Interactive Procedural Modeling Approaches and Visual Illumination Inspection Techniques for Virtual Scenes

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Tim-Christopher Reiner

aus Waiblingen

Tag der mündlichen Prüfung:	17. Juli 2017
Erster Gutachter:	Prof. Dr.-Ing. Carsten Dachsbacher
Zweiter Gutachter:	Prof. Dr. techn. Michael Wimmer

Contents

List of Figures

1 Introduction

This thesis consists of two parts and their topics contribute to two integral parts in the domain of computer graphics: *geometry* and *light*.

The first part deals with the procedural creation of geometry for virtual scenes. It presents procedural modeling techniques for implicit surfaces and shows how to combine individual objects into larger, more complex scenes, and how to render them efficiently. Another chapter shows how to optimize these objects for 3D printing. We will also show a new method for printing objects with textured surfaces.

The second part then covers the lighting and illumination of virtual scenes. New visualization techniques help to reveal the fundamental light transport within a virtual scene. We will also give a brief overview of light manipulation techniques, which assist artists in fine-tuning selected parts of a scene; while the result will no longer be physically correct, it still appears plausible and appealing to the viewer. Finally we will proceed to the real world which embraces us: which inferences can we draw from real world scenes about their illumination? Again, visualizations of the reconstructed light transport will be of assistance to us.

The *interactivity* of all methods and tools is a common theme which follows us through all chapters. Interactivity enables the user to interact with all modeling and visualization environments in real time and allows him to understand the ramifications of each step immediately. With this end in view, all relevant algorithms have been implemented on graphics hardware.

1.1 Chapter Overview

We will now start with a short introduction of the individual chapters of this thesis. This provides an overview of the structure of this work, and it also gives a brief motivation why we chose to tackle each individual problem. The *authored references* section on page 139 lists all publications, which form the basis of this thesis, in the order of their appearance.

After a German summary of all chapters, the main part begins.

Procedural Modeling. Chapter 2 gives an overview of procedural modeling techniques to create objects and textures. They cover algorithms or a set of rules which describe the process of constructing an individual object step by step. This enables us to create more complex objects right up to visually rich scenes in an automated manner.

Procedural modeling comes with two significant drawbacks: it is difficult to control algorithms to deliver the desired results; moreover, evaluating procedural descriptions can take a long time, depending on complexity. The following two chapters tackle these drawbacks. An interactive modeling environment allows for modeling implicit surfaces in an intuitive way. Then, we show how to store evaluations of complex functions efficiently in a runtime cache.

Modeling implicit surfaces. Chapter 3 presents a complete interactive modeling environment for implicit surfaces, which are represented using distance functions. A distance function for an object returns the Euclidean distance to the object's surface for a given point in space. It is very easy to construct distance functions for primitives such as spheres, tori, planes, cylinders, and cones. Implicit surfaces are amenable to ray tracing, and using distance functions enables us to apply the robust *sphere tracing* algorithm to accelerate the numerical evaluations of surface intersections significantly.

Implicit modeling is fascinating: it enables complex operations in an elegant way, preserving unlimited detail, smooth surfaces, and sharp edges. It always maintains a correct topology during modeling, which simplifies many object manipulations. As some operations do not preserve the Euclidean space, we also discuss how to cope with distorted distance values.

A runtime cache for procedural modeling. Techniques such as *constructive solid modeling* and *displacement mapping* can be used to transform simple, planar primitives into much more complex and visually rich objects. Procedural textures for displacement mapping can be generated using *noise functions*. If all objects and textures are evaluated in real-time, rendering will be computationally more expensive and interactivity might no longer be guaranteed.

Chapter 4 presents a runtime cache to store evaluations of noise functions. We access this cache using a novel hashing technique directly on graphics hardware. This enables us to accelerate rendering significantly, while utilizing only a limited amount of graphics memory.

Support-free shapes for fabrication. 3D printing can be regarded as a new output medium in computer graphics and it spawned a large body of research in the recent years. The following two chapters target printers for additive manufacturing based on the *fused deposition modeling (FDM)* method. An FDM printer emits heated, fluid filament through its nozzle and prints an object layer by layer. This does not allow printing overhangs or disconnected islands; these parts require printing additional *support structures*, which have to be removed after printing in a manual, tedious process.

Chapter 5 introduces modeling tools which guarantee that an object can always be printed *support-free*. For this purpose three operators are introduced: *trim*, *preserve*, and *grow*. The trim operator blights overhangs by continuously pruning any parts without support from below. The preserve operator actively blocks the removal of critical parts which are required for the support-free printing of material above. The grow operator adds additional matter to critical parts until these parts can be printed support-free. Again we present an interactive modeling environment to model and print support-free example objects.

Dual-color mixing for FDM printers. FDM printers featuring multiple nozzles can load and alternate between filament in different colors. Chapter 6 presents a method which alternates colors for every layer and modulates the outlines of a 3D object with sine curves. After each pair of layers, the phase of the sine curves is shifted by 180° to transpose local minima and maxima. This results in an overlay of layers with different colors.

We print examples of textured objects using black and white filament. According to the textures, the amplitudes of the sine curves are increased or decreased. The higher the amplitude of the modulated outlines of the white (black) layers, the brighter (darker) the object appears to the viewer. Using this method, we can create the impression of intermediate grayscale tones using only two different filaments.

The parametrization of the outlines leads to salient artifacts where minima and maxima of neighboring layers do not match. The method counteracts by propagating parametrization information from one layer upwards to the next layer above.

Fundamentals of light transport. The second part of this thesis starts with Chapter 7 giving an overview of the fundamentals of light transport in computer graphics. We introduce important radiometric and photometric quantities and explain how light can be interpreted as a high-dimensional *light field*. The *plenoptic function* $L(x, y, z, \theta, \phi)$ plays a key role in all remaining chapters. It returns the radiance for a position and orientation in space, and can be further dependent on the wavelength (λ) and time (t). Using the plenoptic function, we can conceptually reconstruct an image from a scene from any arbitrary viewpoint.

Light transport in virtual scenes. Visualizations help to convey more complex data by providing suitable graphical representations. There exists a large body of well-established visualization techniques for physical quantities such as temperature, pressure, or flow. Visualizing light transport is more complex, however, as light propagates into all different directions at the same time; furthermore, it can be reflected on surfaces or travels through participating media.

Chapter 8 presents novel light inspection tools which help to understand the fundamental light transport within a virtual scene: false-color renderings, spherical plots, light path inspections, volumetric inspections, and a particle flow tool. It also contains a broad user study to assess the usefulness of each individual tool for a variety of different tasks.

Light transport manipulation. Modern animation movies usually rely on physically based image synthesis to render photorealistic scenery. However, artists may want to fine-tune selected parts and intricate details of a scene in a way which is no longer physically correct, yet still very plausible and appealing to the viewer.

Chapter 9 gives a brief overview of selected methods to manipulate light transport in virtual scenes. It also explains how this manipulation can be interconnected with the visualization techniques presented in the previous chapter.

Light transport in real-world scenes. All methods so far targeted virtual scenes, where the light transport can be entirely simulated. In the final Chapter 10 we focus on limited real-world data, where our scenes are represented by only a set of photographs, including depth

information. We have to reconstruct the light transport within a real-world scene first, before we can visualize it.

Initially, we analyze which areas of the scene qualify as potential light sources. If a high dynamic range image is available, we can extract the areas with a high light intensity to classify them as regions emitting light. With additional depth information available, we can start to sample the light field. This enables us to reconstruct a new image of the scene from another viewpoint, which is equivalent to evaluating the plenoptic function. Then again we are able to apply our light transport visualization techniques.

1.2 German Summary – Deutsche Zusammenfassung

Diese Arbeit besteht aus zwei Teilen, die integrale Bestandteile der Computergrafik sind. Im ersten Teil werden *Methoden zur prozeduralen Erzeugung von Geometrie für synthetische Szenen* betrachtet. Hier werden prozedurale Modellierungstechniken für implizite Oberflächen vorgestellt und es wird erläutert, wie einzelne Objekte zu komplexen Szenen zusammengefügt werden können. In einem weiteren Kapitel wird dargestellt, wie sich eine solche Modellierumgebung erweitern lässt, um Objekte für die Ausgabe auf 3D-Druckern zu optimieren. Außerdem wird ein neuartiges Verfahren für das Drucken von texturierten Oberflächen vorgestellt.

Der zweite Teil der Dissertation (ab Seite 85) beschäftigt sich dann mit dem *Beleuchtungsdesign und der Ausleuchtung von virtuellen Szenen*. Es werden zunächst neue Visualisierungstechniken eingesetzt, um den grundlegenden Lichttransport innerhalb einer Szene verständlich darzustellen. Dann werden zusätzliche Werkzeuge zur Manipulation des Lichttransports vorgestellt, um es zum Beispiel Künstlern während der Feinabstimmung zu ermöglichen, einzelne Beleuchtungsaspekte einer Szene hervorzuheben oder Details so abzuändern, dass das Resultat weiterhin physikalisch plausibel erscheint. Abschließend erfolgt ein Transfer der Methoden auf die reale Welt: welche Rückschlüsse können aus Aufnahmen von realen Szenen bezüglich ihrer Ausleuchtung geschlossen werden? Erneut werden Visualisierungen des dabei rekonstruierten Lichttransports unterstützend eingesetzt.

Die *Interaktivität* aller vorgestellten Werkzeuge und Methoden ist ein grundlegendes Ziel dieser Arbeit: Der Nutzer soll in Echtzeit mit den Modellier- und Visualisierungsumgebungen interagieren und die Auswirkungen jedes Arbeitsschrittes unmittelbar nachvollziehen können. Dazu wurden alle relevanten Algorithmen effizient auf Grafikhardware implementiert.

Prozedurale Modellierung. Kapitel 2 gibt eine Einführung in die prozedurale Modellierung, die eine Reihe mächtiger und vielfältiger Techniken, um Objekte und Texturen zu erzeugen, umfasst. Dabei werden Algorithmen oder eine Menge von Regeln ausgeführt, welche den schrittweisen Aufbau eines Objekts beschreiben. Durch die Auswertung einer solchen kompakten prozeduralen Beschreibung können komplexe Objekte bis hin zu reichhaltigen Szenen automatisiert erzeugt werden, ohne diese aufwendig von Hand zu modellieren.

Diese Vorgehensweise bringt jedoch zwei entscheidende Nachteile mit sich: die gewünschten Resultate sind schwer kontrollierbar und die Auswertung einer prozeduralen Beschreibung kann je nach Komplexität sehr viel Zeit in Anspruch nehmen. Die beiden nachfolgend vorgestellten Arbeiten nehmen sich dieser Probleme an: eine interaktive Anwendung vereinfacht

den Modellierungsprozess für implizite Oberflächen. Anschließend wird betrachtet, wie die Ergebnisse solcher aufwendigen Berechnungen in einem Zwischenspeicher effizient abgelegt werden können.

Modellieren impliziter Oberflächen. Um auf intuitive Art implizite Oberflächen zu erzeugen, wird in Kapitel 3 eine vollständige interaktive Modellierungsumgebung für Distanzfunktionen vorgestellt. Eine solche Distanzfunktion lässt sich beispielsweise für eine Kugel mit Radius r wie folgt aufstellen: $\phi(\vec{x}, r) = |\vec{x}| - r$. Sie gibt für einen Punkt $\vec{x}$ den euklidischen Abstand zur Oberfläche der Kugel zurück. Auch andere Primitive wie Tori, Ebenen, Zylinder oder Kegel lassen sich sehr einfach durch Distanzfunktionen beschreiben. Implizite Beschreibungen sind interessant, da sich mit ihnen komplexe Operationen und Verknüpfungen auf elegante Weise bewerkstelligen lassen, wobei unendliche Genauigkeit, glatte Oberflächen und scharfe Kanten erhalten bleiben. Außerdem bleibt eine korrekte Topologie während eines Modelliervorgangs gewährleistet, wodurch vielfältige Objektmanipulationen vereinfacht werden.

Implizite Oberflächen eignen sich sehr gut, um mit Hilfe des Raytracing-Verfahrens ihre Sichtbarkeit von der Position des Betrachters im Raum aus zu ermitteln. Distanzfunktionen beschleunigen dabei die aufwendige numerische Ermittlung der Oberflächenschnittpunkte.

Es werden beispielhaft einige Objekte dargestellt, die mit Hilfe des entwickelten Modellierprogramms erstellt worden sind. Diese werden durch Distanzfunktionen wesentlich kompakter beschrieben als durch diskrete Dreiecksnetze und behalten dabei ihre glatten Oberflächen für beliebig hohe Auflösungen bei.

Zwischenspeicher für Auswertungen. Distanzfunktionen von Primitiven wie Kugeln, Tori oder Kegeln lassen sich einfach aufstellen und schnell auswerten. Durch konstruktive Festkörpergeometrie *(engl. "constructive solid geometry")* können diese Basisprimitive zu komplexeren Objekten kombiniert werden. Um den planaren Oberflächen eine Struktur zu verleihen, besteht außerdem die Möglichkeit, diese mit Hilfe von Displacement-Mapping anhand einer Textur zu verformen. Solche Texturen können durch Noise-Funktionen generiert werden: Sie sind ein mächtiges Werkzeug der prozeduralen Modellierung, um die visuelle Komplexität von Objekten zu erhöhen und diese erst interessant wirken zu lassen.

Die Modellierung von impliziten Oberflächen erfordert eine häufige Auswertung der Distanzfunktionen, wenn Sichtstrahlen mit den Objekten geschnitten werden. Während eine Interaktivität des Programms bei eher einfachen Objekten gewährleistet bleibt, ist dies bei deutlich komplexeren Szenen nicht mehr der Fall.

Der in Kapitel 4 vorgestellte Ansatz legt die Auswertungen von Noise-Funktionen in einem Zwischenspeicher ab. Der Zugriff auf diesen Speicher erfolgt durch ein neuartiges Hashing-Verfahren unmittelbar auf der Grafikkarte. Durch geschickt gewählte Hash-Funktionen kann der reservierte Speicherplatz klein gehalten und gleichzeitig eine deutliche Beschleunigung der Auswertungen der Distanzfunktionen erreicht werden. Ein interaktives Modellieren bleibt somit gewährleistet.

Optimierungen für 3D-Drucker. 3D-Drucker können in der Computergrafik als ein neuartiges Ausgabemedium aufgefasst werden und ihr Forschungsbereich ist im Laufe der letzten

Jahre rasant gewachsen. Die nachfolgenden Arbeiten sind für 3D-Drucker ausgelegt, die auf dem Schmelzschicht-Verfahren basieren: Der Drucker erhitzt einen Plastikstrang, das sogenannte Filament, um dieses im Druckkopf zu verflüssigen und durch eine Spritzdüse auf das Modell Schicht für Schicht aufzutragen. Durch das schichtweise Auftragen können keine Überhänge gedruckt werden, ohne dass diese durch Stützstrukturen Halt finden. Diese Stützen müssen nach dem Drucken aufwendig von Hand entfernt werden.

In Kapitel 5 werden Modellierwerkzeuge vorgestellt, die gewährleisten, dass ein 3D-Objekt stets ohne Hilfe von Stützstrukturen gedruckt werden kann. Dazu werden drei Operatoren eingeführt: *trim*, *preserve* und *grow*. Der Trim-Operator entfernt kontinuierlich Überhänge, die somit erst gar nicht entstehen können. Der Preserve-Operator verhindert das Entfernen von den Teilen des Objekts, auf dem andere Teile unmittelbar aufbauen. Der Grow-Operator fügt solange Materie an kritischen Stellen hinzu, bis diese ohne Stützstrukturen gedruckt werden können. Auch für diese Werkzeuge wurde eine interaktive Modellierungsumgebung entwickelt, mit deren Hilfe beispielhaft einige Objekte modelliert und gedruckt wurden.

Passive Farbmischung für 3D-Drucker. Verfügt der Drucker über mehrere Druckköpfe, können verschiedenfarbige Filamente geladen und abwechselnd gedruckt werden. Das in Kapitel 6 vorgestellte Verfahren wechselt nach jeder Schicht die Farbe und moduliert die Konturen eines 3D-Modells mit Sinuskurven. Nach jeweils zwei gedruckten Schichten wird die Phase der Sinuskurven um 180° verschoben, sodass lokale Maxima und Minima vertauscht werden. Dadurch überlagern sich die Schichten mit ihren verschiedenen Farben.

Es werden einige Beispiele gezeigt, in denen schwarzes und weißes Filament verwendet und Texturen auf 3D-Modelle aufgetragen wurden. Anhand der Texturen werden an den entsprechenden Stellen die Amplituden der Sinuskurven erhöht oder verringert. Je höher die Amplitude der Konturen der weißen (schwarzen) Schichten, umso heller (dunkler) wirkt das Objekt an dieser Stelle auf den Betrachter. Durch dieses Verfahren kann mit nur zwei unterschiedlichen Filamenten der Eindruck von intermediären Farbabstufungen hervorgerufen werden. Sie sind nach dem Drucken deutlich zu erkennen.

Durch die Parametrisierung der Konturen entstehen Artefakte, wenn Minima und Maxima der Sinuskurven nicht exakt überlagert werden. Das Verfahren minimiert diese Artefakte, indem die Parametrisierungen zweier benachbarter Schichten in Abhängigkeit gesetzt werden.

Visualisierung von Lichttransport. Mit Hilfe von Visualisierungen werden komplexe Sachverhalte besser verständlich und grafisch interpretierbar aufbereitet. Klassische Methoden der wissenschaftlichen Visualisierung sind weit verbreitet, um physikalische Größen wie Temperatur, Druck oder Strömungen auf Oberflächen oder in Volumen visuell darzustellen.

Auch Licht kann hierbei als Feld interpretiert werden. Um das Lichtfeld im dreidimensionalen Raum auszuwerten, greifen wir auf die sogenannte plenoptische Funktion zurück: $L(x, y, z, \theta, \phi)$. Diese liefert die Strahldichte *(engl. radiance)* für eine Position und eine Orientierung im Raum, und kann zusätzlich von der Wellenlänge (λ) und der Zeit (t) abhängig sein. Mit ihrer Hilfe lässt sich ein Bild einer Szene aus jedem beliebigen Betrachtungswinkel für einen beliebigen Zeitpunkt erstellen. Für ein besseres Verständnis gibt Kapitel 7 eine Einführung in die Grundlagen des Lichttransports und wie dieser in der Computergrafik behandelt wird.

Lichttransport in virtuellen Szenen. Die Visualisierung von Lichttransport ist komplex, da sich Licht an jeder Position in einem Raum zur gleichen Zeit in jede mögliche Richtung ausbreitet, und zusätzlich an Oberflächen reflektiert oder innerhalb von Medien gestreut wird. Diese Arbeit erweitert etablierte Visualisierungstechniken, um den Lichttransport auf seine wesentlichen Merkmale zu reduzieren und grafisch leicht verständlich aufzubereiten. Ziel ist es, darzustellen, wie sich der Lichttransport an bestimmten Stellen im Raum zusammensetzt.

In Kapitel 8 werden fünf Werkzeuge vorgestellt, die zur Visualisierung von Lichttransport geeignet sind (Abschnitte 8.4.1 bis 8.4.5):

1. Falschfarben-Darstellungen radiometrischer Größen, um zum Beispiel die tatsächliche Helligkeit von Oberflächen einfacher zu erkennen.
2. Sphärische Darstellungen ein- oder ausgehender radiometrischer Größen an einem Punkt im Raum – dazu wird die plenoptische Funktion an dieser Stelle ausgewertet.
3. Darstellungen von Lichtpfaden: Pfeile zeigen Herkunft, Intensität und Farbe der Bestandteile, die für den Farbeindruck an einem Oberflächenpunkt ausschlaggebend sind – ebenfalls durch Auswertung der plenoptischen Funktion.
4. Ein Bereich kann durch einen virtuellen feinen Nebel gefüllt werden. Dadurch wird der Lichttransport innerhalb eines Volumens erst sichtbar, der sonst nicht erkennbar wäre.
5. Um ein besseres Verständnis dafür zu erhalten, wie das Licht in einer Szene fließt, können an einer bestimmten Stelle virtuelle Partikel erstellt werden. Diese folgen anschließend der Richtung des wesentlichen Lichttransports.

Ein wesentlicher Bestandteil dieser Arbeit ist eine umfangreiche Benutzerstudie, in der untersucht wurde, inwieweit die einzelnen Werkzeuge für verschiedene Fragestellungen zur Beleuchtung in unterschiedlichen Szenen effektiv einsetzbar sind. Je nach Fragestellung und Szene existieren unterschiedliche Werkzeuge, die die Auswertung stark vereinfachen.

Manipulation des Lichttransports. Für computergenerierte Bilder und moderne Animationsfilme werden häufig physikalisch korrekte Bildsynthese-Techniken für photorealistische Ergebnisse verwendet. Zur Feinabstimmung und für die Arbeit an Details können im nächsten Schritt Bestandteile der Szene so manipuliert werden, dass das Resultat optisch ansprechender erscheint, und dabei zwar nicht mehr physikalisch korrekt, jedoch weiterhin plausibel ist.

Kapitel 9 gibt einen kurzen Überblick über verschiedene Verfahren, um den Lichttransport in virtuellen Szenen zu manipulieren. Dabei wird auch erläutert, wie die Manipulation direkt mit der Visualisierung von Lichttransport aus dem vorherigen Kapitel gekoppelt werden kann.

Lichttransport in realen Szenen. Die bisherigen Verfahren wurden für virtuelle Szenen vorgestellt, in denen der Lichttransport vollständig simuliert werden kann. Kapitel 10 betrachtet abschließend das Problem, dass für reale Szenen nur eingeschränkte Daten verfügbar sind: in unseren Beispielen verwenden wir eine Anzahl von Photos, die zusätzliche Tiefeninformationen enthalten. Der Lichttransport muss zunächst plausibel rekonstruiert werden, bevor er visualisiert werden kann.

In einem ersten Schritt wird analysiert, welche Bereiche der Szene als potentielle Lichtquelle infrage kommen. Wird eine Szene durch ein Hochkontrastbild *(engl. "high dynamic range image")* repräsentiert, werden die Bildbereiche erfasst, die über eine hohe Lichtintensität verfügen. Diese Bereiche können als Lichtquellen interpretiert werden.

Enthält ein Bild neben den Farbwerten eine zusätzliche Tiefeninformation, kann aus den vorhandenen Informationen das Lichtfeld so weit wie möglich rekonstruiert werden. Durch die Tiefeninformation ist es möglich, die Szene aus der Sicht eines bestimmten Punktes im Raum zu rekonstruieren, sofern die entsprechende Bildinformation in den Eingangsdaten vorliegt: dies entspricht der Auswertung der plenoptischen Funktion und ermöglicht wiederum die Anwendung von Visualisierungstechniken auf den rekonstruierten Lichttransport.

Part I

A Procedural Modeling Approach to Interactive Geometry Creation

2

Procedural Modeling: An Overview

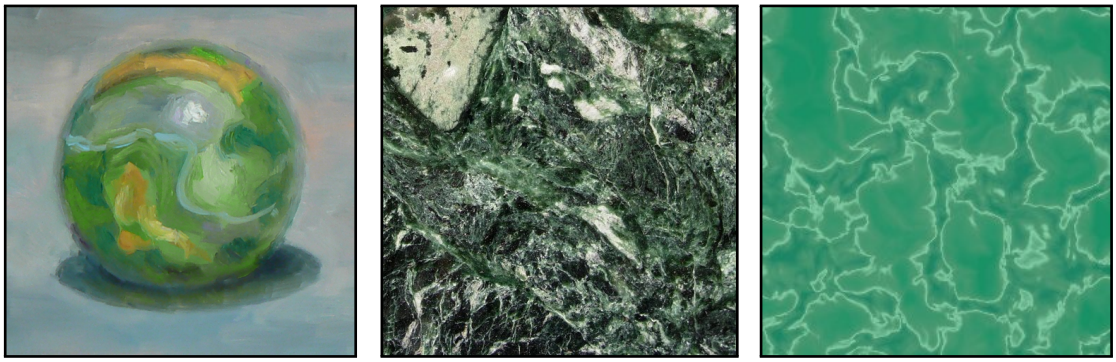
Today, computer-generated imagery is ubiquitous. We enjoy computer graphics in many areas of our daily lives: by watching 3D animation movies with photorealistic special effects, playing immersive video games, or interacting with graphical user interfaces on computers, laptops, and more recently, also our mobile phones. More abstract, we benefit from illustrations, visualizations, and graphs used to convey more complex issues in a visual, more accessible, and more comprehensible way.

We are also surrounded by computer graphics in ways which may not be obvious for us in the first place: browsing advertising material for new furniture, did the manufacturers really arrange a polished, perfectly lit showroom and hire a professional photographer to create impressing imagery? More often than before, we will be looking at a photorealistic rendering instead. In similar fashion, during the construction of new buildings and apartments, even entire residential areas, construction companies distribute elegant renderings of future real estate for prospective buyers.

This all leads us to a key question of computer graphics research: what is a (photo-)realistic computer-generated image? One approach to this question was coined by Alvy Ray Smith, who claimed that photorealism is roughly equivalent to *visual complexity*. As Pat Hanrahan points out in Ebert et al. [2002], visual complexity is mainly based on large amounts of a large variety of primitives. The whole process of exploiting raw computational power to generate visual complexity is called *procedural modeling*.

2.1 Procedural Techniques

Procedural modeling covers a set of powerful procedural techniques. These are algorithms or a set of rules which *describe* how to create a model. Such a model can be a simple texture or a basic object. (It could also be animations, sound, text, or all kinds of generic data.) By adding more steps, loops, or (nested) subroutines to a procedural description, an object or texture can grow tremendously in visual complexity. This approach is very contrary to creating content manually or using cameras or scanners to capture geometry and textures. Figure 2.1 compares painting, photographing, and programming as possible ways to create a marble texture.



(a) *Marble painting* **(b)** *marble photo* **(c)** *procedural marble texture*

Figure 2.1: *Different approaches to create a marble texture:* (a) *detail of the drawing "marble on marble" by Jeff Hayes;* (b) *serpentine marble with a high carbonate content, photo by Andrew Alden;* (c) *procedural texture "African jade" generated by a small C++ function using* libnoise.

Why are procedural techniques so powerful?

One key benefit of procedural modeling is that compact algorithms require only a small amount of storage space, especially when compared to the size of discrete images and model representations. Moreover, procedural descriptions are inherently independent from a fixed resolution: they can be evaluated in any arbitrary detail.

The most powerful feature, however, is *parametric control:* replacing fixed values inside a description with key parameters. Let us again take the procedural marble texture (Figure 2.1c) as an example. An obvious parameter would be the base color to change the overall tone of the texture. Other parameters could be the color of the brighter structures, the intensity of branching, or an integer which specifies the amount of additional structural layers which could be overlaid on top. Changing only few parameters for a model can lead to significantly different outcomes. Smith [1984] coined the term *database amplification* to describe how the programmer is able to create a large amount of detail using only few key parameters.

We will come across many parametrized procedural textures and objects in the next chapters: the iterations of the Menger sponge in Figures 2.4 and 3.7 are specified by a parameter; Chapter 4 features stone, wood, and apple skin textures, which can be altered to a large extent; an intricate procedural texture is presented in Figure 4.13 on page 56: it generates *tree bark* and allows to specify not only colors, but also details of the bark pattern such as cracks, their orientation and alignment, grain and roughness, and the amount of moss on top.

Procedural techniques can be used to create models both on the large and small scale. We can model small particle systems and effects, basic objects and textures, up to more complex objects and entire scenes. These could include rich architecture for urban scenes, or a diverse flora for natural scenes. Even realistic terrains and whole landscapes can be procedurally modeled [*Dachsbacher* 2006], up to entire planets [*Ebert et al.* 2002].

With the advent of powerful programmable graphics hardware, procedural techniques saw another upswing. We are now able to evaluate procedural models in real-time using algorithms directly implemented on graphics hardware.

What are the limitations?

Executing a new procedural algorithm can often be a source of surprise. Modifying only a small part can lead to radical changes in the outcome. On the positive side, this may lead to pleasant new discoveries, and is often described as *serendipity*. Alas, this also implies a considerable loss of *controllability*. It also makes a procedural description difficult to debug.

The compact procedural representations lead to a classic *time versus size* constraint. A more complex, more intricate procedural description might take a long time to evaluate. By contrast, we could simply load a discrete texture image or 3D object directly from a file into memory.

There are certain models which cannot be created procedurally at all. A prime example is the appearance and motion of humans or animals. Moreover, everything involving semantics is inherently difficult to generate automatically: this is why computers and algorithms are unable to produce, e.g., contemporary literature, which is written directly by human authors.

2.2 Texture Mapping and Procedural Textures

Applying textures to a surface is a fundamental technique in computer graphics to add more detail and realism to a virtual scene. In its most basic form, a 2D texture can be regarded as a decal which is pasted on a surface area. This gives the surface an appearance which is not present in the actual geometry itself and simplifies the process of adding visual complexity tremendously. This technique reaches back to the beginning of computer graphics research, when Catmull [1974] presented the first texture-mapped parametric patches.

How can we map a color texture to a basic surface quad? Let the quad be parametrized in each dimension and let two texture coordinates (u, v) span the entire quad, so that its vertices have the coordinates $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$ in the texture space. Let us now load an image with the resolution $width \times height$ from file and store it in a two-dimensional array. When rendering the quad, we will access the texture array for each surface point with the texture coordinates (u, v) as follows: $[u \cdot width], [v \cdot height]$.

Let us now use a *function* to texture the quad instead. One of the simplest procedural textures is a checkerboard pattern. Given (u, v) as an input, it returns *black* if both u and v are ≤ 0.5, or if both are > 0.5. Otherwise, it returns *white*. Another simple function could return a brick pattern. Such a texture can readily be parametrized by allowing to specify the color, width, and height of individual bricks and the mortar in-between.

2.2.1 Anti-Aliasing

At the same time, checkers and bricks are prime examples of patterns prone to severe aliasing artifacts. These artifacts occur when a texture is *undersampled*. In fact, *sampling* and *reconstruction* is integral to computer graphics [*Glassner* 1994; *Foley et al.* 1996]. Especially textures can be interpreted as a signal, and if we render an image without anti-aliasing techniques, this is equivalent to sampling a texture signal once for the center of every pixel in the image plane.

There are several characteristic features and peculiarities regarding signals in the domain of computer graphics. Discrete textures and geometry can only be sampled up to their fixed resolution by nature. Higher frequencies have been inevitably lost during the discretization

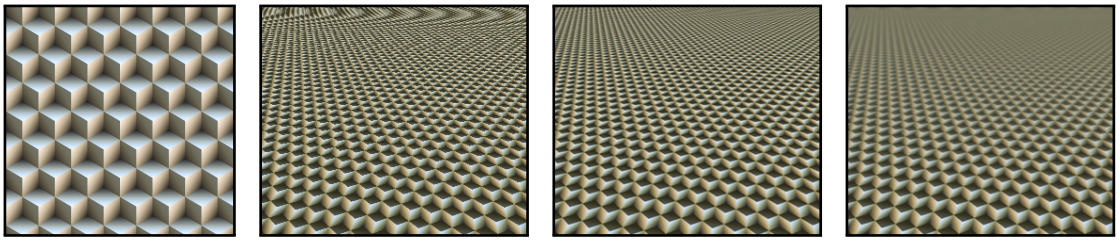


Figure 2.2: From left to right: *a procedural texture with a strong regular pattern; slanting this texture provokes severe aliasing for unfiltered sampling as distance increases; supersampling requires more evaluations of the procedural texture but improves visual quality; blurring to an average color for increasing distance is a cheap yet very effective way to avoid aliasing artifacts.*

process. In the case of a texture, we would draw on interpolation methods to reconstruct intermediate information. On the other hand, sharp, well-defined edges can be indeed highly desirable, but they correspond to indefinitely high frequencies in the signal. Another peculiarity is that the human eye is especially sensitive to regular patterns. This is why we perceive spatial aliasing as particularly disruptive. We can replace these unwanted frequencies with noise by *jittering* our samples. That is, we do not sample using a fixed equidistant grid; instead, all sample points are slightly shifted by random amounts. This results in less perceptible error in the final image, which is significantly less disturbing to the human eye.

The first comprehensive overview of different aliasing artifacts in computer graphics was given in [*Crow* 1977]. The work of Mitchell and Netravali [1988] gives an overview of different reconstruction filters. There exists a range of techniques to fight against aliasing for discrete textures; one of them is *mipmapping* [*Williams* 1983], which targets the *minification problem* and reduces artifacts. It uses prefiltered, downsized versions of a texture for lookup according to the size of the *footprint* of an individual pixel in texture space.

Figure 2.2 shows a procedural texture with a strong regular pattern which exhibits severe aliasing when viewed at a distance. Slanted angles amplify this problem. How can we use anti-aliasing techniques in combination with procedural textures? An obvious approach would be supersampling the texture by evaluating the procedural description multiple times for a single pixel. We could also take measures against aliasing artifacts directly inside a procedural description. This could be done, for instance, by adding a simple box filter (or more sophisticated reconstruction filters) to the code, giving the sample rate as a parameter. A computationally inexpensive approach would be to simply blur the output to an average color according to the distance of the camera. For our work in the next chapter, we will use supersampling in combination with screen-space post processing techniques [*Lottes* 2009].

2.2.2 Solid Textures

Texturing a quad is a trivial process. Sophisticated, much more complex shapes, however, do not feature an intrinsic parametrization. The question then becomes how to map a 2D texture onto the surface of a 3D model, without creating seams, distortions, or other artifacts. Mapping a 2D checkerboard pattern on a 3D surface is a common scenario used to display the visual

quality of different parametrizations. In Chapter 6 we will encounter such a parametrization problem when we procedurally displace the outlines of 3D shapes.

A common mapping technique is called *UV mapping*. It is the tedious manual process of assigning texture coordinates to particular surface points. To model a figurine out of a single material such as wood, marble, or stone, one can also make use of 3D (solid) textures instead. They remove the need for parametrization entirely, as the mere (x, y, z) coordinates of surface points can be mapped directly to 3D texture coordinates (u, v, w).

This is where procedural textures really begin to shine. Let us yet again look at the marble textures (Figure 2.1). Transforming the 2D procedural texture into a solid texture requires just one additional parameter to alter the marble structure slowly and gradually. The w coordinate will be mapped to this parameter. It then creates the impression that we descend deeper into the texture as w increases.

What would be an alternative to procedural textures? We could start with a physical solid block of marble, then gradually removing very thin slabs, and taking photos after each step. (This would obviously destroy the physical marble during the process.) We would then have a solid 3D marble texture consisting of a stack of actual photos. If we did this elaborate and costly process in a professional and careful way, we would likely obtain a magnificent result. Note, however, how much simpler it is to revert to procedural techniques. Aside from that, a procedural texture can easily be made *seamless in all three dimensions*, while our photos are most likely not.

2.2.3 Noise and Turbulence

Looking back at the checkerboard and brick examples, we see how these basic procedural textures all exhibit strong regular patterns. *Noise functions* are typically used in computer graphics to break such a pattern and add variety to create more complex textures. In this context, the requirements for *noise* are commonly stated as follows [*Ebert et al.* 2002]:

1. Noise is repeatable and pseudorandom, i.e., we will always obtain the same results when using the same seed and same inputs.
2. Noise has a known range (usually from -1 to 1) and is bandwidth-limited.
3. Noise does not contain noticeable periodicities or visible regular patterns.
4. Noise is *stationary* and *isotropic*, i.e., translationally and rotationally invariant. Independent of position and orientation in noise space, it will never appear noticeably different.

A simple and efficient class of noise functions is *lattice noise*. Ken Perlin [1985] introduced *Perlin noise* as a form of lattice noise, and presented an improved version in [*Perlin* 2002]. In essence, the core algorithms are built upon the usage of pseudorandom numbers. The fact that we are using pseudorandom (i.e., deterministic), and not truly random numbers, is actually desirable: the same seed allows us to reproduce the exact same results over and over again. The pseudorandom numbers are distributed for each integer coordinate in noise space. This forms the *integer lattice*. The values for the in-between space are then interpolated.

The resulting noise is much more valuable for computer graphics applications than pure *white noise*, which exhibits indefinitely high frequencies. The integer lattice enforces that

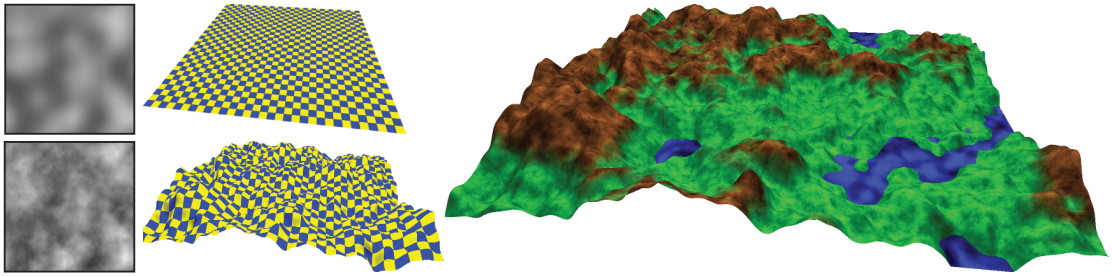


Figure 2.3: Top left: *2D slice through Perlin noise space.* Below: *Accumulating octaves results in turbulence.* Next: *Displacing a quad along its surface normal according to the turbulence texture.* Right: *applying the same turbulence as color textures results in a basic procedural landscape.*

white noise samples are invariably spaced one unit apart. The smooth interpolation then leads to bandwidth-limited frequencies below the Nyquist frequency of an integer lattice step.

Ebert et al. [2002] shows how different octaves of noise functions can then be accumulated in a *turbulence function*. One possible way to construct such a turbulence is:

$$\text{turbulence}_N(\mathbf{x}) = \sum_{i=0}^{N-1} \frac{\text{noise}(2^i \mathbf{x})}{2^i} \quad ,$$

where N octaves (scaled versions of the base noise) blend together. Higher octaves are weighted less. By applying self-similarity we effectively make noise scale-invariant.

According to Darwyn Peachey, Perlin noise is "the function that launched a thousand textures", and it can be unanimously regarded as one of the most important milestones in procedural content generation. Using noise functions allows us to introduce *irregularities* to achieve a higher degree of visual complexity. Figure 2.3 displays a slice through the noise space which is then extended to a turbulence. This turbulence function is used to displace a quad along its normal and to color the surface according to its height. The thresholds between different colors are yet again modulated by a noise function. The result is a very basic procedural landscape which is created using only noise and turbulence functions.

2.3 Procedural Geometry

Besides landscapes, procedural techniques are also very suitable for generating geometry in general. As an example, Figure 2.4 illustrates how to construct a *Menger sponge* one iteration at a time according to a set of rules. For each iteration, a block is subdivided into $3 \times 3 \times 3$ smaller blocks. The central block as well as the six middle blocks of all faces are then removed. Each remaining smaller block is then subdivided again in the next iteration.

Constructive Solid Geometry (CSG) is the umbrella term for applying Boolean operations to (sub-)objects in order to create more complex shapes. Typical operations are *union*, *intersection*, and *difference*. CSG can be used to construct a Menger sponge in an additive manner by unifying smaller cubes. It could also be used in a subtractive manner by iteratively carving out smaller blocks using the difference operator. We will use these CSG operations for various modeling examples in the next chapter.

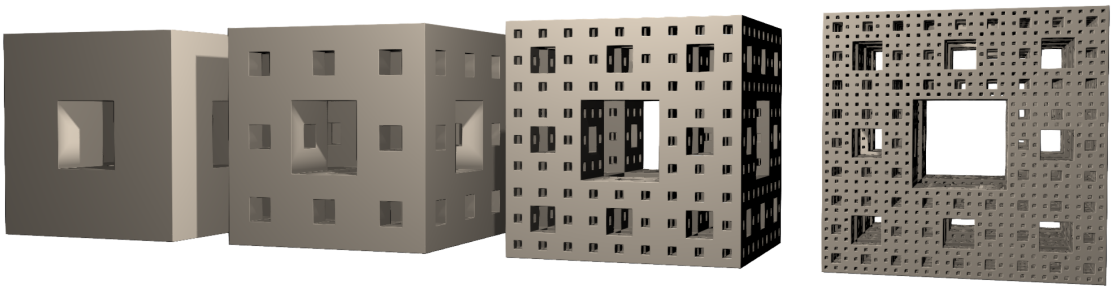


Figure 2.4: *The first iterations of creating a Menger sponge, a 3D fractal.*

Additionally, we will apply *displacement mapping*, as in Figure 2.3. This technique translates surfaces points along their normal according to a texture, the displacement map. Together with (solid) textures, this forms a powerful combination. We will make use of three-dimensional displacement textures in Chapter 4, where we apply a stone structure to example objects.

2.3.1 L-Systems

Procedural geometry can also be generated by advanced geometric modeling techniques. Such techniques include *L-systems*, a powerful grammar-based modeling approach, introduced by Lindenmayer [1968]. An L-system grammar is defined by an *alphabet*, an *initiator*, and a set of *production rules*. The alphabet is a set of *variables* and *constants* (also called *terminals*). The initiator, also called *start* or *axiom*, is a string out of the alphabet which describes the initial state of the L-system. The production rules are used for every iteration of the system. Such a rule consists of the *predecessor* and the *successor*. If a string of variables matches the predecessor, it is replaced by the successor, another string of variables or constants.

L-systems can be used to model fractals with strong regular patterns, such as the Koch curve, the Sierpinski triangle, or the Cantor set. The real beauty of L-systems, though, is their ability to create realistic models of plants, trees, and other natural, organic structures. Prusinkiewicz and Lindenmayer [1990] also published a comprehensive textbook on L-systems and their manifold applications.

2.3.2 Implicit Surfaces

Implicit surfaces are another powerful class of procedural geometry. They have been transferred by Blinn [1982] to computer graphics research. At that time, Blinn called them *blobby* objects and used them to model molecules and electron density clouds. From then on, blobby (or *soft*) objects have spawned a large body of research [*Wyvill et al.* 1986; *Wyvill et al.* 1987].

An implicit surface can be described by an equation $F(x_1, x_2, x_3) = 0$, with $(x_1, x_2, x_3) \in \mathbb{R}^3$. To check if a point $\mathbf{p} = (p_1, p_2, p_3) \in \mathbb{R}^3$ lies on this implicitly defined surface, we evaluate $F(\mathbf{p})$. If the function returns zero, $\mathbf{p}$ is a surface point, otherwise not. These functions can be used to describe a large variety of primitives. They can be easily combined and smoothly merged (see Figure 2.5) to form more complex shapes. The user does not have to provide an explicit description of the shape; these functions are procedurally evaluated during rendering.

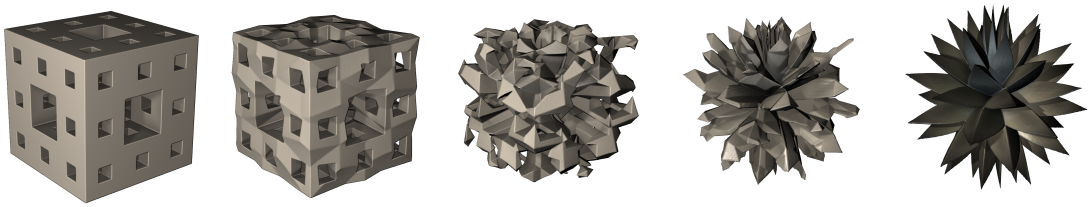


Figure 2.5: *Blending a Menger sponge over to a heavily displaced sphere. Both objects can be implicitly yet intuitively described. The blending operation itself is trivial for implicit surfaces: these two shapes $\phi_A(\mathbf{x})$ and $\phi_B(\mathbf{x})$ are weighted with a blend factor α as (1-α) $\cdot \phi_A(\mathbf{x}) + \alpha \cdot \phi_B(\mathbf{x})$.*

The following two chapters focus primarily on implicit surfaces. For further reading we recommend an overview of interactive techniques for implicit modeling [*Bloomenthal and Wyvill* 1990]. A thorough introduction to implicit surfaces in general is given in [*Bloomenthal and Wyvill* 1997]. We also refer to a textbook on interactive shape design [*Cani et al.* 2008], which also covers the history of shape representations as well as sculpting techniques, as we will present a sculpting approach to interactive shape modeling for 3D printing in Chapter 5.

2.4 More Techniques and Further Reading

The textbook of Ebert et al. [2002] is among the main reference publications about procedural texturing and modeling. This book further covers more advanced modeling and rendering techniques for procedural textures and geometry, also covering areas such as animations, shading, participating media, and fractals. We also refer to Shirley et al. [2009] which covers the fundamentals of computer graphics. It is a reference book with an outline that matches the structure of this thesis very well: covering raster images, addressing implicit surfaces, then leading over to light, color, and global illumination, and closing with the basics of visualization.

Noise itself is such an integral component of procedural techniques that a large body of research exclusively deals with this topic. We refer to Lagae et al. [2010] for a comprehensive survey of procedural noise functions. However, using noise is not the only approach to generate irregular patterns. *Texture synthesis* [*Efros and Leung* 1999] is a method to generate textures, as large as desired, from small sample images (the *exemplar*). The larger synthesized texture then took over the appearance of the smaller exemplar. This can be generalized to synthesize 3D structures and geometry [*Zhou et al.* 2013b], as well as photo collections, videos, artistic brushes, rich materials, and shapes for 3D printing [*Barnes and Zhang* 2017].

We have now seen how versatile and powerful procedural techniques are, but let us again address the issue of *controllability*. Generating rich visual complexity automatically while maintaining control over the outcome at the same time drives the need for interactive "what you see is what you get" environments. There are manifold applications at the interface of procedural and interactive modeling. Lefebvre and Hoppe [2005] developed an interactive tool to control the outcome of synthesized textures. The *SpeedTree* system is widely used in animation movies and video games production, and allows to create visually rich and animated

foliage, plants, and trees. The sheer amount of primitives generated for large botanical scenes then leads to the challenge of rendering such scenes efficiently [*Neubert et al.* 2011].

The *CityEngine* [*Parish and Müller* 2001] enables the automated creation of large 3D urban environments according to a predefined rule set. The entire system expands to the procedural modeling of buildings [*Müller et al.* 2006], facades [*Müller et al.* 2007], and streets [*Chen et al.* 2008]. Like SpeedTree, it is heavily used in the movie production industry. Wonka et al. [2003] presented a grammar-based approach to the automated creation of architecture. Again, such a text-based shape grammar can be hard to control, particularly for artists with a limited computer science background. This is why Lipp et al. [2008] presented an interactive visual editing system for shape grammars to allow a fine-grained control over procedural architecture. Large and detailed urban scenes are another challenge which requires efficient rendering techniques [*Wonka et al.* 2000].

In the next chapter, we will also present an interactive environment for procedural modeling. It allows to model implicit surfaces, which can eventually be arranged to entire scenes. Implicit functions, too, are a powerful way to describe shapes, while being more difficult and less intuitive to control on the function level. In the chapter right after, we will combine these surfaces with procedural noise functions. The seminal work of Perlin and Hoffert [1989] demonstrated how the functional interaction between implicit shapes and noise enables the creation of compelling *hypertextures*. We also address the efficient rendering of our models.

3

Interactive Modeling with Signed Distance Functions

Modeling appealing virtual scenes is an elaborate and time-consuming task. It requires not only training and experience, but also powerful modeling tools that provide the desired functionality to the user. In this chapter, we describe a modeling approach using signed distance functions as an underlying representation for objects. This handles both conventional and complex shape manipulations. Scenes defined by signed distance functions can be stored compactly and rendered directly in real-time using sphere tracing. We demonstrate our approach and provide a full interactive modeling application with immediate visual feedback for the artist. This is a crucial factor for a convenient modeling process. Moreover, dealing with underlying mathematical operations is not necessary on the user level, but at the same time, we provide this functionality for more advanced users. We show that fundamental aspects of traditional modeling can be directly transferred to this novel kind of environment, which results in an intuitive application behavior. We further describe modeling operations which naturally benefit from implicit representations. We show modeling examples where signed distance functions are superior to explicit representations, but discuss the limitations as well.

