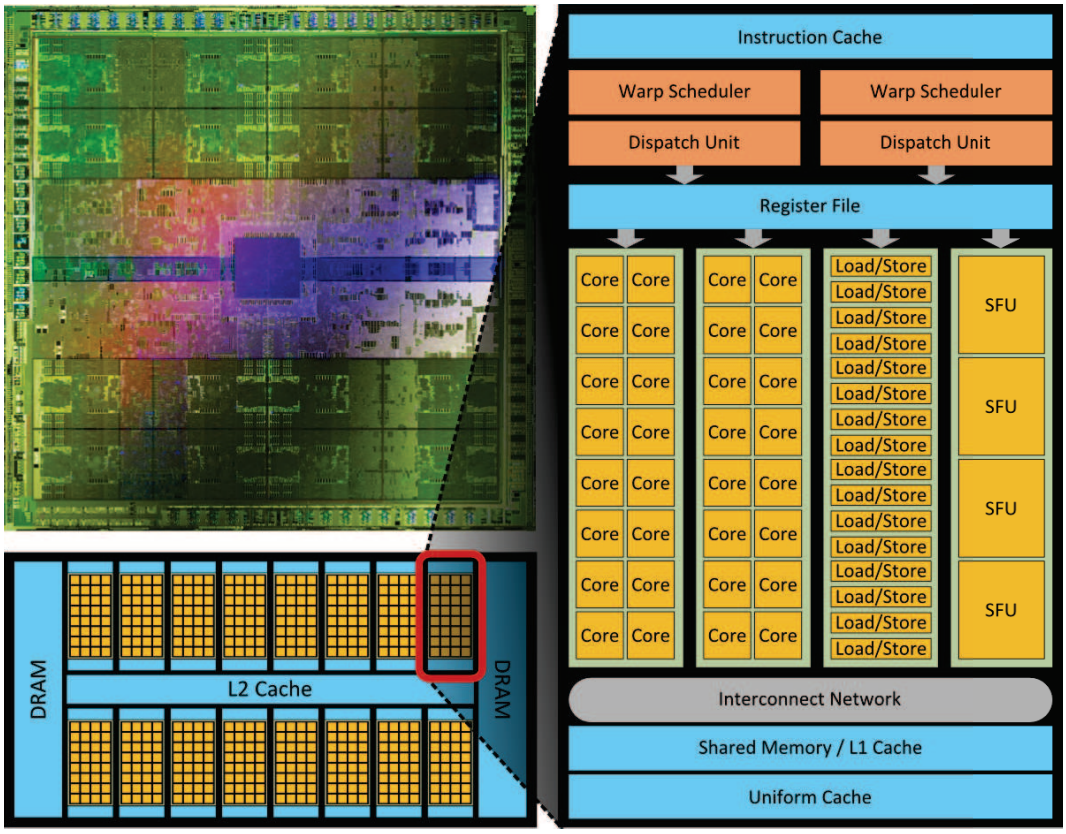



**Figure 1.6:** GPU Architecture of the NVIDIA Fermi series. Top left: Image of the physical die with circuits. Lower left: Conceptual view of the Fermi architecture with many Streaming Multiprocessors (SMs) each having access to the L2 Cache and DRAM. Right: SM overview. Each SM manages several cores that run threads based on the dispatching of the scheduler. All cores have access to the L1 Cache/Shared memory of their SM. Illustration adapted from [Cor09].

## 1.4 GPU Hardware Architecture

Today's GPUs are very well suited for fast parallel processing. Programs, so-called *kernels*, are executed in parallel on different threads. The parallel execution combined with the hierarchical organization of threads (several threads form a warp, multiple warps form a block, a group of blocks is a grid) allows to solve many data-parallel processing problems.

Although hardware manufacturers come up with new hardware each year or even more frequently, for a basic understanding of the parallel processing power we will shortly describe such a GPU architecture at the example of the NVIDIA Fermi series (Fig. 1.6) using CUDA terminology.

In Fermi, as also mentioned above, threads are organized hierarchically, where at the top the GPU consists of 16 so-called Streaming Multiprocessor (SM) of which

one of them is disabled/reserved. Each SM is capable of running groups of threads called *warps*, consisting of 32 threads each, in parallel directly on the chip. Because all threads of a warp are executed simultaneously, no explicit synchronization is required for threads within the same warp regarding cache coherency (i. e. cache is always *in sync*) and inter-thread communication (i. e. no locking required). However, Fermi is able to execute two warps per SM, each scheduled by one of the two warp schedulers. Because both warps are executed independently from each other, explicit synchronization and cache updates are necessary for correct inter-thread communication across warp boundaries. The same is true for threads executed on different SMs and regarding global memory due to out-of-order block execution. Fermi's SMs have 64 KB RAM each for shared memory and L1 cache whereas the global memory available for all SMs is large (up to 2 GB depending on the card) and supported by a unified L2 cache.

At the very low level, threads execute commands from the instruction cache in parallel, interacting with Special Function Units (SFUs) for operations such as square root and cosine, Arithmetic Logic Units (ALUs) for integer operations, or Floating Point Units (FPUs) for floating point / double precision calculations in full compliance with the IEEE 754-2008 floating point standard. All memory transactions are realized by Load/Store Units that interact with the cache or DRAM.

From a programmer's point of view, the most essential thing is that each thread is given a unique ID to identify its position in a warp, block and grid of execution. However, the execution order is unknown, i. e. blocks are not necessarily executed sequentially and warps can be activated/deactivated on a SM to hide memory access latency. Nevertheless, in computer graphics many algorithms can be implemented efficiently by hierarchically splitting the data to be processed and correct mapping to the underlying hardware. The input data, e. g. a 2D image (grid of pixels), can be divided into subparts (horizontal/vertical lines or tiles – *blocks* – of pixels). Each pixel can then be processed by a single thread assuming an embarrassingly parallel algorithm. Finally, programmers have to be aware that the simultaneous execution of threads within a warp is fastest if the corresponding threads take the same steps of execution regarding branching. Otherwise computations for all branches of execution are evaluated and unnecessary results are discarded. If that is the case, the threads are said to *diverge*. Nonetheless, even difficult algorithms can often be implemented using cooperative thread arrays with inter-thread communication.

Other graphics architectures from different vendors such as Intel or AMD (former ATI) exist with similar functionality. The biggest effort of innovating this pipeline completely was done by Intel in 2008. Their Larrabee [Sei+08] architecture was announced to support programmable blending, order-independent transparency as well as an instruction set that would allow prefetching of data into L1 and L2 cache (i. e. explicit cache control) to hide memory latency issues even further. The target was to increase performance by using a software scheduler instead of


**Figure 1.7:** Screenshots from various games. Although high quality texture maps improve image and shading quality, creating smooth and nicely looking geometry is still a challenging problem, especially for animated characters. This is most obvious when silhouettes are undertessellated (top left image with closeup) and a trained eye can often differentiate between local tessellation schemes and nicer subdivision surfaces. Top right: High quality global lighting effects are still difficult to achieve in real time and often low quality solutions are used. This results in pictures that often look *unnatural* due to simplified lighting calculations that avoid the simulation of complex but important interactions of light with the scene. The bottom right picture with closeup reveals aliasing artifacts due to the low number of samples per pixel that were used. As a result, the elements of the fence seem disconnected.

a hardware one, the support of load and store operations from non-contiguous addresses for groups of threads and on-the-fly render-target dependency analysis for parallel drawing processes. However, this architecture never reached the consumer market but is said to have inspired upcoming technologies such as Intel's Xeon Phi series.

## 1.5 Contribution

The algorithms described in this thesis solve some very common problems in real-time graphics (Fig. 1.7). Under the hood, geometry processing algorithms are used that affect various parts of the rendering pipeline (Fig. 1.8). Special focus will be put on the efficient mapping of data structures and algorithms to the GPU. As a key-concept, we will use adaptive approaches that allow fine-grained control over

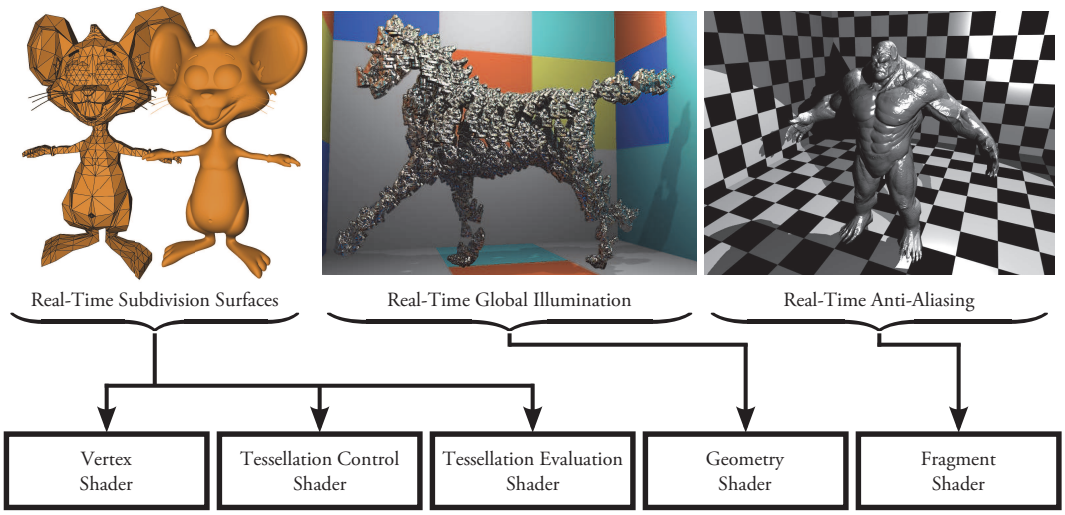



**Figure 1.8:** The contributions in this thesis are focused on, but not limited to, certain pipeline stages, involving real-time subdivision using the tessellation stages (Chap. 2), real-time global illumination exploiting the geometry shader (Chap. 3) and adaptive anti-aliasing in a fragment shader (Chap. 4).

the target Level-of-Detail (LoD) for the underlying problem domain. This domain is a stage-dependent view on the geometry in the form of meshes or faces, cuts through a hierarchical point-sampling of the scene or a pixel-based intermediate representation of the scene.

In particular this thesis makes the following contributions:

1. A new GPU method for synthesizing higher order smooth surfaces known as subdivision surfaces, together with a GPU kernel that builds upon precomputed tables of basis functions and which supports a wide range of interpolating or approximating subdivision schemes for dynamically animated triangle or quad meshes. Even further, our approach is compatible with adaptive tessellation methods and supports recent hardware tessellation units.

2. An algorithm for computing multiple cuts through hierarchical LoD structures such as a trees. This is especially useful for accelerating global illumination techniques that rely on such trees for visibility-dependent computations and, thus, allows a wider range of applications to run these algorithms in real time.

3. A novel approach to perform anti-aliasing in a deferred rendering scenario based on a metric that takes an intermediate geometric representation of the scene as input and drives the shading of the final image. This method provides a good tradeoff between speed and image quality for the expensive process of surface shading.

## 1.6 Thesis Organization

This thesis is organized as follows. Chapter 2 starts with a reminder on how subdivision surfaces are created and why this process is difficult to map to current GPUs. Then, we present our method for synthesizing these surfaces completely on the GPU and in real time together with the underlying data structures that drive our small-scale GPU kernel.

Chapter 3 reviews the theoretical foundations of simulating global illumination effects for the purpose of creating photo-realistic images followed by our ManyLoD algorithm that accelerates these heavily geometry-based computations for point-based rasterization-like approaches.

Chapter 4 describes how aliasing artifacts that occur during rendering can be reduced using supersampling which can be sped up by our metric that allows for faster adaptive shading by using only relevant information from the intermediate geometric representation of the scene.

Following that, is the conclusion of this thesis in chapter 5 and an outlook on related open problems and possible solutions in chapter 6 that should open up and inspire future work for using geometry synthesis to solve real-time graphics problems.

CHAPTER 2

# REAL-TIME SUBDIVISION SURFACES


**Figure 2.1:** Subdivision surfaces are well-known from offline rendering and part of many character rendering pipelines to create smooth surfaces.

Subdivision surfaces are a wide-spread means of creating high-resolution smooth surface geometry from a low-resolution input mesh and very popular in feature animated movies (Fig. 2.1). They are particularly useful when dealing with characters but also for organic shapes in general, and are heavily used in common animation pipelines. For that purpose, only the coarse input geometry – the *control mesh* – needs to be animated and the subsequent upsampling procedure creates the smooth, high-resolution surface.

However, their usage in real-time applications is usually limited and the direct mapping to the programmable pipeline and the corresponding stages is difficult or slow. Often, simpler local tessellation schemes or subdivision approximations are used, e. g. Phong Tessellation [Bou+08b], Gregory Patches [Loo+09], ACC [Loo+08], QAS [Bou+07a] or PN-Triangles [Vla+01].

Nevertheless, running subdivision algorithms efficiently in real time is feasible and even possible provided the information in this chapter.

First, the geometrical background of subdivision surfaces will be revised shortly in section 2.1 which will reveal their underlying recursive nature and explain why these algorithms are particularly difficult to implement on graphics hardware.

We will then present our real-time solution to this problem in section 2.2 which is based on small-scale geometry processing GPU kernels combined with a precomputation step. Our technique takes advantage of the linear nature of subdivision surfaces and also allows intermediate and adaptive LoDs. The visual smoothness can be controlled using Subdivision Shading. Our approach is general enough to support dynamic character animation as well as a wide range of subdivision algorithms. Integration into hardware or software tessellation pipelines alike can be done seamlessly. We demonstrate our technique for a variety of dynamic meshes and compare it to subdivision substitutes in section 2.3 before we conclude in section 2.4.

## 2.1 Background

This section summarizes the necessary background information regarding subdivision surfaces and real-time tessellation on which our method (Sec. 2.2) builds upon. Included is a description about fast curved surface models that will be used later for visual comparisons.

### 2.1.1 Subdivision Surfaces

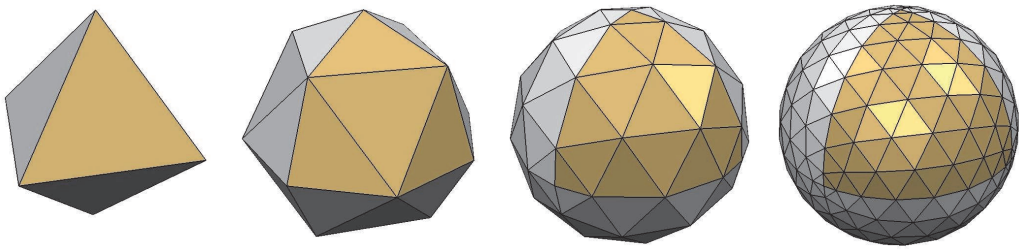


**Figure 2.2:** Subdivision is performed by recursively applying a subdivision scheme to a mesh (left to right). Images taken from [Bot+06].

A subdivision surface [Zor+00] is a smooth parametric surface of arbitrary topology defined by a base polygonal mesh (domain) and a subdivision scheme. A subdivision scheme defines a smooth surface entirely by either approximating or interpolating the vertices of the base mesh, called *control vertices*. Well-known approximating schemes are the Catmull-Clark scheme [Cat+78] for quad meshes, the Loop scheme [Loo87] for triangle meshes and the Loop-Stam scheme [Sta+03] for tri-quad meshes. When interpolation of control vertices is required the Modified Butterfly scheme [Dyn+90] for triangle meshes can be used for instance. Further, the schemes can be characterized by their definition of regular and extraordinary vertices, their splitting convention (face/vertex split) and their geometric continuity properties for regular meshes (most schemes are at least $C^1$-continuous).

Each scheme is defined by a collection of subdivision masks that are applied during a recursive tessellation process interleaved with local filtering operators, and

generates a denser subdivision mesh closer to the (continuous) limit subdivision surface at each step (Fig. 2.2). The set of masks usually covers boundary and interior cases but sometimes additional masks offering more fine-grained control over darts, creases and semi-sharp edges are included as well. For the remainder of this chapter we will rely on the following notation.

**Notations:** We denote $\mathbf{M}^0$ as the base mesh and $S$ as a given local and compact subdivision operator. $\mathbf{M}^k = S^k(\mathbf{M}^0)$ is the subdivision mesh after $k$ steps of subdivision applied to $\mathbf{M}^0$. Any mesh $\mathbf{M}^i$ is composed of a vertex set $V^i = \{v^i_j\}$ and a face set $F^i = \{f^i_j\}$.

For many schemes, subdivided vertices can be projected to their limit position directly, therefore sampling the subdivision surface exactly. When a subdivision scheme is derived explicitly from a spline basis, an exact, recursion-free evaluation allows to sample the surface at arbitrary parameter values [Sta98]. Recursive or parametric evaluations are usually both too expensive for practical interactive applications. However, as any point of a subdivision mesh can be defined as a weighted combination of the vertices of the base mesh, it is possible to precompute tables of basis functions (BFTs) [Bol+02]. The weights contained within these tables are arranged in groups. A group defines a set of linear combinations of base vertices and each so-defined combination provides the position of a fine vertex $v$. Each group corresponds to a common configuration and connectivity of faces rather than defining the weights for all fine vertices of a particular mesh. This way, each group can be used for all faces of the input mesh that have the same connectivity and the complete set of BFTs can be reused for a variety of meshes in the scene.

The weights of the linear combinations depend not only on the connectivity but also on the tessellation level. However, they can be generated for limit positions as well, i. e. the projection of $v$ onto the subdivision surface, which is equivalent to an infinite number of subdivision steps. Overall, BFTs can be used to avoid the recursive evaluation which is either difficult or slow in performance when conducted in a shader.

Alternatively, subdivision surfaces can be sampled efficiently on the CPU at rational parameters by combining translation and scaling functions [Sch+07], therefore computing the basis functions on-the-fly. However, this is not easily adaptable to a GPU implementation.

### 2.1.2 Real-Time Tessellation

Today's GPUs provide good means for running tessellation adaptively and in real time. To do so, the input mesh is supersampled, i. e. the input polygons are replaced by a large number of fine polygons and the fine vertices are displaced according to a given function defined on the base surface. Optionally, the smooth on-surface signal can then be deformed, e. g. using displacement mapping (Fig. 2.3). In real-time applications, the entire process of synthesizing and displaying the

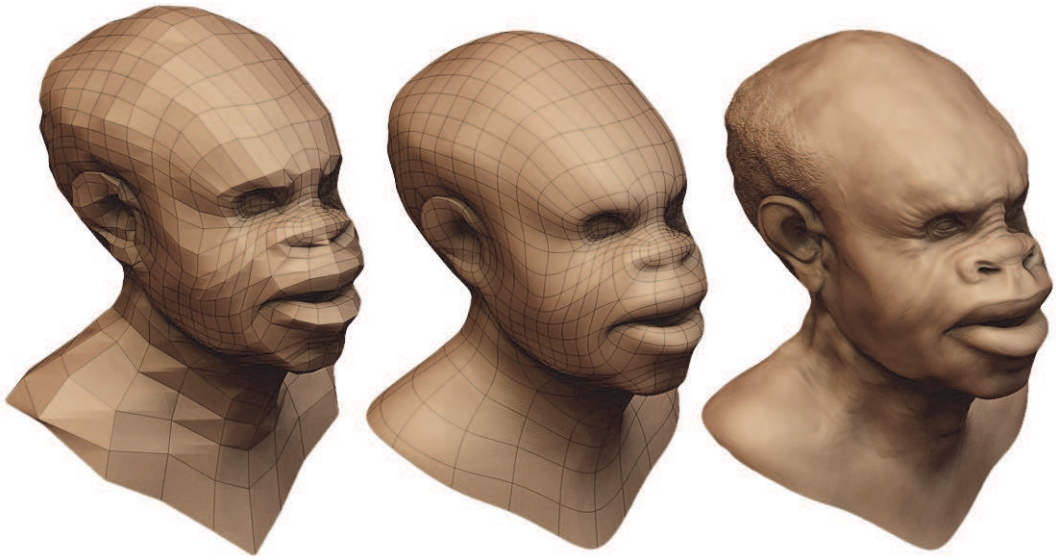


**Figure 2.3:** Left to right: Low resolution input mesh, smooth high resolution subdivision surface, per vertex displacement of the subdivision surface. Images taken from [Pix04].

mesh is repeated at each frame, while the application is responsible for the coarse (dynamic) mesh only. Often, these low resolution meshes are preferred for the purpose of animation, physics simulation or any kind of interaction. Real-time tessellation methods can be implemented on any GPU equipped with vertex shading capabilities using instancing [Bou+05a; Bou+08a], specific extensions (`GL_AMDX_vertex_shader_tessellator`) or recent approaches [Gru12] based on vertex-ID manipulation. Alternatively, GPU computing environments [Sch+09] or current hardware tessellation stages (Sec. 1.3) can be used. In many cases such methods offer the ability to tessellate the input mesh *adaptively* (Fig. 2.4), while providing watertight high resolution meshes with spatially varying density. Although subdivision surfaces may appear as a natural application for real-time tessellation, recursive or parametric evaluation are often considered too slow for typical interactive applications and a number of alternative curved surface models have been developed for this purpose.

### 2.1.3 Fast Curved Surface Models

Substitutes to subdivision surfaces [Ni+09] offer visual smoothness without the need for a recursive evaluation but are usually limited to lower geometric continuity. Their representations are often based on low-degree spline patches [Yeo+09]. For instance, a *Curved PN Triangles* [Vla+01] patch is a triangular Bézier patch built solely from local information of the input triangle (vertex positions and normals). Such local schemes can only mimic higher order continuity by exploiting a similar strategy as Phong Normal Interpolation [Pho73]: a synthetic normal field is generated using interpolation over the input vertex normals, independent of the

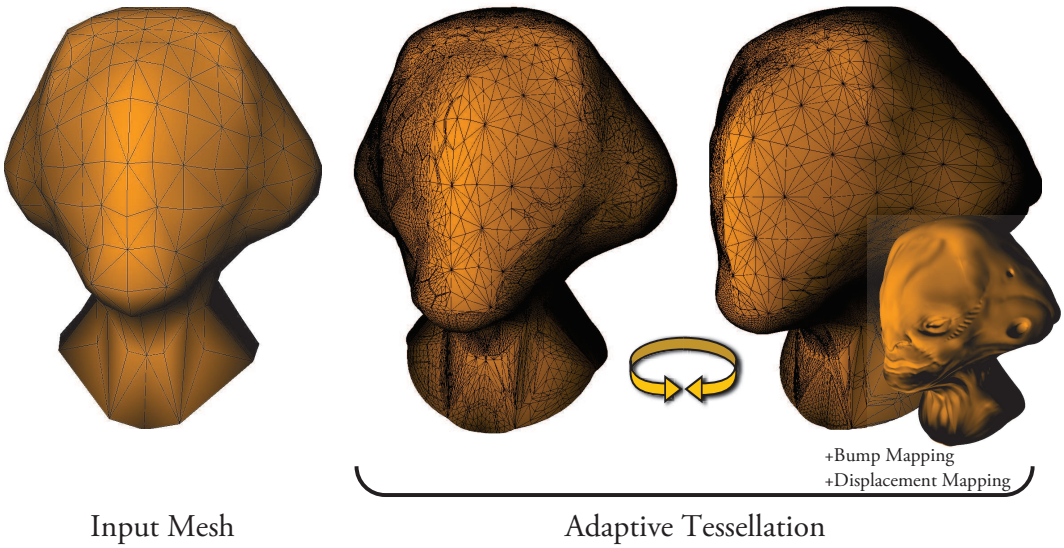



**Figure 2.4:** Adaptive hardware (Phong) tessellation. Left: input mesh. Middle and right: rotation and view-adaptive tessellation. Bottom right: adaptive tessellation combined with bump mapping and displacement mapping.

actual geometric differentials. This interpolation is quadratic [VO+97] in the case of PN Triangles and can be combined with a subdivision basis [Ale+08] as well.

Simpler operators such as Phong Tessellation [Bou+08b] can also offer an economic way to get rid of most of the typical visual artifacts. Each point is defined by a procedure similar to Phong Normal Interpolation. First, each point is projected to the tangent planes defined by the vertex positions and normals of the face to be upsampled. The result is interpolated across the patch using parametric interpolation. An additional *curvature* parameter can be used to interpolate between the so-defined position and the linear (flat) interpolated value.

A number of subdivision surface approximations have been proposed following a similar strategy. Boubekeur et al. [Bou+07a] use quadratic patches to approximate the geometry of a subdivision surface. The necessary vertices for this patch are found by performing one step of subdivision. The results are geometrically more pleasing due to the fact that a single step of subdivision adds vertices on each edge, thus, giving an indication of the direction into which each edge will converge. The visual smoothness is controlled using quadratic normal interpolation but a combination with Subdivision Shading is possible as well. Further approximations use bicubic Bézier patches [Loo+08] but often exhibit shading artifacts in the vicinity of extraordinary vertices due to discontinuities in the normal fields. To improve quality, Gregory Patches [Loo+09] can be used which provide a good approximation to the subdivision surface at the expense of lower geometric quality. This can cause inconsistencies in production pipelines when modeling tools based on true subdivision surfaces give slightly different visual results than the actual real-time application.

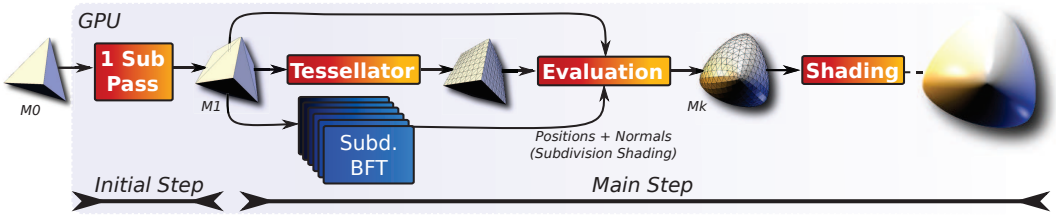



**Figure 2.5:** Our GPU subdivision mesh synthesis pipeline.

Although all of those methods produce smoother high-resolution versions of the input mesh, none of them can reproduce the high quality of a true subdivision surface as we will see later in a few direct comparisons to our method.

## 2.2 GPU Subdivision Kernel

In this section, we present an adaptive subdivision surface meshing algorithm which exploits real-time GPU tessellation to produce dynamic subdivision meshes on-the-fly. Our geometry synthesis kernel is oblivious to the particular subdivision scheme in use and can therefore be combined with all classical ones (i. e. stationary local schemes with compact support). We use basis function tables / tables of basis functions (BFTs) at *maximum level* to index an adaptive triangulation and exploit the same tables to generate a smooth normal field in a similar way as Subdivision Shading [Ale+08]. Our kernel runs at high frame rates for dynamic input base meshes with deep (adaptive) tessellation ratios and is fully compatible with current hardware tessellation pipelines (Sec. 1.3). On the contrary to subdivision substitutes, we do not aim at producing visually smooth surfaces only but rather propose to carry off computations from CPU to GPU for all applications exploiting subdivision meshes (e. g. high-end modeling packages) without the need to switch to a new representation.

### 2.2.1 Overview

The basic idea is to use the tessellation unit (either GPU emulated or hardware supported) for real-time, adaptive, on-the-fly upsampling and evaluate the produced fine vertices using BFTs. As usual with BFTs, several tables have to be generated, one for each input face connectivity configuration. Following the idea of Bolz and Schröder [Bol+02], we do not generate all the intermediate BFTs and directly precompute a maximum table, for instance at level 5 which corresponds to a tessellation pattern of $32\times32$ faces. The weights stored by these maximum BFTs correspond to limit projections: all vertices evaluated using these tables will lay on the (limit) subdivision surface. Therefore, there is no difference between a subdivision vertex at level 3 or 5, for instance, and alternative adaptive triangulations can exploit the same set of (maximum level) tables, regardless of the desired number of vertices and triangles. Our algorithm is summarized in figure 2.5 and makes use of three major components:

**Figure 2.6:** Left: Input coarse mesh. Middle and right: real-time adaptive subdivision of the animated input mesh using our method.

- An initial subdivision step on the GPU to isolate extraordinary vertices, i. e. after this step, we have $\mathbf{M}^1$ in GPU memory and each triangle of $\mathbf{M}^1$ contains at most one extraordinary vertex. Additionally, we compute the normals of $\mathbf{M}^1$ which will be used later for generating a smooth normal field using *Subdivision Shading*. Also, the normals for each triangle are in the same hemisphere after this step, which improves the visual quality when combined with Subdivision Shading.

- Uniform or adaptive tessellation is performed on $\mathbf{M}^1$, refining it directly to level $k$. This step can exploit current hardware tessellation units (Sec. 1.3) or GPU implementations (Sec. 2.1.2).

- A set of precomputed basis function tables $\{B^i\}$ are uploaded and stored on the GPU. At rendering time, vertex positions and normals of $\mathbf{M}^k$ are computed using linear combinations of vertices of $\mathbf{M}^1$, with weights stored in $\{B^i\}$. The spherical averages of the normals on the Gauss sphere are approximated by using a single normalization step [Ale+08].

BFTs speedup the evaluation of points on the limit subdivision surface in contrast to complicated and slow direct evaluation [Sta98] or GPU recursive emulations [Shi+05; Zho+07]. They can be obtained by generating *configuration meshes* which consist of a single triangle and its one-ring neighbourhood. All vertices are placed within the $xy$-plane except one which is moved to $z = 1$. This configuration mesh is subdivided with a given scheme and the basis functions are extracted from the $z$-values. This procedure is repeated for all vertices of the configuration mesh.

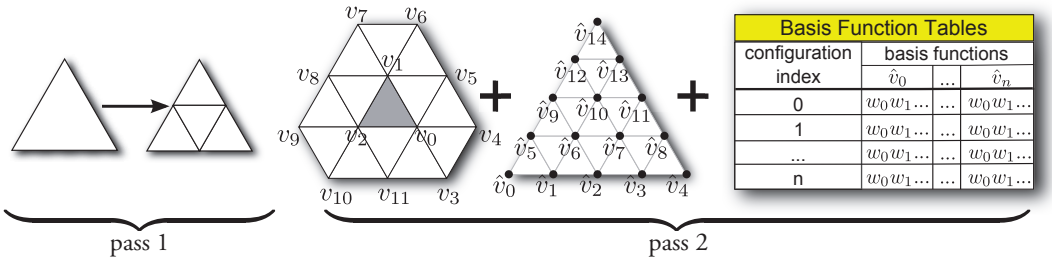



**Figure 2.7:** An input triangle $t^0$ is subdivided once on the GPU resulting in four triangles $t_i^1$ (pass 1). Each triangle $t_i^1$ is tessellated (pass 2 middle) – for instance by using hardware tessellation or by instancing a refinement pattern – and the related basis functions are queried. Finally, for each created fine vertex $\hat{v}_i$ the linear combination given by the weights $w_i$ and the vertices $v_i$ of $\mathbf{M}^1$ is computed.

Unfortunately, the number of basis function tables increases quickly for a mesh with arbitrary connectivity. However, applying a single subdivision step isolates extraordinary vertices and produces faces at level one with one extraordinary vertex at most. This mechanically diminishes the combinatorics for precomputation and the number of BFTs to store for a bounded valence (up to 18 in our experiments). We implement this initial subdivision step on the GPU and use $\mathbf{M}^1$ in all subsequent GPU steps. Consequently, the entire algorithm is executed on the GPU and the input coarse mesh can be provided either from the main (CPU) application (e. g. high end modeling packages such as Maya or 3DS Max) or from the GPU itself (e. g. GPU skinning, procedural geometry). After this step all faces have at most one extraordinary vertex and are ready for further tessellation and evaluation using BFTs. For the sake of simplicity, the discussion will be limited to Loop subdivision because triangles are ubiquitous for real-time rendering and the subdivision scheme produces a surface that is at least $C^1$-continuous everywhere (Fig. 2.6). However, our approach is not specific to this scheme and can be used with other schemes as well (e. g. Catmull-Clark, Butterfly).

### Preprocessing

To prepare the input mesh for our pipeline and to reduce the computational effort during runtime, several preprocessing steps are applied at initialization time. First, we load pregenerated BFTs as in [Bol+02]. We store them as floating point textures which allow random access from within a shader. Afterwards, we allocate the space required to store $\mathbf{M}^1$ on the GPU which will be filled using the initial GPU subdivision step. At this step we impose $\mathbf{M}^1$ to be a uniform tessellation of $\mathbf{M}^0$ to avoid any memory manipulation at runtime and directly specify its connectivity. The advantage is that $\mathbf{M}^0$ can now be animated on the GPU without modifying the memory layout as long as its connectivity does not change (i. e. $F^1$ is static). Note that the GPU memory cost for a given object is small and independent of its actual final resolution and that the BFTs can be shared by all objects in the scene (Fig. 2.7).

