
Soft Shadows, Curved Surfaces and Perceptual Sensitivity: Advanced Methods for Improving Realism in Real-time Rendering

**Weiche Schatten, gekrümmte Flächen
und Wahrnehmungsempfindlichkeit:
Moderne Methoden zur Realismus-
steigerung beim Echtzeitrendern**

Der Technischen Fakultät der
Universität Erlangen-Nürnberg
zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Michael Schwarz

Erlangen — 2009

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung: 30. 04. 2009

Tag der Promotion: 27. 07. 2009

Dekan: Prof. Dr.-Ing. Johannes Huber

Berichterstatter: Prof. Dr.-Ing. Marc Stamminger
Prof. Dr. techn. Michael Wimmer

Abstract

Computer-generated images have become ubiquitous and an important tool in numerous domains. Rendering them at real-time rates allows the interactive exploration of virtual scenes, which is essential for many applications like computer games. Thanks to the wide-spread availability and rapid evolution of graphics hardware offering huge computational power, such a real-time image synthesis is possible for increasingly complex scenes and effects. Concurrently, demands on visual quality are continuously rising, and the desire for realistic appearance is intensifying. However, achieving this goal poses many challenges and comprises a wide range of aspects.

This thesis addresses three significant of these issues and provides new solutions which enhance the realism attainable at real-time rates with graphics hardware. First, soft shadows are covered. These offer valuable visual cues and significantly contribute to realism. We introduce a novel approach for determining light occlusion, which, in particular, correctly handles overlapping shadow casters. Moreover, methods for concentrating computational efforts on relevant light blockers, new advanced occluder approximations, and a scheme for smoothly varying shadow quality to locally adapt rendering costs are presented.

The second focus is on curved surface primitives, whose adoption allows maintaining visual smoothness, as encountered with real-world shapes, irrespective of view and zoom. We provide a comprehensive overview and describe several new contributions for rendering such surfaces via adaptive tessellation. Most notably, a novel, flexible framework is proposed which enables efficiently running all involved major steps entirely on graphics hardware.

Visual perception and its limited sensitivity are at the center of the third topic treated. They play an important role because images are eventually produced to be viewed by a human. We present a graphics-hardware-based approach for rapidly computing a perceptually motivated image metric which predicts tolerable pixel-value deviations, enabling its on-the-fly use during rendering, for instance to exploit scene-level visual masking for guiding level-of-detail selection. Moreover, a novel perceptually-motivated predictor for the perceptibility of visual popping artifacts is introduced and evaluated in a user study.

Kurzzusammenfassung

Computergenerierte Bilder sind in zahlreichen Gebieten allgegenwärtig und ein wichtiges Instrument geworden. Ihre Erzeugung in Echtzeitgeschwindigkeit ermöglicht die interaktive Erkundung virtueller Szenen, was wesentlich für eine Vielzahl von Anwendungen, wie etwa Computerspiele, ist. Dank der weiten Verfügbarkeit und schnellen Entwicklung von großer Rechenleistung bietender Grafikhardware ist solch eine Echtzeitbildsynthese für zunehmend komplexere Szenen und Effekte möglich. Gleichzeitig steigen die Ansprüche an die visuelle Qualität kontinuierlich und der Wunsch nach einem realistischen Aussehen wird stärker. Das Erreichen dieses Ziels wirft jedoch viele Herausforderungen auf und umfaßt eine Vielfalt an Gesichtspunkten.

Diese Arbeit behandelt drei bedeutende dieser Probleme und bietet neue Lösungen, die den in Echtzeit mittels Grafikhardware erzielbaren Realismus erhöhen. Als erstes werden weiche Schatten behandelt. Diese liefern wichtige visuelle Hinweise und tragen wesentlich zu einer realistischen Erscheinung bei. Ein neuartiger Ansatz für die Bestimmung der Lichtverdeckung wird vorgestellt, der insbesondere mit Schatten werfenden Objekten, die sich überlappen, korrekt verfährt. Des weiteren werden Methoden zur Fokussierung der Rechenbemühungen auf relevante das Licht versperrende Objekte, neue fortgeschrittene Approximationen für verdeckende Objekte sowie ein Verfahren, das die Schattenqualität zur lokalen Anpassung der Renderkosten visuell stufenlos variiert, präsentiert.

Der zweite Schwerpunkt liegt auf gekrümmten Flächenprimitiven, deren Einsatz es ermöglicht, sichtbare Glattheit, wie sie bei Formen der realen Welt anzutreffen ist, unabhängig von der konkreten Ansicht aufrechtzuerhalten. Ein umfangreicher Überblick wird gegeben und mehrere neue Beiträge zum Rendern solcher Flächen mittels adaptiver Tessellierung beschrieben. Insbesondere wird ein neuartiges, flexibles Rahmenwerk vorgestellt, das die effiziente Ausführung aller wichtigen dazugehörigen Schritte vollständig auf der Grafikhardware ermöglicht.

Die visuelle Wahrnehmung und ihre eingeschränkte Empfindlichkeit stehen im Zentrum des dritten betrachteten Themengebiets. Sie spielen eine wichtige Rolle, da Bilder letztendlich erzeugt werden, um durch einen Menschen betrachtet zu werden. Ein die Grafikhardware verwendender Ansatz zur schnellen Berechnung einer wahrnehmungsbasierten Metrik, die tolerierbare Pixelwertabweichungen vorhersagt, wird vorgestellt. Dieser ermöglicht ihren fliegenden Einsatz während der Bildsynthese, beispielsweise um szenenweite visuelle Maskierungseffekte bei der Detailstufenauswahl auszunutzen. Darüber hinaus wird ein neuartiges wahrnehmungsbasiertes Verfahren für die Vorhersage der Wahrnehmbarkeit von visuellen Popping-Artefakten eingeführt und mittels einer Nutzerstudie evaluiert.

Contents

| | |
|-----------------|----------|
| Abstract | v |
|-----------------|----------|

| | |
|--|----------|
| 1 Introduction | 1 |
| 1.1 Contributions | 2 |
| 1.2 Outline | 3 |
| 2 Real-time rendering | 5 |
| 2.1 Rendering pipeline | 5 |
| 2.2 Graphics hardware | 7 |
| 2.3 GPGPU and compute APIs | 8 |
| 2.4 Level-of-detail approaches | 10 |
| 2.5 Kinds of realism | 11 |

I Real-time soft shadows

| | |
|--|-----------|
| 3 Overview of real-time soft shadows | 15 |
| 3.1 Soft shadows | 15 |
| 3.2 Real-time approaches | 19 |
| 3.2.1 Image-based approaches | 20 |
| 3.2.2 Geometry-based approaches | 23 |
| 3.2.3 Hybrid approaches | 24 |
| 3.2.4 Approaches for low-frequency environmental lights | 25 |
| 3.2.5 Discussion | 25 |
| 3.3 Quasi-interactive approaches for accurate soft shadows | 26 |
| 4 Soft shadow mapping with occluder backprojection | 29 |
| 4.1 Basic approach | 29 |
| 4.2 Visibility determination with occlusion bitmasks | 33 |
| 4.2.1 Occlusion bitmasks | 33 |
| 4.2.2 Advanced applications | 34 |
| 4.2.3 Discussion | 37 |
| 4.3 Acceleration structures | 39 |
| 4.3.1 Multi-scale shadow map | 39 |
| 4.3.2 Hybrid Y shadow map | 43 |
| 4.3.3 Discussion | 44 |

| | | |
|--|---|------------|
| 4.4 | Occluder approximations | 45 |
| 4.4.1 | Microquads | 46 |
| 4.4.2 | Approximate occlusion bitmasks for microquads | 49 |
| 4.4.3 | Microtris | 49 |
| 4.4.4 | Exact occlusion bitmasks for microquads and microtris | 49 |
| 4.4.5 | Discussion | 52 |
| 4.5 | Coarser occluder approximations | 55 |
| 4.5.1 | Microrects as a generalization of micropatches | 56 |
| 4.5.2 | Construction of microrects at coarser levels | 57 |
| 4.5.3 | Discussion of microrects | 59 |
| 4.5.4 | Biasing problems | 61 |
| 4.6 | Visibility interpolation for multisample support | 62 |
| 4.7 | Results and conclusion | 64 |
| 5 | Level of quality for soft shadows | 69 |
| 5.1 | Possible approaches | 69 |
| 5.1.1 | Multiple algorithms producing different quality | 70 |
| 5.1.2 | Geometric occluder LOD | 70 |
| 5.1.3 | Sparse visibility sampling | 71 |
| 5.1.4 | Intrinsic algorithm parameters | 72 |
| 5.2 | Smooth quality variation for soft shadow mapping | 72 |
| 5.2.1 | Approach | 74 |
| 5.2.2 | Discussion | 75 |
| 5.2.3 | Results | 76 |
| II Rendering of curved surfaces | | |
| 6 | Fundamentals of curved surfaces | 81 |
| 6.1 | Bézier surfaces | 82 |
| 6.1.1 | Bézier patches | 83 |
| 6.1.2 | Bézier triangles | 85 |
| 6.1.3 | PN triangles | 86 |
| 6.2 | Spline surfaces | 90 |
| 6.3 | Subdivision surfaces | 93 |
| 6.3.1 | Direct evaluation | 96 |
| 6.3.2 | Approximation using Bézier surfaces | 97 |
| 6.4 | Algebraic surfaces | 99 |
| 6.4.1 | GPU-based raycasting | 100 |
| 6.5 | Rendering approaches | 101 |
| 6.5.1 | Tessellation | 101 |
| 6.5.2 | Raycasting | 103 |
| 6.5.3 | Direct rasterization | 107 |
| 7 | Adaptive tessellation | 111 |
| 7.1 | Objectives | 111 |
| 7.2 | Recursive refinement | 114 |
| 7.2.1 | Refinement criteria | 114 |

| | | |
|-------|---|-----|
| 7.2.2 | Refining a rectangular domain | 115 |
| 7.2.3 | Refining a triangular domain | 118 |
| 7.2.4 | Locally handling cracks | 119 |
| 7.2.5 | GPU-based implementations | 120 |
| 7.2.6 | Discussion | 122 |
| 7.3 | Tessellation patterns | 123 |
| 7.3.1 | Patterns for rectangular domains | 123 |
| 7.3.2 | Patterns for triangular domains | 126 |
| 7.3.3 | Transition regions | 129 |
| 7.3.4 | Fast pattern generation | 129 |
| 7.4 | Determining tessellation factors | 131 |
| 7.4.1 | Bounding screen-space triangle sizes | 132 |
| 7.4.2 | Bounding the approximation error | 133 |
| 7.4.3 | Rational Bézier patches | 135 |
| 7.4.4 | Bézier triangles | 136 |
| 7.4.5 | PN triangles | 137 |
| 7.5 | Rendering of refinement patterns | 138 |
| 7.5.1 | GPU-based methods | 139 |
| 7.5.2 | Connection patterns for dyadic tessellation of PN triangle meshes . . . | 140 |
| 7.5.3 | Instanced rendering | 146 |
| 7.5.4 | Number of refinement patterns | 147 |
| 7.5.5 | Direct hardware support | 149 |
| 7.6 | Patch-parallel on-the-fly tessellation | 150 |
| 7.6.1 | CudaTess framework for adaptive tessellation | 151 |
| 7.6.2 | Example: bicubic rational Bézier patches | 153 |
| 7.6.3 | Example: PN triangles | 158 |
| 7.6.4 | Discussion | 160 |
| 7.6.5 | Comparison to rendering refinement patterns | 162 |

III Perception-aware rendering

| | | |
|----------|---|------------|
| 8 | Fundamentals of human visual perception | 167 |
| 8.1 | Human perception and psychophysics | 167 |
| 8.2 | Human visual system | 168 |
| 8.3 | Color and color appearance | 172 |
| 8.4 | Visual attention | 173 |
| 9 | Perceptually motivated rendering | 175 |
| 9.1 | Building blocks for computational models | 175 |
| 9.1.1 | Contrast sensitivity functions | 176 |
| 9.1.2 | Visual masking | 180 |
| 9.1.3 | Multi-channel decomposition | 181 |
| 9.2 | Vision models and visual difference metrics | 183 |
| 9.3 | Overview of perceptually motivated applications | 185 |
| 9.4 | Real-time threshold maps | 187 |
| 9.5 | Interactive perceptual rendering pipeline | 188 |

| | | |
|-----------|---|------------|
| 9.6 | Problems in applying perceptual results | 190 |
| 10 | Visual popping | 193 |
| 10.1 | Popping and related treatment approaches | 193 |
| 10.2 | Aspects of perceiving popping | 195 |
| 10.3 | Perceptually motivated popping predictor | 196 |
| 10.3.1 | Overview | 197 |
| 10.3.2 | Discussion | 197 |
| 10.3.3 | Spatio-velocity color vision model | 198 |
| 10.3.4 | Popping regions | 201 |
| 10.3.5 | Examples | 202 |
| 10.4 | User study | 203 |
| 10.4.1 | Experiment I: direct evaluation with simple object | 204 |
| 10.4.2 | Experiment II: indirect evaluation with real-world examples | 207 |
| 10.4.3 | Conclusion | 209 |
| 11 | Conclusion | 211 |
| | Bibliography | 213 |

CHAPTER 1

Introduction

Computer graphics is concerned with generating synthetic images, which are nowadays routinely employed in a plethora of domains as diverse as movies and medical imaging. An important branch aims at rendering such images instantly at real-time rates, thus essentially producing a video stream while being viewed. This real-time rendering allows the interactive exploration of data sets and virtual scenes, and is at the heart of computer games, which have evolved to a huge mass market.

With the maturing of the field, demands on visual quality are increasing. In particular, many applications strive for a high degree of realism, often with the ultimate (long-term) goal of delivering photo-realistic images in real time. This comprises a multitude of aspects like the desire to incorporate global effects such as shadows and interreflecting light. Further objectives include supporting illumination from large sources like the sky, modeling real-world materials like car paint, and employing more detailed and visually smooth geometric objects.

This aim for realism is also reflected in the growing complexity of employed scene assets as well as their origin. For example, real objects are scanned to acquire geometric detail, the motion of actors is captured to obtain animation data for articulated characters, and a vast collection of photographs of a surface sample is used to derive a material description. Another factor which raises expectations of higher detail and quality is the increasing resolution and size of display devices, facilitating paying attention to fine detail.

Simultaneously, driven by the demands especially from the games market, dedicated graphics hardware has evolved tremendously and become a mainstream computer component. Its computational power and memory bandwidth now typically far exceed the ones offered by a standard CPU. Utilizing highly parallel graphics hardware is thus crucial for achieving satisfactory frame rates in real-time rendering, but it is also challenging. In particular, for high performance an appropriate formulation of the task must be devised which, among others, yields a reasonable degree of data parallelism.

However, even with graphics hardware fast response times usually necessitate approximations and quality restrictions. Since the image is synthesized for a human viewer, it is hence expedient to leverage human visual perception for improving rendering efficiency, ideally providing only exactly as much detail as can actually be perceived. Moreover, accounting for perception helps avoiding disturbing visual artifacts which hamper realistic appearance.

Improving realism in real-time rendering has thus many diverse facets, and is the focus of extensive active research. This thesis contributes to these efforts and introduces new approaches and techniques for physically plausible soft shadows, the resolution-independent rendering of curved surfaces, as well as for taking human visual perception into consideration.

1.1 Contributions

In this dissertation, three selected topics are investigated which play an important role in enhancing the realism that is achievable at real-time frame rates. The first covered aspect are soft shadows cast from an area light source. They provide valuable visual cues and are typically essential for a realistic appearance. Building on the general technique of deriving occluder approximations from a shadow map and backprojecting them onto the light source to determine light visibility, we introduce methods which improve on visual quality and performance, and thus on attainable realism. Our contributions include

- a new approach for visibility determination, occlusion bitmasks, that offers a robust solution to the occluder fusion problem and hence to a main obstacle to high quality in previous algorithms (Sec. 4.2),
- efficient acceleration structures for concentrating computations on relevant occluders and detecting completely lit and entirely shadowed points (Sec. 4.3),
- a new breed of occluder approximations extracted from a shadow map with several favorable features, like implicitly avoiding light leaks (Sec. 4.4),
- a new type of occluder approximation which raises reconstruction quality and hence accuracy at coarser resolution levels (Sec. 4.5),
- a visibility interpolation method for cheaply supporting soft shadows in multisample rendering (Sec. 4.6), and
- a practical scheme for smoothly varying soft shadow quality in screen space, which allows adapting rendering efforts according to visual importance (Sec. 5.2).

The second focus of this thesis is on rendering curved surfaces. These are essential to replicate the visual smoothness of many shapes encountered in the real world independent of view and zoom factor. In particular, their use avoids often-observed visibly piecewise-linear silhouettes, which may easily destroy realism. We largely employ adaptive tessellation for rendering, improving on attainable performance. Our main contributions are

- a comprehensive overview of adaptive tessellation approaches, concentrating on those utilizing graphics hardware, which also provides critical reflections on them (Chapter 7),
- a novel, flexible, patch-parallel framework for adaptive tessellation, termed CudaTess, which runs all major steps, like deriving consistent tessellation factors, determining and evaluating surface sample points, and creating the tessellation topology, completely on the graphics processing unit (GPU), and which more generally provides an efficient solution for dynamically generating varying amounts of geometry purely on the GPU (Sec. 7.6),
- a new method for rapidly determining the tessellation factor for a PN triangle such that the approximation error stays small (Sec. 7.4.5),
- a new approach for rendering PN triangle meshes using domain pre-tessellations, which closes gaps inherent to the PN triangle refinement of a base mesh (Sec. 7.5.2), as well as
- a pixel-shader-based approach for raycasting PN triangles (Sec. 6.5.2).