3.1 Introduction

Implicit descriptions of smooth continuous surfaces are well established in computer graphics, next to discrete approximations like the ubiquitous polygonal meshes. Implicit surfaces spark interest due to their intrinsic way of representing solid objects, aptitude for constructive solid geometry, and eminent blending abilities.

However, it is inherently difficult to render implicit surfaces directly in real-time. This requires an extensive incremental search for surface points, which are neither explicitly given nor easy to compute in general. For this reason, indirect visualization techniques became popular, which utilize a polygonizer such as marching cubes [*Lorensen and Cline* 1987] to generate discrete surface approximations, making them fast to render. Alas, these intermediate steps introduce geometric aliasing, giving up smoothness and high-frequency details.

Implicit surfaces are amenable to ray tracing, but it is difficult to find intersections of rays with complex surfaces analytically. Thus, *ray marching* is often used to render implicit surfaces directly, where possible intersections are determined by marching along a ray in small steps

until a surface is hit. Sphere tracing [*Hart* 1996] speeds up ray marching significantly, but requires *distance surfaces* [*Bloomenthal and Shoemake* 1991]. These surfaces are implicitly described by functions that return a measurement or bound of the geometric distance.

It has always been tempting and desirable to use implicit surfaces for shape modeling. Providing an interactive modeling environment for this purpose is difficult, as it involves the necessity to render implicit surfaces in real-time in order to provide constant visual feedback. Indirect visualization techniques are capable of doing so, but introduce another problem: while editing a surface, continuous re-evaluations of its discrete approximation are required. Unless this is achieved in real-time, interactivity gets interrupted at this crucial point. Reducing approximation quality helps, but it removes subtle details and introduces artifacts as well.

Contributions. In this chapter, we make the following contributions:

- *We compose complex implicit surfaces using signed distance functions.* Then, we show how to translate these compositions into a fragment shader that renders defined surfaces directly on graphics hardware. As a result of using signed distance functions, we are able to apply sphere tracing to render a scene efficiently in real-time.

- *We present a complete modeling environment for objects and scenes that are implicitly defined by signed distance functions.* As we are able to render them in real-time, we provide full interactivity for the modeling tool. We are neither dependent on indirect visualization techniques nor restricted by spatial limitations. Likewise, explicit grids or approximations are not required.

3.2 Previous Work

Previously, a lot of pioneer work in implicit shape modeling has been done. In this section, we focus on fundamental and most related work, ranging from the notion and rendering of implicit surfaces to editing operators and modeling environments for them.

Groundbreaking, hypertextures [*Perlin and Hoffert* 1989] combined implicit shapes with textures to model various phenomena, extending the notion of shape by including surrounding volumes. Turk and O'Brien [1999; 2002] introduced new shape transformations and interpolation techniques for implicit surfaces. Loop and Blinn [2006] showed how to render implicit surfaces in real-time on the GPU using analytic techniques for solving for roots of polynomials. Hence, they are limited to fourth order algebraic surfaces, and form objects by assembling piecewise smooth algebraic surfaces, which were introduced by Sederberg [1985].

It becomes cumbersome to evaluate functions defining implicit surfaces of increasing complexity. Consequently, distance fields attracted attention, which explicitly store distance information inside a three-dimensional grid. Again, this leads to geometric aliasing in trade for fast distance evaluations. Obviously, this representation requires a tremendous amount of memory if used naively, and therefore substantially limits the possible extent of a scene. Frisken et al. [2000] proposed adaptively sampled distance fields, which significantly reduce memory consumption for detailed objects.

The level set method, first presented by Osher and Sethian [1988] and thoroughly explained by Osher and Fedkiw [2002], defines a surface using a time-varying scalar function. It spawned a large body of research and gained recognition in modeling surface deformations.

Museth et al. [2002] contributed editing operators for implicit surfaces based on level set models. Surfaces are imported as distance fields into their level set framework, allowing arbitrary surfaces modified with a single procedure. As their framework is based on uniform grids, it suffers from technical limitations again, particularly spatial limits, memory constraints, and artifacts due to discretization.

Comprehensive approaches to interactive implicit modeling were proposed. A primary contribution is HyperFun [*Richard et al.* 1999], that has been extended by others in various directions. In its essential form, it provides a symbolic user interface for direct textual input in a high-level geometric language. This does not extend to real-time surface visualization.

Our approach closely resembles BlobTrees [*Wyvill et al.* 1999], that have been widely adapted. Its hierarchical structure stores implicit surfaces at the leaves and places composition operators at internal nodes. ShapeShop [*Schmidt et al.* 2005b] is a prominent example based on the BlobTree. Interactivity is achieved by introducing spatial caching for objects.

WarpCurves [*Sugihara et al.* 2010], inspired by FiberMesh [*Nealen et al.* 2007] and Wires [*Singh and Eugene* 1998], extends ShapeShop to allow for interactive manipulation of implicit surfaces using explicit curve-based spatial deformations. Martinez Esturo et al. [2010] described a method for continuous deformations of implicit surfaces focusing on volume, continuity, and topology conservation. Related important areas of research in interactive modeling are haptics-based [*Hua and Qin* 2004] and sketch-based [*Schmidt et al.* 2005b; *Igarashi et al.* 1999] user interfaces for convenient sculpting.

All modeling environments mentioned before use indirect visualization techniques. What sets our solution apart is our real-time direct visualization approach to render a scene.

3.3 Describing Scenes with SDFs

This section briefly introduces SDFs (signed distance functions), shows how to describe primitives with them, and how transformations and combinations can be applied in order to create complex objects. Eventually, whole scenes emerge by assembling several objects.

3.3.1 Definition

We will use the notation from Osher and Fedkiw [2002], who provide, inter alia, a comprehensive introduction to implicit surfaces. Let Ω^- and Ω^+ be the space inside and outside of an implicit surface, respectively, and $\partial\Omega$ the interface in between, i.e., the set of points composing the surface. A signed distance function $\phi(\mathbf{x})$ is then defined as

$$\phi(\mathbf{x}) = \begin{cases} \min(|\mathbf{x}-\mathbf{x}_I|) & \text{if } \mathbf{x} \in \Omega^+, \\ 0 & \text{if } \mathbf{x} \in \partial\Omega, \\ -\min(|\mathbf{x}-\mathbf{x}_I|) & \text{if } \mathbf{x} \in \Omega^-, \quad \text{for all } \mathbf{x}_I \in \partial\Omega, \end{cases}$$

and returns the Euclidean distance from $\mathbf{x}$ to its closest surface point, with a negative sign if inside an object. Thus, SDFs are a subset of implicit functions which imply algebraic distances.

3.3.2 Properties

SDFs retain characteristics of implicit functions, yet provide more beneficial properties, such as $|\nabla\phi| = 1$. This is easily comprehensible, since ϕ returns Euclidean distances, and satisfies the criteria of sphere tracing precisely; see Section 3.4.2. Note that the equation does not hold for points equidistant to at least two interface points, where the path of steepest descent is ambiguous. Fortunately, we can safely ignore this issue, as we march along fixed ray directions.

Our goal is to provide a comprehensive set of flexible modeling operations and transformations. In exchange, we have to accept that certain operations produce distorted distance functions that do not return the correct Euclidean distance anymore. For $0 < |\nabla\phi| < 1$, sphere tracing decelerates but still ensures not to penetrate surfaces. Alas, this is not the case for $|\nabla\phi| > 1$, but sphere tracing is robust enough if a bound is known. In Section 3.4.2 we discuss our counteractions to compensate for steep and distorted gradients.

Moreover, SDFs remain monotonic across the interface, which is another pleasant trait. In other words, they do not have a kink like unsigned distance functions where the outside transitions into the inside. This enables reliable gradient reconstructions, e.g., using central differences, which is required for illumination and shading.

3.3.3 Describing Primitives

Many primitives can be described directly using an SDF. For example, $\phi(\mathbf{x}, r) = |\mathbf{x}| - r$ describes a sphere at the origin with radius r. Other basic primitives such as tori, cylinders, cones, and infinite planes are easy to describe as well. Given the SDF of a two-dimensional curve, it can be easily extended to a solid of revolution: distances are evaluated by projecting points in question onto the plane where the curve is defined.

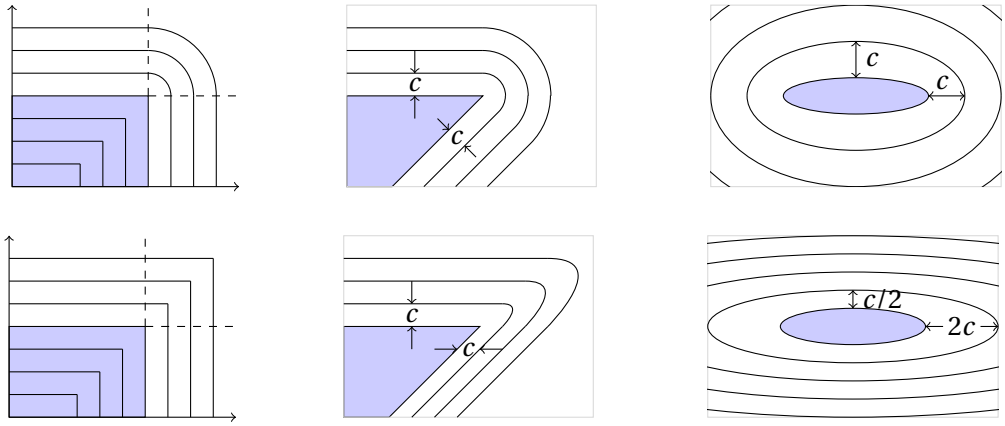
Primitives featuring hard edges between surface regions require SDFs to have corresponding kinks. This can be accomplished by composing individual SDFs for each surface region, using case-by-case analysis which region is closest to a point in question. We try to avoid case-by-case analysis in our shaders, however, as branching is still an expensive operation on the GPU. Instead, we edge surfaces by switching function spaces. For instance, a sphere formed using the L_∞ metric represents an axis-aligned cube. In this way, we can state an SDF for a box with width w, height h, and depth d as

$$\phi(\mathbf{x}, w, h, d) = \max(|x_1| - {}^{w}/_{2}, |x_2| - {}^{h}/_{2}, |x_3| - {}^{d}/_{2}) \ .$$

This SDF returns the Chebyshev distance, which can be smaller than the Euclidean distance. This is the case for a point $\mathbf{p}$ outside the box, whose nearest surface point is $\mathbf{p}_I$, if the line through $\mathbf{p}$ and $\mathbf{p}_I$ is not parallel to one of the coordinate axes (see Figure 3.1a.) Note that this provokes more sphere tracing steps during rendering, but is still much faster and superior to branching thanks to efficient GPU implementations of min/max-functions.

3.3.4 Describing Transformations

In order to apply a transformation T to a surface described by $\phi(\mathbf{x})$, the inverse transformation is applied to the domain. Hence, $\phi(T^{-1}(\mathbf{x}))$ describes the transformed surface.



(a) L_2 *vs.* L_∞ *metric* **(b)** *shearing* **(c)** *non-uniform scaling*

Figure 3.1: (a) *Isocontours differ outside a box for different metrics.* (b) *Shearing leads to increased distances along the shear direction.* (c) *Non-uniform scaling causes both increased and decreased distances along orthogonal axes.* Top/bottom row: *correct/distorted distances.*

Any arbitrary concatenation of translations, rotations, and reflections is a global isometry on Euclidean spaces, i.e., all properties of an SDF are preserved. This, however, does not hold for other transformations such as shearing (Figure 3.1b) and non-uniform scaling (Figure 3.1c). In the case of overvalued distances, sphere tracing might penetrate surfaces or even skip entire objects. We encounter this problem again for morphing, blending, and other non-linear transformations used to model more complex and appealing objects. These transformations distort the Euclidean space to a less predictable extent, and we need to make sure that surfaces are still amenable to sphere tracing. In the following we will discuss transformations known from traditional modeling tools, followed by more artistic ones that are easy to define implicitly.

Rigid Body Transformations

Applying translations and proper rotations preserves the Euclidean space. Thus, to rotate and translate an object described by $\phi(\mathbf{x})$, we evaluate $\phi(R(\hat{e},-\theta)\mathbf{x}-\mathbf{t})$ instead, where $R(\hat{e},\theta)$ is a matrix describing a rotation around an axis $\hat{e}$ by angle θ, and $\mathbf{t}$ is the translation vector.

Scaling

To apply a uniform scale by factor s, we evaluate $s\cdot\phi(s^{-1}\mathbf{x})$. Note the final multiplication by s, since $|\nabla\phi(s^{-1}\mathbf{x})| = s^{-1}$ violates the Euclidean distance. Unfortunately, a compensation for non-uniform scalings by factors s_x, s_y, s_z is not as straightforward (Figure 3.1c). A final multiplication by $\min(s_x,s_y,s_z)$ ensures that the gradient magnitude does not exceed 1, which is desirable for sphere tracing. However, as scaling factors drift apart, distances get overestimated more and more along the axis of the widest stretch, decelerating sphere tracing significantly.

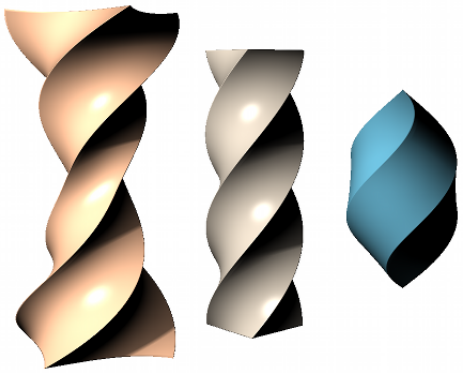


Figure 3.2: *Blending between a box and a sphere with a subsequent twist.*

Constructive Solid Geometry

Particular operations are amazingly simple to apply to implicit surfaces. A prime example is *constructive solid geometry* (CSG). Unions, intersections, and differences of two objects A and B, described by ϕ_A and ϕ_B, satisfy the following statements:

$$\begin{aligned} \phi_{A\cup B}(\mathbf{x}) = 0 \quad &\Leftrightarrow \quad \min(\phi_A(\mathbf{x}), \phi_B(\mathbf{x})) = 0, \\ \phi_{A\cap B}(\mathbf{x}) = 0 \quad &\Leftrightarrow \quad \max(\phi_A(\mathbf{x}), \phi_B(\mathbf{x})) = 0, \\ \phi_{A-B}(\mathbf{x}) = 0 \quad &\Leftrightarrow \quad \max(\phi_A(\mathbf{x}), -\phi_B(\mathbf{x})) = 0. \end{aligned}$$

Note that unions cause wrong distances inside objects, whereas intersections implicate the L_∞ metric outside.

Blending

Blending between two objects A and B is trivial as well, using a blend factor α:

$$(1-\alpha)\cdot\phi_A(\mathbf{x}) + \alpha\cdot\phi_B(\mathbf{x}) \ .$$

Another way of blending between two objects is to evaluate A at the origin, and blend over to B while moving away from the origin, with a stretch factor s defining the transition region:

$$\max(0, 1-s\cdot|\mathbf{x}|)\cdot\phi_A(\mathbf{x}) + \min(1, s\cdot|\mathbf{x}|)\cdot\phi_B(\mathbf{x}) \ .$$

Twisting

A *twist* is easily described implicitly:

$$\xi_a(\mathbf{x}) = \begin{pmatrix} x_1\cos a(x_2) - x_3\sin a(x_2) \\ x_2 \\ x_1\sin a(x_2) + x_3\cos a(x_2) \end{pmatrix},$$

with a linear function a defining amount and offset.

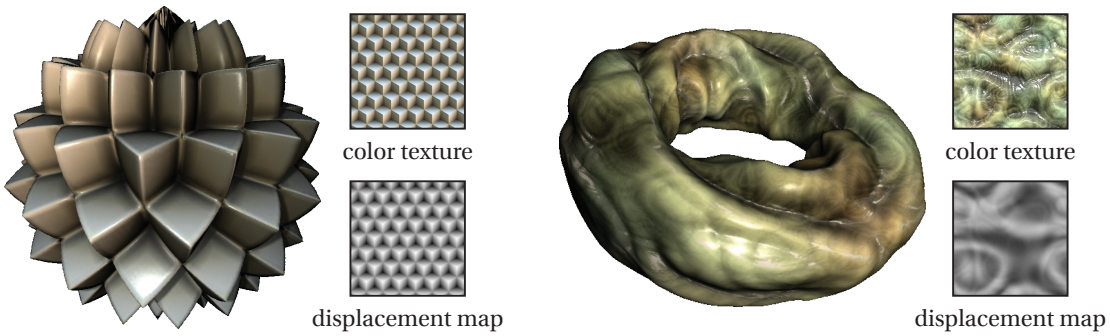


Figure 3.3: *A displaced sphere and an organic structure on a torus using procedural textures.*
Acknowledgements: *we use procedural textures from the free* Filter Forge *online repository.*

Figure 3.2 shows an example for a blend between a sphere and a box, where a twist is applied subsequently. On the far left, a negative α is used. A box is frequently used to demonstrate how implicitly defined edges remain continuous and sharp after transformations.

Displacement Mapping

Displacement mapping can be easily applied to surfaces by simply adding or subtracting values to the results of SDFs. See Figure 3.3 for example objects. Using procedural textures, surfaces remain continuous and smooth. Twists and displacements can also be easily implemented for meshes. This, however, only looks good for highly tessellated meshes. It also requires a constant regeneration of a mesh during an interactive deformation process.

Infinite Repetitions

By applying the modulo operator to the domain, infinite replications of an object can be defined. Figure 3.7 shows two examples. Jittering can be added to break the visible equidistant pattern. For natural scenes, such as many trees in a forest, a Poisson distribution yields much more appealing and convincing results, but it comes with the loss of the implicit topology between neighboring objects. The results of this replication technique are comparable to those of geometry instancing on graphics hardware.

3.3.5 Arranging Scenes

A set of individual objects, where each object is described by its own SDF $\phi_i(\mathbf{x})$, can be easily combined to form a single SDF which represents the entire scene S:

$$\min_{i \in S}(\phi_i(\mathbf{x})) = 0 \Leftrightarrow \mathbf{x} \in \bigcup_{i \in S} \partial\Omega_i.$$

This union of all objects is essentially a large CSG operation .

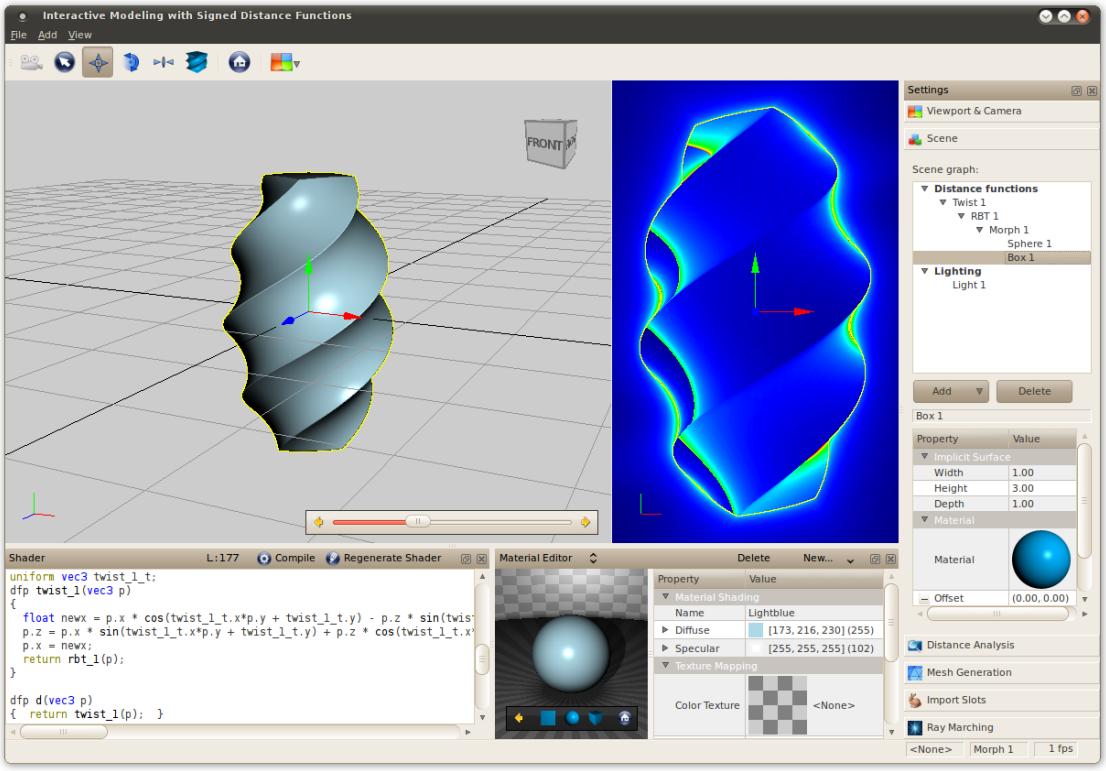


Figure 3.4: *The user interface of our interactive modeling environment. It features a main modeling area and an interface to the scene graph. A work image is displayed to visualize the amount of sphere tracing steps. In addition to using visual manipulators, the user can freely edit the generated fragment shader source to define sophisticated surfaces and transformations.*

3.4 Interactive Modeling using SDFs

Signed distance functions and the transformations described in the previous section form the basis of our modeling environment. Figure 3.4 displays the user interface. We provide an intuitive environment that looks and behaves like traditional modeling software. This is accomplished by sticking to well-known user interface design patterns that allow experienced artists to be productive right from the start.

The key feature is our scene management that has to fulfill two requirements. First, it must be flexible enough to store all shapes, operations, and transformations. Second, it must be possible to transform the scene representation quickly into a fragment shader that is suitable for real-time rendering on graphics hardware.

In this section, we first describe our interactive rendering system for SDFs and the underlying data structures. Performance is crucial to allow for interactive scene manipulation and immediate visual feedback. Next, we discuss the modeling tools that we provide and demonstrate their flexibility by means of several examples. Eventually, we discuss combinations with traditional polygon modeling, and benefits and drawbacks of our modeling approach.

3.4.1 Scene Graph Traversal

We store all scene objects and transformations in a connected acyclic scene graph that is presented to the user as a hierarchical tree structure. Primitives are represented by leaf nodes, while internal nodes store transformations and operations applied to their distance functions. The root node collectively represents the entire scene. The user can modify the hierarchy by rearranging nodes via drag-and-drop at any time.

After modifying the scene graph, i.e., adding, altering, rearranging, or deleting nodes, a new fragment shader has to be generated. This is achieved by a depth-first traversal of the scene graph. Every visited node N prepends its required uniform variables and functions to the shader, applies its transformations, and asks its children to append their source fragments as arguments of the result statement of N. Eventually, primitives write out calls to their corresponding basic signed distance functions. To compose a scene out of individual objects, min-functions are used (see Section 3.3.5).

A small example follows, using the object shown in Figure 3.2. It has been composed using blend and twist transformations on a box and a sphere as follows:

$$\phi\left[\xi_a^{-1}\Big((1-\alpha)\cdot\phi_{\text{Box}}(\mathbf{x})+\alpha\cdot\phi_{\text{Sphere}}(\mathbf{x})\Big)\right].$$

Listing 3.1 shows the generated fragment shader source. There, `p` corresponds to $\mathbf{x}$, `vec2 a` to the linear twist definition of ξ_a, and `alpha` to the blend factor α.

3.4.2 Ray Marching

For a point $\mathbf{x}$ on a ray, evaluating the all-embracing min-function (see Section 3.3.5) of the generated fragment shader yields the distance from $\mathbf{x}$ to its closest surface. One can safely march this distance within a single step along the ray without penetrating a surface. This technique is called sphere tracing [*Hart* 1996], and accelerates ray marching significantly. When the distance is below a certain threshold, $\mathbf{x}$ is considered to be a surface point.

As discussed in the previous section, certain transformations violate the Euclidean distance. A counteracting heuristic method is to sample the space within the scene at several points and then adjust the marching step size according to the gradient. Still, this is likely to fail for extreme pathological surfaces and volatile gradients. If the surface is unintentionally penetrated, and given that the object has not been skipped at all, it is still possible to revert to the last step and proceed again conservatively.

Additionally, we offer an interface to manually decelerate the ray marching process when approaching surfaces. Such a user interaction should be required rarely and only for extreme deformations. We experienced good results by progressively decelerating sphere tracing when approaching surfaces. This allows for marching with larger step sizes far away from volumes, and then proceeding carefully only near the wrinkles of a pathological surface.

3.4.3 Manipulating Objects

Manipulators are convenient and powerful visual tools to interactively alter objects and control transformations. Our modeling tool offers the functionality of common and established

Listing 3.1: Fragment Shader Source Example

```
1  float sp(vec3 p, float r)
2  {
3    return length(p) - r;
4  }
5
6  float box(vec3 p, vec3 dim)
7  {
8    p = abs(p) - dim;
9    return max(max(p.x, p.y), p.z);
10 }
11
12 uniform float alpha;
13 uniform vec2 a;
14
15 float twist(vec3 p)
16 {
17   float new_x = p.x * cos(a.x*p.y + a.y)
18     - p.z * sin(a.x*p.y + a.y);
19
20   p.z = p.x * sin(a.x*p.y + a.y)
21     + p.z * cos(a.x*p.y + a.y);
22
23   p.x = new_x;
24
25   return box(p, vec3(0.5, 1.5, 0.5))
26     * (1.0-alpha) + sp(p, 1.0) * alpha;
27 }
28
29 float phi(vec3 p)
30 {
31   return twist(p);
32 }
```

manipulators, known from professional traditional modeling environments, to work on SDFs. This makes our environment as intuitive and convenient as possible.

Primitives and transformations possess particular, individual properties. For instance, an ordinary sphere has a radius, and a rigid body transformation holds both a translation vector and a rotation quaternion. These properties are not numerically fixed within the fragment shader; instead, they are represented by corresponding uniform variables. Thus, we can modify many scene parameters without regenerating and recompiling the shader.

We propose a three-way mapping between properties, their corresponding uniform variables, and the state of appropriate interactive manipulators. Rigid body transformations, to give an example, are mapped to arrows and disks, for translations along and rotations around particular axes. During manipulation, the state of the manipulator is interpreted and the value of its mapped property is updated. The associated uniform values in the fragment shader are updated upon changes. In this way, rendering a single frame is sufficient to reflect the current state, which is crucial for interactivity.

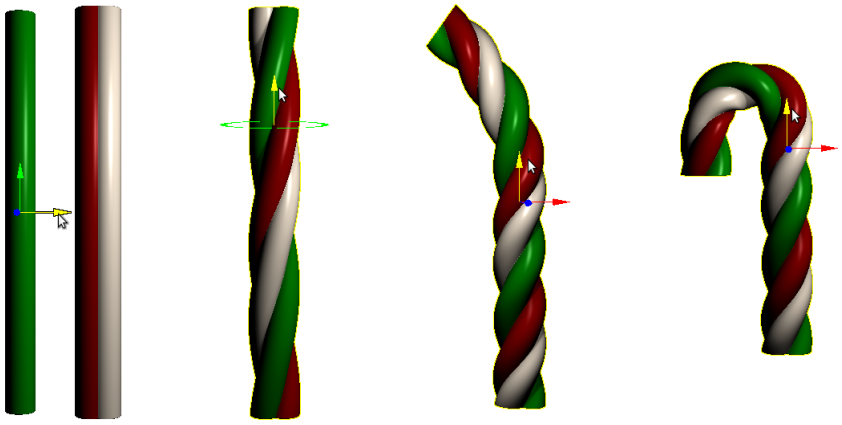


Figure 3.5: *Modeling a candy cane using manipulators for translating, twisting, and bending.*

Figure 3.5 shows a small example, where manipulators are used to model a candy cane. The user starts by placing three cylinders using translation manipulators. Then, joint twisting of these cylinders begins by dragging up a twisting manipulator. Eventually, a candy cane is formed through bending the object.

Our scene management allows to manipulate prior transformations which are now anchored further down inside the hierarchical scene structure. In the previous example, the user is still able to modify the twist parameter in real-time *after* bending the candy cane. The changes are reflected immediately in the final object.

In addition to manipulators, our modeling environment offers a property–value editor. This interface enables manual adjustments and fine-tuning of properties. It further allows to modify certain properties that are not covered by manipulators.

Lastly, an experienced programmer is given the opportunity to directly edit the generated fragment shader source. This provides a wide-ranging freedom of expression to define new types of objects, e.g., fractal shapes such as the Menger sponge (see Figure 3.7).

3.5 Implementation

Our fully functional interactive modeling environment, based on the concepts mentioned before, is written in C++ using Qt 4.7, OpenGL, and GLSL. A few implementation details follow.

3.5.1 Interactive Scene Modeling

One of our integral components for interactive modeling is a fragment shader that is generated by translating the scene graph into a composition of GLSL code fragments. These combined code fragments represent all individual SDFs, operations, and transformations of a scene. Then we append the code for sphere tracing and shading, which is used for real-time rendering.

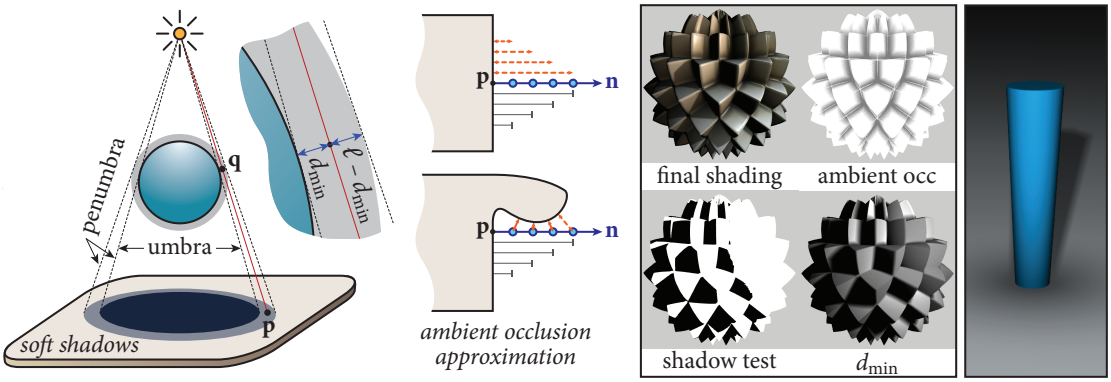


Figure 3.6: Rendering soft shadows: *a shadow ray spawns at* **p** *and ray marching begins. At* **q** *we have reached the minimum distance* $d_{\min}$ *to an occluder. As* $0 < d_{\min} < \ell$, *point* **p** *is in the penumbra.* Ambient occlusion approximation: *the SDF is sampled along the normal* **n** *at a surface point* **p**. *For points on open convex regions, it returns the same value as the distance along the normal, whereas in occluded concave regions, it yields smaller values.* Right: *examples.*

3.5.2 Hybrid Rendering

An interactive modeling environment provides more than a simple view into the scene. Users are able to interact with the application, to select objects, which appear highlighted then, and to manipulate these objects. All GUI elements in our application are rendered with standard OpenGL and overlayed after ray marching. When combining ray marching with rasterization, everything has to merge seamlessly. Thus, the OpenGL modelview-projection matrices are set according to the camera specified for ray marching. The grid is rasterized with enabled depth test against the scene. For picking and highlighting selected objects we use an additional buffer, containing unique IDs for every visible object.

3.5.3 Shading

Surface points are shaded using the gradient as a normal vector, estimated via central differences. We cast shadow rays to each light source, and the shadow test is performed using sphere tracing again. We can also render soft shadow effects easily: we keep track of the minimum evaluated distance $d_{\min}$ to any object during ray marching along a shadow ray (Figure 3.6). For small distances, $0 < d_{\min} < \ell$, we assume that the surface point is located in the penumbra.

This means that for every occluder we also have an imaginary extended occluder where the surface is displaced by ℓ. In the example image, the occluder is a sphere with radius r and the imaginary occluder is a larger sphere with radius $r + \ell$. The penumbra region extends for an increasing ℓ, and $d_{\min}/\ell \in (0,1)$ yields a shadowing factor. Using the `smoothstep` function, which is based on Hermite interpolation, we can soften the linear kinks at the borders of the penumbra region. This avoids visually perceived overshooting artifacts.

Our technique is akin to Parker et al. [1998] and Brabec and Seidel [2002]. However, instead of having to compute auxiliary distances, we already obtained them during ray marching.

Moreover, using SDFs allows us to increase the shading quality by approximating ambient occlusion. The concept is described by Evans [2006] and also illustrated in Figure 3.6. For a surface point $\mathbf{p}$, the distance function of the entire scene is sampled at n points along the surface normal $\mathbf{n}$; few points (like $n = 4$ in the illustration) are sufficient. The distance between adjacent sample points is δ, i.e., the distance between $\mathbf{p}$ and the i-th sampling point is $i \cdot \delta$. Therefore, the location in space of the i-th sampling point is $\mathbf{p} + i \cdot \delta \cdot \mathbf{n}$. An ambient occlusion factor *ao* can then be determined by comparing the distance of the sampling points to $\mathbf{p}$ to the evaluations of the distance function ϕ at the location of the samples as follows:

$$ao(\mathbf{p}) = 1 - k_1 \cdot \sum_{i=1}^{n} \frac{1}{2^i} (i \cdot \delta - \phi(\mathbf{p} + k_2 \cdot i \cdot \delta \cdot \mathbf{n})) \,,$$

where k_1 and k_2 are parameters for fine-tuning. The exponential decay ensures that surfaces further away account less than nearby ones.

We produce plausible soft shadows and ambient occlusion with negligible further cost. They provide important visual cues for modeling and support the user in better perceiving objects and their interrelations [*Luft et al.* 2006; *Ritschel et al.* 2008; *Eisemann et al.* 2011].

3.5.4 Optimizations

With our modeling environment we are able to create complex, large scenes. These, however, are costly to render as many objects contribute to the distance evaluation during ray marching. Bounding volume hierarchies are well suited to reduce the rendering cost in this case. For a sophisticated and more complex object, an SDF may safely return the distance to a bounding box first, given that points in question are outside. Moreover, ray marching is only necessary inside a bounding box. This approach will be important in Chapter 4.

3.5.5 Interface to Polygonal Meshing

Any part of the scene modeled with SDFs can be exported as a triangle mesh. Our environment uses the 3D surface mesh generator of CGAL [*Rineau and Yvinec* 2010], the *Computational Geometry Algorithms Library*. Vice versa, we can integrate triangle meshes into our environment. For this, we transform a mesh into a discrete distance field stored as a regular 3D grid of floating point values. Such objects seamlessly integrate into our modeling framework.

3.6 Results and Analysis

It is impossible to claim that implicit descriptions are generally more appropriate for surface modeling than other established approaches, such as mesh-based, point-based, or voxel-based representations. In the following, we discuss examples where the appropriateness varies from case to case. Moreover, we provide a detailed report on the performance of our system, and discuss the limitations of our approach.

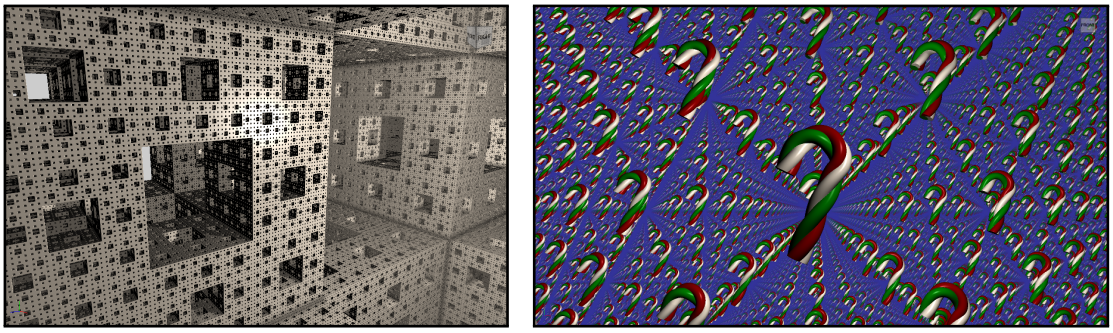


Figure 3.7: *Two examples of exploiting the modulo operator.* Left: *implicitly defined Menger sponge at iteration 5.* Right: *infinite replications of the candy cane over the entire 3D space.*

3.6.1 Examples

Our approach is well suited to model organic structures and natural objects as in Figure 3.3. These objects feature noisy and bumpy surfaces with smooth displacements.

Figure 3.7 shows a Menger sponge constructed using SDFs. We start with a single box and carve out an infinite pattern of rectangular cuboids using the modulo operator. This implicit approach is completely different from constructing a mesh. Our implicit definition is easy to grasp, whereas mesh construction can be tricky and involved. Still, neither technique is substantially more practical.

Modeling a coffee mug is a popular choice to demonstrate CSG abilities. For the artist, the modeling process using our environment is the same as with traditional mesh-based tools. Internally, however, it is less effort and much easier for us to generate and maintain a proper representation. The mug also features smooth continuous surfaces naturally (Figure 3.8).

3.6.2 Performance

Next, we evaluate the performance of our environment on an Intel Core i7-860 system, equipped with an Nvidia GeForce GTX 470 graphics card. The main modeling viewport has a resolution of 1280×1024, and full illumination and shading is enabled.

We obtain a frame rate higher than 400 fps for a single primitive, e.g., a sphere or a box, roughly spanning the entire viewport. We see frame rates of more than 75 fps for the torus depicted in Figure 3.3, displaced by layers of procedural noise functions, and the coffee mug (Figure 3.8). A single candy cane (Figure 3.5) renders with 170 fps. Applying the modulo operator to its domain, as depicted in Figure 3.7, produces infinite replications. With disabled shadows, we still obtain an interactive frame rate of 16 fps for the scene.

A Menger sponge at iteration 0 renders with 220 fps for viewport filling views as depicted in Figure 3.7. The frame rate decreases for increasing iterations: about 83 fps for one, 42 fps for two, and eventually, 18 fps for five iterations. Note that image order rendering, such as sphere tracing, is directly dependent on the viewport resolution; we obtain more than 60 fps in 640×480 resolution for the iteration 5 Menger sponge.

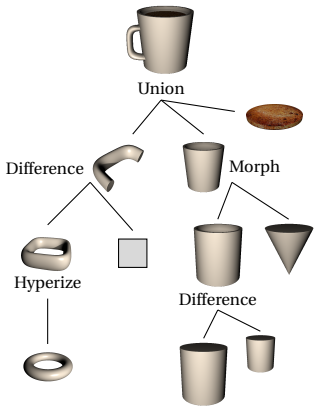


Figure 3.8: *Coffee mug modeled using SDFs and constructive solid geometry.*

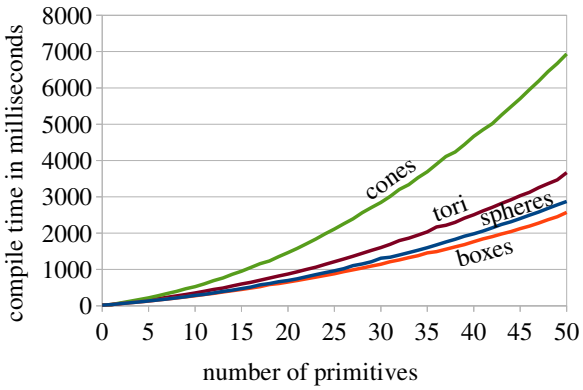


Figure 3.9: *Compiling the fragment shader takes more time for an increasing amount of objects.*

3.6.3 Shader Compilation

The more objects are added to the scene, the longer the compile time for the fragment shader takes. We do not meet a problem for rather plain scenes, as they compile within a few milliseconds, but as scene complexity increases, compile time does so as well. Figure 3.9 shows how long it takes to compile the fragment shader for increasing numbers of different types of primitives. The plain textual generation of the shader itself is negligible.

Using `#pragma optimize(off)` speeds up GLSL shader compilation in trade for rendering performance. This helps to keep up interactivity and responsiveness, while an optimized version is compiled in the background in a separate CPU thread. Note that optimization results vary heavily dependent on the graphics hardware vendor, driver, and operating system.

3.7 Limitations

Rendering performance becomes poor for dense fields full of very detailed and filigree structures, such as blades of grass. Sphere tracing forfeits its efficiency when marching through these areas. However, indirect visualization techniques struggle with these cases as well, since they require unreasonably high-resolution samplings to remain artifact-free.

It is possible that we are no longer able to display more complex objects with sophisticated surface deformations interactively. This is why we address the caching of complex implicit functions for interactive procedural modeling in the next chapter.

Up until now, our modeling environment is unable to provide a convenient interface for sculpting. While previous work (discussed in Section 3.2) shows that it is of course possible to sculpt convincing and sophisticated objects with implicit surfaces, this is clearly the domain where flexible explicit representations, e.g., based on voxels or meshes, perform best. This is why we are going to extend our modeling environment to support voxel-based representations in Chapter 5. There we will implement novel sculpting brushes in the context of 3D printing.

3.8 Discussion

We demonstrated how compositions of signed distance functions can be constructed and used to implicitly represent complex objects and whole scenes. An intuitive and fully functional modeling environment based on SDFs was presented, enabling artists to create appealing scenes without having to deal with the novel underlying data structures.

We showed that interactive modeling and rendering based on SDFs is possible. Neither intermediate steps for indirect visualization, such as using a polygonizer, nor explicit discrete grids are required. This avoids geometric aliasing. Instead, the scene is directly rendered using sphere tracing at interactive frame rates.

However, by adding more and more visual complexity to a scene, we will eventually reach a point where surfaces can no longer be evaluated in real-time. The next chapter tackles this problem by storing the evaluations of complex functions in a cache directly on graphics hardware. We will access this cache using a novel hashing technique. We will show how to extend our modeling environment with this cache to enable the real-time modeling of implicit shapes with more complex surface deformations.

4

A Runtime Cache for Interactive Procedural Modeling

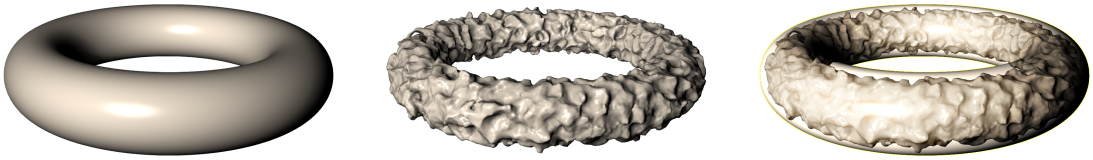
We present an efficient runtime cache to accelerate the display of procedurally displaced and textured implicit surfaces, exploiting spatio-temporal coherence between consecutive frames. We cache evaluations of implicit textures covering a conceptually infinite space. Rotating objects, zooming onto surfaces, and locally deforming shapes now requires minor cache updates per frame and benefits from mostly cached values, avoiding expensive re-evaluations. A novel parallel hashing scheme supports arbitrarily large data records and allows for an automated deletion policy: new information may evict information no longer required from the cache, resulting in an efficient usage. This sets our solution apart from previous caching techniques, which do not dynamically adapt to view changes and interactive shape modifications. We provide a thorough analysis on cache behavior for different procedural noise functions to displace implicit base shapes, during typical modeling operations.

4.1 Introduction

One of the core qualities of procedural modeling is to enable programmers and artists to pass the modeling task of creating rich *visual complexity* on to the computer (see Chapter 2). Ideally, this results in realistic scenes, naturally unfolding up to unlimited detail, starting from very small descriptions [*Ebert et al.* 2002].

In the previous chapter we have discussed implicit surfaces, which are of particular interest: unlike explicit descriptions, they can be evaluated *on demand*, and are naturally amenable to modeling techniques such as constructive solid geometry and blending. This is why implicit shapes are suited to represent both handcrafted, man-made objects, and organic, soft objects.

However, we saw that implicit surface descriptions have two main drawbacks: first, the outcome of a shape is not as easy to control as, for instance, it is for explicit meshes. There is a large body of previous research that proposed various modeling approaches to regain control. We already gave an overview of the most important techniques in the previous chapter. We will briefly address some of these techniques again in the next section, this time with a focus on *caching*. Second, it is more difficult to render implicitly described scenes in real-time than it is for explicit representations. These were the two main challenges during our implementation of an implicit surface modeling environment (Section 3.5).



(a) *Implicit base shape* **(b)** *with textured surface* **(c)** *base shape as bounding shell*

Figure 4.1: *We assume that distance functions of implicit base shapes are fast to evaluate. Only the texture part, by contrast, is expensive, and will be cached. The base shape further acts as a bounding volume, and texture is cached only within this shell. We think of texture mainly as procedural displacements (surface texture in a literal sense) and also include color information.*

We use the *sphere tracing* technique [*Hart* 1996] to render shapes described by *signed distance functions* (SDFs) efficiently. Figure 4.1a shows a torus, which, together with other implicit base shapes, can be easily described with an SDF and directly rendered on the GPU with sphere tracing. To create more complex and interesting shapes, we apply procedural displacement textures (Figure 4.1b) to these base shapes. We require these fine-scale details to create rich visual complexity. Most recent and powerful procedural texturing methods are based on spot noises, obtained by summing a large number of kernels positioned randomly in space [*Lagae et al.* 2010]. Alas, evaluating complex noise functions is very expensive, which is a problem for real-time rendering.

For this reason, we propose a technique for caching evaluations of implicit solid textures. Instead of relying on a fixed, spatially limited grid, our cache adapts to the view conditions and uses available memory where it is most useful. Since simple base shapes are sufficiently fast to evaluate, we cache only the texture part within a bounding volume (Figure 4.1c). The whole process happens lazily and seamlessly during rendering, which integrates smoothly into the on-demand evaluation of implicit procedural descriptions. This allows the user to model shapes without invalidating the cache, while the texture details are efficiently rendered.

Contributions. We propose a novel cache mechanism equivalent to a sparse grid spanning the entire, infinite 3D texture space. In combination with our previous implicit modeling approach, this allows us to interactively create procedural shapes with more complex surface deformations. Our cache mechanism is inspired by recent real-time hashing schemes which we revisit. Key–data pairs are hashed and then inserted in the cache, where new data evicts old data. This is why we do not have to explicitly implement a least-recently-used policy. Our method efficiently allocates blocks of data during insertions. This allows us to exploit the native trilinear interpolation available on graphics hardware, reconstructing interpolated instead of constant texture values per cell. Our scheme is easy to use and implement: it does not require a complex communication between the CPU and GPU and it fits entirely within OpenGL shaders. The technical implementation has no impact on the procedural functions, i.e., we do not have to augment or modify our implicit descriptions in order to use our cache. We also provide a thorough analysis of the cache behavior for a variety of modeling examples.

4.2 Previous Work

As mentioned before, implicit surfaces are amenable to ray tracing. However, in the past (the days before programmable graphics hardware was available) this was far from being feasible in real-time. Instead, we had to resort to indirect visualization techniques for rendering, using intermediate, explicit representations generated by a polygonizer. Specialized techniques have also been proposed: Loop and Blinn [2006] achieved real-time rendering on the GPU by assembling fourth-order piecewise algebraic surfaces (introduced by Sederberg [1985]). Interesting level-of-detail (LOD) techniques have been developed by Hornus et al. [2003].

Distance fields can be used to store explicit distance values, which are then simply queried instead of evaluated. However, they consume a large amount of memory; they can be stored hierarchically using *adaptive distance fields* (ADF) [*Frisken et al.* 2000], significantly reducing memory consumption. Rendering ADFs can be performed efficiently on the GPU, as proposed by Bastos and Celes [2008]: the field is stored into a perfect spatial hash [*Lefebvre and Hoppe* 2006], a static data structure efficiently accessed by the GPU (see also Section 4.3). However, and contrary to our approach, their data structure is fixed and suits only for interactive rendering, not modeling, as it cannot adapt to interactive surface deformations.