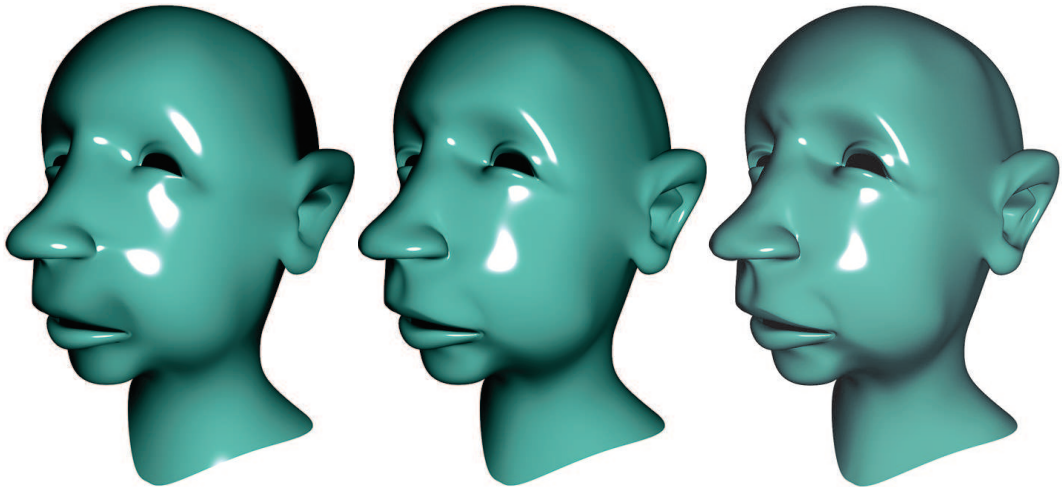


**Figure 2.8:** Left to right: $\mathbf{M}^5$ with the Subdivision Shading procedure starting at level 0, 1 and 2.

### 2.2.2 Initial GPU Subdivision Step

A single step of uniform subdivision multiplies the number of triangles by a small fixed ratio (low data amplification) but may have to handle a large spectrum of connectivity configurations. Again, each vertex $v_i^1 \in V^1$ is a linear combination of the vertices of $V^0$ and a large part of the evaluation cost can be cached by applying a similar strategy as BFTs but specialized to this single step: we record for each vertex of $V^1$ a *computational entry* consisting of weights and indices of its parent vertices in $V^0$ and update $V^1$ each time $V^0$ undergoes a deformation.

```
struct CompEntry {
   float m_fWeights[ n ];
   uint  m_uiIndices[ n ];
};
```

Normal vectors of $V^1$ are computed using a similar approach. This time, each *computational entry* for a surface normal of $\mathbf{M}^1$ has to keep track of the vertices which contribute to its calculation, namely its one-ring neighbourhood.

```
struct CompNormalEntry {
   bool  m_bIsBoundary;
   uint  m_uiIndices[ n ];
};
```

These vertex indices are encoded into the data structure ordered consistently in counterclockwise order together with a flag indicating whether the normal is located on a boundary. For the latter case the related triangle fan is treated as *open* and a different set of tables will be used in the main pass.

Note that the initial subdivision step could apply Subdivision Shading [Ale+08] directly using the normals of the coarse mesh and only a single computational entry map. However, we found the visual result to be too smooth sometimes, resulting in a loss of visual quality (Fig. 2.8). Similarly, one could delay the evaluation of geometric normals to a deeper level for starting Subdivision Shading at the expense of a lower frame rate. Mind that tangent masks can be used to compute geometric normals, which leads to the usual defects around extraordinary vertices [Ale+08].

**GPU Implementation** A GPU implementation of our initial subdivision step depends on the capabilities of the underlying hardware. On older hardware, which does not support random access to (vertex-)buffer resources, the attributes of $\mathbf{M}^0$ need to be converted to such resources in a preliminary pass using the transform feedback stage, where stream tokens should be understood as vertex positions, vertex normals, etc. Nowadays, buffers can be accessed in random order (e. g. using `GL_TEXTURE_BUFFER`) and no such conversion is necessary anymore.

During the first pass, each vertex of $\mathbf{M}^1$ is processed independently, either by instancing a single point $n$ times (old hardware), where $n$ is equal to the number of vertices of $\mathbf{M}^1$ or by using $n$ threads during the compute shader stage. In both cases the vertex is identified based on its ID and the corresponding *computational entry* from GPU memory is applied. Each vertex of $\mathbf{M}^1$ is then computed by evaluating the linear combination as encoded in the entry and the result is streamed into a random access shader resource.

In the following pass, the surface normals of $\mathbf{M}^1$ are computed similarly by again creating $n$ instances of a single point or launching $n$ threads. This time, normal computational entries are queried and combined with the vertices of $\mathbf{M}^1$ from the previous pass. Again, the result is stored in GPU memory, thus, completing $\mathbf{M}^1$ in GPU memory. Note, that a variety of algorithms [Jin+05] can be used during this step for the computation of a vertex normal.

Any additional vertex attributes can be treated with a similar strategy using multiple render targets / streaming to buffers and corresponding computational entries. Texture coordinates can use the same weights as positions [DeR+98] when certain artifacts can be accepted, otherwise strategies for reducing the distortion should be applied [He+10].

Note, that for older hardware instancing of a single point is very fast and can be seen as a thread creation for each vertex (position, normal, etc.). The initial GPU step does not, overall, represent the bulk of the computation.

### 2.2.3 Main GPU Subdivision Step

Positions and normals computed at the aforementioned step as well as the BFTs are used as random access shader resources in the main step. In our pipeline, we represent each patch $f_i^1$ by a simple data structure.

```
main() {
   // load the patch based on gl_InstanceID (instancing)
   // or gl_InvocationID
   BFTPatch p = LoadPatch( ... );

   // load the control attributes of M^1, e.g. direct access
   // via the output layout specifier of the tessellation
   // control stage and glPatchParameteri
   vec3 vPatchPositions[] = LoadPatchPositions( p );
   vec3 vPatchNormals[]   = LoadPatchNormals( p );

   // load the basis function tables for the fine vertex
   // using either gl_Vertex.x (instancing)
   // or the subindex as mentioned above
   float fBFT[] = LoadBFT( p.m_uiIndexBFT, ... );
   vec3 vPosition = vec3( 0.f, 0.f, 0.f );
   vec3 vNormal   = vec3( 0.f, 0.f, 0.f );

   // linear combination using BFTs
   for ( uint i = 0; i < p.m_uiOneRingSize; ++i ) {
      float fWeight = fBFT[ i ];
      vPosition += fWeight * vPatchPositions[ i ];
      vNormal   += fWeight * vPatchNormals[ i ];
   }
   gl_Position = ModelviewProjectionMatrix * vPosition;
   vNormalOut  = NormalMatrix * normalize( vNormal );
   // ...
}
```

**Listing 2.1:** Pseudo code for the main pass of our algorithm. The evaluation can be done either in a vertex shader using instancing or using tessellation evaluation shaders of current hardware.

```
struct BFTPatch {
   uint m_uiPoints[ n ];
   uint m_uiIndexBFT;
   uint m_uiOneRingSize;
};
```

The values stored in `m_uiPoints` index control points of the patch, i. e. vertices of the patch and its one-ring neighbourhood (all regarding $\mathbf{M}^1$). `m_uiIndexBFT` is an index into the basis function table, indicating the set of basis functions to use for the particular patch connectivity. This value is induced by the related face $f_i^1$ on $\mathbf{M}^1$, its location (interior, bordering, etc.) and the valence of the extraordinary vertex (if any). `m_uiOneRingSize` is used later as a break condition while looping over the one-ring neighbourhood.

For each face/patch $f_i^1 \in F^1$ tessellation is performed at desired level, outputting a dense set of parameter values. These values are 2D coordinates of points on $f_i^1$ and the tessellator ensures that the corresponding fine triangulation is generated. We now need to evaluate the 3D position of a fine vertex from this 2D parameter

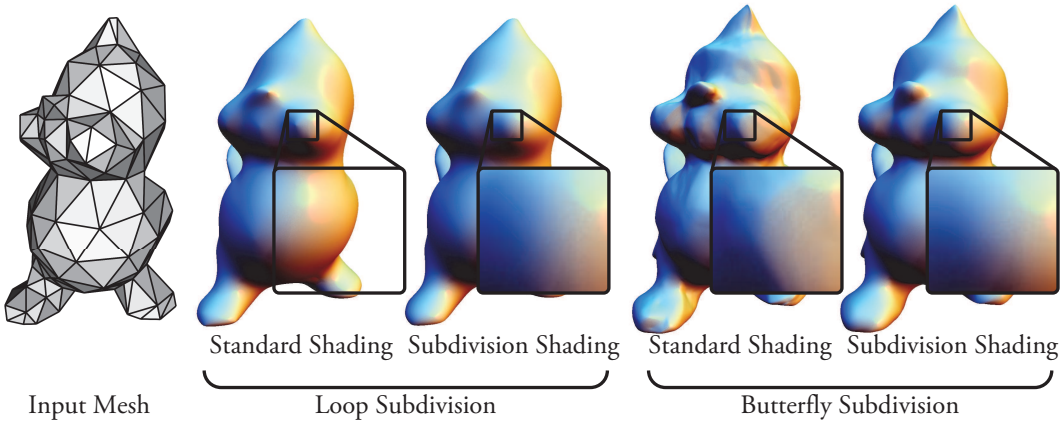



**Figure 2.9:** Direct comparison of Subdivision Shading and standard shading. Subdivision Shading creates a visually smooth normal field that is especially effective for attenuating artifacts around extraordinary vertices (closeups). Images taken from [Ale+08].

$\{u, v\}$ and the values computed at the initial step (Fig. 2.7). We first map $\{u, v\}$ onto an integer subindex: as maximum BFTs correspond to uniform tessellation at maximum level, this subindex is trivially computed as the 1D parameter of $\{u, v\}$ on the tessellation's space filling curve. The evaluation of a surface point is a linear combination of the vertices $v_i^1$ of $\mathbf{M}^1$ and the basis functions as weights.

$$f(u, v) = \sum_i B^i(u, v) v_i^1 \tag{2.1}$$

The basis functions for each surface point are given by the connectivity index (`m_uiIndexBFT`) of the patch and the subindex. The evaluation of equation 2.1 in a shader is very fast as it boils down to a simple loop in the shader code (Lst. 2.1).

The surface normals are computed similarly to the vertex positions by applying Subdivision Shading [Ale+08]. Basically the same weights as used for vertex positions can be applied to vertex normals followed by normalization, which is a fast and visually pleasing approximation to the Subdivision Shading normal and avoids an iterative solution. Note that the actual geometric normals usually lead to lower quality in shading (Fig. 2.9).

**Adaptive Refinement** As mentioned in [Bol+02] the BFTs can be subsampled thus allowing for adaptive refinement as well. Since we consider limit surface points at any level, all basis functions of level $n$ can be seen as a subset of basis functions of level $n + 1$. Tessellation units usually offer adaptive levels of tessellation that are specified using different tessellation ratios for the edges of an input face. Our approach is trivially compatible to such adaptive approaches as our algorithm does not rely on the particular connectivity of $\mathbf{M}^k$: a triangle with different tessellation ratios (i. e. fine adaptive triangulation) can make use of the same BFT (Fig. 2.10). Although it is not possible to sample the subdivision surface at arbitrary parameter values with our method (e. g. for the Loop scheme, all fine vertices must be located

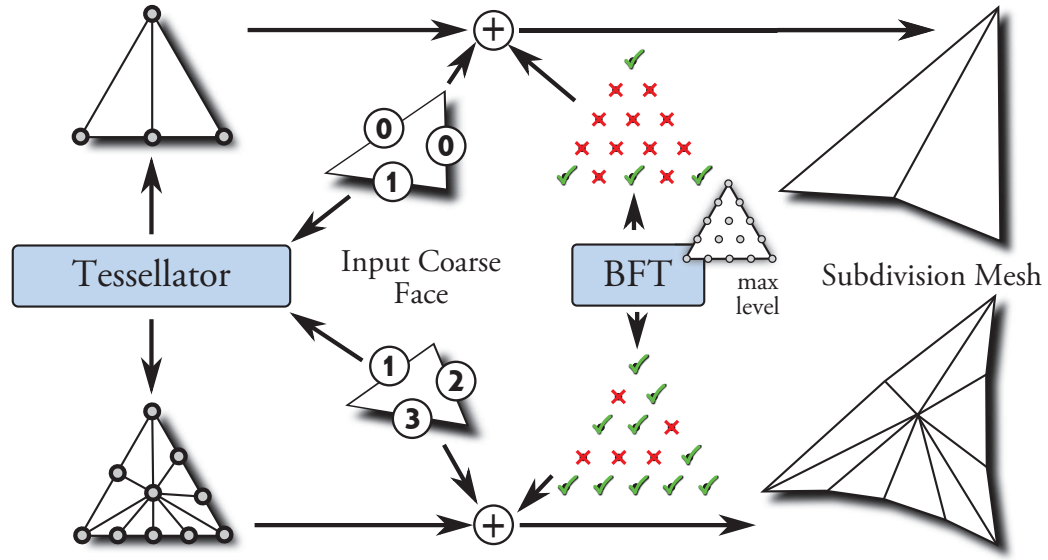



**Figure 2.10:** Adaptive real-time tessellation and BFT subsampling. Different tessellation patterns produced by the tessellator (left) can use the same set of maximum level BFTs.

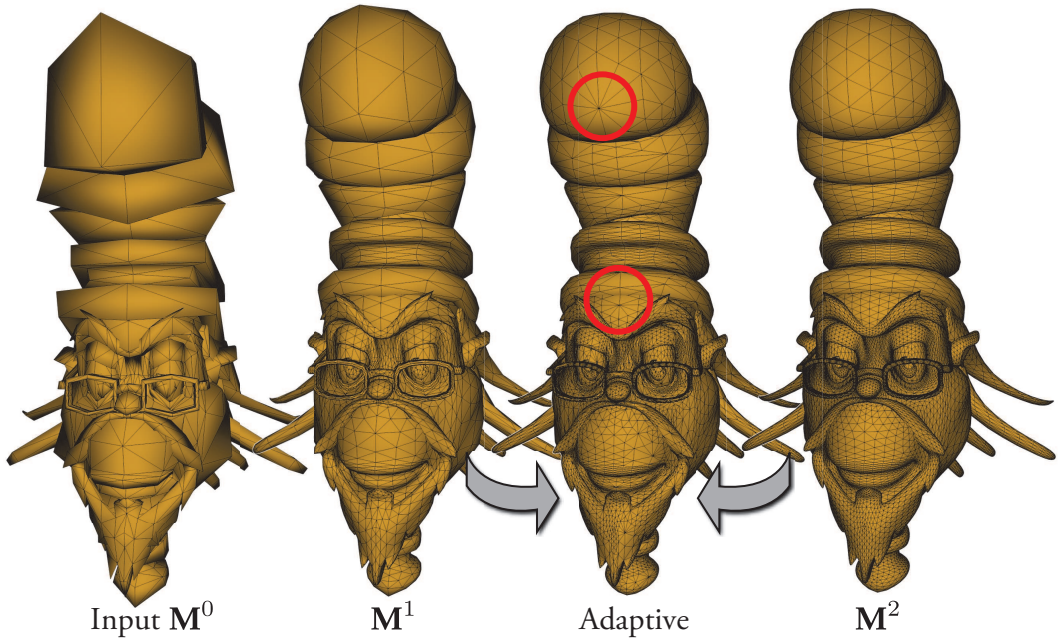



**Figure 2.11:** Our approach with adaptive tessellation, subsampling the BFTs. For demonstration purposes each edge has been assigned a random tessellation factor.

at dyadic split positions), we can still offer linear geomorph transitions, i. e. with all vertices laying on a virtual mesh subdivided at maximum level (Fig. 2.11).

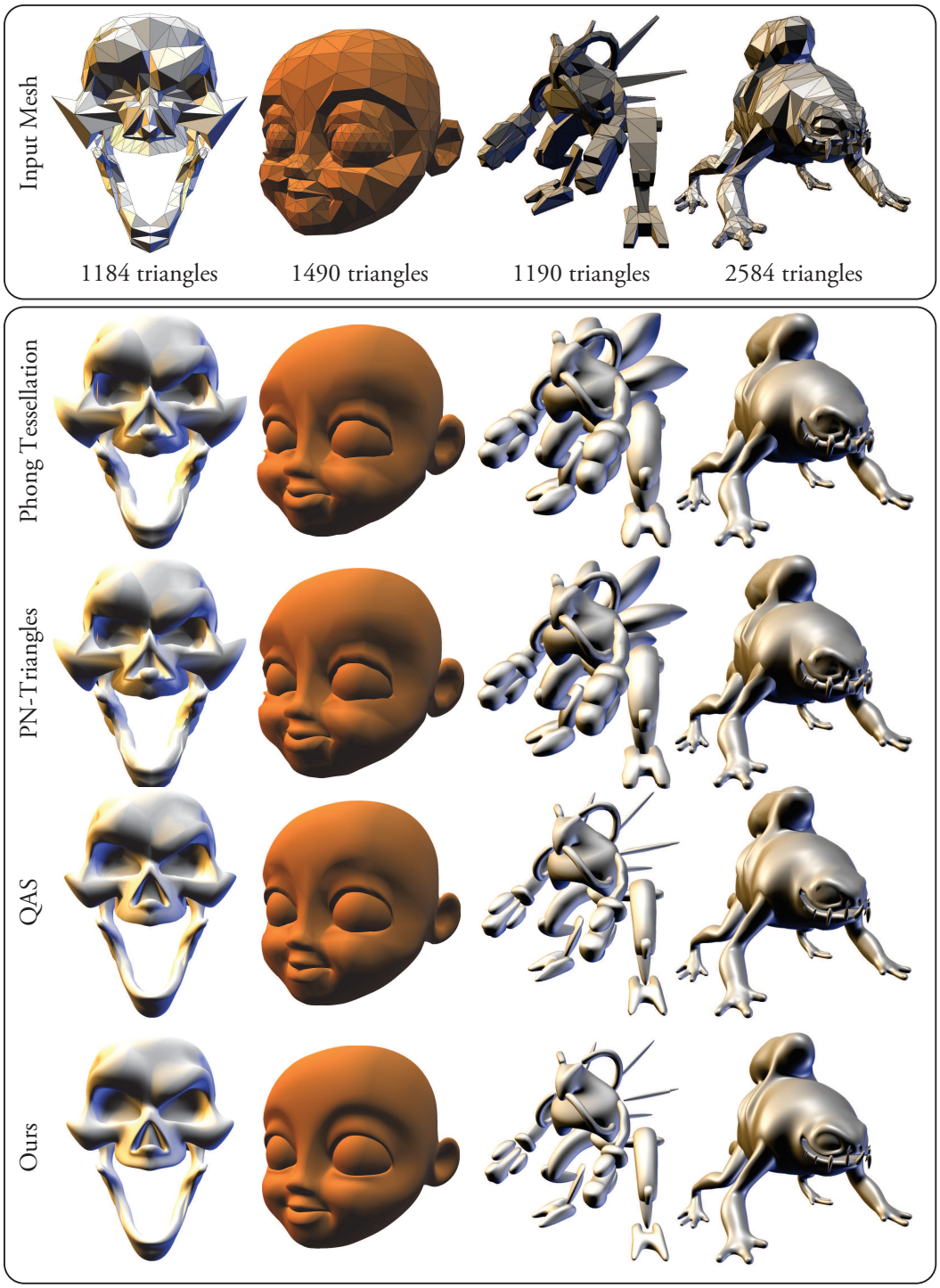



**Figure 2.12:** GPU Refinement: subdivision substitutes versus our GPU subdivision meshes ($\mathbf{M}^5$).

| Model | Head | Guy | MonsterFrog | BigGuy |
|---|---|---|---|---|
| #Input triangles | 524 | 1168 | 2584 | 2900 |
| *Tessellation Level 2 (4×4 split)* | | | | |
| #Output triangles | 8384 | 18688 | 41344 | 46400 |
| *Frame rates (in fps)* | | | | |
| Our Method | 641 | 495 | 334 | 291 |
| *Tessellation Level 4 (16×16 split)* | | | | |
| #Output triangles | 134 k | 299 k | 661 k | 742 k |
| *Frame rates (in fps)* | | | | |
| Our Method | 312 | 176 | 85 | 76 |
| *Tessellation Level 5 (32×32 split)* | | | | |
| #Output triangles | 536 k | 1196 k | 2646 k | 2969 k |
| *Frame rates (in fps)* | | | | |
| Our Method | 103 | 50 | 24 | 21 |
| QAS [Bou+07a] | 80 | 53 | 44 | 23 |
| PN-Tri. [Vla+01] | 109 | 69 | 41 | 35 |
| PT [Bou+08b] | 120 | 77 | 48 | 40 |
| Flat | 137 | 85 | 52 | 45 |

**Table 2.1:** Performance measure of our algorithm for meshes of various size.

## 2.3 Results and Performances

The performance of our algorithm was tested on an NVIDIA GeForce GTX 295 with 1.8 GB graphics memory and an Intel Core i7 2.67 GHz using OpenGL under Windows. Please note, that this is a graphics card from the GeForce-200-series, the predecessor of the Fermi series. We measured frame rates of our algorithm in combination with the Adaptive GPU Refinement Kernel [Bou+08a] as a tessellator emulator. Although this implementation does not reflect the performances obtained with recent genuine hardware tessellation units, it helps to understand how the overall workload for exact subdivision mesh generation compares to fast approximations and subdivision substitutes.

Table 2.1 gives the frame rates for several models at various subdivision levels and compares it to a variety of substitutes. Additional results for our method and the same setup but with a Fermi card can be found in the Annex in section 7.4.

Surprisingly, while the presented subdivision pipeline is, of course, more expensive than local refinement schemes such as Phong Tessellation [Bou+08b] or PN Triangles [Vla+01], it still succeeds at offering real-time performances and significantly higher surface quality (Fig. 2.12). For instance, it is only slightly more expensive


**Figure 2.13:** Per-triangle rendering cost as a function of the tessellation level at which it is synthesized.

than subdivision surface approximation schemes such as the QAS model [Bou+07a] which also relies on an initial subdivision step (that we implemented on the GPU in our framework). We also measure flat tessellation (no fine vertex displacement) to better quantify the cost of the subdivision evaluation. Overall, the performance of our simple approach shows that subdivision surfaces can be created on-the-fly on the GPU with real-time performance and without resorting to substitutes. Of course, our approach relies on input meshes suitable for subdivision and the evaluation at arbitrary parameters is bounded to linear interpolation on the fine triangles.

Subdivision substitutes might still be interesting for cases where very high frame rates are required and quality can be traded for speed. However, for common applications where standard subdivision surfaces with recursive evaluation bounded by a limited number of subdivision steps are used (e. g. high end computer graphics packages for SFX and animation), our experiments show that the computational workload of the CPU can be decreased significantly by offloading the entire process to our two-step GPU algorithm, in particular regarding the per-fine triangle cost for deep subdivision levels (Fig. 2.13). The memory footprint of the BFTs for our experiments was relatively small with about 9 MB for tables of level 5 for interior and boundary cases up to a valence of 18.

The final result is shaded using Subdivision Shading normals and is, regardless of the valence, visually smoother (Fig. 2.14) than linear [Pho73] or quadratic [VO+97; Vla+01; Bou+07a] normal interpolation.

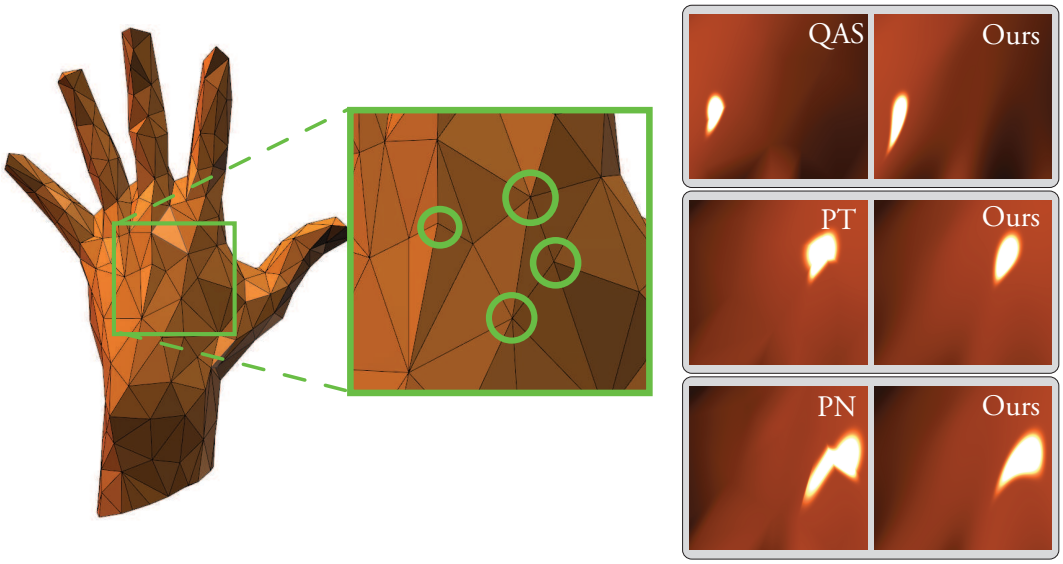



**Figure 2.14:** Surface quality around extraordinary vertices: comparison between quadratic approximation (QAS), Phong Tessellation (PT), PN-Triangles (PN) and our subdivision method. For each scheme the light is placed to produce a specular highlight near the extraordinary vertices and compared to our method.

## 2.4 Conclusion

We have shown that a simple combination of BFTs and GPU tessellation is able to produce dynamic subdivision meshes at high frame rates with dense output. By subsampling the BFTs, adaptive GPU subdivision is made possible and provides flexible LoD control. In addition, our approach can be implemented on any programmable GPU using existing tessellation kernels or current hardware tessellation units. Moreover, it is compatible with many subdivision schemes, does not require any particular scheme-specific setup and can provide smoother shading using Subdivision Shading.

Most recent work [Nie+12] uses a similar approach to ours, namely an initial GPU subdivision step and subsequent usage of BFTs. However, they only perform the initial subdivision step for patches that contain a vertex of irregular valence or patches containing at least one edge that is marked as *creased*. This process is also prepared using CPU precomputation and their generation of BFTs is bounded by the number of extraordinary configurations (irregular valence or creases). This is due to the fact, that they evaluate all regular patches directly using bicubic Bézier patches as they demonstrate their approach for Catmull-Clark subdivision surfaces. However, in irregular cases they utilize temporary GPU storage combined with compute shaders and recursive and adaptive evaluation.

We believe that our method can be useful to developers using subdivision surfaces who want to exploit the latest GPU generations with tessellation capabilities. Also,

our approach relies on precomputed tables only and is therefore very general, making it attractive for a broad range of subdivision algorithms [Zor99]. Overall, subdivision surfaces can be created efficiently in real time without switching to other surface models such as subdivision substitutes.

Our method allows to shift the complete upsampling process to the GPU, thus, it not only frees the CPU to perform other tasks but introduces an additional abstraction layer, where the application is responsible for manipulating geometric descriptors of the target surface only, i. e. a low resolution mesh.

In the following chapter we will use a different, more simple representation of geometry, namely point primitives without connectivity information, but maintain the idea of flexible LoD control. Also, we will not focus on a *single* view for which a single high quality image needs to be rendered but rather on a significantly larger amount of views. Each view only requires a low-resolution rendering and participates in the complex computation of global illumination effects.

CHAPTER 3

# REAL-TIME GLOBAL ILLUMINATION WITH POINTS

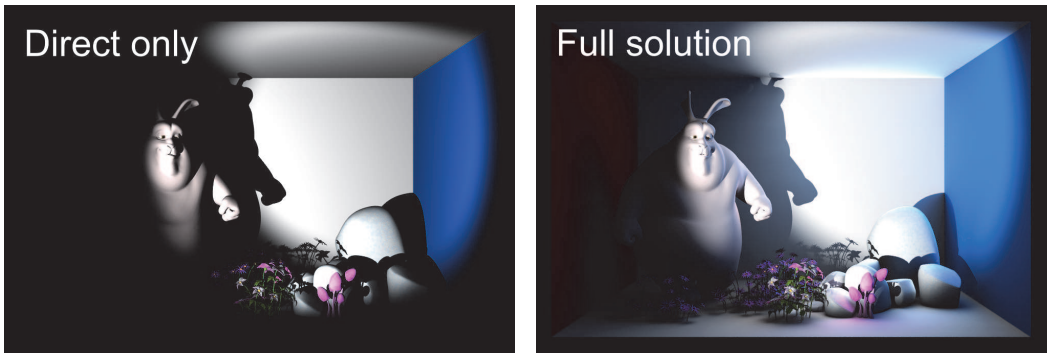



**Figure 3.1:** Left: Scene rendered with direct lighting only. Right: Scene rendered using global illumination (offline) solution. Images taken from [Buc+12].

In the previous chapter we synthesized high-resolution dynamic geometry for the purpose of creating smooth surfaces. We will now focus on the efficient computation of complex light interactions with the scene, that is commonly referred to as Global Illumination (GI) (Fig. 3.1). Simulating GI effects in real time is still a challenging task with the goal to fit one or several GI algorithms into the already small time-per-frame budget of a real-time application (Sec. 1). Here, again, we face a geometric problem when sampled geometry of the scene needs to be synthesized to solve for visibility. However, GI effects play such a tremendous role in generating images that are perceived as *real* that they are essential for any application striving for photo-realistic rendering.

In this chapter important terms and notions regarding light and materials (Sec. 3.1) are briefly reviewed which are essential for understanding complex lighting effects in the scene (Sec. 3.2). Following that is the mathematical background for describing GI by means of the rendering equation (Sec. 3.3). Well-known rendering solutions are summarized subsequently in section 3.4.

We will then present our ManyLoD algorithm in section 3.5 which enables many existing interactive algorithms to run in real time by using a small-scale geometric GPU kernel. More specifically, this algorithm can be combined with a variety of GI algorithms and accelerates visibility based computations for many-view scenarios that rely on a hierarchical Level-of-Detail representation of the scene. These LoD structures are a key component when it comes to scalable rendering. They are often built from raw 3D data and defined as Bounding Volume Hierarchies. Such hierarchies provide coarse-to-fine adaptive approximations and are well-suited for *many-view* rasterization scenarios where the total number of pixels in each view is usually low, while the cost of choosing the appropriate LoD per view is high. This task represents a challenge for existing GPU algorithms and we propose, with ManyLoDs, a new GPU algorithm to efficiently compute many LoDs from a Bounding Volume Hierarchy in parallel by balancing the workload within and among LoDs. Our approach is not specific to a particular rendering technique, can be used on lazy representations such as polygon soups, and can handle dynamic scenes.

We apply our method to various many-view rasterization applications, including Instant Radiosity, Point-Based Global Illumination, and reflection/refraction mapping in section 3.6. For each of these, we achieve real-time performance in complex scenes at high resolutions.

In section 3.7 we will present further extensions to our algorithm that reduce memory requirements and investigate a hybrid (i. e. GPU/CPU) implementation before we conclude in section 3.8.

## 3.1 Lights and Materials

Light is electromagnetic radiation, i. e. energy, and several theoretical models exist to describe light based on quantum mechanics or electromagnetic waves. In computer graphics we are mainly interested in the energy transferred by light and often change between a photon or wavelength-based representation where appropriate.

The *radiant flux* $\Phi$ is defined as the radiant energy $Q$ per unit time and is measured in Watts:

$$\Phi = \frac{dQ}{dt}$$

The *radiance* $L$ is the most important quantity when computing light transport. Simply put, it is the radiant flux per unit projected source area per unit solid angle that a ray of light arriving, leaving or passing through a point $\mathbf{p}$ in direction $\omega$ is carrying. More formally:

$$L(\mathbf{p}, \omega) = \frac{\mathrm{d}^2\Phi(\mathbf{p}, \omega)}{\mathrm{d}A\ \mathrm{d}\omega\ \cos(\mathbf{n}_\mathbf{p}, \omega)}$$

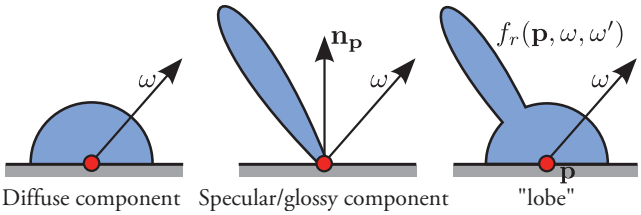



**Figure 3.2:** Left: diffuse reflection. The light coming from direction $\omega$ is reflected equally in all directions. Middle: specular/glossy reflection. Right: example BRDF combining specular and diffuse reflection.

with $\mathrm{d}\omega$ being the differential solid angle and $\mathbf{n_p}$ the surface normal at $\mathbf{p}$. The cosine term ensures that the definition of $L$ does not rely on the orientation of $\mathrm{d}A$ in relation to direction $\omega$.

The view-independent *irradiance* $E$ at $\mathbf{p}$ is the integral over the positive hemisphere centered around $\mathbf{n_p}$:

$$E(\mathbf{p}, \mathbf{n_p}) = \int_{\Omega_+} L_i(\mathbf{p}, \omega') \cos(\mathbf{n_p}, \omega') \, \mathrm{d}\omega' \tag{3.1}$$

As a more visual explanation one could say that irradiance is the sum of all light *seen* by $\mathbf{p}$ when looking in direction $\mathbf{n_p}$ using a fisheye-hemispherical view.

When light hits a surface, it often gets reflected into various directions. If the light is reflected equally in all directions of the hemisphere we speak of *diffuse reflection* (Fig. 3.2 left image). A perfect mirror would reflect light coming from direction $\omega$ according to the law of reflection: $\omega' = \omega - 2(\cos(\mathbf{n_p}, \omega))\mathbf{n_p}$. All in-between types of reflection are usually classified as *glossy* or *specular* (Fig. 3.2 middle). However, real surfaces are rarely perfectly diffuse or perfectly specular and the light is distributed in a combination of reflections. This distribution is known as the Bidirectional Reflectance Distribution Function (BRDF) and the function's response is often called a *lobe* (Fig. 3.2 right image). Boris et al. [Bor+08] visualized the lobes for some real-life objects with different materials using a fluorescent fluid (Fig. 3.3). More sophisticated reflectance distribution function models exist such as the Bidirectional Scattering Surface Reflectance Distribution Function (BSSRDF) but they are out of the scope of this thesis.