The third field addressed by this thesis is human visual perception and its limited sensitivity. On the one hand, rendering efficiency can be improved by exploiting this restricted detection ability to avoid spending effort on producing detail which is eventually invisible to the user, thus enabling a higher realism for a given time budget. On the other hand, perceptual results can be leveraged to quantify visual artifacts which may impact the perceived degree of realism. Covering both aspects, we present

- an approach to rapidly compute a threshold map, a perceptually motivated image metric predicting tolerable per-pixel deviations (Sec. 9.4), which enables exploiting sensitivity-reducing effects like visual masking on a scene level for controlling the employed geometric level of detail (Sec. 9.5),
- a perceptually motivated predictor for estimating whether popping artifacts occur when switching between two levels of detail of an object, which incorporates a spatio-velocity color vision model and aggregates model output to meaningful popping regions (Sec. 10.3), and
- a user study conducted to evaluate the predictor, showing encouraging results (Sec. 10.4).

1.2 Outline

Our treatment of three distinct topics is also reflected in the structure of this thesis, where one part is dedicated to each of them. Note that since our contributions are not entirely unrelated to other research efforts and previous solutions, we decided to present them in their respective context instead of devoting a single chapter exclusively to each of our new methods.

At first, we review some background on real-time rendering in Chapter 2, introducing terms and concepts for the remainder of the dissertation.

Subsequently, soft shadows are covered in Part I. Initially, Chapter 3 provides an overview of soft shadows in general and of existing approaches for rendering (approximations of) them in real time. Chapter 4 then discusses the adopted general soft shadow algorithm and presents our diverse contributions concerning visibility determination, acceleration structures, occluder approximations and cheap multisample support. After that, level-of-quality approaches for soft shadows are considered in Chapter 5 and a practical scheme for smooth quality variation is described.

The second part is concerned with rendering curved surfaces. Chapter 6 reviews important representatives of according primitives, paying special regard to issues related to real-time rendering, and gives an overview of rendering approaches, during which our raycasting method for PN triangles is introduced. Subsequently, a comprehensive treatment of adaptive tessellation techniques is provided in Chapter 7. In particular, our numerous contributions are presented, including a novel, patch-parallel framework which executes all significant steps on the graphics hardware.

Part III focuses on exploiting and accounting for human visual perception during rendering. At first, Chapter 8 covers some fundamental background on perception. Chapter 9 then discusses utilizing core characteristics of visual perception for rendering. It introduces our real-time threshold maps and describes their application to controlling the employed geometric level of detail of objects. By contrast, Chapter 10 is dedicated to the perception of popping artifacts and presents our predictor and its evaluation within a user study.

Finally, Chapter 11 closes this thesis with a brief conclusion.

CHAPTER 2

Real-time rendering

Throughout this thesis, we are primarily concerned with the domain of real-time rendering, the creation of images at rates rapid enough that their instantaneous display induces the notion of a continuous image sequence. Since achieving a high performance is crucial, this naturally involves employing dedicated graphics hardware for carrying out the majority of computations. Moreover, we strive for compatibility with the standard graphics rendering pipeline in the design of our methods to facilitate integration with existing real-time solutions.

In this chapter, we briefly review some related core topics, introducing terms and concepts utilized in the following parts. Note that a basic knowledge of real-time rendering is assumed, nevertheless. A good resource is the book by Akenine-Möller et al. [8]; further background on computer graphics in general is provided by Shirley et al. [356], for instance.

At first, we give a short overview of the rendering pipeline, before covering recent graphics hardware as well as its use for tasks beyond pure rendering, like general data-parallel computations. Subsequently, approaches for adapting the level of detail are discussed. Finally, we briefly elaborate on the pursued goal of realism.

2.1 Rendering pipeline

The rendering pipeline is central to real-time rendering. It decomposes the image synthesis task into several logical stages. On a high level, the application provides geometric data as input and adapts the stages accordingly to yield the desired behavior. The pipeline then processes the items, determines pixel colors and outputs the result into the frame buffer, which typically comprises a color buffer and a depth buffer for visible-surface determination.

Thanks to the graphics APIs OpenGL and Direct3D, the pipeline is essentially standardized and a close mapping of the stages to graphics hardware units exist. Note that as APIs and hardware evolve, different pipeline versions emerge, for instance by introducing new stages. In the following, we focus on the graphics pipeline as defined by Direct3D 10 [38], which is realized by all current graphics hardware. Fig. 2.1 provides an overview. Two kinds of stages can be distinguished. Whereas fixed-function stages lack flexibility and only offer limited control by means of a few state parameters, programmable stages may be freely customized via user-provided programs. These so-called shaders can access constants, sample textures and read from arbitrary buffer locations when processing their input.

The flow of data through the pipeline into a certain output buffer is initiated by a draw call of the employed API. At first, the *input assembly* stage takes the application-specified geometric data, which is typically provided in vertex and index buffers, and prepares it for the subsequent

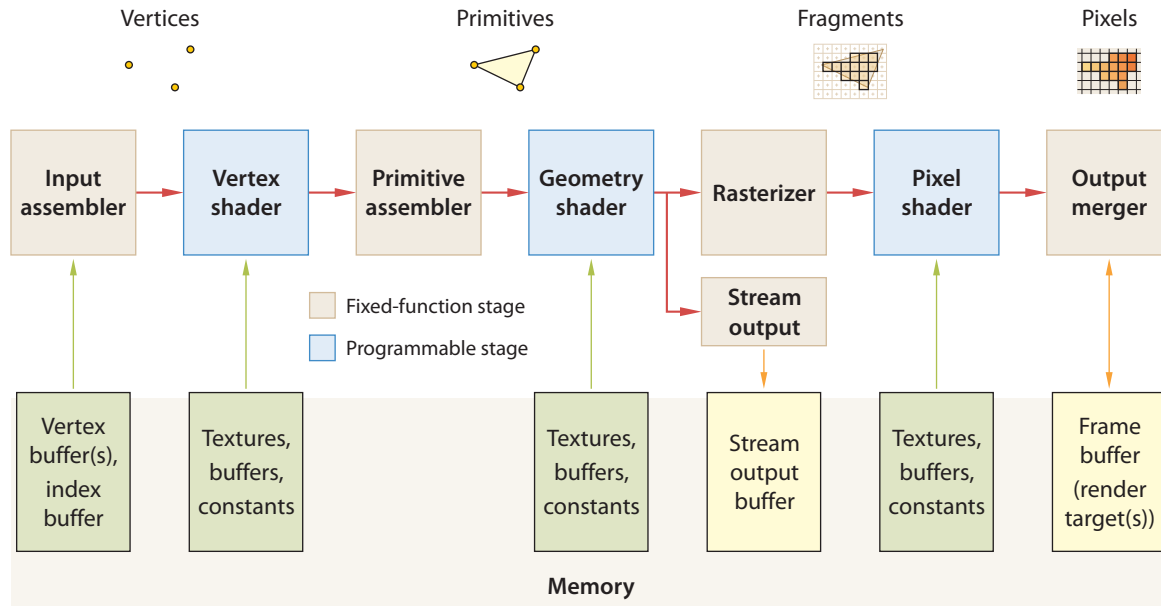


Figure 2.1 Overview of the rendering pipeline.

steps. Different *input primitive topologies* are supported, like point lists and lists and strips of lines or triangles, but also adjacency-augmented ones which further comprise vertices of directly adjoining primitives.

Subsequently, each input vertex is processed independently in the *vertex shader* stage. The invoked shader instance performs per-vertex computations, like the transformation of vertex position into clip space (cf. Fig. 2.2). After that, the processed vertices are combined to individual primitives according to the specified topology by the *primitive assembler*.

These primitives are then (optionally) fed into the *geometry shader* stage. For each primitive, a separate shader instance is launched, which has access to all vertices of the primitive. It is intended to perform primitive-wide computations as, for example, deriving the face normal, and may even account for directly adjoining primitives in case an adjacency-augmented primitive topology is employed. Note that a geometry shader can use either point lists, line strips or triangle strips as output primitive topology, independent from its input. Consequently, the primitive type may be altered in this stage, and more than one output primitive can be emitted per input primitive. It is also possible to effectively discard an input primitive by outputting no primitives.

The resulting primitives can be streamed out to memory via the *stream output* stage. They are recorded as a set of individual primitives in the stream output buffer, which may be used as input for a further pass through the pipeline. Typically, however, the primitives arriving from the geometry shader stage are processed by the *rasterizer*. Each primitive is clipped, transformed to screen space (cf. Fig. 2.2) and then rasterized by generating a fragment for each covered pixel,¹ interpolating vertex attributes accordingly.

Subsequently, a *pixel shader* (also referred to as fragment shader) is run for each fragment. It determines the final color or pixel value and optionally adapts the depth value. The resulting pixel data is then combined with the existing frame buffer content by the *output merger*. This

¹In case of triangles (and no multisampling), a pixel is considered to be covered if its center is inside the triangle.

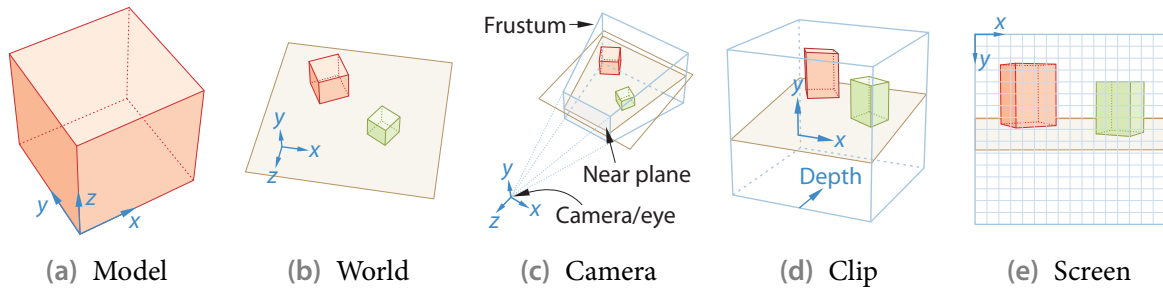


Figure 2.2 Coordinate spaces typically involved in rendering. Each object is initially defined in its own local *model space*, and placed within the scene by a subsequent transformation into *world space*. After that, a transformation into *camera space* (often also called *eye space*) is performed, such that the camera (eye) is placed at the origin and looks down the negative (or sometimes the positive) z axis. In a next step, the scene is subjected to a perspective (or alternatively an orthographic) projection into homogeneous *clip space*, which maps the defined viewing volume (a frustum or an axis-aligned box, respectively) into a cube, clipping away everything outside. After a dehomogenizing divide, resulting in *normalized device coordinates*, a final transformation into *screen space* is carried out. Here, (x, y) coordinates specify pixel location, while z encodes depth.

typically involves a depth test to resolve surface visibility, where the new depth value is compared against the one currently stored in the depth buffer to determine whether the pixel should be updated. Apart from simply replacing the old pixel value, more complex blending operations are supported. Note that the frame buffer may not necessarily contain a single color buffer but can comprise multiple *render targets* (up to eight), each consisting of up to four channels (of possibly 32-bit floating-point precision).

Normally, some optimizations are incorporated into realizations of the pipeline to increase efficiency. For instance, to avoid running the pixel shader for fragments which eventually fail the depth test, this check is already performed in the rasterizer, discarding the fragment unless it passes. In addition to this so-called *early-z test*, often *z-culling* is performed. To this end, a coarser-resolution version of the depth buffer is maintained (possibly at reduced bit depth), storing a conservative depth bound for each screen tile. Before producing any fragments within a tile, the rasterizer then first tests the primitive against the tile's depth bound. Note that these optimizations are only possible if the pixel shader doesn't modify the fragment's depth.

2.2 Graphics hardware

The whole rendering pipeline as detailed in the last section is realized by current consumer graphics hardware like NVIDIA's GeForce GTX 280 or AMD's ATI Radeon HD 4870. Owing to the demands of the games market and facilitated by its huge volume, they feature high computational power and large memory bandwidths at affordable prices and are nowadays an integral part of basically all (entertainment) computers.

The central component of graphics hardware is the graphics processing unit (GPU) [120, 282], which is responsible for carrying out the computations. It features dedicated special-purpose units for handling the fixed-function pipeline stages, like a rasterizer or raster operation processors (ROPs), which implement the output merger. Moreover, extensive compu-

tational resources exist for shader execution. These are actually shared by all shader stages in the prevalent unified shader architecture. By dynamically assigning vertex, geometry and pixel shader instances depending on the actual workload, a high utilization can thus be achieved. Note that since each vertex is processed independently from the other vertices but the same vertex shader is executed for all of them, they may be treated in a data-parallel fashion. The same holds for primitives and fragments. As reflected in their architecture, GPUs heavily exploit this massive parallelism to achieve high performance.

Recent NVIDIA GPUs [218], for instance, feature a large number of scalar arithmetic logic units (ALUs), called *streaming-processors* (SPs), each capable of 32-bit floating-point and integer operations. These are arranged in groups of eight, each constituting a core referred to as *streaming multiprocessor* (SM). The ALUs of a core are run in SIMD (single instruction, multiple data) fashion, that is, while each ALU operates on different data (e.g. a different vertex), they all execute the same instruction at a time. The hardware further directly supports lightweight threads, with one thread being spawned per vertex, primitive or fragment, executing the respective shader program on a single ALU. Groups of 32 threads, referred to as *warps*, are run in a time-sliced way. This multithreading enables a high throughput, keeping ALUs utilized despite thread stalls and thus hiding memory access latency. Note that if the control flow within a shader diverges for simultaneously executed threads, the individual control paths are processed sequentially, thus reducing the effective parallelism and utilization. Apart from the ALUs, a core also comprises special-function units for evaluating transcendental functions and 16 KB of so-called *shared memory*. Cores are further grouped to clusters, each additionally featuring eight texture units for (tri-/bi-/linearly) filtered texture accesses.

As a concrete example, NVIDIA's GeForce GTX 280 has 240 SPs, organized in 10 clusters of three SMs each, which offer a peak computational power of 933 Gflops. The typically 1024 MB of on-board graphics memory are accessed with a bandwidth of 142 GB/s. As this far exceeds the performance achievable on a CPU, which is designed for rapid execution of a few primarily sequential tasks, GPUs are increasingly employed for speeding up data-parallel workloads beyond rendering (see Sec. 2.3).

A notable departure from the current situation, where the architecture of graphics hardware closely matches a fixed graphics pipeline, is pursued by Intel's upcoming Larrabee chip [349]. It is essentially a many-core processor with wide SIMD units, where fixed-function stages like the rasterizer are implemented in software. Consequently, the whole pipeline becomes programmable and may be adapted flexibly to fit an application's particular needs.

2.3 GPGPU and compute APIs

Given the huge computational power and high memory bandwidth offered by GPUs, it has become attractive and desirable to harness these capabilities for data-parallel tasks other than pure pipeline-based rendering. Initially, standard graphics APIs like OpenGL were utilized to this end, forcing the programmer to express the task at hand as a rendering problem. One standard technique that emerged is to store input data items into a texture, capture output data items in the frame buffer, and render an appropriately sized quad, where the triggered pixel shader computes each output item independently, using the input data. The output may then serve as input for the next step. Such GPGPU (general-purpose computation on GPUs) efforts led to the development of several solutions, like more complex GPU-suited data structures, which are useful for rendering, too, enabling advanced computer graphics algorithms. A related

survey, covering major techniques and example applications, is provided by Owens et al. [284].

For general data-parallel computing [282], however, having to access the computational resources via a graphics API is cumbersome, incurs a certain overhead and may even be unnecessarily restrictive. To address this issue, improve ease of use, increase achievable performance and open up new markets, GPU vendors devised dedicated compute APIs. They no longer follow the rendering pipeline but provide a hardware abstraction which exposes more details about and additional capabilities of the GPUs compared to graphics APIs. In particular, memory accesses are more flexible, permitting to write to multiple arbitrary memory locations. On the other hand, some available graphics-specific hardware units like the rasterizer are not exposed and hence cannot be utilized. Note that limited interaction with graphics APIs is possible by mapping buffer resources from a graphics API context into a compute program's address space. Unfortunately, at least with current drivers, the associated overhead can be considerable and sometimes hence constitutes a severe obstacle to high overall performance.

Currently employed compute APIs like ATI Stream (comprising CAL and Brook+) [6], which evolved from ATI's CTM [5], and NVIDIA's CUDA [273] are targeted specifically to the hardware of the respective vendor, preventing written programs to run on GPUs from competitors. This situation is alleviated by the upcoming industry standard OpenCL [185], as well as the introduction of compute shaders in Direct3D 11 [52]. In the latter case, shaders are designed to interact smoothly with the standard rendering pipeline, using the same language (HLSL) and resource types as the shaders in the programmable pipeline stages. Moreover, the pixel shader stage is extended appropriately by allowing random-access memory writes for preparing input to a compute shader.

In compute APIs, a *kernel* (or compute shader) encapsulates a certain computational task. It is applied to a set of *work items* in parallel, launching one thread for each item. The items are organized and indexed according to a multi-dimensional computation domain. This is further structured into *work groups* (also called thread groups or blocks), where all items in a group can cooperate via shared group-local memory and group-wide synchronization operations. Note that this inter-item communication possibility is very powerful and not exposed by graphics APIs.

CUDA

Since CUDA is employed in Sec. 7.6, we provide some more specific detail [270]. A work group is called *block* in CUDA and all threads of a block are executed on the same SM. Each block is further split into warps, with all 32 threads of a warp running in lock-step and hence automatically being in sync. The 16 KB of fast SM-local shared memory are split among all blocks concurrently assigned to a SM. A block's fraction of this memory can be accessed by all threads of the block, allowing communicating data between them. It is often employed as fast data cache, where common data is first brought in from global memory collectively by several threads which then operate on it. Multiple blocks are further structured in a *grid*, defining the computation domain.