Schmidt et al. [2005b] presented ShapeShop, an interactive, sketch-based modeling environment for implicit surfaces. They achieve interactivity by caching the scalar fields in dense grids [*Schmidt et al.* 2005a]. *Caching nodes* are manually inserted into the BlobTree [*Wyvill et al.* 1999]. This significantly increases rendering performance (by an order of magnitude at that time) at the expense of a large memory consumption. Our method addresses this issue and allows to specify the amount of memory available for caching, while providing similar functionality. We further exploit that a given view uses only a small fraction of the total data.

Finally, the Gigavoxel framework [*Crassin et al.* 2009] could be used for caching distance fields. Gigavoxel is a hierarchical data structure which can be efficiently traversed by rays. Rays stop when encountering missing data. This data is then produced before ray tracing resumes. In contrast, our scheme exploits the fact that the data is implicit, seamlessly balancing between computation and memory storage. In particular we can render frames which would not fit entirely within the cache. Contrary to Gigavoxel, our scheme does not require maintenance of a pointer-based hierarchy, resulting in a simpler implementation. Our scheme also exploits the hash eviction mechanism (see Section 4.3) to avoid implementing an explicit least-recently-used policy, removing entirely the need for tracking data usage at every frame.

4.3 Background on Parallel Spatial Hashing

Our method is inspired by recent work on parallel spatial hashing. Such a hashing scheme works with key–data pairs. A key is a unique identifier for the data, typically the (x, y, z) coordinates. Keys are used to insert and retrieve data from the *hash table*, which stores all key–data pairs. The hash algorithm computes the location within the hash table from the key.

Lefebvre and Hoppe [2006] proposed a hashing scheme for computer graphics to store texture data around a surface. While access to the data is efficient, requiring two memory accesses and one addition, the construction process is sequential and much slower.

Alcantara et al. [2009] proposed a parallel hash construction inspired by Cuckoo hashing. Keys can be inserted in a fixed number of locations (typically, four) within the table. Each thread inserts a key, evicting the previously stored key. The thread is then responsible for inserting the evicted key to its next location. This process continues until the last evicted key finds an empty location. All threads insert in parallel and compete for empty slots. Using four locations, the process terminates with high probability if the table is less than 90% full. While construction is much faster, access now requires to test all possible locations for the key. This however remains very fast on modern hardware.

García et al. [2011] improved the construction and access performance by relying on a Robin Hood strategy. Keys now have a full sequence of possible insertion locations. The *age* of a key is the position along the sequence which was used to insert the key. During insertion, keys can only be evicted by older keys: young keys make room for older ones. This allows to fill the table at a higher density without hindering access performance. The insertion sequence also preserves memory coherence, greatly improving performance for coherent access patterns.

Our scheme is based on the work of Alcantara et al. [2009; 2011] and García et al. [2011], but with several modifications: first, we allow for keys to stream in and out of the cache at every frame during rendering. Keys with new data evict keys with older data. Second, our universe is now unbounded: coordinate records are not limited to 32 or 64 bits. Finally, we introduce an allocation and indirection scheme to avoid relocating data around the table when keys are evicted. This reduces the required bandwidth significantly for large data records.

4.4 Overview

Our implicit surfaces are defined as the zero-crossing of real-valued scalar fields. We assume that these scalar fields are obtained from functions which are Lipschitz continuous: there exists a constant Λ so that $|x-y| < \delta \Rightarrow |f(x) - f(y)| < \Lambda\delta$.

We again note ϕ the function defining the surface as in Subsection 3.3.1, and t the function defining the texture. The textured shape is obtained as $\phi_{\text{final}}(\mathbf{p}) = \phi(\mathbf{p}) + \alpha\ t(\mathrm{T}\,\mathbf{p})$, with $\mathbf{p} \in \mathbb{R}^3$ a point in space, α a weight, and T a transformation from world to texture space. The texture function has a bounded influence: There exists M so that for all $\mathbf{p}, |t(\mathbf{p})| \leq M$. Therefore we do not evaluate t when sufficiently far away from the surface since it has no influence there. Throughout this chapter, we assume that t is much more expensive to compute than ϕ.

The sphere tracing pseudo code we execute for every pixel on the image plane is given in Listing 4.1. In this algorithm, `Eye` is the current camera position, `ray_direction` is the direction of the ray from the camera through the pixel center, `epsilon` controls the accuracy of the ray-surface intersection, `step_size` avoids over-stepping and depends on the Lipschitz property of the function, `FarClip` is the maximum distance after which we stop marching and `BkgColor` is the background screen color. Finally, `cached_t` is the cached texture function. If the cache has a sufficient resolution, this is equivalent to calling `t(p)` directly, but results in better performance through reuse of previous computations.

Our goal is to cache the expensive evaluation of t so that the overall rendering time is decreased. Throughout modeling the user may modify ϕ, or may change α and T: these

Listing 4.1: Sphere Tracing Pseudo Code

```
step = Infinity;
p    = Eye;
l    = 0;

while (step > epsilon)
{
  if (outside_coarse_shell)
  {
    // evaluate distance to bounding volume...
    eval = phi(p); // ...which is the base shape
  }
  else
  {
    // inside bounding shell, evaluate also texture
    eval = phi(p) + alpha * cached_t(T * p);
  }

  // march evaluated distance along ray
  step = eval * step_size;
  p = p + step * ray_direction;
  l = l + step;

  if (l > FarClip)
    pixelColor = BkgColor;
}

pixelColor = shade(p);
```

operations *do not* invalidate the cache. If the user changes the overall procedural description of t, however, the cache has to be discarded.

Each time `cached_t` is called, our cache is queried for data. The cache can be imagined as a uniform grid covering the entire texture space, with sparsely filled cells storing information. This produces a discretized, trilinearly interpolated version of t (Section 4.6.1). The cache cells have to be small enough to capture the details of t properly. In our implementation, the user has to specify the cell size manually; we use $(1/192)^3$ in all our examples, a value we determined to be a good trade-off between performance, visual quality, and memory consumption.

For each query, we check whether data is available in the cache. If data is available, we return it directly. If not, we *compute* the new data and insert it into the cache. In our implementation, we insert only the last cache miss encountered along a ray. This limits the number of insertions to one per pixel per frame and avoids long processing times. While rendering the next frame, this data will be available with high probability and the cache miss is unlikely to occur again.

Depending on the current view, the inserted data point may evict older data. It might also fail to insert, as we allow only a single new data point to be inserted per pixel and per frame. Note that even if a data point fails to insert, it will always have another chance at the next frame. We exploit this observation to obtain a simpler, more efficient algorithm. We have implemented everything in OpenGL shaders to avoid slow CPU/GPU communication.

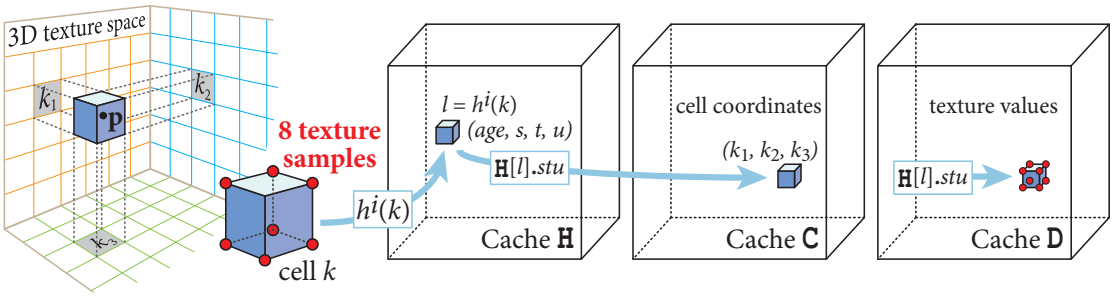


Figure 4.2: Data structure. *Point* **p** *is located inside cell k in texture space. Cell k has the integer coordinates* (k_1, k_2, k_3). *We store this cell with age i inside cache* `H`. *The location l is determined by the hash function* $h^i(k)$. *Cache* `H` *stores the age of the cell as well as stu coordinates. They are used to access cache* `C` *(which stores* k_1, k_2, k_3*) and* `D` *(which stores the texture data for cell k).*

4.5 Cache Mechanism

Rendering for each frame is performed in three steps:

1. **Raymarch:** for each pixel the intersection (if any) is computed, as well as the position of the *last* cache miss (if any). During ray marching the cache is read only.
2. **Insert:** a new slot is reserved in the hash table for each pixel which had a cache miss.
3. **Produce:** the data for each newly inserted key is evaluated and stored in the cache.

Figure 4.2 shows an overview of our data structure. We discretize the 3D texture space so that every point **p** is located inside a cell k. We use the integer coordinates (k_1, k_2, k_3) of such a cell as the key for a hash function to access the cache.

The hash table stores information about each key: the age of the key and the 3D coordinates of the cell. It also stores the data associated with the key, which is the eight texture values evaluated at the corners of the cell. All this information is stored in different tables: `C` stores the spatial coordinates of the cells, `D` stores the data records (the texture values), and `H` stores the age of the keys as well as the location of their data in the other two tables. All tables are 3D arrays with the same size. The cumulative size of the tables is the amount of memory allocated for the cache. This is independent from the resolution of the texture space discretization.

Using a cell k as the key with age i, the hash computes a location $l = h^i(k)$. Information about this key can be found at `H`$[l]$. The spatial coordinates of the cell are stored at `C`[`H`$[l]$.*stu*] and the texture values at `D`[`H`$[l]$.*stu*], where (s, t, u) is the location of the records in `C` and `D`. This indirection scheme allows us to move around keys in `H` without having to update the other tables `C` and `D`. This is a unique advantage of our hashing scheme.

With this mechanism we are now able to implement the `cached_t` function (Listing 4.2). The function `texture2cell` computes which cell of the cache grid encloses **p**. It maps continuous locations to integer coordinates and thereby defines the cache resolution. Note that *resolution* in this context refers to the size of a cache cell in space. In other words, the cache resolution is the density of the grid we use to discretize the 3D texture space. The spatial extent of the cache itself is unbounded. `LastCacheMiss` is later used to insert data.

Listing 4.2: Pseudo Code for cached_t

```
1  cached_t(p, H, C, D)
2  {
3    k = texture2cell(p);
4    d = access(k, H, C, D);
5    if (d == null)
6    {
7      // cache fault, record for insertion
8      LastCacheMiss = p;
9
10     // compute the value instead
11     return t(p);
12   }
13   else
14   {
15     // data is in cache
16     return d;
17   }
18 }
```

4.5.1 Cache Access

Our hash follows the open addressing scheme of García et al. [2011]: each key k is associated with a sequence of possible insertion locations noted $h^i(k)$. The insertion algorithm (discussed in the following Subsection 4.5.2) ensures that the maximum insertion age is bounded: if a key is present in the table, then it is at location $h^i(k)$ with $i < A$. A is called the *maximum age* for the table. In our implementation, it is experimentally determined to a value of 4. Retrieving a key simply amounts to walking along the sequence $h^i(k)$, until either the key is found, or the maximum age is reached. The pseudo code for retrieving a key (Listing 4.3) is given below:

Listing 4.3: Pseudo Code for access

```
1  access(k, H, C, D)
2  {
3    for (int i = 0; i < A; i++)
4    {
5      p_stored = C[ h(k, i) ];
6
7      if (texture2cell(p_stored) == k)
8      {
9        return D[ H[ h(k, i) ].stu ];
10     }
11   }
12
13   // key not in hash
14   return null;
15 }
```

The hash function h is the coherent hash function of García et al. [2011], which is a random translation added to k and determined by i:

$$h^i(k) = (k + O[i]) \mod N,$$

where O is a precomputed table of random offsets and N the size of the hash table. All these quantities are 3D vectors, and vector operations are performed independently on x, y, z.

The coherent hash function significantly improves access performance over randomizing functions [*García et al.* 2011]. Note also that keys with a younger age are faster to retrieve.

4.5.2 Inserting Keys

The insertion algorithm is the heart of our method. Our goal is quite different from a typical hashing scheme, and the following paragraphs emphasize these differences.

Cache policy

While the hash table is empty at the start of a modeling session, it fills up with incoming data quickly. Once full, new data should still be accepted, but we have to erase some older data first. We also want the histogram of ages to be favorably biased toward recently inserted data: it is more likely to be useful for the next views. Younger keys also lead to faster access.

Existing spatial hashing schemes are meant to build a hash table from a predetermined set of keys, seeking to fill the hash table entirely (García et al. report fill rates of up to 99%). Once the table is full, *all* further insertions fail. This implies that our cache could never be updated beyond this point. A possible approach would be to implement an explicit least-recently-used strategy, keeping track of which keys are accessed every frame. However, this requires a lot of bandwidth during rendering: each thread needs to write in the hash at every access. Instead, we would like to have the access to be very fast, so that the cache is overall beneficial.

Our first intuition was to exploit a modified Robin Hood policy, comparing the ages of keys. However, a policy inspired by Cuckoo hashing provides a simpler and better answer. Similarly to Cuckoo hashing, we blindly evict keys. A new key therefore always evicts a key already in the hash. The evicted key is tested for its next location, but is only allowed to evict a key which was also already in the hash (i.e., not inserted within the current frame). We limit the number of iterations, so that only a few successful evictions can occur before the key falls out of the cache. This process is illustrated in Figure 4.3. In addition, any key reaching the maximum age automatically falls out of the cache. The result is an increased probability of falling out of the cache for older keys, as revealed by the histograms in Figure 4.9.

Data storage

Another important technical issue is that existing schemes store the data along the keys. This is convenient, since keys and data can be stored simultaneously in 64 bits words: a *single* atomic operation *both* evicts and inserts the key–data records (`atomicExchange` for Alcanatara et al., `atomicMax` for García et al.). These algorithms are designed around this idea: without atomic operations, a value should first be read, tested, and only then would the hash table be

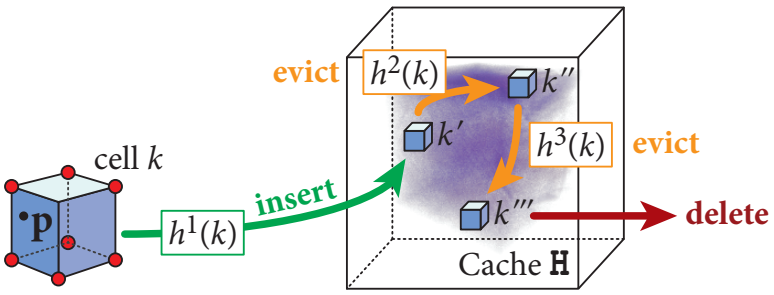


Figure 4.3: Insert example: *point* **p** *generated a cache miss. It is located inside cell k. We insert this cell at age* 1 *into the cache* H. *However, the slot is already occupied by k′, which is evicted. This generates a second eviction for k″. The last key for k‴ cannot be reinserted and is deleted.*

written. In-between reading and writing, another thread could change the table, producing inconsistencies. Atomic operations avoid this issue by performing the entire reading and writing process within a single atomic call.

Moreover, in our case, we cannot store the data alongside the keys. We sample the texture function for every corner of a cell. This data, eight texture values, is too large for being stored and moved along with the key, and must be kept in a separate table. Keeping both tables synchronized is similarly difficult: in-between writes to both tables, another thread may change the first table, again generating inconsistencies. We require these large data records for efficient trilinear interpolations to reconstruct the texture values (see Section 4.6.1).

We introduce a new hash algorithm which avoids having to move data records around, while ensuring that the data structure is consistent at all times. In our scheme, the data of a key k is the eight texture values at the corners of cell k. This data remains stored at the same location in table D, as long as the key is in the hash and regardless of the number of times the key moves within H due to evictions. This location, noted (s,t,u), is stored along the age of the key in H.

An additional benefit is that our keys are no longer limited to coordinates fitting within 32 bits: we can store the coordinates in another table C, again at the (s,t,u) location. We thus have 32 bits coordinate *triples* (96 bits) available to cover the unbounded 3D texture space.

Duplicated keys

Neighboring pixels are likely to generate similar cache misses, and will therefore try to insert the same keys simultaneously into the hash. Our insertion mechanism handles these cases well and ensures that all keys stay unique. We now discuss this mechanism in more detail.

Algorithm

The algorithm takes the following parameters: p is the coordinates of the insertion point, d the data to be stored, H is the hash table, C stores the (k_1,k_2,k_3) coordinates of the inserted cell k which encloses p, and D stores the texture data. H is a table of 32 bits storing the age (4 bits) and the data coordinates stu (24 bits). The remaining 4 bits are unused.

All new keys detected within the frame are inserted in parallel, each thread running the insertion algorithm. We flag keys in the table as "new" by setting their `stu` coordinate to null. After insertion, we return the location of the key in `H` along with the `stu` coordinates for its data in `C` and `D`. Note that the `stu` record of the key in `H` is *not* updated immediately. All insertions for the frame must terminate first. This synchronization ensures that if other insertion threads are still running, they still detect the key as new (`stu == null`).

Listing 4.4 shows the pseudo code to insert a single key. We now describe each step of the algorithm and provide a line-by-line walkthrough. Please keep in mind that any key which fails to insert, or falls out of the cache, always has another chance at the next frame if it is still used. This is why we reject keys aggressively to keep the algorithm simple.

Line 2–3: `inserted_l` records where the new key will be inserted in `H`. `inserted_stu` records which slot will be allocated for the data of the new key in `C` and `D`.

Line 4–6: `k` is the key. We use the coordinates of the cell enclosing **p** in texture space as the key. Since the insertion request was generated on a cache miss, `k` is *new:* it is not in the hash when the insertions start. `key_info` is a 32 bits bit-field storing the age of the key being inserted and its `stu` coordinates. `stu` is null since it is unknown for the new key.

Line 7–8: The insertion algorithm runs for only a maximum number of iterations described by the maximum age A. Any key still not inserted at the end falls out of the cache (deletion). This always happens once the cache is full: an old key must be deleted to make room for a new key.

Line 9: An insertion location `l` is computed from the key coordinates and its age. The age is incremented in every iteration at line 44.

Line 10: The hash table is read at location `l`, and the result is stored in `prev`.

Line 11: We test whether the key can be inserted at the current location. It is the case if the current slot is empty, or if it is occupied by a key that can be evicted. We do *not* allow the eviction of empty keys (`stu == null`). The main reason is that unless an empty slot is found, we will give the `stu` of the last evicted key to the new key. Therefore, only keys with a valid `stu` can be safely evicted. Section 4.5.3 discusses this in more detail.

Line 13–14: The key is tentatively written in `H` line 13, using an atomic operation. The atomic compare and swap (CAS) operation only writes `key_info` if the value in `H` is still `prev`. The value read by `atomicCAS` in `H` is returned and tested in line 14. If the value is still the same, the write succeeded: the key which was previously there has been evicted. We will try to reinsert this evicted key in lines 15–30. If the value returned by `atomicCAS` has changed, no write occurred and we have discovered a conflict: in the meantime another thread wrote in `H` since our first read in line 10. We then check whether we must stop in lines 33–35. This is important to avoid duplicated keys, also further explained in Section 4.5.3.

Line 16–18: We record the location where the first (new) key is inserted. We will return this value to the caller at the end of the eviction sequence.

Line 19: In case of a successful insertion, two cases can occur. In the first case we inserted the key at an empty location (lines 20–23) and the insertion terminates. We record the location of the empty slot in `inserted_stu` as the location for the data of the new key. In the second

Listing 4.4: Pseudo Code for insert

```
1  insert(p, d, H, C, D) {
2    inserted_l     = null;
3    inserted_stu   = null;
4    k              = texture2cell(p);
5    key_info.age   = 1;
6    key_info.stu   = null;
7    iter           = 0;
8    while(iter++ < A) {
9      l    = h(k, key_info.age);
10     prev = H[l];
11     if (prev == empty || prev.stu != null) {
12       // evict the key
13       read = atomicCAS( & H[l], prev, key_info);
14       if (read == prev) {
15         // atomic insertion succeeded
16         if (inserted_l == null) {
17           inserted_l = l;
18         }
19         if (read == empty) {
20           // this was an empty slot
21           inserted_stu = l;
22           // we found an empty slot, return
23           break;
24         } else {
25           // a key was evicted, try to re-insert
26           k = texture2cell( C[prev.stu] );
27           key_info.age = prev.age;
28           key_info.stu = prev.stu;
29           inserted_stu = prev.stu;
30         }
31       } else {
32         // conflict during write
33         if (key_info.stu == null) {
34           break;  // exit if inserting a new key
35         }
36       }
37     } else {
38       // a new key concurrently inserted is encountered
39       if (key_info.stu == null) {
40         break; // exit if inserting a new key
41       }
42     }
43     // try next location
44     key_info.age++;
45     if (key_info.age >= A) {
46       break;
47     }
48   }
49   // done, return insertion location and stu
50   return (inserted_l, inserted_stu);
51 }
```

case we inserted the key by evicting another (lines 25–29). We have to re-insert the previous key, and therefore re-initialize insertion. Note that we read the coordinates of the evicted key in `C` (line 26). This is correct since we never evict new keys for which an entry in `C` is not yet available (see line 11). We keep track of the `stu` record of the evicted key in `inserted_stu` (line 29). If the key cannot be reinserted, it will disappear from the cache. We will then reuse its `stu` location for the new key we initially inserted.

Line 32–35: The write in `H` failed due to a concurrent write. We must stop if we are currently inserting the new key. This avoids duplicated insertions as will be explained in Section 4.5.3.

Line 38–41: The current slot in `H` is occupied by a new key inserted in parallel by another thread. We must stop if we are currently inserting the new key to avoid duplicated insertions.

Line 44: The age of the current key is increased. The algorithm will try to insert it at its next location in the next iteration.

Line 45–47: The maximum allowed age is reached for a key: the algorithm stops and the key falls out of the hash table (deletion).

The entire process runs for limited iterations. Many keys will fall out of the hash, but if the cache is large enough and the camera stops moving, the process stabilizes after a few frames.

4.5.3 Correctness

The algorithm has to ensure two important properties: 1) no key appears more than once, despite several threads trying to insert the same key; 2) no `stu` coordinate is used twice.

Avoiding duplicates

Duplicated keys are detected during insertion. If a new key is encountered while the thread is also trying to insert a new key (line 39), this could be a case of duplication. Since the table `C` is not yet updated for new keys, we cannot check the coordinates and stop instead (line 40).

If a write conflict occurs while inserting a new key (line 33), we also have to stop. If the conflict is with a new key, it could be a duplication. If the conflict is with an older key evicted by another thread, we again must stop. Indeed, let us assume that we regardlessly inserted the new key at its next location: if another thread is trying to insert the same key later, it will evict the older key which created the conflict *before* detecting the duplication.

This guarantees that inserted keys are unique. Note that these tests are only necessary for new keys, since keys being evicted are always unique. In any case, the new key which failed to insert will have another chance at the next frame.

Allocation of data slots

The `stu` locations are allocated to new keys based on the following observation: at the end of a full insertion sequence we either 1) have found an empty slot, or 2) removed a key from the cache. Let us assume the new key was successfully inserted (otherwise we had no data to store). In both cases, we are given a slot for storing the data of the new key:

1. If an empty slot is found, we simply set `stu` to the location of the empty slot (line 21).

2. If a key falls out of the hash, we erase its data by reusing its `stu` data location for the data of the new key (line 29).

Note that by construction we are guaranteed that empty slots in H correspond to empty slots in D and C: a key reaching an empty slot in H fills the corresponding locations in H, C, and D. Slots in H are never emptied once occupied. Therefore, slots in C and D cannot be occupied while empty in H. This ensures that we never erase data which is in use.

4.5.4 Producing data

The last step, after ray marching and insertion, is to produce the data associated with the keys. All threads are synchronized before producing the data. This ensures that all insertions terminated. The `stu` coordinates of successfully inserted keys are then written at the key locations in H. Next, the key coordinates are written in C, and their data records are produced in D at their allocated `stu` coordinates. We discuss the content of each data record in Section 4.6.1.

4.6 Implementation

We implemented our method using OpenGL 4.2 shaders. Each of the three steps *raymarch*, *insert*, and *produce* is implemented in its own shader. We perform all three steps in sequence. Temporary results are stored in OpenGL render buffers: `LastCacheMiss` after the *raymarch* step, and `inserted_l`, `inserted_stu` after *insert*. These three passes are necessary to synchronize all threads. We exploit the `shader_image_load_store` extension to write directly into textures when updating the hash and producing data. This extension also provides the `atomicCAS` (atomic compare and swap) operation required by our approach.

4.6.1 Efficient Trilinear Interpolation

An explicit and manual implementation of trilinear interpolation in a shader requires eight accesses to the data structure. In our case this implies eight possible cache misses and eight possible insertions, *for each marching step along the ray*. While this is possible, our tests resulted in very poor performance, much slower than eight times the cost of a single access.

Lefebvre and Hoppe [2006] faced a similar problem for texture interpolation. Their solution was simple and elegant, despite a significant increase in memory usage. Rather than storing single data points, they proposed to store interpolation cells made of 2^3 samples. Since these cells are stored in a volume texture, native hardware interpolation can be used *within a cell*. However, neighboring cells are not stored together, implying a significant overhead since each data point now potentially appears eight times in different interpolation cells. This overhead can be reduced by relying on larger blocks.

We follow the same approach, but in our case the overhead is limited: our cache adapts to the view. We therefore designed our cache algorithm for this blocking scheme and store data in full 2^3 floating point precision blocks. Recall that this is why we use the dedicated cache D.

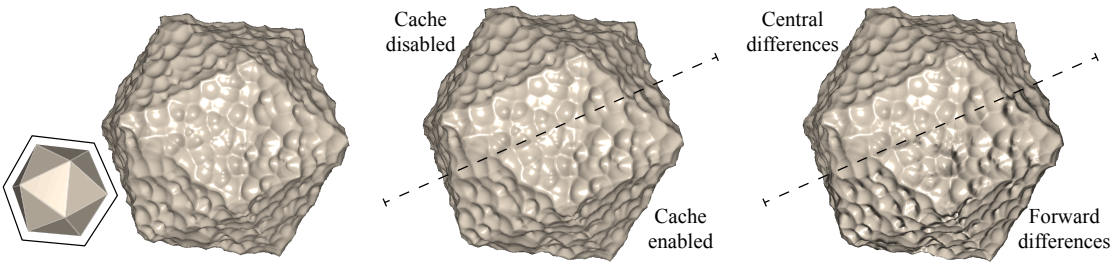


Figure 4.4: *Textured icosahedron (with implicit base shape) used for our analysis (Figures 4.9 to 4.8). We see that using our cache maintains high-quality surfaces. Estimating gradients via central differences yields superior shading results, but is slower than using forward differences.*

During the *produce* step of our algorithm, we compute the eight texture values at the corners of the cache cell enclosing the inserted point (also see Figure 4.2). This implies that each data point is computed up to eight times more often: each data point is located at the center of a $2 \times 2 \times 2$ block of cells in texture space. One possibility to address this overhead would be to maintain a separate cache for single data points. Using larger blocks would also reduce this overhead since only boundary samples would be duplicated.

4.7 Results and Analysis

We now analyze the cache behavior and then present results of interactive modeling sessions.

4.7.1 Setup

In our test environment we display a textured icosahedron (Figure 4.4), filling the viewport, at varying rotation speeds. Implicit base shapes, such as an icosahedron, can be easily obtained using *generalized distance functions* [*Akleman and Chen* 1999]. Because the hash cannot fit the entire data, keys will get inserted and deleted from the cache at every frame. Depending on the rotation speed, more or less cache misses will occur in-between frames. In all technical tests we use the same spot noise: spherical kernels, carving out from the surface, are randomly placed in space. The viewport resolution is 800×800. Unless otherwise specified, the maximum age is $A = 4$. All results were obtained on an Nvidia GTX 580 graphics card with driver version 295.51.

4.7.2 Maximum Age

We investigate the effect of changing the maximum age A. The tradeoff is that a higher maximum age leads to a more efficient insertion, and hence less deletions from the cache, but with slower access. Figure 4.5 shows the behavior of the cache for different values of A. The shape is rotating on the screen so that a similar amount of new data appears at every frame.

Note that the number of deletions is very high for $A = 2$: the cache deletes keys aggressively, among which many are in use for the current view. The overall rendering time suffers from

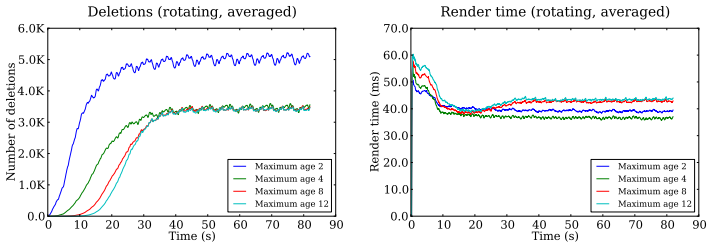


Figure 4.5: *Number of cache deletions* (left) *and rendering time* (right) *for varying maximum ages A.* *(Icosahedron, rotation speed 2 rad/s.)*

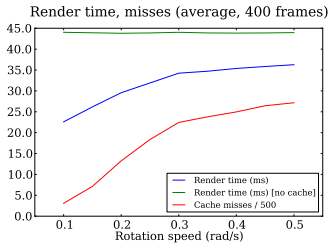


Figure 4.6: *Cache misses and rendering time as a function of rotation speed.*

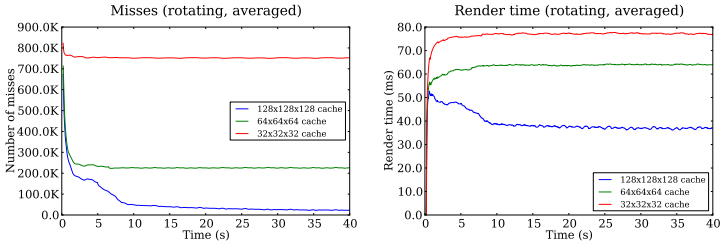


Figure 4.7: *Number of cache misses* (left) *and rendering time* (right) *for a varying hash table size.* *(Icosahedron, rotation speed 2 rad/s.)*

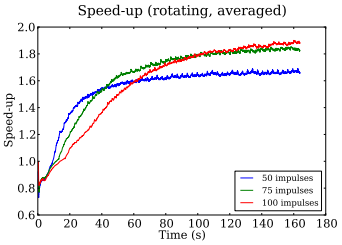


Figure 4.8: *Cache speed-up ratios for different noise complexities.* *(1 rad/s.)*

this missing data. $A = 4$ behaves similarly to $A = 8$ and $A = 12$ after 40 seconds. Before that point it is more difficult to insert keys and it needs more time until stabilization.

The rendering time for $A = 8$ and $A = 12$ is acceptable after 20 seconds, but then slows down after 40 seconds: the cache is able to keep more keys in the hash, but has to push them further away along their sequences, reducing insertion and access performance for little benefit.

In the light of these facts, we experimentally selected $A = 4$ as a good tradeoff.

4.7.3 Hash Table Size

For a fixed cache cell size $(1/192)^3$ in texture space, we analyze the impact of a decreasing hash table size. The rotation speed is fixed to 2 radians per second. The number of cache misses and the render time are given in Figure 4.7. Clearly, a larger hash table size directly benefits rendering by strongly reducing the number of cache misses. The constraint here is essentially how much memory can be allocated by the host. In our implementation we use a hash table size of 128^3, which requires a total size of 80 MB for all tables. Several such caches therefore easily fit on a single GPU, which allows to cache different procedural textures.

4.7.4 Cache Misses and Render Time

We now analyze how the number of cache misses and the rendering time evolves for varying rotation speeds. For a given rotation speed, the values are averaged over 400 frames to allow

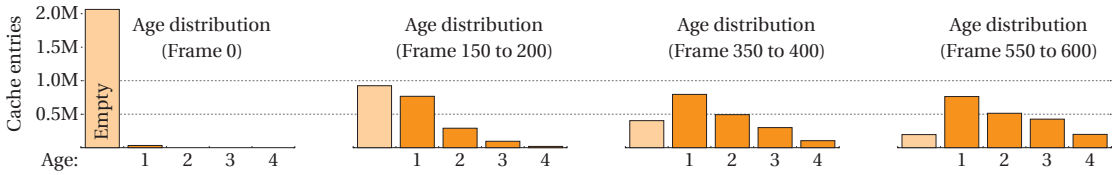


Figure 4.9: *Histograms showing the distribution of the ages of cache entries, after 0, 200, 400, and 600 frames. The histogram stabilizes after this point. Note the bias toward younger keys.*

for the cache to stabilize. Results are given in Figure 4.6. Slower rotations are faster to render, but note how the curves flatten as rotation speed increases: faster rotations do not require to refill the cache entirely, since many keys remain shared between consecutive frames. This is a good property for modeling, where rotating objects is a very common operation.

4.7.5 Procedure Complexity and Cache Benefit

We analyze the benefit of using the cache for different procedural complexities. To this end, we significantly increase the number of impulses of our spot noise. We set the texture weight low so that the shape geometry is not significantly modified. The resulting performance ratio between using the cache or not is shown in Figure 4.8 for a rotation speed of 1 rad/s.

This shows that increased complexity leads to a larger performance ratio. However, this difference depends on the rotation speed. It is maximal for a fixed viewpoint—once everything is in the cache the rendering time is the same—and the difference decreases for larger rotation speeds. If many new keys appear, a complex procedure will again provide less benefit than a simple one due to the large production of data.

In the first few frames the speed-up ratio is below one, since the cache has to be filled with a large number of missing data. Please also note that the speed-up is relatively low due to the rotation—it can be seen in Figure 4.6 that the speed-up is much larger at low rotation speeds.

4.7.6 Histogram of Ages

As explained in Section 4.5.2, one of our objectives is to bias the histogram of keys toward younger keys. Figure 4.9 shows the age distribution of cache entries obtained for a rotation speed of 2 radians per second. We see how the histogram stabilizes and favors younger keys.

4.7.7 Precision and Shading

The cache stores a discretized version of the texture, which is trilinearly interpolated. This results in a slightly different surface when enabling the cache (emphasized in Figure 4.10). Our examples use a cell size of $(1/192)^3$, with the object fitting exactly into the unit cube. In practice we did not find the differences visually disturbing (Figure 4.4, center). Visual popping, however, may occur when keys are deleted or inserted. This could be entirely avoided by artificially limiting the resolution of the analytic function, evaluating it on a lattice with interpolation.

Figure 4.10 visualizes the loss of precision both in screen space and on the object's surface. *Left:* subtle shading differences *(blue)* and silhouette mismatches *(red)* appear in screen space.

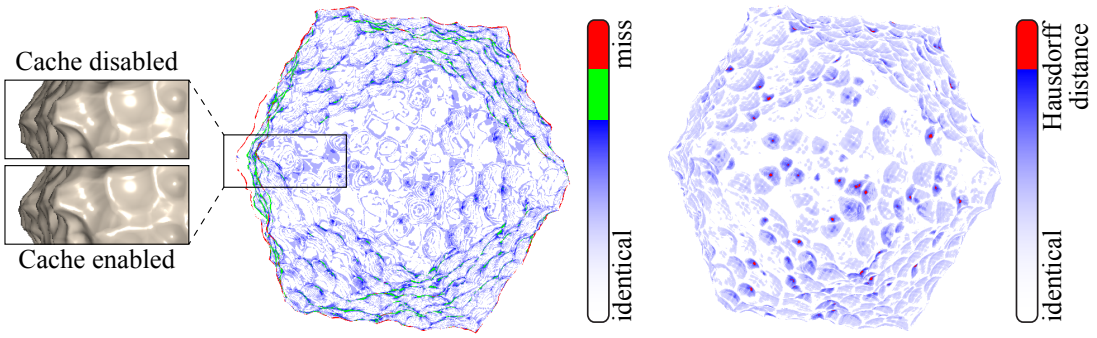


Figure 4.10: *Surfaces compared with cache enabled/disabled.* Left: *for each pixel on the image plane, the Euclidean distance between these surfaces along the view ray is visualized; silhouette mismatches are red.* Right: *on the uncached surface, the distance to the cached version is mapped; in red regions the Hausdorff distance is reached.*

At the green curves, the deviation is 15 times a cell length along the view ray. *Right:* enabling the cache yields a low-distortion surface. We numerically computed the Hausdorff distance by sampling one surface and performing a gradient descent onto the other (and vice versa). For all our examples and for a variety of cell sizes the Hausdorff distance levels out at roughly 65% of a cell length. Red areas indicate that the Hausdorff distance is about to get reached. Of course, this is assuming that the cell size specified by the user is small enough to capture high frequencies. If it is not the case, the Hausdorff distance can be much higher as thin features may be entirely missed.

Computing the surface normals has a significant impact on performance. Using central differences requires six additional texture queries after surface intersection. Using forward differences halves the amount of queries, but results in less smooth, artifact-prone shading (see Figure 4.4 to the right). However, performance then almost doubles from 30 to 58 fps for the the icosahedron with increased spot noise impulses, and with caching enabled. Without caching, performance increases from 12 to only 14 fps. Still, we opted for higher quality central differences for all examples and during performance measurements.

4.7.8 Long-Term Cache Behavior

An important question is whether the cache behavior remains consistent in time. Figure 4.11 presents a complex modeling session, with a step-by-step construction of a fountain. It is assembled using two intersected generalized distance functions for the core base shape, followed by two pairs of intersected boxes to form the hexagonal border. Another two pairs of intersected boxes model the ground. The basin of the fountain is carved out by a difference operation. Two cylinders, another box, and another difference operation form the central column. Two more intersected boxes are used for the water. This sums up to 15 shapes and 8 CSG operations. A complex procedural stone texture, which features both color and displacement, is applied to the base shape during the entire session.

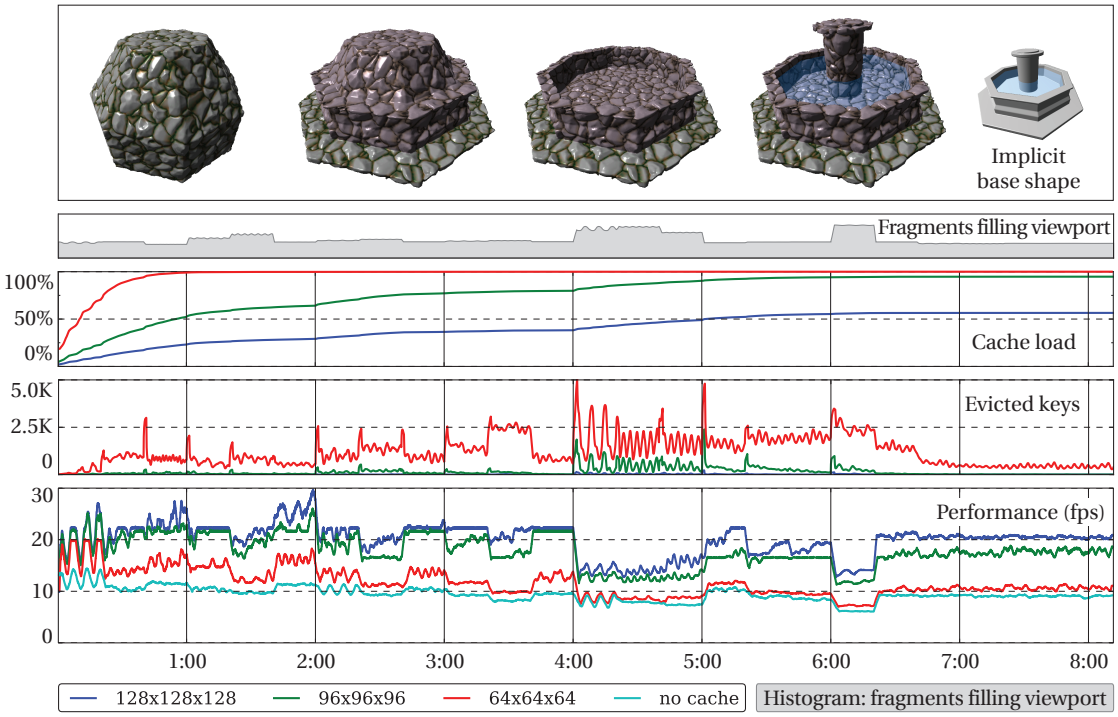


Figure 4.11: *Modeling a more complex shape: a fountain is constructed step by step using CSG. The graph gives an insight into the cache behavior throughout this eight minutes long session.*

To analyze the cache behavior over a longer period of time, we scripted a modeling session during which the fountain is assembled. This script allows us to play back the same session repeatedly, profiling the cache with different choices of parameters.

Every 20 seconds, a new shape is added or a CSG operation is performed. Each new shape is inserted off-center and then gradually translated into place to emulate a typical modeling operation. Three of the boxes appear only for associated, following CSG operations and thereafter disappear. These boxes are large in spatial extent and thus have a heavy impact on the cache while they are visible on the screen. The camera constantly rotates at 0.5 rad/sec.

See Figure 4.11 for our following analysis of three different cache sizes during the eight minutes long session. On the very top a histogram shows the proportion of fragments visible in the viewport. This directly influences performance, as ray marching is an image order rendering technique. Note the three plateaus; this is when extensive CSG proxy geometry is visible. The next plot reveals that the smallest cache (size 64^3, red curve) fills up within the first minute and then suffers clearly from insertions and involved evictions. The largest cache (size 128^3, blue curve), by contrast, has a generous amount of memory at its disposal and never fills up entirely during the session; consequently, little evictions occur inside the largest cache, but almost half of its allocated memory remains unused. In-between, a cache with size 96^3 (green curve) becomes entirely used, does not waste memory, yet generates few evictions: this mid-sized cache offers a good tradeoff and is most efficient. The cache size is the decisive factor for the amount of evicted keys, which is clearly reflected in the center plot.

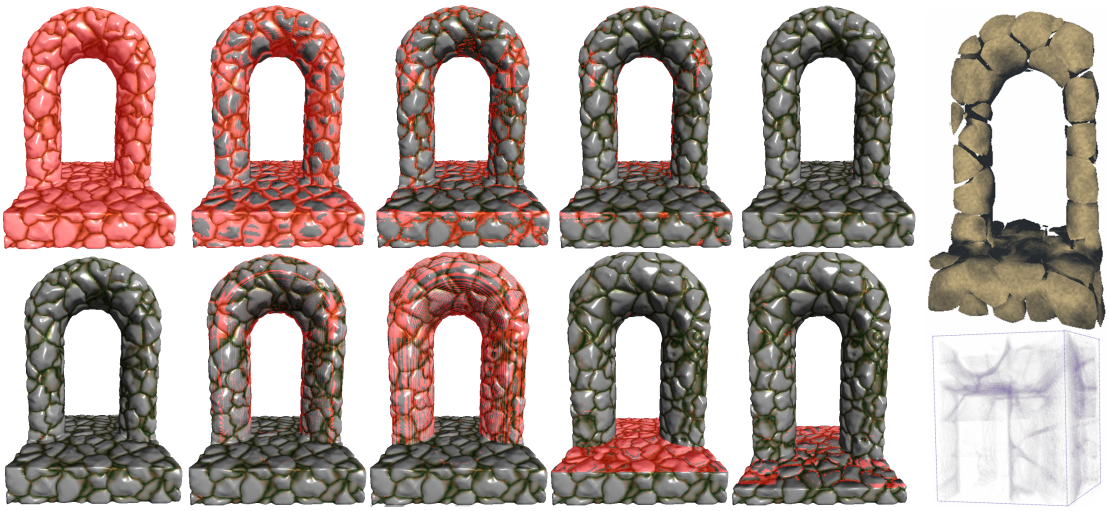


Figure 4.12: Top row: *the first five frames after clearing the cache. Cache misses appear in red.* Bottom row: *the user first increases the radius of the parametric arch and then lowers the base.* Right: *a volume visualization of the cache content during a typical modeling session.*

This in turn impacts the performance, as the bottom plot reveals. The cache remains beneficial throughout the entire session: note how the green curve gradually catches up with the blue curve, again and again after each change in the scene. Large proxy geometry used for CSG is visible between minutes 4:00 and 5:00. Hereafter, the mid-size cache, even though entirely full now, still provides a large benefit, despite the evictions required to fit in newly inserted data; plus, it does not waste memory as the largest cache does. Even if full, the caches quickly evict old data to adapt to abrupt large changes automatically, thus preserving performance.

In the end performance is slightly lower than initially, but note how the "no cache" curve also goes down: this highlights that there is no inherent cache degeneration, instead, the model just gets more and more complex.

4.7.9 Interactive Modeling Sessions

Figure 4.12 shows the evolution of cache misses when displaying a stone arch. Without the cache the arch is rendered at 11 fps, against 36 fps with the cache (fixed view, 800×800 resolution). Next, the same arch is being interactively deformed: the arch is parametric and the user changes its thickness and the height of its base. Thanks to the cache, only the modified parts require further computations. During this interaction performance ranged from 15 to 27 fps, depending on the magnitude of the changes. Performance is at its maximum when the user performs small, precise adjustments: this is an important property, because then a higher frame rate is most needed for fast visual feedback.

This also suits well for sketch-based applications; in Figure 4.13, the user sketches "SMI", which is extruded while sketching, and simultaneously inserted into the cache. Another example is given where we independently cache three different noise textures for multiple

Figure 4.13: *More examples.* Top left: *a scene with three independently cached textures.* Bottom left: *a sketching application.* Right: *Modeling with a complex tree bark texture; interaction remains smooth while adjusting parameters, because only the expensive base noise is cached.* **Acknowledgements:** *Iñigo Quilez kindly allowed us to use his procedural apple textures.*

objects. This allows the user, for instance, to adjust the placement and shapes of apples, without influencing other, unassociated caches.

Figure 4.13 also shows a session where an expensive tree bark texture is applied to a cylinder. After enabling the cache, performance raises from barely 6 to almost 40 fps. The user can adjust a variety of parameters, controlling the final appearance. Changing these does not invalidate the cache, which stores the base noise used to generate the bark appearance instead of the final result. Additionally, interaction remains smooth while the user can stretch, bend, and rotate cracks: we access already cached values for different positions in screen space with a simple transformation of the texture domain.

4.8 Limitations

There are several limitations in our current approach. First, the spatial resolution of the cache is homogeneous in space: for scenes with a large extent, the cache quickly saturates due to distant surfaces. A typical solution to this issue is to implement a multi-resolution cache [*Bastos and Celes* 2008; *Castro et al.* 2008]. This would also accommodate for cases where the data exhibits a non-uniform resolution across space.

A second limitation is that large shapes, which suddenly appear or disappear from the screen, lead to peaks in cache misses. This generates a high number of insertions. This may reduce performance for a few frames below the original performance obtained without using

the cache. We assume this problem could be alleviated by further limiting the maximum number of insertions per frame. Optimizing insertions further would also reduce this issue.

Shading has a significant impact on performance (Section 4.7.7). It should be investigated whether efficiently caching texture derivatives can accelerate surface normal approximation.

There is, of course, a more fundamental limit to the benefit of a cache: as it exploits spatio-temporal coherence, any change in the definition of the texture triggers a cache-wide refresh, and all prior values have to be discarded. There is little to be done about this issue.

4.9 Discussion

In the previous chapter we have presented an interactive procedural modeling environment. We have used purely analytical forms which support high-frequency detail, but more complex textures for these shapes could no longer be evaluated in real-time. This is why we have now addressed this issue and provide a significant improvement to our modeling environment.

Our runtime cache is a first step toward significantly accelerating the rendering and modeling of textured implicit surfaces. It integrates seamlessly into the on-demand evaluation of implicit procedural descriptions. We seek for the best balance between computation time and memory usage by exploiting spatio-temporal coherence between consecutive frames. Recent work in this area [*Schwärzler et al.* 2013; *Haaser et al.* 2015] targets many applications for accelerating the rendering of high-quality, highly detailed surfaces.