## 3.2 Global Illumination

Light, emitted from a light source, is often bouncing multiple times within the scene before reaching the human eye where it is perceived as colour. Emitted light, hitting at most one surface and bouncing directly into the eye is called *direct light*. However, in many cases light takes more complex paths that involve several bounces within the scene that contribute to the *indirect light*. Global illumination simply means the addition of direct and indirect illumination. While direct

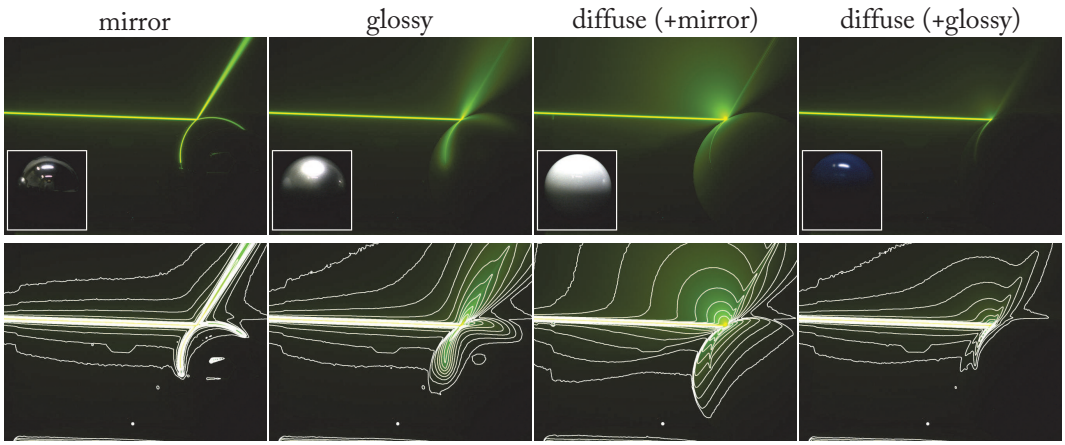



**Figure 3.3:** Visualization of light transport using a fluorescent fluid. Top row: Interaction of a laser with objects of different materials. Bottom row: Superimposed intensity iso-lines to visualize the *lobes*. Images taken from [Bor+08].

illumination effects are usually easy to compute, as most necessary computations can be done locally, indirect illumination effects involve more effort as they depend on global information. However, they play such an important role in creating images that are perceived as *real* that is it impossible to neglect them. Some well-known phenomena caused by light (Fig. 3.4) are briefly described in the following.

**Colour Bleeding** Colour bleeding is caused by diffuse interreflections, i. e. light hits a diffuse surface, where it gets reflected to another surface before eventually reaching the eye. The colour from the first surface *bleeds* onto the second surface.

**Indirect Shadows** Indirect shadows can occur when the indirect illumination is blocked by an object. They are usually smooth and hard to sample noise-free. The higher the amount of indirect light that is blocked, the more pronounced these shadows appear.

**Scattering** The paths of light are also affected by the medium it is transported in. This involves scattering or diffusion when light hits particles of *participating media* such as fog or smoke. Further, we speak of *subsurface scattering*, when light hits a surface, enters the object, and bounces (possibly) multiple times within the object before it leaves the object again at a potentially different point than it entered. This phenomenon is visible on translucent objects and materials such as wax, skin or even marble. These scattering effects cannot be expressed by BRDFs that only describe on-surface reflections. Instead, more general Bidirectional Scattering Reflectance Distribution Functions (BSSRDFs) are used [Jen+01].

**Caustics** Caustics are caused by light hitting at least one specular surface before it arrives at a diffuse surface where it gets reflected to the eye. These effects are commonly known from glass or water causing caustics on the bottom of a pool.

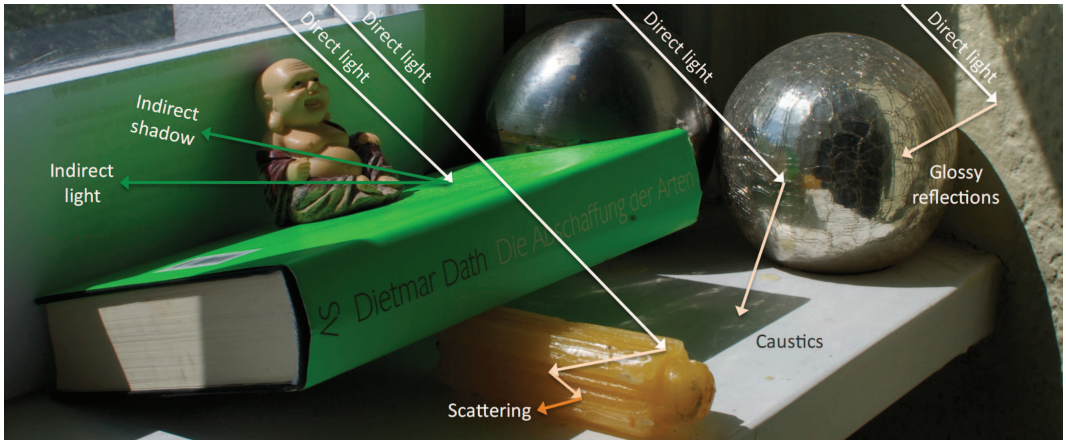



**Figure 3.4:** Photograph of global illumination effects demonstrating multiple bounces of light within a scene. Image taken from [Rit+12].

In computer graphics, we use the *rendering equation* to describe and compute the light transport in the scene that leads to the previously mentioned effects.

## 3.3 Rendering Equation

The rendering equation introduced by Kajiya [Kaj86] is a general formulation of light transport in the scene and the fundamental equation when simulating GI. It describes the outgoing radiance $L_o$ of a surface point $\mathbf{p}$ in direction $\omega$:

$$L_o(\mathbf{p}, \omega) = L_e(\mathbf{p}, \omega) + \int_{\Omega_+} f_r(\mathbf{p}, \omega, \omega') L_i(\mathbf{p}, \omega') \cos(\mathbf{n_p}, \omega') \, \mathrm{d}\omega' \tag{3.2}$$

1. $L_e$ is the emitted radiance at $\mathbf{p}$ in direction $\omega$, which is only relevant for light sources.
2. $\Omega_+$ is the upper hemisphere at $\mathbf{p}$ centered around the surface normal $\mathbf{n_p}$.
3. $f_r$ is the BRDF that returns the reflected amount of energy from direction $\omega'$ in direction $\omega$ at $\mathbf{p}$, characterizing the optical material properties (Sec. 3.1).
4. $L_i$ is the incoming radiance from direction $\omega'$ at $\mathbf{p}$.
5. $\cos(\mathbf{n_p}, \omega')$ is a pure geometrical term. Imagine shining a flashlight at different angles onto a surface. The amount of photons hitting the surface is always the same, however the number of photons per area is decreasing with increasing angle and the brightness is proportional to the cosine.

The incident radiance at $\mathbf{p}$ comprises light coming directly from a light source or from indirect light (Fig. 3.5), thus, the rendering equation depends upon itself.

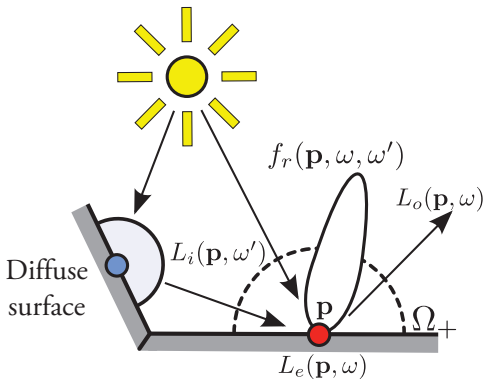



**Figure 3.5:** The rendering equation describes the outgoing radiance $L_o$ at $\mathbf{p}$ in direction $\omega$ as the sum of the emitted light $L_e$ at $\mathbf{p}$ in direction $\omega$ and the integral over the hemisphere.

Also, the integral over the hemisphere cannot be solved analytically, except for some special cases, which makes it difficult to use in practice. In the following, several global illumination techniques are described briefly together with their advantages and disadvantages.

## 3.4 Rendering Solutions for Global Illumination Effects

In this section we summarize well-known rendering solutions for computing global illumination effects. Please mind that an in-depth description is out of the scope of this thesis and we refer the interested reader to Veach's PhD thesis [Vea97] as well as to a recent state of the art report regarding interactive GI [Rit+12].

**Radiosity** Radiosity, also called finite elements, was introduced by Goral et al. [Gor+84] and can be used to compute global illumination under the assumption that all surfaces are diffuse. It is based on the observation that indirect illumination often varies only slowly across planar surfaces and works as follows:

1. The input geometry is discretized into patches. Each patch should be small enough to assume that the reflected illumination is constant.
2. Computation of *form factors* for all pairs of patches. The form factor describes the energy transfer between two patches based on their orientation and visibility to each other.
3. Solving a system of linear equations to compute the radiosity for each patch.
4. Displaying the result.

This process requires a large amount of memory for the storage of the form factors and has a high complexity of $\mathcal{O}(N^2)$ that can be reduced to $\mathcal{O}(N \log N)$ using a hierarchical approach [Han+91]. An advantage is that the solution is independent

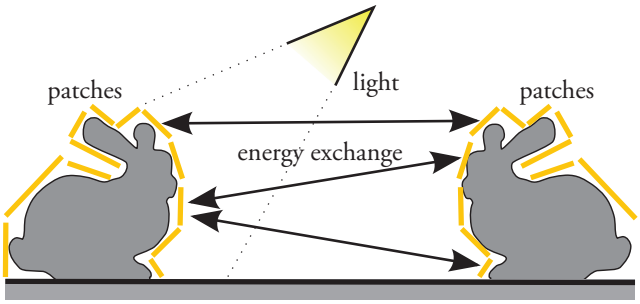



**Figure 3.6:** Radiosity.

of the viewpoint and once the matrix of equations is available even the light settings can be changed. Radiosity can be combined with a final gathering step [Lis+93] for improved accuracy. A thorough introduction can be found in [Hec93].

GPUs can be used to accelerate various steps of the algorithm. Meyer et al. [Mey+09] presented a parallel version for creating the required links for a hierarchical approach that runs entirely on the GPU. The form factor calculation can be sped up using GPUs [Nie+01] by using texture maps instead of expensive geometric patch subdivision. Further, Carr et al. [Car+03] demonstrated that the Jacobi iterations to solve the matrix of equations can be accelerated using a GPU kernel.

**Antiradiance** Antiradiance [Dac+07] can be used to simulate GI without explicit visibility computation. Here, the outgoing radiance of a sender point is distributed to all receiving points in the scene and the erroneously propagated light is compensated by distributing *antiradiance*. The energy distributed by a point $\mathbf{p}$ as antiradiance is equal to the incident radiance at $\mathbf{p}$ but distributed to the lower hemisphere. Therefore, the light transport in the scene can be computed iteratively by alternating the distribution of radiance and antiradiance where the number of iterations is equal to the (layered) depth complexity of the scene. The authors present a GPU implementation derived from a radiosity implementation, i. e. patches are used for exchanging energy, and the process can benefit from the same hierarchical approaches as the finite element method described previously.

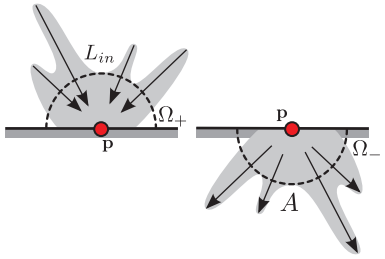



**Figure 3.7:** Antiradiance

**Path Tracing** Path tracing was introduced by Kajiya in 1986 [Kaj86] and is an extension to traditional (backwards) ray tracing. In traditional ray tracing, rays are sent from the eye through each pixel and at the intersection with the scene additional rays are shot in the direction of the light sources to compute shadows. All other additional rays are treated as either perfectly reflecting or perfectly refracting. Path tracing extends this principle and sends out additional rays at each

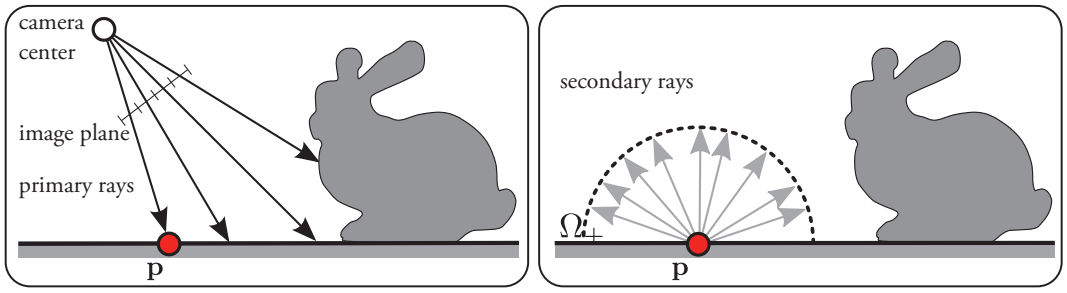



**Figure 3.8:** Path Tracing.

intersection point based on russian roulette to solve the rendering equation using random sampling. These additional rays are used for diffuse or specular reflections, or refractions. The results converge to the true solution but require a high amount of samples and thus computation time due to the slow convergence rate of $\frac{1}{\sqrt{n}}$, where $n$ is the number of samples. With an insufficient amount of samples the results exhibit high variance, i. e. noise, especially in areas that are not directly lit. We refer the reader to [Car+01; Pha+10] for a broader overview and to [Wal+09] for ray tracing of animated scenes.

A large body of work is currently focused on utilizing the GPU for ray or path tracing [Car+02; Par+10]. Here, the biggest challenge is to treat rays efficiently and coherently without causing threads to diverge [Bou+07b; Man+07; Ail+09; Ail+10; Ail+12] which is a difficult task especially for secondary rays.

**Bidirectional Path Tracing** Bidirectional path tracing [Vea97] is an extension to path tracing that achieves faster convergence and allows more efficient handling of difficult light paths. The basic idea is to shoot rays into the scene not only from the camera but also from the light sources. The scene intersection points from both types of rays transfer energy between each other if no blocker object is positioned between them.

**Metropolis Light Transport** Often times when doing path tracing, light comes from *hard-to-reach* locations of the scene, e. g. a door that is only slightly open. Metropolis light transport [Vea+97], as a variation of path tracing, mutates those paths to find similar ones that contribute largely to the image. This strategy can reuse existing sub-paths and accept or reject mutated paths leading to more acceptable solutions in less computational time.

**Instant Radiosity** Instant radiosity developed by Keller [Kel97] uses an approach similar to bidirectional path tracing. First, photons are shot from the light sources and at the intersection point with the scene, secondary light sources are placed. These secondary light sources are called *Virtual Point Lights (VPLs)* and carry the radiant flux of the scene intersection point. For the final rendering of the scene, each point is influenced by non-occluded primary and secondary light sources.

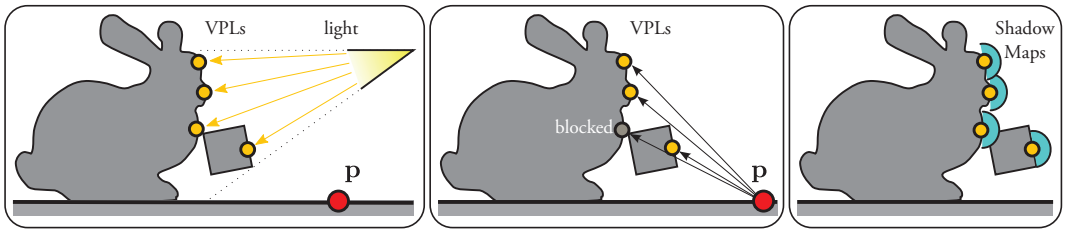



**Figure 3.9:** Instant Radiosity. Left: VPL creation. Middle: Visibility by raycasting. Right: Visibility using shadow mapping (e. g. Imperfect Shadow Maps).

However, the accurate computation of visibility for light sources (primary and VPLs) still requires raycasting or shadow mapping.

In a real-time context, VPLs are often created using Reflective Shadow Maps (RSM), that is, the position of additional light sources is determined by sampling the pixels of an enriched shadow map [Dac+05]. This enriched shadow map is created using multiple render targets storing normal, flux and depth information. Nevertheless, each VPL requires visibility information for a correct evaluation. The cost of computing an additional shadow map per VPL can be alleviated by ignoring [Dac+05] or approximating [Rit+08] visibility. In the latter case, a point-sampled representation of the scene helps to achieve high performance during the generation of Imperfect Shadow Maps (ISM) and can be improved by choosing the required set of VPLs carefully according to the current view [Rit+11]. As a further improvement, the number of required VPLs can be reduced by hierarchical organization [Wal+05] or clustering [Rit+11; Pru+12]. For temporal coherence, Laine et al. [Lai+07] use an incremental approach that adds and removes VPLs dynamically.

**Irradiance Caching** Instead of computing the irradiance (Eq. 3.1) per pixel or subpixel, it is usually sufficient to compute it for a subset of points in the scene and interpolate the results. This is the basic idea of irradiance caching [War+88], again, based on the observation that the indirect diffuse light varies only slowly. A receiving point for which the indirect light is computed is called a *cache* which holds a single irradiance value $E$ without directional information. These cache entries can be stored in an octree to speed up the search for existing cache entries during interpolation. Instead of using a fixed set of sample points, a new cache entry is created *on demand*, i. e. when no suitable interpolants can be found. A good metric to determine if a new entry is required and a weighting function to interpolate cache entries is provided by Tabellion and Lamorlette [Tab+04]. Their metric avoids heavy oversampling and is suitable for highly detailed scenes.

Due to the sparse sampling of irradiance values in the scene, irradiance caching speeds up the computation of indirect illumination for the final image but is limited to perfectly diffuse reflections and should be combined with other approaches to compute the specular/glossy components of the BRDF (e. g. Monte Carlo

importance sampling). However, the on-demand strategy is sequential and prevents an easy mapping of the whole process to the GPU.

**Radiance Caching** Radiance Caching [Kri+05] also computes the incident indirect light only for a subset of points similar to irradiance caching. Again, the cache entries are stored in an octree for easy access. This time, however, the cache entries are more powerful as they also encode directional information using (hemi-) spherical harmonics [Gau+04] as well as a local frame and two translational gradients. This enables the interpolation for general low frequency BRDFs and ensures smooth interpolation of cache entries. For high frequencies the authors suggest to revert to Monte Carlo integration.

A recent approach [Sch+12] allows the interactive interpolation of radiance caches by exploiting a mip-mapping-based algorithm. For the specular part, instead of performing the expensive reflectance evaluation per-pixel, the workload is shifted to a per-cache level. For each cache entry a series of mip-map levels is computed, each representing a different level of glossiness, that allow a constant time lookup during the final shading of a pixel.

**Photon Mapping** Photon mapping [Jen96] can handle specular and glossy reflections and works in two steps. First, photons are emitted from the light sources and the subsequent paths are constructed. At each vertex along the path, the incident illumination is stored and saved in a global photon and caustics map. Second, the final image is rendered using ray casting for primary rays and indirect illumination is computed using the values from the maps of the previous step by raycasting or density estimation. As an advantage, the photon map is independent of the main camera and, thus, allows to alter the camera position.

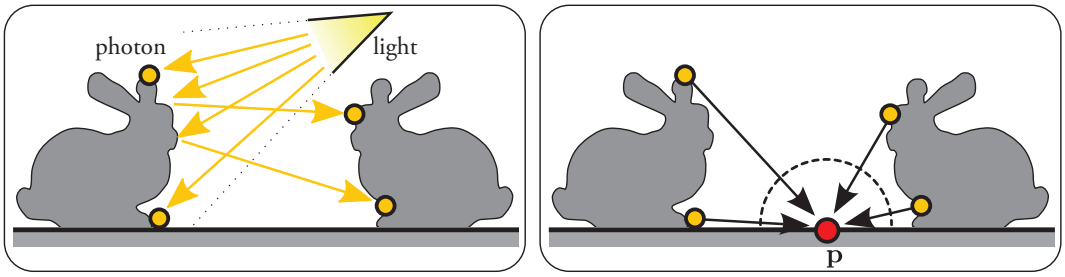



**Figure 3.10:** Photon Mapping. Left: Photons are emitted into the scene. Right: Illumination from photons is combined.

The first partly GPU based photon mapping implementation was presented by Ma and McCool [Ma+02], and an entirely graphics hardware based version was later presented by Purcell et al. [Pur+03]. The latter is based on a uniform grid for the photon map to simplify the implementation. The final radiance estimation, though accelerated by a k-nearest neighbour search, still takes most of the computation time. Yao et al. [Yao+10] perform photon tracing in image space instead of object space and use multiple environment maps to store the result. Hachisuka and

Jensen [Hac+10] use a stochastic scheme to store photons in a spatial map instead of keeping a list of photons, thus, improving GPU parallelism.

**Point-Based Global Illumination** The idea of Point-Based Global Illumination (PBGI) [Chr08] is to use a densely sampled point representation of the scene to compute a noise-free approximation of ambient occlusion and colour bleeding. The point samples represent surface elements – *surfels* – that store spherical functions (e. g. spherical harmonics) to approximate the reflected light and the complete set of samples is organized hierarchically in a tree. The tree is traversed for each *receiving point* for which the indirect light needs to be computed and a rasterization-dominant approach is used to render the required nodes of the tree into the low resolution framebuffer of a receiving point.

PBGI can be done on the GPU as well as demonstrated in [Rit+09]. A per-pixel final gathering step is, however, limited to low resolution renderings. Faster results can be obtained by performing final gathering only on a subset of pixels and upsample the result using techniques based on bilateral filtering operators [Slo+07] or a guided image filter with a precomputation step [Bau+11].

**Grid-Based Techniques**

Grid-based methods have gained great interest in the context of real-time GI effects. This is due to the fact that computations regarding a grid are often much easier to map to graphics architectures. An irradiance distribution function can be computed at the vertices of a grid [Gre96; Gre+98] and the irradiance at a point $\mathbf{p}$ within a grid's cell can be approximated by evaluating the irradiance distribution function of the bounding grid vertices and trilinear interpolation.

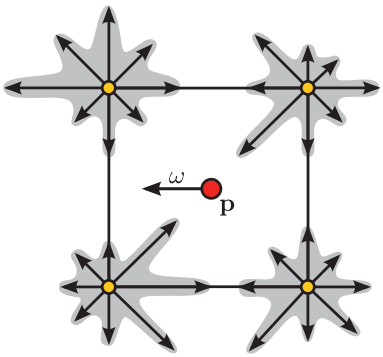



**Figure 3.11:** Irradiance Volume in 2D. Image based on [Gre96].

Kaplanyan and Dachsbacher [Kap+10] convert VPLs, created from RSMs, to a spherical harmonics basis and inject them into a cascaded grid that encompasses the entire scene. The illumination is then diffused across cell boundaries through empty space by iteration and subsequently used to approximate the indirect illumination. Thiedemann et al. [Thi+11] create a voxel grid of the scene in two steps. First, they rasterize each object into a texture atlas and capture world positions. Then, for each valid texel of the atlas a vertex is generated to fill the voxel grid texture. Such a voxel grid can be used for further ray or path tracing. Alternatively, RSMs can be used to create additional light sources and a per pixel gathering approach allows to compute the indirect light (by backprojecting hitpoints into the RSMs to retrieve the radiance). Papaioannou [Pap11] computes *radiance hints* in a spatial grid. A radiance hint represents a volume cell for which a radiance field is approximated by stochastically sampling the RSMs from

different positions within each cell. These samples approximate the radiance field within the cell and are encoded using a spherical harmonics basis. Again, trilinear interpolation can be used to acquire the approximate solution of the indirect lighting term. Another promising approach was presented by Crassin et al. [Cra+11], who inject VPLs from RSMs into the leaf nodes of a sparse voxel hierarchy. The incoming radiance in those cells is then propagated to nodes higher in the hierarchy similar to mip-mapping and stored as a Gaussian lobe representation. The indirect illumination is gathered from the octree on a per pixel basis by tracing a set of voxel cones.

Other real-time techniques based on precomputation [Slo+02] exist but are limited to static scenes. Coarser approximations to GI such as ambient occlusion are fast and efficient (Sec. 7.2) but only capture a fraction of the lighting effects that can be achieved by a full solution. In the following, we will present our method that accelerates approaches that can make use of views to approximate visibility (ISM) or for cache computation as in PBGI and in general in (ir)radiance caching.

## 3.5 ManyLoDs: Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination

LoD algorithms [Lue+02] are a necessity for efficient rendering techniques that seek to depict today's complex and ever-growing virtual worlds. Besides direct rendering of geometry into the view of a virtual observer, there exists a range of techniques that require rendering the scene from many additional views, e. g. into classic shadow or reflection maps. While LoDs are well-understood for direct rendering and often computed incrementally [Xia+96; Hop96], the novelty of our approach lies in its specific target of many-view rasterization, which means that not one but *many* LoDs have to be extracted concurrently. Our ManyLoD approach renders a high number of views using a fast LoD extraction algorithm that is designed to fit modern GPUs. Typical examples are Instant Radiosity or PBGI (Sec. 3.4), where the number of views can easily reach many thousands, each of them requiring a specific LoD. For such applications, most principles of current GPU LoD techniques are contradicted; the rasterization cost is comparatively low (only few pixels are actually drawn), but the cost for selecting the various LoDs for each view is high. Our basic idea to address such *many-view* problems is to exploit fine-grained parallelism to progressively define many cuts in a tree, where each cut corresponds to the LoD of a particular view. The parallelized many-view LoD is achieved by dynamically creating many threads based on a small-scale iterative data-amplification mechanism (e. g. via the geometry shader's stream output). To preserve a balanced workload, we can limit the action of each thread to a 1-edge walk (either up or down) in the tree structure. We demonstrate our approach on a classical point tree based on a Bounding Sphere Hierarchy (BSH) and apply it to several rendering techniques, including adaptive point-based rendering, Instant Radiosity, PBGI and reflection/refraction mapping.

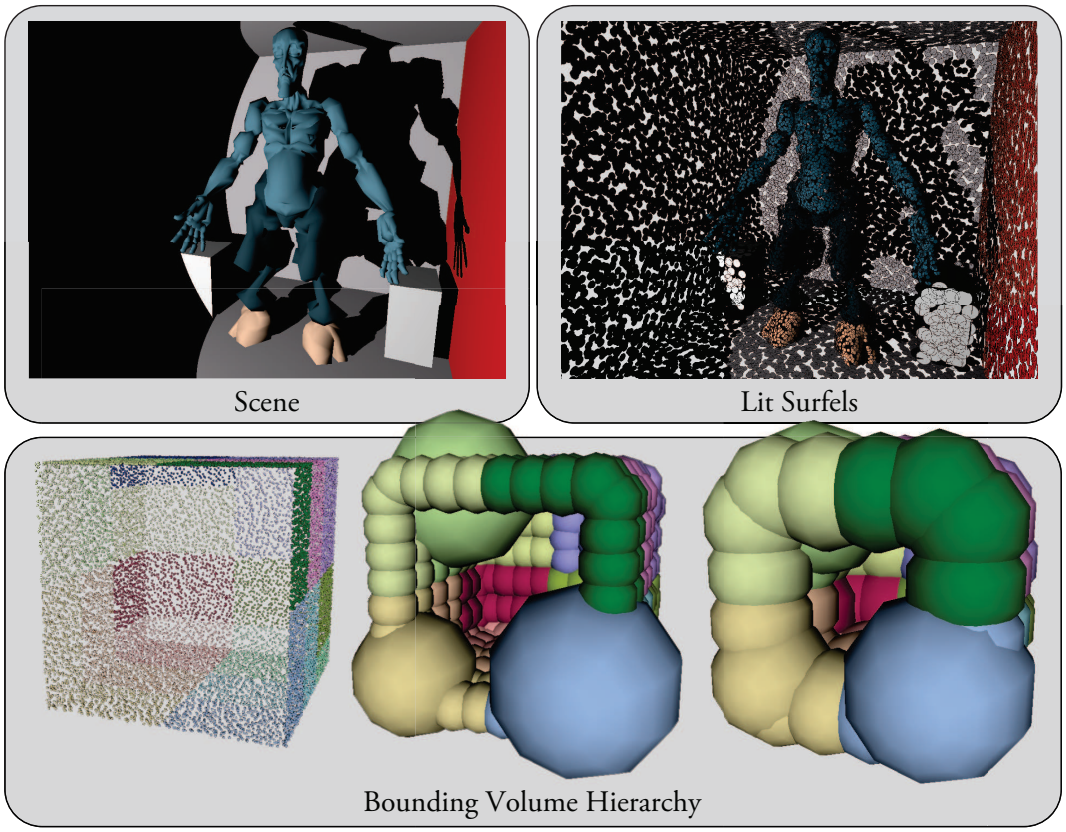



**Figure 3.12:** Top row, left to right: Input scene, lit surfels generated from a point-sampling of the scene. Bottom row: Different levels of a Bounding Volume Hierarchy using bounding spheres and a perfect complete binary tree. The overly conservative bounding spheres in the middle are caused by the $2^n$ sampling and split criterion. The spheres are coloured according to the different branches of the tree.

With our many-view cuts, we make the following contributions:

- Fine-grained parallel LoD selection for a large number of views
- Adaptation of incremental and lazy update schemes to many-view problems

### 3.5.1 Related Work

A large number of techniques cover LoDs [Lue+02] for 3D shapes and aim at representing and rendering geometric data, with a target amount of available time and/or memory. In this section, we focus on recent Hierarchical Level-of-Detail (HLoD) techniques based on trees and their applications to GI methods.

**Hierarchical Data Structures.** Hierarchical space subdivision structures [Ben75; Jac+80; Fuc+80] – and particularly Bounding Volume Hierarchies (BVHs) (e. g. using spheres) [Hub93] – are often used to represent different LoDs of a 3D shape.

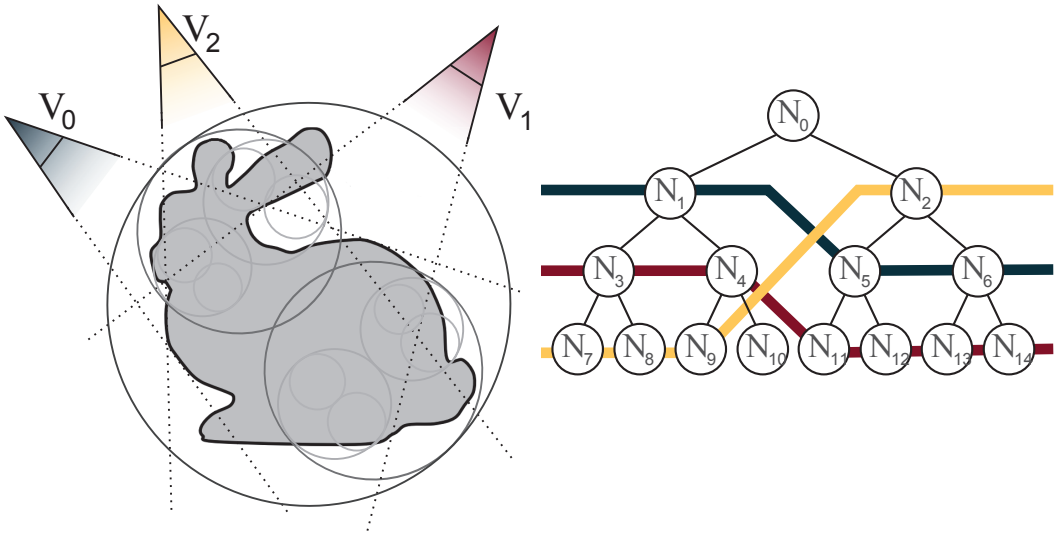



**Figure 3.13:** Three cuts (blue, red and yellow line) for three views $V1$, $V2$ and $V3$ on the scene (Bunny) contained in its Bounding Volume Hierarchy.

The idea is to define leaves of a spatial tree as geometric samples (points, vertices) and inner nodes as an approximation of their subtree (coarser polygons, sparser point sets). The construction can be done in a top-down manner by recursively splitting the shape's bounding volume, or bottom-up, by successively merging neighbouring elements into larger, i. e. upper, internal nodes.

Mesh-based LoDs [Hop96] including frame coherence [Xia+96] have recently been implemented effectively on GPUs [Hu+09]. However, a number of rendering techniques work even better using a much simpler, *point-based* structure.

Point-based HLoDs can be generated from unorganized point clouds without explicit connectivity information by storing point sets at various resolutions directly as internal tree nodes and leaves (Fig. 3.12). In particular, QSplat [Rus+00] uses a BSH organized as a binary tree where leaf nodes store surfels [Pfi+00] covering the surface (point, normal and optionally colour samples), while internal nodes store representative bounding spheres and normal cones of their respective subtree. Progressive visualization is possible by a coarse-to-fine level extraction. While QSplat was originally developed for out-of-core CPU execution, alternative HLoDs have been made in-core parallel for GPU execution via early GPU programmability [Dac+03]. These so-called sequential point trees can eventually be used in an out-of-core context, by decomposing a large point set into a forest of such trees defined as leaves of a coarse-grain out-of-core octree [Wim+06]. Hybrid techniques have also been introduced to combine point-based LoDs with mesh representations at finer scale, either in-core [Bou+05b] or out-of-core [Gob+05]. Ultimately, a given LoD is defined as a view-dependent *cut* in the underlying HLoD structure

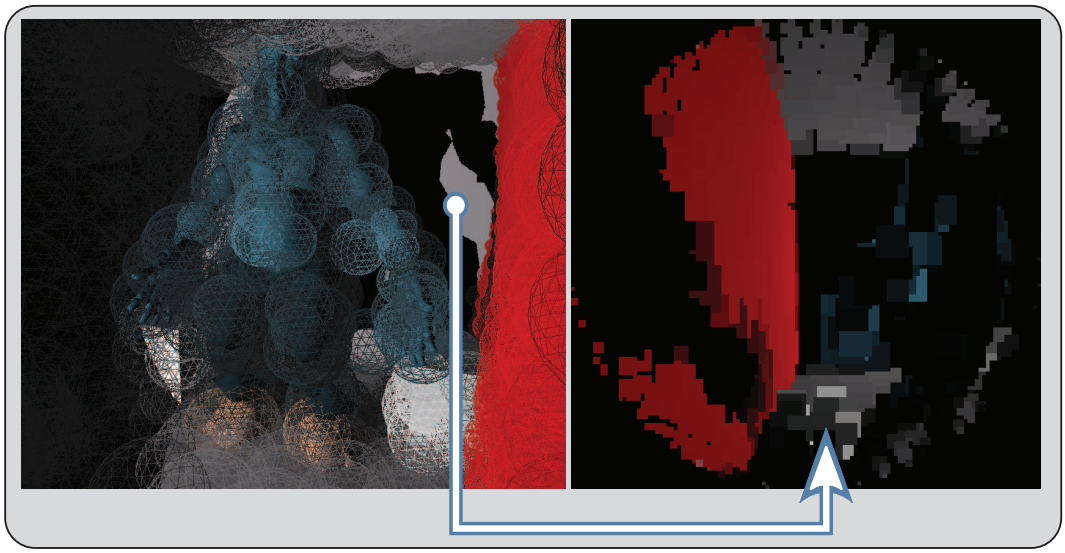


**Figure 3.14:** Cut for a single view placed on the wall (marked by disc). Left: Visualization of the spheres contained in the cut. Right: Rendered view generated from the cut using points.