Regarding memory accesses, each thread can perform uncached reads from and writes to arbitrary locations in global memory. For maximum throughput, however, the concurrent accesses within a (half-)warp should allow of coalescing. It is also possible to perform cached reads by resorting to textures. To exchange data with an OpenGL context, buffer objects can be mapped to CUDA's global memory. Finally, threads may access their block's part of shared memory for intra-block cooperation.

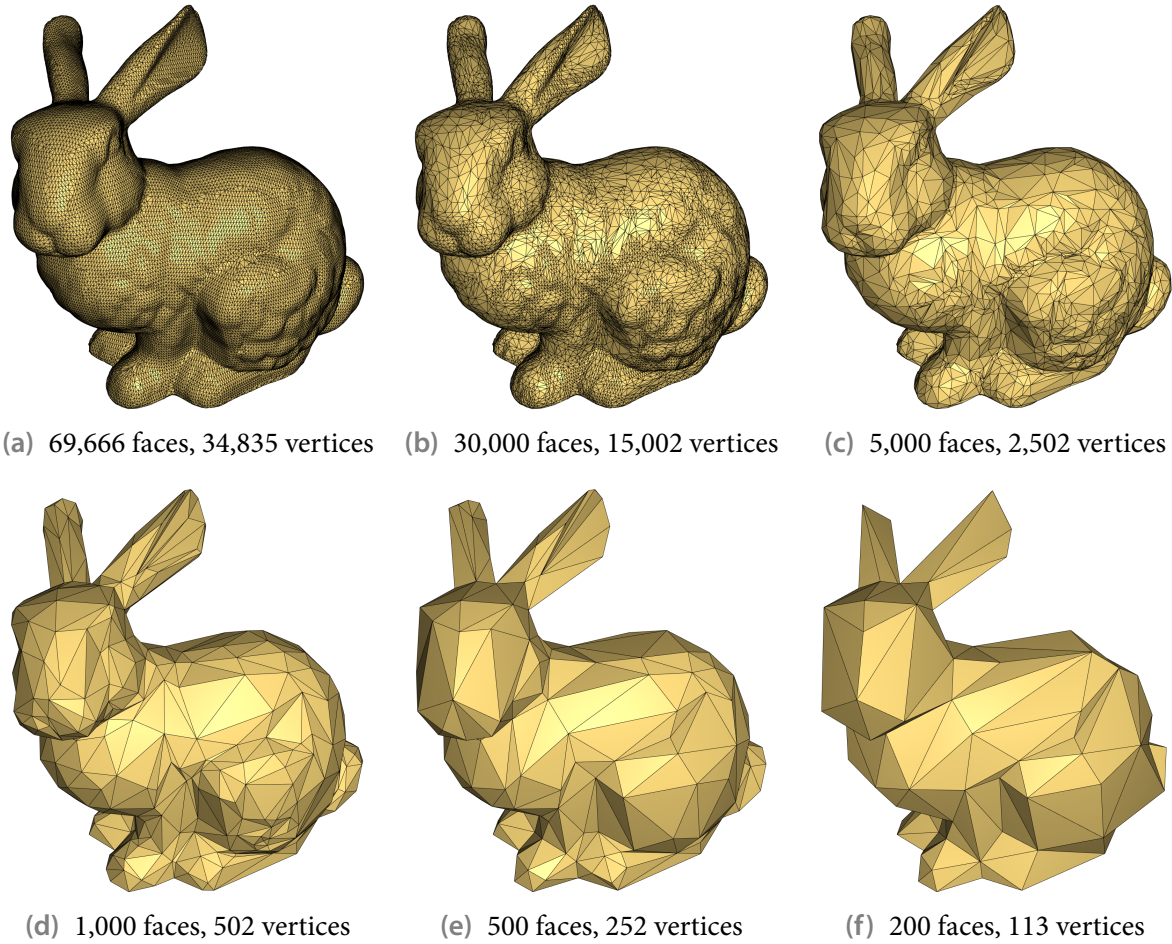


Figure 2.3 Example of view-independent discrete geometric LOD.

2.4 Level-of-detail approaches

An important and often employed technique in real-time rendering is adapting the *level of detail* (LOD) of scene elements to meet a certain budget, like limited memory resources and especially maximum rendering time per frame. While it hence allows trading visual quality for performance, it is also an effective means to avoid rendering excessive detail which cannot be discerned anyway. Most notably, the geometric complexity and hence the triangle count of objects are commonly reduced with increasing distance and decreasing screen-space extent. This helps avoiding eventually rendering triangles of pixel- or even sub-pixel-size and thus positively affects efficiency and attainable frame rate without (severely) compromising quality.

Normally, three different LOD types are distinguished. Whereas *discrete LOD* provides a small number of different element versions of varying complexity, *continuous LOD* offers a whole continuum of LODs, which enables fine changes in overall element complexity. A special kind of continuous LOD is *view-dependent LOD* (or *adaptive LOD*), where local, selective adaptations of detail are possible, e.g. at the silhouettes or in back-facing parts, allowing to take the actual view into account.

Many LOD methods have been developed which are concerned with the geometric complexity of single objects. Classical discrete LOD [74] employs a few automatically derived or hand-crafted variants of decreasing triangle count for a certain model (see Fig. 2.3). Though

still being widely used, this simple approach lacks flexibility compared to continuous [164] and view-dependent LOD [165, 237, 405] techniques, where mesh complexity can be adapted at a granularity of single triangles. However, since LOD is typically updated on the CPU and graphics hardware excels when processing batches of geometry, operating on a triangle level proves nowadays to be a severe performance bottleneck, unlike the situation when these methods [240] were devised. Consequently, more recent efforts adopt a coarser granularity and work on whole patches (chunks of triangles) [45, 72, 73, 322, 411], but also concentrate mainly on huge models. Furthermore, representations other than triangle meshes are increasingly employed for coarser levels, as for instance voxels [140]. Yoon et al. [410] provide a current review of related LOD approaches. Note that spurred by the growing flexibility of graphics hardware, there is regained interest in fine-grained LOD adaptation with updates being executed entirely on the GPU [168].

Ideally, switching a model's LOD should not be perceivable by the viewer. Otherwise annoying popping artifacts can arise, which hamper realism (see Chapter 10). One popular remedy is to smooth the transition between two LODs, either by geomorphing [164] one triangle mesh into the other, or by rendering both LODs and blending them in image space [139]. A recent variant [329] of the latter approach renders just one (alternating) LOD per frame and combines it with the rendering of the other LOD from the previous frame, typically copying the pixel color from either one of the two LODs, with the selection proportion roughly matching the blend factor.

Furthermore, several LOD schemes have been devised for aspects other than a single object's geometry. Cook et al. [79], for instance, deal with the aggregate detail induced by a large number of elements, like encountered when modeling plants or hairs. Aiming at preserving the aggregate's appearance, in particular its area and contrast, their stochastic approach employs only a subset of appropriately adapted elements for rendering. LOD methods also exist for acquired material data, usually utilizing some multi-resolution representation [76, 243], as well as for complex shaders [276, 277, 290]. Moreover, an image-space LOD approach for shader execution [406] was devised, where expensive shading computations are performed at a lower resolution and the results are subsequently upsampled to full resolution in a discontinuity-respecting way.

One main driving factor for using LOD techniques is reducing (unnecessary) complexity and computational load. However, they can also be useful to improve the visual quality by avoiding or at least reducing aliasing. Note that ideally, a coarser LOD employed for a certain view corresponds to an appropriately filtered version of the finest LOD. But to achieve this, the necessary effort may actually increase compared to just using the finest LOD, conflicting with the goal of reducing costs. Examples include methods for resolution-dependent surface reflectance [154, 371, 372] and approaches for antialiasing shaders [382].

2.5 Kinds of realism

When images are synthesized in computer graphics, often the goal of realism is pursued. However, the notion of realism is rather fuzzy and may depend on the specific context. Addressing this lack of a generally applicable and accepted definition, Ferwerda [124] identifies three varieties of realism:

- *Physical realism* strives for creating an image which when displayed yields the same visual stimulus as the represented real scene, that is, the same spatially varying (spectral) power

distribution should be induced. This naturally requires an accurate definition of the scene in terms of physical quantities and involves computationally expensive, physically-based light transport simulations, making this kind of realism unsuitable for real-time rendering. Moreover, (commodity) display devices are typically not capable of evoking the desired power distribution.

- Usually, an image is produced to be viewed by a human user, who captures and further processes it with the human visual system (see Chapter 8). *Photo-realism* accounts for this and aims at inducing the same visual response as the depicted real scene. Consequently, characteristics like the trichromatic encoding of spectral power distributions (colors), the adaptation to varying illumination levels and colors, or the limited detectability of stimulus deviations can be exploited. Note that an image which is considered indistinguishable from a photograph of the scene may not necessarily be regarded as photo-realistic according to Ferwerda's definition, not least because a photograph can feature manifestations of the camera's optical system, like distortions and lens flare.
- Focusing on performing a visual task, *functional realism* merely seeks to provide the same visual information as the real scene to accomplish this job. Note that the degree of realism is highly dependent on the task at hand and hence the purpose of the image. In particular, visual abstractions and illustrative rendering techniques can prove advantageous for attaining a high functional realism.

Realistic real-time rendering typically targets some looser sort of photo-realism, where an image should only appear as if it is photo-realistic but doesn't actually need to be photo-realistic. Most notably, deviations from a photo-realistic reference version which go unnoticed when the image is viewed by itself but are clearly visible when the reference is available for comparison are usually deemed unproblematic. For instance, inconsistencies between the shape of a shadow and the casting object or the light's shape, distorted reflections on bumpy surfaces as well as inaccurate refractions often remain undetected. This tolerance of the human visual system is frequently exploited in real-time rendering as it allows cheaply approximating some effects without affecting the perceived degree of realism.

To quantify to which degree deviations from a reference are possible while still inducing the same scene appearance, some psychophysical experiments were conducted. Ramanarayanan et al. [306] investigated how much blurring and warping of illumination maps is acceptable depending on geometry bumpiness and material glossiness. They also studied how the composition of a complex aggregate of many objects, where perception focuses on the collection as a whole instead of on individual objects, can be varied without affecting the appearance with respect to numerosity, variety and arrangement [305].

In this thesis, when striving for realistic real-time rendering, we ultimately aim for photo-realism. Note, however, that we only focus on some aspects at a time. For instance, when covering soft shadows in Part I, we try to accurately account for the influence of light-blocking scene objects and the light's extent on the shadow's shape but completely ignore the contribution of indirect illumination resulting from reflected light.

PART I

Real-time soft shadows

CHAPTER 3

Overview of real-time soft shadows

Soft shadows are an important global effect that can significantly enhance the realism of rendered scenes. It is hence highly desirable to support at least reasonable approximations of them at real-time frame rates. In this chapter, we first discuss soft shadows in general, touching their character, importance and resulting challenges. Subsequently, we provide a brief overview of the multitude of approaches that have been devised for real-time rendering of (approximate) soft shadows. The particular technique on which we mainly focused our work—soft shadow mapping with occluder reconstruction and backprojection—is treated extensively in the following chapter, detailing our contributions. Finally, we review approaches which produce fairly accurate soft shadows but currently only achieve at best interactive frame rates for more complex scenes.

3.1 Soft shadows

When physical plausibility and global light transport are taken into account while rendering a scene, the final color of a pixel is typically derived from the radiance¹ $L(\mathbf{p}, \boldsymbol{\omega}_{\text{cam}}(\mathbf{p}))$ at the corresponding visible scene point \mathbf{p} exiting in the direction towards the camera $\boldsymbol{\omega}_{\text{cam}}(\mathbf{p})$. The radiance values may be determined by solving the rendering equation [175]

$$L(\mathbf{p}, \boldsymbol{\omega}) = L_e(\mathbf{p}, \boldsymbol{\omega}) + \int_S f_r(\mathbf{p}, \boldsymbol{\omega}' \rightarrow \boldsymbol{\omega}) \frac{\langle \mathbf{n}_{\mathbf{p}}, -\boldsymbol{\omega}' \rangle_{\text{sat}} \langle \mathbf{n}_{\mathbf{x}}, \boldsymbol{\omega}' \rangle_{\text{sat}}}{\|\mathbf{p} - \mathbf{x}\|^2} V(\mathbf{p}, \mathbf{x}) L(\mathbf{x}, \boldsymbol{\omega}') dA_{\mathbf{x}}, \quad (3.1)$$

where we assume opaque objects for simplicity and hence employ the saturating dot product $\langle \mathbf{a}, \mathbf{b} \rangle_{\text{sat}} = \max(0, \langle \mathbf{a}, \mathbf{b} \rangle)$ to ensure non-negative cosine values. S denotes the set of all surfaces in the scene, $\boldsymbol{\omega}' = \boldsymbol{\omega}'(\mathbf{p}, \mathbf{x})$ is the (normalized) direction from point \mathbf{x} to \mathbf{p} , and $\mathbf{n}_{\mathbf{y}}$ designates the surface normal at point \mathbf{y} . $L_e(\mathbf{p}, \boldsymbol{\omega})$ is the radiance emitted from \mathbf{p} while the integral collects the radiance incident on \mathbf{p} that gets reflected in direction $\boldsymbol{\omega}$, with the BRDF f_r describing the reflectance distribution. The binary visibility term $V(\mathbf{p}, \mathbf{x})$ equals one if \mathbf{x} is visible from \mathbf{p} , i.e. the line segment connecting them is not intersected by any other scene elements; otherwise $V(\mathbf{p}, \mathbf{x}) = 0$.

For reasons of clarity, we henceforth focus on a single light source and consider only direct lighting, thus ignoring interreflections. Then, (3.1) simplifies to an integral over the set of all

¹Radiance L is a radiometric quantity measuring the light energy per unit time and unit solid angle and unit projected area.

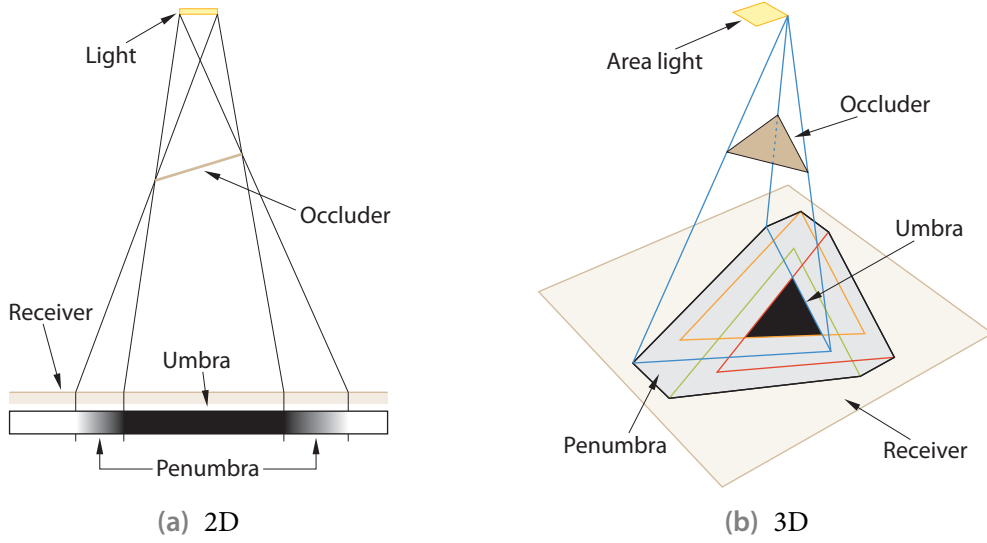


Figure 3.1 Geometry of soft shadow cast by a single linear/triangular occluder from a linear/rectangular light source onto a planar receiver.

light source surface points \mathcal{L} , with $L(\mathbf{x}, \boldsymbol{\omega}') = L_e(\mathbf{x}, \boldsymbol{\omega}')$ for all $\mathbf{x} \in \mathcal{L}$:

$$L(\mathbf{p}, \boldsymbol{\omega}) = \int_{\mathcal{L}} f_r(\mathbf{p}, \boldsymbol{\omega}' \rightarrow \boldsymbol{\omega}) \frac{\langle \mathbf{n}_{\mathbf{p}}, -\boldsymbol{\omega}' \rangle_{\text{sat}} \langle \mathbf{n}_{\mathbf{x}}, \boldsymbol{\omega}' \rangle_{\text{sat}}}{\|\mathbf{p} - \mathbf{x}\|^2} V(\mathbf{p}, \mathbf{x}) L_e(\mathbf{x}, \boldsymbol{\omega}') dA_{\mathbf{x}}. \quad (3.2)$$

Shadows result from light being blocked by some occluder. More formally, they occur at scene points \mathbf{p} from which light points \mathbf{x} are occluded by scene objects, i.e. where the set $\mathcal{V}(\mathbf{p}) = \{\mathbf{x} \in \mathcal{L} \mid V(\mathbf{p}, \mathbf{x}) = 0\}$ is non-empty. A major task in deriving shadows is hence determining the visibility relations between the ordinary scene points and the light points.

Sometimes, shadows are further classified as cast shadows and attached shadows [187]. If the surface at a point \mathbf{p} is facing away from a light point \mathbf{x} , no radiance is transferred between them and the resulting shadow is called *attached*. Typically, such cases are easily dealt with because the term $\langle \mathbf{n}_{\mathbf{p}}, -\boldsymbol{\omega}' \rangle_{\text{sat}} \langle \mathbf{n}_{\mathbf{x}}, \boldsymbol{\omega}' \rangle_{\text{sat}}$ evaluates to zero.² On the other hand, if \mathbf{p} is facing towards a light point \mathbf{x} with $V(\mathbf{p}, \mathbf{x}) = 0$, the occurring shadow is said to be *cast* by some scene object (the one which contains the point closest to \mathbf{x} that lies on the line segment connecting \mathbf{p} and \mathbf{x}). In case the shadow-casting object is different from the object to which \mathbf{p} belongs, the shadow is referred to as *extrinsic shadow*. Otherwise an *intrinsic shadow* is formed due to *self-shadowing*.