Our algorithm is very general and could be used to cache many other quantities in space, such as lighting, opacity, or eventually full hypertextures [*Perlin and Hoffert* 1989]. In Chapter 2 we have referred to more sophisticated procedural modeling techniques that extend to the highly (or fully) automated creation of landscapes, whole cities, and diversified natural scenes. It would be interesting to adapt our work to accelerate key parts of these techniques as well.

3D printing as a new application field for our modeling environment

We have presented an intuitive modeling environment which allows us to create interesting 3D objects. These objects are displayed on the screen, but many users want to go one step further and create physical representations of their models using a 3D printer. In the recent years, affordable 3D printers for hobby users have emerged in the market and gained strong attention. This also spawned a large body of research in fabrication methods.

This is why we now address 3D printing as a new application field in the remainder of the first part of this thesis. We will first develop new tools for our modeling environment which allow to create shapes optimized for fabrication on consumer hardware. We will introduce voxels as a new data structure to represent our objects. A voxel-based representation requires a much higher memory allocation than implicit surfaces. Both approaches are very contrary to each other. It turns out that 3D printers are inherently restricted to print smaller 3D objects, since they have to fit on the build plate during printing. We cannot utilize the unbounded, conceptually infinite space of our implicit techniques anymore. Still, both approaches can be combined: we can easily voxelize an implicit shape by sampling its surface inside a 3D volume.

In Chapter 6 we then revisit the texturing of 3D objects. We will present a new method to print two-tone texture-mapped models which exhibit both geometric and color information.

5

Interactive Modeling of Support-free Shapes for Fabrication

In this chapter we introduce an interactive sculpting approach that enables modeling of *support-free* objects: objects which do not require any support structures during 3D printing. We propose three operators—*trim, preserve, grow*—to maintain the support-free property during interactive modeling. These operators let us define brushes that perform either in an unconstrained manner (adapting the shape to the brush effect), or selectively discard changes inside the brush volume. Our technique can be applied to many modeling operations and we demonstrate it on brushes for adding or removing matter. We describe an efficient implementation of a voxel-based modeling tool that produces only support-free shapes, and show example shapes modeled within minutes.

5.1 Introduction

Interactive modeling and sculpting is an accessible and fun way to create 3D shapes. Many users would like to go one step further and 3D print their objects, but unfortunately this may turn into a frustrating experience as most shapes cannot print without temporary support structures. This is the case for filament printers, which are preferred by home users due to their low hardware complexity and low material cost (PLA and ABS thermoplastics in particular). Unfortunately, shapes having overhangs require support structures to hold the filament that would otherwise fall.

Support structures have to be manually removed after printing, a long and tedious operation that can lead to breaking the object if not done carefully. For most users, and non-experts in particular, the requirement for supports hinders the use of 3D printing.

Recently there has been a strong effort to improve support techniques. Supports are crucial in a general setting to allow any shape to be fabricated. Yet there exists a class of *support-free* shapes which print without supports. Browsing model sharing websites such as Makerbot Thingiverse reveals that most popular models belong to this category. This is not surprising: these models are much easier to print. In this chapter we investigate tools enabling users to create complex, yet easily printable shapes.

The sculpting metaphor has been thoroughly explored in the Computer Graphics community, from the seminal work of Galyean and Hughes [1991] to advanced software used by the

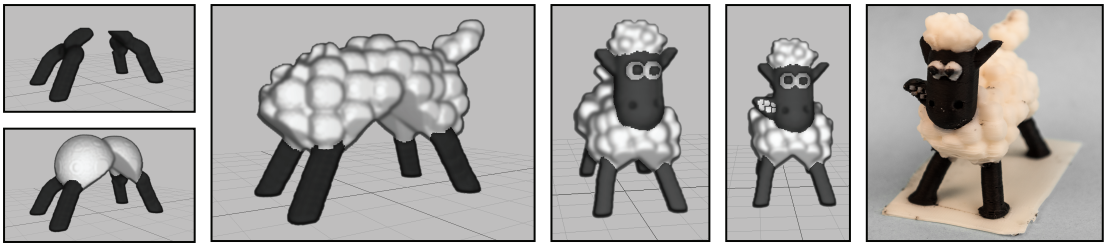


Figure 5.1: *Interactive modeling session for the* SHAUN *model. The user starts by sculpting four base legs and then attaches two spheres to form the body. Note how the spheres get trimmed to produce a support-free arch. The user then applies little spheres to model the wool and finally sculpts the head.* Right: *3D printed result. During modeling the shape always stays support-free.*

movie and video game industries, e.g., ZBrush. It is accepted as one of the most accessible techniques, as witnessed by the emergence of virtual clay software targeted at non-experts, such as Cubify Sculpt or VRClay. We follow this natural way of modeling and propose an interactive tool based on brushes for adding, removing, or locally modifying matter.

These brushes always preserve the support-free property. Starting from a shape that can be printed without support, a modeling session using our brushes will always lead to a new shape which is again support-free. In a standard modeling tool, adding or removing matter can lead to parts which are no longer printable, such as overhangs and islands (see Section 5.3.1 for a definition). To avoid these issues, either the brush has to be constrained to discard some changes, or the shape has to be modified to adapt to the changes made by the brush. We propose both options: using constrained brushes, the user knows that he will not damage the shape around the brush; using the unconstrained brushes, the user has full sculpting freedom and the shape around adapts to enforce printability.

In this work we introduce four basic brushes: *add-trim*, *remove-preserve*, *add-grow*, and *remove-trim*. The first two brushes are constrained, while the remaining two brushes are unconstrained. The operators *trim*, *preserve*, and *grow* observe printability constraints during brushing. These operators are general and can be applied to other modeling brushes, which we demonstrate by implementing a *smooth-trim* brush later.

Our method applies to any 3D printing technology that requires supports for overhangs and islands, e.g., technologies based on filament or resin. However, we focus on *fused deposition modeling printers* in this thesis. We implemented our entire voxel-based system on the GPU, performing all operations at interactive frame rates. We demonstrate our approach with a variety of 3D prints modeled with our interactive environment.

5.2 Previous Work

Modeling for fabrication

Recently many approaches appeared to help modeling shapes in a fabrication context. Several methods consider shape balance [*Prévost et al.* 2013; *Bächer et al.* 2014], others focus on

structural properties [*Stava et al.* 2012; *Zhou et al.* 2013a; *Umetani and Schmidt* 2013], or help to design mechanisms [*Coros et al.* 2013; *Koo et al.* 2014; *Hergel and Lefebvre* 2015]. Our work fits within the same trend but puts emphasis on interactive modeling of support-free shapes, a challenge which to the best of our knowledge has not been considered before.

Support structures

Support structures are a well-identified bottleneck of additive manufacturing. Therefore, researchers have considered ways to reduce the amount of support. In particular, novel types of structures have been recently proposed [*Wang et al.* 2013; *Schmidt and Umetani* 2014; *Vanek et al.* 2014a; *Dumas et al.* 2014]. These approaches use less material than the standard downwards extrusions by relying on truss-like structures, such as trees and scaffoldings. Other approaches divide a shape into several parts with the primary objective of reducing print time [*Luo et al.* 2012; *Chen et al.* 2015; *Vanek et al.* 2014b; *Herholz et al.* 2015; *Hu et al.* 2014], whereas this also has the potential of reducing support requirements. Unfortunately this leads to a manual assembly and gluing step which we would like to avoid: it can be complex on intricate geometries and leaves seams on the surfaces. Reducing support can also be achieved by a better selection of the print orientation with respect to the build direction [*Thompson and Crawford* 1995; *Alexander et al.* 1998; *Zhang et al.* 2015]. The approach of Hu et al. [2015] goes further by also deforming the model to reduce the need for supports.

These techniques alleviate the difficulties related to supports whenever it is unavoidable. Our goal is different as we seek to help users *directly model* shapes that print without support.

Self-supported structures

A number of techniques consider the definition of support-free structures. Such structures have a specific geometry that allow their fabrication without any support. The shapes modeled by our technique belong to this category.

Hu et al. [2014] propose to approximate a shape by a set of *pyramids*. A pyramid is defined as a height field with a flat base. Assuming the object is filled, such a shape is guaranteed to print without support. Hornus et al. [2015] propose a generic definition of a support-free shape, which leads to algorithms to create minimally fitting envelopes and maximum inner carvings. We follow a similar analysis and use it as a basis to implement our brushes.

5.3 Our Approach

We model the shape as a voxel grid V of resolution N^3. This is a natural representation for both sculpting algorithms [*Ferley et al.* 1999] and for expressing fabrication constraints. A voxel is indexed as $V(i, j, k)$ where the third coordinate aligns with the vertical z-axis, which is the build direction. We typically rely on voxels having a side length of 0.25 mm (250 µm), which is within the standard range of layer thicknesses for filament printers. Each voxel $V(i, j, k)$ stores a density (1: solid, 0: empty) and a material ID.

Before fabrication the shape is *sliced* by intersecting it with uniformly spaced planes [*Dinh et al.* 2015]. Each slice contains a set of contours delimiting inner areas to be fabricated as a

physical layer by the 3D printer. Layers are fabricated one after the other, from bottom to top. The layer above will bond to the layer below, progressively forming a solid object. For the sake of simplicity we align the slices with the voxel grid, and obtain slice S_k as the plane of voxels

$$S_k = \{V(i,j,k) \mid V^{(i,j,k)}_{\text{density}} = 1,\ \forall (i,j) \in N^2\}.$$

The slice contour is then extracted from S_k, which contains only solid voxels. This could be refined by using a higher voxel resolution along the build direction (leading to thinner layers), or by interpolating between slices or all voxels.

5.3.1 Fabrication Constraints

There are two major fabrication constraints to consider: overhangs and islands. Overhangs are almost horizonal parts of the object which prevent material to properly adhere to the layer below. Islands are disconnected regions that appear during fabrication, typically protrusions with a downward angle. On filament printers, overhangs and islands lead to failed prints with dangling material: the filament being deposited is not supported from below and either fails to properly bond or falls by gravity. On resin printers similar defects appear: disconnected regions end up floating in the resin tank and bond at random places on the object, while overhang areas can distort when the print is detached from the fabrication surface.

Hornus et al. [2015] showed that both overhangs and islands can be detected through morphology operations on the object slices. Indeed, when considering slices, the overhang constraint translates to verifying whether the slice S_{k+1} is entirely included within a dilation of slice S_k by a disk that captures the maximum allowed overhang, a condition we denote as:

$$S_{k+1} \subseteq S_k \oplus \mathscr{B}_r\ , \tag{5.1}$$

where $\oplus$ denotes the dilation operator ($\ominus$ the erosion), $\mathscr{B}_r$ is a discrete disk of radius $r = h{\cdot}\tan\theta$, h being the layer thickness, and θ the max overhang angle. We use $h = 0.25$ mm, $\theta = 45°$ and perform all operations in the discretized voxel setting. Intuitively, $\mathscr{B}_r$ indicates how far S_k can grow without producing an excessive overhang, while $S_k \oplus \mathscr{B}_r$ denotes the largest area that can be supported in S_{k+1}. Islands directly contradict Equation 5.1 since they only exist in S_{k+1}.

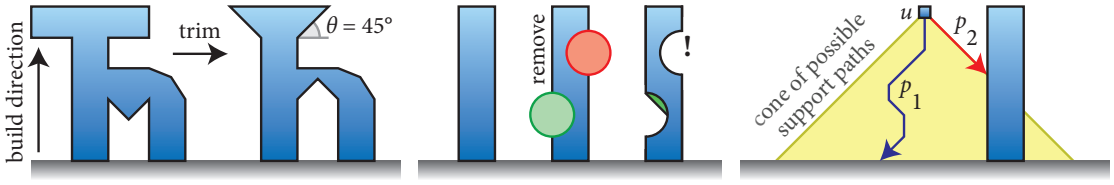
A support-free shape has to enforce Equation 5.1 everywhere. Areas violating the constraint in slice S_{k+1} are easily detected as $S_{k+1} \setminus (S_k \oplus \mathscr{B}_r)$. Note that we ignore here one special case of filament printers which is the possibility to print straight bridges within a layer.

In our setting we use Equation 5.1 to determine whether each voxel is *supported*. A voxel in slice S_{k+1} is supported if and only if $V(i,j,k+1) \in S_k \oplus \mathscr{B}_r$, it is unsupported otherwise. Ground voxels $V(i,j,0)$ are always supported.

5.3.2 Brushes (Overview)

The four main brushes *add-trim*, *remove-preserve*, *add-grow*, and *remove-trim* are unrestricted in their shape and size. As their names indicate, two add matter and two remove matter.

There are two ways of preserving fabricability: either by constraining changes within the brush (add-trim, remove-preserve), or by modifying the shape around the brush (add-grow, remove-trim). We now introduce three operators that recover from overhangs and islands (*trim*, *preserve*, and *grow*), together with our four main brushes.



(a) *global trim operation* **(b)** *remove-preserve example* **(c)** *growing an unsupported voxel*

Figure 5.2: *Operations overview.* (a) *A global trim operation removes unsupported parts of the shape such as floaters, islands, and overhangs; only the support-free subset remains.* (b) *The green brush removes but also preserves voxels required for the shape to remain support-free.* (c) *Growing the shape can be done in many ways* (p_1)*; we propose closest supporting paths* (p_2).

Trim (add-trim, remove-trim)

Trimming removes any problematic part from the shape, deleting matter as illustrated in Figure 5.2a. We perform a bottom-top sweep, removing unsupported voxels within all slices. This operation removes all unsupported voxels; it cascades from bottom to top and thus can have a global impact on the shape.

Add-trim: this brush adds matter inside the brush volume, but only the subset which is supported. (This is equivalent of filling the entire brush and then calling the trim operation.) It is a useful tool to add details while ensuring no overhangs or islands are created. It never affects the surrounding shape and the effect remains localized inside the brush volume.

Remove-trim: this brush deletes all matter inside the brush and then calls a global trim operation. The effect propagates upwards in cascade during the sweep, to account for previously supported voxels that have just turned into unsupported voxels. Eventually only the support-free subset of the shape remains.

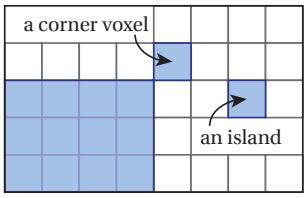
Preserve (remove-preserve)

The previous *remove-trim* brush can lead to undesirably large deletions. We therefore propose another removal brush that attempts to *preserve* the shape above the brush. Particularly we want to preserve voxels inside the brush volume necessary for support above.

To achieve this, we consider the minimal necessary support for a slice S_k. Let U_{k-1} be the smallest surface required at slice $k-1$ to support S_k, that is $S_k \subseteq U_{k-1} \oplus \mathscr{B}_r$ (Equation 5.1).

Let us assume for now that $S_k = (S_k \ominus \mathscr{B}_r) \oplus \mathscr{B}_r$, that is S_k is invariant by morphological opening. Under this assumption it follows from Equation 5.1 that $U_{k-1} = S_k \ominus \mathscr{B}_r$. It is minimal since removing any voxel would produce a surface that cannot support S_k, as $S_k = U_{k-1} \oplus \mathscr{B}_r$.

This remark leads us to a simple algorithm to preserve voxels necessary for support. First delete all voxels within the brush volume. Then sweep down from the slice just above the brush to the slice just below, replacing in sequence each slice by $S_k = S_k \cup U_k$ with $U_k = S_{k+1} \ominus \mathscr{B}_r$. This guarantees that the minimal set of necessary voxels is reintroduced, *under the assumption that* $S_k = (S_k \ominus \mathscr{B}_r) \oplus \mathscr{B}_r$. Figure 5.2b shows an example where a part of the green brush has been preserved.



Unfortunately the assumption breaks in two notable cases (see inset). The first case is due to spurious voxels that are not included in the opening of S_k. These voxels correspond to corners that cannot be captured by $\mathscr{B}_r$. The second case is more problematic and is due to small islands: surfaces smaller than $\mathscr{B}_r$ in S_k. While we tried several approaches, we find that for modeling purposes the simplest technique is to trim these voxels. Corners disappear without further impact. Unfortunately trimming islands removes the entire part located above. Still, the tool remains natural to use, as the removed islands were disconnected parts that would otherwise fall.

Grow (add-grow)

This brush is the most challenging to define: it allows to paint in free space while growing the shape to ensure the newly added matter is printable.

For this purpose we define a *grow* operation. It is akin to support techniques, where a temporary structure is built to support the shape during fabrication (see Section 5.2). However, we seek to define a *modeling* operation that will modify the shape to embed the support inside its geometry. The requirements are very different. Support structures are optimized to use little material, to print fast and to snap easily. Instead we define an operation that behaves in a predictable manner, connects seamlessly to the existing shape and feels natural to the user. In addition, while support generation techniques have access to the final object, our technique discovers it incrementally as it is modeled by the user.

Our key intuition is that the shape should grow in a minimal way, avoiding overhangs and islands while adding matter which naturally connects to the existing shape. Let $u = V(i,j,k)$ be a single unsupported voxel in slice S_k. Consider a path $V(i_0,j_0,k_0),...,V(i_p,j_p,k_p),...,u$ of *empty* voxels that is monotonically growing, i.e., $k_{p+1} = k_p + 1$. Such a path is said to be *supporting* if and only if:

1. $k_0 = 0 \quad or \quad V(i_0,j_0,k_0) \in S_{k_0-1} \oplus \mathscr{B}_r$,
2. $V(i_p,j_p,k_p) \notin S_{k_{p-1}} \oplus \mathscr{B}_r$,
3. $V(i_p,j_p,k_p) \in \{V(i_{p-1},j_{p-1},k_{p-1})\} \oplus \mathscr{B}_r$,

where 1) states that the first voxel is supported by the slice below, 2) states that other voxels are not supported by the shape, but 3) would be supported by their voxels below in the path if they were solid. Any such supporting path can be added to the shape to support voxel u. There is a large number of possibilities: all possible support paths lie within a cone of θ slope with its tip on u, see Figure 5.2c. The choice is critical as it defines *how* the shape grows during modeling.

We propose to consider *closest supporting paths*: supporting paths that remain as close as possible to the existing surface. We define the cost of a path as $C(v_0,...,v_k) = \sum_i (D(v_i))$ where $D(v)$ is the distance between the empty voxel v and its closest filled voxel within the slice plane. The rationale is to reconnect with the shape while staying as close as possible to existing surfaces. The closest supporting paths achieve this by minimizing the distance between the path and the existing shape. In practice, computing D for every slice is expensive.

We reformulate the cost to take only the distance on the ground plane into account: $C(v_0,...,v_k) = k + D_{\text{ground}}(v_0)$. With this cost definition the closest supporting paths can be computed very efficiently by a bottom-top sweep through V (see Section 5.4). This achieves a good compromise during interactive modeling (see Section 5.5).

A possible way to implement the add-grow brush would be to paint unconstrained voxels freely, and later reconnect them to the shape using the grow operation: for each unsupported voxel fill the closest supporting path with solid voxels. This does not yield satisfying results due to the discrete nature of the voxel grid: two neighboring voxels may produce two slightly different supporting paths, producing small gaps.

Instead, we only consider the closest support path at the center of the brush. We follow the path and apply the brush everywhere along it. This generates a support-free structure which naturally connects the current stroke to the shape.

The final behavior of the brush, as perceived by the user, is to generate surfaces connecting the matter being added to the shape in an "as close as possible" manner. It is therefore predictable and quite natural to use after a few minutes. It lets the user freely paint without being stopped mid-air by the constraint enforcement.

Generic Brushes

The trim, preserve, and grow operators can be used in combination with other modeling operations. As an example, we have implemented a *smooth-trim brush* which sets a voxel's density according to an average of its neighborhood. Outliers and isolated voxels are removed by subsequent trimming.

The tool can also import voxelized models from a file. Since we want our shapes to be always support-free, we propose *import-trim* and *import-grow* operations to either remove unsupported parts immediately or to expand them accordingly.

5.4 Implementation

We implemented our modeling tool using OpenGL 4.5 and make extensive use of compute shaders for all modeling operations. The entire modeling space spans 512^3 voxels, which allows us to sculpt fairly large objects in the context of printing: using a layer thickness of 0.25 mm and isotropic voxels, every dimension can be up to 12.8 cm in size.

We allocate 3D textures for the voxel space, the shape of the brush, and a shortest path map for the add-grow brush. We use two channels (red and green) for the voxel space texture to store density (1: full, 0: empty) and material ID separately. (This information can be easily merged into a single channel to save memory, but our algorithms are much simpler to explain using separate channels.) On this voxel space we dispatch compute shaders for every modeling operation; we do not keep an additional copy of the shape in system memory.

As in our notation of Section 5.3, the z-axis aligns to the vertical build direction. For a single slice, this allows for efficient dispatches of compute shaders in the x-y-plane and bigger work group sizes. Note that while we experience a significant speedup due to coherent memory access patterns on the GPU, results can vary considerably depending on the internal GPU layout and driver being in use.

Figure 5.3: Left: *examples of an add-trim stroke, a remove-preserve operation where a cube is subtracted from a pillar, and an add-grow stroke which automatically creates a bridge underneath.* Center: *the* PUMPKIN *model can be used as a tea-light holder.* Center Right: *a more intricate sculpture.* Right: *more prints of the* SHAUN *model. All shapes are support-free.*

5.4.1 Rendering and Shading

Instead of converting the object into an intermediate representation using meshes, we render directly by ray marching efficiently through the voxel space using a 3D Digital Differential Analyzer [*Amanatides and Woo* 1987]. We use a deferred shading approach to render the scene: a framebuffer object (FBO) holds textures to store geometry (G-buffer), material ID, surface normals, and depth information.

We filter both the geometry and normal buffers to introduce smooth surfaces and to make the rendered result appear less blocky and significantly less prone to aliasing effects. Bilateral filters and silhouette effects help to distinguish between individual parts of the object. We keep a copy of the non-smoothed G-buffer for modeling and picking purposes.

As this shading approach is inexpensive on modern GPUs and our compute shaders operate on the entire voxel space, the frame rate is largely independent of the amount of filled voxels. As a valuable benefit, adding more matter by filling voxels does not slow down the modeling tool: large sophisticated scenes render as fast as small and simple objects.

5.4.2 Brushes (Implementation)

When the user starts brushing, we save and lock a copy of the G-buffer. This allows for brushing along the surface of an existing object. There is another advantage during erase operations: if the user carves a hole through an object in the front, any objects behind remain unaffected, because they are still not visible in the G-buffer copy we are operating on.

Considering $\theta = 45°$ we can set $\mathscr{B}_r$ to be a 3×3 structuring element. Performing a dilation on a voxel then leads to filling its eight surrounding neighbors within the same slice. Eroding a slice will retain only voxels with a full neighborhood. Using these morphological operations we now show how to implement the *trim*, *preserve*, and *grow* operations for our brushes.

A *global trim operation* (on the entire 3D voxel space) can be implemented by sweeping from bottom to top (layer 0 is always supported):

```
for (int layer = 1; layer < RESOLUTION; layer++) do
    trimShaderProgram->setUniformValue("layer", layer);
    glDispatchCompute(RESOLUTION/32, RESOLUTION/32, 1);
```

The shader removes all voxels without support from below:

```
layout(rg8ui, binding=0) uniform uimage3D voxelTex;
uniform int layer;
layout(local_size_x=32, local_size_y=32, local_size_z=1) in;
void main(void) {
pos = (gl_GlobalInvocationID.x, gl_GlobalInvocationID.y, layer);
if ( ⋀_{i,j∈{−1,0,1}} imageLoad(voxelTex, pos.x+i, pos.y+j, pos.z-1).x == 0)
 { imageStore(voxelTex, pos, ivec4(0, 0, 0, 0)); }}
```

We implemented the *add-trim brush* by dispatching a shader inside the brush volume and then adding only voxels with support from below (no subsequent trim operation required):

```
pos = brush_position + gl_GlobalInvocationID; m = materialID;
if ( ⋁_{i,j∈{−1,0,1}} imageLoad(voxelTex, pos.x+i, pos.y+j, pos.z-1).x == 1)
 { imageStore(voxelTex, pos, ivec4(1, m, 0, 0)); }
```

The *remove-preserve brush* is implemented by dispatching a shader inside the brush volume, layer by layer, from *top to bottom*, and then dilating empty space from the layers above:

```
pos = brush_position + gl_GlobalInvocationID;
if ( ⋁_{i,j∈{−1,0,1}} imageLoad(voxelTex, pos.x+i, pos.y+j, pos.z+1).x == 0)
 { imageStore(voxelTex, pos, ivec4(0, 0, 0, 0)); }
```

The *grow operation* for the *add-grow brush* is more involved. We allocate a 3D path map that spans the entire voxel space. First, for every voxel on the ground plane, we compute the distance to the closest filled voxel on the ground plane and store this information in the ground layer of the path map. Next, we sweep through the voxel space from bottom to top, propagating information inside the path map. For every voxel v in layer $k+1$ we look at its 3×3 neighborhood in the layer k below and detect voxel u which stores the minimum path cost. We increase this cost by one and store it for v. In a second channel of the path map, we encode the direction how to navigate from v back to u; there are nine possible directions. In the case of a tie (equal costs) we favor going straight down over moving diagonally.

The user starts sketching a stroke path S in free space while we map the mouse cursor position to the corresponding voxel on the sketch plane. During sketching the amount of voxels in S is monotonically increasing. For every new voxel v_i added to S we dispatch a compute shader which then navigates voxel by voxel from v_i through the path map until it reaches the surface or the ground. This forms a new supporting path P. We emit the brush centered at every voxel $p_i \in P$. This ensures that all added matter along S is always supported (see Section 5.3, *grow*). We update the path map every time the user releases the mouse button after sketching.

5.5 Results and Limitations

We developed and tested our modeling tool on a system with an Intel Core i7-4790K CPU and an Nvidia GeForce GTX 980 graphics card. We are well able to achieve interactive frame rates for all modeling operations at a Full HD resolution of 1920×1080.

Figure 5.1 depicts an interactive modeling session for the SHAUN model. Modeling this shape without prior practice runs took half an hour, including several *undo* operations. A user typically starts modeling from the ground. For SHAUN, we started forming the legs and then the main body part before finishing with the head. The final print is almost 6 cm tall and roughly 7 cm long. In order to obtain camera-ready results, we slowed down the print head significantly to move at 7 mm/s. This led to printing times of 16 hours for the single-color and 17 hours for the dual-color print on a Makerbot Replicator. To improve stability during printing, we placed the model on a very thin raft which can be easily peeled off.

Figure 5.3 shows more results, including examples of brush operations. The PUMPKIN model was printed on an Ultimaker 2 in under two hours. The intricate arms do not require support structures. We also printed a more detailed and intertwined sculpture.

There are limitations to our current approach. We do not consider minimal thickness during modeling; our slicer preserves thin features, and properties such as a one-thread thickness will clearly result in failed prints. Considering minimal thickness is an interesting venue of future work. This could possibly be achieved through morphological closures or openings, or by maintaining a volume distance field. During printing, we observed heavy stress on the legs of the SHAUN model. We would like to combine our work with approaches which quickly give feedback on mechanical properties during interaction [*Umetani and Schmidt* 2013; *Xie et al.* 2015]. We strictly enforce the creation of support-free shapes throughout the entire modeling session. It would be interesting to consider selectively relaxing this constraint from time to time, and ultimately return to a support-free state.

5.6 Discussion

With the techniques presented in this chapter we hope to make modeling for 3D printing more accessible, in particular to non-expert users. Interactivity provides an enjoyable modeling experience, while the support-free shapes greatly simplify 3D printing. A key challenge in achieving our goal was to design modeling brushes that perform the usual operations (add, remove, smooth, etc.) while preserving the support-free property. We proposed three elementary operations to apply during modeling: trim, preserve, and grow. They are generic, fast, and allow the definition of many other support-free modeling operations.

While our tool is easily accessible for hobby users—none of us in an artist—we saw the demand for implementing a history for undo operations in very early stages of development. We received encouraging feedback for our four main brushes, and in addition, smoothing was found to be particularly useful.

In Chapter 4 we saw how applying textures to shapes leads to increased visual complexity. In the next chapter, we revisit textures and show how to apply them to 3D printable shapes.

6

Dual-Color Mixing for Fused Deposition Modeling Printers

We now detail a method that leverages the two color heads of recent low-end fused deposition modeling (FDM) 3D printers to produce continuous tone imagery. The challenge behind producing such two-tone imagery is how to finely interleave the two colors while minimizing the switching between print heads, making each color-printed span as long and continuous as possible to avoid artifacts associated with printing short segments. The key insight behind our work is that by applying small geometric offsets, tone can be varied without the need to switch color print heads within a single layer. We can now effectively print (two-tone) texture mapped models capturing both geometric and color information in our output 3D prints.

6.1 Introduction

Recent technological advances have made 3D printers affordable for home users and their increased use is foreseeable. Unsurprisingly, 3D printing has gained a lot of attention in the computer graphics community, and researchers are contributing to this new output medium in the areas of geometric modeling, rendering, and perception. Obviously it is very desirable to print 3D objects in full color, however, low-cost entry-level printers are restricted to print using only monochrome filaments. In this chapter we present a dithering technique for printers with at least two nozzles. We produce interleaved color patterns, which convey an impression of smooth color blending, which ultimately provides a broader color (or grayscale) spectrum. We used the MakerBot Replicator 2X desktop printer for all our experiments with black and white filament for maximum contrast.

Contributions. We show how to construct a basic checkerboard pattern which gradually dissolves into an in-between tone, while minimizing the switching of nozzles at the same time; we propose a simple scheme to project this pattern onto arbitrary shapes; we demonstrate how to texture map these objects efficiently; finally, we reveal a technique to avoid artifacts associated with printing using two nozzles, significantly improving print quality. We also discuss the view-dependent appearance of the textured prints, including slanted surfaces.

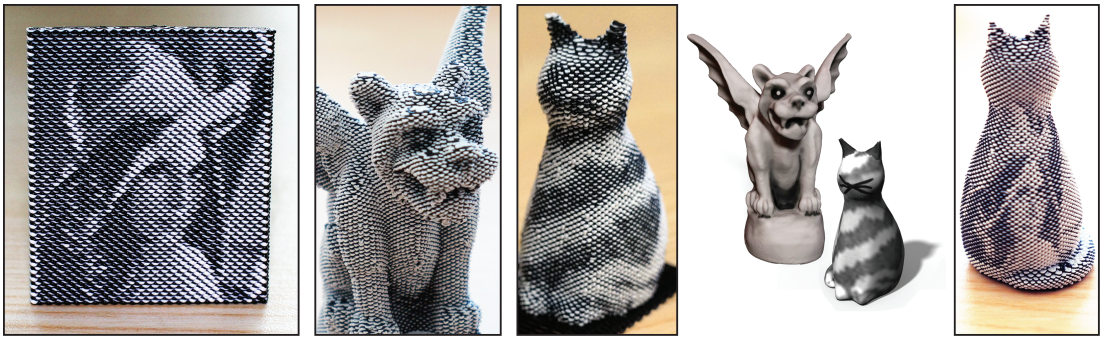


Figure 6.1: Left: *Lena image 3D printed on a MakerBot Replicator 2X. This plate is 3.5 cm tall and about 2 mm thick. We achieve approximately 50 dpi vertically with a 60× 71 texture modulated on 142 alternating black and white layers.* Center: *3D models printed with continuous tone.* Corresponding texture maps: *an ambient occlusion texture gives the* GARGOYLE *(10.2 cm tall) an aged appearance while we fit* LUCYCAT *(5.4 cm tall) with a calico coat.* Right: *the Lena texture remains sharp after projection on a curved shape.* **Acknowledgements:** *we thank Zoltan Bourne (Lucy the cat) and Julie Dorsey (Gargoyle) for permission to use their models.*

6.2 Previous Work

Recent advances in 3D printers along with their increased availability have led to numerous research efforts. We refer to a recent state-of-the-art report from Hullin et al. [2013], which, among others, gives a broad overview of recent fabrication methods. There also exist numerous investigations on improving the structural integrity of 3D printed forms [*Telea and Jalba* 2011; *Stava et al.* 2012]. For large objects, methods have been developed to effectively partition the model into pieces that can be fabricated within a fixed print volume [*Luo et al.* 2012]. Methods for printing both articulated and deformable structures have been proposed [*Bickel et al.* 2010; *Bächer et al.* 2012; *Calì et al.* 2012; *Coros et al.* 2013]. The technique of Wang et al. [2013] focuses on reducing the cost of printing, while other approaches exist for optimizing quality [*Hildebrand et al.* 2013]. All of these techniques, however, focus on the geometric form of the printed shape.

Our work is more closely aligned to that of OpenFab and Spec2Fab [*Vidimče et al.* 2013; *Chen et al.* 2013] in that we focus on the appearance of the printed shape and not just its geometric form. The general idea of printing color is not a new one. For example, the work of Levin et al. [2013] showed that full BRDFs (see upcoming Section 7.4) can be fabricated, and Hašan et al. [2010] use multi-material 3D printers to approximate a given or measured heterogeneous subsurface scattering function. These works make use of high-end printers which support color printing, such as printers from 3D Systems (powder-based Projet series) and Mcor Technologies (paper-based). Although these printers can produce high-quality color prints, it is unclear whether their costs can come down to reach the consumer space any time soon. A new company, BotObjects, is offering an FDM-based printer where the filaments are mixed to produce a variation in color. It will be possible to create slow gradations in color throughout the object, but not sharp transitions between colors without time and

material costly purging and refilling of the mixing chamber. Our work, by contrast, is aimed at leveraging low-cost consumer printers based on FDM. We restrict our research to focus on the reproduction of continuous tone, contrary to a full BRDF. Our work is specifically targeted at the broad hobbyist and consumer 3D printing communities providing an effective practical solution whose quality scales as the printers increase in both accuracy and precision.

In the low-end space, some attempts have been made to print continuous tones. Hobbyists started to feed multiple filaments into a single nozzle and tried to mix the color at the head of the printer. While this type of approach may have promise, it does not allow the rapid color changes necessary to print high-frequency detailed imagery on the surface of 3D shapes. We refer to this approach as active color mixing, while we consider our work as a passive mixing technique. Cho et al. [2003] used dithering to produce tone variation. From our experiments we found this type of approach impractical on low-end FDM printers; they cannot print short spans or small dots without producing significant geometric artifacts due to the lack of precise flow rate control of filament from the heads of the nozzle. The target of our work is to enable the appearance of such dithering while maintaining long contiguous spans of single color.

6.3 Algorithm

To print continuous variation of tone using two colors one could adapt a halftone technique to 3D printers: very small regions of the model would be assigned different colors and long continuous print spans produced by the slicer would be divided up into very short spans (or dots) of interleaved color. While at first glance this straightforward approach may sound promising, FDM printers produce significant artifacts when filament streams are not long and continuous; this is due to the rate at which the printer can start and stop the flow of melted filament. Furthermore, such an approach would require incessant switching between print heads, increasing the total printing time significantly. Thus our challenge is to derive a scheme that minimizes switching between print heads while maximizing the length of continuous spans of filament emitted from the nozzle.

6.3.1 Tone Variation using Geometric Offsetting

An initial experiment involved printing a test cylinder where every other layer from the slicer was printed in alternating black and white color. Such a print when viewed at a distance should produce a constant gray appearance. To our surprise our test print produced more variation in tone than expected. Upon close examination we noticed that subtle misalignment of the color print heads lead to slightly offset circles alternating every other layer. Thus one side of the cylinder appears darker where the black filament protruded, and the other side lighter due to more exposed white filament. In-between the color went to a smooth tone of gray where the two filaments were aligned and equally exposed. This very simple test print leads to the observation that subtle geometric offsetting between layers enables the control of variation of tone due to occlusion and gravity: overhanging filament sags a little, closing the gap by hiding the recessed filament color. This allows us to modulate the tone by offsetting or insetting each layer slice producing the desired tonal variation along the vertical print axis.

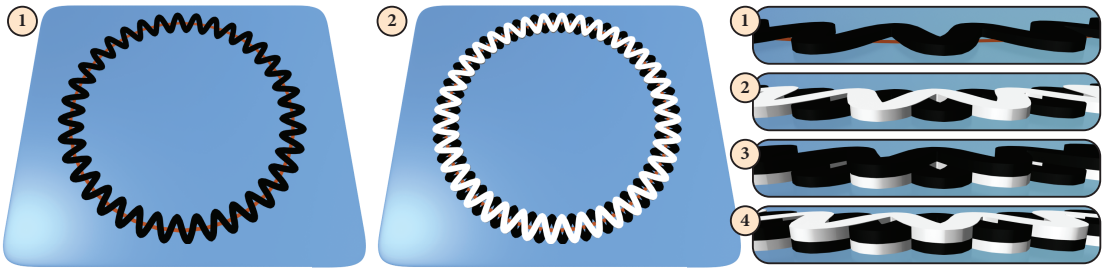


Figure 6.2: *Printing the first layers of a cylinder with interleaved color patterns.* Left: *the original outline, visualized as a circle on the build plate, is modulated with a sine wave to print the first layer in black.* Middle: *the second layer is printed in white, but the phase of the sine wave has been shifted by* π. Right: *the closeups show how the second layer sags into the first, resulting in an interleaved effect. The third layer is printed in black again keeping the same phase as the second layer. The fourth layer is then printed in white reverting to the original phase of the first layer. This technique produces a basic checkerboard pattern.*

In Figure 6.2 we show how to exploit this observation to produce interleaving patterns on a test cylinder. We apply a high frequency sine pattern to the circular outlines (visualized as a circle on the build plate) during slicing and print the first layer in black. By phase shifting its sine pattern by π, the second layer printed in white overlays into the first, producing an interleaved chain of very short horizontally alternating black and white strips. We retain this phase and print the third layer in black directly atop the second layer. The fourth layer is again printed in white, reverting to the original phase of the first layer. The outcome is a basic checkerboard building block which is simulated and visualized in the closeups.

We use this technique to construct the cylinder in Figure 6.3a. The closeup visualizes the G-Code produced by the slicer: always after printing a pair of two layers we shift the phase by π. This leads to filament extruded into mid-air, but due to gravity the layers of the real print interleave with each other. Comparing (a) with (b) demonstrates that the appearance of a printed result can be simulated well before starting an actual printing process. This allows for faster modeling iterations and avoids costly trial-and-error prints.

In Figure 6.3c we again alternate between black and white for every layer, but do not shift the phase between pairs of layers; instead, we investigate the effect of phase shifting the white layers against the black layers for varying amounts. Starting at the bottom, layers are not shifted against each other. Note how horizontal stripes can be seen. Upward, the shift is $1/4\pi$, $1/2\pi$, $3/4\pi$, and π in the center. Note how this allows printing *vertical stripes* while switching colors on a *horizontal basis*. Further upward, the shift is $5/4\pi$, $3/2\pi$, $7/4\pi$, until there is eventually no more shifting again.

Modulating the amplitudes of the sine waves locally then controls the tonal variation. To make the tone appear brighter, we attenuate the amplitude of black layers. Vice versa, attenuating white layers makes the tone appear darker. In Figure 6.3d, we can perceive a smooth gradient on the cylinder. This modulation technique is also used to apply texture mapping to our prints, as demonstrated in the teaser (Figure 6.1).

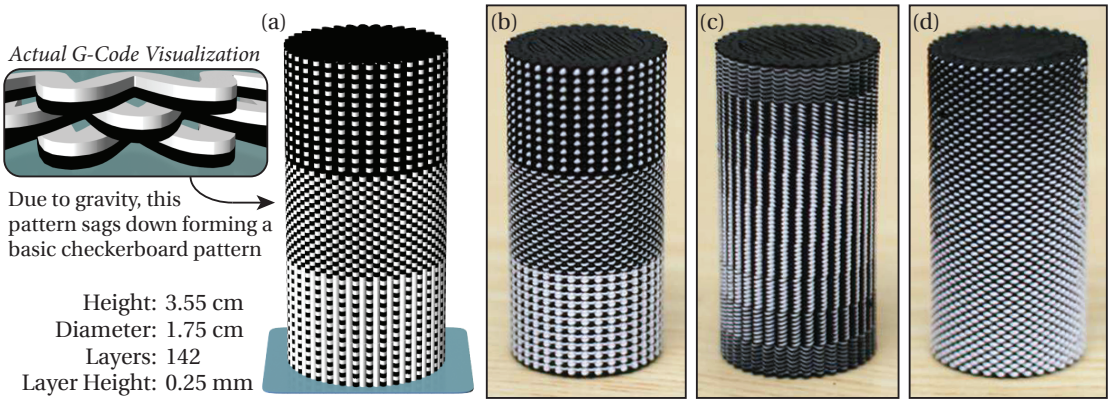


Figure 6.3: (a) *Cylinder built using the basic checkerboard pattern from Figure 6.2. In the top and bottom parts, three out of four layers are printed in black and white, respectively. This is a visualization rendered with mental ray, whereas* (b) *shows a photo of the actual 3D printed result. In* (c)*, we shift the phase between black and white layers.* (d) *Here we gradually attenuate amplitudes to produce a smooth gradient on the cylinder.*

Note that this approach avoids the rapid switching of color print heads. Within each layer we produce the same number of long continuous spans that would have been emitted from the slicer for single color printing. Thus our method only incurs the overhead required to switch color print heads between layers, which is minimal, and the artifacts associated with starting and stopping filament streams are no worse than that of regular single color printing.

6.3.2 Arbitrary Shapes

The checkerboard pattern nicely dissolves into an in-between tone when viewed from a distance and is the foundation for texture mapping our models. We can map this regular pattern perfectly onto the previous cylinder examples or other extruded shapes. For arbitrary shapes, however, we either have to alter the sine pattern frequency or allow for errors in phase between levels. Unfortunately, even small deviations in this pattern lead to salient artifacts in the prints, as the human eye instantly recognizes subtle pattern misalignments. Thus the challenge becomes bounding distortions in frequency and phase between layers.

While one could perform some global optimization using all layer slices in parallel, our solution operates locally on the current layer slice and requires only the knowledge of the past $k \geqslant 1$ sliced layers. We iteratively perform a projection and relaxation process, moving from bottom to top, which integrates seamlessly into the workflow of existing slicers with minimal memory and computational overhead.

Our slicer transforms a given 3D object into n contiguous slices. Each slice is a set of polygons P_ℓ where $\ell \in [0,\ n-1]$ denotes the ℓ-th slicing layer starting at the bottom of the model and proceeding upward. Our goal is to convert the straight line segments of each of these polylines into high frequency sine waves with near constant wavelength λ. Always after two layers, sine waves get phase shifted by π. To obtain the desired checkerboard pattern, waves must be aligned as well as possible between layers, i.e., peaks and valleys have to match.

Note that for general surfaces it is clearly not possible to maintain a constant λ, nor is it possible to always preserve alternate phase alignment between layers. For example, a cone will have layers comprised of circles with decreasing radii where the total number of waves decreases between layers.

Our algorithm works by placing *samples* on the polygons in each layer P_ℓ. Each sample s corresponds to a peak or valley of a wave on odd or even pairs of layers, respectively. By controlling the spacing and number of samples within a layer ℓ we can optimize for maintaining a near constant wavelength λ. By aligning samples between successive layers we can optimize the phase alignment. We employ a projection and relaxation strategy to map samples from layer to layer. Starting with the base layer P_0 we compute an optimal number of samples for all polygons $p_0^i \in P_0$ as follows:

$$k^i = \lfloor A(p_0^i)/\lambda \rfloor, \text{ where } A(\cdot) \text{ is the arc length of } p_0^i.$$

These samples are then equally spaced along p_0^i by an arc length distance of $A(p_0^i)/k^i$.

Given this initial sample assignment for the base layer, we perform both projection and relaxation on all subsequent layers until we reach the top of the model. Specifically, given an already sampled layer ℓ, we take each sample $s \in s_\ell^i$ and project it as s' onto the next layer up, unless the distance between s and s' would be greater than a certain threshold. This avoids the projection of samples in areas where slices are changing rapidly; such areas will have discontinuities anyway. The projection forms new samples $s_{\ell+1}^j$ with $j \le i$. We then relax their locations, spacing them more evenly apart while avoiding excessive phase distortion with respect to the layer(s) below. Next, we prune any samples that are spaced less than λ in arc length apart. We detect empty spans between samples longer than 2λ and insert new evenly spaced samples to fill that gap such that each new sample is spaced no closer than λ from its adjacent samples. Once new samples have been inserted and deleted we reapply relaxation to better space the samples, and then proceed to the next layer. The high level approach is summarized in Algorithm 1.

Figure 6.4 shows how this technique processes the LUCYCAT model. A great advantage of our projection scheme is that it is completely agnostic about topology changes—see for instance Figure 6.4b—and naturally handles any change of number in polygons between successive layers. After all samples have been placed on all layers, we can easily tessellate the outline

Algorithm 1 Projection–Relaxation Scheme

Input: n sliced layers, i.e., n sets of polygons
Output: locations on slices to place peaks or valleys of sine curves
  assign initial sample placement on base layer $\ell = 0$
  for each layer $\ell \in [1, n-1]$ **do**
    project samples from layer $\ell - 1$ onto layer ℓ
    relax sample locations on layer ℓ (and optionally $\ell - 1, \ldots$)
    remove samples closer than λ on layer ℓ
    insert new samples on layer ℓ in gaps larger than 2λ
    relax sample locations on layer ℓ (and optionally $\ell - 1, \ldots$)
  end for

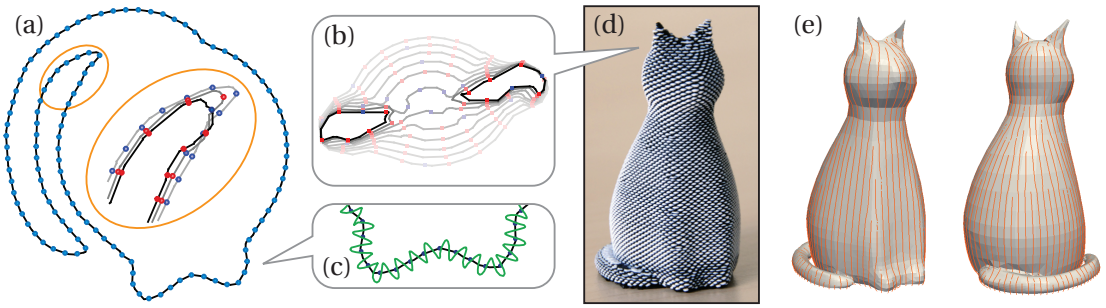


Figure 6.4: LUCYCAT *model.* (a) *Base layer 0 with initial placement of samples (blue dots).* Inset: *projection of samples around the tail onto layers 1 and 2. Projected samples are shown in red, newly inserted samples again in blue.* (b) *Projection of samples to start forming the ears.* (c) *Applying sine wave pattern to the outline.* (d) *Resulting 3D print.* (e) *Traceline visualization.*

in-between samples with a sine pattern (Figure 6.4c), which eventually produces the G-Code to 3D print the model (Figure 6.4d).

By connecting projected samples, we can visualize the entire process as a series of tracelines moving up the side of a model (Figure 6.4e). When the model narrows, some of the tracelines will dead end, and in widening regions new tracelines form. The tracelines, i.e., samples, are provably spaced between λ and 2λ apart: any samples spaced less than λ are pruned, and any samples spaced greater than 2λ do not exist since a new sample was inserted in-between. Our relaxation maintains the constraint that any two samples are bounded in distance $d \in [\lambda, 2\lambda)$ from one another, i.e., we never move samples to locations that violate this criteria.

6.4 Implementation

We implemented our dual-color mixing technique within our own slicer, however, this would be straightforward to integrate into other existing slicers: they could iteratively project and relax samples while processing layer by layer and then modulate the outlines. Our slicer is further parallelized to process the input model using multiple threads.