(Fig. 3.13) and, in the following, we will put this LoD selection into the context of GI computation.

**Points** Points are a well-studied rendering primitive often resulting from a 3D capture process by a laser scanner, stereovision or from a (re-)sampling process applied to a digital scene. The resulting high density point cloud usually consists of tens or hundreds of millions of points [Lev+00]. Connectivity information is not always available and not even required for a convincing visual representation [Zwi+01]. An efficient data structure for rendering such a high number of points is crucial and HLoDs are a natural match to this challenge.

**Point Sampled LoD for Global Illumination.** The use of point sets for interactive GI has recently gained attention: VPLs [Kel97] have inspired many economic approximations for GI techniques. Laine et al. [Lai+07], assume neglectable motion, which allows the reuse of VPLs over time, Hašan et al. do so by temporal clustering [Haš+08].

Many GI methods rely on the ability to render many views of the scene from various locations. Consequently, for such many-view rasterizations, the process boils down to the definition of numerous cuts in the scene's hierarchy, i. e. one for each point of view (example location and cut in Fig. 3.14). In practice, considering a given HLoD structure and a given frame, hundreds or thousands of different cuts have to be generated.

Most HLoD methods have only focused on single-view extraction. Defining a cut sequentially – even if all cutting processes [Rit+09] or collections of nodes are treated in parallel [Hu+09] – does not exploit modern graphics architectures to the fullest. In particular, when all cuts have to be adaptive, a non-uniform distribution of computational workload is common. Balancing this workload is an important issue.

It turns out that many of the recent light-gathering methods rely on such a large number of views to be effective [Bun05; Chr08; Rit+09], but only little work has been devoted to this task. In particular, one key problem is to define a parallel many-cut algorithm which can be balanced on a fine-grained parallel architecture *within* and *among* the views' LoD extraction. This issue motivated us to address all views at once and to generate many LoDs in parallel instead of processing the views independently.

GPU LoD selection, such as in the context of intersection test acceleration [Zho+08; Lau+09; Gor+10] and parallel link creation for radiosity [Mey+09], can be done efficiently on modern graphics architectures [Eis+09].

We exploit the fact that graphics pipelines are best suited for high numbers of threads (Sec. 3.5.2). Sequential-over-$n$ and parallel-in-each-view approaches do not produce enough threads to be efficient (Sec. 3.6.2). We demonstrate how our approach can be successfully applied to smooth indirect shadows [Rit+08] and natural illumination from environment maps, among others.

### 3.5.2 ManyLoDs

Our algorithm computes a multi-cut in a BVH, corresponding to the many LoDs from a large number of viewpoints, using fine-grained parallelism that fits modern GPUs. We will use the following terminology: A BVH is a hierarchical set of nodes with a tree structure: BVH $:= \{N_i\}$. A *node-view* is a pair $(N_i, V_j)$ corresponding to a node in the BVH and a given view. Further, we define a *cut* $C$ through a BVH according to a criterion $c$, which is a scalar function of node-views, with the property:

$$\forall N_i, N_j, V_k : c(N_i, V_k) \leq c(N_j, V_k) \Leftrightarrow \text{level}(N_i) < \text{level}(N_j),$$

that is, $c$ decreases monotonically with increasing BVH level. In our experiments $c$ is defined as the pixel size of node $N_i$ in view $V_j$ or 0 if the node is culled. The cut $C$ is the set $C := \{(N_i, V_j) \mid c(N_i, V_j) < \epsilon \;\&\; c(\text{parent}(N_i), V_j) > \epsilon\}$, where parent$(N_i)$ is the parent node of $N_i$ and $\epsilon$ a user-defined threshold. We define a node-view $(N_i, V_j)$ as *valid* iff $c(N_i, V_j) < \epsilon$.

We will now describe our algorithm for a general parallel machine that can append elements to lists, i. e. using prefix scan [Ble89] (Sec. 3.7) or geometry shader vertex stream output (Sec. 1.3). Further, we assume that a BVH is given. We will first describe our basic method and then extend it to an incremental and a lazy-update approach. Details for a geometry-shader implementation are given in section 3.5.3.

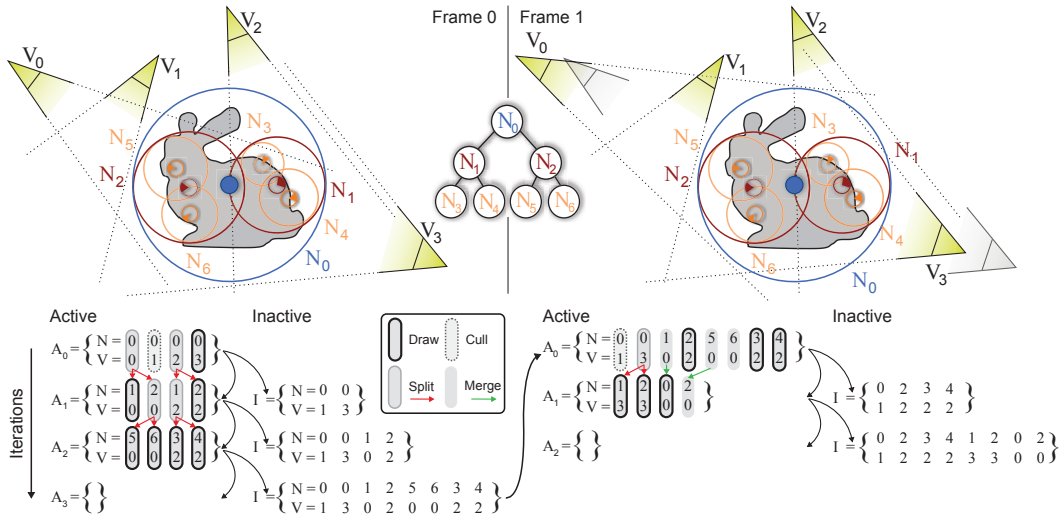



**Figure 3.15:** Basic flow of our algorithm. In two consecutive frames (left and right) the scene geometry (Bunny) represented as a BVH (spheres) is rendered into multiple views (V0 to V3). Left: the tree is traversed using multiple iterations (vertical arrow) in parallel over all node-views. The traversal starts with the active list $A_0$ containing the root node $\mathrm{N}_0$ for each view. In every iteration, the node-views are either culled, merged (green arrows), split (red arrows), or drawn. The remaining active list is used as input for the next iteration step. Culled nodes are not removed from the list to preserve them as a starting point for future frames. After all iterations the active list $A_3$ is empty and the inactive list $I$ contains the cut for all views. Right: The result of the last frame is used as input for the next frame with altered camera positions. This leads to fewer iteration steps (see incremental version Sec. 3.5.2).

### Basic approach

To find all cuts through an $n$-ary tree of maximum height $h$ for $m$ views in parallel, we manage two lists $I$ and $A$ of node-views: List $I$ is write-only and contains *inactive* node-views, that is, node-views which are found to be in the cut and do not require further processing. Node-views in $A$ are considered *active* and do need further processing (Fig. 3.15 left side only).

Initially, we set $I = \emptyset$ to be empty and $A_0 = \{(N_0, V_0), \ldots, (N_0, V_{m-1})\}$ to contain the root node paired with each view. To find the appropriate LoD for all views, we perform $h$ steps: In step $k = 1 \ldots h$ a parallel kernel is executed on all node-views $a \in A_{k-1}$. The kernel appends $a := (N_a, V_a)$ to $I$ if it is valid. Otherwise, new node-views resulting from a splitting operation are appended to $A_k$, one for each child node of $N_a$ and with the view $V_a$. After $h$ steps, $I$ contains the multi-view cut and can be used for further adaptive processing and rendering.

### Incremental Approach

Instead of starting from $I = \emptyset$ and $A_0 = \{(N_0, V_0), \ldots, (N_0, V_{m-1})\}$ we start from the resulting cut of the last frame [Xia+96], i. e. last frame's $I$ is used as $A_0$, building

on the assumption that cuts are likely to remain similar (Fig. 3.15, right side). This requires some modifications to our algorithm. Firstly, node-views must be merged when $(\text{parent}(N_i), V_j)$ becomes valid, e. g. the camera moves further away. Secondly, node-views that are culled must not be discarded to preserve them as a starting point for later frames, e. g. a node-view gets out of the frustum and back in again. However, we can merge culled nodes when the parent node is culled. Consequently, the new kernel performs one out of three different actions on a node-view $a \in A_{k-1}$ (mind the definition of *validity* of section 3.5.2):

1. If $a$ is valid and its parent $a'$ is not: $a$ is appended to $I$.
2. If $a$ is valid, its parent $a'$ is valid and $a$ is the first child: $a'$ is appended to $A_k$. Note, that none of the other children of $a'$ are moved to $I$. This prevents having $n$ copies of the same parent node-view in $I$.
3. If $a$ and its parent $a'$ are not valid, all children of $a$ are appended to $A_k$.

If the scene has temporal coherence (which is the case for our applications), i. e. the geometry and the views change smoothly, the amount of work (i. e. operations to produce $I$) has shown to be at least one order of magnitude less when compared to a full restart of our algorithm at the root node-views.

#### Lazy Update Approach

When approximate cuts are sufficient, we can limit the number of iterations on $A$ to $q$. For example, if $q = 1$ during each frame only a single parallel pass is made over all active node-views which is instantaneously turned into a list of inactive ones. Consequently, a node-view is either left *as is*, culled, merged with sibling node-views, or split, but the process is not repeated within a frame. Therefore, the precision of the cuts might lag behind because $I$ might contain invalid node-views. However, the algorithm is simplified drastically by limiting the number of traversed edges to $q$ per node-view.

### 3.5.3 Implementation

This section provides implementation details for our algorithm regarding pre-processing and runtime. Our approach consists of two main parts. First, we build the BVH in a sequential *pre-process*. Without loss of generality, we use a binary BVH [Rus+00] with a bounding sphere and bounding cone for each node bounding the positions and the normal field of its subtree. We use a perfect and complete tree to simplify our implementation. Second, for each frame at *runtime* the BVH is updated once, if necessary, and the multi-cut is computed.

#### Pre-process

First, we sample the scene's surface uniformly into a set $P$ of $n = 2^d$ points. To transfer scene deformations to the BVH [Rit+08], each point $p \in P$ is expressed in

```
while( !isEmpty( A_k ) ) {
 for each a in A_k parallel {
    if( !isValid( parent( a ) ) {
       if( isValid( a ) ) {
          draw( a );
          append( I, a );
       } else {
          append( A_{k+1}, allChildren( a ) );
       }
    } else if( isValid( a ) & isFirstChild( a ) ) {
       append( A_{k+1}, parent( a ) );
}}}
```

**Listing 3.1:** Pseudo code for our incremental algorithm in a geometry shader using geometry stream-out capabilities. The definition of validity is given in Sec. 3.5.2.

local coordinates $p = (s, t, i_{\text{tri}})$, referencing the barycentric coordinates $(s, t, u := 1 - s - t)$ of $p$ on triangle $i_{\text{tri}}$.

Second, we build the topology of a complete binary tree of height $d + 1$ from $P$ in $k$ steps, during which we virtually split the set of nodes by re-ordering elements in $P$. To avoid sorting in every step of iteration, we use the approach of Wald and Havran [Wal+07]. Here, all nodes are sorted once for each dimension and then a divide-and-conquer approach is applied on these three sorted lists to construct the tree in $\mathcal{O}(N \log N)$. Step $k \in [0 \dots d{+}1]$ considers $2^k$ subsequences $Q_j$ of length $2^{d-k}$. For each $Q_j$, we sum up the volume of the left and the right sequence based on the $x$-, $y$- and $z$-median element. Then, we split $Q_j$ into two sequences by selecting the splitting plane which gives the minimal summed volume. Both resulting sets become nodes in the next step. Each resulting node $N_0 \dots N_{2^{d+1}-1}$ stores a sphere, bounding the positions, and a cone, bounding the normals, of all nodes below. While this tree is admittedly simple (for a discussion see section 3.8), it is very easy to update in every frame and has a small memory footprint due to its implicit structure.

### Runtime

During each frame, we update the BVH (in case of dynamic scenes) before computing the multi-cut.

**Update** The BVH update is performed in a bottom-up manner. For every sample point $(s, t, i_{\text{tri}})$, i. e. every leaf node, we start a thread and update the point positions using the barycentric coordinates $(s, t)$ and triangle $i_{\text{tri}}$. This essentially transfers the deformation from the scene polygons to the BVH [Rit+08]. To update the bounding information of the remaining nodes, we proceed in a bottom-up order and start threads for all nodes of a level. Each thread merges the bounds of its children. This is similar to a 1D mip-map construction, but with special merging rules for bounding spheres and bounding cones. Note, that this step is only performed once, independently of the number and positions of the views.

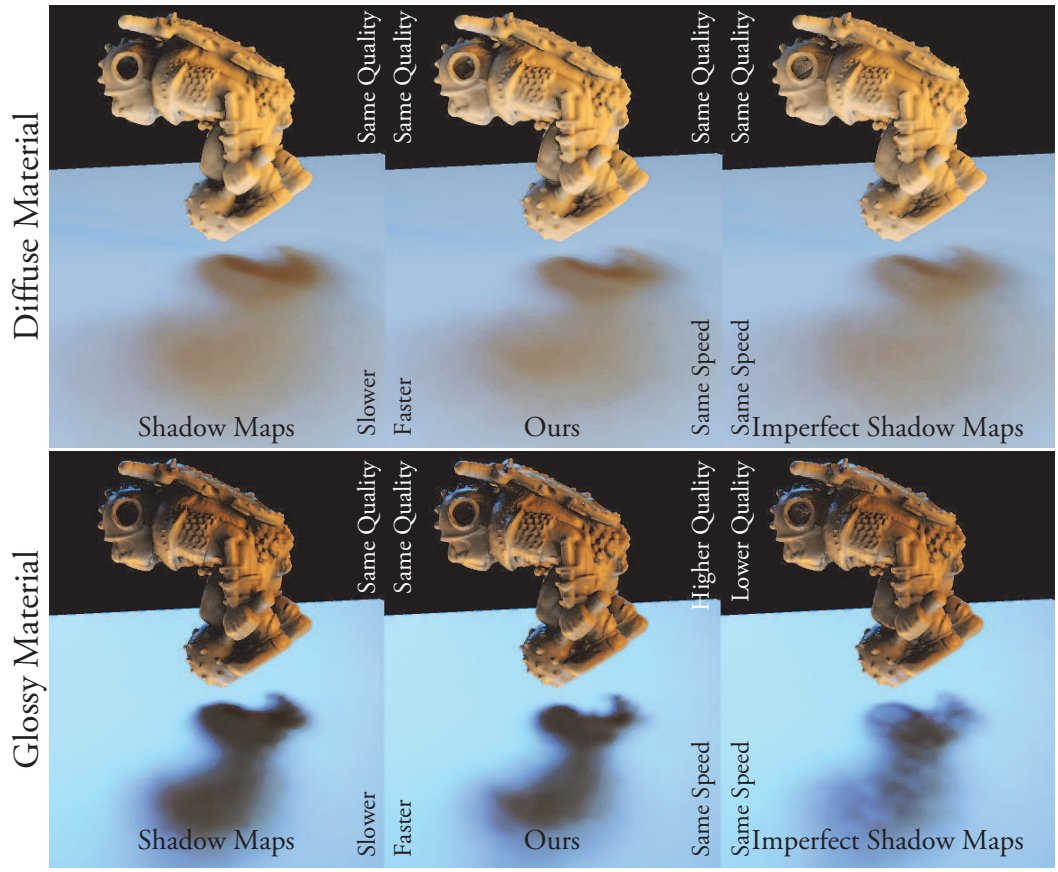



**Figure 3.16:** A dynamic character (500 k polygons, 500 k points) under natural illumination, using diffuse (top) and specular (bottom) materials. Natural illumination using 1024 point lights and comparison to common reference shadow maps (top left, bottom left), our many-view approach to rasterize shadow maps (top middle, bottom middle) and ISM (top right, bottom right). Our approach and ISM were adjusted to the same computation time of 4 ms. For diffuse surfaces, our approach and ISM result in similar quality at similar speed. For glossy surfaces, the imperfections in ISM are visible, whereas the quality of our approach remains uncompromised.

**Computing the Cut** We implemented our algorithm using the transform feedback / geometry stream-out capabilities of our Shader Model 5.0-compliant graphics card. Such hardware allows us to separately append elements to individual output streams while still maintaining a high number of threads. We store $I$ and $A_k$ in vertex streams and process them in a geometry shader as depicted in listing 3.1. Furthermore, we can draw nodes directly to the output when the corresponding node-view is inserted into $I$. This avoids one final iteration over all inactive node-views after the cuts have been computed. Consequently, we have the following options for a node-view $a$: split (create $n$ children based on the arity of $a$ and append them to the active list), merge (append the parent of $a$ to the active list if $a$ is the first child), draw (draw the node to the corresponding view and append it to the inactive list), cull (append $a$ to the inactive list, to not lose it as a starting point

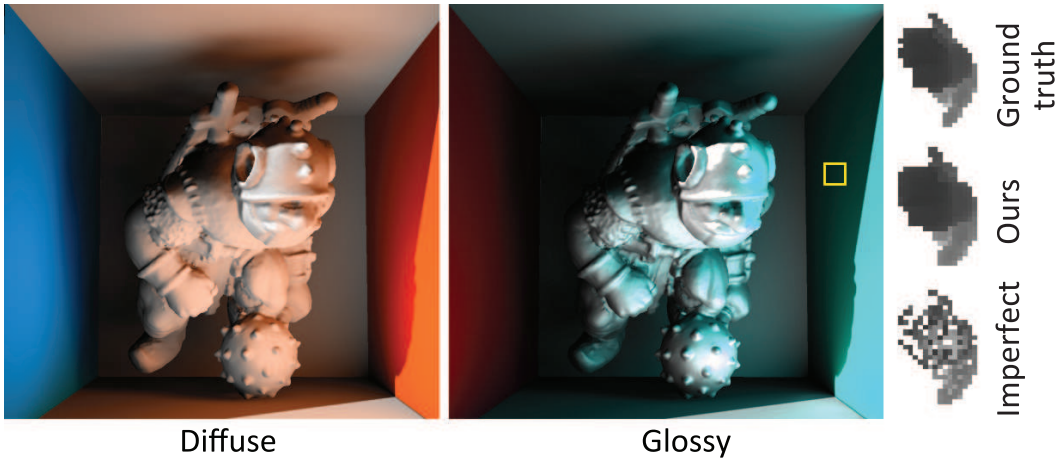



**Figure 3.17:** Instant Radiosity using 1024 Virtual Point Lights for the Grog mesh (500 k triangles, 500 k samples). Rendering time of 2 ms for diffuse (left) and glossy (middle) material at a resolution of 1024×1024. For a fixed time budget of 2 ms our algorithm is closer to the shadow mapping ground truth than ISM. The latter reveals holes for close occluders as indicated for the marked region.

in future frames). This geometry shader appends only a small number of values to a list (if any), checks the validity of two node-views and traverses not more than one edge up or down the tree. Such small-scale data amplification is considered an optimal scenario for a geometry shader.

As our implementation uses a BSH, we simply draw a point (GL_POINT) with a radius according to the respective bounding sphere. Each view is represented by a tile in a destination texture, which effectively avoids rendering $m$ cuts into $m$ separate textures (Fig. 3.19 right side).

## 3.6 Applications and Results

In this section, various applications of our ManyLoD algorithm to GI based techniques are described followed by a performance analysis of the various steps of our algorithm.

### 3.6.1 Applications

We will demonstrate our algorithm for various well-known techniques such as GI based on Instant Radiosity, PBGI and reflection/refraction mapping.

#### Instant Radiosity

Instant Radiosity [Kel97] has proven to be an excellent means for efficient GI. So-called VPLs are emitted from the primary light sources to simulate one bounce of indirect illumination. However, computing visibility from VPLs is a bottleneck

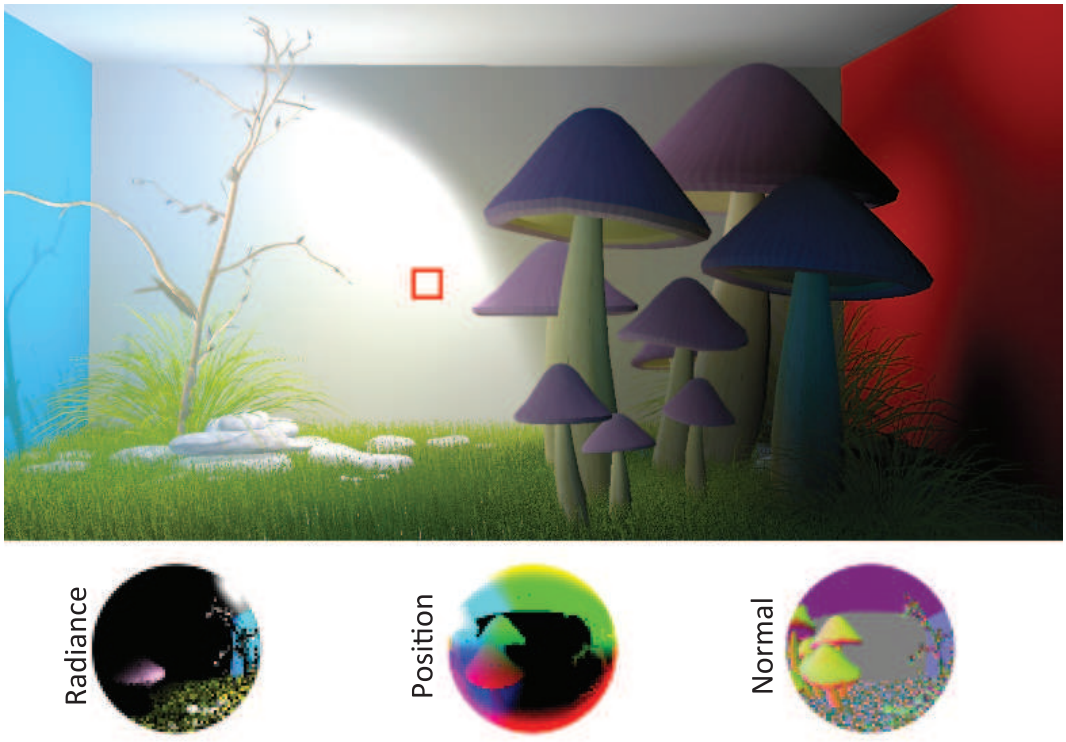



**Figure 3.18:** Top: Point-Based GI in a scene (700 k triangles, 1 M points) rendered at a resolution of 1024×1024 with 4096 views in 29 ms. Bottom: Radiance, position and normal for one indirect lighting view (red box).

and often evaluated either lazily [Lai+07], leading to temporal lag, or imperfectly [Rit+08], limiting the size of the scene. Here, a shadow map is computed for each VPL, and, for a realistic representation, a large number of VPLs is usually required. We compute the cut of the scene for all VPLs in parallel and improve upon previous work [Rit+08] in terms of quality at comparable speed, as demonstrated in figure 3.16 and for a glossy setting in figure 3.17. The figures exhibit the typical ISM imperfections. Given the same computation time, our method adapts the point density to avoid holes. To achieve temporal coherence, we enforce that each VPL maps to itself in subsequent frames by fixing the sampling pattern over time.

### Point-Based Global Illumination

Point-Based Global Illumination has become a valuable alternative to ray tracing and is increasingly used in production [Bun05; Chr08; Rit+09]. These techniques proceed as follows: The scene is rendered once for the actual view and the visible pixels are backprojected into the scene. For many of those pixels the scene is rendered again from their world-position into a set of indirect lighting views. This fits well to many-view rasterization. Finally, every indirect lighting view image is convolved with the BRDF simulating one-bounce GI. Previous work

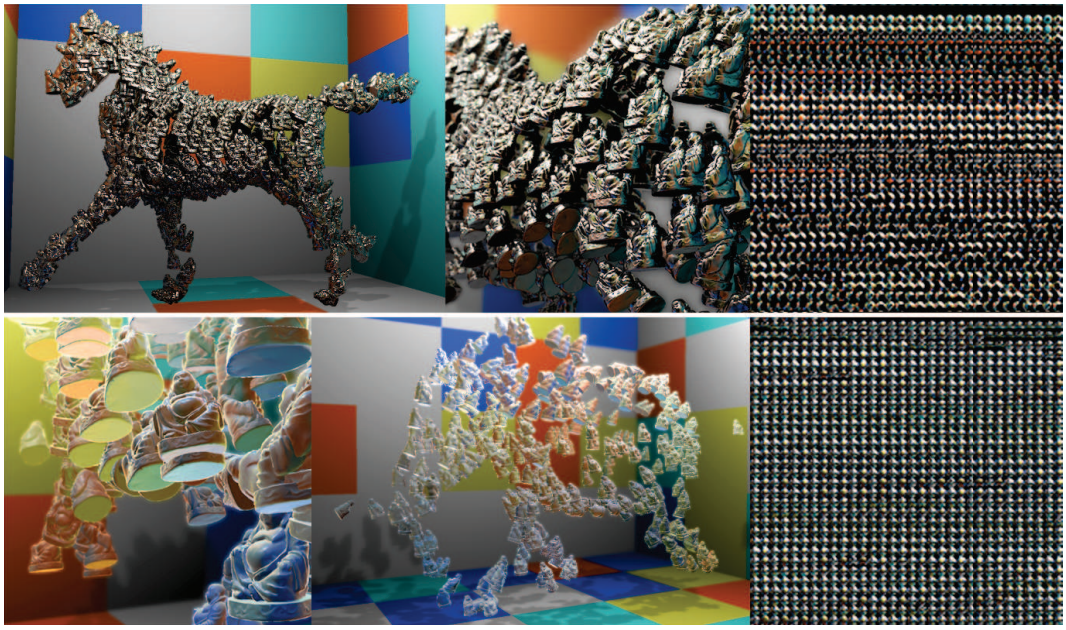


**Figure 3.19:** Rendering of 400 reflecting (top) and refracting (bottom) objects (2 M polygons, 2 M points), animated in the shape of an animal running inside a Cornell box. Our approach rasterizes all objects into every object's reflection map (right) in 40 ms.

was sequential [Chr08] or parallel-over-views [Bun05; Rit+09] with a sequential geometry loop in every thread. In contrast, we parallelize everything and find all cuts in a lit BVH for all visible pixels. While our quality matches previous work [Bun05; Chr08; Rit+09], it shows increased performance (Fig. 3.18).

### Reflections

Besides ray-tracing, environment mapping is a well-known method to render reflections [Bli+76; SK+05] and refractions [Wym05]. However, computing many environment maps is time-consuming, as all geometry needs to be processed. With our approach, we can produce an environment map per object by finding cuts in the BVH in parallel. In figure 3.19, we computed reflection maps on-the-fly for hundreds of objects.

### Further applications

Our algorithm is also applicable to a variety of point-based rendering techniques, e. g. single-view adaptive point-based rendering. Single-view rendering is just a special case of many-view rendering (with $m = 1$). Nevertheless, our approach can accelerate such techniques, e. g. rasterization of large meshes (Fig. 3.20).

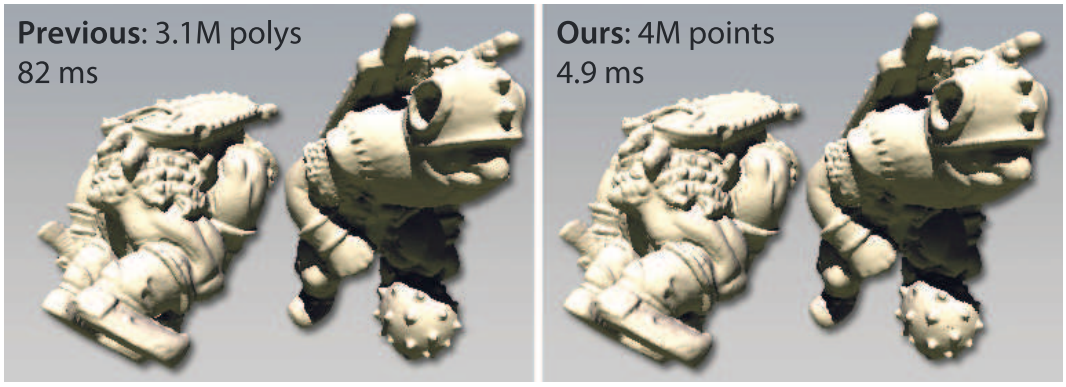



**Figure 3.20:** Timing (shading cost not included) for the Grog model (1.4 M vertices) using plain OpenGL (left) and our method (right).

| Technique | Dragon 4 M | Grog 1.3 M |
|---|---|---|
| Polys | 3.2 s | 1.1 s |
| 8 k points (ISMs) | 114 ms | 120 ms |
| Ours, non-incremental | 120 ms | 55 ms |
| Ours, incremental | 8 ms | 5.3 ms |
| Ours, lazy | 4.9 ms | 2.9 ms |

**Table 3.1:** Performance of our algorithms versus other approaches. We use the scene from Fig. 3.16 with 1024 VPLs and move the mid-sized area light by $\approx$ 10 degrees per frame.

| | Dosch 2.1 M ply 1.3 M pts | Grog 1.3 M ply 1.3 M pts | Sponza 72 k ply 1.3 M pts |
|---|---|---|---|
| Build | 10 s | 9 s | 7 s |
| Update | 26 ms | 26 ms | 15 ms |
| Poly | 46 ms | 11 ms | 3.8 ms |
| $c = 1$ px | 3.0 ms | 2.7 ms | 3.4 ms |
| $c = 2$ px | 1.4 ms | 1.7 ms | 1.6 ms |
| $c = 4$ px | 0.7 ms | 0.8 ms | 3.8 ms |

**Table 3.2:** Performance of our algorithm for building and updating the BVH and cut finding for different levels of quality in 1024$\times$1024. Our terminal condition $c$ is the size of the bounding sphere of a node in screenspace given in pixels.

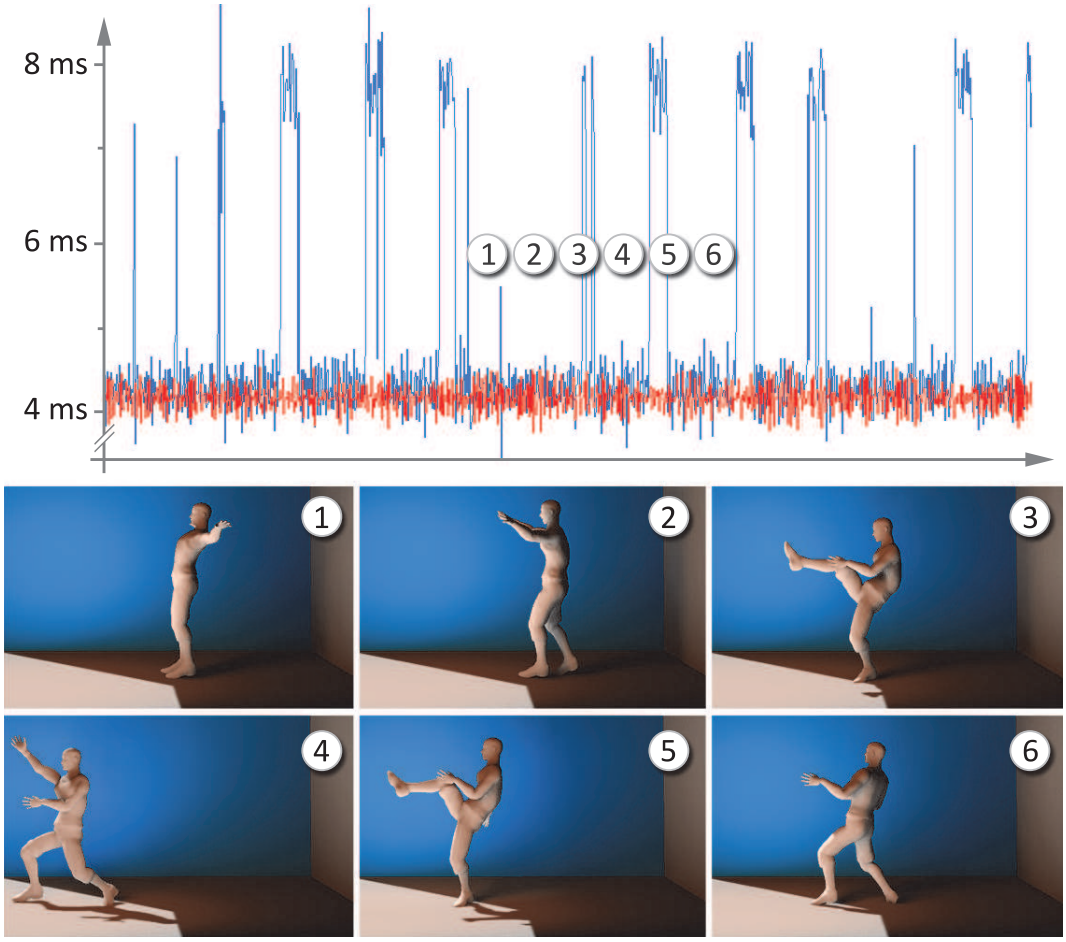



**Figure 3.21:** A plot of the computation time for every frame (top) for the kicking animation (bottom). The blue plot shows the computation time for the full, the red one the computation time for the lazy approach. Computation time increases for fast, i. e. incoherent parts of the animation (kicking in 3 and 5). Using the lazy approach, this can be prevented, at virtually the same quality.

### 3.6.2 Results

We implemented our algorithm using OpenGL 4.1 and measured its performance using an NVIDIA GeForce GTX 480 equipped with 1.5 GB of memory. Timings are given for meshes of different sizes in table 3.1 and each individual component of our algorithm in table 3.2. A complete rebuild of the underlying BVH takes substantially more time than a simple update due to the re-sorting of the sampled-points and the update of the GPU buffers.
The quality of the BVH depends on the mesh deformations because the tree structure does not change unless a full rebuild is performed. For most common deformations, e. g. character animations, we can re-use the tree structure and avoid such a complete tree rebuild. If there is little coherence, as in the case of fast animation or camera movement, our lazy updates can be used instead (Fig. 3.21).