To further discuss the geometry of shadows, consider the general setup in Fig. 3.1, where a light source is partially blocked from a *receiver* by an *occluder*, that is, the occluder is casting a shadow onto the receiver. Note that in case of self-shadowing, receiver and occluder are the same object. Regions \mathcal{U} where the light source is completely hidden by the occluder and where hence (3.2) yields $L(\mathbf{p}, \boldsymbol{\omega}) = 0$ for all $\mathbf{p} \in \mathcal{U}$ constitute the *umbra*. Points from which only a part of the light source is visible form the *penumbra*. Together, umbra and penumbra regions establish shadows. The remaining points are typically referred to as completely lit.

²Note that surfaces belong to objects of some thickness and hence there is always a point \mathbf{p}' in between \mathbf{p} and \mathbf{x} , causing $V(\mathbf{p}, \mathbf{x}) = 0$. However, since the distance between \mathbf{p}' and \mathbf{p} may be arbitrarily small, numerical precision and robustness problems can lead to wrongly indicating mutual visibility of \mathbf{p} and \mathbf{x} . Therefore, it is advantageous that occlusion is already enforced by the saturating dot products becoming zero.

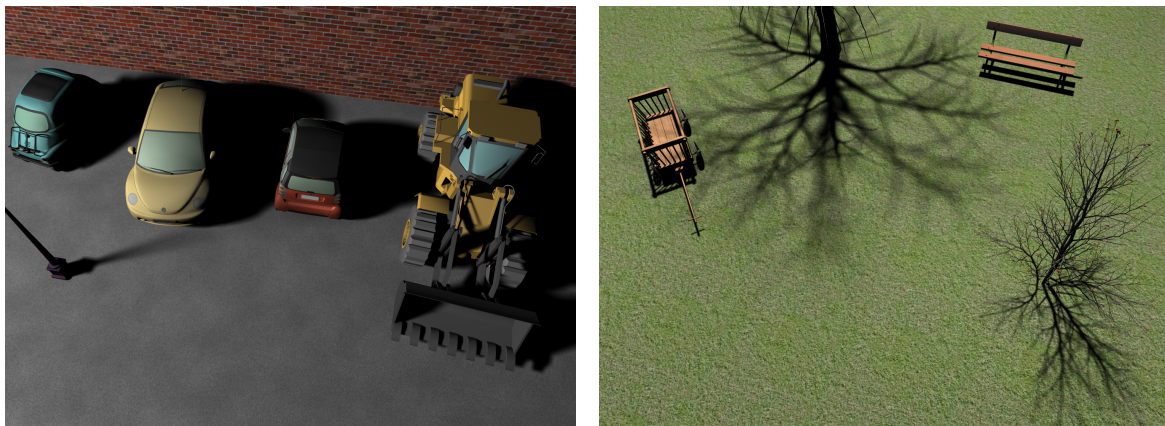


Figure 3.2 Examples of soft shadows. They convey information about the direction of the light source as well as about the spatial relationship of the scene objects. Notice how the shadow cast by the street light hardens towards its pole’s contact with the ground (left). Also note that even though the upper part of the tree is not visible, its shadow allows roughly inferring the tree’s height and branch structure (right).

The size and shape of shadows as well as their partition into umbra regions and penumbrae depend on the spatial extent of the light source. In the limit case of a point light, light visibility is exclusively determined by the visibility of the sole light point and is hence of binary nature, i.e. the light is either completely visible or entirely occluded but cannot be partially blocked. Therefore, each appearing shadow comprises only an umbra but no penumbra. Due to this lack of any intermediate degree of shadowing, such shadows are called *hard shadows*. Although point lights are typically not encountered in reality, they are often employed in real-time rendering because of their simplicity and the resulting savings in computational efforts.³ Most notably, shadow determination only requires computing a single binary point-to-point visibility.

On the other hand, if the light has a real spatial extent (unlike a point), it may not just be completely visible or invisible but also be partially visible. Consequently, occurring shadows feature transition regions of partial illumination, the penumbrae. Because of these shadowing gradients, they are referred to as *soft shadows*. Note that in some settings no visible point exists from which the light source is completely occluded, that is, soft shadows don’t necessarily contain umbra regions. Extended light sources come in several flavors ranging from distant environmental lights defined by irradiance maps to area lights. Typically, we consider only nearby lights of simple shape, like rectangular or spherical light sources, not least because it is often such lights which are responsible for producing visually dominating and distinctly recognizable soft shadows.

Referring to the generic setting in Fig. 3.1 again, straightforward geometric considerations show that as the size of a light source grows, a shadow’s penumbra region increases, while its umbra shrinks and eventually even disappears. The part of the penumbra growing outwards with respect to the umbra at point light size is sometimes called *outer penumbra*, while the

³It is tempting to consider a distant extended light source like the sun as a single point light. In many cases, however, such an approximation negatively impacts realism because the actually cast shadows often would feature small but still noticeable penumbra regions, i.e. they would not appear as completely hard—which contradicts the point light assumption. It is also worth recalling that, as seen from earth, the sun still subtends a solid angle of roughly 0.00006 sr.

portion extending inwards and replacing the umbra is referred to as *inner penumbra*. Furthermore, keeping the light fixed but moving the occluder towards the receiver reduces the overall shadow size and decreases the relative penumbra portion. A direct consequence is that if an occluder touches a receiver, the cast shadow essentially only comprises an umbra at the region of contact but with increasing distance becomes wider and dominated by a growing penumbra, which ultimately may supersede the umbra completely.

Such shadow *hardening on contact*, like with the street light's contact shadows in Fig. 3.2, is an example of the important cues [245] provided by soft shadows and their significance for a realistic appearance. They convey information about the shape of objects, like, for instance, the size of lights, the silhouette of shadow-casting occluders, or the surface geometry of receivers. Moreover, soft shadows offer cues concerning the spatial relationship of objects, eliminating or at least reducing the ambiguity of relative placement. As further demonstrated by the examples in Fig. 3.2, soft shadows are hence crucial for both realism and depth perception.

In typical scenes it often occurs that for a point in shadow the light is blocked by multiple occluders, which may be different objects or distinct parts of the same object. The individual regions of the light source hidden by these occluders can be disjoint or overlapping. In the latter case, the solid angles covered by multiple occluders unite, effectively leading to an *occluder fusion*.

Challenges for real-time rendering

Over time, many algorithms have been devised for rendering both hard and soft shadows. The somewhat outdated surveys by Woo et al. [402] and Hasenfratz et al. [159] provide a comprehensive overview of related methods. More recent developments are briefly reviewed in the next sections. In general, rendering of soft shadows poses many challenges, especially when real-time frame rates are aimed for.

Typically, only direct illumination is considered. Moreover, to save computations, the corresponding lighting equation (3.2) is usually simplified by assuming terms like the BRDF to be constant, evaluating them just for one predominant direction towards the light source. Therefore, only the visibility $V(\mathbf{p}, \mathbf{x})$ and, if non-uniform, textured light sources are to be supported, the light's emitted radiance $L_e(\mathbf{x}, \boldsymbol{\omega}')$ are actually integrated over the light area. Such an approximation is justified because soft shadows directly result from spatially varying light visibility, whereas the ignored changes in the remaining terms usually are visually less important and dominant, especially for diffuse receivers and reasonably sized lights. It is also questionable whether these missed shading variations have a larger impact on realism than indirect illumination, which is completely omitted.

Consequently, central to computing soft shadows is determining the fraction of the light area that is visible from a scene point. However, accurately answering point-to-region visibility queries is expensive for more complex scenes. While graphics hardware has reached a tremendous computational power, in order to obtain real-time performance current algorithms still have to resort to imposing constraints on the scene that enable simplifications and precomputations, or to introducing approximations. This typically incurs some limitations, like restricting the support for dynamic changes, requiring large amounts of memory, and settling for a (hopefully) good estimate of light visibility. For instance, to keep the number of occluders processed per pixel reasonably low, only a subset of the potential occluders may be considered. A related challenge is ensuring temporal coherence for the produced soft shadows, especially if the approximations involve some dynamic sampling.

Since visibility is a non-local problem, information about the scene geometry must be made available to the shaders computing the soft shadows. There are two general approaches to achieve this. On the one hand, an explicit scene representation may be constructed and provided to all fragments. In many cases this is just a shadow map offering a point sampling of some of the occluders, but in principle one may also utilize a spatial structure like a kd-tree filled with the scene triangles or a subset of them. Alternatively, scene information can be distributed implicitly by successively issuing primitives describing the scene geometry such that each causes the visibility data of the affected pixels to be updated accordingly. Examples include wedges corresponding to silhouette edges and polygonal footprints of scene triangles. Typically, each such primitive conservatively covers the influenced pixels, and for each generated fragment the stencil buffer is adjusted or the output of the triggered pixel shader gets blended into some buffer representing the current light visibility. In finding a suitable way to convey global scene information, issues like generation time and required storage space but also ease of occluder processing have to be dealt with. Moreover, employing acceleration structures may be worthwhile if the gained savings exceed the involved building costs.

A major challenge is how to correctly handle occluder fusion because each occluder is processed individually but the regions of the light source hidden by them may overlap. Consequently, computing the visibility integral (potentially weighted by a light texture) for each occluder and then simply compositing the resulting visibility values is merely a fast approximation which generally doesn't yield correct results. By contrast, we have to maintain information about the light regions blocked by occluders, update it for each occluder, and only in a final step compute the visibility integral, accounting for all processed occluders. However, given the constraints of graphics hardware and the aim for real-time frame rates, representing and combining region information on a pixel level is non-trivial. Therefore, only recently robust solutions to occluder fusion have been devised for interactive and real-time rendering. They all essentially follow our approach [333], which, to the best of our knowledge, was the first one published.⁴ It is discussed in more detail in Sec. 4.2.

3.2 Real-time approaches

A large variety of algorithms have been developed for rendering (approximate) soft shadows at real-time frame rates. One major class of them utilizes an image-based representation of the considered occluding scene objects; such approaches are covered in Sec. 3.2.1. By contrast, geometry-based methods employ explicit geometric primitives for each processed occluder to derive the soft shadows (Sec. 3.2.2). Moreover, some hybrid approaches exist (Sec. 3.2.3).

Another line of distinction is between algorithms merely striving for visually plausible soft shadows and those aiming at more physical correctness. While the first class typically makes crude approximations, like extending hard shadows by outer penumbrae, the latter approaches usually project occluders onto the light area to more accurately determine light visibility. It is worth noting that especially such physically plausible algorithms are often the result of adaptations of CPU-based off-line algorithms made possible by increases in the computational power and programmability of GPUs.

Unless otherwise noted, all techniques presented in the following target soft shadows produced by reasonably sized spherical or rectangular area light sources.

⁴A different technique presenting basically the same underlying idea was developed concurrently by Eisemann and Décoret [107] and presented at the same venue as our approach.

3.2.1 Image-based approaches

Image-based approaches to soft shadow rendering typically extend standard shadow mapping [347, 398] for hard shadows. Here, the scene is rendered from a light point \mathbf{x} into a depth map, recording the scene objects visible to \mathbf{x} . This *shadow map* thus captures the occluders closest to the considered light point. To determine whether a point \mathbf{p} is in shadow, it is transformed into shadow map space and compared against the related shadow map entry. If this element's depth is smaller, it is closer to the light point than \mathbf{p} and hence occludes \mathbf{x} from \mathbf{p} .

Note that the sampling of the scene performed by the shadow map differs from that of the view camera, leading to aliasing problems. In particular, fine structures may be missed and silhouettes are easily captured at a too coarse resolution. Moreover, biasing of the depth values recorded in the shadow map is required to avoid *surface acne* artifacts due to incorrect self-shadowing. While many techniques have been developed to address these shortcomings for hard shadows (Lloyd et al. [225] provide an overview), they are beyond our scope, not least because they are often not directly applicable to related soft shadow algorithms.

On the other hand, image-based algorithms offer several advantages. First, current graphics hardware features direct support for both the generation and the query of shadow maps. Second, they typically scale better with scene complexity than geometry-based approaches. Third, such methods can readily deal with geometry altered in the pixel shader stage via alpha masking, pixel kills or depth modifications. Especially the selective discarding of fragments is fundamental to many recent techniques for raycasting surfaces (cf. Secs. 6.4.1 and 6.5.2) and for adapting the silhouette in per-pixel displacement mapping algorithms [68, 278].

Plausible faking by adapting hard shadows

Many early techniques which try to fake soft shadows in a plausible way are based on the idea of starting with the hard shadow obtained for a point light source placed at the center of the extended light and enlarge it outwards to generate a penumbra region [287]. Brabec and Seidel [53] search a shadow map for the nearest texel containing depth information of an occluder and use both the distance to the corresponding texel and the read depth value to estimate a pixel's placement within the found occluder's soft shadow, thus adding outer penumbras and replacing some hard shadows by inner penumbras. In similar work, Kirsch and Döllner [186] move all involved computations to the GPU but become further restricted to handle only inner penumbras. By contrast, Arvo et al. [14] create outer and inner penumbras by applying a modified flood-fill algorithm in screen space to the umbra regions obtained from a shadow map.

All algorithms of this class suffer from several inherent limitations which often lead to clearly visible artifacts. For instance, since the search radius is usually bounded and only the occluders closest in shadow map space are found, relevant occluders might get ignored, limiting the penumbra's extent and causing transition artifacts. Moreover, the umbra regions are often significantly overestimated, especially with techniques only accounting for outer penumbras.

Blurring of hard shadow map results

Another related option is to simply blur hard shadows via *percentage-closer filtering* (PCF) [311]. Here, a scene point is not just compared against its corresponding shadow map entry but to all texels within a certain neighborhood. The binary results are then averaged to yield a percentage value estimating light visibility. Non-extremal values effect penumbra regions, whose extents

depend on the size of the considered PCF kernel relative to the shadow map resolution. To better account for varying penumbra widths, for each fragment, Fernando [122] searches the shadow map for entries closer to the light and utilizes the average depth of the found potential occluders to dynamically compute the kernel size for the filtering step.

While the attained results with these *percentage-closer soft shadows* (PCSS) are often visually pleasing, at least for simple settings, the approach suffers from several shortcomings. For instance, it assumes a single planar occluder parallel to the shadow map's near plane, which is not always a good approximation. Moreover, when processing shadow map samples, occluders are identified by just checking whether a recorded point is closer to the light than the point \mathbf{p} corresponding to the current fragment. Consequently, shadow map entries which are outside the point–light pyramid (with apex \mathbf{p} and the area light as base) and hence don't block the light at all from \mathbf{p} may be wrongly considered as occluders. This can lead to incorrect estimates of the average depth of the relevant occluders, as well as to wrong visibility values—even if the single planar occluder assumption holds. Another major obstacle is that both searching the shadow map for occluders as well as performing percentage-closer filtering requires many shadow map accesses, thus limiting the achievable performance.

The PCF overhead can be significantly reduced by resorting to alternative shadow map representations that allow prefiltering. Note that arbitrary rectangular filter sizes may then easily be supported by constructing a corresponding summed-area table [83, 206]. One class records a concise description per texel of the distribution of depth samples, which can be filtered directly, and then derives light visibility for each fragment by applying statistical estimation formulae. The first such approach were *variance shadow maps* [95], which store the first two moments of linear depth and employ Chebyshev's inequality. However, since this only yields an upper bound on the probability that a point is farther away from the light, and hence on the percentage of shadow map samples farther away, variance shadow maps are prone to *light bleeding* artifacts, where parts of the shadow interior are too bright. Other related methods include *warped variance shadow maps* [207], *approximate cumulative distribution function shadow maps* [145], and *exponential shadow maps* [321].

Alternatively, the depth comparison of shadow map entries with a scene point \mathbf{p} may be approximately decomposed into terms depending only on the shadow map depth values or only on \mathbf{p} , respectively, again enabling prefiltering. *Convolution shadow maps* [12] use a Fourier series expansion, while *exponential shadow maps* [13] employ exponential factors. The same approach can also be applied to derive the average occluder depth [11], thus largely removing the overhead of the occluder search step, too.

Reconstructing and backprojecting occluders

Aiming at physically plausible soft shadows for rectangular area lights, an important group of algorithms employs a shadow map obtained from the light source's center and reconstructs potential occluders in world space from it. These are then backprojected from the currently considered scene point \mathbf{p} onto the light's plane to estimate the visible fraction of the light area. Note that our techniques, further detailed in the next two chapters, belong to this class. The various methods mainly differ in what kind of occluder approximation is employed, how the computations are organized, and how the occluded light area is derived. Among others, these choices directly influence performance, generality, robustness and visual quality.

Many approaches just adopt shadow map texels unprojected into world space as micro-occluders [17, 19, 31, 147]. However, gaps in the occluder reconstruction may emerge, which

lead to visible *light leaks*. This can be alleviated by extending each micro-occluder to its neighbors in texture space [147]. Another remedy suggested is to employ a multi-layered shadow map generated via depth peeling [111, 246], but to only consider for each texel the farthest micro-occluder still closer to the light than \mathbf{p} [30]. Alternatively, one may construct larger occluders by extracting contours from the shadow map [148]. We developed a different kind of micro-occluder [333, 335], which implicitly avoids gaps and is also less prone to surface acne artifacts.