6.4.1 Texture Mapping

For textured input models, our slicer evaluates and stores texture coordinates for every vertex forming the polylines of the dissected model. While tessellating the outlines these coordinates are interpolated for texture lookup. The result is then used to attenuate the corresponding amplitudes. Since we have parametrized our layers with a target fixed wavelength, according to the Nyquist limit we are not able to capture textures at all frequencies. Bilinear interpolation produces aliasing on the prints if the texture resolution is too high; thus we apply trilinear interpolation *(mipmapping)* to properly prefilter textures based on our target sample spacing. For our teaser (Figure 6.1) we prefiltered the Lena image using Lanczos resampling to produce a 60×71 texture which creates a perfect mapping of texels to 71 pairs of layers.

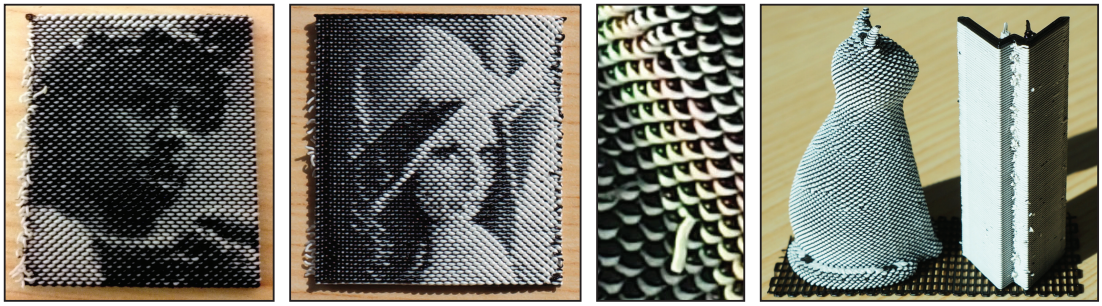


Figure 6.5: *Common artifacts when printing with two nozzles are filament strips sticking out of the model. They can be prevented by including a cleaning tower (far right) into the G-Code, which prints in parallel to the model. The cleaning tower primes nozzles and prevents salient artifacts such as superfluous filament strips, significantly improving quality.*

6.4.2 Priming Nozzles

There is at most one active nozzle at a time during the printing process, and due to the heat the inactive nozzle slowly starts extracting unwanted strips of filament. They smear the actual print, leading to salient artifacts in form of small strips sticking out of the model. Figure 6.5 reveals several of these artifacts.

Delayed filament extraction after switching nozzles is another problem. During the Lena image print, white layers started on the left, and black layers on the right. The delayed extraction leads to the impression of an exaggerated contrast.

Some consumer printers manufactured by Stratasys avoid these issues by moving the print head back to a rear corner for nozzle switching, where surplus filament is dumped and the nozzle gets primed on a wire brush. Our technique carries this over to MakerBot printers: we print a *cleaning tower* next to the model. Such a cleaning tower is directly integrated into the G-Code and its W-shaped form primes nozzles and improves its own stability. Hergel and Lefebvre [2014] go further and print a disposable shell closely around the object to avoid artifacts and improve the quality of multi-filament prints.

6.5 Results

All results were printed using a MakerBot Replicator 2X desktop printer. The layer thickness is set to 0.25 mm for high-quality prints. If two layers are shifted against each other, they transform into an interleaved chain with 0.5 mm thickness. The cylinders in Figure 6.3 are rather small and when viewed from a distance their color is perceived as a nice in-between tone. This is the key achievement of our work. Moreover, these cylinders now feature an interesting touch and feel and provide more grip. Holding these prints in the hands, their surfaces appear like woven fabric. Turning them a little exposes them to a viewing angle dependency of perceived shades, as some geometric offsets start hiding others.

LUCYCAT has a very printer-friendly shape (Figure 6.4) which does not require any support structures. Much more complex and involved is the GARGOYLE model, portrayed in Figure 6.6.

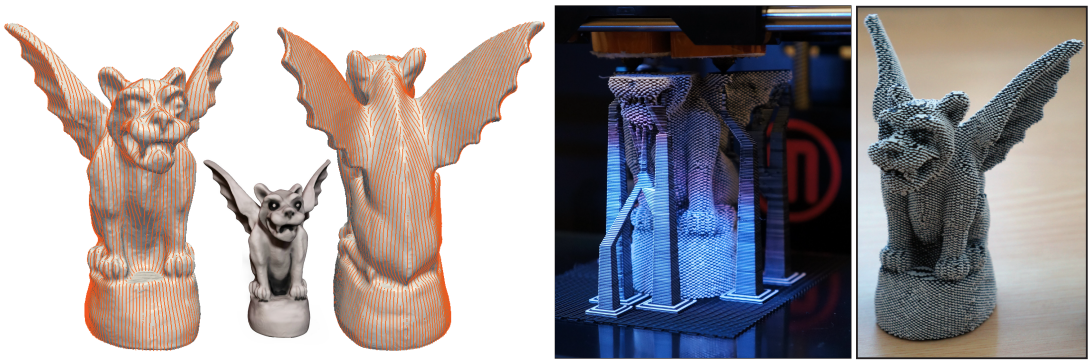


Figure 6.6: GARGOYLE *model.* From left to right: *tracelines of the samples generated with our projection and relaxation scheme; the intricate model being built during the printing process, requiring multiple support structures; the final texture mapped result.*

We texture mapped our model into a texture atlas, and used a 3D painting system to paint the cracks of this figurine to accentuate shading. This gives the model an aged look, as if dirt collected in the cracks over a long time.

We determined an optimal wavelength λ of 1200 microns to space our samples apart. We keep below a projection threshold of 600 microns to project samples up to subsequent layers. During a relaxation phase we correct 5% of the misalignment for each sample from its perfectly centered location between neighboring samples. When modulating outlines with sine patterns, the maximum amplitude is 750 microns, which can be attenuated down to 375 microns.

Listed below is more information about our models with timings how long it takes to print them using a single color, our dual-color mixing technique, and with enabled cleaning tower:

	LUCYCAT	GARGOYLE
Layers	216	408
Height	5.4 cm / 2.1 in	10.2 cm / 4.0 in
Single Color	0 h 45 m	5 h 48 m
Dual-Color	0 h 58 m	6 h 56 m
with Cleaning Tower	1 h 42 m	8 h 28 m

The additional processing time *for the slicer* to incorporate the dithering technique is negligible, particularly in relation to the actual printing time. Whether a model is textured or not does not have any impact on the duration of the prints. Our method also produces more geometry, but even for GARGOYLE the uncompressed G-Code remains below 40 MB, which nowadays is far from being a constraint.

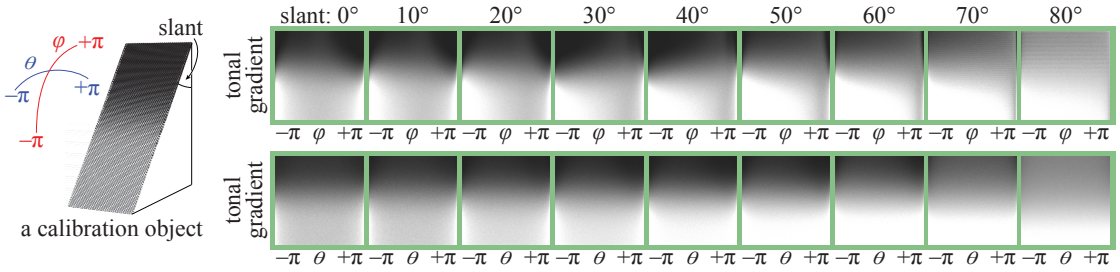


Figure 6.7: *Set of gamut maps obtained from a synthetic calibration object which was generated for various surface slopes. For varying viewing angles φ and θ the intensity response of the gradient is encoded into the corresponding column of such a map.*

6.6 Analysis

A key property of our approach is that we trade geometric detail for continuous tone imagery. We found this loss of precision acceptable for our models, but assume our input polygons have been filtered to remove high frequency geometry detail which is clearly higher than the frequency of the applied sine waves. This limits the effective curvature of our models; otherwise outlines would start to overlap with too much filament extracted at these regions, resulting in severe artifacts. We now characterize the impact of our geometric processing on the quality of the appearance of our prints.

6.6.1 View-Dependent Appearance and Gamut Specification

By observing the resulting prints we realize that their appearance is highly anisotropic. In order to investigate this issue in greater depth and in an efficient and flexible way, we set up a ray tracing environment which allows us to generate synthetic renderings of our prints. In practice we found that by modeling the geometric detail of each print down to the G-Code level, a standard global illumination engine produced results that were nearly indiscernible from the real-world prints. They adequately capture all of the larger shading impacts resulting from chosen interleaving patterns. More specifically, we found that any perceived impacts resulting from small changes to the local dielectric microfacet shading model were minor in comparison to the more macroscopic anisotropic shading impacts resulting from the geometric patterning of the filaments. Thus we found that any reasonable BRDF approximation of plastic ABS filament sufficed, whereas accurately modeling each strand of geometry was critical for achieving correct synthetic reproductions.

We developed a virtual gonioreflectometer on top of our custom rendering system and were able to rapidly analyze the anisotropic impacts of the filament patterns on the perceived shading. In Figure 6.7 we examine a synthetic calibration object, textured with a gradient map, with its plastic ABS filament approximated with a standard Phong-like shading model. The printer paths were modeled as sphere-capped cylinders, where warping was performed to distort them to have oval cross-sections to better simulate the impacts of gravity. We produce slanted variants of this object in steps of 10 degrees until the surface is almost horizontal, to show how the gamut behaves for varying surface slopes.

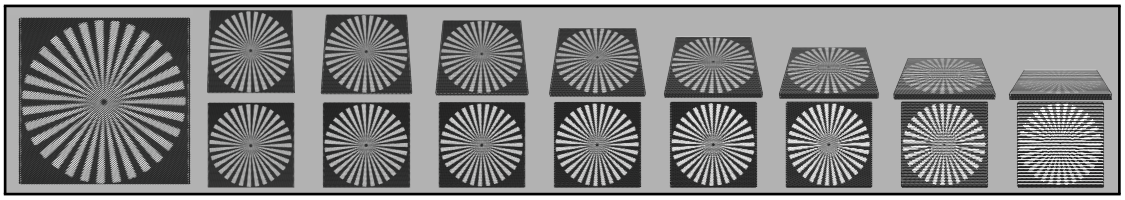


Figure 6.8: *The effective texture resolution degrades for more horizontal surfaces. We increase the slant of an* 8×8 *cm test object* (far left) *in steps of 10 degrees.* First row: *the viewing direction is fixed.* Second row: *views perpendicular to the surface normal.*

The gonioreflectometer views each slanted tile face on, i.e., the viewing direction is perpendicular to the surface normal, and then locally averages sets of pixels. This results in an intensity response of the tonal gradient. For all slanted objects we analyze the gamut for varying viewing angles, and rotate the gonioreflectometer first in a vertical fashion (varying φ; fixed $\theta = 0$), and then horizontally (varying θ; fixed $\varphi = 0$), i.e., remaining parallel to the individual layers of the object.

Each obtained *gamut map* contains one column per viewing angle encoding the measured intensity response of the tonal gradient. As expected, the high anisotropy of the perceived shade is reflected in these maps. Viewing the surface directly on (cf. center column in a gamut map) produced the largest gamut, however, the range of the gamut is limited in that it does not reach full black or white. As the material is viewed more edge on, the gamut starts to collapse and the gray level intensities move toward complete black or white. This effect also occurs when focusing on curved parts, e.g., gamut degrades when looking more toward the silhouettes of a model. The gamut dissolves completely for extremely slanted objects. Consequently, a wider gamut is only visible under a limited range of viewing angles, but it might be interesting to inspect the anisotropic appearance as future work.

6.6.2 Resolution on Slanted Surfaces

Our approach works by alternating filament colors every other layer enabling long continuous print paths. The obvious limitation with this approach is that under extreme slopes the tonal resolution degrades. We investigate this effect in Figure 6.8, where we again gradually increase the slant of a synthetic test object. In the first row, we retain a fixed camera position with a fixed viewing direction parallel to the individual layers of the object. In the second row, the viewing direction is perpendicular to the surface normal. Note that the resolution does not visibly degrade until thirty degrees from flat, and even at twenty degrees from flat the resolution print pattern is still prominently visible.

The loss of resolution on nearly horizontal surfaces could be avoided by reorienting the model within the print volume. The Lena image (Figure 6.1), for instance, prints upright vertically instead of lying flat on the build plate. This approach may not work for all models and can introduce the need for support structures, and problematic regions may still exist. These slopes can be improved by adjusting the layer thickness to the smallest amount supported by the print device, slowing down printing, but improving the effective printed resolution.

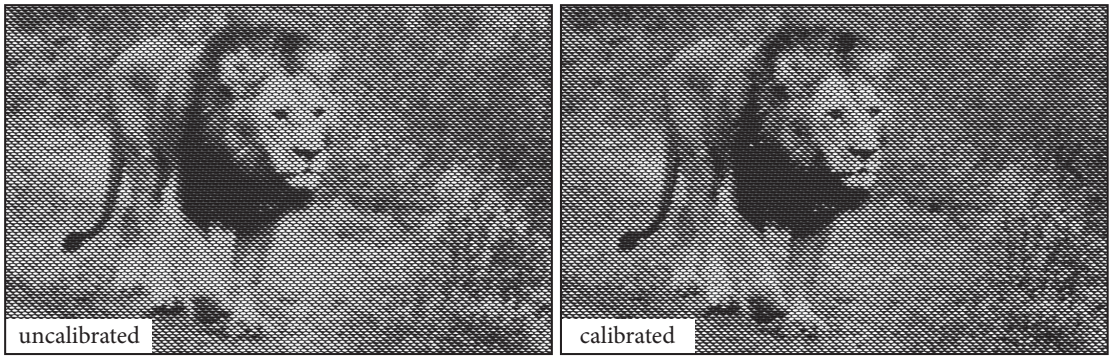


Figure 6.9: *Our calibration method improved the contrast on this print.*

For larger models, we could imagine adapting the approach from Hildebrand et al. [2013] to separate models and then aligning individual chunks to be printed vertically.

6.6.3 Calibration of Tone

Our analysis shows that tones in a given model map to different tones after interleaved printing, which is due to small-scale surface details and more importantly due to their mutual occlusion. By analogy, for display systems such as monitors and projectors, this non-linearity is called *gamma* and its compensation is known as *gamma correction*.

The virtual gonioreflectometer enables us to measure the appearance of a calibration object over a broad range of viewing angles, which we used to calibrate and optimize our prints. Given both the slope and orientation of the surface with respect to the viewer, we can calibrate our prints using a corresponding gamut map. Rather than optimizing for a specific view location (possible future work), we assume we are always locally viewing the geometry perpendicular to the surface and employ the center column of a gamut map.

This column shows how a gradient modeled from black to white is perceived after printing. For these intensities we create an inverse map, so that for a computed gray value we find an index in this column for which this value is closest. We precompute such an inverse intensity table for fast look-ups and use it to obtain calibrated prints. Figure 6.9 demonstrates the calibration effect resulting in increased perceptible detail.

6.6.4 Parametrization Seams

To analyze the generated parametrizations of our projection and relaxation scheme, we equidistantly sample all polylines of our models. We propagate these samples up to their layer above, and determine how well the desired checkerboard pattern is preserved. The divergence from the optimal pattern is then accumulated as error. A sample has a tessellation value (displacement along the surface normal) t_1 in the range from -1 to 1 as this is the result of the sine function. Its projected sample above has a tessellation value of t_2. (Assume the model is not textured and ignore the phase shifting after every two layers.) We then assess the parametriza-

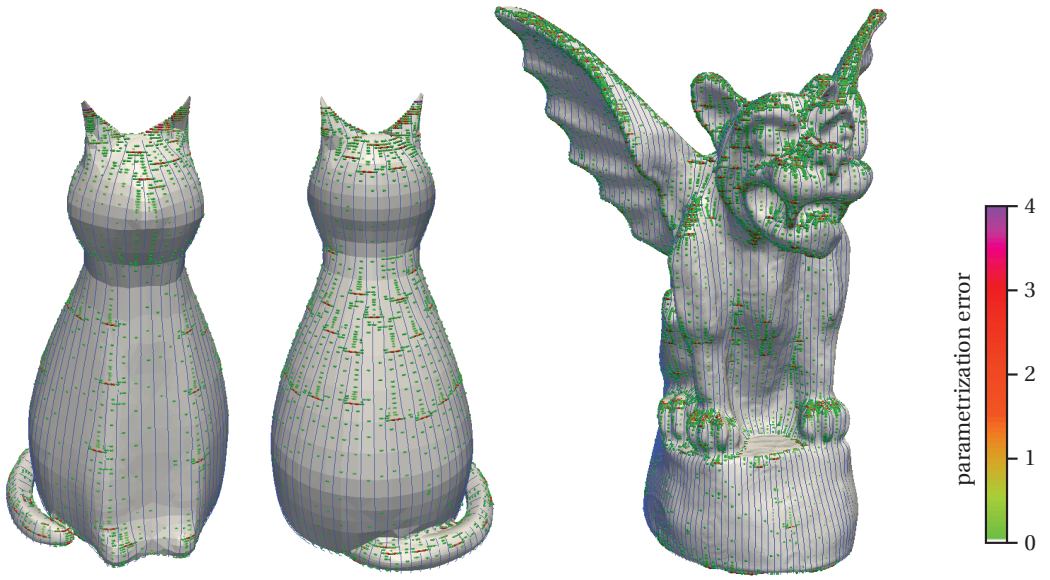


Figure 6.10: *Visualizing parametrization error: noticeable seams (in red) occur primarily where tracelines form or end.*

tion error between these samples using the metric $d(t_1, t_2) = (t_1 - t_2)^2 \in [0,4]$. Samples are discarded if they exceed the projection threshold.

Figure 6.10 visualizes parametrization seams on our test models. A transfer function maps "no error" to transparent and fades over to opaque green for an error value of 0.05, then orange at 1.5, red at 3.0, and purple at 4.0. Parametrization seams in the green range are invisible to the naked eye, but higher error leads to noticeable discontinuities. Regions where streamlines begin and end are primarily affected: there will be a horizontal seam where areas with wavelengths λ and 2λ adjoin each other, visualized in red.

The surface of LUCYCAT is to more than 80% free from error (107799 out of 133539 samples in total). Another 18% of the samples fall into the interval $(0, 0.5)$ where error is not yet perceptible. The remaining 2% of the surface area contains salient seams. This number might sound low, but it is enough to be a disturbing factor to the human eye. (Imagine, by analogy, a 10×10 image with two corrupt pixels.) An overall analysis of the GARGOYLE model results in similar numbers (79% error-free and 19% not yet perceptible errors), but the visualization reveals that error primarily accumulates at intricate sections of the head and strongly curved parts of the wings, where most seams will occur.

Besides our projection scheme, we experimented with piecewise planar projections onto the sides of a model, which, by contrast, produces vertical seams and would not be practical for intricate shapes. In another experiment, we adapted known subdivision techniques, which smoothly insert new vertices into a mesh: in areas in-between λ and 2λ, we applied different blending functions to create intermediate wave patterns. Unfortunately, they feature frequencies much higher than λ, which makes them impractical to print. On the positive side, they scatter parametrization issues evenly along the whole model, avoiding singularities.

Figure 6.11: *More textured examples.* Left: *a soda can with a high-resolution texture. The center image shows the physical 3D print.* Right: *a printed Tiki figurine spans the full range of tones.*

We have also tried modulating outlines with 3D solid noise instead of a sine pattern. Arbitrary slices through noise space are stationary and bandwidth-limited, eliminating the need for parametrization (see Subsection 2.2.3). Instead of phase shifting a curve, we inverted the noise function. Alas, this pattern does not dissolve into an in-between tone, and individual colors remain prominent. However, we can imagine mapping great visual complexity onto shapes by using colors with less contrast, e.g., brown and black, to model a tree trunk; dark and light gray to model rocks; or blue and white to simulate a water surface.

6.7 Discussion

Recent advances in 3D printing have led to a number of low-cost printers in the consumer space. Although the geometric fidelity of these printers is quite high, we are still on the way to full color printing at the consumer level. Recently low-end printers with two color heads have arrived on the market, and we demonstrated how they can produce interleaved color patterns necessary to print two-tone images. With our passive color mixing technique we achieve a greater perceived color space and can print two-tone texture mapped objects.

Figure 6.11 shows two more examples of larger prints. They feature high-resolution surface textures and span the full range of in-between tones. Textures may not exclusively consist of regions with continuous tonal variation. If a texture contains a large part of solid color which can be perfectly represented by one of the used filaments, it would be worthwhile to instruct the slicer that the corresponding chunk of the model gets printed using only that filament. This would be compatible with our method, but requires nozzle switching within a layer.

This method could be extended to exploit three (or more) nozzles. Our technique works very well for mostly vertically aligned shapes and should be broadened to account for rather horizontal surfaces, while investigating how to alter geometry in order to work against viewing angle dependency. There is a large body of research on parametrization methods which could be applied to further enhance mapping quality on arbitrary surfaces. We believe our work

will find many applications in new passive color mixing techniques, and even for single color prints, it can be used to create interesting geometric patterns on the shapes of models to give them a new tactile finish.

We have now also reached the end of the first part of this thesis.

We began with an overview of procedural modeling techniques and then presented an interactive modeling environment for implicit surfaces. We further described a novel cache mechanism to store evaluations of complex noise functions directly on graphics hardware. Then we transferred our modeling environment to the world of 3D printing and explored novel techniques for this new type of output medium. We will now close the *geometry* part and continue with the second part of modeling virtual scenes, which focuses on *light*.

Part II

Visual Illumination Inspection Techniques

7

Fundamentals of Light Transport in Computer Graphics

What is a (photo-)realistic computer-generated image? We have already addressed this key question directly at the beginning of Chapter 2. We have pointed out why visual complexity, i.e., rich geometric detail, is so important to model realistic scenes. This is, however, only one part of the equation. The remaining challenge is the realistic *illumination*, and, associated with it, the realistic *rendering* of such a scene. To approach this problem, we have to provide a better understanding how light interacts with matter.

This is why we now give an introduction to the fundamentals of light transport in computer graphics. We start with geometrical optics as a simple model to describe light propagation. Then we go through fundamental radiometric quantities and their photometric counterparts. Next, we show how light interacts with surfaces and participating media. This will then lead us to the *rendering equation*. We also briefly describe the global illumination methods we will subsequently use for rendering. An important section follows: we explain how the light in our environment can be interpreted as a high-dimensional light field and introduce the *plenoptic function* as a concept to sample a light field for any given position and orientation in space.

This chapter forms the theoretical framework for the second part of this thesis, where we simulate light transport using global illumination methods. This allows us to describe novel techniques to visualize *light transport* in virtual and real-world scenes. We also briefly discuss how these techniques can be combined with the *artistic manipulation* of light transport.

7.1 Geometrical Optics

In the first place, we need a conception of light itself and the way light propagates through space and interacts with matter. The comprehensive theoretical groundwork exists for a long time and includes the Maxwell equations, the special theory of relativity, and quantum mechanics. This allows us to comprehend how light interacts with matter both on the large and small scale. Fortunately, it turns out that it is almost always sufficient to rely on much simpler models in computer graphics.

Geometrical optics (or *ray optics*) is one of the simplest models to describe light propagation and we can trace its origin back to ancient Greece. It makes the simplified assumption that light particles travel along straight lines. This neglects the concept of wave-particle duality

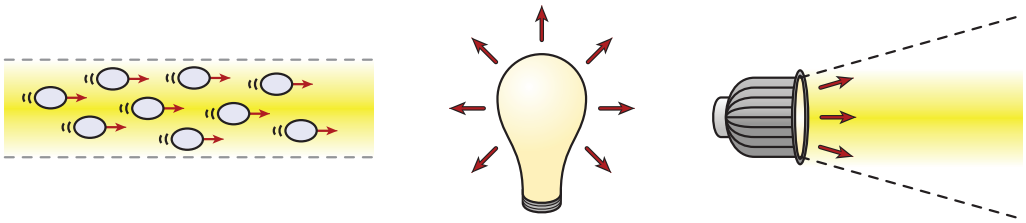


Figure 7.1: Left: *we think of straight light rays as a stream of photons which carry radiant energy.* Center: *the radiant flux is the amount of energy emitted (or transmitted, or received) per unit time.* Right: *radiant intensity is the radiant flux per unit solid angle.*

which describes the behavior of quantum-scale objects, however, the ramifications are usually indiscernible to the human eye. Geometrical optics allows us to use *rays*, i.e., straight lines, as primitives. Rendering methods such as ray tracing [*Whitted* 1979] emerged from this model.

Other principles of geometrical optics include that the direction in which light travels along a ray is of no significance. This is known as the *Helmholtz reciprocity* and allows ray tracing to spawn rays at the position of the camera and trace them backwards to the light sources. Moreover, light rays are allowed to cross and they do not interfere with each other in free space. They are *reflected* on glossy and specular surfaces such as a mirror according to their incident angle and the surface normal. Light rays are also *refracted* according to Snell's law when they travel through the interface of two media with different indices of refraction.

7.2 Radiometry and Radiometric Quantities

Our visual system enables us to interpret our environment using our eyes as a sense organ. The *retina* is the inner coat of our eyes and consists of *rod* and *cone cells*. These *photoreceptor cells* are light-sensitive and respond to the visible light. The light itself is a form of *electromagnetic radiation*. Radiometry is the study of measuring such electromagnetic radiation.

Overview. We will now introduce fundamental radiometric quantities (radiant energy, flux, intensity, irradiance, and radiance). See Figures 7.1 and 7.2 for associated illustrations.

Radiant energy. Electromagnetic radiation consists of electromagnetic waves, which are *radiating* through space. These waves carry *radiant energy*. We will use the symbol Q to denote radiant energy. Its unit is *joule* (J).

Note that we still use the terms *wave* and *wavelength*, even though we use the simplified model of geometrical optics. In our case, we think of light traveling along straight lines as a *stream of photons*. We simply *assign* wavelengths as an attribute to straight light rays and individual photons. Such a photon then carries the radiant energy $Q = h \cdot f$, where h is the Planck constant and f the frequency of the electromagnetic wave. The frequency in turn is $f = c/\lambda$, where c is the speed of light and λ the wavelength. For monochromatic light, the radiant energy is then dependent on the amount of photons N: $Q = N \cdot h \cdot f$.

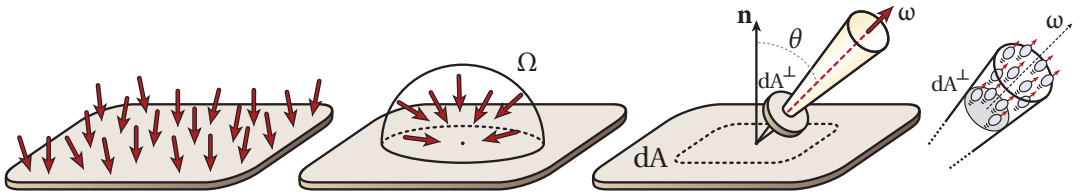


Figure 7.2: Left: *another illustration of radiant flux, here for a receiver.* Center: *irradiance is the radiant flux per unit area.* Right: *radiance is the radiant flux per projected unit area per solid angle, i.e., the energy of the photons which travel within this solid angle per unit time.*

Radiant flux. Taking into account the *time* of some radiant energy transport results in the *radiant flux* (or *radiant power*). We will use the symbol Φ to denote radiant flux:

$$\Phi = \frac{\partial Q}{\partial t} \quad \left[\mathrm{W} = \frac{\mathrm{J}}{\mathrm{s}}\right] \quad .$$

Radiant intensity. We can now relate the radiant flux to the solid angle, which gives us the *radiant intensity:*

$$I = \frac{\partial \Phi}{\partial \Omega} \quad \left[\frac{\mathrm{W}}{\mathrm{sr}}\right] \quad .$$

Irradiance. We can also relate the radiant flux to the area of an illuminated surface. This results in the *irradiance:*

$$E = \frac{\partial \Phi}{\partial A} \quad \left[\frac{\mathrm{W}}{\mathrm{m}^2}\right] \quad .$$

Radiance. The *radiance* is typically considered as the most important radiometric quantity in computer graphics. It is the radiant flux emitted, reflected, transmitted, or received by a surface, per solid angle and per projected area, and therefore determines how bright a surface will be perceived by an observer:

$$L = \frac{1}{\cos\theta} \cdot \frac{\partial^2 \Phi}{\partial A \, \partial \Omega} \quad \left[\frac{\mathrm{W}}{\mathrm{sr} \cdot \mathrm{m}^2}\right] \quad .$$

We see how integrating the radiance over the whole hemisphere Ω leads back to the irradiance:

$$E = \frac{\partial \Phi}{\partial A} = \int_\Omega L \cdot \cos\theta \; \mathrm{d}\omega \quad .$$

7.3 Photometry and Photometric Counterparts

Radiometry covers the measurement of the entire spectrum of electromagnetic waves, from radio waves, microwaves, infrared, visible, and ultraviolet light, up to X-rays and gamma rays. *Photometry*, on the other hand, takes into account the sensitivity of the human eye and focuses on the portion of the visible electromagnetic spectrum.

The luminosity function. The human eye is only sensitive to the visible spectrum of electromagnetic waves. This sensitivity is expressed in the *luminosity function* (or *luminous efficiency function*). This function peaks at unity at the wavelength where the human eye is most sensitive to light. Moving towards the borders of the visible spectrum, the function value decreases, as it requires a higher intensity of incoming light to perceive the same brightness. For wavelengths outside the visible spectrum, the function returns zero.

In practice there are two different luminosity functions used, one for *photopic vision* and one for *scotopic vision*. The photopic luminosity function is used for well-lit conditions such as daylight and peaks at unity at 555 nm. The eye adapts to lighting conditions and responds differently to low levels of light. We then apply the scotopic luminosity curve (peak at 507 nm).

Weighting a radiometric quantity with the luminosity function returns its photometric counterpart. We briefly introduce the most relevant photometric quantities for our work next.

Luminous energy. The *luminous energy* is the radiant energy weighted by the luminosity function. This implies that visible light in a range where the human eye is most sensitive will carry the highest amount of luminous energy, whereas even strong X- or gamma rays with a high radiant energy carry zero luminous energy.

We use the symbol Q_v to denote luminous energy. (The symbols of photometric quantities will be identical to their radiometric counterparts, with an additional index "v" as in "visual".) Its unit is *lumen second*. The unit *lumen* itself applies to the *luminous flux*, described next. We can integrate over time $[0, T]$ to determine the luminous energy from the luminous flux:

$$Q_\mathrm{v} = \int_0^T \Phi_\mathrm{v}(t)\,\mathrm{d}t \quad .$$

Luminous flux. The *luminous flux* describes the power of a light source and how much visible light it radiates. Its unit is *lumen* (lm). We can determine the luminous flux by integrating the radiant flux over all wavelengths, weighted by the luminosity function V:

$$\Phi_\mathrm{v} = 683.002 \frac{\mathrm{lm}}{\mathrm{W}} \int_0^\infty V(\lambda)\,\Phi(\lambda)\,\mathrm{d}\lambda \quad .$$

Here, we use $K_\mathrm{max} = 683.002\ {}^{\mathrm{lm}}/_{\mathrm{W}}$ as an additional scaling factor to transfer from radiant to luminous flux. It is the maximum value of the *luminous efficacy*, which is reached for an ideal monochromatic light source with $\lambda = 555$ nm under photopic vision. The luminous efficacy is defined as $K = {}^{\Phi_\mathrm{v}}/_{\Phi}$ and describes how much of the emitted radiation actually falls into the visible spectrum, in other words, how well a light source emits radiation that is visible.

Luminous intensity. The *luminous intensity* is the luminous flux per unit solid angle. Its unit is *candela* (cd). It is a useful quantity to describe a light source which emits different amounts of light into different directions, for instance, a spotlight.

Illuminance. The *illuminance* is the luminous flux per unit surface area. Its unit is *lux* (lx), which is lumen per square meter. As a side note: we often come across the term *exposure* in photography. The luminous exposure is the illuminance of the image plane times the exposure duration, i.e., it is measured in $\mathrm{lx} \cdot \mathrm{s}$.

Luminance. The *luminance* is the luminous intensity per unit area, or, vice versa, the illuminance per unit solid angle. Its unit is candela per square meter. It describes the amount of light for a given solid angle that is emitted or reflected from a surface. As luminance exists everywhere in space, it can also be used to describe the amount of light which passes through a virtual area around a point in space.

The luminance, just as its radiometric counterpart, the radiance, is most closely linked to our visual perception. The luminance of a surface gives us an indication of the brightness the human eye perceives.

Summary. The following table summarizes all discussed quantities:

Radiometric quantity	Symbol	Unit	**Photometric quantity**	Symbol	Unit
Radiant energy	Q	J	Luminous energy	Q_v	$\mathrm{lm}\cdot\mathrm{s}$
Radiant flux	Φ	W	Luminous flux	Φ_v	lm
Radiant intensity	I	$\mathrm{W/sr}$	Luminous intensity	I_v	cd
Irradiance	E	$\mathrm{W/m^2}$	Illuminance	E_v	lx
Radiance	L	$\mathrm{W/sr\cdot m^2}$	Luminance	L_v	$\mathrm{cd/m^2}$

7.4 Interactions with Surfaces and Participating Media

By now we have discussed how we use geometrical optics to model light propagation. We have also presented the fundamental radiometric and photometric quantities to describe the energy that light rays (streams of photons) carry. Unless we are dealing with *participating media* (which we address at the end of this section), we can make an important assumption: the *conservation of radiance* (and luminance as a consequence) when light travels through empty space. It means in effect that we neglect our surrounding air and instead treat empty space as a vacuum where radiance does not decrease. The amount of exitant radiance emitted from a light source will then be identical to the amount of incident radiance received on a surface from that light source, for light which travels along a straight line between source and receiver, and which is not blocked by any occluders on its way.

What happens, however, when light hits a surface? It may get partially (or completely) *absorbed*. The surface then absorbs the energy of a photon. This means that electromagnetic energy is transformed into internal energy, such as thermal energy, of the surface matter.

Light can also be *reflected* on a surface. A naive assumption would be that light scatters uniformly when it hits a surface. This is true for a *Lambertian reflection*, which is perfectly diffuse. A Lambertian surface exhibits an isotropic radiance. The radiant intensity obeys Lambert's cosine law and is directly proportional to the cosine of the angle between the incident light and the surface normal. This implies that a fixed point on a Lambertian surface is always perceived with the same brightness independent of the observer's position and viewing angle.

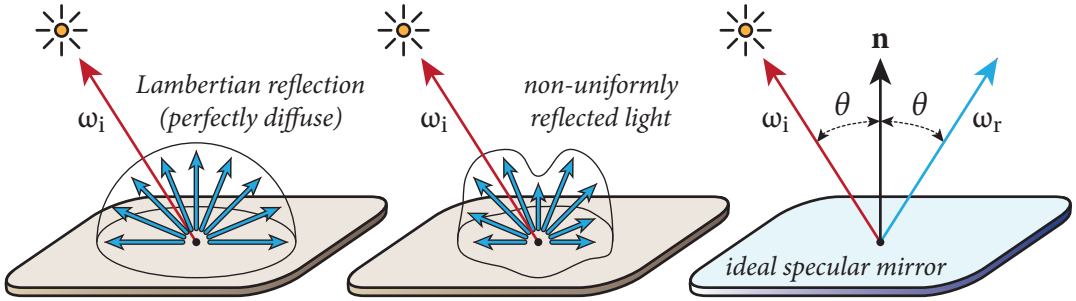


Figure 7.3: *Illustrations of bidirectional reflectance distribution functions.* Left: *a Lambertian (ideal diffuse) surface reflects incoming light uniformly in all directions of the hemisphere.* Center: *a typical surface reflects light in a non-uniform way.* Right: *ideal specular reflection.*

The bidirectional reflectance distribution function

In reality, however, objects are composed of various materials which do not reflect light in a perfectly diffuse way. To describe how light is reflected on a surface for a certain incident direction ω_i and exitant (reflected) direction ω_r, we make use of the *bidirectional reflectance distribution function* (BRDF). This function was introduced by Nicodemus [1965] and is pivotal for realistic computer graphics. In its basic form, the BRDF is defined as:

$$f_r(\omega_i, \omega_r) = \frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}E_i(\omega_i)} = \frac{\mathrm{d}L_r(\omega_r)}{L_i(\omega_i)\ cos(\theta_i)\ \mathrm{d}\omega_i} \quad ,$$

where "i" and "r" are indices for "incident" and "reflected", and θ_i is the angle between ω_i and the surface normal. We see that the BRDF returns the ratio of reflected radiance along ω_r to the surface irradiance along ω_i.

Moreover, physically correct BRDFs exhibit the following three properties: the result is always positive; the Helmholtz reciprocity applies (see Section 7.1 and [*Rayleigh* 1900]), i.e., $f_r(\omega_i, \omega_r) = f_r(\omega_r, \omega_i)$; and energy is conserved, i.e., $\forall \omega_r : \int_\Omega f_r(\omega_i, \omega_r)\ \cos\theta_i\ \mathrm{d}\omega_i \leq 1$.

Figure 7.3 illustrates the reflections for perfectly diffuse, general, and ideal specular surfaces. While it is straightforward to describe perfectly diffuse and specular reflections analytically, the question remains how to express the BRDF of a particular surface material. Typical approaches to express a BRDF are usually categorized as *empirical*, *physical*, and *experimental*.

One of the earliest empirical approaches, still regularly used today, is the *Phong reflection model* [1975]. Perhaps the best-known model in computer graphics, it includes both diffuse and specular reflections, and further brightens a surface using a constant ambient term to approximate global illumination in a very coarse way. Such an empirical model tries to emulate the surface reflection by close observation, in contrast to physical (or theoretical) models [*Cook and Torrance* 1982; *He et al.* 1991], which are based on much more computationally expensive, accurate physical models.

An experimental approach to acquire a BRDF is to use a *gonioreflectometer*, which gradually measures the ratio of reflected light on a surface for a discrete set of incident and exitant angles that covers the hemisphere. Such measurements take a long time and lead to large records of acquired reflection data, but result in a very accurate and realistic description of a surface

reflection for one specific material. Using a BRDF *fitting* method, a target (analytical) BRDF model can be parametrized to reflect such acquired data as well as possible, according to a particular metric. Such a metric could be perceptually motivated or, for instance, based on the root mean square error of input and output values.

We refer to Cook and Torrance [1982] and Ngan et al. [2005] for background information about empirical and experimental BRDFs. We further refer to Westin et al. [2004] and Montes and Ureña [2012] for comprehensive overviews and comparisons of different BRDF models.

Participating media

We have just discussed how light interacts with surfaces. There we have applied the conservation of radiance by treating empty space as a vacuum. In reality, however, the space between objects is usually filled with air or liquids. The air in our surroundings contains impurities and particles which interact with photons. It means in effect that light travels through *participating media*. It is an acceptable simplification to put clean air on a level with vacuum when light travels short distances. It is also sufficient for basic computer graphics scenarios with limited spatial extent, such as indoor scenes.

The longer light travels through air, however, the more scattering occurs. We can directly observe this effect for sunlight which travels through the atmosphere. There, oxygen and nitrogen molecules scatter short-wavelength light (violet and blue) up to an order of magnitude more often than long-wavelength light (yellow and red). This particular scattering is called *Rayleigh scattering* and causes *diffuse sky radiation*. This is why the daytime sky appears blue. It also explains the yellow and reddish hues of the sun at sunset.

In particular, participating media covers aerosols such as fog, steam, smoke, dust, and haze. Those interact heavily with light and the radiance may no longer remain constant along a light ray. Now we require more sophisticated models for light transport which take effects such as absorption, emission, *out-scattering*, and *in-scattering* into account.

When light travels through participating media, a photon might interact with a particle. This probability increases with increasing density or size of the particles. Just like a photon can be absorbed on a surface, it can also be absorbed by a particle and converted into another form of energy, again, such as heat. The probability that such an absorption event occurs is described using the *absorption coefficient* σ_a. In addition to that, media such as *fire* further *emit* light, where other forms of energy are converted to visible light.

Another possible type of interaction is that a photon is *scattered* during a collision with a particle. In the case of an *out-scattering* event inside a participating medium, an incoming light ray loses radiance along its direction, while the radiance along the direction of the deflected photon increases. Again, the probability of such an event is described using the *scattering coefficient* σ_s. As both absorption and out-scattering lead to a loss of radiance along a light ray, these events form the *extinction coefficient:* $\sigma_\mathrm{t} = \sigma_\mathrm{a} + \sigma_\mathrm{s}$.

In-scattered radiance along a light ray, by contrast, covers all radiance which is deflected *into* this ray. It can be determined by integrating incoming radiance over all directions, weighted by a *phase function* $p(\mathbf{x}, \omega_\mathrm{i}, \omega_\mathrm{r})$, since in-scattered radiance is unlikely to be isotropic in a participating medium. Instead, a phase function is dependent on a point $\mathbf{x}$ inside the medium and describes the scattering distribution of incoming and scattered directions ω_i and ω_r.

Rendering with participating media is in itself a large part of computer graphics research. Even though we will address participating media at certain points during the next chapters, it is not taken into consideration by the techniques presented in this thesis. This is why we refer to [*Cerezo et al.* 2005; *Jarosz* 2008] for a comprehensive overview and for further reading.

7.5 The Rendering Equation

After introducing the fundamental radiometric quantities and the BRDF as a model to describe surface reflections, we are now able to compose the *rendering equation* [*Kajiya* 1986; *Immel et al.* 1986]. It describes the outgoing radiance at a point as the emitted plus reflected radiance:

$$\underbrace{L_\mathrm{o}(\mathbf{x},\omega)}_{\text{outgoing}} = \underbrace{L_\mathrm{e}(\mathbf{x},\omega)}_{\text{emitted}} + \int_{\Omega^+} \underbrace{f_\mathrm{r}(\mathbf{x},\omega',\omega)}_{\text{BRDF}} \underbrace{L_\mathrm{i}(\mathbf{x},\omega')}_{\text{incident}} \underbrace{(\omega'\cdot\mathbf{n})}_{\text{foreshortening}} \,\mathrm{d}\omega' \ .$$

Here, L_o is the total outgoing radiance at the point $\mathbf{x}$ in space along the direction ω. It is the sum of the radiance L_e which is emitted at the same point into the same direction and the reflected radiance at this point. The integral covers the reflected radiance: it adds up the incident radiance L_i at point $\mathbf{x}$ from all incoming directions ω', multiplied with a BRDF and the foreshortening factor. The incoming directions are within the unit hemisphere Ω^+ centered at $\mathbf{x}$ with the normal $\mathbf{n}$. The BRDF takes into account how much radiance is reflected at $\mathbf{x}$ from an incoming direction ω' into the outgoing direction ω. The foreshortening factor $\omega'\cdot\mathbf{n}$, also frequently written as $\cos\theta_\mathrm{i}$, weakens the radiance according to the incident angle θ_i.

Note that the rendering equation can be written in different forms and its depiction differs widely throughout literature. In its original form, it describes how much radiance reaches point $\mathbf{x}$ from another point $\mathbf{x}'$, factoring in all other points $\mathbf{x}''$ which reflect light from $\mathbf{x}'$ to $\mathbf{x}$. It can also be extended by λ and t to account for individual wavelengths and time.

Historically, the rendering equation has been presented *after* the first global illumination methods were already known. However, Kajiya [1986] proved that all preceding methods can be directly derived from the rendering equation. We will give an overview in the next section.

7.6 Overview of Global Illumination Techniques

We have addressed the creation of geometry in the first part of this thesis. Most of the time the focus was on an individual shape or object. For this use case it was sufficient to rely on a *local* illumination model for rendering. This covers *direct* illumination which considers the light that is emitted directly from a light source. What distinguishes *global* illumination (GI) is the additional inclusion of *indirect* illumination, which takes into account the interaction between all objects of a scene. In particular, this enables the rendering of more complex effects such as diffuse interreflections and caustics.

Strictly speaking, we have already implemented some rudimentary global illumination effects back in Chapter 3, Section 3.5.3. This refers to the casting of shadow rays and the integration of soft shadows and ambient occlusion. However, we have relied on clever approximations that produce plausible and nice-looking, albeit not physically accurate effects.

There is a variety of GI methods that compute realistic lighting and produce more photo-realistic renderings. Since it is virtually impossible to solve the rendering equation analytically for anything but the simplest scene, the GI methods apply different numerical approximations. These approaches are usually segmented into two principal categories: *finite element* and *Monte Carlo methods*. (See Chapter 10 for *image-based rendering* as a different option.)

Finite element methods

Radiosity is the best-known application of the finite element method to solve the rendering equation. The first radiosity method was transferred from the field of heat transfer to computer graphics by Goral et al. [1984] and assumes that all surfaces are Lambertian reflectors.

In a first step all surfaces are subdivided into small patches (the finite elements). The higher the degree of subdivision, the higher the quality of the results and the duration of computation. The method then computes *form factors* for each individual pair of patches. A form factor describes how well two patches can see each other and if the space in-between is (partially) obstructed. It ranges from zero, where no energy is exchanged between two patches, to one, where all energy is exchanged. These form factors are then inserted as coefficients in a system of linear equations. Solving the linear system determines the radiosity for each surface patch. The radiosity corresponds to the perceived brightness and is the combined emitted and reflected energy which leaves a surface patch per unit area and unit of time.

Solving the system of linear equations is computationally expensive and takes a long time for more complex scenes. Moreover, a scene with n patches requires an $O(n^2)$ memory storage for all form factors. *Progressive radiosity* [*Cohen et al.* 1988] is an iterative approach to address these problems. This reformulated version of the original algorithm creates approximate solutions which progressively converge toward the final image. Hanrahan et al. [1991] proposed a *hierarchical radiosity* algorithm, which significantly reduces memory requirement and the amount of form factors needed. This approach is further extended in the work of Smits et al. [1994] which creates a new layer of clusters above the patch hierarchy.

The radiosity method handles diffuse interreflections well. Since all surfaces are Lambertian reflectors, the lighting in a scene is independent from the viewpoint. This allows to precompute the radiosity for a scene once and then use it for interactive walkthrough environments. It is one of the reasons why the radiosity method was well received for applications in architecture.

On the other side, the original radiosity method is unable to handle glossy and specular surfaces, together with resulting more complex phenomena such as caustics. Although respective extensions have been proposed [*Rushmeier and Torrance* 1990], and radiosity still being an inherent part of many of today's rendering applications, it can now be considered a rather historical method. In computer graphics research, it has been gradually superseded by Monte Carlo methods, which are able to handle all lighting phenomena in a natural way.

Monte Carlo methods

Monte Carlo algorithms apply *repeated random sampling* to approximate results. In practice they can be used to solve an otherwise completely intractable task. Kajiya [1986] presented *path tracing* as a numerical method to solve the rendering equation.

For each pixel in the image plane, path tracing continues to shoot rays into the scene, accounts for all interactions, and accumulates the contribution of each path to the final image. This means in effect that repeated random sampling is applied to the rendering equation. A welcome side benefit is that multiple samples naturally lead to anti-aliased images. On the downside, the output image suffers from clearly recognizable noise, which gradually and slowly decreases until a high number of samples is reached for every pixel: given N samples, the error estimation decreases in the form of $1/\sqrt{N}$.

What sets path tracing apart from Whitted-style ray tracing [1979] is that it accounts for all types of light paths. This can be best explained using the Heckbert notation [1990]. It allows to describe a full path from a light source (L) to the eye (E), with diffuse (D) or specular (S) interactions in-between. Every light path then matches the regular expression $L(D|S)^*E$.

These arbitrary combinations of diffuse and specular interactions allow for optical phenomena such as caustics. They appear after multiple specular reflections are projected on a diffuse surface: LS^+DE. Whitted's ray tracing algorithm, by contrast, terminates with a diffuse reflection: LDS^*E. The substring S^+D does not match DS^*. This is why caustics are supported by path tracing, but not ray tracing. The same applies to diffuse interreflections, which can be expressed as D^+. They are naturally supported by path tracing and the radiosity method we have discussed before, which handles all paths of the form LD^*E.