The difference is perceptually insignificant due to the low-frequency nature of most global illumination effects.

## 3.7 ManyLoD Variations

Unfortunately, our ManyLoD algorithm presented in section 3.5 does not come without its drawbacks. In the process of rendering, three lists are required: one containing the node-views to be processed (*todo*-list), a second list containing all node-views belonging to the final cut (*finished*-list) and a third list that is used for ping-ponging the todo list to avoid read/write race conditions that might occur when our kernel operates on the todo list. This implies high memory usage considering that each list might potentially contain $\#(\text{views}) \times \#(\text{leaves})$ elements, i. e. all views to be processed *see* all leaf nodes. As a side effect, less GPU memory is available for other resources such as the tree to be processed, render targets, etc.

This section will present a solution to that problem that will avoid temporary storage of the final and temporary cut in global memory at all by using *in-place* shading and a stack-based traversal method exploiting shared memory.

The second variation of our algorithm deals with a hybrid, i. e. CPU/GPU, approach where the input data is split and processed by different physical units.

These two versions can also be used within high performance architectures that do not support display output. But first, we will review the principles of the prefix scan operator on which both techniques rely.

### 3.7.1 Prefix Scan

The prefix scan operator as introduced by Blelloch [Ble89] is a fundamental algorithm in parallel computing with many applications [Ble93] such as solving of linear systems, implementing radix sort and stream compaction. A binary associative operator $\bigoplus$ and its identity element $i$ applied to an input sequence

$$\left[\ e_0, e_1, \ldots, e_{n-1}\ \right]$$

returns

$$\left[\ i, (i \bigoplus e_0), (i \bigoplus e_0 \bigoplus e_1), \ldots, (i \bigoplus e_0 \bigoplus e_1 \bigoplus \ldots \bigoplus e_{n-2})\ \right].$$

As an example, the addition operator, applied to the following sequence

$$\left[\ 2, 3, 1, 0, 4, 2, 1, 2\ \right]$$

will result in

$$\left[\ 0, 2, 5, 6, 6, 10, 12, 13\ \right].$$

Which is generally called an *exclusive* scan. This is the scan version on which we will concentrate and leave the reader with the side note, that an inclusive scan can

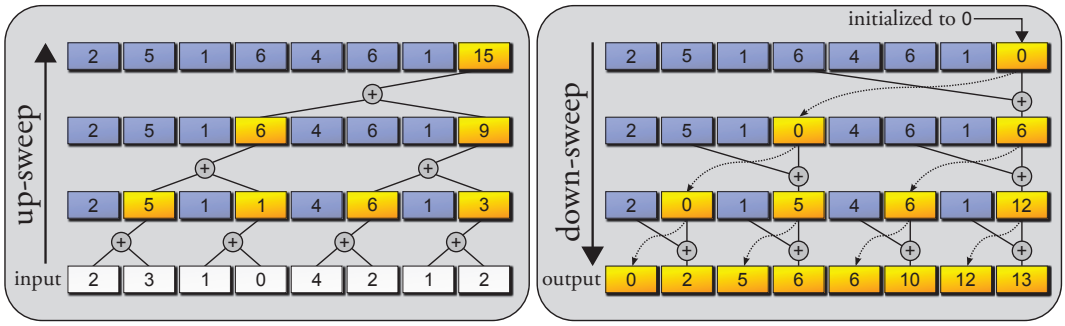



**Figure 3.22:** Up-sweep (left) and down-sweep (right) phases for an exclusive scan operation. The process is demonstrated *in place*, i. e. on a single array without temporary storage. The input sequence is marked in white whereas changes to the array are marked in yellow and unmodified values are represented in blue. Please note, that the up-sweep phase is depicted bottom to top, whereas the down-sweep phase is shown top to bottom and starts by initializing the last element to $0$. During each step of the down-sweep phase, each element first updates its right child node with the sum of its own value and the former (blue) value of its left child. Afterwards, it passes its own value to its left child (dashed arrow).

be generated from an exclusive scan by removing the first element and appending an element that holds the sum of the last element of the exclusive scan and the last element of the input sequence, resulting in $\left[\ 2, 5, 6, 6, 10, 12, 13, 15\ \right]$.

Blelloch describes a parallel methodology for implementing these algorithms with a runtime complexity of $\mathcal{O}(\log_2 n)$.

The exclusive scan consists of two phases. During the first phase – the *up-sweep* or *reduce* phase – the sum over all elements is computed in $\log_2(n)$ steps. Then, the last element is set to $0$ before executing the second – the *down-sweep* – phase. During this last phase the elements are processed in reverse order by first summing up the value of the element to be processed and its left child's value, and passing the result to the right child node. Secondly, each node passes its own value to its left child. Many different [Har+07; Dot+08] and specialized [Bax11; Hwu11] implementations exist. A simple version that operates on a single array demonstrating the process for the above-mentioned input sequence is provided in figure 3.22.

For the remainder of this section, we will concentrate on a very specialized version of the scan operator, the so-called *ballot* scan. In short, the ballot scan allows to execute an intra-warp scan where each thread of the warp contributes to exactly one bit of the result in very few operations. This sort of scan is particularly useful for the case where the elements of the sequence are either equal to the identity element or a constant $c_{\text{fixed}}$. Fermi (Sec. 1.4) provides this feature in CUDA via the `__ballot` instruction. A sample implementation based on Baxter's scan [Bax11] is given in listing 3.2. First, each thread computes its own flag or the flag is read from an array. Then, the `__ballot` scan instruction is executed where the thread with ID $m$ contributes to the $m$-th bit. Afterwards, the result for each thread is available

```
__inline__ __device__
static uint BallotScan_SingleWarp(
   const volatile uint* puiFlags,
   uint* puiSum )
{
   uint tid = threadIdx.x;
   uint uiFlag = puiFlags[ tid ]; // read flag
   uint uiBits = __ballot( uiFlag ); // ballot scan
   uint uiMask = bfi( 0, 0xFFFFFFFF, 0, tid ); // thread's bitmask
   uint uiExc = __popc( uiMask & uiBits ); // count bits for thread
   uint uiTotal = __popc( uiBits ); // total result: count all bits
   *puiSum = uiTotal;
   return uiExc;
}
```

**Listing 3.2:** CUDA ballot scan using a single warp.

by simply masking the result (set bits 0:m-1 to 1 and clear bits m:31) and counting the bits. The total sum over the elements can be found by simply counting all bits that are set to 1 (with `__popc`).

### 3.7.2 ManyLoD with in-place Shading

As mentioned before (Sec. 3.7), our ManyLoD algorithm has high memory requirements due to the assumed worst-case size of the todo-, finished- and temporary lists. The variant presented in this section will focus on the basic approach of ManyLoDs, i. e. at each frame the traversal starts at the root-node-views and allows only draw, cull and split operations.

When closely examined, it turns out, that the finished list is only used as a temporary storage as well. A traversal unit – the *cutfinder* – generates all cuts, that are then used by some *render component*, e. g. by splatting all node-views into their respective micro-view. Both units are executed sequentially, thus requiring temporary storage of the multi-cuts. The separation into different programs – a cutfinder represented by a geometry shader and a render component represented by a vertex and fragment shader with optional geometry shader for splatting discs – is dictated by the rendering pipeline (Sec. 1.3). It is currently impossible to efficiently run a shader program which creates an arbitrary, dynamically determined amount of primitives during one stage, and recursively execute the same kernel on part of this output data (please note, that the geometry shader's output is very limited and performance is decreasing with increasing output). However, these limitations can be overcome using a compute environment such as NVIDIA's CUDA that enables executing the render component *on demand*, i. e. directly when node-views belonging to the cut have been found.

The worst-case list size of $\#(\text{views}) \times \#(\text{leaves})$ elements derives from the breadth-first tree traversal of the algorithm. At each frame and for each step of iteration, a single non-leaf element in the todo-list might trigger $n$ elements to be appended

```
void Traverse( Node node ) {
   stack< Node > nodeStack;
   nodeStack.push( node );
   while( !nodeStack.IsEmpty() ) {
      Node current = nodeStack.pop();
      // Check for Split/Draw/Cull event:
      int iEvent = Process( current );
      if ( iEvent == SPLIT ) {
         PushAllChildrenRightToLeft( current, nodeStack );
}}}
```

**Listing 3.3:** Pseudo code for stack-based preorder tree traversal.

(for an $n$-ary tree), i. e. the todo-list might grow exponentially. However, it also means that a single view that *sees* all leaf-nodes requires storage of $n^d$ elements (where $d$ is the tree's depth). To avoid these circumstances, we will use a depth-first traversal method where the space complexity is proportional to the tree depth (Lst. 3.3).

In contrast to the original ManyLoDs, where each step of iteration is mapped to a kernel/shader program execution, we move the iterations to an inner `while`-loop, thus, after a single kernel execution, all node-views belonging to the final cut are found *and* rendered. Another difference is, that in the original version, during each iteration, the node-view to be processed by each thread is determined completely at runtime and is independent from all other computations within a block. With our in-place ManyLoDs, a group of views and their respective node-views is processed by a single block, i. e. a thread can only process a node-view belonging to this group. Although, we consequently lose the ability to balance the workload among the complete set of views, parallelism and work balancing is still maintained for the group of views treated within a block. As an advantage, this enables fast shared computations (via shared memory and L1 cache) and avoids global atomic operations during cut finding. More specifically, we utilize shared memory (Sec. 1.4) for mainly 3 tasks:

- A shared stack of node-views that require further processing.
- Temporary storage of finished/inactive node-views that are ready to be drawn.
- Scratch memory for prefix scan and a few block-local atomic variables.

Please note, that the term *stack* here is used conceptually only and in practice all push and pop operations are executed warp-wise, and only the first thread in each warp is responsible for atomic operations. However, all push operations will append elements at the end, likewise all pop operations remove elements from the end.

```
void TraverseAndRender() {
   uint tid = ThreadIdx.x;
   __shared__ stack< NodeView > s_TodoStack;
   __shared__ stack< NodeView > s_TempFinished;
   uint tid = TreadIdx.x;
   s_TodoStack.push( NodeView( 0, tid ) );      // root node-view
   while( !s_TodoStack.IsEmpty() && !allThreadsTerminated() ) {
      NodeView current = s_TodoStack.pop();
      int iEvent = GetEvent( current );         // draw, split, cull
      uint uiTodoIdx = Scan( /* ... */ );       // scan for todo
      uint uiFinishedIdx = Scan( /* ... */ );   // scan for finished
      switch( iEvent ) {
         case SPLIT:    // push all children onto stack at uiTodoIdx
         case DRAW:     // add node-view to finished at uiFinishedIdx
      }
      Render( s_TempFinished );
}}
```

**Listing 3.4:** Pseudo-code for ManyLoD with in-place shading. Please note, that all synchronization instructions have been omitted for clarity, i. e. each line should be seen as being executed by all threads simultaneously. During the `while` loop, node-views are popped from the stack and processed. Node-views to be drawn are added to a finished list, that is cleared, i. e. rendered, at the end of each `while`-cycle by the render component. In case of a split event, the child nodes are pushed onto the stack.

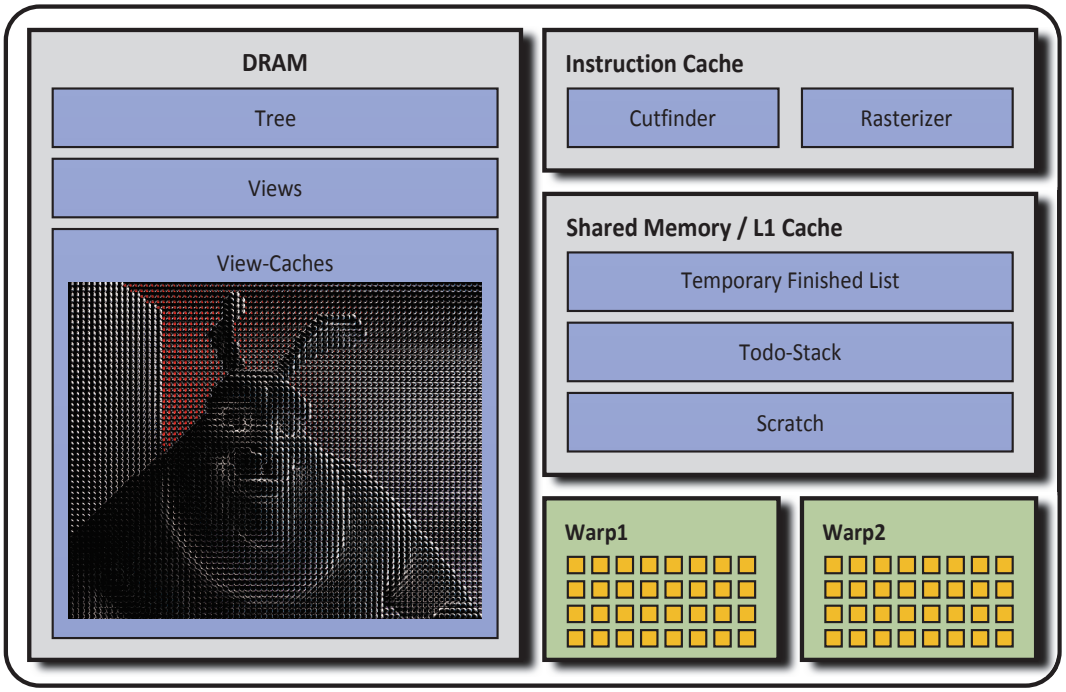



**Figure 3.23:** GPU components of ManyLoD with in-place shading.

The complete algorithm (Lst. 3.4) can be mapped efficiently to the GPU (Fig. 3.23) and works as follows: First, for kernel launching, the set of views is partitioned

into groups, e. g. based on screen-space tiles, and each group is handled by a single block.

During kernel execution, the stack of node-views to be processed – *todo-stack* – is first populated with root node-views for all views processed within the block. Then, a `while`-loop combining cut finding and in-place shading is executed until the stack is empty and all threads have terminated.

During cut finding, each warp pops a set of node-views from the stack. For each node-view (as in the traditional basic ManyLoDs) an action is performed, that is, it either needs to be drawn, split or culled (Sec. 3.5.2 and 3.5.2). As each element requires an index for altering the todo-stack or the temporary finished list, a scan operation for stream compaction is executed (based on the event). For an $n$-ary tree, where each node has exactly $n$ children with the exception of leaf nodes, a split-event will produce exactly $n$ children, and the general scan operation can be replaced by a fast ballot scan (Sec. 3.7.1). Then, to get the correct index, the ballot scan result for the todo-list is simply multiplied by $n$. Before the render component is executed, we synchronize all threads within the block, to make sure that all threads participate equally during rendering.

Our rendering component is responsible for rasterizing points to *view caches*. Depending on the application (Sec. 3.6) this often requires a depth and an optional colour-rendertarget. For our experiments we used 32 bit single precision floating point values for the depth and 24 bits for RGB colour values. In both cases, all rasterized pixels of a point primitive (determined by simple point-in-circle computation) invoke an atomic-min operation on the render target. The case requiring two render targets needs special attention to ensure that both targets contain the correct values while avoiding excessive locking. Please note, that most current graphics architectures do not support atomic-min operations on 64 bit, so that combining the values using bit shifting and a single atomic-min is not an option. Instead, we use the same approach as FreePipe [Liu+10] to pack the depth and colour values into two separate 32 bit values, where each contains the compressed depth-value in the higher bits. The process, for depth values in the normalized range of $[0, 1]$ and $8 \times 8 \times 8$ bit colour values, each in the range of $[0, 255]$, works as follows: First, depth values are mapped to the $[0.5, 1]$ range, which contains mostly numbers whose IEEE-754-2008 compliant floating point representation dictates a sign bit of $0$ and an exponent of $126$.[1] The value is then interpreted as an unsigned integer, shifted to the left by 9 bits, and its last 12 bits reset to zero. [2] Of course, precision is lost during this process, but we are now able

[1] The value $1.0$ in IEEE-754 representation has a a sign bit of $0$, an exponent of $127$ and a mantissa of $0$, thus $(-1)^0 \times 2^{(127-127)} \times (1 + \frac{0}{2^{23}}) = 1.0$ as opposed to the claim in [Liu+10], that "[...] *all the depth values will have the same signs (1 bit) and exponents (8 bits)* [...]". This can be verified using the program in section 7.3.

[2] This mapping causes values of $1.0$ (far plane) to be interpreted erroneously as $0.0$ (near plane). However, this effect has been neglected as values of exactly $1.0$ are rare in practice. For initial clearing of depth targets, we use the largest possible value, i. e. `0xFFFFF000` in FreePipe encoding – all 20 mantissa bits are set and 12 bits are reserved for colour values.

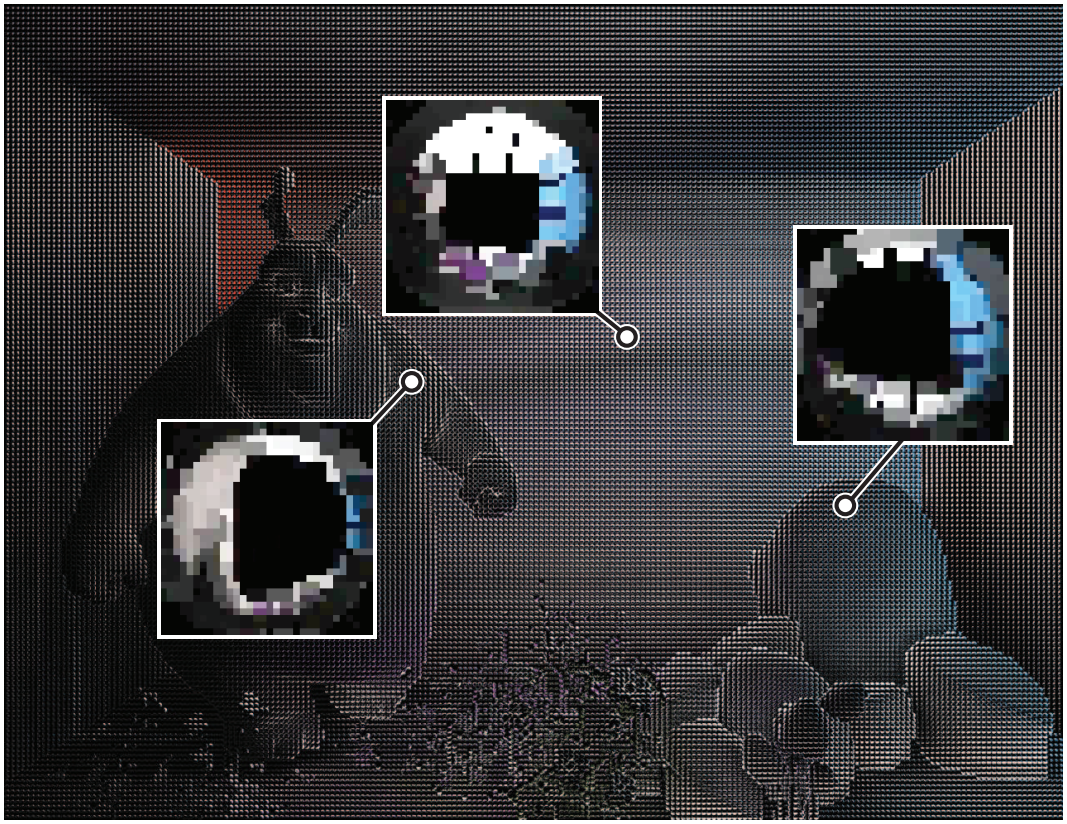


**Figure 3.24:** Rendering of 33280 views (208×160), where each view has a resolution of 32×32 pixels.

to populate two 32 bit values with this number, merge them with the 12 highest and lowest bits of the colour channels and perform two sequential atomic-min operations.

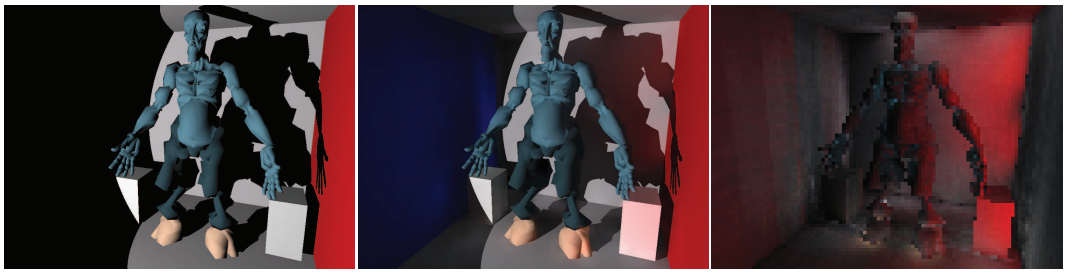
**Scheduling** Each thread that participates during cutfinding requires a certain portion of the shared memory, thus limiting the amount of *cutfinders* that can run on a block. For our experiments, the todo-stack has a capacity of $x(1 + (n - 1)d)$ node-view elements, where $x$ is the number of cutfinders, $n$ the arity of the tree and $d$ the tree's depth. Additionally, $x \times k$ node-views can be buffered temporarily before rasterization, where $k$ is a user-defined value. This enables us to launch 128 cutfinding threads per block. The remaining threads are idle during cutfinding but participate during rasterization. We chose a resolution of $32 \times 32$ pixel per micro-buffer so that each thread of the block is responsible for a single pixel.

### Results

We measured the performance of our in-place shading algorithm for various numbers of views (Tab. 3.3 and Fig. 3.24). With increasing number of views we achieve higher performances when running 128 instead of 64 cutfinders per

| Views | Technique | #Cutfinders | Cutfinding | Cutfinding +Rendering |
|---|---|---|---|---|
| 34×25 | Scan | 64 | 63.8 | 80.3 |
| | Ballot | 64 | 61.6 | 78.1 |
| | Scan | 128 | 65.4 | 84.3 |
| | Ballot | 128 | 64.4 | 82.9 |
| 68×51 | Scan | 64 | 234.4 | 293.5 |
| | Ballot | 64 | 225.6 | 285.4 |
| | Scan | 128 | 134.2 | 170.5 |
| | Ballot | 128 | 131.7 | 168.3 |
| 204×153 | Scan | 64 | 1829.3 | 2309.6 |
| | Ballot | 64 | 1766.5 | 2243.2 |
| | Scan | 128 | 1012.3 | 1307.9 |
| | Ballot | 128 | 995.1 | 1289.1 |

**Table 3.3:** Performance of ManyLoDs with in-place shading for various numbers of views. We measured the performance for cutfinding as well as cutfinding combined with rendering of the scan and the ballot scan version using 64 or 128 cutfinders per block. Each micro-view had a resolution of 32×32 pixels. All timings in milliseconds.



Direct Light Only | Direct + Indirect Light | Indirect Light (before upsampling)

**Figure 3.25:** ManyLoDs with in-place shading, indirect light was computed using PBGI and 102×76 views combined with bilateral upsampling. Each view had a resolution of 64×64 pixels.

block. Our implementation based on the ballot scan outperforms the regular scan implementation in all cases with the additional benefit of reduced shared memory usage and simplified implementation. We used a discarding rendering component for each technique which simply cleared the finished list without performing any rendering. This way, we were able to measure the time spent for cutfinding only, in contrast to cutfinding and rendering. Note, that the target texture size for 204×153 views is 6528×4896 due to a micro-buffer size of 32×32 pixels.

Indeed, our in-place shading version is similar to the original micro-rendering code as it utilizes CUDA to splat nodes as soon as possible onto a micro-view. However, due to our node-view data structure we can use the scan operation for stream compaction in shared memory and distribute the workload among all cutfinders

within a block in contrast to the micro-rendering strategy where one thread is solely responsible for finding the cut of a single view.

By executing both the cut finding component as well as the cut rendering component in CUDA, we do lose benefits of hardware accelerated rasterization using specialized transistors, nevertheless, we gained implicit grouping of views that might open up future research topics such as finding the *upper bound* cut for a group of views which might be used as a starting point for all views processed by a block.

Although our in-place shading technique cannot reach the performance of a purely shader-based implementation, we achieve reasonable speed and are able to handle more views (Fig. 3.25), higher per-view resolutions or deeper trees due to the reduced memory requirements. We believe that this extension is useful for memory constrained systems such as movie productions where the number of views is high and the worst-case list size easily exceeds the available memory.

### 3.7.3 Hybrid ManyLoD

In this section we will briefly present a hybrid, i. e. CPU/GPU, implementation of the basic ManyLoD algorithm (Sec. 3.5.2). This work is a collaboration with Quoc Dinh Nguyen conducted in the scope of the MediaGPU project. The goal of this implementation was to achieve even higher performance for our algorithm by sharing the workload of cut computation between the CPU and the GPU. We utilized StarPU [Aug+11] for task scheduling and provide CPU and GPU kernels to the library. All GPU kernels are implemented using NVIDIA's CUDA and a prefix scan based on a global segmented scan [Har+07]. More specifically, we divided the cutfinding process into two separate kernels: event generation and prefix scan. The event generation kernel classifies each node-view in the todo-list and generates the appropriate event (i. e. integer ID): either draw, cull or split. The prefix scan kernel is responsible for computing the required indices for the todo and finished list for each node-view (similar to the in-place shading algorithm in section 3.7.2), but also for writing back the actual result into global memory.

We set up StarPU to dedicate either 1 or 4 tasks for event generation and the prefix scan and measured the performance on a machine equipped with an Intel Xeon Processor E5540 with 2.53 GHz, 4 GB RAM and an NVIDIA GTX 480 (Tab. 3.4). As a reference we used a CUDA implementation independent of StarPU. The fastest hybrid cut computation was achieved with the event generation running on the GPU and the prefix scan operation running on the CPU with a task ratio of 1:1. Still faster than the reference is a similar version with a task ratio of 4:1. All other versions remain slower than our reference which might be attributed to the overhead due to splitting and merging of CPU/GPU data during iterations as well as the relatively small number of views. Still, we believe that a hybrid evaluation might be useful when it is necessary to harness the computational power of all physical processors within the machine.

| Events (GPU) | Events (CPU) | PrefixScan (GPU) | PrefixScan (CPU) | #Event Tasks | #Prefix Scan Tasks | Timings |
|---|---|---|---|---|---|---|
| – | – | – | – | – | – | 116 ms |
| ✓ | ✗ | ✗ | ✓ | 1 | 1 | 81 ms |
| ✓ | ✗ | ✗ | ✓ | 4 | 1 | 114 ms |
| ✓ | ✗ | ✓ | ✗ | 1 | 1 | 122 ms |
| ✓ | ✗ | ✗ | ✓ | 4 | 4 | 141 ms |
| ✓ | ✗ | ✓ | ✗ | 4 | 1 | 153 ms |
| ✓ | ✓ | ✗ | ✓ | 4 | 1 | 154 ms |
| ✓ | ✓ | ✗ | ✓ | 4 | 4 | 182 ms |
| ✓ | ✗ | ✓ | ✓ | 4 | 4 | 192 ms |
| ✗ | ✓ | ✗ | ✓ | 4 | 4 | 222 ms |
| ✓ | ✗ | ✓ | ✗ | 4 | 4 | 223 ms |
| ✓ | ✓ | ✓ | ✓ | 4 | 4 | 227 ms |
| ✗ | ✓ | ✓ | ✓ | 4 | 4 | 262 ms |
| ✓ | ✓ | ✓ | ✗ | 4 | 4 | 262 ms |
| ✗ | ✓ | ✓ | ✗ | 4 | 4 | 303 ms |

**Table 3.4:** Performance evaluation of our heterogeneous CPU/GPU implementation for 4096 views. The first row describes our ManyLoD algorithm running in a CUDA only framework and serves as a reference implementation. All timings in milliseconds.

## 3.8 Discussion and Conclusion

In this chapter, we presented several methods to traverse a BVH for multiple views in parallel using shaders, CUDA or a hybrid CPU/GPU setup. Each corresponding kernel exploits parallelism effectively by coupling nodes to views and performing iterative refinement of the cuts (in global, shared or main/video memory).

If the resulting cut from the previous frame is available and as long as camera motion and mesh deformations are coherent our incremental approach can be applied, otherwise a full traversal of the BVH is required. Unfortunately, very incoherent deformations, e. g. explosions, result in completely different BVHs and are kept as future work. For the demonstrated applications we only experimented with binary BVHs. However, higher arity trees can be addressed as well and require only to create $n$ children in the case of a splitting event. Also, the completeness of the tree is not strictly necessary. An unbalanced tree can be represented by a double-linked-list representation to support dynamic updates but requires additional memory. In general, our algorithm is compatible with any tree structure that can be mapped to a GPU, but relies implicitly on the performance of

the respective tree traversal operations. Using a complete tree though, gives a good tradeoff between tree-update performance and a sufficient sampling of the scene. Future hardware, that supports drawing of points and triangles simultaneously from within the same shader, would allow a tree with triangles stored at leaf-nodes and arbitrary bounding volumes for inner nodes.

Besides the shader based version, we presented an in-place shading algorithm implemented entirely in CUDA. This algorithm reduces the memory requirements significantly by performing stream compaction of node-views in shared memory. Although this version is limited to the basic ManyLoD approach and cannot exploit hardware rasterization, it allows to process a group of views on a single block while balancing the workload among all threads belonging to that block. The gained memory can then be used to process bigger trees, more views or views with higher resolution.

Finally, we presented a hybrid CPU/GPU implementation also based on the basic ManyLoD algorithm which could be extended to the incremental or lazy approach as well. Here, the overall workload is shared between different physical processors of the system and StarPU has been used for efficient task scheduling to achieve even higher performances.

Besides the presented ones, other rendering algorithms can profit from our method as well. Multi- or binocular-view-stereo, depth-of-field and motion blur can be achieved by rasterizing many views distributed across a stereo domain, a lens or in time. Irradiance volumes [Gre+98] require computing incoming radiance for a grid of $n^3$ view samples. Image-based photon mapping [Yao+10] shoots photons using environment maps placed at $n$ optimized view positions which could be generated using our approach. Further, acceleration techniques such as LightCuts [Wal+05] or row-column sampling [Ha07] may help to reduce the number of views, but still require to rasterize many of them. For future work, we would also like to consider transparency, e. g. participating media and out-of-core data streaming.

Overall, we have seen how to determine the required LoDs for a large number of views in parallel using small-scale geometric GPU kernels. In the next chapter, we will define an LoD per pixel by means of a geometric GPU kernel to remove aliasing artifacts in supersampling deferred rendering contexts.

CHAPTER 4

# ANTI-ALIASING IN DEFERRED SHADING

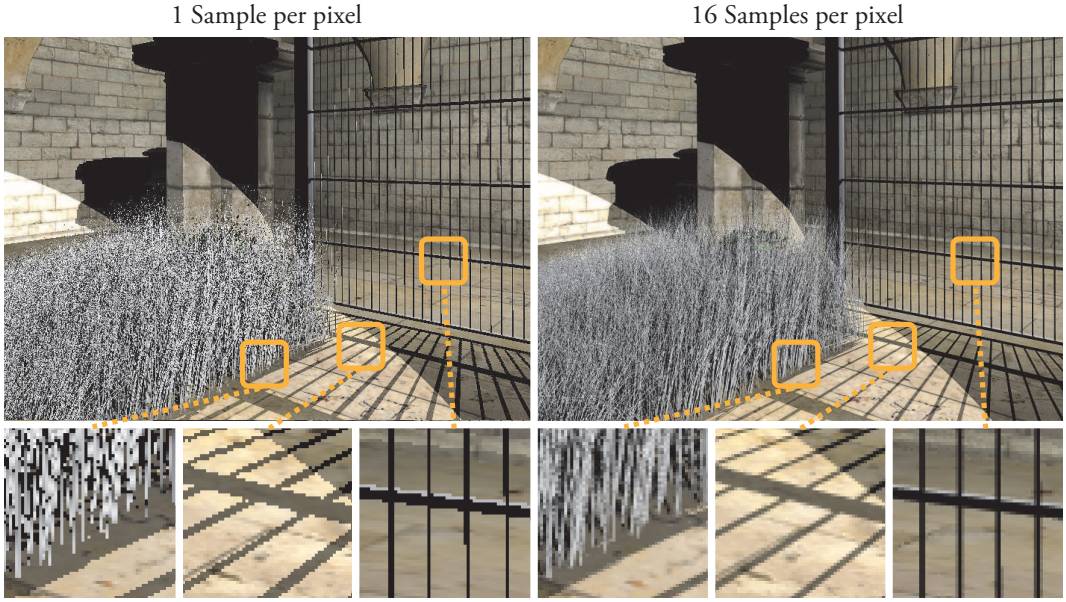



**Figure 4.1:** Left: Common aliasing artifacts due to an insufficient amount of samples per pixel. Right: With 16 samples per pixel, most of the artifacts are reduced.

Up to this point we have seen how to create high resolution surfaces and compute sophisticated lighting effects using adaptive approaches based on geometry synthesis. In this chapter we will tackle the last stage of the rendering pipeline that is responsible for image generation. More specifically, we will deal with efficient surface shading, computed from an intermediate scene representation, and the handling of aliasing artifacts. Once more, we will apply an adaptive strategy to speed up this process, this time, based on geometric analysis.

The process of applying lighting effects to the surface of an object is commonly called *shading*. Shading in real-time applications can be performed using a *forward-rendering* approach, which sends the scene's geometry to the GPU and performs lighting calculations for each rasterized pixel. During this process, the scene's

information resides only temporarily in GPU memory, thus, shading has to be computed right away. Traditionally, this includes shading pixels that might end up hidden, which leads to a substantial drop in performance when a large number of light sources is involved as lights cannot be culled efficiently. In addition, image-based post-processing effects that rely on scene attributes such as normals are often more difficult to combine with this approach because the temporary information is lost after shading. To avoid shading hidden fragments, one can use a *z-prepass*; the $z$-Buffer is priored by rendering the scene once without shading and only the depth values are stored. In a subsequent pass the geometry is transformed and rendered again but only fragments surviving the depth test are shaded. Nonetheless, even then, the computations do not map perfectly to the GPU when triangles are pixel-sized or very small and an overhead is added. A better approach is *deferred shading*. Instead of lighting each polygon, one writes the polygon's attributes to a Geometry Buffer (G-Buffer) [Sai+90]. This buffer commonly stores normals, depth values, material properties, etc. Pixels of the final image are then shaded via a post process by making use of the captured data in the G-Buffer, which results in higher shading performance. However, as a single G-Buffer pixel will result in a single pixel in the final output, aliasing can become a severe problem. One can address this issue by downsampling the resulting image. Nevertheless, this process is costly, as all full resolution pixels are shaded, even in regions where no Anti-Aliasing (AA) would have been required.