Concerning the order of computation, Atty et al. [19] loop over all micro-occluders and scatter the occlusion caused by them to all affected pixels in a soft shadow map, which finally gets projected onto the scene. While this approach is of high efficiency, it requires a separation of objects into shadow casters and shadow receivers. Other approaches don't suffer from this limitation because they adopt a gathering strategy, where for each processed scene point \mathbf{p} all relevant occluder approximations are extracted from the shadow map and backprojected to derive light visibility. Apart from enumerating all shadow map texels within a search area determined by the intersection of the point-light-area pyramid and the near plane used for shadow map generation [17, 147], one may instead just sample according to a Gaussian Poisson distribution [31] or even perform regular subsampling [30]. To keep the number of processed occluders not actually hiding the light from \mathbf{p} low, the search area can be tightened by means of an acceleration structure [147]. The multi-scale representations devised by us [333, 336] are very effective in this regard, often significantly improving performance. Yet another option for iterating over relevant occluders is to hierarchically traverse a min/max mipmap pyramid of the shadow map [93].

Light visibility is typically computed by summing up the light areas covered by the backprojected occluders. Similarly, the relative solid angles covered by the micro-occluders' bounding spheres may be added up [31]. However, the backprojections and solid angles, respectively, often overlap, causing light occlusion to be overestimated, which in turn can lead to objectionable artifacts. We developed a robust solution to the underlying occlusion fusion problem, namely by tracking the visibility of light sample points with an occlusion bitmask [333]. This not only prevents overlapping artifacts but also enables including occluders from further shadow maps.

Real-time performance may necessitate enforcing an upper bound on the number of micro-occluders considered per fragment. Apart from subsampling [30, 31], one solution is to employ a coarser minimum mipmap level of the shadow map for extracting occluder approximations [147]. Related light-space and screen-space multi-resolution approaches are presented by Guennebaud et al. [148], which, however, suffer from robustness and visual quality problems. We investigated several aspects of having to cope with a restricted time budget, like smooth spatial variation of soft shadow quality and corresponding local adaptation of computational efforts, coarser-level occluder approximations, and cheap support for multisample rendering [336].

Chapter 4 provides more detail about this class of algorithms, commonly referred to as *soft shadow mapping* approaches. The main focus is put on our contributions, including accurate occlusion combination and hence correct occluder fusion handling, effective acceleration structures, as well as improved occluder approximations. These enable high visual quality at real-time frame rates and make soft shadow mapping an attractive candidate for rendering physically plausible soft shadows.

Multi-layered approaches

While a single shadow map only captures the occluders closest to a dedicated light point \mathbf{x} , several approaches exist which take further samples along the corresponding rays emanating from \mathbf{x} into account. One class generates shadow maps from several points on the light source and merges them into a single extended shadow map for the light center. For instance, Agrawala et al. [7] employ layered depth images [352] as representation for their layered attenuation maps, recording samples visible from any of the considered light points, whereas St-Amour et al. [362] adopt deep shadow maps [226], storing occlusion as a function of depth for each texel. Although such methods allow rendering rather accurate soft shadows in real time once the extended shadow map has been created, the generation of this structure is typically only possible at interactive rates, at best. In particular, high quality usually requires considering many light sample points and thus acquiring and incorporating a large number of shadow maps.

A different and cheaper algorithm is suggested by Eisemann and Décorêt [106, 108]. They establish planes parallel to the area light and, within each resulting slice, project the scene geometry away from the light source onto the slice's far plane, recording the covered parts in a (binary) occlusion texture. A convolution of this characteristic function with a suitably scaled light source kernel approximately determines the percentage of light blocked by the occluders in a slice [361]. Exploiting this observation, occlusion textures are first prefiltered and then appropriately combined for each fragment to derive light visibility. While the method is fast and often provides visually pleasing soft shadows, the conversion into planar occluders at a small number of light distances can lead to artifacts. For instance, light occlusion due to blockers within the same slice as the point to be shaded may be missed or underestimated.

3.2.2 Geometry-based approaches

Although image-based approaches have many advantages, like readily supporting versatile geometry rendering techniques beyond triangle meshes, they inherently suffer from limitations due to the involved sampling in shadow map space, resulting in aliasing problems. Moreover, a shadow map typically only captures a subset of the occluders and may provide a poor sampling of surfaces almost normal to the shadow map's near plane.

By contrast, geometry-based approaches utilize an explicit representation of the occluder geometry to derive light visibility. It is thus possible to consider all occluders and avoid aliasing problems. On the downside, such algorithms are typically slower than image-based methods and more susceptible to the occluder fusion problem because not just the blockers closest to the light are processed. Only recently, techniques have been devised that effectively address the latter issue, yielding pretty accurate soft shadows. Unfortunately, these currently don't run at real-time frame rates except for simple scenes. They are briefly covered in Sec. 3.3.

Soft shadow volumes

Several approaches generalize *shadow volumes* [82, 162] developed for hard shadows. In this method, first silhouettes with respect to a point light are identified. Subsequently, a volume is constructed for each silhouette by extruding its edges along the direction away from the light. Noting that a point is in shadow if it is enclosed by at least one volume, an according counter is kept for each pixel (typically in the stencil buffer). Then, the shadow volumes are rendered, with the counter being increased for front-facing faces of the volume and decreased for back-facing ones. While accurate, this algorithm is fill-rate intensive and requires extracting silhouettes.

A corresponding extension for soft shadows was introduced by Assarsson and Akenine-Möller [15, 16]. They determine silhouettes as seen from the center of the light source and render a penumbra wedge for each silhouette edge. The triggered pixel shader projects the edge onto the light source and determines the covered area of the according radial section. These areas are accumulated and used to modulate a visibility buffer initialized by a standard shadow volume pass. The resulting soft shadows are of rather high quality if the following conditions are met. First, the used silhouette edges have to be roughly identical to those seen from other points on the light source. As a direct consequence, soft shadows produced for large area lights are typically not very accurate. Second, it is necessary that the silhouettes' projections onto the light area don't overlap, precluding common setups where occluder fusion occurs.

To somewhat alleviate the latter limitation, Forest et al. [130] suggest keeping track not only of the covered area but also maintaining a bounding box of the occluded regions for each quarter of the light source. This allows detecting potential overlaps and estimating their magnitude, but due to the looseness and coarseness of the approximations it doesn't provide a robust and satisfactory solution to handling overlapping silhouette projections.

3.2.3 Hybrid approaches

Apart from approaches exclusively using either image-based occluder representations or accurate object-space geometric information, respectively, some hybrid algorithms exist which operate in image space but also take explicit object information into account. They typically use a standard shadow map to identify an initial umbra region, and render extra geometry for the silhouette edges into a further texture to handle penumbra regions.

Attenuating geometry attached to silhouettes

Chan and Durand [66] fake soft shadows by augmenting hard shadows with outer penumbrae. They extrude the silhouettes parallel to the shadow map's near plane by a global user-specified amount with quadrilaterals. These so-called *smoothies* are assigned a visibility gradient that increases outwards from the silhouettes. They are rendered from the light's center into a smoothie buffer, which is then used to attenuate pixels, creating the illusion of penumbrae. While the method adapts the penumbra width according to the relative placement of the occluder with respect to the light and the receiver, the radius of the assumed spherical light is ignored. In a similar approach, Wyman and Hansen [404] account for the light size by constructing a cone for each silhouette vertex and a sheet connecting adjacent cones for each silhouette edge. This auxiliary geometry is rendered into a *penumbra map* from the center of the light, again establishing soft-shadow-like gradients emanating from the silhouettes that are used to augment hard shadows.

Since both methods produce only outer penumbrae, the resulting soft shadows often appear too dark. To alleviate this, Cai et al. [59] attach not only outer fins to silhouette edges but also inner fins, and generate separate penumbra maps for them. However, since inner penumbrae are realized by brightening hard shadows according to the inner penumbra map, light leaking problems occur if multiple occluders overlap. This can be alleviated by decomposing the scene into multiple layers and creating layer-specific penumbra maps. Nevertheless, artifacts due to overlapping fins may still occur. Also note that while often visually plausible, adding inner and outer penumbrae is only an approximation.

Using a non-pinhole light camera

A rather different approach is pursued by Mo et al. [253]. They employ a non-pinhole camera positioned at the light center \mathbf{x} to also capture surfaces in the penumbra region normally hidden from \mathbf{x} . This *soft shadow occlusion camera* effectively bends camera rays at depth discontinuities, thus looking around occluder silhouettes. The corresponding distortion of the camera image in the vicinity of silhouettes is described by a distortion map. It is created by extruding silhouette edges parallel to the near plane and rendering the resulting quads into the map. Instead of actually rendering the scene with this camera, the computed distortion is directly employed to derive a point's placement within the penumbra. While interesting, this method only works for relatively small lights and suffers from artifacts if extruded silhouette edges overlap in the distortion map. Indeed, essentially all published example images show clearly noticeable artifacts, precluding visually pleasing results.

3.2.4 Approaches for low-frequency environmental lights

Although we concentrate on shadows cast by small area light sources, it is worth noting that some real-time object-space algorithms exist for large low-frequency light sources like environment maps. They typically represent both incident radiance as well as light visibility as a directional function in the spherical harmonics (SH) basis. Note that because visibility is a high-frequency signal due to its binary nature, a projection into the SH basis always incurs an approximation error and causes smoothing.

Mainly targeting articulated characters, Ren et al. [313] first construct a hierarchical sphere set approximation for each scene object in a preprocess. During runtime, visibility is determined by accumulating the occlusion due to the spheres. Since each sphere covers a circular solid angle, a corresponding generic pretabulated SH projection can easily be utilized. By multiplying the resulting visibility functions for the single spheres (or adding them in log space), correct occluder fusion is performed, albeit the low-frequency nature of the SH representation prevents accurate results. Sloan et al. [359] later improved the method and extended it to incorporate indirect lighting.

While the sphere approximation and the choice of the SH basis enable real-time performance, they also cause unrealistic and missing shadows in locations where rather higher frequencies occur, e.g. where a box touches the ground. In particular, contact shadows are often wrongly shaped and partially lose their contact, which is suboptimal, since they constitute visually important features [376]. This inaccuracy would become especially noticeable if the environmental light featured a small region of high radiance, like the sun.

By contrast, such key lights are explicitly supported by Snyder and Nowrouzezahrai [360] who consider the special case of shadowing dynamic height fields. They approximate the horizon blocking the environmental light by visibility wedges and utilize pretabulated SH projections for them to determine the overall visibility.

3.2.5 Discussion

Summing up, one major class of algorithms for rendering soft shadows at real-time frame rates seeks only or at least primarily for visual plausibility. This comprises mainly earlier approaches developed for older graphics hardware as well as recent methods striving for simplicity and high speed, like the ones blurring hard shadow maps. Generally, such algorithms resort to heuristics

like adding outer and inner penumbrae to hard shadows and sometimes even ignore the actual light size.

Many other techniques, especially more recent ones, try to approximately determine physically correct shadowing. To obtain accurate soft shadows, a rather exact knowledge is required about the fraction of the light source's area that is visible from a point to be shaded. In principle, two approaches exist for computing this point-to-region visibility. The first option is to sample the light source's surface and determine the resulting shading-point-to-light-point visibility relations. Overall light visibility is then obtained by averaging these binary results. Note that our method for correctly handling occluder fusion adopts such a strategy (see Sec. 4.2). Moreover, almost all more recent quasi-interactive algorithms for accurate soft shadows, covered in the next section, follow this approach.

Alternatively, regions of the light covered by the projections of the occluding geometry onto the light source area may be combined to determine exact light visibility. Maintaining and merging such region information, however, is non-trivial, and therefore, typically approximations are made, like representing visibility in the SH basis. While this still enables properly accounting for occluder fusion, the simple method of just summing up the areas of the projections is not able to correctly handle overlaps.

It is interesting to note that often essentially a visibility image of the light source surface is produced. A straightforward way to generate it for a certain point to be shaded is to render the scene from this point, focusing the frustum onto the light source. Since this has to be done for each visible scene point, the approach is far too expensive if the rendering is performed explicitly into a separate color buffer for each point. Therefore, sample-based methods usually maintain a per-fragment bit field to store a low-resolution binary image. Other algorithms operate solely on a single visibility percentage value, i.e. on a real-valued visibility image of size 1×1 , but cannot correctly perform occluder fusion. Typically, the visibility image is then created by successively projecting potential occluders onto the light source, updating the image to incorporate the according occlusion regions. Real-time performance often also demands to consider only a sparse set of potential occluders for deriving the visibility of the light source.

3.3 Quasi-interactive approaches for accurate soft shadows

While all real-time algorithms discussed so far generate only approximate soft shadows, several approaches exist which can produce accurate soft shadows. On current hardware, however, they only achieve at best interactive frame rates for more complex scenes.

Most of them tackle the visibility problem by placing sample points on the light source and computing the resulting binary point-to-point visibility relations. A related straightforward method, which we employed for obtaining reference images, is to acquire a shadow map for each light sample point and average the resulting hard shadows [161]. Note that due to the different sampling of the scene, care is necessary to avoid surface acne and aliasing problems.

Backprojecting silhouettes

Soft shadow volumes as presented above in Sec. 3.2.2 were later adopted for off-line raytracing and appropriately extended to generate accurate soft shadows [203, 214]. Most notably, penumbra wedges are constructed not just for silhouettes as seen from the light center but for all edges constituting a silhouette from some point on the light. Moreover, instead of adding up the light areas occluded by silhouette loops, many light sample points are considered. For

each such point, a counter stores the number of silhouette loops whose projections enclose the point. From these relative depth complexity values, the binary light-point visibilities can be derived by casting one ray to any sample point with the smallest counter value.

Forest et al. [131] adapt this improved variant to rasterization-based rendering with graphics hardware. They initialize the depth complexity counters with a standard shadow volume pass and update them by successively rendering a penumbra wedge for each considered silhouette edge. The invoked pixel shader determines the affected light sample points via texture lookups and outputs the corresponding counter changes. With the counters being stored in multiple color buffers, additive blending incorporates these updates. As the number of render targets is limited, several counters are packed into a single color channel. Nevertheless, only a rather small number of light sample points (usually 64) is supported within a single rendering pass, causing soft shadows to be quite noisy unless interleaved sampling [182, 348] is utilized, which, however, leads to some banding artifacts. Moreover, counters may overflow and numerical robustness problems seem to arise, making many example images suffer from quite noticeable artifacts, like single bright pixels within shadows.

Backprojecting triangles

Alternative approaches avoid silhouette extraction and require only maintaining a single bit per light sample point, storing binary visibility. Instead of operating on silhouettes, they loop over all triangles of potential occluders. For each triangle and receiver point \mathbf{p} , the bitmask corresponding to the occluded light points is determined and incorporated into the bit field encoding the sample points' visibility at \mathbf{p} via a bitwise OR operation. Note that our real-time method pursues exactly this approach, but works on micro-occluders extracted from shadow maps instead of on scene triangles.

An effective strategy to realize such a procedure for arbitrarily distributed blocker triangles is to scatter the occlusion due to each triangle to all affected receiver points. It was first proposed by Laine and Aila [202] within an off-line algorithm, where a hierarchical structure is used to quickly identify the shadowed receiver points (corresponding to pixel centers).

Adapting the approach to rasterization-based rendering, Eisemann and Décoret [107] restrict themselves to a planar receiver. For each triangle, a bounding box is rendered that conservatively covers the triangle's influence region, i.e. the set of potentially affected points on the receiver. The triggered pixel shader determines the bitmask designating the light samples occluded by the triangle by consulting a look-up texture. Utilizing logical pixel operations (only supported by OpenGL), these bitmaps are accumulated in multiple integer-valued color buffers. While yielding accurate soft shadows, this algorithm only supports planar and some bumpy shadow receivers and doesn't account for self-shadowing.

Addressing these shortcomings, Sintorn et al. [358] improved and extended the method. They first transform the scene points visible from the camera into shadow map space (for the light center) and store them in an alias-free shadow map, where each texel maintains a list of receiver points mapped to this shadow map entry. Subsequently, a hexagon is rendered in shadow map space for each triangle, conservatively covering the affected receiver points. Note that each created fragment corresponds to a shadow map texel and hence has access to the related list of receiver points. Traversing this list, the launched pixel shader derives for each receiver point a bitmask capturing the occlusion due to the currently processed blocker triangle. Again, these bitmaps are accumulated in multiple buffers. However, since the number of bound render targets is restricted, only a fixed number of receiver points can be considered for each texel within

a single rendering pass, often requiring multiple passes to completely process the per-textel lists of receiver points. Not least because of this, the algorithm doesn't achieve real-time frame rates yet except for some simple setups featuring a small light source and a low scene triangle count. However, the produced soft shadows are accurate and of high visual quality.

Beam tracing

While all these methods only compute a sampling of light visibility, some approaches explicitly derive the (non-overlapping) regions of the light source occluded by scene geometry, allowing exactly determining visibility. However, the complexity of handling and updating such region information makes them expensive and causes a GPU-based realization to not appear reasonable (yet). On the other hand, several CPU-based techniques exist. For instance, Overbeck et al. [281] describe a fast CPU-based beam tracer [160]. Here, a (pyramidal) beam is cast from each receiver point to the triangular or quadrilateral area light. Each time a scene triangle is hit by the beam, the beam is split, cutting out sub-beams hitting the triangle and leaving sub-beams missing the triangle. Eventually, the union of all miss sub-beams identifies the visible part of the light source. Note that performance degrades for highly tessellated scenes as a beam potentially undergoes many splits.

CHAPTER 4

Soft shadow mapping with occluder backprojection

Rendering soft shadows greatly helps realism but for real-time performance currently approximations are still necessary. Of the many methods reviewed in the last chapter, a prime candidate is the soft shadow mapping approach, where an approximation of the occluder geometry, constructed from a shadow map, is backprojected onto the light source to determine light visibility. This image-based technique features the desirable properties of yielding physically plausible soft shadows and being simple enough to allow fast execution. On the other hand, the basic algorithm suffers from several shortcomings which in general preclude high visual quality and keep down the achievable frame rate due to performing unnecessary work. Addressing and alleviating these weaknesses, we developed various solutions which make soft shadow mapping a practical method for delivering high-quality physically plausible soft shadows in real time.