Path tracing, together with an accurate scene description and using realistic BRDFs, allows to render photorealistic images. It is an *unbiased* rendering method: the results are often used as ground truth, or reference images, for comparison with other rendering methods. We have already addressed that this requires a high number of samples until we have converged to a noiseless image. That is why improved versions of the algorithm have been presented, notably *bi-directional path tracing* [*Lafortune and Willems* 1993] and *Metropolis light transport* [*Veach and Guibas* 1997]. Both approaches aim for a faster convergence while sampling the rendering equation. In addition to shooting rays starting at the camera, bi-directional path tracing simultaneously generates rays starting at a light source. These subpaths are then connected to form a full light path. Metropolis light transport tries to mutate already found paths to create new, different paths, for instance, by adding, removing, or displacing vertices. A path mutation is either accepted or rejected, which is determined by comparing the relative contributions of the initial and mutated path to the image. This works well for intricate lighting scenarios where transport phenomena exist which are difficult to explore with other strategies.

Photon mapping

Photon mapping [*Jensen* 1996] is a two-pass approach which had significant impact on global illumination methods. It can be used, for instance, in combination with path tracing to reduce the rendering time for a result of comparable quality. A *photon map* is constructed in the first pass. The algorithm traces photons emitted from a light source. These photons may be absorbed, reflected, or refracted. Upon a surface interaction, a photon is stored in the photon map. The actual rendering takes place in the second pass, where the photon map is used to estimate global illumination. The stored photons close to an intersection point then contribute to the radiance at this point. Unlike the previous approaches we have discussed, this is now a *biased* method. This means that an image will not converge to the exact reference solution.

Progressive photon mapping

Hachisuka et al. [2008] presented *progressive photon mapping* (PPM). This extension to the original photon mapping approach is now a multi-pass algorithm: after the initial ray tracing pass an arbitrary number of photon tracing passes follows, iteratively updating the result. Each pass increases the accuracy of the global illumination simulation. PPM also introduced a new radiance estimate and a solution now converges to the correct radiance values.

Hachisuka and Jensen [2009] then presented *stochastic progressive photon mapping* (SPPM), which is in turn another extension to the previous PPM approach. It is more effective for rendering distributed ray tracing effects such as depth-of-field, motion blur, or glossy reflections. This now requires to compute the average radiance value over the region (instead of an isolated point) of such a distributed ray tracing effect.

Kaplanyan and Dachsbacher [2013] further analyze and derive optimal parameters for the radiance estimation of PPM. This *adaptive progressive photon mapping* approach minimizes the overall error while converging to a final result, by automatically and locally balancing noise and bias via an adaptive data-driven gather radius selection. This improves the visual quality for both progressive rendering applications and almost converged images.

The PPM algorithm, together with a new stochastic spatial hashing scheme, has been efficiently ported to the GPU by the original authors [*Hachisuka and Jensen* 2010] and further improved by Knaus and Zwicker [2011]. These techniques are especially suited for interactive applications, and we will be using them for rendering in the following chapter.

7.7 The Light Field and the Plenoptic Function

In this more conceptual section, we will discuss the *light field* [*Gershun* 1939] as a way to interpret the light in our surroundings as a high-dimensional field. This gives a better understanding of how we will simulate, track, and reconstruct the light transport in subsequent chapters. For every point in space, the light field describes how much radiance is transported along rays for all possible directions. We can sample this field using the *plenoptic function:*

$$L(x, y, z, \theta, \phi, \lambda, t) \ .$$

This seven-dimensional function, introduced by Adelson and Bergen [1991], returns the radiance for a position (x, y, z) in space along a ray with the orientation (θ, ϕ). It can further be dependent on the wavelength λ and time t. Access to the plenoptic function and the full light field means in effect that we are able to reconstruct arbitrary images of a scene for any given position, orientation, and point in time.

This ability will be pivotal for the upcoming chapters. Whenever we are interested in the light transport in key areas of a scene, we will begin to track interactions in the surroundings. Conceptually this is equivalent to locally sampling and reconstructing the light field.

The light field and its applications have regained research interest with the seminal works of Levoy and Hanrahan [1996] and Gortler et al. [1996]. This includes more robust approaches to image-based rendering which we will discuss later in Chapter 10.

7.8 Further Reading

We have now discussed the fundamentals of light transport in computer graphics, along with the essential global illumination methods. We complete this chapter by referring to a broad range of literature which gives a much more comprehensive overview and keeps track of all the various approaches to global illumination and image synthesis.

Focus on a single rendering method

Early work edited by Glassner [1989] gives a first introduction to ray tracing. It covers the essential algorithms for ray tracing and surface interactions, addresses sampling, reconstruction, and basic acceleration techniques, and finally shows how to implement a full ray tracer. The radiosity method is specifically targeted in Cohen et al. [1993] and Sillion and Puech [1994]. Both textbooks first explain the theoretical framework of global illumination and then provide a more practical set of instructions to implement integral components of the radiosity method. In his revised textbook, Jensen [2012] thoroughly explains the photon mapping approach, together with practical implementation details and optimization strategies.

Covering a variety of global illumination methods

The comprehensive textbook by Dutré et al. [2006], like many other, begins with the basic physics of light transport. It discusses the different strategies to solve the rendering equation and then consecutively explains a set of the most important global illumination algorithms. Dachsbacher and Kautz [2009], together with Ritschel et al. [2012], give another broad overview of global illumination methods, with a focus on interactive, GPU-based techniques, and on fully dynamic scenes. Targeting Monte Carlo methods, recent work by Zwicker et al. [2015] discusses adaptive sampling and reconstruction strategies to improve rendering quality.

More extensive textbooks

In a two-volume book, Glassner [1994] covers a broad range of image synthesis principles. It begins with a more thorough and in-depth introduction to the human visual system, how our surroundings can be understood as signals, and hence details sampling and reconstruction techniques. The remaining chapters again deal with the physics of light transport and global illumination techniques. The work of Shirley et al. [2009] covers all fundamentals of computer graphics which are relevant for both parts, *geometry* and *light*, of this thesis. The final chapters also focus on the basics of visualization. The *Physically Based Rendering* textbook [*Pharr et al.* 2016] gives an in-depth explanation of the mathematical theory behind the creation of imagery which can be indistinguishable from actual photos. Simultaneously, it describes a practical implementation of a full-featured photorealistic rendering system.

8

Selective Inspection and Interactive Visualization of Light Transport in Virtual Scenes

This chapter presents novel interactive visualization techniques to inspect the global light transport in virtual scenes. First, we propose a simple extension to photon mapping to gather required lighting information. We then introduce a set of five light inspection tools which process this data to provide further insights. Associated visualizations help the user to comprehend how light travels within a scene, how the lighting affects the appearance of a surface, and how objects cause optical effects such as caustics. We implemented all tools for direct usage in real production environments. Rendering is based on progressive photon mapping, providing interactivity and immediate visual feedback. We conducted a user study to evaluate all techniques in various application scenarios and discuss their individual strengths and weaknesses. Moreover, we present feedback from domain experts.

8.1 Introduction

We already see a lot of things when we have a close look at a final rendered image: we gain insight into a scene and its geometry, and may become aware of the materials used. Many settings of lighting are evident, especially for viewers familiar with computer graphics, yet we are often lacking "the big picture": how was this particular shading achieved and why is there a strong caustic? Generally speaking: in what way is light traveling within this scene?

Visualizing the transport of light is a powerful way to provide answers to these questions, but it is also a difficult task: light propagates through space in every possible direction at the same time and is reflected at surfaces or scattered within participating media. Being able to convey this information in a visual and meaningful way can be of great help for architects, engineers, lighting designers, and maybe even for graphics researchers when working on global illumination methods. Recent work demonstrates that artistic light manipulation [*Kerr et al.* 2010; *Ritschel et al.* 2010; *Ritschel et al.* 2009] can be intuitive when the user easily grasps the phenomena which are modified. The visualization of complex light transport can thus foster the development of more intricate manipulations.

Figure 8.1: *Our light inspection tools provide key information about the light transport within virtual scenes. Introduced are, from left to right,* (a) *the false-color rendering,* (b) *spherical plot,* (c) *light path inspection,* (d) *volumetric inspection, and* (e) *particle flow tools. These images also represent the scenes that we used for a user study that we conducted to evaluate all tools.*

In this chapter, we investigate interactive light transport visualization for virtual scenes. Our approach is based on a global illumination (GI) computation using extended photon mapping, where every photon carries additional information about its light path. We present interactive techniques for examining the whys and wherefores of light phenomena by a set of *light inspection tools*. Two are based on familiar visualizations in the context of lighting: false-color renderings for depicting on-surface information, and spherical plots for directional radiance distributions. The other three target the visualization of light transport in free space and adapt visual representations from other visualization domains, namely arrows for depicting complex light transport paths, flowing particles, and volume renderings. Figure 8.1 presents all tools.

All tools support the idea of *selective* inspection: they can be set up to inspect only direct, indirect, reflected, or refracted light. This filtering allows us to focus on lighting phenomena of particular interest, and is helpful for tackling complex lighting situations difficult to grasp in their entirety.

We conducted a formal user study, where twenty subjects evaluated all light inspection tools in various application scenarios. We hence discuss their individual strengths and weaknesses. Moreover, we collected feedback from architects and professional modeling artists, how our tools could be used in their respective fields of work.

8.2 Previous Work

In lighting design, artists usually rely on their experience, and try out renderings with different light settings, possibly separating different lighting components such as direct and indirect light, or diffuse and glossy reflections [*Obert et al.* 2008]. In the research community, different depictions have been used for analyzing light transport, e.g., intensity plots for frequency analysis [*Durand et al.* 2005], light transport matrices [*Bai* 2010], or light field parametrizations [*Gortler et al.* 1996].

Many depictions allow only experts to analyze the light transport, and are not suited for artistic lighting design. Common techniques visualize only local interactions, e.g., surface irradiance using false-color renderings, or directional distributions of reflected light at surface points using spherical plots. It is also not straightforward to apply existing visualization techniques from other domains, e.g., flow visualization [*Hauser et al.* 2003]: in contrast to

Figure 8.2: *A frame from the 3D animation movie* DER BESUCH. *We will inspect the shading and illumination of this scene to demonstrate some of our light transport inspection tools.* **Acknowledgements:** DER BESUCH *is Conrad Tambour's graduation film at the Filmakademie Baden-Württemberg and we would like to thank Conrad a lot for providing us one of his scenes.*

fluids or gases, light flows through space in every possible direction at the same time, and is scattered and reflected virtually everywhere. In this chapter, we apply existing and newly adapted visualization techniques for visualizing the flow of light in virtual scenes.

8.3 Extended Photon Mapping Framework

We implemented photon mapping and all light inspection tools as a plug-in for Autodesk Maya 2012, a professional 3D computer graphics software widely used in the industry. This naturally provides a user interface which is familiar to modelers and artists. Autodesk Maya has also been used to create the 3D animation movie DER BESUCH (Figure 8.2). We will use a scene out of this movie to demonstrate our tools on several occasions.

We opted for photon mapping [*Jensen* 2012] to compute global illumination (GI), which is often used together with final gathering [*Christensen et al.* 2006]. Our plug-in, however, uses stochastic progressive photon mapping (SPPM) [*Hachisuka and Jensen* 2009; *Hachisuka and Jensen* 2010], accelerated with spatial hashing and multi-GPU support using the improved technique of Knaus and Zwicker [2011]. This enables interactive rendering and allows to acquire important information about the light transport for our tools.

SPPM is particularly well suited for interactive previews, but less favorable for high-quality rendering. Note that we build upon the natural direction of light propagation for visualization, i.e., any GI method constructing light paths starting at light sources can be used with our tools.

We decided in favor of photon mapping because of the following advantages: it naturally and easily provides all information we are interested in; it supports various materials robustly, allowing for specular and translucent materials; also, progressive photon mapping is capable of rendering previews interactively, providing instant visual feedback. For the future, we can add support for participating media; we consider light interaction with only surfaces for now.

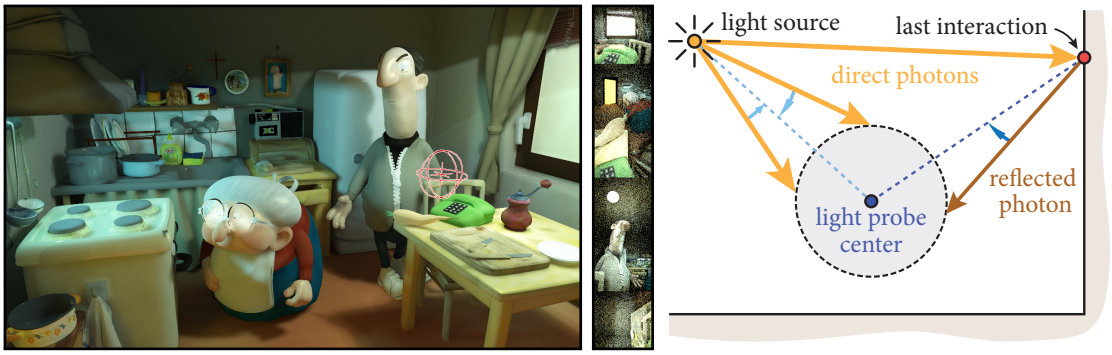


Figure 8.3: *A light probe placed above the table (red wireframe sphere) and its gathered photons (approximately 8 million for illustration) shown as an unfolded cube map.* Illustration: *we collect photons which pass through a light probe. The incoming direction snaps to the center.*

8.3.1 Enriched Photons

In photon mapping, photons are emitted from light sources, possibly reflected at surfaces or refracted. If a photon is diffusely reflected or absorbed, it is stored in a photon map. We augment our photons to carry additional information we will later require for lighting inspection. That is, we assign all light sources and objects in the scene an individual ID and store *enriched photons* with the following additional attributes:

- the ID of the light source which emitted the photon,
- the IDs of the last three objects the photon interacted with,
- the last three interaction locations along the photon's path,
- the last three fluxes (as RGB triplets) the photon carried,
- and all material interaction types so far, which can be diffuse or specular reflections, refractions, or none for direct light.

We store only three interactions which proved to be sufficient for all our test scenes. Light transport typically becomes less directed after multiple interactions, which resulted in cluttered and less expressive visualizations. (Also see Figure 8.7 on how to avoid clutter.)

8.3.2 Gathering Information with Light Probes

We now detail how we gather light transport information in a certain region of interest (ROI).

Collecting photons with a light probe

The *plenoptic function* [*Adelson and Bergen* 1991], see Section 7.7 in the previous chapter, is a seven-dimensional function $P(x, y, z, \theta, \phi, \lambda, t)$ that describes the radiant energy distribution in a scene at location (x, y, z), for direction (θ, ϕ), wavelength λ, and time t. For the inspection tools, we do not evaluate this function at a single position for two reasons: first, we want to let the user specify the ROI for the inspection; second, the probability that one emitted photon travels *exactly* through (x, y, z) is negligible. Instead, we gather enriched photons that pass

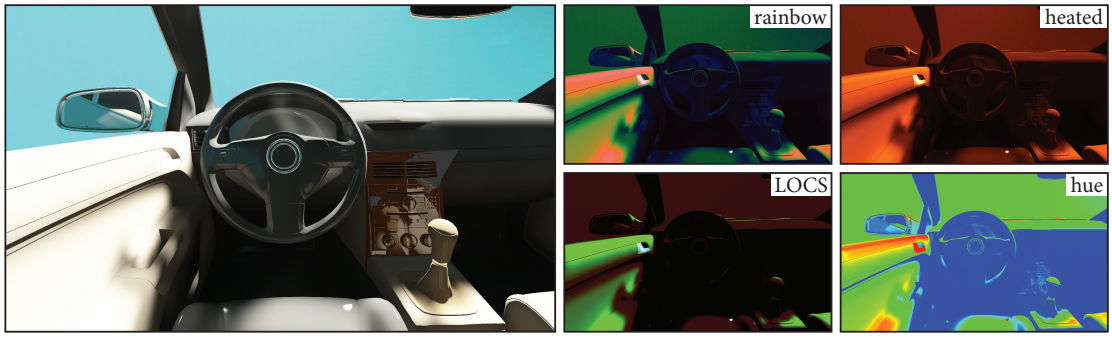


Figure 8.4: *With false-color renderings we can better distinguish the true brightness of a surface.*

through *light probes*, certain regions in space or virtual disk-shaped surfaces whose sizes are set by the user. We distinguish between front and back sides when gathering on a disk: when placed above a surface, this enables us to distinguish incident from reflected (exitant) light. With enriched photons, we can select light transport components that are not distinguishable with the plenoptic function and enable the user to specify which photons to store, i.e., selecting either all, directly emitted, diffusely reflected, specularly reflected, or refracted photons.

Light probe data

For spherical plots, light path, and volumetric inspection tools, we store the photons collected by light probes in cube environment maps (Figure 8.3). We do not store a photon's direction directly; instead, we use the direction from its last interaction point to the probe center. This ensures that all photons are mapped to the same incident direction (Figure 8.3 to the right). To collect information for a light probe, we typically emit 300,000 photons in total. This number is user-specified and can be increased for larger scenes. We continue shooting for progressive photon mapping. (Note that our tools would work with standard photon mapping as well.) We accumulate the total incident flux in a light probe environment map, but keep all information only from the enriched photon which arrived last for each discrete direction. This eviction strategy (storing only the last photon) implicates that the probability of storing a photon from a specific light transport path (e.g., a caustic) is proportional to its relative contribution to the probe, as the order of incident photons is random.

8.4 Light Inspection Tools

This section details the set of light inspection tools featured in the chapter. We follow the steps of a classical visualization pipeline for introducing each tool: we explain how data is acquired, processed, and finally mapped to geometry. The user can select between different types of inspected illumination: only direct light (photons without material interaction so far); only indirect light (photons with at least one material interaction); only diffuse reflections; only specular reflections; only caustics (photons refracted or specularly reflected at their last material interaction); or simply all of the above.

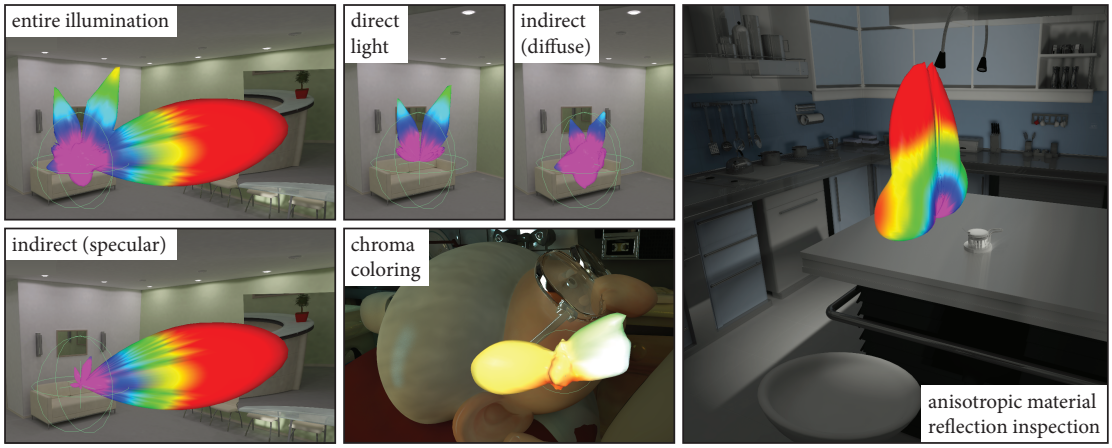


Figure 8.5: Left and Center: *spherical plots locally inspecting the entire illumination, direct light, diffuse indirect light, and caustics. Chroma coloring tells apart different light sources.* Right: *a visualization of the surface BRDF, here for anisotropic material reflection inspection.*

8.4.1 False-color Renderings

The first inspection method is false-color rendering of the surfaces' brightness. Our motivation to include this tool is its simplicity, and the fact that engineering software often provides similar depictions of the surface irradiance or radiant exitance (e.g., Autodesk Ecotect Analysis).

To capture view-dependent effects, we map outgoing radiance (either equally weighted or according to perceived luminance) to a scalar value for each pixel on the image plane. It is then used to index color maps (Figures 8.4 and 8.1a). We let the user choose between rainbow, heated body, linear optimized color space (optimized for a linear-perceptible difference), and hue maps. See Silva et al. [2011] for a survey of different color maps. Note that hue maps (or "rainbow" color maps) are usually considered as less suited [*Borland and Taylor* 2007].

8.4.2 Spherical Plot Tool

Spherical plots are often used for illustrations in computer graphics. We use them to depict directional aspects of light, captured using a light probe. More specifically, we visualize the radiance passing through a spherical region (sphere-shaped probe) or a surface (disc-shaped probe). The plot radius for a given direction is determined by the magnitude of the radiance (scaled by a user-defined parameter). We use either the chroma of the radiance directly or false-coloring as with the previous tool to color the plot's surface. To prevent flickering and to fill in missing data, we use a low-pass filtered version of the light probe as an input. Figure 8.5 shows a range of spherical plots with different phenomena selected.

To construct a spherical plot, we start with a spherical mesh tessellated around the inspector and its corresponding light probe. For each vertex, its adjacent triangles are projected into the space covered in the environment map of the light probe. Then, we average color and radiance inside this footprint. Eventually, the vertex is displaced according to the radiance, and is colored either using a false-color map or the color found in the environment map.

Figure 8.6: *The light path tool is used to inspect where the caustic is coming from, and reveals that it is a reflection of the sunlight from a mirror at the back wall.*

8.4.3 Light Path Inspection Tool

The light path tool is more involved than the previous ones. It is meant for either on-surface or free space inspection. It visualizes the key light paths that contribute to the lighting of a particular point. For example, it can be used to detect the responsible light source for a strong caustic (Figure 8.6). Given only a final rendered image of a complex scene, it can be hard for a user who is not familiar with the scene to reconstruct where lighting phenomena originated from. Typical approaches are turning different lights off and on, moving them around, and inspecting light linking (when lights are linked to affect only certain objects); this can be a tedious and time-consuming process.

This inspection tool provides this information and displays it using several arrows. We also have to reduce the presented information in a meaningful way, as a raw cluttered view rather confuses. Again, we use a light probe to collect enriched photons, letting the user specify its size, and for disc-shaped probes also the orientation. The next step is to cluster the collected photons, so that we can then create arrows to answer questions such as “where is this indirect light coming from?”. We have identified two meaningful clustering strategies.

Clustering for light paths

This strategy clusters photons by their last interactions. The clustering can be best explained using the light path notation [*Heckbert* 1990]. For example, let us assume that we collected photons with the paths $LDDD$, $LDSS$, and $LDSD$. Since we first cluster by the last interaction, we obtain two clusters: (1) diffuse interactions $LDDD$ and $LDSD$, and (2) specular interactions $LDSS$. For the visualization, we create one arrow per cluster, whose tip points at the light probe. Its origin is determined as follows: first, we compute the centroid of every cluster as the sum of its photons’ positions weighted by their respective flux. To ensure that an arrow starts on a surface, we snap this starting position to the nearest photon hit from the cluster.

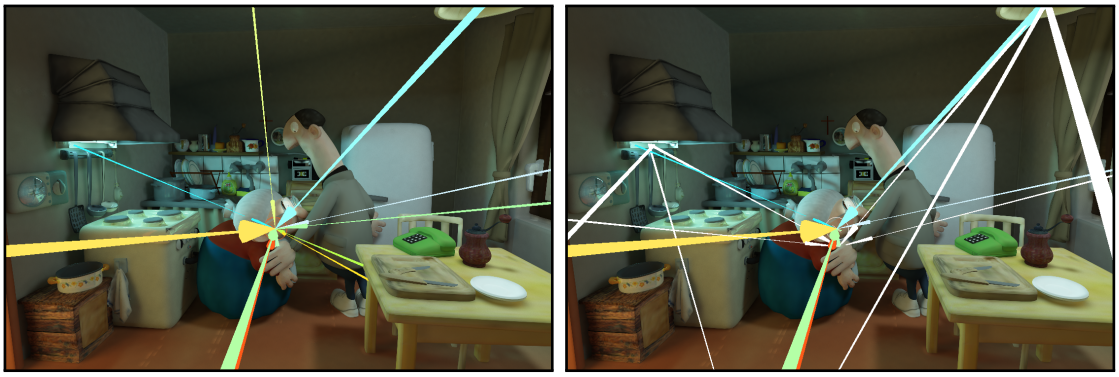


Figure 8.7: *The light path tool is prone to create visual clutter. This phenomenon amplifies for multiple light path bounces. This is why we prune paths according to a cut-off threshold* ϵ. Left: *single-segmented paths with* $\epsilon = 0.1$. Right: *one additional bounce with* $\epsilon = 0.16$.

To construct the full light path, we apply the clustering algorithm recursively. In our example, we would partition cluster (1) according to the second last interactions, which yields two subclusters *LDDD* and *LDSD*. Similar to the first step, we compute the cluster centroids and create arrows (which now end at the origin of previous ones). In contrast to the original light path notation, we explicitly distinguish between specular reflections and refractions. For glossy materials, Russian roulette decides during photon mapping if a path (and thus an enriched photon) is specular or diffuse.

Clustering for object IDs

Another powerful criteria for clustering is to use object IDs obtained from the modeling process. For example, such an object ID could be “lampshade”, “table”, or “fridge”. These IDs carry semantic information for manually modeled scenes and thus typically produce very intuitive visualizations. The (recursive) clustering works exactly the same way as before, the only difference being that the interaction type is replaced by the object ID. We can still use the selection mechanism and collect only photons according to a specific *last interaction* type.

Avoiding clutter

We obtain a large number of arrows after clustering, especially when clustering object IDs. As we want to visualize only a certain number of important light paths, we have to discard insignificant ones. The user can specify a cut-off threshold ϵ: if a light path’s relative contribution to the probe’s total collected photons is less than ϵ, it is discarded; e.g., $\epsilon = 0.1$ ensures not to include paths with a contribution of less than 10% of the total collected flux.

Eventually, a light path is visualized by a sequence of arrows created as described above. An arrow is colored according to the chroma of the flux a light path segment carries; likewise, its thickness is set to reflect the overall contribution of the path properly. Figure 8.6 demonstrates the usefulness of the selective aspect of this inspection tool, and Figure 8.7 shows an example in a scene with a complex lighting situation.

Figure 8.8: *Volumetric inspection sensitive to only specular caustics* (left), *and to all lighting components* (right). *Note how selective inspection emphasizes lighting phenomena.*

8.4.4 Volumetric Inspection Tool

A matter of particular interest is *how much light flows* within an entire region of space. To investigate this, the user can freely position a *volumetric inspector* which is a virtual box filled with single scattering homogeneous participating media. This allows for analysis of caustics and shadows in free space. To capture these phenomena, we simply rasterize the paths of photons inside the volume and accumulate their contributions to voxels of a 3D texture.

The volume data is stored in a uniform grid with a resolution of N^3, where N is between 32 and 256, and specified by the user. If a photon intersects the bounding box of the volume, we perform the 3D rasterization of a line using a 3D Digital Differential Analyzer [*Amanatides and Woo* 1987], i.e., the ray thickness as well as the sampling footprint is one voxel. We accumulate out-scattered light of the ray's radiance in all traversed cells of the grid. For the sake of visualization, we ignore the attenuation of the ray in the media. The scattering is isotropic and the user can specify the density of the participating media in order to amplify lighting phenomena. Again, photon selection rules can be applied to inspect individual phenomena.

During rendering, the volume is ray marched and blended over the scene for visualization. As the volume is updated progressively, the blending factor is inversely proportional to the number of contributing photons. Figure 8.8 gives an example how volume caustics can be detected in free space. The selective aspect ("only specular caustics") is pivotal in this case.

8.4.5 Particle Flow Tool

The functionality of the particle flow tool (see Figures 8.1e and 8.9) is inspired by a common approach to analyze flow by injecting particles or dye into media. It also reminds of a wind tunnel. Our tool randomly spawns particles within a user-specified spherical emitter. The user also specifies whether the particles travel along the propagation direction of light, or if they are attracted by prominent sources of illumination.

For every time step, we compute one small light probe for every particle at its current position in space; again, all selection mechanisms as described previously can be applied. We compute the sum over all photons' directions weighted by their respective flux and use this value to determine the speed and direction for an individual particle. Inspired by flow

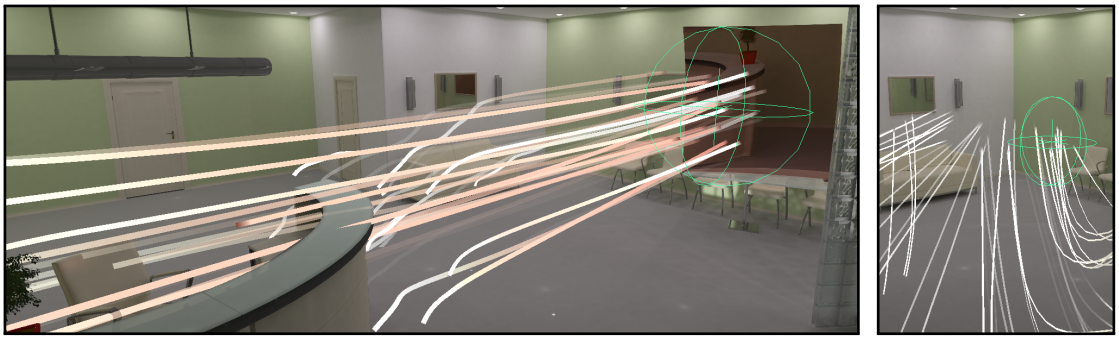


Figure 8.9: *The particle flow tool is inspired by injecting particles or dye into flowing media.* Left: *particles react to only caustics, and follow light reflected at a mirror.* Right: *no selective criterion is specified and the entire illumination is taken into account. This leads to a rather undirected flow and emphasized how selective inspection is especially important for this tool.*

visualization, we now update a particle's position using this vector with a simple Newton integration scheme. Thus, particles roughly follow the most dominant flow of light. As a consequence, particles do not move along straight lines as photons do: they tend to bend at intersections of light beams or flow smoothly around obstacles (Figure 8.9). If they are set to be attracted by the light sources, then the direction is simply mirrored. Particles are rendered as shaded stripes with trails which are cubic Bézier splines. The user can furthermore adjust the spawn rate and global factors to adjust the particles' speed and trail length.

8.5 Implementation and Performance

Our inspection tools are directly integrated into Autodesk Maya 2012 as a plug-in; stochastic progressive photon mapping [*Hachisuka and Jensen* 2010] is implemented using OpenCL and optionally supports multi-GPU usage. All reported timings in this chapter have been measured on a single-GPU system, described in the next section, used for the user study.

For spherical plots and light path inspection, we emit 300,000 photons in total, collect them with light probes, and store them in a cube map with a resolution of $128^2 \times 6$. The performance is evidently dependent on the scene complexity. For the scene in Figure 8.3 (consisting of 1.5 million triangles), it takes approximately two seconds, while we rebalance SPPM more toward photon shooting during this process.

To accelerate photon shooting, we use a kd-tree, which is built once and updated asynchronously in a separate thread when a scene is modified. The full rebuild takes about four seconds for the scene shown in Figure 8.3. The clustering of photons for the light path inspection always takes less than a second. For the particle tool we emit approximately 16,000 photons per frame to ensure interactive rendering; the cube map for every particle has a size of $32^2 \times 6$. The volumetric inspection tool is updated with 16,000 emitted photons per frame.

8.6 User Study

We conducted a formal user study to evaluate the usefulness of all light transport inspection tools for various application tasks. To this end, we constructed five different scenarios with corresponding Maya scenes and tasks to be accomplished.

8.6.1 Overview

Twenty participants took part in the user study and were asked to solve all tasks by operating Maya themselves and using our plug-in. We set up four identical personal computers equipped with Intel Core i7-2600 processors and Nvidia GTX 580 graphics cards. Each computer was connected to two Samsung SyncMaster 2443 monitors at their native resolution of 1920×1200, with a viewing distance of approximately 35 inches. All environments had roughly the same lighting conditions without distracting incidence. All tasks were recorded for further analysis.

Every subject was introduced to the inspection tools in a training scene consisting of a directional light source, a mirror, and two diffuse surfaces. Supervisors answered upcoming questions until every subject felt comfortable in using all tools and (basic) handling of Maya. Every subject received a questionnaire. First, they were to rate their computer graphics knowledge and experience in modeling applications, before starting with the actual tasks.

8.6.2 Tasks

While constructing the tasks, we had particular tools in mind which will probably suit best. Moreover, we intentionally included cases where certain tools should be utterly inappropriate. Of course, our hidden agenda was not revealed to the subjects. The order of tasks was shuffled for every subject to scatter biasing effects, such as mental fatigue or gaining experience in using the tools. During each task, subjects were free to choose their own workflow by using each inspection tool in a sequence of their own choice; likewise, there were no time limits.

Task 1. Subjects were provided with a scene of a car interior, focusing on the dashboard. (It is the scene we saw in Figure 8.4.) They were asked to find possible sources of discomfort glares caused by reflections of metallic parts within the interior.

Task 2. We provided a scene of a piano room. (It is the scene from Figure 8.6.) Subjects were asked to find an appropriate region to place a reading desk, with a good atmosphere where the illumination is as ambient and smooth as possible.

Task 3. We provided a scene from DER BESUCH (Figure 8.2). We asked how the surface color at the granny's cheek is composed (see Figures 8.1c, 8.5 chroma coloring, and 8.7). Subjects were to find the contributing sources of direct and indirect illumination.

Task 4. The scene contains a lobby of an office building with a reception desk (Figures 8.8 and 8.9). An interior designer placed a metallic sculpture as a decoration there, which causes unwanted reflections that disturb people in front of and working behind the reception desk.

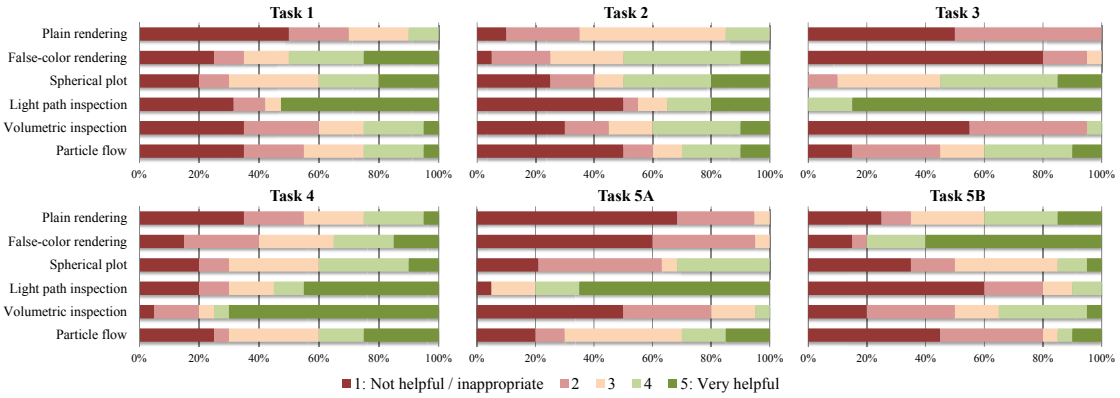


Figure 8.10: *These graphs summarize the results of our user study. For every task (see Section 8.6 for an overview) we report the subjects' ratings of all light transport inspection tools.*

Subjects were asked to make themselves clear how and why these reflections occur, in order to relocate the sculpture afterward, making sure to avoid unwanted reflections.

Task 5. This task consists of two subtasks. Subjects were provided with a scene of a kitchen. We can see this scene in Figure 8.1e and in Figure 8.5 to the right. Several surfaces are made of brushed metal which causes highly anisotropic reflections. In Subtask A, they were asked to find the light source causing a particular reflection on the ceiling. In Subtask B, they were asked to rotate the brushing orientation of the brushed metal on the kitchen table until the lighting is as homogeneously distributed as possible.

8.6.3 Analysis and Evaluation

All subjects rated their computer graphics knowledge fairly high (mean: 4.25, standard deviation: 0.7) on a scale from 1 (no prior knowledge) to 5 (expert level). This is a consequence of acquiring mostly undergraduate and graduate students and researchers with computer graphics knowledge for the study. Every subject attended at least an introductory graphics lecture. Modeling experience was rated rather average (mean: 2.65, standard deviation: 0.8), but the majority used various modeling applications before; the scale ranged from 1 (no experience) to 5 (work as a professional).

All subjects stated that they were able to solve Tasks 3 and 4. For Tasks 1, 2, and 5A, around a quarter admitted that they were only partly confident in their results. Slightly more than half of the subjects stated that they were fully able to solve Task 5B, but apparently we formulated the goal for this task too strict: when looking at the subjects' results, we were quite satisfied. Six subjects completed all tasks within a timeframe of 1 hour to 1:30 hours. Three subjects took up more than 3 hours (up to 3:20 hours) to solve all tasks. The remaining subjects were roughly in the 2 to 2:30 hours range. Timings do not include the supervised preparatory training.

Assessment of light transport inspection tools

Most interesting is how subjects rated the individual inspection tools. Figure 8.10 shows how the tools performed in every task. Using no visualizations (plain rendering) is included for comparison as well, to expose how much insight into the scene was already given.

Plain renderings. It turns out that plain renderings already allow subjects to *get familiar* with many lighting situations quickly. Many subjects distributed lighting in Task 5B by simply tracking changes in the rendering. They were also mostly confident in directly finding nicely illuminated areas for reading in Task 2. Complex lighting situations, by contrast, are difficult to grasp: the anisotropic reflections in Task 5A were misleading, and the blending of numerous different light sources in Task 3 made backtracking very difficult.

False-color renderings. They are able to *emphasize the brightness of surfaces*. Using color maps in Task 1, several subjects already felt confident in having solved the task. Task 3, which exhibits the most professional lighting design, is a clear counterexample: not having real color information anymore makes it impossible to distinguish between different colored light sources. Similarly, this tool is of no help in Task 5A due to the confusing anisotropy.

Spherical plots. They are particularly helpful to grasp *directional aspects locally*. Shading the plot's surface with real chroma turned out to be especially useful for Task 3. The tool's locality is both its strength and its weakness: it is not that convenient for inspecting larger regions or causal relationships.

Light path inspection. This tool worked best in Task 3, which we consider a prime example: all subjects were conveniently able to solve this task. It was also able to help finding the light source causing the particular reflection in Task 5A properly. On the other hand, the tool fails to depict global lighting situations as in Task 5B. The notable break in the judgement of this tool for Task 1 is interesting. An analysis of the recordings revealed that some subjects placed this tool on the dashboard or door handle cup and received impractical results, and subsequently rated the tool as inappropriate. The remaining half of the subjects placed the tool at the head and neck support, and oriented its normal to match the driver's viewing direction. They were then able to state that the tool provided significant new insights.

Volumetric inspection. This tool performed strong in Task 4, where subjects used it to investigate caustics in free space with a large spatial extent. A counterexample is Task 3, where the lighting is too soft (which is a design goal in many artistic scenes). Some subjects stated that they appreciated an additional benefit of the tool: it has to be placed only once without further need to move it around during inspection.

Particle flow. Remarkable is the invariably large standard deviation of the particle flow tool for all tasks (1.3, 1.5, 1.3, 1.5, 1.3, 1.3). Our first hypothesis was the presence of a correlation between the subjects' experience in graphics or modeling and the rating of the tool, but the

analysis of the questionnaires indicated only a vague correlation (and no further correlations). The recordings, however, revealed the reason for the ambivalent results: some subjects did not make proper use of the selection mechanism and were thus simply not able to solve the tasks. Interestingly, we received feedback from subjects using it properly that selective filtering is particularly important for this tool. As expected, it was still not rated helpful in all tasks, especially not in Task 5B.

Summary. Our user study shows that different inspection tools are required and best suited for different application scenarios and tasks. As expected, false-color rendering was only helpful in Tasks 1 and 5B where the strength of illumination was examined. The spherical plot tool performed well when a local directional aspect was examined, but performed poor for all tasks requiring "the big picture". Volumetric inspection is well suited to comprehend caustics. As discussed before, particle flow was only found to be useful when used selectively. Light path inspection worked well in many cases (Tasks 1, 3, 4, 5A) and is definitely worth further investigation. All subjects agreed that interactivity is crucial for our inspection tools: on a scale from 1 (not really necessary) to 5 (crucial; key aspect) one half each opted for 4 and 5.

User study feedback

The subjects were further able to provide open comments and gave us valuable feedback on how the light inspection tools could be improved. We received manifold hints for fine-tuning the user interface; additionally, we also got valuable high-level suggestions. In the artistic scene of Task 3, for instance, a key light path spawns at the lamp above the kitchen table. When selecting only indirect light for the inspection, this path seemingly remains: this is due to strong indirect light from the lampshade. While our solution is technically valid, the clustering—obviously lacking semantic information—does not treat the light bulb and the lampshade as a common source of direct light as humans do. Two subjects suggested to fade out the photon tracers in the volumetric tool to assess the light distribution better. Lastly, several subjects stated that the particle tool would be more useful if the trails would follow the incoming light in case of hitting a surface, and, e.g., reflections could be identified; this can be achieved by rejection-sampling the light probe.

8.7 Domain Expert Feedback

Architects

We consulted two professional architects independently and asked them about possible fields of use in their area. Both explained that, above all, both natural and artificial light sources form the illumination of a building and its interior rooms, and are addressed individually. Both architects stated that they simply trust their feelings for routine tasks in creating ground plans and follow rules of thumb, such as bearing in mind that windows facing north allow the least sunshine to pass through (in the northern hemisphere).

If requested by building contractors or demanding customers, however, architects create models which account for the solar altitude over the seasons and the sun path. They believe

light inspection tools can be of help while planning the lighting of complex buildings, such as museums. They can further imagine that these tools could be used to create convincing illustrations for demanding customers to reveal them how light propagates through buildings, i.e., applying visualizations for marketing purposes. This idea extends to interior design to show customers directly the effects of changing interior materials like wallpapers or floorings.

Artists and modeling professionals

Furthermore, we asked a group of professional modeling artists for their feedback after providing them with our light inspection tools. If a professional artist works for months on a single scene, it is guaranteed that he possesses a full comprehension of all lighting phenomena and our light inspection tools might not be capable of providing additional information. However, this changes rapidly if another artist is confronted with this scene for the first time, which is not uncommon in today's large production teams: then, it is a long and tedious task to figure out how the lighting is set up and the light inspection tools can be of great help. The volumetric inspection tool was said to be helpful to test the smoothness of the lighting in a scene; an application that we did not think of.

8.8 Discussion and Recent Advances

In this chapter we presented novel light inspection tools and their associated interactive visualization techniques. They help users to get a better understanding of the light transport in virtual scenes. We conducted a user study and concluded that individual techniques are useful and appropriate for different tasks that we categorized. We collected feedback from domain experts and discussed applications from decision-making aids for architects and lighting designers to purely artistic tasks. The feedback of the user study highlighted possible improvements, such as inspecting shadows by including the tracing of shadow photons.

We consider our work as a first step in the interactive visualization and analysis of light transport in virtual scenes. More visualization techniques have recently been adapted. The work of Zirr et al. [2015] shows how to visualize coherent structures of light transport. This now provides a global insight into a scene using a dense visualization technique. Another approach transfers *information visualization* techniques to provide insight into physically based rendering settings [*Simons et al.* 2016]. A specific use case is addressed in [*Spencer et al.* 2015], which targets the development and optimization of photon map denoising.

Our framework can serve as a basis for selective *artistic light transport manipulation.* In the next chapter, we will briefly present an approach which combines the selection and visualization of light paths with new manipulation tools. We believe that you can modify best what you can see, grasp, and understand. By selecting only specific paths, manipulators can be adjusted to affect only certain lighting phenomena, such as caustics.

9

Interactive Light Transport Manipulation Techniques

In this chapter we will first state a motivation why it is highly desirable to *manipulate* the illumination of a virtual scene. Then we provide an overview of existing concepts and associated manipulation tools. The main section gives a brief summary how to combine the visualization of light transport with the selection and manipulation of specific areas and parts of a scene.

9.1 Introduction

We have first addressed the issue of *controllability* in the context of procedural content generation in Chapter 2. While procedural techniques are eminently suitable to automatically generate a vast amount of rich detail, it proves difficult to alter an algorithm to achieve a very specific adjustment in the outcome. If an artist models a certain object, he might consider the following steps: first, using a procedural description to generate a complex base shape. Next, exporting such a shape as a discrete model to a file, and then loading this file into a 3D modeling tool. This enables the artist to finetune the object and make arbitrary alterations. He might even once again apply other procedural algorithms to alter the surface and give the object a new finish.

Creating objects and arranging them to form an appealing scene is only the first step in generating a realistic image or animation. What we have to do next is assigning proper materials to every object and set up an attractive lighting scenario. We can then position the camera and apply a global illumination method to render the scene.

A 3D computer graphics software provides a much better and more intuitive controllability of a scene than a procedural description does. An interactive software offers a main modeling area in a "what you see is what you get" fashion, albeit in lower quality. This, together with fast preview renderings, allows for swift modeling iterations.

If the lighting of a scene becomes more and more complex, the artist might encounter difficulties in understanding certain lighting phenomena in his renderings. This is why we have introduced selective light transport inspection techniques in the previous chapter. These tools support the analysis of the root cause of such lighting phenomena.

Alas, it also becomes more difficult to control the overall lighting and appearance of the scene. If the artist is already in the stage of finetuning, he might want to alter very specific and

isolated parts. In a global illumination environment, however, isolated parts are inherently difficult to modify, as there is a mutual dependency between all objects and materials. Imagine we want to alter a shadow or a caustic. Repositioning the associated object or contributing light sources may have unwanted ramifications in other parts of the scene. Perhaps the object and light sources are already in a perfect position and should not be touched at all?

One approach would be to load a finished rendering into an image processing program. The artist could then carefully work on the post-processing in the image space. This requires practice and skills to produce a flawless result. Moreover, the workload rises dramatically for an animation unless this process can be automated for individual frames. A key problem remains: if the scene has to be altered and rerendering is required, the manual post-processing has to be redone.

This chapter covers another approach to modify lighting phenomena. We will present various *light transport manipulation* techniques to tackle this problem. They allow to manipulate isolated phenomena directly in a virtual scene. A global illumination method will then render a final image where the manipulated light transport is seamlessly factored in.

9.2 Overview of Light Transport Manipulation Tools

Specialized visualization and interaction techniques have been developed in the context of artistic light manipulation. Kerr and Pellacini [2009] identified three different approaches: direct control of lighting parameters; indirect control via feature manipulation, e.g., dragging shadows; and goal-based optimization where the user paints or sketches the desired lighting and illumination.

The simplest methods do not require inspection tools, as they modify intuitive parameters, or are easy to depict since the phenomena are very isolated (direct light or reflection in these cases): Barzel [1997] describes an artist-controllable and non-physical lighting model, taken further by Kerr et al. [2010] who describe BendyLights, an artistic lighting model where light travels along splines. Supporting visualizations make their user interface more convenient. Related work enables the editing of mirror-like reflections [*Ritschel et al.* 2009] and on-surface signal deformation [*Ritschel et al.* 2010]. They use custom-tailored interfaces.

Indirect light manipulation is tempting, as artists can intuitively work on the desired effect. Pellacini et al. [2002] describe a user interface for designing shadows, which are treated like primary modeling primitives allowing movement, scaling, and rotation on the scene's surfaces. Okabe [2007] presented an appearance-based user interface to design image-based lighting environments by solving the inverse shading problem, and Pellacini [2010] presents a sketch-based interface for editing natural illumination. Obert et al. [2010] carry on this idea to editing shadows in all-frequency lighting environments by precomputing and storing scene visibility information separately from lighting and BRDFs.