In this chapter, we will briefly review the causes for aliasing artifacts in section 4.1. We then examine deferred shading techniques that are common in real-time applications and describe their approaches to reduce or cope with aliasing artifacts in section 4.2.

Afterwards, we will present our solution in section 4.3 which is faster than the above-mentioned brute-force scheme that processes all pixels, and which leads to high-quality results. At the core, our algorithm evaluates two orthogonal metrics on the intermediate representation of the scene and allows for adaptive sampling of a supersampled G-Buffer during deferred shading. One metric is sensitive to geometric aliasing, whereas the other handles shading artifacts. Our approach concentrates the workload where needed, allows for faster shading in various supersampling scenarios and can even be combined with efficient anisotropic texture filtering.

## 4.1 Aliasing Artifacts

Formally, a function $f$ with $f(x) \in \{-t, t\}$ can be reconstructed from its samples, if the sampling interval is not larger than $\frac{1}{2\omega}$. If the function is sampled less than this so-called Nyquist-frequency, $f$ cannot be reconstructed completely and we call $f$ *undersampled* (Fig. 4.2). In rendering this can lead to aliasing artifacts of different nature.

**Geometric Aliasing** Geometric aliasing occurs at the boundary of primitives.

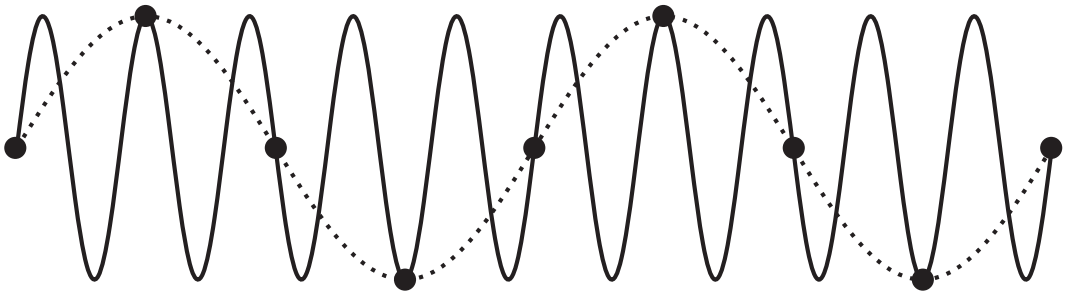


**Figure 4.2:** The function (solid line) is not sampled dense enough so that high frequencies are interpreted as low frequencies (dotted line). Figure adapted from [Wol07].

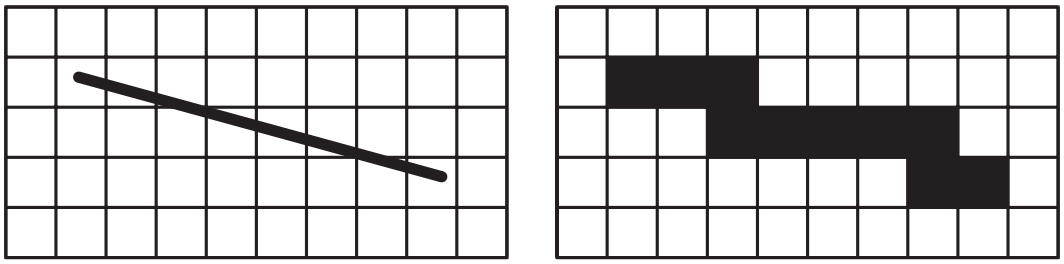


**Figure 4.3:** Left: Nearly perfect line with a raster grid as overlay. Right: Raster grid with filled pixels based on a single sample per pixel.

For instance, a line to be rasterized is described by its two bounding vertices but mathematically consists of an infinite set of points fulfilling $l(x) = \vec{v_0} + t(\vec{v_1} - \vec{v_0})$, with $t \in [0, 1]$. During rasterization, the line has to be sampled dense enough to ensure no loss of information. If this requirement is not met, aliasing artifacts can occur (Fig. 4.1 grass and bars and Fig. 4.3). The same is, of course, true for polygons and points.

**Shading Aliasing** Further aliasing artifacts can occur during surface shading when the on-surface signal, such as textures or the shading itself, is sampled insufficiently. Texture filtering methods [Hec89] can be used to solve the former. For instance, mip-mapping creates a hierarchical LoD representation and allows trilinear interpolation, i. e. bilinear interpolation within and among the texture LoDs. However, this process is only correct for surfaces that are orthogonal to the viewing direction and does not account for the angular distortion – called *anisotropy* – when looked at from an angle (Fig. 4.4). Anisotropic filtering solves this problem by denser sampling along the axis of greatest change, effectively increasing the required number of texture lookups.

Texture dependent techniques can suffer from similar artifacts. For instance shadow mapping can sometimes lead to jagged shadow boundaries that reveal the underlying pixel nature if the shadow signal cannot be reconstructed correctly from a texture (Fig. 4.1, middle closeup).

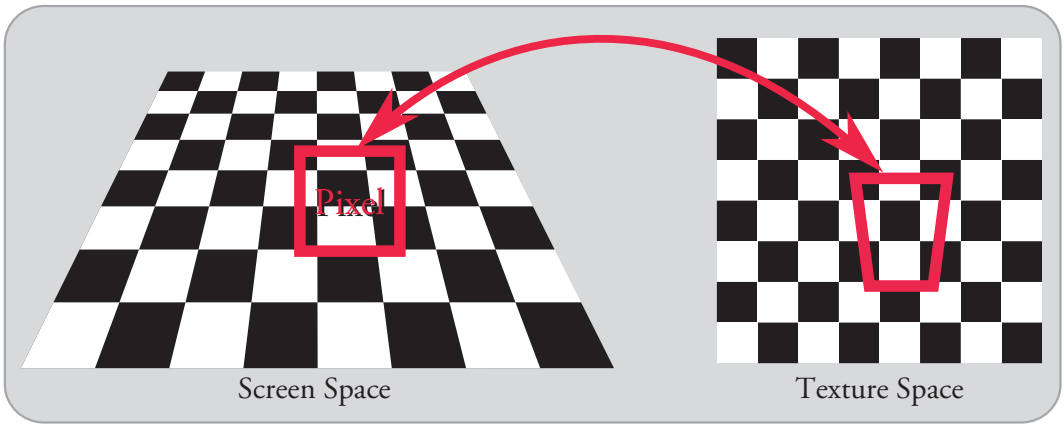



**Figure 4.4:** The left image shows a square pixel in screen space, whose corresponding footprint in texture space (right image) depends on a non-square region that needs to be sampled correctly.

## 4.2 Deferred Shading

Deferred shading techniques operate on the intermediate representation of the scene which is stored in several geometric buffers that altogether form the so-called G-Buffer (Fig. 4.5). The representation of these temporary buffers frequently implies a large memory footprint as all necessary information required for shading needs to be stored. The process of creating those buffers is often very memory and bandwidth intensive but can be combined with the previously mentioned $z$-prepass as well to exploit early $z$-culling [Mit+04]. To do so, the $z$-Buffer is priored by rendering the scene once without shading (i. e. only positional information is used) and just storing depth values. This depth-buffer is then used when producing the G-Buffer or for forward rendering. As hidden polygons are discarded, overdraw, memory access and bandwidth are reduced. However, scenes with alpha-masks and depth-modifying fragment shaders are more difficult to combine with this mechanism.

To reduce the memory footprint, the G-Buffer data is usually packed, e. g. only $\mathbf{n}_x$,$\mathbf{n}_y$ and $\text{sign}(\mathbf{n}_z)$ is stored for normals and $\mathbf{n}_z$ is reconstructed via: $\mathbf{n}_z = \text{sign}(\mathbf{n}_z)\sqrt{1 - \mathbf{n}_x^2 - \mathbf{n}_y^2}$. View-space positions for lighting can be deduced from the pixel position and the depth buffer. Often, it is possible to find a trade-off between precision, speed, and memory bandwidth. Several variations of deferred shading have been developed.

**Traditional Deferred Shading** Traditional deferred shading uses two passes. It was first suggested in 1988 by Deering et al. [Dee+88] using a hardware implementation based on a Triangle Processor, that writes out the triangle's information and a shading unit called Normal Vector Shader. To remove aliasing effects, they suggest, similar to supersampling, to render in higher resolution and

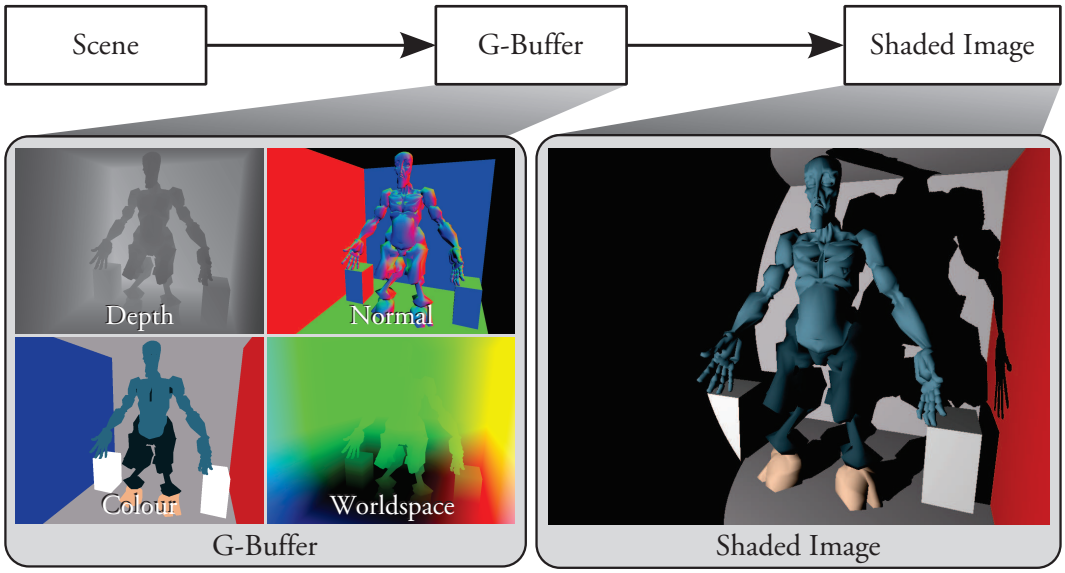



**Figure 4.5:** Left: G-Buffer storing depth, normal, colour and worldspace information of the visible portion of the scene. Right: Shaded image generated from the G-Buffer (and additional shadow maps).

downscale the result. Alternatively, one can render the geometry twice from a slightly jittered viewpoint.

More generally, Saito and Takahashi [Sai+90] used deferred shading for post-processing, e. g. image enhancement. The first pass fills the G-Buffer and the second pass is used for shading. To cope with AA and transparency, they recommend an *extended G-Buffer* that stores all surfaces belonging to a pixel for further processing.

Storing more surface information per pixel is crucial for AA and other effects. McCormick et al. [McC+99] use an accumulation G-Buffer and a list of distinct materials in each pixel. Recently, Liktor and Dachsbacher [Lik+12] followed the same principle. Here, the framebuffer stores an index to a Compact Geometry Buffer, which holds the shading samples in a linear buffer that is represented as a linked list. Hereby, stochastic supersampling becomes possible, as well as the reuse of shading values, and a variable shading rate.

One problem of deferred shading is that it can lead to *Übershaders* when different light types, different number of lights and various BRDFs need to be accounted for. Forward rendering allows for separation of object-based shaders, whereas deferred shading relies on a single global shader. If only a few limited BRDFs are involved, a precomputed 3-dimensional texture can be used which is indexed via an additional Material-ID parameter of the G-Buffer [Shi05], e. g. as $(\text{dot}(\mathbf{n}, \mathbf{l}), \text{dot}(\mathbf{n}, \mathbf{h}), \text{Material-ID})$, where $\mathbf{l}$ is the unit-vector pointing into the direction of the light source, $\mathbf{n}$ is the surface normal and $\mathbf{h}$ the half vector (half between view and light vector).

**Light Pre-Pass** Another solution to the material problem is a light pre-pass approach [Eng08]. While the first pass is similar, but skips writing an albedo value, the second pass differs. Here, the lights' volumes are rendered as convex bounding geometry (cones for spotlights, spheres for point lights, full-screen quads for directional lights) [Har+04]. Their contribution to the underlying G-Buffer pixels is accumulated in a separate texture. During the final pass, the geometry is rendered, including the material properties, and lighting is calculated based on the previously-derived light contribution. This approach even enables the use of different fragment shaders but the geometry has to be send twice. As an optimization, the stencil buffer can be used during the second pass, to exclude pixels in the final pass that are not affected by any light. One advantage is that, because the geometry is sent twice, AA can be used in the final pass, but lighting artifacts might occur due to the prestorage of the lighting terms.

**Inferred Lighting** For inferred lighting [Kir+09], the scene is rendered to a low resolution G-Buffer, with an additional attribute combining a normal-group-id and an object-id. The second pass is identical to light pre-pass rendering but uses a low resolution target as well. In the final pass, the geometry is rendered at actual resolution once again and a bilinear filter is used to upsample the G-Buffer and lighting buffer via a so-called *Discontinuity Sensitive Filtering*. The unique id and depth are used to efficiently reject pixels from different objects and thus, non-edge-regions can be shaded faster. Nonetheless, aliasing artifacts can occur where brightly lit and dark surfaces meet due to the upsampling of the lighting buffer. Additionally, the $z$-prepass optimization described above cannot be applied as the initial depth buffer (first pass) is at low resolution.

**Anti-Aliasing** For AA multiple samples per pixels need to be taken. This implies many shading evaluations in forward rendering, e. g. Multi-Sample AA (MSAA) is a very common approach. Here, coverage (visibility) is sampled finely, while shading is evaluated coarsely. It has been shown in [Thi09] that MSAA can be combined with deferred shading using a customized resolve operation to achieve plausible results. However, the stored samples are derived from MSAA samples, i. e. not all samples correspond to actual surface samples. Further, a fixed shading rate is used for edges instead of a variable one. SBAA [Sal+12] can be used to reduce storage costs in these scenarios by limiting the amount of per-pixel surface information considered for shading. This involves discarding primitives with small coverage to trade accuracy for a smaller memory footprint. A large body of work on real-time post-process anti-aliasing seeks to ameliorate image quality by adjusting the values in the colour buffer after shading a low resolution buffer and a good survey is provided by Jimenez et al. [Jim+11].

FXAA [Lot09], MLAA [Res09] or DLAA [And11] are purely RGB-based post process filters and can be used to get rid of some type of aliasing artifacts. They resample the input texture with a slight offset for edge regions, thus effectively blurring the edges. Although this can remove some of the deficiencies, it can lead to an overall blur. Further, false positives for non-edge regions can appear, and

the approach is often not temporally coherent. Some of those filters are known to work in supersampling modes, an academic description and the way samples are selected are however not available. SRAA [Cha+11] is similar to MLAA but relies on a super-resolution depth (& optional normal) buffer. Subpixel shading values are reconstructed using bilateral upsampling; weights of neighbouring shading pixels are weighted based on the depth/normal. In contrast to our method, they use a fixed subset of normal and depth values that is chosen blindly.

In [Val07] MSAA x2 is used to average the result of two samples. This leads to shading artifacts at boundaries of objects with different materials. CSAA (Coverage-Sampled AA) [You07] cannot be combined with deferred shading at all.

Low resolution rendering and careful upsampling can help to increase rendering speed. In [Yan+08], the authors render the image in low resolution and use bilateral upsampling for the final pass based on normal and depth information. A similar idea is known from the demoscene [Swo09] for multi-sample AA. A hierarchical deferred shading algorithm is used in [Ki10]. Here, the G-Buffer is generated for various lower resolutions, so-called *R-Buffers*, similar to mip-mapping. Rendering is performed from low to high resolution and shading from non-edge regions is copied to the next level. This way, edges are shaded at the highest resolution only.

Finally, Lauritzen [Lau10] uses a tile based deferred shading approach combined with quad-based light culling. Indeed they offer an adaptive shading rate using normal and depth information with additional G-Buffer storage cost for depth derivatives. However, this metric neglects shading discontinuities and does not offer anisotropic texture filtering.

Supersampling AA (SSAA) methods render in higher resolution and downsample the result. The samples are placed on a grid in an ordered, regular, sparse, rotated or jittered fashion. With SSAA, shading costs are increased, as more pixels need to be processed by the fragment shader stage but provide more subpixel accuracy for the final pixel. We will demonstrate how to reduce this shading cost and trade quality for speed which is especially useful when shading costs per pixel are very high.

## 4.3 Adaptive Supersampling for Deferred Anti-Aliasing

The idea of our algorithm (Fig. 4.6) is to produce a high resolution G-Buffer, but to perform shading only on a selected subset of these pixels. This subset is chosen via our metrics that predict where more samples are required.

At first, the scene's information is rendered into an enlarged G-Buffer of size $(m \times w) \times (m \times h)$, where $w$ and $h$ are the width and height of the window, and $m \in \mathbb{N}$. Each pixel of the final image corresponds to a sampling window of size $m \times m$ in the large G-Buffer.

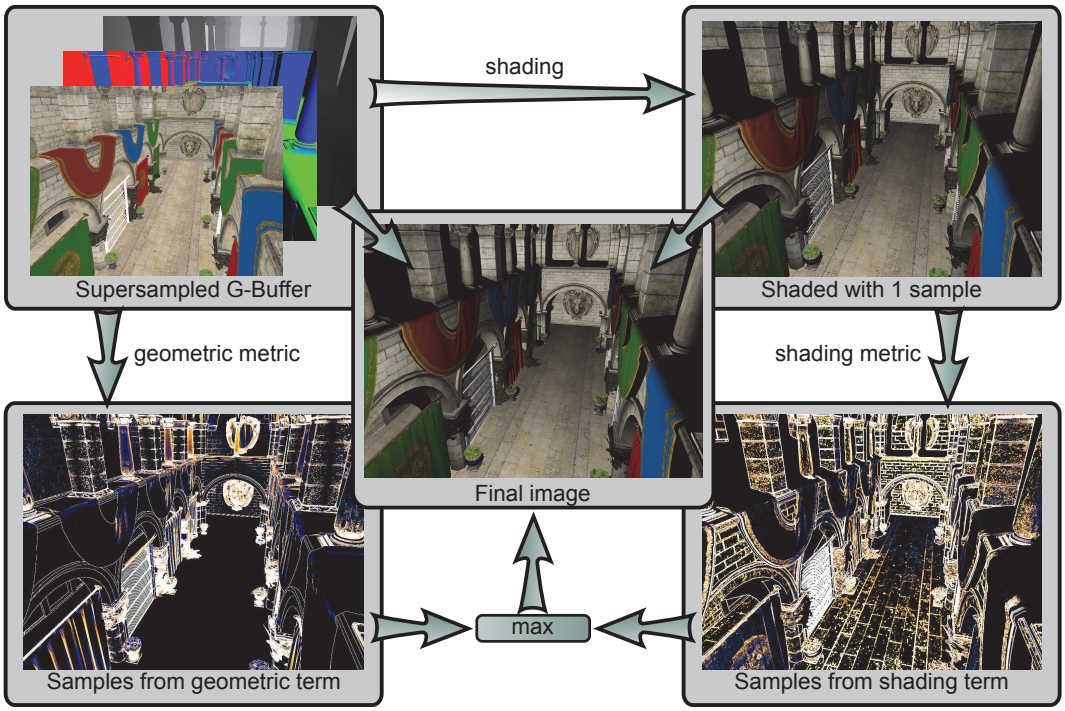



**Figure 4.6:** Overview of our method: Our deferred shading pass renders to an enlarged G-Buffer, i. e. each pixel of the final image corresponds to a window of this buffer. Then, the G-Buffer is evaluated using our geometric metric. Also, the first sample for each pixel is shaded and our shading metric is applied. For the final image, we shade each pixel adaptively based on the results of our metrics and the previously rendered image with one sample per pixel.

Secondly, we shade one sample for each pixel and store the result in a separate buffer $\mathcal{S}$, which has, hence, a resolution of $w \times h$. Finally, we will derive the number of additional samples $k$ that should be involved in shading a given pixel. This is done via two metrics, where the first one analyzes the contents of the G-Buffer within a sampling window and the second looks at the actual shading values in $\mathcal{S}$.

### 4.3.1 Determining the Required Sample Number

For a sampling window of size $m$, a maximum of $n = m^2$ samples per pixel is possible. Each metric has a user-defined constant $c$ to scale the resulting value that should be adapted according to the scene. As aliasing occurs at either geometric discontinuities or discontinuities caused by shading, we separate the evaluation and suggest the following geometric metrics.

**MinMax Depth (MM)** The required number of samples for this metric depends on the minimum and maximum depth value ($d_{\text{min}}$ and $d_{\text{max}}$) encountered in the sampling window:

$$k = c \times (d_{\text{max}} - d_{\text{min}}).$$

| Sampling | | Reference | ND | MM | NC | DV | NV |
|---|---|---|---|---|---|---|---|
| 2×2 | real-time | 3.80 | 3.29 | 3.01 | 3.36 | 3.22 | 3.27 |
| | stalled | 18.56 | 11.52 | 10.09 | 10.72 | 10.06 | 11.18 |
| | MSE ($\times 10^{-4}$) | 0.00 | 8.26 | 12.88 | 9.03 | 13.07 | 8.60 |
| | Samples % | 100.00 | 38.10 | 31.20 | 32.50 | 30.90 | 34.60 |
| 3×3 | real-time | 11.37 | 7.69 | 6.14 | 7.27 | 6.28 | 7.59 |
| | stalled | 97.36 | 42.77 | 29.00 | 39.36 | 30.49 | 39.89 |
| | MSE ($\times 10^{-4}$) | 0.00 | 3.47 | 6.47 | 3.86 | 6.28 | 3.80 |
| | Samples % | 100.00 | 26.90 | 16.0 | 22.80 | 16.80 | 23.00 |
| 4×4 | real-time | 21.48 | 15.68 | 10.90 | 13.66 | 11.54 | 15.33 |
| | stalled | 174.76 | 89.35 | 54.45 | 61.02 | 61.03 | 81.87 |
| | MSE ($\times 10^{-4}$) | 0.00 | 13.55 | 25.71 | 14.28 | 24.17 | 14.60 |
| | Samples % | 100.00 | 23.20 | 9.90 | 20.00 | 11.20 | 18.30 |

**Table 4.1:** Performance and quality of our geometric metrics. The reference evaluates all samples for each pixel. MSE indicates the Mean Squared Error regarding the reference. The percentage of samples indicates the overall amount of samples that participated in the shading of the final image. All timings are given in ms. Our metric based on minimal and maximal depth ( MM ) is the fastest, but has quite a high error. The metric with the lowest MSE and highest percentage of samples is marked in green ( ND ).

**Normal Cone (NC)** This metric focuses on the normals in each sampling window. We calculate the average normal $\bar{\mathbf{n}}$ from the sampling window and compare all other normals $\mathbf{n}_{ij}$ against it, effectively computing the conservative cone centered around the mean normal. The resulting number of samples is given by:

$$k = c \times \sqrt{1 - (\min_{ij}(\text{abs}(\text{dot}(\bar{\mathbf{n}}, \mathbf{n}_{ij}))))}.$$

**Depth Variance (DV)** We compute the variance of the depth values $d_i$ in the sampling window for the corresponding pixel to deduce a first estimate of the required number of samples, higher variance results in more samples:

$$k = c \times (\frac{\sum d_i^2}{n} - (\frac{\sum d_i}{n})^2).$$

**Normal Variance (NV)** We compute the mean normal $\bar{\mathbf{n}}$ of the sampling window and $k$ as

$$k = c \times \sum_i (1 - \text{dot}(\mathbf{n}_i, \bar{\mathbf{n}}))^2.$$

**Normal and Depth (ND)** This metric combines the result of our NC and DV metric as

$$k = \max(k_{\text{NC}}, k_{\text{DV}}).$$

A comparison regarding quality and speed of these different geometric metrics (MM, NC, DV, NV and ND) is given in table 4.1. Our ND-metric, accounting for normals *and* depth values, resulted in the least mean-square error while keeping the sample count low and will be used in all subsequent descriptions. However, shading based artifacts are not considered in any of these metrics and require a different approach.

**Shading based metric** This metric uses the shading texture $\mathcal{S}$ as input. Then, for each pixel the range $r$ of luminosity values is computed by looking at the four direct neighbouring pixels, resulting in $r = l_{\max} - l_{\min}$. The sample count is then defined as:

$$k = \begin{cases} 0 & \text{if } r < \max(t_{minThreshold}, l_{\max} \times t_{edgeThreshold}) \\ c \times r & \text{otherwise} \end{cases}$$

This is similar to metrics found in RGB-filter based AA methods such as FXAA [Lot09].

Finally, the result of the geometric and the shading metric are combined using a max filter. Notwithstanding, depending on the usage scenario (e. g. pure geometric aliasing), one could omit the shading metric. Furthermore, even other sampling methods based on the BRDF [Bag+12] could be integrated, but the above solution is faster to compute and sufficed in all experiments.

### 4.3.2 Sampling the Window

$$\begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix} \begin{pmatrix} 2 & 6 & 3 \\ 5 & 0 & 8 \\ 1 & 7 & 4 \end{pmatrix} \begin{pmatrix} 1 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 0 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}$$

**Figure 4.7:** Index matrices for sizes $m = 2, 3, 4$ that are used to sample the large G-Buffer.

Once the required sample count has been computed, we use $k$ samples from the G-Buffer of the pixel's corresponding sampling window to compute the final shading. Therefore, we distribute the $k$ samples according to an index matrix known from ordered dithering [Bay73] (Fig. 4.7). These matrices are constructed such that the average distance between two consecutive numbers is maximized. Hereby, we can ensure a good distribution of the samples that are taken from the supersampled G-Buffer.

### 4.3.3 Implementation

For our implementation, we chose a G-Buffer format (Fig. 4.8) that is similar to those found in games [Val07]. Additionally, we store all textures of the scene in a 3D-array. This allows us to delay the albedo lookup until the final shading pass where texture aliasing artifacts can be removed by using anisotropic filtering.

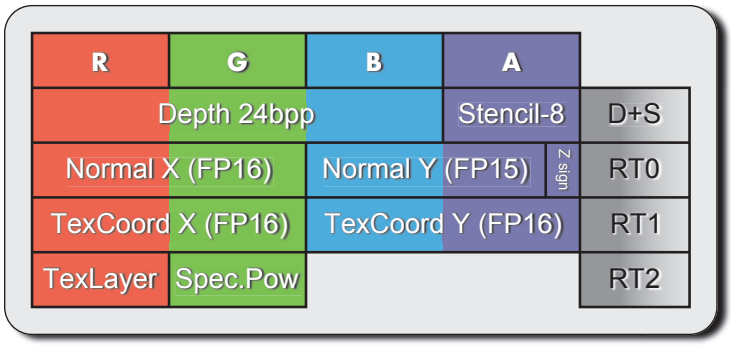



**Figure 4.8:** Our G-Buffer format. Each channel has 8 bits.

```
GBuffer g = RenderToGBuffer();
Tex oneSample = ShadeFirstSample();

// evaluation using our two metrics
Tex samplecount = Evaluate( g, oneSample );

ShadeRestAdaptively( samplecount, oneSample, g );
```

**Listing 4.1:** Pseudo code for our algorithm.

We implemented our algorithm using multiple passes (Lst. 4.1). The first pass is a deferred shading pass where the destination G-Buffer is $m^2$ times larger. As mentioned above, we do not store the albedo-texture value but the corresponding texture-layer index and texture coordinates.
The next pass computes the required number of samples per pixel by evaluating the sampling window with our metric. The final pass combines the result of the shade-first texture $\mathcal{S}$ and performs adaptive shading using the rest of the G-Buffer; samples are picked from the sampling window for each pixel according to the sample count and the dither matrices (Fig. 4.7).

### 4.3.4 Shading a sample

Each sample that contributes to the final pixel is shaded by evaluating the BRDF and performing a texture lookup into the scene's array texture. Although our metric does incorporate shading, we have to avoid texture artifacts.

By computing the derivatives of the texture coordinates in the G-Buffer and taking into account the number of used samples in the window, we can use anisotropic texture lookups to better approximate the underlying texture footprint of the sample. The derivative computation is available through the `dFdx/dFdy` functions in GLSL. The anisotropic texture lookup is then performed using the `textureGrad` function. Such a filtering would fail at the boundary of two different materials. Fortunately, we can detect these discontinuities because they usually


**Figure 4.9:** Left: reference using full 3×3 supersampling with anisotropic filtering, middle: our approach with anisotropic filtering, right: our approach without anisotropic filtering.

appear as a high derivative in the texture layer/coordinates. The influence of the anisotropic texture lookups on the result are shown in figure 4.9.

### 4.3.5 Evaluating several samples

There are several possibilities to evaluate multiple samples for a given pixel and our approach can employ several alternatives, whose choice depends on the hardware specifications.

The easiest solution is to read out the number of needed samples and use a loop in a shader to evaluate and accumulate the result of the samples taken from the window. In practice, this often proved most efficient, despite potential divergence between the different threads.

Another solution is to write out the required number of needed samples per pixel into a stencil target. For the final shading pass, the stencil test is enabled and a fullscreen quad is drawn for each possible sample count $k \in 2, \ldots, m^2$ where the stencil test discards all fragments different to $k$. This effectively groups all pixels with the same sample count during shading but requires the stencil mask to hold the correct number of samples and thus stencil write capabilities from within a shader. At the time of writing, this extension has not been integrated into the OpenGL standard (`GL_ARB_shader_stencil_export`). Similarly, the sample count can be written to the depth buffer using `gl_FragDepth` followed by an evaluation using multiple screen-sized quads which are set to the corresponding depth and a depth comparison with `GL_EQUAL`. However, this resulted in slower performance in all our tests of roughly 20%. The sample count values can benefit from *bucketing* by dynamically computing a histogram of the values and reducing the co-domain. Such a strategy, however, involves further investigation to avoid compromising quality and is left for future work.


**Figure 4.10:** a) Sample count per pixel as heatmap. b) Sorted sample counts. c) Low resolution sample count using a max filter on a 6x6 window.

Further, we experimented with sorting the pixels according to their computed sample count, inspired by [Hob+09] (Fig. 4.10b). Unfortunately, a group of pixels with the same sample count (e. g. all pixels with $k = 4$) does not necessarily have a high image-space coherence (Fig. 4.10a, 4.12), so that texture accesses within this group are usually incoherent, hereby, limiting performance. This version with sorted samples was three times slower than our presented version.

Also, we tried using a low resolution sample count texture by applying a max-filter to the original sample count texture. This filter preserves high sample counts on edges that require AA. Now, each texel of this low resolution sample count texture corresponds to a group of pixels in the image. Unfortunately, this increases the total number of samples drastically because high sample counts *leak* into regions that would only require a low number of samples (Fig. 4.10c). Again, sorting the low-resolution sample count values could not improve performance (also roughly three times slower).

### 4.3.6 Results

In this section, we present the results of our algorithm. We compare our method to a reference solution that evaluates all $n$ samples for all pixels with anisotropic filtering enabled. The reference and our algorithm are based on an ordered grid supersampling AA implementation. Performance results are illustrated in table 4.2. For each version, we measured the performance of real-time shading and shading that stalls the fragment shader (per sample), thus, effectively simulating a heavy and complicated fragment shader as can be found in offline or interactive rendering.
Also, we calculated the mean squared error (MSE), the PSNR (peak signal-to-noise ratio) and the percentage of samples in contrast to the reference solution. The timings were measured on an Intel Core i7 2.67 GHz with a GeForce GTX 480 with 1536 MB VRAM at a (application) resolution of $1024 \times 768$ for our OpenGL-based implementation. We used the Sponza scene with 280 K triangles and 194 K vertices for our tests.

| Supersampling | Method | ms/frame | stalled | MSE ($\times 10^{-4}$) | PSNR dB | Samples % |
|---|---|---|---|---|---|---|
| 2×2 | Reference | 26.0 | 78.7 | – | – | 100 |
| | Ours | 22.9 | 72.2 | 5.0 | 91.14 | 57.8 |
| 3×3 | Reference | 34.9 | 158.36 | – | – | 100 |
| | Ours | 30.69 | 143.77 | 3.08 | 93.24 | 53.40 |
| 4×4 | Reference | 57.48 | 317.76 | – | – | 100 |
| | Ours | 51.68 | 268.35 | 3.79 | 92.34 | 53.14 |

**Table 4.2:** Performance of our algorithm for the Sponza scene. The reference evaluates all samples for each pixel. PSNR (peak signal-to-noise ratio) and MSE (Mean Squared Error) given regarding the reference, similarly the percentage of samples shaded in contrast to the full evaluation is given. All timings in ms.

The impact of our geometric and shading term are shown separately in figure 4.11. Our geometric term also reduces the artifacts when small geometric details cannot be captured by our shading texture $\mathcal{S}$.

Our adaptive sampling algorithm outperforms the reference solution in all cases while keeping the quality at a similar level. Even for real-time shading scenarios, with low-cost fragment shaders, we can observe a gain in performance. Our metrics reduce the required number of samples to shade roughly by half. However, when shading only half of the samples, a performance boost of a factor of two is not possible due to the divergence of threads on the GPU and some incoherent texture accesses between neighbouring pixels.

Our metric gives only a small error (Fig. 4.12) as it is sensitive to changes in depth, and normal, as well as shading discontinuities.

Overall, our algorithm proves to be very useful for supersampling scenarios, especially, when the shading cost per pixel is high (Tab. 4.2 stalled versions).