In this chapter, we first review the basic soft shadow mapping approach and then describe our contributions in detail. These typically concern orthogonal aspects, and hence may be adopted independently. In particular, they can selectively be applied to and mixed with alternative methods. At first, we deal with visibility determination and present our technique for solving the important occluder fusion problem within real-time rendering. Note that, while introduced in the context of soft shadow mapping, our solution is also valid for and applicable to other approaches for rendering soft shadows.

Subsequently, we cover acceleration structures which help to effectively restrict computations to relevant occluders. Our hybrid shadow map incurs little overhead but significantly reduces the required processing effort, thus considerably speeding up rendering. The following two sections are dedicated to occluder approximations derived from a shadow map. The introduced approaches often produce improved reconstructions of the occluder geometry, enhancing visual quality. Finally, we describe a simple visibility interpolation scheme enabling multisample support at negligible extra costs, before concluding the chapter with results.

4.1 Basic approach

Recall from Sec. 3.2.1 that soft shadow mapping comprises a whole group of algorithms which follow the same general idea but differ in how they implement it. In the following, we concentrate on the variant introduced by Guennebaud et al. [147]. Note that most other algorithms [17, 31] are pretty similar to it, except the initial, occluder-driven method by Atty et

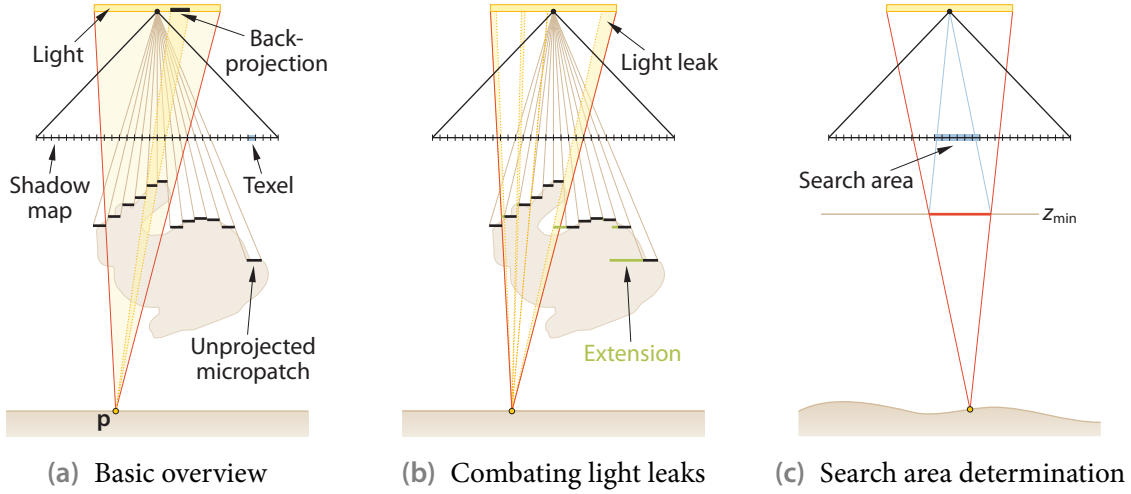


Figure 4.1 (a) Basic soft shadow mapping derives occluder approximations by unprojecting shadow map texels into world space, and backprojects these micropatches onto the light source, accumulating the occluded areas, to determine its visibility from a point p . (b) Gaps emerging in the reconstruction of blockers can cause light leaks and are (approximately) closed via micropatch extension. (c) To process only relevant occluders, a conservative search area in the shadow map is determined by exploiting knowledge about the minimum depth value z_{\min} in (a region of) the shadow map.

al. [19], which, however, requires distinguishing between shadow casters and shadow receivers, thus ignoring self-shadowing. Like for many other versions [30, 93, 148] too, the presented basic approach serves as foundation for our contributions detailed in the subsequent sections. While at least our method could deal with arbitrarily shaped planar light sources, we restrict ourselves to rectangular area lights.

At first, a standard shadow map is generated from the center of the extended light source.¹ This depth map provides a point sampling of the scene geometry visible from the light center, and hence a representation of a subset of the occluders. An approximation of the captured geometry is generated by constructing a *micro-occluder* for each shadow map element. Typically, the whole texel is just unprojected into world space, resulting in a rectangular *micropatch* parallel to the shadow map’s near plane, as illustrated in Fig. 4.1 a. Alternative occluder approximations are discussed in Sec. 4.4.

The light visibility and hence the degree of shadowing is computed in a pixel shader, either as part of the shading calculation during rendering the scene objects or within a separate pass in a deferred shading setup [91, 157]. For each fragment with associated scene point p , the shadow map is traversed, constructing a micropatch on the fly for each texel. If a micropatch is closer to the light source than p , it potentially blocks some part of the light. In this case, the micropatch is backprojected from p onto the light plane and clipped against the light’s extent to determine the occluded light area. Note that clipping and area computation are simple operations because the micropatch’s projection constitutes an axis-aligned rectangle in light plane space. The individual covered light areas of all backprojected micropatches are summed up to get an estimate of the overall occluded light area. Relating this to the total light area yields a visibility factor describing the percentage of light visible to p .

¹In principle, any sample point on (or behind) the light source may be used instead of the center.

Instead of naively looping over all texels, ideally only those micro-occluders are processed for a certain fragment which actually project onto the light source and hence block some light. A corresponding rectangular shadow map *search area* encompassing these relevant micro-occluders is given by intersecting the shadow map's near plane with the point-light pyramid defined by the point \mathbf{p} and the rectangular area light. This conservative first estimate can be further tightened if the depth range $[z_{\min}, z_{\max}]$ of the samples within the search area is known. By intersecting the plane $z = z_{\min}$ with the pyramid and projecting the result onto the near plane, the search area may then be refined iteratively (see Fig. 4.1 c). Knowledge about the depth range of the search area further allows identifying fragments in umbra and completely lit regions, where no micro-occluders need to be processed at all. More precisely, if $z_p > z_{\max}$ holds, where z_p denotes the shadow map depth value for point \mathbf{p} , it can safely be assumed that the light is totally blocked.² Similarly, $z_p \leq z_{\min}$ ensures that the whole light source is visible.

To quickly determine the depth range of a shadow map area, an acceleration structure is used. The *hierarchical shadow map* (HSM) [147], which essentially equals a hierarchical z-buffer [142], is a mipmap-like pyramid [399] for the original shadow map that stores successively aggregated minimum and maximum depth values at each coarser level. To answer a depth range query, typically the finest level is chosen where up to 2×2 adjacent texels conservatively cover the area in question. While this keeps the number of required texture fetches constant, the actually considered shadow map area is usually larger than the specified area, resulting in looser depth bounds. As a consequence, the search area is often unnecessarily large and classifications as entirely shadowed or completely lit may be prevented. By contrast, our acceleration structures detailed in Sec. 4.3 allow much more fine-grained area queries, thus drastically reducing the amount of soft shadow computations.

Even if only relevant micro-occluders are considered, their number can easily reach and exceed several thousand for a single fragment. Such large counts take a considerable amount of time to process, frequently preventing real-time performance. A simple approach to alleviate this problem is to resort to a coarser-resolution shadow map for constructing micro-occluders. In practice, the minimum channel of an appropriate level of the HSM is employed to this end. Typically, an upper bound on the number of micro-occluders considered during visibility determination is imposed. For each fragment, the finest HSM level is then selected where the search area comprises few enough texels to meet this budget.

Shortcomings

The basic soft shadow mapping algorithm suffers from several shortcomings. Solely using a single shadow map to convey occluder information naturally provides only a sampling of a subset of all light blockers. Consequently, any derived occluder reconstruction is typically just an approximation of the actual geometry. In particular, fine structures may be missed and light blockers which cannot be seen from the light source's center are wrongly ignored, which can lead to noticeable artifacts. An often-quoted example is a box below a large light source hovering over a ground plane, where only the box's top face is accounted for but the remaining parts are ignored, causing the umbra to be completely missed (see Fig. 4.3 on page 35). This can be alleviated to a certain degree by our method for visibility determination (cf. Sec. 4.2), which readily supports dealing with multiple shadow maps. However, while additional depth maps improve accuracy, creating and processing them increases the required amount of time.

²Actually, this assumption is only correct if any two micropatches adjacent in texture space correspond to two neighboring parts of the same surface, which, however, is typically assumed (see shortcomings below).

The adoption of a single shadow map (or a small number of them) is hence a deliberate design decision trading accuracy for speed.

Another problem is that gaps can occur between neighboring micropatches. Usually, such gaps are not actual occluder-free regions but undesired holes in the reconstruction of surfaces. These result from the piecewise-constant approximation with micropatches and lead to disturbing light leaks. Given the lack of information allowing a correct discrimination, it is hence reasonable to try to close all gaps. To this end, Guennebaud et al. [147] consider the left and bottom neighbors in the shadow map for each micropatch, dynamically extending it appropriately to the borders of these neighbors (see Fig. 4.1 b). We observe, however, that gaps towards the diagonal neighbor may still exist [333], which can be alleviated by explicitly accounting for this neighbor, too [29]. In Sec. 4.4, we present an alternative to micropatches which implicitly closes any gaps and further improves on the reconstruction of occluders from the shadow map.

Note that invariably closing gaps is a necessary approximation. It knowingly accepts over-occlusion artifacts due to blockers that are wrongly introduced by assuming that two adjacent shadow map texels sample the same surface. Actually, correctly dealing with such gaps in absence of further information is a general problem also encountered in raytracing of depth images [181, 224]. While heuristics were developed, like performing gap filling between two adjacent micropatches only if their depth difference is below a user-specified threshold [7], they are far from robust. Therefore, we feel the best we can do to reliably avoid light leaks is to close all gaps without exception.

A further issue with the basic algorithm concerns the way the occlusion of individual micropatches is combined. Accumulating the areas covered by them is only correct if the projections of the micropatches onto the light source don't overlap. This, however, is typically not the case, although initially recording only occluders visible from the light's center surely helps obviating overlaps. The resulting incorrect occluder fusion leads to overestimating light occlusion and may cause clearly objectionable artifacts. To avoid them, we developed a robust solution for visibility determination that properly deals with arbitrary such overlaps. It is described in the next section.

As mentioned before, only a rough conservative estimate of the depth range for arbitrary rectangular shadow map areas can be obtained with a small fixed number of accesses into the HSM. Therefore, often significantly more micropatches are processed than actually required, which negatively impacts performance. Mitigating this problem of performing irrelevant work, we devised more effective acceleration structures, covered in Sec. 4.3.

Furthermore, selecting the *shadow map (mipmap) level*³ for micro-occluder construction individually per fragment can lead to noticeable transition artifacts. If different levels are used for two adjacent pixels, different occluder approximations get employed, which generally yield an unequal amount of light blocking. In particular, resorting to a coarser level for deriving micropatches typically causes an overestimation of the occluders' size. While we present an approach for coarser micro-occluders in Sec. 4.5 that improves the approximation quality, the avoidance of transition artifacts is discussed in Chapter 5 as part of a more generic scheme for adapting computational efforts.

³We use the more general term *shadow map level* instead of *HSM level* in the following, since this multi-resolution information may also be provided by alternative representations other than the HSM.

4.2 Visibility determination with occlusion bitmasks

The visibility of the light source is typically determined by summing up the areas covered by individual micro-occluders. This simple approach, however, ignores potential overlaps, which can lead to severe artifacts like those in Figs. 4.32 b and 4.33 b on page 65 and 66, respectively. To address this grave problem, we developed a new algorithm, *bitmask soft shadows* (BMSS) [333]. It samples the light source and employs a bit field to maintain the visibility state for the light points. Providing a solution to the underlying occluder fusion problem, BMSS enable further applications, like incorporating occluder information from multiple shadow maps or correctly handling multi-colored light sources.

4.2.1 Occlusion bitmasks

Instead of keeping and updating a single visibility factor, we sample the light source visibility via several point-to-point relations in our BMSS algorithm. More precisely, we place sample points on the light source and use a bit field to track which of them are occluded. The resulting *occlusion bitmask* provides a discrete representation of which light area parts are occluded. We then directly use the number of set bits to determine the light's visibility.

As before, each relevant micro-occluder is backprojected but rather than computing its area, we determine a bitmask reflecting which light samples are occluded by it, and incorporate this into the occlusion bitmask with a bitwise OR. Note that in case of overlapping micro-occluders the same bit gets set multiple times, i.e. occluder fusion is automatically dealt with correctly.

In principle, the sample points can be placed arbitrarily on the light source. Ideally, they are decorrelated and follow some random pattern to obviate banding artifacts. As explicitly looping over all samples is prohibitive, lookup textures may be employed. Indeed, for micropatches a simple strategy similar to summed-area tables [83] is possible because their backprojections are just axis-aligned rectangles on the rectangular light source. Assuming the light area is parameterized as a unit square $[0, 1]^2$, we generate a lookup texture of size $(n_x + 1) \times (n_y + 1)$ for each group of 128 samples (4 integer channels à 32 bit), with texel (i, j) storing the bitmask corresponding to the covered area $[0, i/n_x] \times [0, j/n_y]$. By utilizing the identity

$$\text{bitmask}([x_0, x_1] \times [y_0, y_1]) = \text{bitmask}([0, x_1] \times [0, y_1]) \text{ AND } \\ \left(\text{NOT} \left(\text{bitmask}([0, x_1] \times [0, y_0]) \text{ OR } \text{bitmask}([0, x_0] \times [0, y_1]) \right) \right),$$

the bitmask representing the occlusion due to a micropatch covering the region $[x_0, x_1] \times [y_0, y_1]$ can then easily be determined with just three texture accesses per group of 128 samples.

Nevertheless, given the often high number of micropatches processed per fragment, performing at least three texture fetches for each of them can be rather expensive. Therefore, we typically restrict the positioning and number of sample points such that fast updates of the occlusion bitmask are possible using only arithmetic operations. As illustrated in the bottom row of Fig. 4.2, we considered regularly placed, uniformly spaced sample points for bit fields of size 8×8 , 16×16 and 32×32 . These allow efficiently updating the occlusion bitmask to incorporate the light blocking due to a new micropatch. To this end, we first map the micropatch's axial extent into bit ranges via scale, bias and integer conversion operations and then derive a bitmask for the micropatch, which gets OR-ed with the occlusion bitmask.

While 16×16 sample points often offer enough levels of visibility discrimination, a strictly regular sampling pattern is usually suboptimal and can lead to clearly visible discretization ar-

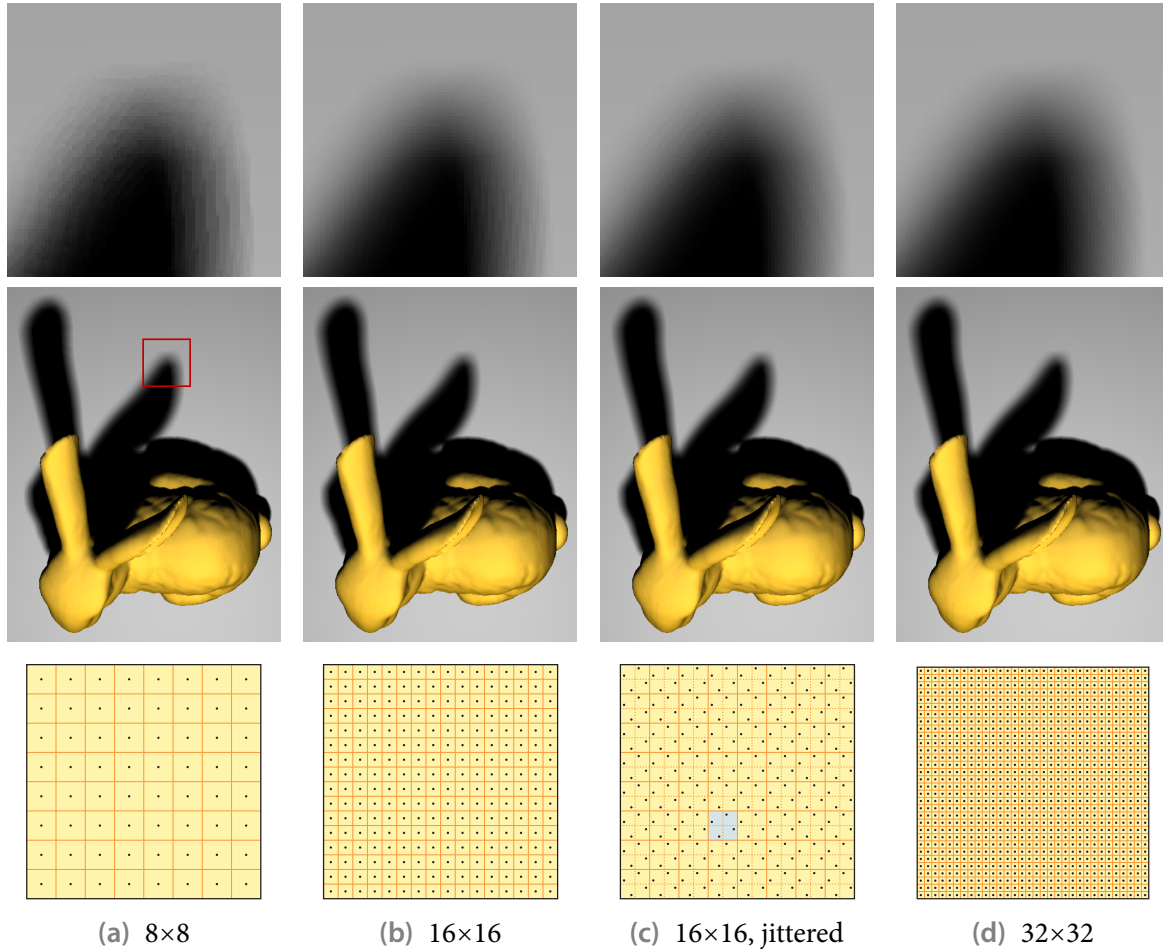


Figure 4.2 Overview of the various sampling patterns considered. Top rows: resulting soft shadows with zoom-in. Bottom row: placement of used sample points on a unit light source. In (c), the 2×2 RGSS pattern is highlighted.

tifacts. We alleviate this by regularly jittering the sample positions such that the resulting sampling pattern is identical to an 8×8 tiling of the 2×2 rotated grid super sampling (RGSS) pattern. This is known to offer good antialiasing quality for nearly vertical and horizontal edges [8]. Similarly, our jittered sampling pattern is well suited to deal with the encountered axis-aligned rectangles. In particular, it often closely matches the quality of the regular 32×32 sampling pattern.