Painting interfaces for lighting design have been investigated over the last two decades, however, Kerr and Pellacini [2009] found that this approach is less suited as users tend to sketch rather than accurately paint goal images, which in turn hinders the optimization procedure. Work in this field most often infers parameters of light sources from target images [*Poulin*

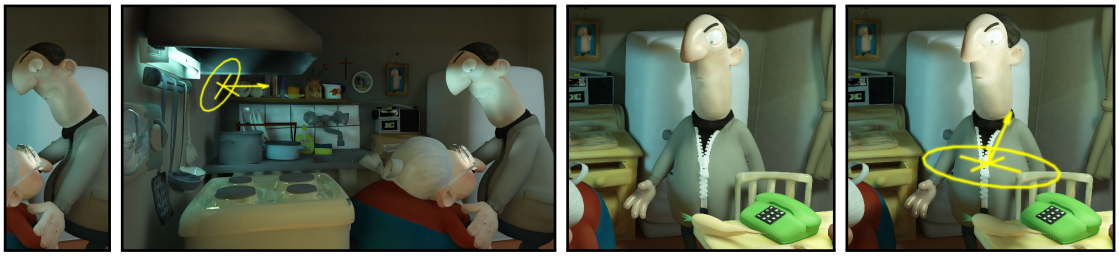


Figure 9.1: *Selective manipulation allows for basic but interesting effects, such as bending the flow of light into a new direction* (left)*, or reflecting light on an imaginary disc* (right).

and Fournier 1992; *Schoeneman et al.* 1993; *Pellacini et al.* 2007], or adjusts surface reflection [*Kawai et al.* 1993; *Obert et al.* 2008].

We refer to Schmidt et al. [2014] for a thorough state-of-the-art report. It covers a broad range of techniques to edit appearance, lighting, and material in an artistic way. This work also derives a more rigorous segmentation of all methods and provides a structure to this growing area of research.

9.3 Selective Manipulation with Enriched Photons

Let us revisit our extended photon mapping framework (Section 8.3) from the previous chapter. There, we have augmented photons with additional information about their interactions with the scene (Subsection 8.3.1). These enriched photons enable a *selective* inspection and visualization of light transport. If required, our inspection tools can be set to inspect only direct or indirect light, or diffuse or specular reflections.

In this spirit, we can transfer the selective aspect to the manipulation of light transport. Such a filtering approach then enables us to target only certain phenomena for manipulation, such as diffuse indirect light or caustics. The tracing of shadow photons [*Jensen* 2012] would also allow to target shadows. Including object IDs further allows to target a narrower set of light paths with formulations such as “target only direct light from this light source” or “target only indirect light reflected from this lampshade”.

Figure 9.1 shows two examples of selective manipulation. The user can freely place a disc inside the scene. Such a disc can then either bend or reflect incoming light according to a user-specified vector. Again, the manipulator is integrated as an element of the user interface of Autodesk Maya. This enables artists to locate and orient such a manipulator in a familiar way. The state of the manipulator also interpolates between key frames in animated scenes.

The manipulator offers control over the magnitude of its influence. This can be specified by a parameter in the $[0,1]$ range. For 0, the manipulation has no effect, i.e., an incoming photon keeps its direction. For 1, the direction of the photon is set to the user-specified vector for bending or reflecting. For in-between values, these directions are interpolated.

When a photon reaches the manipulator’s region of influence, the manipulator can either deflect it immediately or emit a second photon, which is deflected instead while leaving the

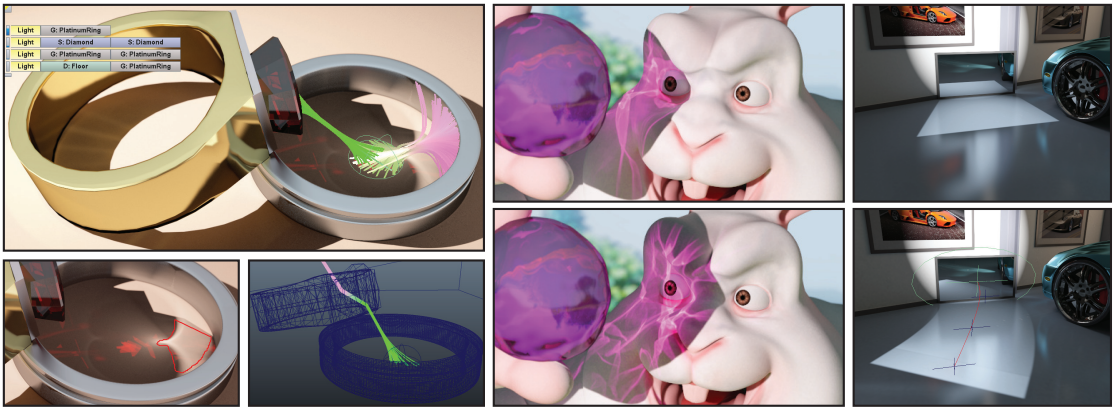


Figure 9.2: *Example images from Schmidt et al. [2013].* Left: *stroking on the image plane allows for a very intuitive and convenient selection of lighting phenomena.* Center: *the tools seamlessly integrate into production environments and allow the editing (here: amplification) of lighting phenomena.* Right: *a "BendyCaustics" effect by targeting a specific set of reflected light paths.*

original photon unaffected. In the first case, all of the scene's energy is preserved; in the second case, more energy is added. At the same time we allow for light amplification or reduction.

We can smooth the boundaries of affected regions easily. Whether a photon is affected by the manipulator is determined by a probability function. This function returns the highest values at the center of a manipulator and declines while moving toward the boundary. Parametrizing this probability function then allows for both sharp cutoffs and smooth gradients.

In the next section we discuss a more general approach. It allows a more specific and fine-grained control over which set of light paths should be the target for artistic light transport manipulation. This means that "light path selection" will be introduced as a third component to our interplay of "light path visualization" and "light path manipulation".

9.4 Path-Space Manipulation of Physically Based Light Transport

Schmidt et al. [2013] introduces a novel set of light transport manipulation techniques that operates on path space. The user can operate either *directly* or *indirectly* on paths. Using the direct approach the user selects and transforms grouped paths. Alternatively, the user can transform proxy objects in a scene: the path space is then adjusted indirectly according to these edits. The path-space approach allows to edit more complex shading effects. This sets these techniques apart from previous methods which target only specific phenomena.

This work is interesting in the context of this thesis as it is an extension to the light transport inspection tools discussed in the previous chapter. They now allow a more complex path-space visualization and analysis. Moreover, the visualized quantities can now be manipulated.

Figure 9.2 shows some example images. In one scene the user wants to target a specific caustic. By stroking around the caustic directly on the image plane, the system automatically provides a list of possible light paths which contribute to this area. In another scene, the

ramifications of such a caustic are amplified. In the scene to the right the selective aspect is used to create a "BendyCaustics" effect (in the spirit of BendyLights [*Kerr et al.* 2010]).

9.5 Discussion

We had a very pragmatic approach to lighting in the first part of this thesis. At that time we were primarily focusing on singular shapes and objects. In the first instance we required a light source to enable the mere *display* of an object. While shadows look nice and add realism to a scene, it was more important for us that they provide important visual cues. They enable a better interpretation of the position and orientation of an object within a scene.

In an analogous manner, more sophisticated shading methods enhance the visual quality of a rendering, but more importantly, this helped us grasp the ramifications of our complex surface deformations. For instance, the approximation of ambient occlusion emphasizes surface wrinkles and interrelations between objects. Technical drawings and illustrations carry this pragmatic approach to extremes. Such a drawing is as functional and objective as possible and uses only clear perspectives and projections. Any distracting illumination is avoided to facilitate the interpretation of how an object is constructed.

Now, the primary purpose of *cinematic lighting* could not be more different: it is *storytelling*, as Calahan [1999] points out. We are now facing completely new requirements for light sources and illumination settings. Lowell [1992] and Calahan further point out how a good lighting design leads the viewer to key story elements; enhances mood and atmosphere; creates depth; conveys the time of day and season; reveals a character's personality and situation; and finally, complements a stringent cinematic composition.

Part of this can be achieved by redirecting, relocating, strengthening, weakening, softening, and in general altering light sources. A change in tone and brightness may drastically modify an appearance. A good lighting design may even require an exchange of light sources between shots of the same scene. We already saw how finetuning selected phenomena is inherently difficult in a global illumination setting, where all components of a scene are interrelated.

Calahan [1999] argued that the development of future lighting tools *"will be driven by the desire to see on the screen what we are able to visualize in our minds".* This is why we now addressed light transport manipulation techniques which were designed in this spirit. They are especially useful for visually rich and complex virtual scenes, where it is easy to get lost in the sheer magnitude of primitives, objects, materials, and (direct and indirect) sources of light. They are primarily based on physically correct rendering methods, but allow more artistic modifications which are only possible in synthetic cinema. We further pointed out how helpful supporting visualizations are to grasp which phenomena are modified.

The discussed work also argues for a seamless integration into real production environments. By sticking to familiar environments and established user interfaces an artist can be productive right from the start. Producing plausible and appealing results in high visual quality extends to animations where manipulations affect multiple frames.

10

Understanding and Reconstructing Real-World Light Transport

The final chapter of the main part transfers our previous approaches to inspect and visualize light transport to the real world which surrounds us. We will first address the application fields which require a profound understanding of illumination and light transport. We then give an overview of previous research which deals with lighting visualization and the reconstruction of real-world scenes. In the main section we present an interactive environment which allows to reconstruct and inspect the light transport in scenes captured by a set of photos.

10.1 Introduction

The light and the illumination of our environment affect us every moment of our daily life. Our endogenous *circadian rhythm* is controlled by a *circadian clock*, which oscillates with a phase of one solar day of the earth. This rhythm aligns continuously with environmental cues and the light-dark cycle. Light is the environmental cue which has the strongest impact on our circadian rhythm [*LeGates et al.* 2014]. This is why we wake up gently at sunrise, and our alertness, strength, coordination, and reaction time peaks during daytime. At nightfall, melatonin secretion starts and we begin to feel tired.

We illuminate our surroundings accordingly. Guidelines and a comprehensive set of regulations enforce the proper lighting of office environments. Studies clearly indicate that well-lit surroundings increase productivity and well-being, whereas the lack of exposure to daylight can induce negative effects on the physical and mental health of workers [*Edwards and Torcellini* 2002; *Boyce et al.* 2003; *California Energy Commission* 2003]. Exposure to daylight has also been linked to the quality of sleep at night [*Boubekri et al.* 2014]. There is, however, a counterside to excessive broad daylight: an interior can simply be too bright to be comfortable. When the sun is low in the sky, it might also shine through windows directly into our eyes. Moreover, highly reflective materials can produce veiling reflections and disability glare.

At home, we prefer a more pleasant mellow and ambient lighting in the evening. It provides us with *visual comfort*. This is why lighting is such an important part of interior design. With the advent of cost-effective LED technology, we could also observe how car manufacturers added ambient lighting to the interior of their models.

We can use various techniques to measure the illumination in the settings we have addressed. The most practical approach is to use a *light meter*. It is a physical instrument that measures photometric quantities which we have introduced back in Section 7.3. It can be used, for instance, to examine the lighting in an office environment and to verify that it complies with regulations. More sophisticated devices are also able to measure how surfaces absorb, scatter, or reflect light. The work we present in this chapter helps users to draw inferences about an illumination setting from a set of photos.

Contributions. We present an interactive environment that allows to import a set of different photos of a scene, which also contain depth information. We then reconstruct the light transport directly in image space, as faithful as possible, to draw conclusions about the illumination inside the scene. This allows to answer questions such as "where is light coming from and how is it propagating through space?". For this reason we revisit our previous visualization tools from Chapter 8 and show how to reimplement spherical plots. We also introduce a new inspection tool, a *virtual light meter*, which allows us to measure photometric quantities.

10.2 Previous Work

Creating an illumination setting requires preceding thoughts and calculations. The computer proved to be an invaluable tool for support. It allows the prediction and validation of the outcome of an illumination setting. In this section we first review specific methods and tools which have been developed for this use case. Then we address how to reconstruct real-world scenes. We also discuss image-based rendering and the visualization of real-world light transport.

Rendering with Radiance

The *Radiance* lighting simulation and rendering system is a set of global illumination tools that dates back to 1986 and has been continuously further developed since then. These tools are capable of simulating and visualizing the lighting in developing, existing, and reconstructed scenes. They are targeted to computer graphics researchers, lighting engineers, and designers. The *Rendering with Radiance* textbook [*Larson and Shakespeare* 2004] gives a comprehensive overview of all tools and shows how they can be applied to real-world scenarios.

Lighting Engineering

Other than providing pure visual comfort, there are more serious use cases where proper illumination is a crucial factor. They include road safety at night, tunnels in particular, which require minimum levels of illumination. This also applies to buildings, basements, and parking structures. The necessity of comprehensive safe illumination extends to large structures such as airports. The field of study which deals with the accurate determination of required illumination is called *lighting engineering*. We refer to Simons and Bean [2001] for a thorough textbook about the various use cases of lighting engineering and its applied calculations.

Reconstructing real-world scenes

The reconstruction of 3D models from images, photos, videos, and other scanner data is an old and challenging problem at the interface of computer vision and computer graphics. We will now give an overview of techniques to capture geometry, lighting, materials, and more specific phenomena. For further reading, we refer to the comprehensive textbook edited by Magnor et al. [2015] which explains how to capture reality using video acquisition technology.

3D scanners. The most practical approach to capture geometry is to use 3D scanners. They are the specific device for this use case and capture the distance from a sensor to the nearest surface within their field of view. This results in a point cloud of individual geometric samples. Multiple scans from different positions are required to capture all sides of an object or even entire scenes. This results in a set of multiple point clouds which have to be *aligned* to a common reference system. The whole model or scene can then be reconstructed from the merged point clouds. See [*Curless* 2000; *Bernardini and Rushmeier* 2002] for an overview of different scanning technologies and a more in-depth explanation of the 3D scanning pipeline.

Geometry from images and videos. Professional 3D scanners are a costly equipment and scanning large areas can be a tedious process. This is why using standard cameras is an appealing idea to reconstruct geometry. Slabaugh et al. [2001] give a survey of methods for volumetric scene reconstruction from photos. The seminal work of Seitz and Dyer [1999] introduced voxel coloring for photorealistic scene reconstruction. The approach from Pollefeys and Gool [2002] automatically builds realistic 3D models from 2D images acquired with handheld cameras. In 2010, the launch of *Microsoft Kinect* sparked new research interest. It is a low-priced motion sensor which tracks both depth and color. The *KinectFusion* framework allows to acquire indoor scenes interactively [*Newcombe et al.* 2011; *Izadi et al.* 2011]. Follow-up work from Whelan et al. spatially extends the KinectFusion approach to allow capturing large, unbounded environments in real-time [2012] and further improves globally consistent surface reconstructions and lighting [2015a; 2015b; 2016].

Intrinsic images. After the seminal works of Land and McCann [1971] and Barrow and Tenenbaum [1978], *intrinsic image decomposition* became a challenging problem in computer graphics and vision. In essence, an image is separated into a shading and a reflectance layer, and possibly more intrinsic layers to extract other isolated characteristics. This allows to address questions concerning the *effects of lighting* and the *surface reflectance* independently from each other. Intrinsic images are of great value for a variety of graphics and vision problems, such as removing or displacing shadows, transferring surface textures between images, and image relighting. Weiss [2001] shows how to derive intrinsic images from image sequences, whereas the work of Tappen et al. [2005] recovers intrinsic images from a single image. For a thorough overview and comparison of more recent decomposition algorithms we refer to Bell et al. [2014]. Having access to the intrinsic layers of a captured image becomes very useful in the context of scene reconstruction to avoid capturing fixed lighting or unwanted shadows. The availability of inexpensive color and depth (RGB-D) motion sensors (see launch

of *Microsoft Kinect* above) led to more specific methods for intrinsic image decomposition from RGB-D imagery [*Barron and Malik* 2013; *Chen and Koltun* 2013].

Stereo images. Since our visual system includes two eyes, it allows us to process various complex tasks, including the assessment of depth (interpreting the distance to an object and the interrelation between objects). By analogy, using two (or more) cameras to generate different views of the same scene enables us to reconstruct a point in 3D space, i.e., we can determine its original location if we know all camera parameters and projections. In computer vision, this process is known as *triangulation*. It is a non-trivial problem in reality as images may contain various imperfections; thus, we can only estimate locations to a certain degree.

Reconstructing depth information from *stereo images* is known as *stereo image pair matching*. Pairing each point in one image with its corresponding point in the other image is known as the challenging *correspondence problem* [*Barnard and Thompson* 1980]. A natural extension to this approach is to match points in more than two input images. See Hartley and Zisserman [2003] for comprehensive material on multiple view geometry, and Lazaros et al. [2008] and Bleyer and Breiteneder [2013] for an overview of stereo vision algorithms.

Lighting. The seminal work of Debevec and Malik [1997] shows how to recover high dynamic range radiance maps from photos. The work of Nayar et al. [2006] sets up a high frequency illumination system to separate direct and indirect (global) components of a scene. O'Toole and Kutulakos [2010] analyze and reconstruct the light transport matrix using a few dozen low-dynamic range photos. Reddy et al. [2012] carry on this research and acquire light transport in frequency space, decomposed into direct, near-range, and far-range components. O'Toole et al. [2012] enable fine-grain segmentation and control over the light paths in a photo.

Materials. A faithful reproduction of a scene includes applying accurate and realistic material properties to objects. The seminal work of Ward [1992] measures and models anisotropic reflection. Marschner et al. [2000] introduced an image-based BRDF measurement. Ramamoorthi and Hanrahan [2001] presented a signal-processing framework for inverse rendering to recover BRDFs. Han and Perlin [2003] measure bidirectional texture reflectance with a Kaleidoscope. An experimental analysis and image-driven navigation of BRDF models is explained in the work of Ngan et al. [2005; 2006]. Gosh et al. [2007] focus on a fast capture of BRDFs and prevent aliasing issues by minimizing high-frequency noise. Recent work reconstructs measured BRDFs using a very limited number of samples [*Nielsen et al.* 2015]; the robust method of Aittala et al. [2015] requires only two photos, one with and the other without flash, consecutively acquired with the camera of a standard mobile phone. Xu et al. [2016] presented a related two-shot approach for near-field perspective cameras.

More specific phenomena. Matusik et al. [2002] acquire and display 3D models of objects that are impossible to scan with traditional scanners. This includes the acquisition of highly specular and fuzzy materials, such as fur and feathers. Goesele et al. [2004] focus on the acquisition of translucent objects. Hawkins et al. [2005] tackle the problem of acquiring time-varying participating media and Gu et al. [2006] address time-varying surfaces.

Image-based rendering and the light field

All traditional image synthesis methods take a description of a scene as an input, including various objects, materials, and lights. This description is then rendered into an output image. *Image-based rendering* works in a different way: in essence, it generates new, different views of a static scene with fixed illumination from an input set of pre-acquired images. This approach is interesting because the cost of rendering is now decoupled from the scene complexity. The idea here reminds us of the early work on *environment mapping* [*Blinn and Newell* 1976; *Greene* 1986], where we could view the environment only from a fixed position; Chen and Williams [1993] were the first to propose *view interpolation* to overcome this problem.

The seminal work of Levoy and Hanrahan [1996] introduces *light field rendering*. It is a more robust approach to image-based rendering. New images are created by slicing the high-dimensional light field. Gortler et al. [1996] call this light field a *lumigraph*. Capturing a complete light field using a conventional camera is a tedious and lengthy process, whereas using a *light field camera* (or *plenoptic camera*) simplifies this process significantly. Such a light field camera captures additional information about the direction of the light rays.

Working with light fields then enables interesting applications. For instance, Masselus et al. [2003] use an image-based technique to relight real-world objects illuminated by an incident light field. Garg et al. [2006] model the reflectance field between the incident and exitant light fields. The enormous information stored in a light field also leads us to more sophisticated techniques to reconstruct real-world scenes, which we discuss below.

Recent advances in real-world scene reconstruction using light field cameras

Light field cameras are based on early research by Gabriel Lippmann [1908]. The standard concept is to place an array of microlenses at the focal plane of the camera. There, incoming light rays are split up into conical shapes: each microlens on the focal plane now covers a *circular area* on the image sensor. This obviously reduces the effective sensor resolution, but enables us to acquire directional information about incoming light: a light ray perpendicular to the image plane will hit the sensor in the center of a circular area, while more slanting rays will be redirected more toward the boundaries. This allows the manipulation of both focus and (to a minor degree) the viewpoint after an image has been taken.

The advent of more affordable and handheld light field cameras has led to interesting new research. The seminal work of Ng et al. [2005] describes light field photography in detail. More recent work demonstrates that light field cameras are a powerful tool for one-shot passive 3D shape capture. Tao et al. [2013] combine defocus and correspondence for depth estimation. Follow-up work addresses glossy and specular surfaces [*Tao et al.* 2014; *Tao et al.* 2015]. Wang et al. [2015] faithfully take occlusion into account. More research reconstructs specular surfaces which are inherently difficult to capture while estimating the corresponding BRDF at the same time [*Wang et al.* 2017].

Direct real-world light transport visualization

So far we have discussed a variety of techniques to reconstruct real-world scenes and to visualize the associated light transport. How can we directly visualize the *de facto* light transport

in our environment without relying on computer graphics simulations? Hullin et al. [2008] describe a simple setup using fluorescent fluids that enables to visualize various optical phenomena, such as reflections, refractions, subsurface scattering, caustics, and interreflections.

More recently, the concept of *femto photography* regained interest. The idea is to record the propagation of ultrashort light pulses at high speed and it was first demonstrated in early works by Nils Abramson [1978]. Velten et al. [2013] capture and visualize light propagation using a streak camera synchronized to a pulsed laser. The work of Heide et al. [2013] extracts images from time-of-flight sensor data which obviates the need for high-speed detectors.

10.3 Our Approach

To reconstruct real-world light transport information, two distinct approaches come to mind:

1. Reconstruct a full 3D model of a scene with matching light sources and apply realistic material properties to all surfaces. This could be achieved using the various techniques discussed in the previous work section. We can then run a full global illumination simulation and apply our inspection techniques introduced back in Chapter 8.

2. Do not reconstruct an intermediate scene representation. Try to infer information directly from a set of photos (i.e., image space samples) of the scene.

Both approaches are very challenging and are located within the main active research topics of the computer vision community. In the previous section, we have outlined and referred to some of the major challenges regarding the reconstruction of the geometry, materials, lighting, and more specific phenomena of real-world scenes. Common problems in these areas include noisy sensor data with outliers and missing data in general; moreover, computer vision research frequently encounters ill-posed problems with unclear solutions.

In our pragmatic approach, set up to assess the usefulness of light transport inspection in real-world scenes, we will be operating directly in image space most of the time. However, our method indeed reconstructs the original 3D location of surface points. For that purpose we use photos with additional *depth information* and take note of the camera position and orientation. Our main focus will be on providing an interactive environment.

In our work we make a variety of simplifying assumptions: with correct information about depth and camera position and orientation, we can directly reconstruct surface locations. We further assume that input images were taken at the same point in time and do not contain too much motion, noise, or distracting fixed lighting such as highlights and reflections. The system is able to detect potential light sources in high-dynamic range imagery and then offers the user an interface to refine the initial classification. In order to produce a meaningful visualization, we require that the input images cover all important areas of a scene. We do not distinguish between different surface materials and we analyze scenes without participating media.

A brief description of all steps of our approach follows:

Data acquisition. We use high-dynamic range (HDR) photos with depth information as an input to our system. While our approach could produce limited results with tone-mapped

low-dynamic range photos, we obtain information about the raw illuminance on a sensor plane only with HDR input. Depth information can be acquired using a separate depth sensor or using 3D scanner data. We also have to record the camera position and orientation.

Loading files into system. A scene is represented by a set of *shots*. A shot is one HDR photo with attached depth and camera information. A shot is therefore a fixed *view* of the scene, whereas any view can be reconstructed by interpolating between different shots (given that sufficient data is available). In a typical computer vision setting, *image registration* covers the transformation of data from multiple input images from different viewpoints (with possibly different camera calibrations and settings) into a common coordinate system. In Subsection 10.4.2 we detail how we combine data from different shots into a single environment map of a light probe. Multiple shots are required to capture all areas of a scene. The amount of required shots increases for more complex scenes with multiple occluders.

Image space segmentation. The system can segment a shot of a scene into different areas. We then propose which areas cover potential light sources due to a high illuminance above a threshold. After the initial segmentation by our system, the user can select or deselect areas to refine the tagging. This allows us to distinguish between direct and indirect sources of light.

Light probes. We again use *light probes* to collect data required for visualizations. We did this already back in Subsection 8.3.2 when we were dealing with virtual scenes. There we were gathering photons which entered the region of interest around a light probe. Now, instead, for every shot of a scene, we spawn rays from the center of a probe and intersect them with surfaces in the image space. Finally we merge all results in a combined environment map.

Reconstructing the light field. Recall from Section 7.7 that the we can use the *plenoptic function* $L(x, y, z, \theta, \phi, \lambda, t)$ to determine the radiance for location (x, y, z) and orientation (θ, ϕ). We consider a light probe environment map as a sample of the light field for a fixed location, covering all orientations in discrete steps. In an environment map we have converted radiance to luminance and store it as discrete RGB triples, integrated over all wavelengths λ. For reasons of simplification we have to assume that all shots were taken at the same moment t.

10.4 Implementation

We have implemented an interactive environment, written in C++, using Qt 5.8 and OpenGL. We now give some technical details about the implementation of the steps outlined above.

10.4.1 Scene Segmentation

After importing a scene into our environment, we display all shots of this scene in the main widget. We offer different tone mapping operators to the user, allowing to adjust values such as gamma and exposure. This is necessary to produce tone-mapped low-dynamic range images

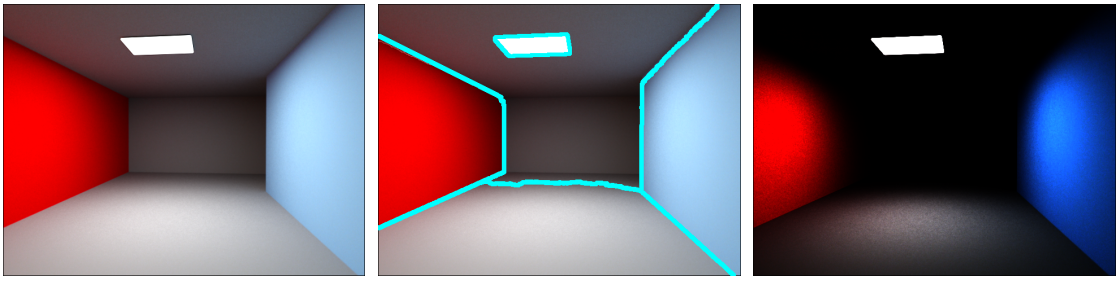


Figure 10.1: *Example of an image-space scene segmentation for a simple high-dynamic range input image. Adjusting tone mapping parameters reveals sources of direct and indirect lighting.*

which can be displayed on a monitor with a limited gamut and color space. We always keep high-dynamic range information in the background.

Such HDR information already allows to detect areas which are potential light sources, since they usually exhibit a much higher illuminance than other areas. In some cases the difference in illuminance even spans few orders of magnitude.

The system allows to cluster different areas of a scene. We use the hierarchical segmentation approach from Paris and Durand [2007] using mean shift clustering. For every pixel of a shot we construct a *feature vector*. This vector contains the (x, y) pixel coordinates in screen space, the reconstructed 3D (x, y, z) coordinates in space, and the high-dynamic range RGB values. This yields eight-dimensional vectors which we reduce to three dimensions via principal component analysis to enable a fast clustering in the feature space.

According to the illuminance of each segment, we estimate whether it covers a direct light source. Figure 10.1 shows an example where the system identified a direct light source on the ceiling correctly. The user can then manually refine this tagging. We keep a Boolean flag for each pixel to indicate whether it belongs to a direct light source. This information can later be used to filter out direct or indirect light for visualizations.

10.4.2 Sampling the Light Field with Light Probes

Similar to our previous approach for virtual scenes (Subsection 8.3.2), the user can place light probes on surfaces or in the 3D space to sample the light field. We then dispatch a compute shader for every shot which spawns rays from the light probe center into all directions covered by an environment map. We ray march directly in image space until a ray intersects with a surface point. Note that a light probe does not have to be visible inside a shot, but, obviously, we require a visible surface intersection point to store information in an environment map.

We also produce a *confidence map* for every shot. Such a confidence map stores a confidence value for every pixel we stored in the environment map based on the following criteria:

1. *The dot product of the view vector and the surface normal.* Areas parallel to the image plane have a higher confidence value. The goal is to avoid warped and distorted areas.

2. *The distance from a surface point to the camera.* The assumption is that a shorter distance indicates a better coverage of this area in this shot, and thus leads to a higher confidence.

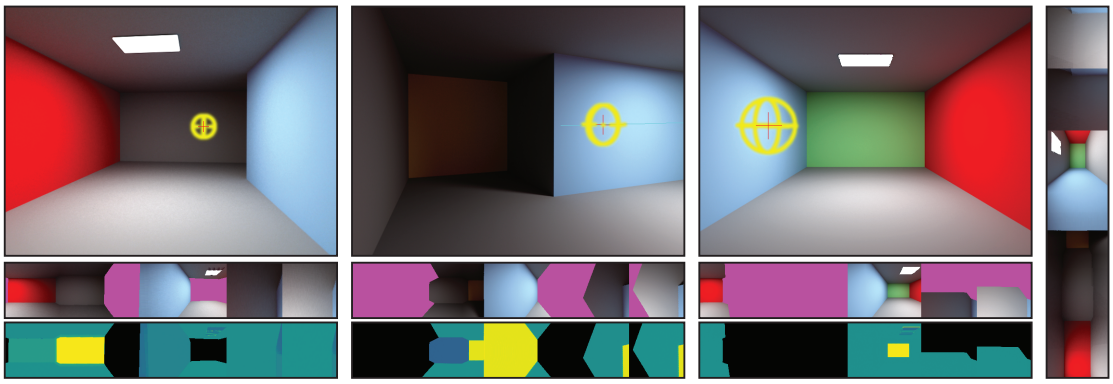


Figure 10.2: *Three shots of a very simple example scene. The user has positioned a light probe (in yellow) inside the scene. Below each shot the environment and confidence maps are displayed. (Magenta: no information; yellow: highest confidence.)* Far right: *combined environment map.*

All individual environment maps for the individual shots are then merged in a combined environment map for a light probe. According to its confidence value, a contributing point might be favored or penalized. Figure 10.2 shows an example for a very simple synthetic test scene. It contains three shots where each covers an area not visible in the other shots. The user has placed a light probe; while moving the mouse inside a shot, he also sees his position in space in the other shots. For each shot, environment and confidence maps are created. Note how every map covers different areas with varying confidence values. This allows to create a combined environment map which covers the full view of the light probe.

10.4.3 Revisiting Light Transport Inspection Tools

Let us briefly revisit our previous light transport inspection tools from Subsections 8.4.1 to 8.4.5, and discuss how we could reimplement them in our new environment.

With HDR imagery, we can again make use of *false-color renderings* to emphasize the true illuminance of a surface. In a way, we can regard this as a special tone-mapping operator which maps raw luminance values to a low-dynamic range color set.

Spherical plots can be implemented in an analogous manner as we did before: such a plot is generated by starting with a highly-tessellated sphere which is displaced according to the (low-pass filtered) environment map of a light probe.

Reimplementing the *light path inspection tool* would be more involved since we no longer have access to augmented photons which carry additional information about their interactions with the scene. One solution could be a faithful reconstruction of numerous light paths.

Volumetric inspection could be reimplemented by combining all possible entry and exit points of a virtual box; each pair of points defines a line to be intersected with all surfaces in the image space. If the intersections with the scene have been found, we crop a line to a segment which lies within the scene and passes through the virtual box. The endpoints of such a line segment are then colored according to the luminance of the surface points they touch.

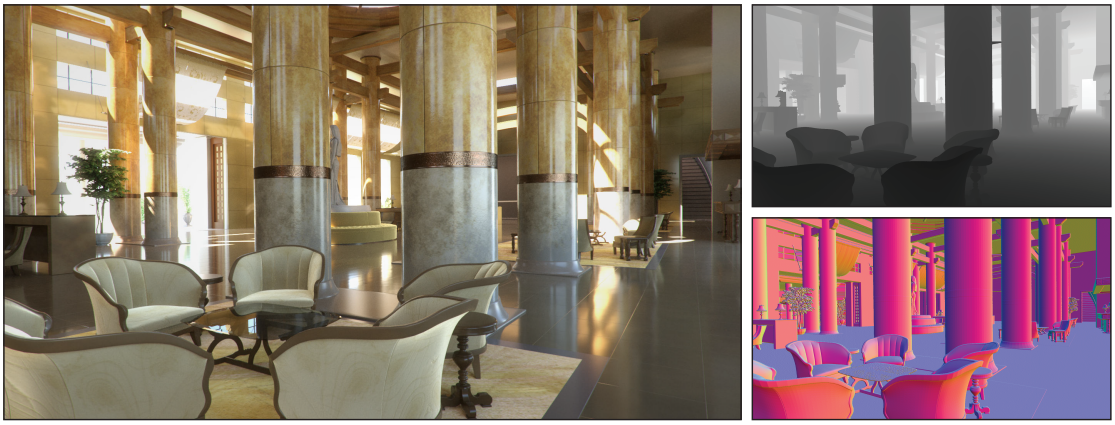


Figure 10.3: Left: *a more complex synthetic example of a hotel lobby.* Top Right: *associated depth map to reconstruct 3D coordinates.* Bottom Right: *the reconstructed surface normals.* **Acknowledgements:** *this scene is from the LuxRender show-off pack; author: Peter Sandbacka.*

A luminance value for all points along this line segment can be linearly interpolated. The part of the line which passes through the virtual box can then be blended over the final image.

The *particle flow tool* could be reimplemented, again by sampling the light field with smaller light probes for each particle. However, our previous user study indicated that both the volumetric and particle flow tools are rather unfocused and less directed when examining the entire illumination. We can no longer filter out large parts of overlaying light transport to inspect only specific phenomena, such as only specularly reflected light.

This is why we chose to display spherical plots in combination with false-color renderings for our results in this chapter. We will also introduce a new tool: a virtual light meter. This tool allows the user to scan surfaces and receive information about the illuminance; optionally, the user can switch to inspect the luminance for any point in a scene by orienting a virtual cone shape. We then integrate the luminous energy traveling through this shape.

10.5 Results

We will now examine a more complex synthetic test case before leading over to a real-world scene captured by photos. In the end we show how to measure photometric quantities (here: either illuminance or luminance) with our system using a virtual light meter.

10.5.1 A Synthetic Test Case

We have tested our interactive environment with synthetic test cases first. This makes testing a new method easier since we can compare the outcome against the ground truth determined by a global illumination simulation. Figure 10.3 shows one of our test scenes: a hotel lobby, modeled by Peter Sandbacka, and rendered using *LuxRender*, a physically based rendering engine. We used *LuxRender* to produce multiple shots of photographic quality, each including a depth map, for this scene. We do not import any other data into our environment.

Figure 10.4: Top Left: *false-color rendering.* Bottom Left: *illuminance contour lines.* Center and Right: *spherical plots visualizing the flow of light above a sitting area.* Insets: *chroma coloring of the plot and filtering out peaks by visualizing only indirect lighting.*

Having access to depth values and knowing the camera position, orientation, and field of view, we can reconstruct 3D coordinates of all pixels for each shot. This in turn allows us to estimate the surface normals, again in image space using the central differencing scheme.

In Figure 10.4 we can now see some results. We start with a false-color rendering of the HDR input using the *viridis* color map. With this perceptually uniform color map, we can better distinguish the wide range of luminance values over several orders of magnitude.

We can also execute a segmentation shader which brackets these high-dynamic range values together. At the interface between segments we then create vector paths to display the illuminance isocontour lines in screen space. These lines are rendered atop of the image.

Let us now assume the user wants to inspect the flow of light above a sitting area in the lobby. This can be done by placing a light probe at the desired location. The user first moves the mouse cursor to the center of table. The surface normal which points upwards is then displayed. The user can use the mouse wheel to raise or lower the light probe along the surface normal until the desired height is reached.

The light probe then samples all shots to create a combined environment map. A spherical plot is created according to the luminance values found. Using chroma coloring nicely integrates the plot into the environment, but it might be more difficult to interpret. Again using the viridis color map, we can better distinguish between the actual luminance values mapped to the spherical plot. The user can further opt to inspect only indirect lighting to filter out peaks.

10.5.2 A Real-World Example

Figure 10.5 shows a real-world example scene. This room is exposed to daylight by two windows in adjacent walls. Some of the furniture, namely the wardrobe and the table, consist of highly glossy and reflective materials. We took several shots of this scene with a Nikon D60 digital single-lens reflex camera. This camera allows to save photos in a raw image format, which enables us to access the raw sensor data.

Figure 10.5: *Different shots of a real-world scene. A spherical plot points to the windows and balcony of the room, which are not visible in the third shot.* Bottom Right: *a night shot.*

For this simple setup it was sufficient to render a few boxes as proxy geometry to reconstruct very smooth gradients and depth information. It would be interesting to offer the user a simple sketch-based interface directly inside our tool, similar to the work of Zeleznik et al. [1996], to create a basic approximate scene geometry for a set of input images without depth information.

A light probe has been placed in the center of the room and a spherical plot visualizes the flow of light trough this point. We can see how this plot points toward the windows, although these are only visible in the other shots. At the top of the plot, we can notice a peak which points toward reflections from the inner lampshade. At the bottom of the plot, we see a few peaks which are probably unwanted. They point toward reflected daylight on the table, which was visible in another shot. We will discuss the ramifications of *fixed lighting* in Section 10.6.

Figure 10.5 also includes a night shot of the scene. Again, the center of the room has been sampled by a light probe. We have *reoriented* the spherical plot in an inset to better display its shape. The strongest peak which now points to the left is caused by the ceiling lamp. The two peaks now at the bottom are caused by the two smaller lamps. The area now at the top of the spherical plot is caused by strong indirect lighting of the glossy wardrobe.

The spherical plots in this scene are set up to account for both direct and indirect light. Again, chroma coloring of the plot results in a smooth integration into the scene, whereas a false-color visualization helps to comprehend the magnitude of the luminance.

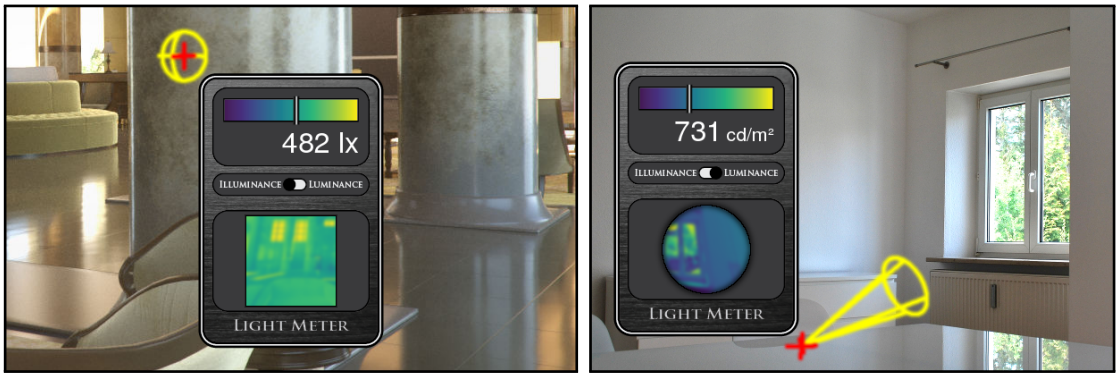


Figure 10.6: *A virtual light meter allows to measure photometric quantities.*

10.5.3 A Virtual Light Meter

We propose a virtual light meter to inspect photometric quantities at specified locations. This tool is inspired by state-of-the-art approaches in lighting analysis: applications such as Autodesk Ecotect can print illuminance values on a lattice which extends over the scene.

Figure 10.6 shows two examples. With our tool the user can move the mouse over surfaces to inspect the illuminance. The tool samples the environment in the same way as a light probe does. Instead of sampling the hemisphere around a surface point, the user can switch to a cone-shaped receiver which integrates only over directions covered by the cone. While holding the control key, the user can change the orientation of the cone by moving the mouse, and adjust the solid angle with the mouse wheel.

The tool also displays the part of the environment which contributes to the measurement. Note that this tool simply integrates over the raw values we obtained from a photosensor or global illumination simulation, and correct values will be displayed only in the case of meaningful input data.

10.6 Limitations

Since we rely on a couple of photos as input data, there are several limitations to our approach. While we are well able to obtain precise depth information for our synthetic test scenes, this is not always the case when capturing real-world data. Specular and glossy surfaces reflect laser beams and light patterns, making it difficult to measure the correct depth. Moreover, at the moment we do not handle transparency and refractive materials correctly. This is an active research topic: Sinha et al. [2012] presented an image-based system that handles reflections well; their approach could help to improve the rendering quality of our visualizations.

By taking photos of a scene we capture a *fixed lighting scenario*. This leads to artifacts when specular highlights remain at a fixed position, instead of changing according to the view. Photos may also include unwanted flash lights, shadows, or reflections of the photographer or scanning device. Taking photos also implies that we are dealing with static scenes, whereas our previous techniques for virtual scenes can be readily used for fully animated scenes.

For our scenes we took a sufficient number of shots to capture most of the surfaces. This is a usual approach for scene acquisition. However, we might not be able to capture all surfaces for more complex scenes. Occluded surfaces will not be covered in an environment map then. This leads to visualizations as if those surfaces were not present at all.

10.7 Discussion

The second part of this thesis began with an introduction to the fundamentals of light transport in computer graphics, also covering basic radiometric and photometric quantities. We then presented visual tools to inspect the light transport in virtual scenes. In this chapter we have transferred some of our previous techniques to the real-world which surrounds us.

In the previous work section we outlined several challenges at the interface of computer graphics and vision and we pointed to a variety of related work which tackles a broad range of problems when reconstructing real-world scenes.

We presented a pragmatic approach to reconstruct information from a set of HDR input images with depth information. For the assessment of light transport inspection tools in real-world scenes we made a variety of strong simplifying assumptions. We started from the premise that we are dealing with (more or less) perfect input data, and further neglect commonly occurring computer vision problems.

We again implemented an interactive environment and used light probes to sample the light field, this time using image space techniques, and recreated some spherical plots to demonstrate our approach. We also proposed a virtual light meter to measure photometric quantities. While facing new limitations without a full global illumination simulation at hand, we show how to infer basic insights about the flow of light in real-world scenes.

11

Conclusions

Throughout this thesis we have covered several research projects in two principal domains of computer graphics: the creation of geometry and the illumination of virtual scenes. We have seen that both *geometry* and *light* are key aspects to create appealing imagery. Our procedural modeling approach allows us to create scenes with rich visual complexity. We have also introduced tools which help to understand and improve the illumination of such scenes.

We have first presented a complete interactive modeling environment for implicit surfaces. This allows us to model implicit shapes in a very intuitive way. With our "what you see is what you get" interface we have returned the controllability of the outcome to the user. What sets our system apart from previous approaches is that we are directly rendering the scene. We are not dependent on intermediate representations created by a polygonizer.

This modeling environment is based on purely analytical forms and allows for resolutions in arbitrary detail. On the other hand, we can no longer deal with more complex surface deformations in real-time. This is why we have introduced a runtime cache to store the results of complex displacement functions. With a fixed amount of memory allocated, such a cache dynamically adapts to new views and accelerates the display of our scenes.

Many users would like to go one step further and create physical representations of their models. We have thus implemented new sculpting tools for our modeling environment. For that reason we have extended our system with a voxel-based representation of shapes. It allows us to process a variety of modeling operations very efficiently in the 3D voxel space. We have introduced new operations which assist the user in creating shapes that print without the need for support structures. This makes 3D printing more accessible to home users.

We again addressed surface details and presented a new method to print textured objects with affordable consumer hardware. Using only two different types of filament, we can now print patterns which dissolve into an in-between tone when viewed from a distance. This means in effect that we are able to print two-tone imagery.

In the second part of this thesis we have addressed visual illumination inspection tools. We have first introduced a set of novel tools to visualize and inspect the high-dimensional light transport in virtual scenes. These tools help to understand more complex lighting scenarios. A powerful feature is that we can choose to inspect only *selected* phenomena, for example, caustics. This allows us to filter out large parts of overlaying light transport which is of no interest at that time. We have also conducted a thorough user study to evaluate the

usefulness of all tools for different tasks. We found that the proper usage of our tools can provide significant new insights about the complex illumination of a virtual scene.

One possible use case then is to support the artistic *manipulation* of light transport. We have briefly discussed a sophisticated approach which combines the selection, inspection, visualization, and manipulation of specific lighting phenomena. This approach handles animations well and seamlessly integrates into a modern 3D modeling environment.

Finally, we have also transferred our methods to the real-world to inspect the illumination in scenes captured by photos with depth information. We are now operating directly in image space without having access to a full global illumination simulation in the background. We are deriving information about the light transport from these limited samples as faithful as possible, as free as necessary. Again, we sample the plenoptic function with light probes. In the end we revisited our inspection techniques and applied them to synthetic and real test cases.

The *interactivity* is a primary feature of all our methods and the common theme throughout all chapters. The user always receives immediate visual feedback after every modeling and inspection step. It makes all our methods more convenient and more accessible, and we have also received encouraging feedback from users in this regard. They have unanimously agreed that interactivity is crucial and of great value for all our tools.

Acknowledgements

It has been a long journey from my first day as a Ph.D. student to the completion of my dissertation. I am very grateful to everybody who supported me during these years.

In the first place, I'd like to express my sincere gratitude to Carsten Dachsbacher, my advisor, for his permanent support. His ideas and substantial contributions formed the foundation of our research projects. I'm very glad that I was part of his first group of Ph.D. students!

My gratitude is extended to Michael Wimmer, my second reviewer. He made comprehensive contributions to many areas of computer graphics research, especially in topics addressed by my work, and I am very pleased to have him in my thesis committee. At this point I'd also like to thank Boris Neubert and Björn Hein who kindly accepted to take the role of examiners.

I'd like to give special thanks to Sylvain Lefebvre, who gave me the opportunity to work in his team. We did exciting research in close collaboration and I had a wonderful year in Nancy!

Next I'd like to thank Nathan Carr, who hosted me at Adobe Research, for a successful graduate internship. I'm very grateful for the great experience I had in this fruitful environment.

I could not have succeeded without the hard work and invaluable support of my co-authors! We accomplished so much together. I cannot thank all of them enough.

I must acknowledge with deep thanks my parents and family. Thank you so much for everything! I am very happy to have you around me.

Authored References

The contributions of this thesis have been published in the following computer graphics journals. These papers are the foundation of this thesis and have been extended to a certain degree. They are listed in the order in which they appear as individual chapters:

- Tim Reiner, Gregor Mückl, and Carsten Dachsbacher. **Interactive Modeling of Implicit Surfaces using a Direct Visualization Approach with Signed Distance Functions.** *Computers & Graphics 35(3), Proceedings of Shape Modeling International 2011.*
- Tim Reiner, Sylvain Lefebvre, Lorenz Diener, Ismael García, Bruno Jobard, and Carsten Dachsbacher. **A Runtime Cache for Interactive Procedural Modeling.** *Computers & Graphics 36(5), Proceedings of Shape Modeling International 2012.*
- Tim Reiner and Sylvain Lefebvre. **Interactive Modeling of Support-free Shapes for Fabrication.** *Eurographics 2016 Short Paper.*
- Tim Reiner, Nathan Carr, Radomír Měch, Ondřej Šťava, Carsten Dachsbacher, and Gavin Miller. **Dual-Color Mixing for Fused Deposition Modeling Printers.** *Computer Graphics Forum 33(2), Proceedings of Eurographics 2014.*
- Tim Reiner, Anton Kaplanyan, Marcel Reinhard, and Carsten Dachsbacher. **Selective Inspection and Interactive Visualization of Light Transport in Virtual Scenes.** *Computer Graphics Forum 31(2), Proceedings of Eurographics 2012.*

Moreover, the author has contributed to the following paper, which is briefly addressed in Chapter 9 in the context of artistic light transport manipulation:

- Thorsten Schmidt, Jan Novák, Johannes Meng, Anton Kaplanyan, Tim Reiner, Derek Nowrouzezahrai, and Carsten Dachsbacher. **Path-Space Manipulation of Physically-Based Light Transport.** *ACM Transactions on Graphics 32(4), SIGGRAPH 2013.*

Bibliography

[ABRAMSON 1978] Nils Abramson. Light-in-flight recording by holography. *Optics Letters*, 3(4):121–123, October 1978.