### 4.3.7 Discussion and Conclusion

We presented a method to increase performance for supersampling anti-aliasing scenarios in the context of deferred shading. Our results are usually close to the reference, but involve much less computation time. The proposed metrics provide a good tradeoff between performance and accuracy with a controllable error.

In the future, we would like to design light-source aware metrics in order to improve shading near specular highlights, e. g. one could already combine our solution with a light pre-pass [Eng08]. Further, we would like to address thread divergence by using a tile-based approach similar to [Lau10] Also, which is common for AA methods, temporal coherence is an interesting problem and we would like to try compensating for such artifacts by averaging the sample count over several frames, in the spirit of [Her+10].

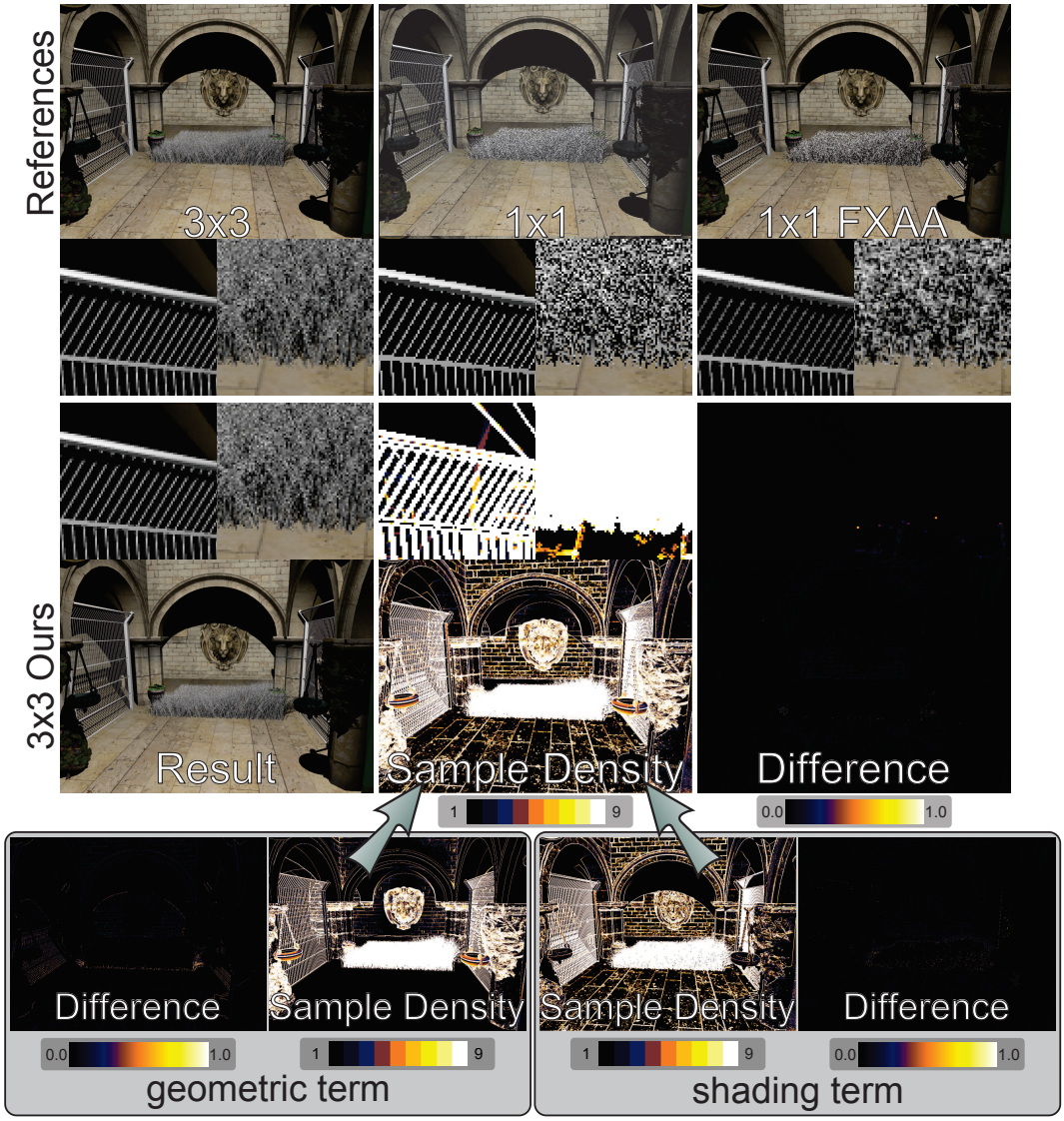



**Figure 4.11:** Importance of both our terms, all difference images were computed with respect to the 3×3 supersampling reference solution. Top row left to right: 3×3 supersampling reference, deferred shading with a single sample, deferred shading with a single sample and FXAA post-processing (high quality settings). Mid row: our adaptive version, (final) sample density of our algorithm, difference image with respect to the 3×3 supersampling reference solution. Bottom row: sample density and difference for our geometric and shading term. Geometric features below the Nyquist frequency are not captured well by our shading term but are reduced using our geometric term. This is even more apparent when the camera is moving.

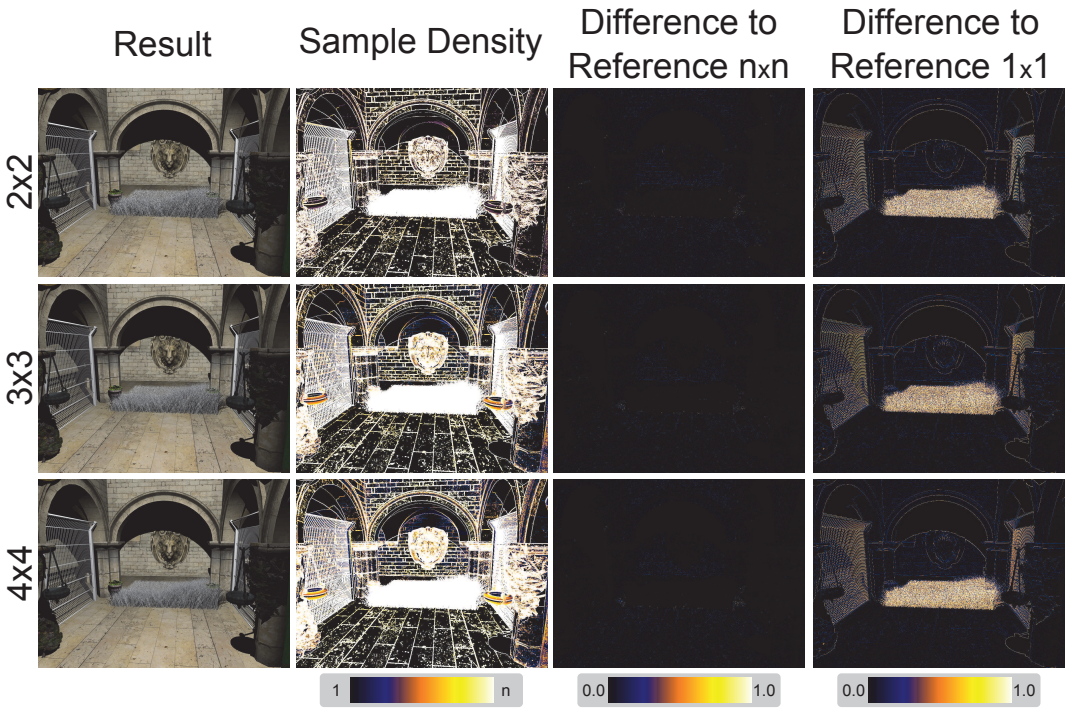



**Figure 4.12:** Our method at different supersampling rates. The third column shows the difference to the reference image at the same sampling rate whereas the last column compares to a $1 \times 1$ casual deferred shading. Both display the absolute error encoded as a heatmap. Our method gives results similar to the reference while being faster to compute.

CHAPTER 5

# CONCLUSION

In this thesis, several methods that help to bridge the gap between quality known from offline rendering and speed of real-time algorithms have been presented. To achieve this goal, all algorithms make heavy use of small-scale GPU kernels that perform geometry processing at their basis. Additionally, each algorithm offers the ability to trade quality for speed which allows for fine-grained control of the overall result, that could be adapted depending on the user's hardware. The general approach used in this thesis is to define suitable lightweight data structures that can be evaluated rapidly and in parallel on the GPU. Moreover, we demonstrated that LoDs can be defined on a per face/mesh, per view or per pixel basis for fast computations. Each technique exploits a different stage of the programmable pipeline (Sec. 1.3) and is ready for a combination with recent compute shaders. In detail, we presented contributions to classical topics in computer graphics all in the context of real-time processing: Subdivision surfaces, Global Illumination and Anti-Aliasing.

**Synthesizing Subdivision Meshes using Real Time Tessellation** We presented a method to create subdivision meshes on-the-fly and in real time suitable for hardware and software tessellation units alike. Our approach is based on a single GPU subdivision step that could be implemented using recent compute shaders, followed by an upsampling step that generates the final mesh at the desired subdivision level by evaluating precomputed tables of basis functions per fine vertex. The basis functions can be subsampled to match adaptive tessellation patterns, thus, allowing finer control over the LoD per-face. The whole process runs on the GPU controlled by a high-level structure – the low-resolution mesh – that drives the synthesis. In practice it implies that existing applications can easily integrate our approach and maintain their low-resolution meshes which are often already available for physics or animation purposes. As demonstrated on a variety of character meshes, our approach is suitable for real-time applications such as computer games but also for modeling software and should be considered as a high quality replacement for subdivision substitutes when possible.

**ManyLoDs: Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination** In this work, we presented a novel approach to compute the multi-cut through a hierarchically organized bounding volume structure for many views in parallel. This proved to be very efficient for speeding up computations for global illumination algorithms that rely on multiple views. A *view* can be understood as a cache entry holding indirect lighting information such as in the case of (ir)radiance caching or instant radiosity. Our parallel multi-cut computation strategy exploits the GPU's computational power by coupling views and nodes to *node-views*. We maintain a todo-list that is initially filled with all root node-views and a finished list that is initially empty. The todo-list is then processed in parallel using several iterations until the final cuts for all views is contained in the finish list. We demonstrated that this cost can be reduced by reusing the result of the previous frame or amortizing the cost of the computation over multiple frames. Further, we presented a hybrid GPU/CPU implementation as well as a specialized implementation that alleviates the high memory requirements by performing all computations in-place.

**Adaptive Supersampling for Deferred Anti-Aliasing** To achieve high quality image generation multiple samples per pixel are a necessity. We presented a new adaptive shading approach based on supersampling in combination with deferred shading. We therefore, first, generate a supersampled intermediate representation of the scene in form of a G-Buffer that is then analyzed using two metrics that account for geometry and appearance. As a result we acquire a buffer containing the required number of samples per pixel that drives the subsequent shading step. Our approach allows to perform the final shading of a pixel adaptively by concentrating the workload where needed. This leads to reduced shading costs and, thus, higher performance especially if the cost of shading a pixel is high.

CHAPTER 6

# PERSPECTIVES/FUTURE WORK

We have presented various approaches for improving performance for real-time applications by using geometry processing, but the journey does not stop here. Before we suggest future directions for further research, we will briefly examine the next generation of graphics hardware which will influence the implementation of our algorithms on more recent hardware, but also inspire new approaches and strategies for solving problems in computer graphics.

## 6.1 Next-Generation Graphics Hardware

**NVIDIA's Kepler** NVIDIA's Kepler series is the successor of Fermi. Besides an increase in pure computational power among other things – the number of processor cores went up from 32 to 192, more registers are available per thread, threads within a warp can exchange data without travelling through shared memory – Kepler allows to use the GPU more efficiently by offering additional (concurrent) hardware work queues that can be fed from different streams, e. g. different CPUs, which significantly improves the overall GPU utilization. On the graphics side, this new series offers display resolutions with more than 8 million pixels, support for up to 4 monitors, adaptive vertical synchronization and improved texture management. Those specifications and the fact that multi-display systems equipped with many CPU cores are a challenging but not exotic configuration, will require further research regarding stream-based rendering as well as out-of-core data management.

More importantly regarding algorithm design, Kepler comes with a new mechanism called *Dynamic Parallelism* that allows to launch kernels directly from within other kernels, thus, avoiding unnecessary communication with the CPU (Fig. 6.1). This could be of interest for designing new GPU accelerated (ir)radiance caching algorithms for on-demand cache creation but also for recursive tessellation. Although untested, this also seems to be an ideal way for implementing our ManyLoD algorithm. During cutfinding, split events could launch new kernel executions and unnecessary CPU iterations could be avoided.

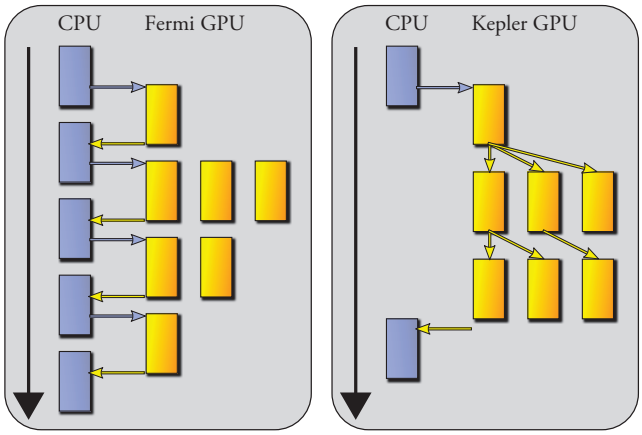



**Figure 6.1:** Dynamic Parallelism in Kepler reduces the amount of CPU/GPU communication. Image adapted from [NVI12].

**Compute Shader** The intent of the compute shader stage is to facilitate general purpose computations on the GPU. It encourages and unifies GPU library development as it often allows effective parallel implementations that were previously only made possible by switching to a GPU compute environment, or by misusage of the graphics pipeline (e. g. by rendering quads in screen-space in combination with a fragment shader).

Especially in the context of deferred shading, we believe the compute shader will dominate implementations of the final shading pass. This is already known from tile-based deferred shading but also from console graphics architectures.

In some cases it might even be more efficient to replace pipeline stages with specialized compute shaders and measure the performance in contrast to the default stages. For instance, customized rasterization stages could benefit this process, e. g. for (massive) point-based rendering where accuracy can be traded for speed. However, $z$-Buffer related implementations face the same hardware limitations as compute environments. Atomic operations on non 32-Bit variables are not allowed, thus, requiring either explicit bit-fiddling when using customized depth tests (Sec. 3.7.2) or binning of primitives. [1]

In general, customized stages could sometimes achieve higher speed than default stages by using additional knowledge about the input data, coarser approximations or simplified representations.

Unfortunately, at the time of writing this, the compute shader is clearly inferior to compute environments as it does not offer low-level specialized hardware commands and forbids synchronizations in nested scopes.

[1] From http://developer.download.nvidia.com/opengl/specs/GL_EXT_shader_image_load_store.txt Issue 6: *"Should we support 64-bit atomics on images? Should we support atomics at all on formats with 8-, 16-, 64-, or 128-bit texels? RESOLVED: No, we will only support 32-bit atomic operations on images."*

## 6.2 Future Directions for Real-Time Geometry Synthesis

We hope that the algorithms presented in this thesis will inspire further investigation of using geometry processing for enabling real-time performance. Also, precomputation, lazy and over-time amortized evaluation as well as fine-grained adaptive schemes in general should be considered when dealing with performance-critical problems. In the following, we will offer directions directly related to the algorithms presented in this thesis.

### Real-time Subdivision

We have seen that precomputation based on basis function tables (BFTs) largely improves performance when dealing with subdivision surfaces. Although the memory consumption of the BFTs is not as critical as for early GPU implementations [Bol+04], we think that the memory footprint could be reduced by compressing the values stored within the tables combined with on-the-fly decompression during evaluation. Special care must be taken regarding symmetry to guarantee crack-free subdivision.

### Future ManyLoD

Our ManyLoD algorithm allows for fast cut computation for multiple views in parallel. The computation of the multi-cut could be improved by dynamically or statically clustering groups of views and computing the cut of such a *grouped*-view before further per-view refinement. The challenge here is to find or select a good representative for the grouped view that minimizes all following per-cut computation. As we have seen in the in-place ManyLoD algorithm using CUDA (Sec. 3.7.2), groups of views can already be treated within a block. Nevertheless, an implementation in a pure shader environment would require additional management of the lists containing the group-cuts.

An implementation of the in-place ManyLoD using the programmable pipeline is currently not possible, due to the above-mentioned limitations for synchronization within a compute shader but also due to missing direct access to the rasterization stage from within a compute shader.

Additionally, we believe that our ManyLoD approach could be useful besides computer graphics. In artificial intelligence simulations, path planning and decision making is often based on per-agent visibility, e. g. for obstacle or enemy avoidance [Rey87]. Such visibility computations could be handled by our many-view system.

Further, physically-based N-Body simulations could be approximated by organizing the interacting entities in a tree and computing a cut for each object. In this case a cut would represent all the forces acting on an object.

For future work, we would also like to investigate subsurface scattering algorithms which could also benefit from view caches. Views could be oriented to face the inside of an object, thus, approximating the thickness of an object instead of inter-object visibility. A similar idea based on ambient occlusion has been presented in [CBB11].

### Anti-Aliasing in Deferred Shading

As we have demonstrated, adaptive evaluation on a per-pixel level can improve rendering times. In the future, we would like to add temporal coherence, in the spirit of [Jim+12], to improve upon quality. In addition, we believe that supersampling methods might be able to handle transparent objects as well by employing screendoor transparency (stippled rendering). Such an evaluation requires special care when selecting the samples from the corresponding per-pixel window.

CHAPTER 7

# ANNEX

## 7.1 Publications

- *M. Holländer and T. Boubekeur*
  **Synthesizing Subdivision Meshes using Real Time Tessellation**
  In: IEEE Pacific Graphics (2010).

- *M. Kasap, M. Holländer, A. Aksay, P. Kelly, D. Monaghan, C. Conaire, N. Magnenat-Thalmann, T. Boubekeur, E. Izquierdo, and N. O'Connor*
  **3D Realistic Animation of a Tennis Player**
  In: Summer School "Engage" (2010).

- *M. Holländer, T. Ritschel, E. Eisemann, and T. Boubekeur*
  **ManyLoDs: Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination**
  In: Computer Graphics Forum (Proceedings EGSR) (2011).

- *J. Huang, T. Boubekeur, T. Ritschel, M. Holländer, and E. Eisemann*
  **Separable Approximation of Ambient Occlusion**
  In: Eurographics (Short Papers) (2011).

- *M. Holländer, T. Boubekeur, and E. Eisemann*
  **Adaptive Supersampling for Deferred Anti-Aliasing**
  In: Journal of Computer Graphics Techniques (Accepted for publication).

## 7.2 Real-Time Ambient Occlusion

First of all, we would like to stress the fact that the first of author of the publication summarized in this section is Jing Huang [Hua+11].

Ambient Occlusion (AO) is a popular and wide-spread means of approximating GI. It helps in improving the perception of volumes, concave regions and contact areas of objects in the scene. We will briefly review the required background regarding AO and then present our separable approximation to screenspace AO, which splits the 2D evaluation into two 1D kernels with subsequent stochastic evaluation. For further reading, we delegate the interested reader to the survey by Méndez-Feliu and Sbert [MF+09].

### 7.2.1 Ambient Occlusion

AO considers only diffuse surfaces and defines the ambient lighting affecting a point $\mathbf{p}$ as

$$A(\mathbf{p}) = \frac{1}{\pi} \int_{\Omega_+} V(\mathbf{p}, \omega) \cos(\mathbf{n}_\mathbf{p}, \omega) \ \mathrm{d}\omega,$$

where $V(\mathbf{p}, \omega)$ is the visibility function which returns $0$ if the ray from direction $\omega$ is blocked and $1$ otherwise. To restrict the evaluation to close occluders, $V$ can be coupled to a distance-based falloff function. As described in section 3.3, the integral over the hemisphere $\Omega_+$ can be evaluated using Monte Carlo Integration, however, this is usually too time-consuming for real-time applications. Instead, it can be approximated in screenspace based on the idea that the camera's depth buffer values around each pixel give an approximation of the scene in the vicinity of the pixel's world location. To evaluate the AO term, random 3D samples around the pixel are projected into the depth buffer with a subsequent comparison to the stored value [Mit07]. This screen-space approximation can be understood as a local filter operating in 2D screenspace on a pixel $\{i, j\}$ with corresponding world location $\mathbf{p}_{i,j}$ and normal $\mathbf{n}_{i,j}$:

$$A_{\text{SSAO}}(i, j) = \frac{1}{k^2} \sum_{x=i-k/2}^{i+k/2} \sum_{y=j-k/2}^{j+k/2} V(\mathbf{p}_{i,j}, \omega_{x,y}) \cos(\mathbf{n}_{i,j}, \omega_{x,y}), \tag{7.1}$$

where $\omega$ is a sample point on the hemisphere centered around the pixel's world location $\mathbf{p}_{i,j}$ and $k$ a user-provided kernel size. Screen Space Ambient Occlusion (SSAO) has the advantage of being scene independent and scales linearly to the amount of pixels processed.

### 7.2.2 Separable Approximation of Ambient Occlusion

Although equation 7.1 is not formally separable, it can be approximated using two separate 1D filters ($A_x$ and $A_y$) similar to bilateral image filtering [Pha+05]. Consequently, each 1D filter will be evaluated in 1D only, restricting the sample directions $\omega$ to a unit half circle. This can be achieved in two rendering passes,

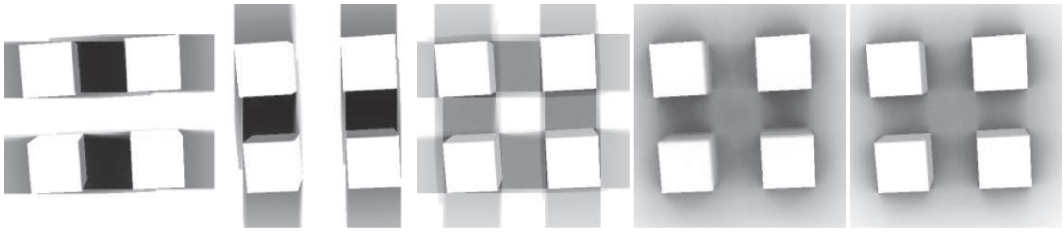



**Figure 7.1:** The artifacts of a simple averaging of $A_x$ and $A_y$ can be improved significantly by using local frames that are randomized per pixel.

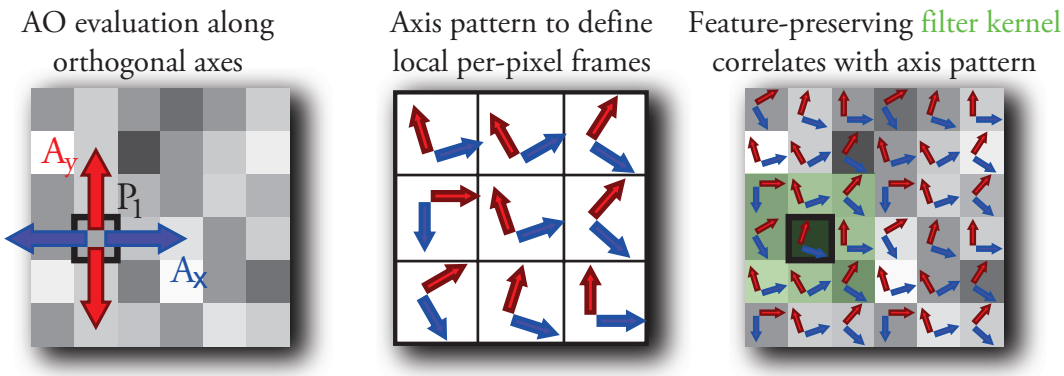



**Figure 7.2:** Principle: our separable approximation combines two 1D evaluations in orthogonal screen directions, e. g. $x$-,$y$ axis, left. By changing coordinate frames per pixel, we can derive a stochastically valid approximation of the 2D occlusion by combining the result of neighbouring samples.

each fixing either $x$ or $y$ to 0. Averaging the results of both passes gives the most basic form of such a separable approximation but neglects occluders in diagonal directions (Fig. 7.1). To fix this, we replace the global $\{x, y\}$ frame by a local frame that is randomized (Fig. 7.2). This can be achieved using interleaved sampling [Kel+01], where the noise pattern size matches our kernel size. Finally, we use feature preserving smoothing [Bav+08; Rei+09] to attenuate SSAO artifacts.

### 7.2.3 Results

We have implemented our algorithm using OpenGL/GLSL and tested the performance on a GeForce GTX 480 with 1536 MB VRAM in combination with an Intel Core i7 2.67 GHz. All results were measured at a resolution of $1024 \times 768$ for well-known SSAO techniques (Tab. 7.1) together with a perceptual difference metric in Lab colour space [Yee04] (Fig. 7.3).

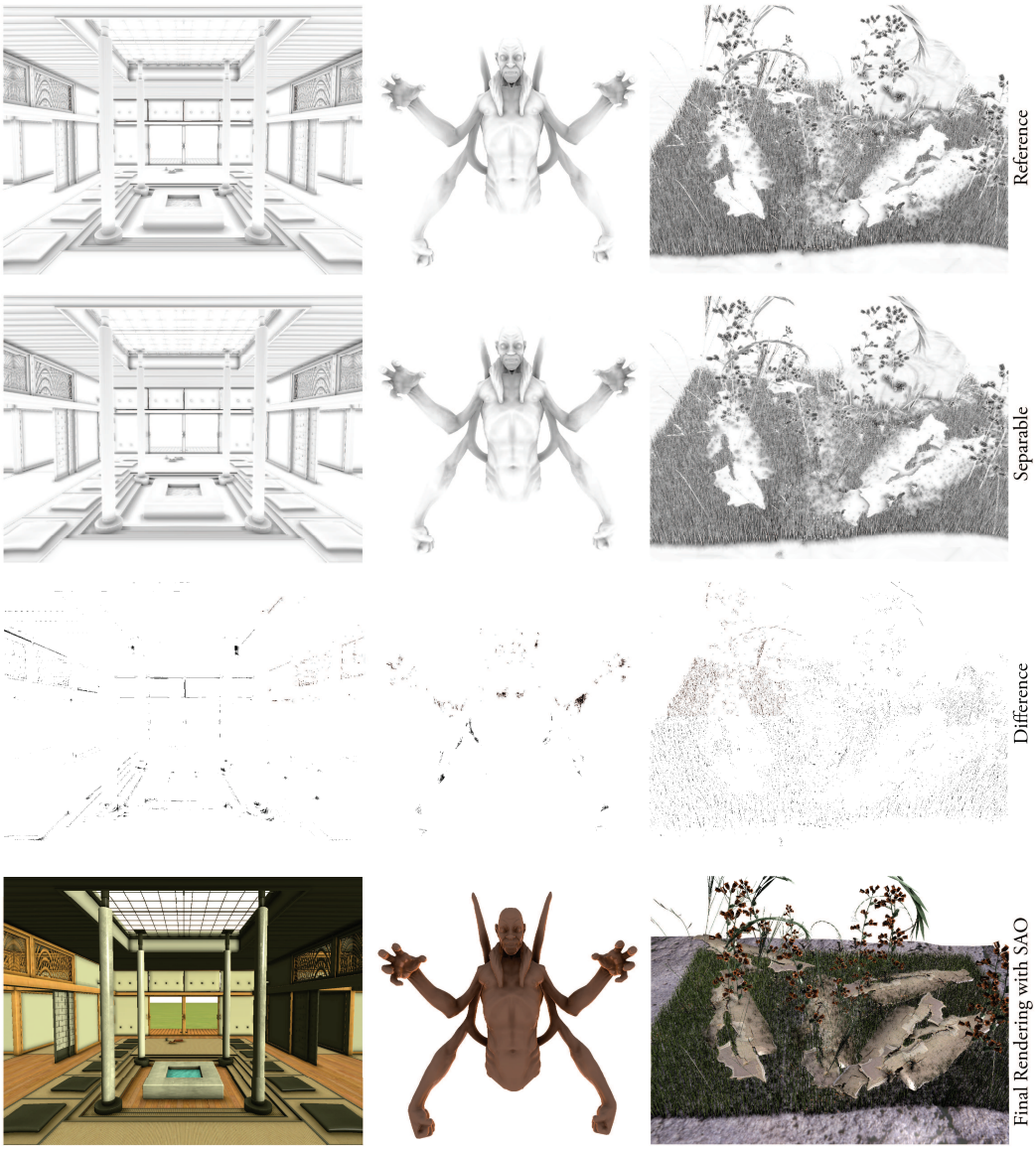



**Figure 7.3:** Comparison between reference (first row) and separable AO evaluation (second row). Additionally, we computed the perceptual difference (third row) based on [Yee04] for each image pair. White values stand for no difference while black values indicate visible shifts.

### 7.2.4 Conclusion

We have presented a fast separable approximation to screenspace ambient occlusion by separating the kernel into two 1D kernels coupled with per-pixel random local frames. Our approach significantly improves computation times by reducing the complexity from $\mathcal{O}(k^2)$ to $\mathcal{O}(k)$, can extend existing algorithms and does not reduce the visual quality significantly.

| Size | Samples | Separable | Crytek [Mit07] | Vol. Obs. [Loo+10] | HBAO [Bav+08] |
|---|---|---|---|---|---|
| 5 | $5 \times 5$ | ✗ | 3.2 ms | 3.5 ms | 3.6 ms |
| | $5 \times 2$ | ✓ | 3.4 ms | 3.6 ms | 3.5 ms |
| 11 | $11 \times 11$ | ✗ | 13.9 ms | 14.8 ms | 15.0 ms |
| | $11 \times 2$ | ✓ | 5.8 ms | 5.9 ms | 6.0 ms |
| 21 | $21 \times 21$ | ✗ | 49.7 ms | 51.9 ms | 51.9 ms |
| | $21 \times 2$ | ✓ | 9.9 ms | 9.9 ms | 10.2 ms |

**Table 7.1:** Performance of our separable version compared to non-separable versions evaluated for various methods and different filter sizes.

## 7.3 IEEE-754 Floating Point Numbers

All IEEE-754 floating point numbers in the $[0.5, 1[$ range have an exponent of 126 (Lst. 7.1). However, 1.0 has an exponent of 127.

## 7.4 Real-Time Subdivision Surfaces on Fermi using Instancing

The results in figure 7.4 where obtained using the same configuration as described in section 2.3 – an Intel Core i7 2.67 GHz using OpenGL under Windows. However, this time we replaced the graphics card with an NVIDIA GTX 480 from the Fermi series and measured the performance, again, using a hardware tessellation emulator based on [Bou+08a] for fair comparisons. By simply updating the graphics card we observed a performance boost of a factor of 5 compared to the previous generation, i. e. the MonsterFrog mesh was upsampled to level 5 and rendered in 7.69 ms instead of 41.6 ms (compare to Tab. 2.1), while still not harnessing the hardware tessellation unit's power.

## 7.5 Acknowledgements

The BigGuy and MonsterFrog meshes are taken from the DirectX SDK. Original meshes by Bay Raitt. Imrod mesh developed by Dmitry Parkin. Demon mesh by Nick Zuccarello. Sponza mesh adapted from Crytek, original mesh by Marko Dabrovic.

```
#include <stdio.h>
#include <iostream>
#include <cuda_runtime.h>
typedef unsigned int uint32_t;
union FloatUint {
   float          m_Float;
   uint32_t       m_Uint;
   struct {
      uint32_t    Mantissa : 23;    ///< Mantissa has bits 0..22
      uint32_t    Exponent :  8;    ///< Exponent has bits 23..30
      uint32_t    Sign     :  1;    ///< Sign has bit 31
   } Format;
   __device__ FloatUint( float f ) : m_Float( f ) {}
};

__device__ void printFloat( FloatUint f ) {
   char szBits[ 33 ] = { 0 };
   for( int i=0; i<32; ++i ) {
      szBits[ 31 - i ] = ( f.m_Uint & ( 1 << i ) ? '1' : '0' );
   }
   printf( "%.8f %s s: %d Exp: %d Man: %d\n",
          f.m_Float, szBits,
          f.Format.Sign, f.Format.Exponent, f.Format.Mantissa );
}

__global__ void FloatingPointTests() {
   printFloat( 0.99999994f );
   printFloat( 0.5f );
   printFloat( 1.f );
}

int main( int iArgc, char* szArgs[] ) {
   FloatingPointTests<<< 1, 1 >>>();
   cudaDeviceSynchronize();
   return EXIT_SUCCESS;
}

> 0.99999994 00111111011111111111111111111111 s: 0 Exp: 126 Man: 8388607
> 0.50000000 00111111000000000000000000000000 s: 0 Exp: 126 Man: 0
> 1.00000000 00111111100000000000000000000000 s: 0 Exp: 127 Man: 0
```

**Listing 7.1:** IEEE-754 floating point representations.

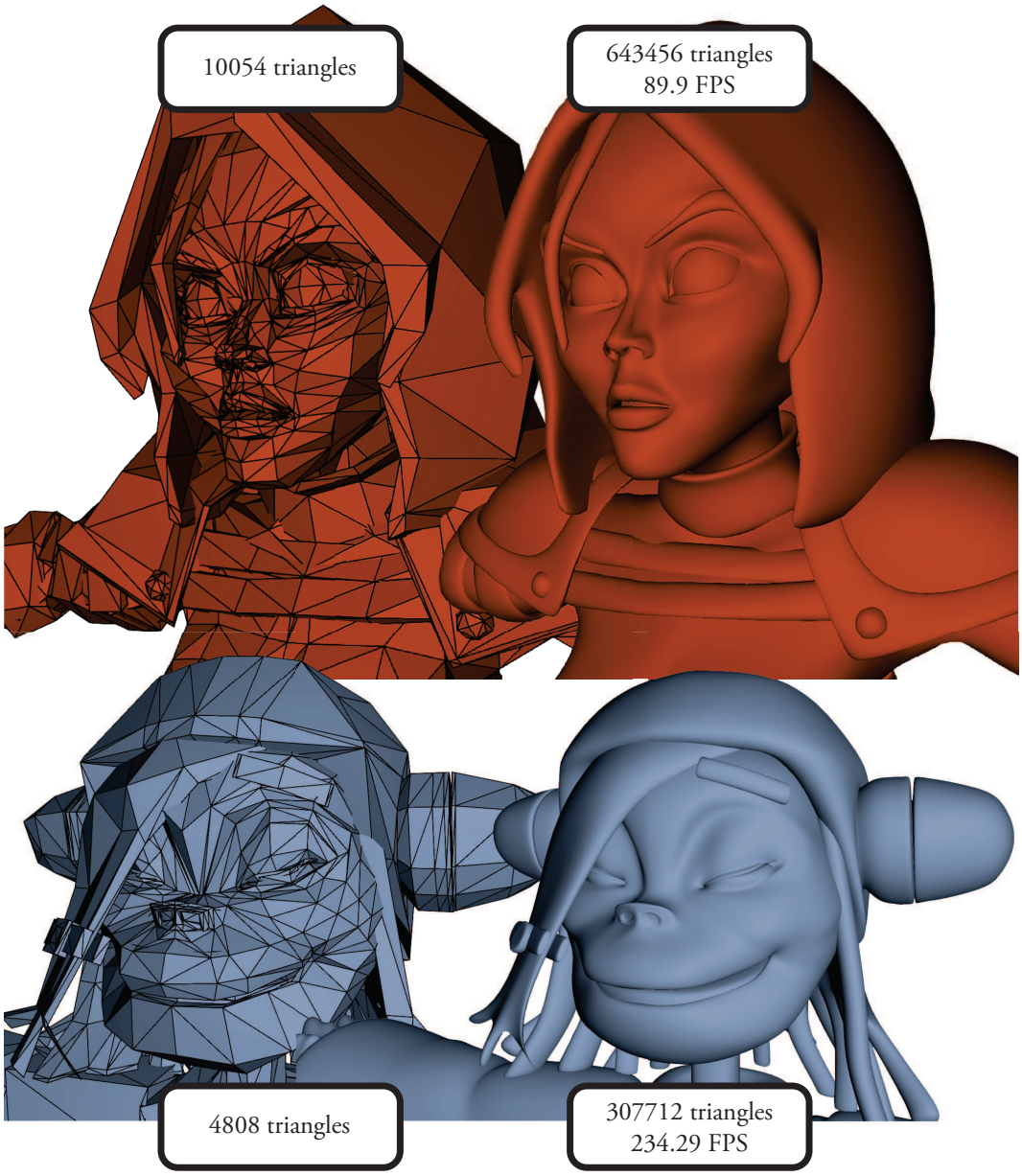



**Figure 7.4:** Dynamically synthesized subdivision surfaces using instancing on Fermi (GTX 480).

# BIBLIOGRAPHY


[Ail+09] T. Aila and S. Laine. "Understanding the efficiency of ray traversal on GPUs". In: *Proc. High Performance Graphics*. 2009, pp. 145–149.