Note that updating an occlusion bitmask for other micro-occluders than micropatches can become significantly more involved because their projections may not be axis-aligned rectangles. When we deal with such alternative micro-occluders in Sec. 4.4, both approximate and accurate techniques for bitmask determination are hence discussed.

4.2.2 Advanced applications

Since occlusion bitmasks can correctly handle arbitrarily overlapping micro-occluders and also provide explicit information about which parts of the light source are blocked instead of offering just a visibility factor, new applications beyond standard soft shadow mapping become possible.

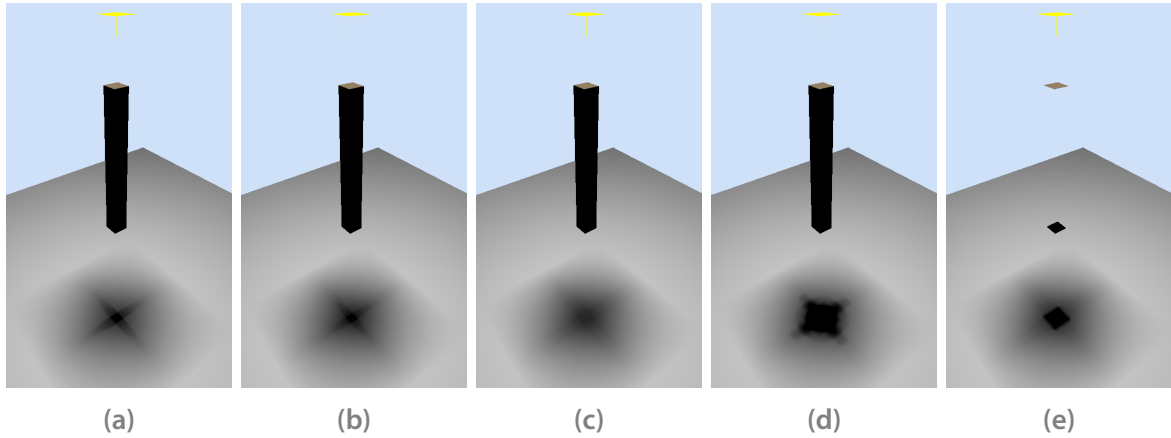


Figure 4.3 Single silhouette artifact setup. (a) Reference image. (b) Occlusion bitmaps with two depth layers (16×16 , jittered). (c) Standard soft shadow mapping with area accumulation. (d) Maximum of the occlusion accumulated separately for two depth layers as suggested by Atty et al. [19]. (e) Maximum of the occlusion due to two separately considered planar light blockers.

Multiple shadow maps

So far, we have used only information from a single shadow map. Concerning visual quality, our algorithm hence improves “merely” on artifacts due to overlaps encountered in standard soft shadow mapping with area accumulation. In several cases, these artifacts can be rarely noticeable and hence may even be acceptable. This is mainly because by solely processing occluders visible from a dedicated point on the light source, the possibility of overlap is kept small compared to techniques like soft shadow volumes in the first place.

On the other hand, thanks to correctly handling occluder fusion, occlusion bitmaps make it further possible to incorporate occluder information extracted from multiple shadow maps in the visibility determination process. This enables us to basically capture all occluders and correctly account for them, barring approximation-related inaccuracies, like invariably closing all gaps. To capture additional occluders, one could render shadow maps from multiple sample points on the light source. An easier way is to perform depth peeling [111, 246] from a single sample point. Note that to keep the cost of additional rendering passes low, a smaller frustum can be used for all but the first depth layer. For instance, one might focus only on objects directly below the light source, since usually most occlusion artifacts occur there.

By this ability to resort to multiple shadow maps, occlusion bitmaps lift one severe limitation of alternative real-time soft shadow algorithms, namely processing only those occluders which are visible from a single sample point. As demonstrated by the well-known single silhouette artifact setup [15] in Fig. 4.3, this restriction can lead to objectionable artifacts that deprive the image of important depth cues (cf. Subfig. c). In contrast, employing depth peeling to capture additional occluders (Subfig. b) allows us to come quite close to the reference obtained from averaging 1024 hard shadows (Subfig. a). Furthermore, Subfigs. d and e illustrate that heuristics, like determining the occlusion due to the closest front faces and due to the farthest back faces separately and taking their maximum, do not necessarily yield correct results.

In addition, it is possible to use a separate shadow map per object or group of object. For



Figure 4.4 Phlegmatic dragon illuminated by a two-colored EG logo light source (inset).

instance, in case of a scene composed of a moving object and other objects currently not moving, we can derive occlusion bitmasks for them at different rates and combine these bitmasks each frame to determine the correct light visibility.

Multi-colored light sources

Since an occlusion bitmask provides detailed spatial information about the occluded parts of the light source, bitmask soft shadows can readily be adapted to deal with multi-colored lights.

In case of few different unique colors, we use bitmasks identifying sample points of the same color. For each, we perform a bitwise AND with the occlusion bitmask and count the set bits to derive the corresponding visible fraction of the overall light area. These visibility factors then serve as weighting factors for the colors. Fig. 4.4 shows an example using a bit field of size 32×32 . Note that this extension introduces far less overhead than approaches using a 4D texture [15] or a summed-area table [147], which necessitate multiple texture accesses per backprojected micropatch and also cannot correctly handle occluder fusion.

In case of arbitrarily textured light sources with many different colors, it becomes more efficient to employ lookup textures for subgroups of the bit field in order to derive the color for the visible fraction of the light source. For instance, a 2D texture of size $256 \times \lfloor n/8 \rfloor$ may be created in case of n light sample points and groups of 8 bits. A separate row is dedicated to each group, storing the visible light color for all 256 bit combinations. During rendering, a texture lookup is then performed for each group, using the integer encoded by the group's bits as column index.

Note that non-rectangular planar light sources can be interpreted as a special case of a two-colored rectangular light source. To this end, a rectangular light source whose extent encom-

passes the non-rectangular one is employed together with a bitmask identifying those sample points which are actually within the non-rectangular light's area.

4.2.3 Discussion

Occlusion bitmasks offer a robust solution to the occluder fusion problem. The underlying key idea is approaching point-to-region visibility determination as a sampling problem, involving multiple point-to-point visibilities.

Relation to casting shadow rays

Similarly, in ray-based image synthesis methods like distributed ray tracing [80], the visibility of a light source is computed by casting shadow rays to multiple light sample points. Each ray is tested for intersection against the actual scene geometry. Note that once any surface is hit, the light point corresponding to the ray is known to be occluded, i.e. no further scene objects have to be processed.

By contrast, bitmask soft shadows work only on an approximation of the occluder geometry derived from a shadow map (or multiple maps). Moreover, the computation is structured differently. Instead of looping over the sample points, gathering occlusion from light blockers, all potentially relevant micro-occluders are enumerated and projected onto the light area to identify occluded light points, thus scattering the occlusion caused by the micro-occluders.

Discretization artifacts

Since we determine the visibility via point sampling, a hard transition occurs when a sample finally gets occluded and hence a bit becomes set. Therefore, the encountered light visibility factors can only take on a limited number of different discrete values, defined by the size of the bit field. It is hence pertinent to employ enough light samples to avoid banding artifacts, where single discernable color bands appear instead of a smooth gradient. Considering that usually both displays and color buffers allocate at least eight bits per color channel (using a non-linear encoding, though), not less than 256 light sample points should be used, although sometimes significantly fewer may suffice.

On the other hand, discretization artifacts can also arise if more than one bit changes state when transitioning between two points corresponding to two adjacent pixels in the final image. Note that such a case is not that unlikely because micropatches are axis aligned and the employed sampling pattern is typically (semi-)regular. Hence, artifacts are particularly severe if a large silhouette edge of a shadow caster is in good alignment with a border of the light source as in the scene depicted in Fig. 4.3.

In practice, however, discretization artifacts are often acceptable, especially since they are usually masked by the texture of the shadow-receiving surface and hence remain (largely) imperceptible. To alleviate such artifacts, nevertheless, the determined visibility may be stored in an intermediate visibility buffer, which can then be filtered according to simple heuristics [333].

Implementation notes

In our implementation of occlusion bitmasks, a reasonable effort was spent on optimization. Nevertheless, updating the bitmask for the jittered 16×16 sampling pattern still requires 131 scalar instructions in our GLSL version. Note that such a high number of arithmetic operations

```

1 void UpdateMask(in vec4 lPos,
2               inout uvec4 mask0,
3               inout uvec4 mask1)
4 {
5     lPos = vec4(16.0) * lPos;
6     uvec4 bitrangeP = uvec4(lPos + vec4(0.5 + 0.25));
7     uvec4 bitrangeM = uvec4(lPos + vec4(0.5 - 0.25));
8
9     unsigned int maskXEvenY = (1u << bitrangeP.z) - (1u << bitrangeP.x);
10    unsigned int maskXOddY  = (1u << bitrangeM.z) - (1u << bitrangeM.x);
11    unsigned int maskYEvenX = (1u << bitrangeM.w) - (1u << bitrangeM.y);
12    unsigned int maskYOddX  = (1u << bitrangeP.w) - (1u << bitrangeP.y);
13
14    unsigned int maskX = maskXEvenY + (maskXOddY << 16u);
15
16    uvec4 maskY;
17
18    maskY.x = (maskYEvenX      ) & 1u;
19    maskY.y = (maskYEvenX >> 2u) & 1u;
20    maskY.z = (maskYEvenX >> 4u) & 1u;
21    maskY.w = (maskYEvenX >> 6u) & 1u;
22    maskY.x += (maskYOddX << 1u) & 2u;
23    maskY.y += (maskYOddX >> 1u) & 2u;
24    maskY.z += (maskYOddX >> 3u) & 2u;
25    maskY.w += (maskYOddX >> 5u) & 2u;
26    maskY.x += (maskYEvenX << 15u) & (1u << 16u);
27    maskY.y += (maskYEvenX << 13u) & (1u << 16u);
28    maskY.z += (maskYEvenX << 11u) & (1u << 16u);
29    maskY.w += (maskYEvenX << 9u) & (1u << 16u);
30    maskY.x += (maskYOddX << 16u) & (2u << 16u);
31    maskY.y += (maskYOddX << 14u) & (2u << 16u);
32    maskY.z += (maskYOddX << 12u) & (2u << 16u);
33    maskY.w += (maskYOddX << 10u) & (2u << 16u);
34
35    mask0 |= (maskY * 0x5555u) & maskX;
36
37    maskY.x = (maskYEvenX >> 8u) & 1u;
38    maskY.y = (maskYEvenX >> 10u) & 1u;
39    maskY.z = (maskYEvenX >> 12u) & 1u;
40    maskY.w = (maskYEvenX >> 14u) & 1u;
41    maskY.x += (maskYOddX >> 7u) & 2u;
42    maskY.y += (maskYOddX >> 9u) & 2u;
43    maskY.z += (maskYOddX >> 11u) & 2u;
44    maskY.w += (maskYOddX >> 13u) & 2u;
45    maskY.x += (maskYEvenX << 7u) & (1u << 16u);
46    maskY.y += (maskYEvenX << 5u) & (1u << 16u);
47    maskY.z += (maskYEvenX << 3u) & (1u << 16u);
48    maskY.w += (maskYEvenX << 1u) & (1u << 16u);
49    maskY.x += (maskYOddX << 8u) & (2u << 16u);
50    maskY.y += (maskYOddX << 6u) & (2u << 16u);
51    maskY.z += (maskYOddX << 4u) & (2u << 16u);
52    maskY.w += (maskYOddX << 2u) & (2u << 16u);
53
54    mask1 |= (maskY * 0x5555u) & maskX;
55 }

```

Listing 4.1 Example GLSL code for updating a bitmask to incorporate the occlusion due to a rectangle covering the region $[lPos.x, lPos.z] \times [lPos.y, lPos.w] \subseteq [0, 1]^2$. The sample points are placed according to the jittered 16×16 pattern.

nicely hides memory latency due to texture fetches necessary for extracting micro-occluders from the shadow map.

In case of tracking visibility by a 32×32 bit field, 32 scalar registers are required just for storing the bit field. This high register count negatively affects the number of concurrently processed threads, thus limiting overall speed. Indeed, when executing the same instructions but using only 16 scalar registers, the performance increases by roughly 75% (on an NVIDIA GeForce 8800 GTS). We therefore employ temporary array variables, which are stored in slower local memory but enable a higher degree of parallelism. Compared to the register-based variant, we observed a speed-up of more than 50% (again on a GeForce 8800 GTS).

Efficiently processing bit fields is non-trivial but many sophisticated algorithms and formulations exist. For instance, counting the number of set bits is done by hierarchically adding up subgroups of bits in parallel [10]. As updating an occlusion bitmask to incorporate a micropatch's contribution is central and requires some care to achieve reasonable performance, we provide representative GLSL code for the challenging jittered 16×16 case in Listing 4.1.

4.3 Acceleration structures

For high performance, it is essential to avoid useless computations and spend the available time only on those parts which actually affect the result. In particular, only micro-occluders should be processed which actually influence the light visibility of a shadow-receiving point. Therefore, we aim to keep the number of those micro-occluders to a minimum whose backprojections don't cover the light area at all and which hence don't contribute to the final occlusion value. Since, in practice, micro-occluders are constructed for all texels within a rectangular region of the shadow map, the search area, this translates to determining the smallest bounding rectangle encompassing all relevant texels.

As detailed in Sec. 4.1, an initial coarse estimate of the search area can be iteratively refined and thus pruned with an acceleration structure. Such an auxiliary construct enables determining a conservative bound of the depth range of a specified shadow map region in constant time. Furthermore, an acceleration structure implicitly provides a multi-resolution representation of the shadow map that allows creating fewer but coarser micro-occluders to satisfy some imposed upper bound on the number of micro-occluders processed per fragment.

Recall that one simple acceleration structure is the hierarchical shadow map, which, however, often requires choosing query regions larger than the search area. Therefore, the HSM typically yields quite loose bounds and hence unnecessarily big search areas. As this negatively restricts the achievable performance, we developed an alternative, the multi-scale shadow map (Sec. 4.3.1), which significantly improves search area pruning. To combine small construction and storage costs with tight bounds, we further devised a hybrid between these two acceleration structures, termed Y shadow map (Sec. 4.3.2).

4.3.1 Multi-scale shadow map

Our *multi-scale shadow map* (MSSM) [333] is an alternative multi-scale representation of the shadow map which is not pyramidal, like the HSM and classical mipmaps, but retains the original resolution across all levels. In this acceleration structure, a texel at level i stores the minimum and maximum depth values of a neighborhood region of size $2^i \times 2^i$ centered at the

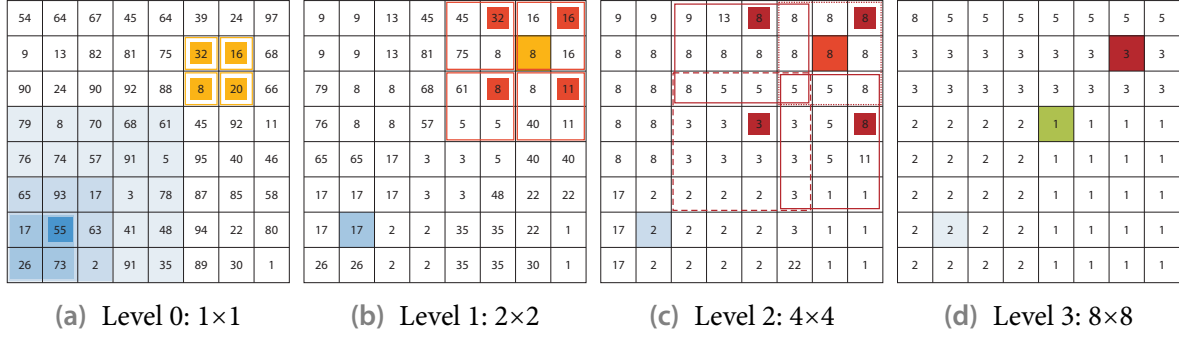


Figure 4.5 Example of a multi-scale shadow map (z_{\min} channel) of resolution 8×8 : The blue texels in levels 1–3 contain the minimum values of the identically colored regions in level 0. Each fully colored orange/reddish texel is derived from the values of the texels in the previous level of the same shade, which together cover a region as indicated by the colored rectangles. The overall minimum can be accessed via the green texel.