[ADELSON AND BERGEN 1991] Edward H. Adelson and James R. Bergen. The plenoptic function and the elements of early vision. In M. Landy and J. Anthony Movshon, editors, *Computational Models of Visual Processing*, pages 3–20. MIT Press, 1991.

[AITTALA ET AL. 2015] Miika Aittala, Tim Weyrich, and Jaakko Lehtinen. Two-shot SVBRDF capture for stationary materials. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 34(4):110:1–110:13, July 2015.

[AKLEMAN AND CHEN 1999] Ergun Akleman and Jianer Chen. Generalized distance functions. In *Proceedings of Shape Modeling International*, pages 72–79. IEEE, 1999.

[ALCANTARA ET AL. 2009] Dan Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John Owens, and Nina Amenta. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 28(5):154:1–154:9, December 2009.

[ALCANTARA ET AL. 2011] Dan Alcantara, Vasily Volkov, Shubhabrata Sengputa, Michael Mitzenmacher, John Owens, and Nina Amenta. Building an efficient hash table on the GPU. GPU Computing Gems, Jade Edition, 2011.

[ALEXANDER ET AL. 1998] Paul Alexander, Seth Allen, and Debasish Dutta. Part orientation and build cost determination in layered manufacturing. *Computer-Aided Design*, 30(5):343–356, 1998.

[AMANATIDES AND WOO 1987] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Proceeding of Eurographics 1987*, pages 3–10, 1987.

[BÄCHER ET AL. 2012] Moritz Bächer, Bernd Bickel, Doug L. James, and Hanspeter Pfister. Fabricating articulated characters from skinned meshes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 31(4):47:1–47:9, 2012.

[BÄCHER ET AL. 2014] Moritz Bächer, Emily Whiting, Bernd Bickel, and Olga Sorkine-Hornung. Spin-it: Optimizing moment of inertia for spinnable objects. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 33(4):96:1–96:10, 2014.

[BAI 2010] Jiamin Bai. Interactive visualization for light transport matrices. *Online reference*. `http://vis.berkeley.edu/courses/cs294-10-sp10/wiki/images/f/f0/JiaminBai.pdf`, 2010.

[BARNARD AND THOMPSON 1980] Stephen T. Barnard and William B. Thompson. Disparity analysis of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(4):333–340, April 1980.

[BARNES AND ZHANG 2017] Connelly Barnes and Fang-Lue Zhang. A survey of the state-of-the-art in patch-based synthesis. *Computational Visual Media*, 3(1):3–20, 2017.

[BARRON AND MALIK 2013] Jonathan T. Barron and Jitendra Malik. Intrinsic scene properties from a single RGB-D image. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, pages 17–24. IEEE Computer Society, 2013.

[BARROW AND TENENBAUM 1978] Harry G. Barrow and Jay M. Tenenbaum. Recovering intrinsic scene characteristics from images. *Computer Vision Systems*, pages 3–26, 1978.

[BARZEL 1997] Ronen Barzel. Lighting controls for computer cinematography. *Journal of Graphics Tools*, 2(1):1–20, 1997.

[BASTOS AND CELES 2008] Thiago Bastos and Waldemar Celes. GPU-accelerated adaptively sampled distance fields. In *Shape Modeling International 2008*, pages 171–178. IEEE, 2008.

[BELL ET AL. 2014] Sean Bell, Kavita Bala, and Noah Snavely. Intrinsic images in the wild. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 33(4):159:1–159:12, July 2014.

[BERNARDINI AND RUSHMEIER 2002] Fausto Bernardini and Holly Rushmeier. The 3D model acquisition pipeline. *Computer Graphics Forum*, 21(2):149–172, 2002.

[BICKEL ET AL. 2010] Bernd Bickel, Moritz Bächer, Miguel A. Otaduy, Hyunho Richard Lee, Hanspeter Pfister, Markus Gross, and Wojciech Matusik. Design and fabrication of materials with desired deformation behavior. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 29(4):63:1–63:10, 2010.

[BLEYER AND BREITENEDER 2013] Michael Bleyer and Christian Breiteneder. *Stereo Matching: State-of-the-Art and Research Challenges*, pages 143–179. Springer London, 2013.

[BLINN AND NEWELL 1976] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, October 1976.

[BLINN 1982] James F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[BLOOMENTHAL AND SHOEMAKE 1991] Jules Bloomenthal and Ken Shoemake. Convolution surfaces. *SIGGRAPH Computer Graphics*, 25:251–256, July 1991.

[BLOOMENTHAL AND WYVILL 1990] Jules Bloomenthal and Brian Wyvill. Interactive techniques for implicit modeling. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, I3D '90, pages 109–116, New York, NY, USA, 1990. ACM.

[BLOOMENTHAL AND WYVILL 1997] Jules Bloomenthal and Brian Wyvill, editors. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[BORLAND AND TAYLOR 2007] David Borland and Russell M. Taylor. Rainbow color map (still) considered harmful. *IEEE Computer Graphics and Applications*, 27(2):14–17, 2007.

[BOUBEKRI ET AL. 2014] Mohamed Boubekri, Ivy N. Cheung, Kathryn J. Reid, Nai-Wen Kuo, Chia-Hui Wang, and Phyllis C. Zee. Impact of windows and daylight exposure on overall health and sleep quality of office workers: A case-control pilot study. *Journal of clinical sleep medicine, American Academy of Sleep Medicine*, 10(6):603–614, 2014.

[BOYCE ET AL. 2003] Peter Boyce, Claudia Hunter, and Owen Howlett. *The Benefits of Daylight through Windows*. Lighting Research Center, Rensselaer Polytechnic Institute, 2003.

[BRABEC AND SEIDEL 2002] Stefan Brabec and Hans-Peter Seidel. Single sample soft shadows using depth maps. In *In Graphics Interface*, pages 219–228, 2002.

[CALAHAN 1999] Sharon Calahan. Storytelling through lighting, a computer graphics perspective. In A. A. Apodaca, L. Gritz, and R. Barzel, editors, *Advanced RenderMan: Creating CGI for Motion Pictures*, pages 223–233. Morgan Kaufmann, 1999.

[CALÌ ET AL. 2012] Jacques Calì, Dan A. Calian, Cristina Amati, Rebecca Kleinberger, Anthony Steed, Jan Kautz, and Tim Weyrich. 3D-printing of non-assembly, articulated models. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 31(6):130:1–130:8, 2012.

[CALIFORNIA ENERGY COMMISSION 2003] California Energy Commission. Windows and offices: a study of office worker performance and the indoor environment, 2003.

[CANI ET AL. 2008] Marie-Paule Cani, Takeo Igarashi, and Geoff Wyvill. *Interactive Shape Design*. Synthesis Lectures on Computer Graphics and Animation. Morgan & Claypool, 2008.

[CASTRO ET AL. 2008] Rener Castro, Thomas Lewiner, Hélio Lopes, Geovan Tavares, and Alex Bordignon. Statistical optimization of octree searches. *Computer Graphics Forum*, 27(6):1557–1566, 2008.

[CATMULL 1974] Edwin Earl Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Computer Science Department, University of Utah, 1974.

[CEREZO ET AL. 2005] Eva Cerezo, Frederic Pérez, Xavier Pueyo, Francisco J. Seron, and François X. Sillion. A survey on participating media rendering techniques. *The Visual Computer*, 21(5):303–328, 2005.

[CHEN AND KOLTUN 2013] Qifeng Chen and Vladlen Koltun. A simple model for intrinsic image decomposition with depth cues. In *IEEE International Conference on Computer Vision*, pages 241–248, 2013.

[CHEN AND WILLIAMS 1993] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 279–288, New York, NY, USA, 1993. ACM.

[CHEN ET AL. 2008] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. Interactive procedural street modeling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 27(3):103:1–103:10, August 2008.

[CHEN ET AL. 2013] Desai Chen, David I. W. Levin, Piotr Didyk, Pitchaya Sitthi-Amorn, and Wojciech Matusik. Spec2Fab: a reducer-tuner model for translating specifications to 3D prints. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 32(4):135:1–135:10, 2013.

[CHEN ET AL. 2015] Xuelin Chen, Hao Zhang, Jinjie Lin, Ruizhen Hu, Lin Lu, Qixing Huang, Bedrich Benes, Daniel Cohen-Or, and Baoquan Chen. Dapper: Decompose-and-pack for 3D printing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 2015.

[CHO ET AL. 2003] Wonjoon Cho, Emanuel M. Sachs, Nicholas M. Patrikalakis, and Donald E. Troxel. A dithering algorithm for local composition control with three-dimensional printing. *Computer-Aided Design*, 35(9):851–867, 2003.

[CHRISTENSEN ET AL. 2006] P. H. Christensen, J. Fong, D. M. Laur, and D. Batali. Ray Tracing for the Movie 'Cars'. *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 2006.

[COHEN ET AL. 1988] Michael F. Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A progressive refinement approach to fast radiosity image generation. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 75–84, New York, NY, USA, 1988. ACM.

[COHEN ET AL. 1993] Michael F. Cohen, John Wallace, and Pat Hanrahan. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[COOK AND TORRANCE 1982] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics*, 1(1):7–24, January 1982.

[COROS ET AL. 2013] Stelian Coros, Bernhard Thomaszewski, Gioacchino Noris, Shinjiro Sueda, Moira Forberg, Robert Sumner, Wojciech Matusik, and Bernd Bickel. Computational design of mechanical characters. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 32(4):83:1–83:12, 2013.

[CRASSIN ET AL. 2009] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, pages 15–22. ACM, 2009.

[CROW 1977] Franklin C. Crow. The aliasing problem in computer-generated shaded images. *Commun. ACM*, 20(11):799–805, November 1977.

[CURLESS 2000] Brian Curless. From range scans to 3D models. *SIGGRAPH Computer Graphics Newsletter*, 33(4):38–41, November 2000.

[DACHSBACHER AND KAUTZ 2009] Carsten Dachsbacher and Jan Kautz. Real-time global illumination for dynamic scenes. In *SIGGRAPH Courses*, 2009.

[DACHSBACHER 2006] Carsten Dachsbacher. *Interactive Terrain Rendering: Towards Realism with Procedural Models and Graphics Hardware*. PhD thesis, 2006.

[DEBEVEC AND MALIK 1997] Paul E. Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 369–378, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[DINH ET AL. 2015] H. Quynh Dinh, Filipp Gelman, Sylvain Lefebvre, and Frédéric Claux. Modeling and toolpath generation for consumer-level 3D printing. In *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, 2015.

[DUMAS ET AL. 2014] Jérémie Dumas, Jean Hergel, and Sylvain Lefebvre. Bridging the gap: Automated steady scaffoldings for 3D printing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 33(4):98:1–98:10, July 2014.

[DURAND ET AL. 2005] Frédo Durand, Nicolas Holzschuch, Cyril Soler, Eric Chan, and François X. Sillion. A frequency analysis of light transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 24(3):1115–1126, 2005.

[DUTRÉ ET AL. 2006] Philip Dutré, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination, 2nd edition*. AK Peters, 2006.

[EBERT ET AL. 2002] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 3rd edition, 2002.

[EDWARDS AND TORCELLINI 2002] L. Edwards and P. Torcellini. *A Literature Review of the Effects of Natural Light on Building Occupants*. National Renewable Energy Lab., 2002.

[EFROS AND LEUNG 1999] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the International Conference on Computer Vision*, ICCV '99, Washington, DC, USA, 1999. IEEE Computer Society.

[EISEMANN ET AL. 2011] Elmar Eisemann, Michael Schwarz, Ulf Assarsson, and Michael Wimmer. *Real-Time Shadows*. CRC Press, Taylor & Francis Group, 2011.

[EVANS 2006] Alex Evans. Fast approximations for global illumination on dynamic scenes. In *ACM SIGGRAPH 2006 Courses*, pages 153–171, New York, NY, USA, 2006. ACM.

[FERLEY ET AL. 1999] Eric Ferley, Marie-Paule Cani, and Jean-Dominique Gascuel. Practical volumetric sculpting. In *Implicit Surfaces*, 1999.

[FOLEY ET AL. 1996] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice (Second Edition in C)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[FRISKEN ET AL. 2000] Sarah Frisken, Ronald Perry, Alyn Rockwood, and Thouis Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 249–254. ACM, 2000.

[GALYEAN AND HUGHES 1991] Tinsley A. Galyean and John F. Hughes. Sculpting: An interactive volumetric modeling technique. *Proceedings of the 18th annual conference on Computer graphics and interactive techniques (SIGGRAPH '91)*, 25(4):267–274, 1991.

[GARCÍA ET AL. 2011] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent parallel hashing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 30(6):161:1–161:8, 2011.

[GARG ET AL. 2006] Gaurav Garg, Eino-Ville Talvala, Marc Levoy, and Hendrik P. Lensch. Symmetric photography: Exploiting data-sparseness in reflectance fields. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, EGSR '06, pages 251–262, 2006.

[GERSHUN 1939] Andreī Gershun. The Light Field. Translated by Parry Moon and Gregory Timoshenko. *Studies in Applied Mathematics*, 1939.

[GHOSH ET AL. 2007] Abhijeet Ghosh, Shruthi Achutha, Wolfgang Heidrich, and Matthew O'Toole. BRDF acquisition with basis illumination. In *International Conference on Computer Vision*, pages 1–8, 2007.

[GLASSNER 1989] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press Ltd., London, UK, 1989.

[GLASSNER 1994] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[GOESELE ET AL. 2004] Michael Goesele, Hendrik P. A. Lensch, Jochen Lang, Christian Fuchs, and Hans-Peter Seidel. Disco: Acquisition of translucent objects. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 23(3):835–844, August 2004.

[GORAL ET AL. 1984] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 213–222, New York, NY, USA, 1984. ACM.

[GORTLER ET AL. 1996] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. *SIGGRAPH '96*, pages 43–54, 1996.

[GREENE 1986] Ned Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, November 1986.

[GU ET AL. 2006] Jinwei Gu, Chien-I Tu, Ravi Ramamoorthi, Peter Belhumeur, Wojciech Matusik, and Shree Nayar. Time-varying surface appearance: Acquisition, modeling and rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 25(3):762–771, 2006.

[HAASER ET AL. 2015] Georg Haaser, Harald Steinlechner, Stefan Maierhofer, and Robert F. Tobler. An incremental rendering VM. In *Proceedings of the 7th Conference on High-Performance Graphics*, HPG '15, pages 51–60, New York, NY, USA, 2015. ACM.

[HACHISUKA AND JENSEN 2009] Toshiya Hachisuka and Henrik Wann Jensen. Stochastic progressive photon mapping. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 28(3):141:1–141:8, 2009.

[HACHISUKA AND JENSEN 2010] Toshiya Hachisuka and Henrik Wann Jensen. Parallel progressive photon mapping on GPUs. In *ACM SIGGRAPH Asia Sketches*, SIGGRAPH Asia '10, pages 54:1–54:1, 2010.

[HACHISUKA ET AL. 2008] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 27(5):130:1–130:8, December 2008.

[HAN AND PERLIN 2003] Jefferson Y. Han and Ken Perlin. Measuring bidirectional texture reflectance with a kaleidoscope. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 22(3):741–748, July 2003.

[HANRAHAN ET AL. 1991] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, pages 197–206. ACM, 1991.

[HART 1996] John C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12:527–545, 1996.

[HARTLEY AND ZISSERMAN 2003] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, New York, 2nd edition, 2003.

[HAUSER ET AL. 2003] Helwig Hauser, Robert S. Laramee, Helmut Doleisch, Frits H. Post, and Benjamin Vrolijk. The state of the art in flow visualization, part 1: Direct, texture-based, and geometric techniques. Tr-vrvis-2002-046, VRVis Research Center, Austria and Delft University of Technology, The Netherlands, 2003.

[HAŠAN ET AL. 2010] Miloš Hašan, Martin Fuchs, Wojciech Matusik, Hanspeter Pfister, and Szymon Rusinkiewicz. Physical reproduction of materials with specified subsurface scattering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 29(4):61:1–61:10, 2010.

[HAWKINS ET AL. 2005] Tim Hawkins, Per Einarsson, and Paul Debevec. Acquisition of time-varying participating media. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 24(3):812–815, July 2005.

[HE ET AL. 1991] Xiao D. He, Kenneth E. Torrance, François X. Sillion, and Donald P. Greenberg. A comprehensive physical model for light reflection. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, pages 175–186, New York, NY, USA, 1991. ACM.

[HECKBERT 1990] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics (Proceedings of SIGGRAPH '90)*, 24(4):145–154, 1990.

[HEIDE ET AL. 2013] Felix Heide, Matthias B. Hullin, James Gregson, and Wolfgang Heidrich. Low-budget transient imaging using photonic mixer devices. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 32(4):45:1–45:10, July 2013.

[HERGEL AND LEFEBVRE 2014] Jean Hergel and Sylvain Lefebvre. Clean color: Improving multi-filament 3D prints. *Computer Graphics Forum (Proceedings of Eurographics)*, 33(2):469–478, 2014.

[HERGEL AND LEFEBVRE 2015] Jean Hergel and Sylvain Lefebvre. 3D fabrication of 2D mechanisms. *Computer Graphics Forum (Proceedings of Eurographics)*, 34:229–238, 2015.

[HERHOLZ ET AL. 2015] Philipp Herholz, Wojciech Matusik, and Marc Alexa. Approximating free-form geometry with height fields for manufacturing. *Computer Graphics Forum (Proceedings of Eurographics)*, 34(2):239–251, 2015.

[HILDEBRAND ET AL. 2013] Kristian Hildebrand, Bernd Bickel, and Marc Alexa. Orthogonal slicing for additive manufacturing. *Computers & Graphics (Proceedings of Shape Modeling International)*, 37(6):669–675, 2013.

[HORNUS ET AL. 2003] Samuel Hornus, Alexis Angelidis, and Marie-Paule Cani. Implicit modeling using subdivision curves. *The Visual Computer*, 19:94–104, 2003.

[HORNUS ET AL. 2015] Samuel Hornus, Sylvain Lefebvre, Jérémie Dumas, and Frédéric Claux. Tight printable enclosures for additive manufacturing. Technical report, INRIA, 2015.

[HU ET AL. 2014] Ruizhen Hu, Honghua Li, Hao Zhang, and Daniel Cohen-Or. Approximate pyramidal shape decomposition. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 33(6), 2014.

[HU ET AL. 2015] Kailun Hu, Shuo Jin, and Charlie C.L. Wang. Support slimming for single material based additive manufacturing. *Computer-Aided Design*, 65:1–10, 2015.

[HUA AND QIN 2004] Jing Hua and Hong Qin. Haptics-based dynamic implicit solid modeling. *IEEE Transactions on Visualization and Computer Graphics*, 10:574–586, September 2004.

[HULLIN ET AL. 2008] Matthias B. Hullin, Martin Fuchs, Boris Ajdin, Ivo Ihrke, Hans-Peter Seidel, and Hendrik P. A. Lensch. Direct visualization of real-world light transport. In *Proceedings of Vision, Modeling, and Visualization*, pages 363–372, 2008.

[HULLIN ET AL. 2013] Matthias B. Hullin, Ivo Ihrke, Wolfgang Heidrich, Tim Weyrich, Gerwin Damberg, and Martin Fuchs. State of the art in computational fabrication and display of material appearance. In *Eurographics State of the Art Reports*, 2013.

[IGARASHI ET AL. 1999] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3D freeform design. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 409–416, 1999.

[IMMEL ET AL. 1986] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 133–142, 1986.

[IZADI ET AL. 2011] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and Andrew Fitzgibbon. KinectFusion: Real-time 3D reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 559–568, New York, NY, USA, 2011. ACM.

[JAROSZ 2008] Wojciech Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, UC San Diego, September 2008.

[JENSEN 1996] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, 1996.

[JENSEN 2012] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping, 2nd Edition*. Taylor & Francis, 2012.

[KAJIYA 1986] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[KAPLANYAN AND DACHSBACHER 2013] Anton Kaplanyan and Carsten Dachsbacher. Adaptive progressive photon mapping. *ACM Transactions on Graphics*, 32(2):16:1–16:13, April 2013.

[KAWAI ET AL. 1993] John K. Kawai, James S. Painter, and Michael F. Cohen. Radioptimization: goal based rendering. In *SIGGRAPH '93*, pages 147–154, 1993.

[KERR AND PELLACINI 2009] William B. Kerr and Fabio Pellacini. Toward evaluating lighting design interface paradigms for novice users. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 28(3):26:1–26:9, 2009.

[KERR ET AL. 2010] William B. Kerr, Fabio Pellacini, and Jonathan D. Denning. Bendylights: artistic control of direct illumination by curving light rays. *Computer Graphics Forum*, 29(4):1451–1459, 2010.

[KNAUS AND ZWICKER 2011] Claude Knaus and Matthias Zwicker. Progressive photon mapping: A probabilistic approach. *ACM Transactions on Graphics*, 30(3):25:1–25:13, 2011.

[KOO ET AL. 2014] Bongjin Koo, Wilmot Li, JiaXian Yao, Maneesh Agrawala, and Niloy J. Mitra. Creating works-like prototypes of mechanical objects. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 2014.

[LAFORTUNE AND WILLEMS 1993] Eric Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proceedings of the Third International Conference on Computational Graphics and Visualization Techniques*, 1993.

[LAGAE ET AL. 2010] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony DeRose, George Drettakis, David S. Ebert, John P. Lewis, Ken Perlin, and Matthias Zwicker. A survey of procedural noise functions. *Computer Graphics Forum*, 29(8):2579–2600, 2010.

[LAND AND MCCANN 1971] Edwin H. Land and John J. McCann. Lightness and retinex theory. *Journal of the Optical Society of America*, 61(1):1–11, Jan 1971.

[LARSON AND SHAKESPEARE 2004] Greg Ward Larson and Rob Shakespeare. *Rendering With Radiance: The Art And Science Of Lighting Visualization*. Booksurge Llc, 2004.

[LAZAROS ET AL. 2008] Nalpantidis Lazaros, Georgios Christou Sirakoulis, and Antonios Gasteratos. Review of stereo vision algorithms: from software to hardware. *International Journal of Optomechatronics*, 2(4):435–462, 2008.

[LEFEBVRE AND HOPPE 2005] Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 24(3):777–786, 2005.

[LEFEBVRE AND HOPPE 2006] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 25(3):579–588, 2006.

[LEGATES ET AL. 2014] Tara A. LeGates, Diego C. Fernandez, and Samer Hattar. Light as a central modulator of circadian rhythms, sleep and affect. *Nature Reviews Neuroscience*, 15:443–454, 2014.

[LEVIN ET AL. 2013] Anat Levin, Daniel Glasner, Ying Xiong, Fredo Durand, Bill Freeman, Wojciech Matusik, and Todd Zickler. Fabricating BRDFs at high spatial resolution using wave optics. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 32(4):144:1–144:14, 2013.

[LEVOY AND HANRAHAN 1996] Marc Levoy and Pat Hanrahan. Light field rendering. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 31–42, New York, NY, USA, 1996. ACM.

[LINDENMAYER 1968] Aristid Lindenmayer. Mathematical models for cellular interaction in development: Parts I and II. *Journal of Theoretical Biology*, 18, 1968.

[LIPP ET AL. 2008] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 27(3):102:1–102:10, August 2008.

[LIPPMANN 1908] Gabriel Lippmann. Épreuves réversibles donnant la sensation du relief. *Journal de Physique Théorique et Appliquée*, 7(1):821–825, 1908.

[LOOP AND BLINN 2006] Charles Loop and Jim Blinn. Real-time GPU rendering of piecewise algebraic surfaces. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 25:664–670, 2006.

[LORENSEN AND CLINE 1987] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics (SIGGRAPH 1987)*, 21(4):163–169, 1987.

[LOTTES 2009] Timothy Lottes. FXAA. Technical report, NVIDIA White Paper, 2009.

[LOWELL 1992] Ross Lowell. *Matters of light & depth: creating memorable images for video, film & stills through lighting*. Broad Street Books, 1992.

[LUFT ET AL. 2006] Thomas Luft, Carsten Colditz, and Oliver Deussen. Image enhancement by unsharp masking the depth buffer. *ACM Transactions on Graphics*, 25(3):1206–1213, 2006.

[LUO ET AL. 2012] Linjie Luo, Ilya Baran, Szymon Rusinkiewicz, and Wojciech Matusik. Chopper: partitioning models into 3D-printable parts. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 31(6):129:1–129:9, 2012.

[MAGNOR ET AL. 2015] Marcus Magnor, Oliver Grau, Olga Sorkine, and Christian Theobalt, editors. *Digital Representations of the Real World: How to Capture, Model, and Render Visual Reality*. CRC Press, 2015.

[MARSCHNER ET AL. 2000] Stephen R. Marschner, Stephen H. Westin, Eric P. F. Lafortune, and Kenneth E. Torrance. Image-based bidirectional reflectance distribution function measurement. *Applied Optics*, 39(16):2592–2600, June 2000.

[MARTINEZ ESTURO ET AL. 2010] J. Martinez Esturo, C. Rössl, and H. Theisel. Continuous deformations of implicit surfaces. In *Proceedings of Vision, Modeling, and Visualization*, 2010.

[MASSELUS ET AL. 2003] Vincent Masselus, Pieter Peers, Philip Dutré, and Yves D. Willems. Relighting with 4D incident light fields. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 22(3):613–620, July 2003.

[MATUSIK ET AL. 2002] Wojciech Matusik, Hanspeter Pfister, Addy Ngan, Paul Beardsley, Remo Ziegler, and Leonard McMillan. Image-based 3D photography using opacity hulls. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 21(3):427–437, July 2002.

[MITCHELL AND NETRAVALI 1988] Don P. Mitchell and Arun N. Netravali. Reconstruction filters in computer-graphics. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 221–228. ACM, 1988.

[MONTES AND UREÑA 2012] Rosana Montes and Carlos Ureña. An overview of BRDF models. Technical report, University of Granada, 2012.

[MUSETH ET AL. 2002] Ken Museth, David E. Breen, Ross T. Whitaker, and Alan H. Barr. Level set surface editing operators. *ACM Transactions on Graphics*, 21:330–338, July 2002.

[MÜLLER ET AL. 2006] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 25(3):614–623, July 2006.

[MÜLLER ET AL. 2007] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of facades. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 26(3), July 2007.

[NAYAR ET AL. 2006] Shree K. Nayar, Gurunandan Krishnan, Michael D. Grossberg, and Ramesh Raskar. Fast separation of direct and global components of a scene using high frequency illumination. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 25(3):935–944, July 2006.

[NEALEN ET AL. 2007] Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. FiberMesh: Designing freeform surfaces with 3D curves. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 26(3), 2007.

[NEUBERT ET AL. 2011] Boris Neubert, Sören Pirk, Oliver Deussen, and Carsten Dachsbacher. Improved model- and view-dependent pruning of large botanical scenes. *Computer Graphics Forum*, 30:1708–1718, 2011.

[NEWCOMBE ET AL. 2011] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality*, ISMAR '11, pages 127–136, Washington, DC, USA, 2011. IEEE Computer Society.

[NG ET AL. 2005] Ren Ng, Marc Levoy, Mathieu Brédif, Gene Duval, Mark Horowitz, and Pat Hanrahan. Light field photography with a hand-held plenoptic camera. Technical report, Stanford University Computer Science, 2005.

[NGAN ET AL. 2005] Addy Ngan, Frédo Durand, and Wojciech Matusik. Experimental analysis of BRDF models. In *Proceedings of the 16th Eurographics Conference on Rendering Techniques*, EGSR '05, pages 117–126. Eurographics Association, 2005.

[NGAN ET AL. 2006] Addy Ngan, Frédo Durand, and Wojciech Matusik. Image-driven navigation of analytical BRDF models. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, EGSR '06, pages 399–407. Eurographics Association, 2006.

[NICODEMUS 1965] Fred E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Applied Optics*, 4(7):767, 1965.

[NIELSEN ET AL. 2015] Jannik Boll Nielsen, Henrik Wann Jensen, and Ravi Ramamoorthi. On optimal, minimal BRDF sampling for reflectance acquisition. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 34(6):186:1–186:11, October 2015.

[OBERT ET AL. 2008] Juraj Obert, Jaroslav Křivánek, Fabio Pellacini, Daniel Sýkora, and Sumanta N. Pattanaik. iCheat: A representation for artistic control of indirect cinematic lighting. *Computer Graphics Forum*, 27(4):1217–1223, 2008.

[OBERT ET AL. 2010] Juraj Obert, Fabio Pellacini, and Sumanta N. Pattanaik. Visibility editing for all-frequency shadow design. *Computer Graphics Forum*, 29(4):1441–1449, 2010.

[OKABE ET AL. 2007] Makoto Okabe, Yasuyuki Matsushita, Li Shen, and Takeo Igarashi. Illumination brush: Interactive design of all-frequency lighting. In *Proceedings of Pacific Graphics*, pages 171–180, 2007.

[OSHER AND FEDKIW 2002] Stanley J. Osher and Ronald P. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 1st edition, October 2002.

[OSHER AND SETHIAN 1988] Stanley J. Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton–Jacobi formulations. *Journal of Computational Physics*, 79:12–49, November 1988.

[O'TOOLE AND KUTULAKOS 2010] Matthew O'Toole and Kiriakos N. Kutulakos. Optical computing for fast light transport analysis. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 29(6):164:1–164:12, December 2010.

[O'TOOLE ET AL. 2012] Matthew O'Toole, Ramesh Raskar, and Kiriakos N. Kutulakos. Primal-dual coding to probe light transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 31(4):39:1–39:11, July 2012.

[PARIS AND DURAND 2007] Sylvain Paris and Frédo Durand. A topological approach to hierarchical segmentation using mean shift. *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR'07)*, 2007.

[PARISH AND MÜLLER 2001] Yoav Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 301–308, New York, NY, USA, 2001. ACM.

[PARKER ET AL. 1998] Steven Parker, Peter Shirley, and Brian Smits. Single sample soft shadows. Technical Report UUCS-98-019, Computer Science Department, University of Utah, 1998.

[PELLACINI ET AL. 2002] Fabio Pellacini, Parag Tole, and Donald P. Greenberg. A user interface for interactive cinematic shadow design. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 21(3):563–566, 2002.

[PELLACINI ET AL. 2007] Fabio Pellacini, Frank Battaglia, Keith Morley, and Adam Finkelstein. Lighting with paint. *ACM Transactions on Graphics*, 26(2):9, 2007.

[PELLACINI 2010] Fabio Pellacini. envyLight: an interface for editing natural illumination. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 29(4):34:1–34:8, 2010.

[PERLIN AND HOFFERT 1989] Ken Perlin and Eric Hoffert. Hypertexture. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, pages 253–262, 1989.

[PERLIN 1985] Ken Perlin. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, pages 287–296, 1985.

[PERLIN 2002] Ken Perlin. Improving noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 681–682, 2002.

[PHARR ET AL. 2016] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory To Implementation*. Morgan Kaufmann Publishers, 3rd edition, 2016.

[PHONG 1975] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[POLLEFEYS AND GOOL 2002] Marc Pollefeys and Luc Van Gool. From images to 3D models. *Communications of the ACM*, 45(7):50–55, July 2002.

[POULIN AND FOURNIER 1992] Pierre Poulin and Alain Fournier. Lights from highlights and shadows. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics*, pages 31–38, 1992.

[PRÉVOST ET AL. 2013] Romain Prévost, Emily Whiting, Sylvain Lefebvre, and Olga Sorkine-Hornung. Make it stand: Balancing shapes for 3D fabrication. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 32(4):81:1–81:10, July 2013.

[PRUSINKIEWICZ AND LINDENMAYER 1990] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, NY, USA, 1990.

[RAMAMOORTHI AND HANRAHAN 2001] Ravi Ramamoorthi and Pat Hanrahan. A signal-processing framework for inverse rendering. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 117–128, 2001.

[RAYLEIGH 1900] John William Strutt, 3rd Baron Rayleigh. On the law of reciprocity in diffuse reflexion. *Philosophical Magazine*, 49:324–325, 1900.

[REDDY ET AL. 2012] Dikpal Reddy, Ravi Ramamoorthi, and Brian Curless. Frequency-space decomposition and acquisition of light transport under spatially varying illumination. In *Proceedings of the 12th European Conference on Computer Vision*, pages 596–610, 2012.

[RICHARD ET AL. 1999] Valery Adzhiev Richard, Richard Cartwright, Eric Fausett, Anatoli Ossipov, Er Pasko, and Vladimir Savchenko. Hyperfun project: a framework for collaborative multidimensional f-rep modeling. In *Proceedings of Implicit Surfaces '99, Eurographics/ACM SIGGRAPH Workshop*, pages 59–69, 1999.

[RINEAU AND YVINEC 2010] Laurent Rineau and Mariette Yvinec. 3D surface mesh generation. In *CGAL User and Reference Manual*. CGAL Editorial Board, 3.6 edition, 2010.

[RITSCHEL ET AL. 2008] Tobias Ritschel, Kaleigh Smith, Matthias Ihrke, Thorsten Grosch, Karol Myszkowski, and Hans-Peter Seidel. 3D unsharp masking for scene coherent enhancement. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 27(3), 2008.

[RITSCHEL ET AL. 2009] Tobias Ritschel, Makoto Okabe, Thorsten Thormählen, and Hans-Peter Seidel. Interactive reflection editing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 28(5):129:1–129:7, 2009.

[RITSCHEL ET AL. 2010] Tobias Ritschel, Thorsten Thormählen, Carsten Dachsbacher, Jan Kautz, and Hans-Peter Seidel. Interactive on-surface signal deformation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 29(4):36:1–36:8, 2010.

[RITSCHEL ET AL. 2012] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. The State of the Art in Interactive Global Illumination. *Computer Graphics Forum*, 31(1):160–188, 2012.

[RUSHMEIER AND TORRANCE 1990] Holly E. Rushmeier and Kenneth E. Torrance. Extending the radiosity method to include specularly reflecting and translucent materials. *ACM Transactions on Graphics*, 9(1):1–27, January 1990.

[SCHMIDT AND UMETANI 2014] Ryan Schmidt and Nobuyuki Umetani. Branching support structures for 3D printing. *ACM SIGGRAPH 2014 Talks Program*, 2014.

[SCHMIDT ET AL. 2005a] Ryan Schmidt, Brian Wyvill, and Eric Galin. Interactive implicit modeling with hierarchical spatial caching. In *Shape Modeling International 2005*, pages 104–113. IEEE, 2005.

[SCHMIDT ET AL. 2005b] Ryan Schmidt, Brian Wyvill, Mário Costa Sousa, and Joaquim Jorge. Shapeshop: Sketch-based solid modeling with blobtrees. In *Proceedings of SBIM 2005*, pages 53–62, 2005.

[SCHMIDT ET AL. 2013] Thorsten-Walther Schmidt, Jan Novak, Johannes Meng, Anton Kaplanyan, Tim Reiner, Derek Nowrouzezahrai, and Carsten Dachsbacher. Path-space manipulation of physically-based light transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 32(4), 2013.

[SCHMIDT ET AL. 2014] Thorsten-Walther Schmidt, Fabio Pellacini, Derek Nowrouzezahrai, Wojciech Jarosz, and Carsten Dachsbacher. State of the art in artistic editing of appearance, lighting, and material. In *Eurographics 2014 State of the Art Reports / extended journal version for Computer Graphics Forum*, Strasbourg, France, 2014. Eurographics Association.

[SCHOENEMAN ET AL. 1993] Chris Schoeneman, Julie Dorsey, Brian Smits, James Arvo, and Donald Greenberg. Painting with light. *SIGGRAPH '93*, pages 143–146, 1993.

[SCHWÄRZLER ET AL. 2013] Michael Schwärzler, Christian Luksch, Daniel Scherzer, and Michael Wimmer. Fast percentage closer soft shadows using temporal coherence. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, pages 79–86, New York, NY, USA, 2013. ACM.

[SEDERBERG 1985] Thomas W. Sederberg. Piecewise algebraic surface patches. *Computer Aided Geometric Design*, 2(1–3):53–59, 1985.

[SEITZ AND DYER 1999] Steven M. Seitz and Charles R. Dyer. Photorealistic scene reconstruction by voxel coloring. *International Journal of Computer Vision*, 35(2):151–173, 1999.

[SHIRLEY ET AL. 2009] Peter Shirley, Steve Marschner, and Michael Ashikhmin. *Fundamentals of Computer Graphics*. Taylor & Francis, 2009.

[SILLION AND PUECH 1994] François X. Sillion and Claude Puech. *Radiosity and Global Illumination*. Morgan Kaufmann Publishers, San Francisco, 1994.

[SILVA ET AL. 2011] Samuel Silva, Beatriz Sousa Santos, and Joaquim Madeira. Using color in visualization: A survey. *Computers and Graphics*, 35(2):320–333, 2011.

[SIMONS AND BEAN 2001] Ronald H. Simons and Arthur R. Bean. *Lighting Engineering: Applied Calculations*. Architectural Press, 2001.

[SIMONS ET AL. 2016] Gerard Simons, Marco Ament, Sebastian Herholz, Carsten Dachsbacher, Martin Eisemann, and Elmar Eisemann. An interactive information visualization approach to physically-based rendering. In *Proceedings of Vision, Modeling & Visualization*, 2016.

[SINGH AND EUGENE 1998] Karan Singh and Fiu Eugene. Wires: a geometric deformation technique. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 405–414, New York, NY, USA, 1998. ACM.

[SINHA ET AL. 2012] Sudipta Sinha, Johannes Kopf, Michael Goesele, Daniel Scharstein, and Richard Szeliski. Image-based rendering for scenes with reflections. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 31(4), 2012.

[SLABAUGH ET AL. 2001] Greg Slabaugh, Bruce Culbertson, Tom Malzbender, and Ron Schafer. A survey of methods for volumetric scene reconstruction from photographs. In *Proceedings of the 2001 Eurographics Conference on Volume Graphics*, VG'01, pages 81–101, 2001.

[SMITH 1984] Alvy Ray Smith. Plants, fractals, and formal languages. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 1–10, New York, NY, USA, 1984. ACM.

[SMITS ET AL. 1994] Brian Smits, James Arvo, and Donald Greenberg. A clustering algorithm for radiosity in complex environments. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 435–442, 1994.

[SPENCER ET AL. 2015] Ben Spencer, Mark W. Jones, and Iksoo Lim. A visualization tool used to develop new photon mapping techniques. *Computer Graphics Forum*, 34(1):127–140, 2015.

[STAVA ET AL. 2012] Ondrej Stava, Juraj Vanek, Bedrich Benes, Nathan Carr, and Radomír Měch. Stress relief: improving structural strength of 3D printable objects. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 31(4):48:1–48:11, 2012.

[SUGIHARA ET AL. 2010] Masamichi Sugihara, Brian Wyvill, and Ryan Schmidt. WarpCurves: A tool for explicit manipulation of implicit surfaces. *Computers & Graphics (Proceedings of Shape Modeling International)*, 34(3):282–291, 2010.

[TAO ET AL. 2013] Michael Tao, Sunil Hadap, Jitendra Malik, and Ravi Ramamoorthi. Depth from combining defocus and correspondence using light-field cameras. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2013.

[TAO ET AL. 2014] Michael W Tao, Ting-Chun Wang, Jitendra Malik, and Ravi Ramamoorthi. Depth estimation for glossy surfaces with light-field cameras. In *Proceedings of the IEEE European Conference on Computer Vision Workshops (ECCVW)*, 2014.

[TAO ET AL. 2015] Michael Tao, Jong-Chyi Su, Ting-Chun Wang, Jitendra Malik, and Ravi Ramamoorthi. Depth estimation and specular removal for glossy surfaces using point and line consistency with light-field cameras. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2015.

[TAPPEN ET AL. 2005] Marshall F. Tappen, William T. Freeman, and Edward H. Adelson. Recovering intrinsic images from a single image. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(9):1459–1472, 2005.

[TELEA AND JALBA 2011] Alexandru Telea and Andrei Jalba. Voxel-based assessment of printability of 3D shapes. In *Proceedings of the international conference on Mathematical morphology and its applications to image and signal processing (ISMM)*, pages 393–404, 2011.

[THOMPSON AND CRAWFORD 1995] David C. Thompson and Richard H. Crawford. Optimizing part quality with orientation. In *Solid Freeform Fabrication Symposium Proceedings*, 1995.

[TURK AND O'BRIEN 1999] Greg Turk and James F. O'Brien. Shape transformation using variational implicit functions. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 335–342, 1999.

[TURK AND O'BRIEN 2002] Greg Turk and James F. O'Brien. Modelling with implicit surfaces that interpolate. *ACM Transactions on Graphics*, 21:855–873, October 2002.

[UMETANI AND SCHMIDT 2013] Nobuyuki Umetani and Ryan Schmidt. Cross-sectional structual analysis for 3D printing optimization, 2013. SIGGRAPH Asia Technical Brief.

[VANEK ET AL. 2014a] Juraj Vanek, Jorge A. Garcia Galicia, and Bedrich Benes. Clever support: Efficient support structure generation for digital fabrication. *Computer Graphics Forum*, 33(5):117–125, 2014.

[VANEK ET AL. 2014b] Juraj Vanek, Jorge A. Garcia Galicia, Bedrich Benes, Radomír Měch, Nathan Carr, Ondrej Stava, and Gavin Miller. Packmerger: a 3D print volume optimizer. *Computer Graphics Forum*, 33(6):322–332, 2014.

[VEACH AND GUIBAS 1997] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76, 1997.

[VELTEN ET AL. 2013] Andreas Velten, Di Wu, Adrian Jarabo, Belen Masia, Christopher Barsi, Chinmaya Joshi, Everett Lawson, Moungi Bawendi, Diego Gutierrez, and Ramesh Raskar. Femto-photography: Capturing and visualizing the propagation of light. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 32(4):44:1–44:8, July 2013.

[VIDIMČE ET AL. 2013] Kiril Vidimče, Szu-Po Wang, Jonathan Ragan-Kelley, and Wojciech Matusik. Openfab: A programmable pipeline for multi-material fabrication. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 32:136:1–136:12, 2013.

[WANG ET AL. 2013] Weiming Wang, Tuanfeng Wang, Zhouwang Yang, Ligang Liu, Xin Tong, Weihua Tong, Jiansong Deng, Falai Chen, and Xiuping Liu. Cost-effective printing of 3D objects with skin-frame structures. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 32(6):177:1–177:10, 2013.

[WANG ET AL. 2015] Ting-Chun Wang, Alexei Efros, and Ravi Ramamoorthi. Occlusion-aware depth estimation using light-field cameras. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2015.

[WANG ET AL. 2017] Ting-Chun Wang, Manmohan Chandraker, Alexei Efros, and Ravi Ramamoorthi. SVBRDF-invariant shape and reflectance estimation from light-field cameras. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2017.

[WARD 1992] Gregory J. Ward. Measuring and modeling anisotropic reflection. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 265–272, New York, NY, USA, 1992. ACM.

[WEISS 2001] Yair Weiss. Deriving intrinsic images from image sequences. In *Proceedings of the 8th IEEE International Conference on Computer Vision (ICCV)*, volume 2, pages 68–75, 2001.

[WESTIN ET AL. 2004] Stephen H. Westin, Hongsong Li, and Kenneth E. Torrance. A comparison of four BRDF models. *Eurographics Symposium on Rendering*, 2004.

[WHELAN ET AL. 2012] Thomas Whelan, John McDonald, Michael Kaess, Maurice Fallon, Hordur Johannsson, and John J. Leonard. Kintinuous: Spatially extended KinectFusion. In *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, Pittsburgh, PA, July 2012.

[WHELAN ET AL. 2015a] Thomas Whelan, Michael Kaess, Hordur Johannsson, Maurice Fallon, John J. Leonard, and John McDonald. Real-time large-scale dense RGB-D SLAM with volumetric fusion. *International Journal of Robotics Research*, 34(4–5):598–626, April 2015.

[WHELAN ET AL. 2015b] Thomas Whelan, Stefan Leutenegger, Renato Salas Moreno, Ben Glocker, and Andrew Davison. ElasticFusion: dense SLAM without a pose graph. In *Proceedings of Robotics: Science and Systems*, Rome, Italy, July 2015.

[WHELAN ET AL. 2016] Thomas Whelan, Renato F. Salas-Moreno, Ben Glocker, Andrew J. Davison, and Stefan Leutenegger. ElasticFusion: real-time dense SLAM and light source estimation. *International Journal of Robotics Research*, 35(14):1697–1716, 2016.

[WHITTED 1979] Turner Whitted. An improved illumination model for shaded display. *Proceedings of the 6th annual conference on Computer graphics and interactive techniques (SIGGRAPH '79)*, pages 343–349, 1979.

[WILLIAMS 1983] Lance Williams. Pyramidal parametrics. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '83, pages 1–11, New York, NY, USA, 1983. ACM.

[WONKA ET AL. 2000] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 71–82, London, UK, 2000. Springer-Verlag.

[WONKA ET AL. 2003] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 22(3):669–677, July 2003.

[WYVILL ET AL. 1986] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.

[WYVILL ET AL. 1987] Geoff Wyvill, Brian Wyvill, and Craig McPheeters. Solid texturing of soft objects. *Computer Graphics 1987: Proceedings of CG International '87*, pages 129–141, 1987.

[WYVILL ET AL. 1999] Brian Wyvill, Eric Galin, and Andrew Guy. Extending The CSG Tree. Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. *Computer Graphics Forum*, 18(2):149–158, June 1999.

[XIE ET AL. 2015] Yue Xie, Weiwei Xu, Yin Yang, Xiaohu Guo, and Kun Zhou. Agile structural analysis for fabrication-aware shape editing. *Computer Aided Geometric Design*, 35–36:163–179, 2015.

[XU ET AL. 2016] Zexiang Xu, Jannik Boll Nielsen, Jiyang Yu, Henrik Wann Jensen, and Ravi Ramamoorthi. Minimal BRDF sampling for two-shot near-field reflectance acquisition. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 35(6):188:1–188:12, 2016.

[ZELEZNIK ET AL. 1996] Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. SKETCH: an interface for sketching 3D scenes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 163–170, 1996.

[ZHANG ET AL. 2015] Xiaoting Zhang, Xinyi Le, Athina Panotopoulou, Emily Whiting, and Charlie Wang. Perceptual models of preference in 3D printing direction. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 2015.

[ZHOU ET AL. 2013a] Qingnan Zhou, Julian Panetta, and Denis Zorin. Worst-case structural analysis. *ACM Transactions on Graphics*, 32(4):137:1–137:12, 2013.

[ZHOU ET AL. 2013b] Shizhe Zhou, Anass Lasram, and Sylvain Lefebvre. By-example synthesis of curvilinear structured patterns. *Computer Graphics Forum (Proceedings of Eurographics)*, 32(2), 2013.

[ZIRR ET AL. 2015] Tobias Zirr, Marco Ament, and Carsten Dachsbacher. Visualization of coherent structures of light transport. *Computer Graphics Forum (Proceedings of EuroVis)*, 34(3), May 2015.

[ZWICKER ET AL. 2015] Matthias Zwicker, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler, and Sung-Eui Yoon. Recent advances in adaptive sampling and reconstruction for monte carlo rendering. *Computer Graphics Forum*, 34(2):667–681, May 2015.