[Ail+10] T. Aila and T. Karras. "Architecture considerations for tracing incoherent rays". In: *Proc. High Performance Graphics*. 2010, pp. 113–122.

[Ail+12] T. Aila, S. Laine, and T. Karras. *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum*. NVIDIA Technical Report NVR-2012-02. NVIDIA Corporation, 2012.

[Ale+08] M. Alexa and T. Boubekeur. "Subdivision Shading". In: *ACM ToG (Proc. SIGGRAPH Asia)* 27.5 (2008), 142:1–142:4.

[And11] D. Andreev. *DLAA*. http : / / and . intercon . ru / releases / talks / dlaagdc2011/. 2011.

[Aug+11] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures". In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198.

[Bag+12] M. Bagher M., C. Soler, K. Subr, L. Belcour, and N. Holzschuch. "Interactive rendering of acquired materials on dynamic geometry using bandwidth prediction". In: *Proc. Symposium on Interactive 3D Graphics and Games (I3D)*. 2012, pp. 127–134.

[Bau+11] P. Bauszat, M. Eisemann, and M. Magnor. "Guided Image Filtering for Interactive High-quality Global Illumination". In: *Computer Graphics Forum*. Vol. 30. 4. 2011, pp. 1361–1368.

[Bav+08] L. Bavoil, M. Sainz, and R. Dimitrov. "Image-space Horizon-based Ambient Occlusion". In: *ACM SIGGRAPH Talks*. 2008, 22:1–22:1.

[Bax11] S. Baxter. *Modern GPU – Scan*. 2011. URL: http://www.moderngpu.com/intro/scan.html.

[Bay73] B. Bayer. "An optimum method for two-level rendition of continuous-tone pictures". In: *IEEE International Conference on Communications* 1 (1973), pp. 11–15.

[Ben75] J. Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[Ble89] G. Blelloch. "Scans as Primitive Parallel Operations". In: *IEEE Transactions on Computers* 38 (11 1989), pp. 1526–1538.

[Ble93] G. Blelloch. "Prefix sums and their applications". In: *Synthesis of Parallel Algorithms* (1993), pp. 35–60.

[Bli+76] J. Blinn and M. Newell. “Texture and reflection in computer generated images”. In: *Communications of the ACM* 19.10 (1976), pp. 542–547.

[Bol+02] J. Bolz and P. Schröder. “Rapid evaluation of Catmull-Clark subdivision surfaces”. In: *ACM Web3D*. 2002, pp. 11–17.

[Bol+04] J. Bolz and P. Schröder. *Evaluation of Subdivision Surfaces on Programmable Graphics Hardware*. 2004.

[Bor+08] M. Boris, A. Seidel, and H. Lensch. “Direct Visualization of Real-World Light Transport”. In: *Vision, modeling, and visualization* (2008), p. 363.

[Bot+06] M. Botsch, M. Pauly, C. Rossl, S. Bischoff, and L. Kobbelt. “Geometric modeling based on triangle meshes”. In: *ACM SIGGRAPH Courses*. 2006, p. 1.

[Bot+10] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, and B. Lévy. *Polygon mesh processing*. AK Peters Limited, 2010.

[Bou+05a] T. Boubekeur and C. Schlick. “Generic mesh refinement on GPU”. In: *Proc. ACM SIGGRAPH/Eurographics conference on Graphics hardware*. 2005, pp. 99–104.

[Bou+05b] T. Boubekeur, P. Reuter, and C. Schlick. “Surfel Stripping”. In: *ACM Graphite*. 2005.

[Bou+07a] T. Boubekeur and C. Schlick. “QAS: Real-time quadratic approximation of subdivision surfaces”. In: *Proc. IEEE Pacific Graphics*. 2007, pp. 453–456.

[Bou+07b] S. Boulos, D. Edwards, J. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. “Packet-based whitted and distribution ray tracing”. In: *Proc. Graphics Interface*. Vol. 1. 2007.

[Bou+08a] T. Boubekeur and C. Schlick. “A flexible kernel for adaptive mesh refinement on GPU”. In: *Computer Graphics Forum*. Vol. 27. 2008, pp. 102–113.

[Bou+08b] T. Boubekeur and M. Alexa. “Phong Tessellation”. In: *ACM ToG (Proc. SIGGRAPH Asia)* 27.5 (2008), pp. 1–5.

[Bou10] T. Boubekeur. “A View-Dependent Adaptivity Metric for Real-Time Mesh Tessellation”. In: *IEEE International Conference on Image Processing*. 2010.

[Buc+12] B. Buchholz and T. Boubekeur. “Quantized Point-Based Global Illumination”. In: *Computer Graphics Forum (Special Issue: EGSR)* (2012).

[Bun05] M. Bunnell. “Dynamic Ambient Occlusion and Indirect Lighting”. In: *GPU Gems 2* (2005), pp. 223–233.

[Car+01] M. Carlo, R. Tracing, J. Arvo, P. Hanrahan, H. Jensen, D. Mitchell, M. Pharr, P. Shirley, J. Arvo, and M. Fajardo. “State of the Art in Monte Carlo Ray Tracing for Realistic Image Synthesis”. In: *SIGGRAPH Course Notes*. 2001.

[Car+02] N. Carr, J. Hall, and J. Hart. "The ray engine". In: *SIGGRAPH/Eurographics Workshop on Graphics Hardware*. 2002, pp. 37–46.

[Car+03] N. Carr, J. Hall, and J. Hart. "GPU algorithms for radiosity and subsurface scattering". In: *Proc. ACM SIGGRAPH/Eurographics conference on Graphics hardware*. 2003, pp. 51–59.

[Cat+78] E. Catmull and J. Clark. "Recursively generated B-spline surfaces on arbitrary topological meshes". In: *Computer Aided Design* 10.6 (1978), pp. 350–355.

[CBB11] M. B. Colin Barré-Brisebois. *Approximating Translucency for a Fast, Cheap and Convincing Subsurface Scattering Look*. Game Developers Conference. 2011. URL: http://dice.se/wp-content/uploads/Colin_BarreBrisebois_Programming_ApproximatingTranslucency.pdf.

[Cha+11] M. Chajdas, M. McGuire, and D. Luebke. "Subpixel reconstruction antialiasing for deferred shading". In: *Proc. Symposium on Interactive 3D Graphics and Games (I3D)*. 2011, PAGE–7.

[Chr08] P. Christensen. "Point-based approximate color bleeding". In: *Pixar Technical Notes* 2.5 (2008), p. 6.

[Cra+11] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann. "Interactive indirect illumination using voxel cone tracing". In: *Computer Graphics Forum*. Vol. 30. 7. 2011, pp. 1921–1930.

[Dac+03] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. "Sequential point trees". In: *ACM Transactions on Graphics (TOG)* 22.3 (2003), pp. 657–662.

[Dac+05] C. Dachsbacher and M. Stamminger. "Reflective shadow maps". In: *Proc. Symposium on Interactive 3D Graphics and Games (I3D)*. 2005, pp. 203–231.

[Dac+07] C. Dachsbacher, M. Stamminger, G. Drettakis, and F. Durand. "Implicit visibility and antiradiance for interactive global illumination". In: *ACM Transactions on Graphics (TOG)* 26.3 (2007), p. 61.

[Dee+88] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt. "The triangle processor and normal vector shader: a VLSI system for high performance graphics". In: *ACM SIGGRAPH Computer Graphics* 22.4 (1988), pp. 21–30.

[DeR+98] T. DeRose, M. Kass, and T. Truong. "Subdivision surfaces in character animation". In: *SIGGRAPH*. Vol. 98. 1998, pp. 85–94.

[Doo+78] D. Doo and M. Sabin. "Analysis of the behaviour of recursive division surfaces near extraordinary points". In: *Computer Aided Design* 10.6 (1978), pp. 356–360.

[Dot+08] Y. Dotsenko, N. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli. "Fast scan algorithms on graphics processors". In: *Proc. International Conference on Supercomputing*. 2008, pp. 205–213.

[Dyk+04] C. Dyken and M. Reimers. “Real-time linear silhouette enhancement”. In: *Proc. Mathematical Methods for Curves and Surfaces* (2004), pp. 135–143.

[Dyn+90] N. Dyn, D. Levine, and J. Gregory. “A butterfly subdivision scheme for surface interpolation with tension control”. In: *ACM transactions on Graphics (TOG)* 9.2 (1990), pp. 160–169.

[Eis+09] C. Eisenacher, Q. Meyer, and C. Loop. “Real-time view-dependent rendering of parametric surfaces”. In: *Proc. Symposium on Interactive 3D Graphics and Games (I3D)*. 2009, pp. 137–143.

[Eng08] W. Engel. *Diary of a Graphics Programmer: Light Pre-Pass Renderer*. http://diaryofagraphicsprogrammer.blogspot.com/2008/03/light-pre-pass-renderer.html. 2008.

[Fis+09] M. Fisher, K. Fatahalian, S. Boulos, K. Akeley, W. Mark, and P. Hanrahan. “DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering”. In: *ACM Transactions on Graphics (TOG)*. Vol. 28. 5. 2009, p. 150.

[Fuc+80] H. Fuchs, Z. Kedes, and B. Naylor. “On Visible Surface Generation by A Priori Tree Structures”. In: *ACM SIGGRAPH*. 1980.

[Gau+04] P. Gautron, J. Krivanek, S. Pattanaik, K. Bouatouch, et al. “A novel hemispherical basis for accurate and efficient rendering”. In: *Rendering techniques, Eurographics symposium on rendering*. 2004, pp. 321–330.

[Gee10] K. Gee. *Direct3D 11 Overview*. http://developer.download.nvidia.com/presentations/2008/NVISION/NVISION08_Direct3D_11_Overview.pdf. 2010.

[Gob+05] E. Gobbetti and F. Marton. “Far voxels: a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms”. In: *ACM Transactions on Graphics* 24.3 (2005), pp. 878–885.

[Gor+10] R. Goradia, S. Kashyap, P. Chaudhuri, and S. Chandran. “GPU-Based Ray Tracing of Splats”. In: *Computer Graphics and Applications (PG)*. 2010, pp. 101–108.

[Gor+84] C. Goral, K. Torrance, D. Greenberg, and B. Battaile. “Modeling the interaction of light between diffuse surfaces”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 18. 3. 1984, pp. 213–222.

[Gre+98] G. Greger, P. Shirley, P. Hubbard, and D. Greenberg. “The irradiance volume”. In: *IEEE Computer Graphics and Applications* 18.2 (1998), pp. 32–43.

[Gre96] G. Greger. “The Irradiance Volume”. MA thesis. Citeseer, 1996.

[Gru12] H. Gruen. “GPU Pro 3: Advanced Rendering Techniques”. In: GPU Pro series. CRC Press, 2012. Chap. Vertex Shader Tessellation.

[Ha07] M. Hašan, F. Pellacini, and K. Bala. "Matrix row-column sampling for the many-light problem". In: *ACM Transactions on Graphics* 26.3 (2007), p. 26.

[Hac+10] T. Hachisuka and H. Jensen. "Parallel progressive photon mapping on GPUs". In: *ACM SIGGRAPH ASIA Sketches*. 2010, p. 54.

[Han+91] P. Hanrahan, D. Salzman, and L. Aupperle. "A rapid hierarchical radiosity algorithm". In: *ACM SIGGRAPH Computer Graphics*. Vol. 25. 4. 1991, pp. 197–206.

[Har+04] S. Hargreaves and M. Harris. "Deferred shading". In: *Game Developers Conference, D3D Tutorial Day*. 2004.

[Har+07] M. Harris, S. Sengupta, and J. Owens. "Parallel prefix sum (scan) with CUDA". In: *GPU Gems* 3.39 (2007), pp. 851–876.

[Haš+08] M. Hašan, E. Velázquez-Armendáriz, F. Pellacini, and K. Bala. "Tensor clustering for rendering many-light animations". In: *Computer Graphics Forum*. Vol. 27. 4. 2008, pp. 1105–1114.

[He+10] L. He, S. Schaefer, and K. Hormann. "Parameterizing subdivision surfaces". In: *ACM Transactions on Graphics (TOG)* 29.4 (2010), p. 120.

[Hec89] P. Heckbert. "Fundamentals of texture mapping and image warping". MA thesis. Citeseer, 1989.

[Hec93] P. Heckbert. *Introduction to Finite Element Methods*. Global Illumination Course Notes Siggraph 93. Available online at http://www.cs.cmu.edu/afs/cs/user/ph/www/heckbert.html, 1993.

[Her+10] R. Herzog, E. Eisemann, K. Myszkowski, and H.-P. Seidelm. "Spatio-Temporal Upsampling on the GPU". In: *Proc. Symposium on Interactive 3D Graphics and Games (I3D)*. 2010.

[Hob+09] J. Hoberock, V. Lu, Y. Jia, and J. Hart. "Stream compaction for deferred shading". In: *Proc. High Performance Graphics*. 2009, pp. 173–180.

[Hop96] H. Hoppe. "Progressive Meshes". In: *Computer Graphics* 30.Annual Conference Series (1996), pp. 99–108.

[Hu+09] L. Hu, P. Sander, and H. Hoppe. "Parallel view-dependent refinement of progressive meshes". In: *Proc. Symposium on Interactive 3D Graphics and Games (I3D)*. 2009, pp. 169–176.

[Hua+11] J. Huang, T. Boubekeur, T. Ritschel, M. Holländer, and E. Eisemann. "Separable Approximation of Ambient Occlusion". In: *Eurographics (Short Papers)* (2011).

[Hub93] P. Hubbart. "Interactive Collision Detection". In: *Proc. IEEE Symp. on Research Frontier in Virtual Reality*. 1993.

[Hwu11] W. Hwu. *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., 2011.

[Jac+80] C. Jackins and S. Tanimoto. "Oct-trees and their use in representing three-dimensional objects". In: *CGIP* 14 (1980), pp. 249–270.

[Jen+01] H. Jensen, S. Marschner, M. Levoy, and P. Hanrahan. "A practical model for subsurface light transport". In: *Proc. Computer graphics and interactive techniques*. 2001, pp. 511–518.

[Jen96] H. Jensen. "Global illumination using photon maps". In: *Rendering Techniques* 96 (1996), pp. 21–30.

[Jim+11] J. Jimenez, D. Gutierrez, J. Yang, A. Reshetov, P. Demoreuille, T. Berghoff, C. Perthuis, H. Yu, M. McGuire, T. Lottes, et al. "Filtering approaches for real-time anti-aliasing". In: *ACM SIGGRAPH Courses* (2011).

[Jim+12] J. Jimenez, J. Echevarria, T. Sousa, and D. Gutierrez. "SMAA: Enhanced Subpixel Morphological Antialiasing". In: *Computer Graphics Forum*. Vol. 31. 2. 2012, pp. 355–364.

[Jin+05] S. Jin, R. Lewis, and D. West. "A comparison of algorithms for vertex normal computation". In: *The Visual Computer* 21.1 (2005), pp. 71–82.

[Kaj86] J. Kajiya. "The rendering equation". In: *ACM SIGGRAPH Computer Graphics* 20.4 (1986), pp. 143–150.

[Kap+10] A. Kaplanyan and C. Dachsbacher. "Cascaded light propagation volumes for real-time indirect illumination". In: *Proc. Symposium on Interactive 3D Graphics and Games (I3D)*. 2010, pp. 99–107.

[Kel+01] A. Keller and W. Heidrich. "Interleaved Sampling". In: *12th Eurographics Workshop on Rendering*. 2001, pp. 269–276.

[Kel97] A. Keller. "Instant radiosity". In: *SIGGRAPH: Proc. Computer graphics and interactive techniques*. 1997, pp. 49–56.

[Ki10] H. Ki. "Game Programming Gems 8: Multi-Resolution Deferred Shading". In: *Cengage Learning Emea* 8 (2010), pp. 32–38.

[Kir+09] S. Kircher and A. Lawrance. "Inferred lighting: fast dynamic lighting and shadows for opaque and translucent objects". In: *Proc. ACM SIGGRAPH 2009 Symposium on Video Games*. ACM. 2009, pp. 39–45.

[Kri+05] J. Krivanek, P. Gautron, S. Pattanaik, and K. Bouatouch. "Radiance caching for efficient global illumination computation". In: *IEEE Transactions on Visualization and Computer Graphics* 11.5 (2005), pp. 550–561.

[Lai+07] S. Laine, H. Saransaari, J. Kontkanen, J. Lehtinen, and T. Aila. "Incremental Instant Radiosity for Real-Time Indirect Illumination". In: *Proc. Eurographics Symposium on Rendering*. 2007, pp. 277–286.

[Lau+09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. "Fast BVH construction on GPUs". In: *Computer Graphics Forum*. Vol. 28. 2. 2009, pp. 375–384.

[Lau10] A. Lauritzen. "Deferred rendering for current and future rendering pipelines". In: *SIGGRAPH Course: Beyond Programmable Shading* (2010).

[Lev+00] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. "The Digital Michelangelo Project : 3D Scanning of Large Statues". In: *Proc. ACM SIGGRAPH*. 2000, pp. 131–144.

[Lik+12] G. Liktor and C. Dachsbacher. "Decoupled deferred shading for hardware rasterization". In: *Proc. Symposium on Interactive 3D Graphics and Games (I3D)*. 2012, pp. 143–150.

[Lis+93] D. Lischinski, F. Tampieri, and D. Greenberg. "Combining hierarchical radiosity and discontinuity meshing". In: *Proc. Computer graphics and interactive techniques*. 1993, pp. 199–208.

[Liu+10] F. Liu, M. Huang, X. Liu, and E. Wu. "FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects". In: *Proc. Symposium on Interactive 3D Graphics and Games (I3D)*. 2010, pp. 75–82.

[Loo+08] C. Loop and S. Schaefer. "Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches". In: *ACM Transactions on Graphics* 27.1 (2008), pp. 1–11.

[Loo+09] C. Loop, S. Schaefer, T. Ni, and I. Castaño. "Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation". In: *ACM Transactions on Graphics* 28.5 (2009), pp. 1–9.

[Loo+10] B. J. Loos and P.-P. Sloan. "Volumetric Obscurance". In: *Proc. Symposium on Interactive 3D Graphics and Games (I3D)*. 2010, pp. 151–156.

[Loo87] C. Loop. "Smooth subdivision surfaces based on triangles". In: *Master's thesis, University of Utah, Department of Mathematics* (1987).

[Lot09] T. Lottes. *FXAA*. http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf. 2009.

[Lue+02] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2002.

[Ma+02] V. Ma and M. McCool. "Low latency photon mapping using block hashing". In: *Proc. ACM SIGGRAPH/Eurographics conference on Graphics hardware*. 2002, pp. 89–99.

[Man+07] E. Mansson, J. Munkberg, and T. Akenine-Moller. "Deep coherent ray tracing". In: *IEEE Symposium on Interactive Ray Tracing*. 2007, pp. 79–85.

[McC+99] P. McCormick, C. Hansen, and E. Angel. "The deferred accumulation buffer". In: *Journal of Graphics Tools* 4.3 (1999), pp. 35–46.

[Mey+09] Q. Meyer, C. Eisenacher, M. Stamminger, and C. Dachsbacher. "Data-Parallel Hierarchical Link Creation for Radiosity". In: *Eurographics Symposium on Parallel Graphics and Visualization*. 2009, pp. 65–70.

[MF+09] À. Méndez-Feliu and M. Sbert. "From obscurances to ambient occlusion: A survey". In: *The Visual Computer* 25.2 (2009), pp. 181–196.

[Mit+04] J. Mitchell and P. Sander. "Applications of explicit early-Z culling". In: *Real-Time Shading Course, SIGGRAPH* (2004).

[Mit07] M. Mittring. "Finding next gen: CryEngine 2". In: *ACM SIGGRAPH Courses*. 2007, pp. 97–121.

[Mun+08] J. Munkberg, J. Hasselgren, and T. Akenine-Möller. "Non-uniform fractional tessellation". In: *Proc. ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. 2008, pp. 41–45.

[Ni+09] T. Ni, I. Castaño, J. Peters, J. Mitchell, P. Schneider, and V. Verma. "Efficient substitutes for subdivision surfaces". In: *ACM SIGGRAPH Courses*. 2009, pp. 1–107.

[Nie+01] K. Nielsen and N. Christensen. "Fast texture-based form factor calculations for radiosity using graphics hardware". In: *Journal of Graphics Tools* 6.4 (2001), pp. 1–12.

[Nie+12] M. Nießner, C. Loop, M. Meyer, and T. Derose. "Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces". In: *ACM Transactions on Graphics (TOG)* 31.1 (2012), p. 6.

[NVI12] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture*. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. 2012.

[Pap11] G. Papaioannou. "Real-time diffuse global illumination using radiance hints". In: *Proc. ACM SIGGRAPH Symposium on High Performance Graphics, HPG*. Vol. 11. 2011, pp. 15–24.

[Par+10] S. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, et al. "Optix: A general purpose ray tracing engine". In: *ACM Transactions on Graphics (TOG)* 29.4 (2010), p. 66.

[Pfi+00] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. "Surfels: Surface elements as rendering primitives". In: *SIGGRAPH*. 2000, pp. 335–342.

[Pha+05] T. Pham and L. Van Vliet. "Separable bilateral filtering for fast video preprocessing". In: *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*. 2005, pp. 1–4.

[Pha+10] M. Pharr and G. Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2010.

[Pho73] B. Phong. "Illumination of Computer-Generated Images". PhD thesis. University of Utah, 1973.

[Pix04] Pixologic. *Rendering ZBrush Displacement-Maps*. 2004. URL: http://www.zbrushcentral.com.

[Pru+12] R. Prutkin, A. Kaplanyan, and C. Dachsbacher. "Reflective Shadow Map Clustering for Real-Time Global Illumination". In: *Eurographics-Short Papers*. 2012, pp. 9–12.

[Pur+03] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan. "Photon mapping on programmable graphics hardware". In: *Proc. ACM SIGGRAPH/Eurographics conference on Graphics hardware*. 2003, pp. 41–50.

[Rei+09] C. Reinbothe, T. Boubekeur, and M. Alexa. "Hybrid Ambient Occlusion". In: *EUROGRAPHICS '09 Areas Papers* (2009).

[Res09] A. Reshetov. "Morphological antialiasing". In: *Proc. High Performance Graphics*. 2009, pp. 109–116.

[Rey87] C. Reynolds. "Flocks, herds and schools: A distributed behavioral model". In: *Proc. Computer graphics and interactive techniques*. 1987, pp. 25–34.

[Rit+08] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz. "Imperfect shadow maps for efficient computation of indirect illumination". In: *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)* (2008), pp. 1–8.

[Rit+09] T. Ritschel, T. Engelhardt, T. Grosch, H. Seidel, J. Kautz, and C. Dachsbacher. "Micro-rendering for scalable, parallel final gathering". In: *ACM SIGGRAPH Asia 2009 papers*. 2009, pp. 1–8.

[Rit+11] T. Ritschel, E. Eisemann, I. Ha, J. Kim, and H. Seidel. "Making Imperfect Shadow Maps View-Adaptive: High-Quality Global Illumination in Large Dynamic Scenes". In: *Computer Graphics Forum*. Vol. 30. 8. 2011, pp. 2258–2269.

[Rit+12] T. Ritschel, C. Dachsbacher, T. Grosch, and J. Kautz. "The State of the Art in Interactive Global Illumination". In: *Computer Graphics Forum* 31.1 (2012), pp. 160–188.

[Rus+00] S. Rusinkiewicz and M. Levoy. "QSplat: A multiresolution point rendering system for large meshes". In: *Proc. Computer graphics and interactive techniques*. 2000, pp. 343–352.

[Sai+90] T. Saito and T. Takahashi. "Comprehensible Rendering of 3-D Shapes". In: *ACM SIGGRAPH Computer Graphics*. Vol. 24. 4. 1990, pp. 197–206.

[Sal+12] M. Salvi and K. Vidimče. "Surface Based Anti-Aliasing". In: *ACM SIGGRAPH Symposium on Interactive 3D Rendering and Games*. 2012, pp. 159–164.

[Sch+07] S. Schaefer and J. Warren. "Exact evaluation of non-polynomial subdivision schemes at rational parameter values". In: *Computer Graphics and Applications*. 2007, pp. 321–330.

[Sch+09] M. Schwarz and M. Stamminger. "Fast GPU-based adaptive tessellation with CUDA". In: *Computer Graphics Forum* 28.2 (2009).

[Sch+12] D. Scherzer, C. Nguyen, T. Ritschel, and H. Seidel. "Pre-convolved Radiance Caching". In: *Computer Graphics Forum*. Vol. 31. 4. 2012, pp. 1391–1397.

[Seg+12] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 4.3, Core Profile)*. Tech. rep. The Khronos Group Inc., 2012.

[Sei+08] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, et al. "Larrabee: a many-core x86 architecture for visual computing". In: *ACM Transactions on Graphics (TOG)*. Vol. 27. 3. 2008, p. 18.

[Shi+05] L. Shiue, I. Jones, and J. Peters. "A realtime GPU subdivision kernel". In: *ACM Transactions on Graphics* 24.3 (2005), pp. 1010–1015.

[Shi05] O. Shishkovtsov. "GPU Gems II: Programming Techniques for High-Performance Graphics and General-Purpose Computation, chapter 9. Deferred Shading in STALKER". In: *Addison Wesley* 2 (2005), pp. 143–166.

[SK+05] L. Szirmay-Kalos, B. Aszódi, I. Lazányi, and M. Premecz. "Approximate Ray-Tracing on the GPU with Distance Impostors". In: *Comput. Graph. Forum* 24.3 (2005), pp. 695–704.

[Slo+02] P. Sloan, J. Kautz, and J. Snyder. "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments". In: *ACM Transactions on Graphics (TOG)*. Vol. 21. 3. 2002, pp. 527–536.

[Slo+07] P. Sloan, N. Govindaraju, D. Nowrouzezahrai, and J. Snyder. "Image-based proxy accumulation for real-time soft global illumination". In: *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*. 2007, pp. 97–105.

[Sta+03] J. Stam and C. Loop. "Quad/triangle subdivision". In: *Computer Graphics Forum*. Vol. 22. 2003, pp. 79–85.

[Sta98] J. Stam. "Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values". In: *SIGGRAPH*. 1998, pp. 395–404.

[Swo09] M. Swoboda. *Deferred Rendering in Frameranger*. http://directtovideo.wordpress.com/2009/11/13/deferred-rendering-in-frameranger. 2009.

[Tab+04] E. Tabellion and A. Lamorlette. "An approximate global illumination system for computer generated films". In: *ACM Transactions on Graphics (TOG)*. Vol. 23. 3. 2004, pp. 469–476.

[Thi+11] S. Thiedemann, N. Henrich, T. Grosch, and S. Müller. "Voxel-based global illumination". In: *Proc. Symposium on Interactive 3D Graphics and Games (I3D)*. 2011, pp. 103–110.

[Thi09] N. Thibieroz. "ShaderX7: Advanced Rendering Techniques". In: Shaderx series. Charles River Media, 2009. Chap. Deferred Shading with Multisampling Anti-Aliasing in DirectX10.

[Val07] M. Valient. "Deferred rendering in Killzone 2". In: *The Develop Conference and Expo*. 2007.

[Vea+97] E. Veach and L. Guibas. “Metropolis light transport”. In: *Proc. Computer graphics and interactive techniques*. 1997, pp. 65–76.

[Vea97] E. Veach. “Robust Monte Carlo methods for light transport simulation”. PhD thesis. Stanford University, 1997.

[Vla+01] A. Vlachos, J. Peters, C. Boyd, and J. Mitchell. “Curved PN triangles”. In: *Proc. Symposium on Interactive 3D graphics*. 2001, pp. 159–166.

[VO+97] C. Van Overveld and B. Wyvill. “Phong normal interpolation revisited”. In: *ACM ToG* 16.4 (1997), pp. 397–419.

[Wal+05] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. Greenberg. “Lightcuts: a scalable approach to illumination”. In: *ACM SIGGRAPH*. 2005, p. 1107.

[Wal+07] I. Wald and V. Havran. “On building fast kd-trees for ray tracing, and on doing that in O (N log N)”. In: *Interactive Ray Tracing 2006, IEEE Symposium on*. 2007, pp. 61–69.

[Wal+09] I. Wald, W. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. Parker, and P. Shirley. “State of the art in ray tracing animated scenes”. In: *Computer Graphics Forum*. Vol. 28. 6. 2009, pp. 1691–1722.

[War+88] G. Ward, F. Rubinstein, and R. Clear. “A ray tracing solution for diffuse interreflection”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 22. 4. 1988, pp. 85–92.

[Wim+06] M. Wimmer and C. Scheiblauer. “Instant Points”. In: *Proc. Symposium on Point-Based Graphics 2006*. 2006, pp. 129–136.

[Wol07] G. Wolberg. “Sampling, Reconstruction, and Antialiasing George Wolberg”. In: (2007).

[Wym05] C. Wyman. “An approximate image-space approach for interactive refraction”. In: *ACM Transactions on Graphics* 24.3 (2005), pp. 1050–1053.

[Xia+96] J. Xia and A. Varshney. “Dynamic view-dependent simplification for polygonal models”. In: *Proc. Visualization*. 1996, pp. 327–334.

[Yan+08] L. Yang, P. Sander, and J. Lawrence. “Geometry-Aware Framebuffer Level of Detail”. In: *Computer Graphics Forum*. Vol. 27. 4. 2008, pp. 1183–1188.

[Yao+10] C. Yao, B. Wang, B. Chan, J. Yong, and J.-C. Paul. “Multi-Image Based Photon Tracing for Interactive Global Illumination of Dynamic Scenes”. In: *Computer Graphics Forum (Proc. of the EGSR)* 29 (4 2010), pp. 1315–1324.

[Yee04] H. Yee. “A perceptual metric for production testing”. In: *Journal of Graphics Tools* 9.4 (2004), pp. 33–40.

[Yeo+09] Y. Yeo, T. Ni, A. Myles, V. Goel, and J. Peters. “Parallel smoothing of quad meshes”. In: *Vis. Comput.* 25.8 (2009), pp. 757–769.

[You07] P. Young. *Coverage-Sampled Antialiasing*. http://developer.download.nvidia.com/assets/gamedev/docs/CSAA_Tutorial.pdf. 2007.

[Zho+07] K. Zhou, X. Huang, W. Xu, B. Guo, and H. Shum. "Direct Manipulation of Subdivision Surfaces on GPUs". In: *ACM Transactions on Graphics (SIGGRAPH)* (2007).

[Zho+08] K. Zhou, Q. Hou, R. Wang, and B. Guo. "Real-time kd-tree construction on graphics hardware". In: *ACM SIGGRAPH Asia*. 2008, pp. 1–11.

[Zor+00] D. Zorin, P. Schröder, A. Levin, L. Kobbelt, W. Sweldens, and T. DeRose. "Course Notes Subdivision for Modeling and Animation". In: *ACM SIGGRAPH*. 2000.

[Zor99] D. Zorin. "Subdivision zoo". In: *Subdivision for modeling and animation* (1999), pp. 65–104.

[Zwi+01] M. Zwicker, H. Pfister, J. Van Baar, and M. Gross. "Surface splatting". In: *Proc. Computer graphics and interactive techniques*. 2001, pp. 371–378.

[Cor09] N. Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Whitepaper. 2009. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.