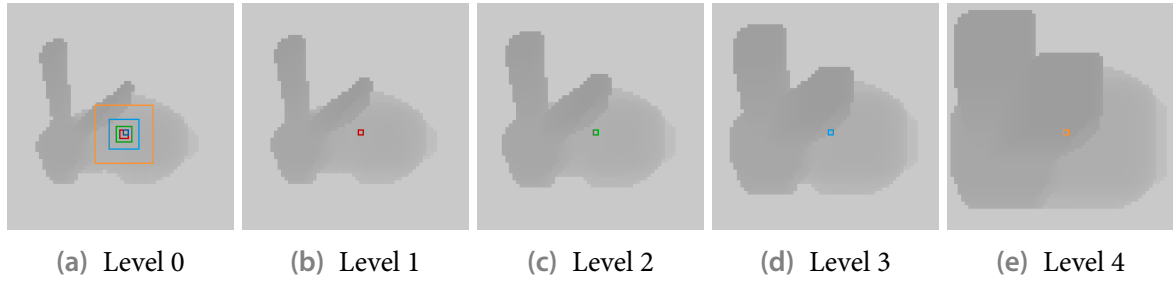


Figure 4.6 Example of a multi-scale shadow map, showing the z_{\min} channel of the first five levels. Each texel in level $i > 0$ contains the minimum depth value of a region of size $2^i \times 2^i$ in level 0 centered at the texel’s location. In particular, the highlighted texel in a level $i > 0$ covers the identically colored region in level 0.

texel in level 0. As illustrated in Fig. 4.5, each texel at level $i > 0$ can be derived from four texels t_{uv} of level $i - 1$. Because the neighborhood region is clamped to the shadow map extent, a clamp-to-edge texture wrapping mode is employed when accessing texels t_{uv} . Note that resulting overlaps of neighborhood regions don’t cause any problems, since only their extrema are combined. Typically, an MSSM gets stored in a 2D array texture because this enables the dynamic selection of the sampled level within a shader. An example of such a stack of depth ranges for all power-of-two-sized neighborhoods is shown in Fig. 4.6.

Concerning the application of the MSSM for search area pruning, we follow the general approach described before in Sec. 4.1. Initially, the search area is determined as the (clamped) projection of the light source onto the shadow map’s near plane. We then obtain a first bound $[z_{\min}, z_{\max}]$ for the search area’s depth range via a single MSSM sample. If the currently considered point \mathbf{p} is outside this range, it is either completely lit or in umbra, rendering any micro-occluder processing unnecessary.

Otherwise, z_{\min} is used to refine the search area. Subsequently, we take four samples from the appropriate MSSM level such that their associated (possibly partially overlapping) neighborhood regions tightly cover the search area to get a more accurate depth range bound. This is utilized to further refine the search area before finally starting with any micro-occluder back-

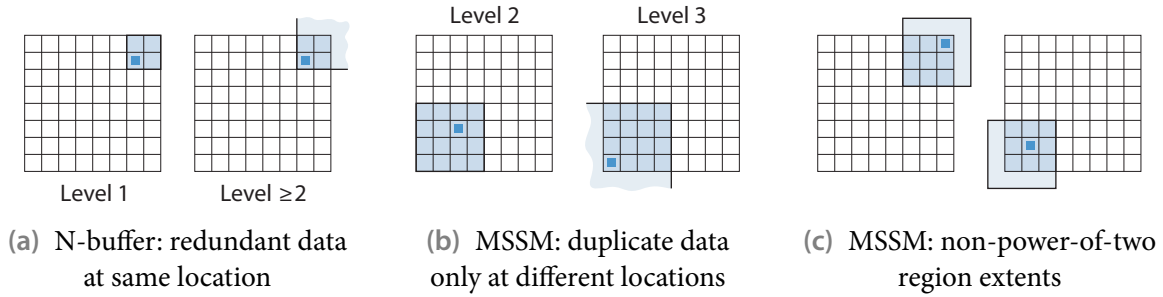


Figure 4.7 Compared to an N-buffer, the MSSM provides more detailed information near the borders.

projection. Note that, in principle, further pruning iterations can be performed, although experience shows that typically the resulting improvements are marginal at best.

As the information stored in the MSSM is essentially a superset of that in the HSM, it can readily be used to construct micro-occluders from coarser levels $i > 0$. To this end, an MSSM level i is subsampled at a rate of one in 2^i texels, making the actual sampling positions adhere to the same grid as implicitly imposed by the HSM.⁴

Relation to N-buffers

Though developed independently, the MSSM shares many similarities with the N-buffer [89] and hence may be considered a variant of it. The main difference is that we store a neighborhood region's depth range at its (discretized) center instead of its lower left corner. This has several immediate consequences, ultimately improving the results for search areas touching or crossing the shadow map's border.

Essentially, the MSSM provides more information than an N-buffer because no redundant data is stored.⁵ As illustrated in Fig. 4.7 a, in an N-buffer the $2^i \times 2^i$ texels at the upper-right corner in the i -th level are identical to the corresponding elements in all higher levels because the region about which information is gathered doesn't change any longer for these elements. By contrast, duplicate data in an MSSM always occurs at different locations and arises only if a smaller neighborhood for one texel coincides with the larger neighborhood for *another* texel after clamping against the shadow map border (cf. Fig. 4.7 b).

Avoiding redundancy by centering the neighborhood at a texel translates into the availability of more fine-grained information near the borders. In particular, unlike with N-buffers, directly supported query regions of non-power-of-two extent are not restricted to the top-most rows and right-most columns (see Fig. 4.7 c).

⁴Note that such a positional restriction is essential, since otherwise neighboring pixels in the final image may sample the shadow map at an identical scale but at slightly translated and hence different sample points, which leads to artifacts.

⁵From an information-theoretical point of view, all aggregated data stored in addition to the base data is redundant, i.e. all levels $i > 0$ contain redundant information. In contrast, we only refer to data as redundant if identical information is already stored at a single and directly related location, like the same texel in a different level. That is, we consider aggregated data as new information, under the rationale that performing such aggregation is the very purpose of an acceleration structure.

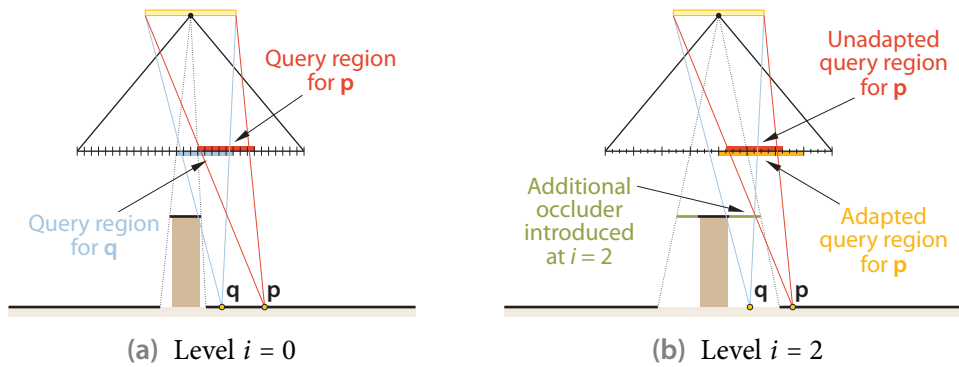


Figure 4.8 For point p , the minimum depth recorded in all texels within the shadow map region defined by intersecting the near plane with the point-light pyramid equals p 's depth. (a) Consequently, no micropatch blocks light from p if micro-occluders are constructed from shadow map level 0, and querying the MSSM for this region correctly indicates that p is completely lit. (b) However, if micropatches are extracted from shadow map level 2, p becomes partially shadowed. Therefore, when accessing the MSSM (featuring the resolution of shadow map level 0), the query region must be enlarged to completely cover the footprint of level-2 texels. Otherwise, p would incorrectly be classified as completely lit, causing a visibly noticeable hard transition from the adjacent point q (of the neighboring pixel).

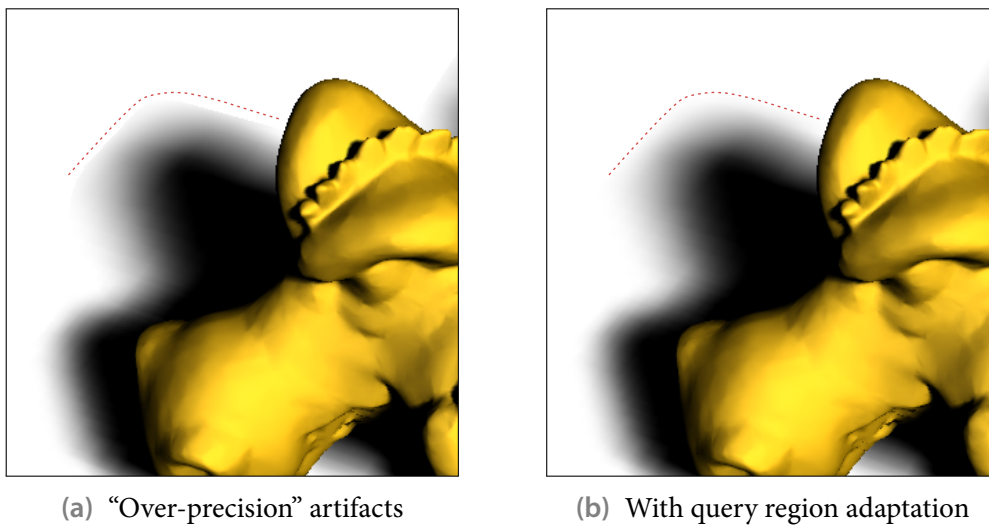


Figure 4.9 Using the MSSM may lead to "over-precision" artifacts at the interface to fully lit regions (a), which can be avoided by adapting the region queried during search area pruning appropriately (b). The red dashed line indicates the actual interface introduced by the coarser-level micro-occluders that get backprojected.

"Over-precision" artifacts

When an upper bound on the number of micro-occluders considered per fragment is imposed and hence a coarser shadow map level may be employed for constructing micro-occluders, using the MSSM to derive the search area requires some extra care. In such a case, the area queried for its depth range has to be grown to account for the current budget to avoid that resulting classifications as needing no backprojection are sometimes too accurate.

| Acceleration structure | Size | Construction time | Memory footprint |
|------------------------|----------------|-------------------|------------------|
| HSM | 1024^2 | 0.26 ms | 10.67 MB |
| MSSM | 1024^2 | 3.92 ms | 88.00 MB |
| YSM | $1024^2/256^2$ | 0.46 ms | 14.67 MB |
| HSM | 2048^2 | 0.87 ms | 42.67 MB |
| MSSM | 2048^2 | 18.65 ms | 384.00 MB |
| YSM | $2048^2/256^2$ | 1.07 ms | 46.67 MB |

Table 4.1 Cost of acceleration structures on an NVIDIA GeForce 8800 GTX. Note that the YSM figures are for a non-truncated HSM part.

More precisely, at higher levels $i > 0$, regions covered by an MSSM sample only align with texel boundaries at level 0 but not necessarily with the boundaries of the texture-space support of the coarser micro-occluders later constructed at levels $j > 0$ during visibility determination. As a consequence, when the search area is computed at the original shadow map resolution,⁶ those micro-occluders overlapping the search area border get ignored which only become relevant micro-occluders because of the contribution of samples outside the search area during minimum aggregation (cf. Fig. 4.8). If this “over-precision” ultimately results in classifying a pixel as fully lit, discontinuity artifacts can appear at the interface between penumbra and completely lit regions, as demonstrated in Fig. 4.9. To alleviate this issue, we determine the level j from the current search area and the imposed micro-occluder budget, and grow the area before querying its depth range to completely include the support regions of all partially covered level- j micro-occluders. Note that this effectively reduces the resolution of the finest MSSM levels.

4.3.2 Hybrid Y shadow map

As demonstrated in Fig. 4.10, the MSSM can significantly reduce the search area size compared to the HSM, and substantially more often enables classifying a pixel as being in an umbra or a completely lit region. On the down side, both its construction and its memory footprint render the MSSM too expensive for shadow map sizes greater than 1024^2 , since the resolution is not reduced across levels (cf. Table 4.1). By contrast, the HSM entails significantly lower costs because of its pyramidal nature, which, however, is also responsible for the usually much more conservative results.

Seeking to get the best of both approaches, we employ a hybrid between the HSM and the MSSM. This so-called *Y shadow map*⁷ (YSM) [336] is constructed by combining the first n levels of the HSM with an MSSM built from level $n - 1$ of the HSM. Note that typically an upper limit on the number of considered micro-occluders is imposed, in which case an MSSM requires growing the query region according to this budget. Therefore, a full-resolution MSSM doesn’t make much sense for the finest levels anyway.

We store the YSM distributed over a 2D texture with mipmap chain for the HSM part and

⁶Note that while the search area is determined, its final size is unknown yet and hence the level from which micro-occluders are to be extracted.

⁷The “Y” stems not only from the word *hybrid* but also reflects the development of the levels’ resolution. The upper two converging lines correspond to the pyramidal part, which transitions into a stack of equal-sized levels, represented by the lower vertical line.

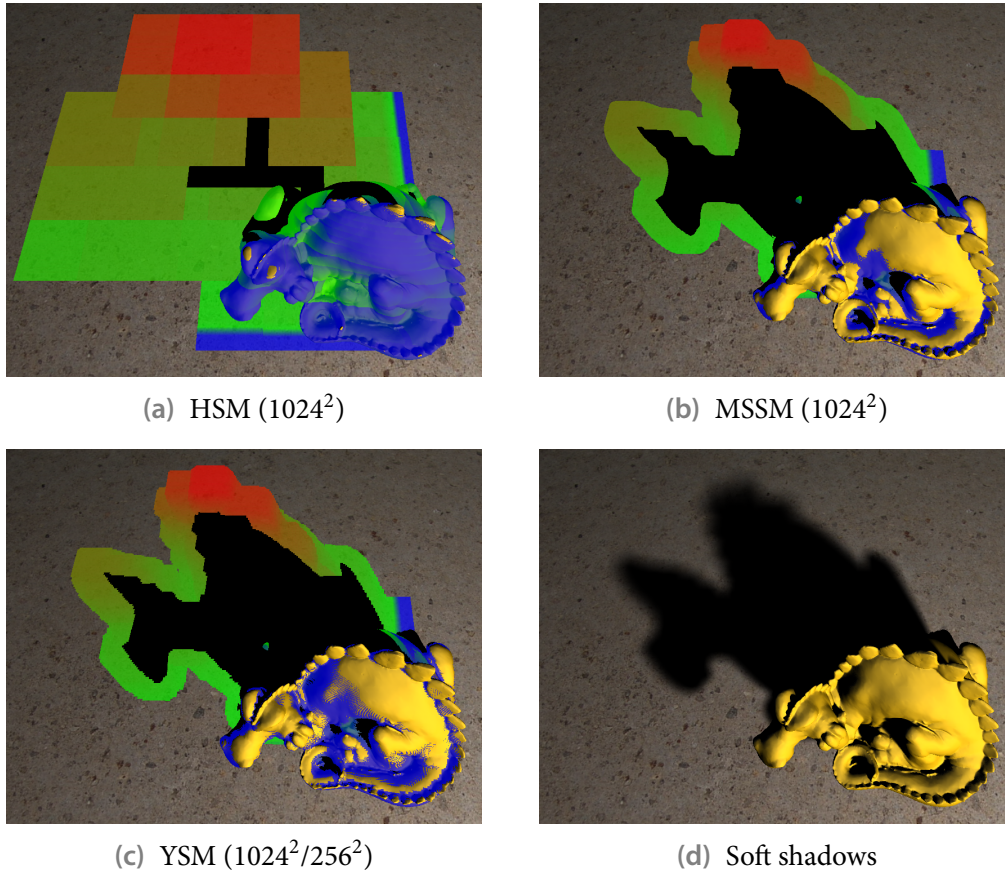


Figure 4.10 Comparison of the effectiveness of acceleration structures. Pixels which are not classified as either in umbra or completely lit are highlighted. The color coding indicates the size of the search area in a shadow map of size 1024^2 , and hence the number of (level-0) micropatches which have to be backprojected to determine the actual degree of the light's occlusion (blue: 10 micropatches or less; green: 400 micropatches; red: 6000 micropatches or more).

a 2D texture array for the MSSM part. Note that MSSM level 0 does not need to be explicitly stored because it is identical to HSM level $n - 1$. While it is possible to truncate the HSM pyramid at level n , where the MSSM begins, in our implementation we actually construct a full HSM. Although this is of no use for search area pruning, it allows us to always employ solely the HSM for micro-occluder construction during visibility determination, improving texture fetch locality and avoiding branching.

In practice, choosing a resolution of 256^2 for the MSSM part provides a good trade-off between additional construction cost and memory footprint compared to the HSM on the one hand and potentially less tight search areas compared to a full-resolution MSSM on the other hand (cf. Table 4.1 and Fig. 4.10). In our experience, the sometimes slightly increased number of pixels requiring micro-occluder processing when using the YSM in lieu of the MSSM is usually more than compensated for by the lower construction times.

4.3.3 Discussion

Acceleration structures are essential to keep the number of processed micro-occluders to a minimum by bounding the actually relevant ones in shadow map space. As exemplarily shown

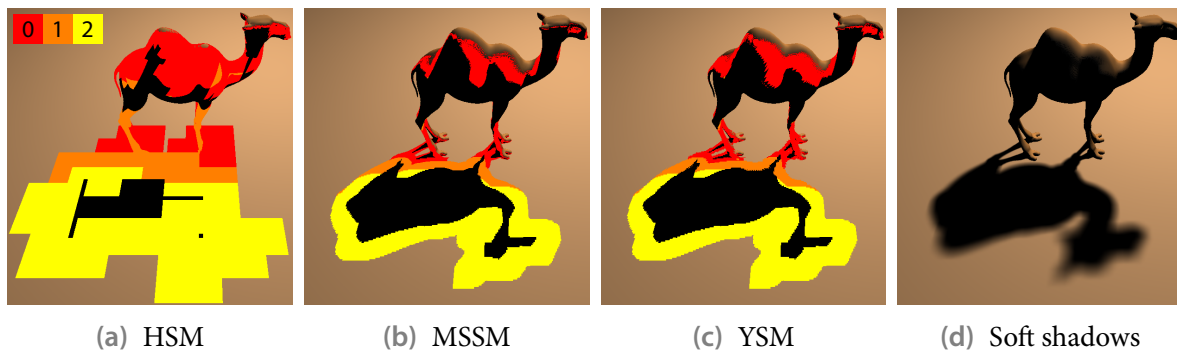


Figure 4.11 Shadow map levels used to meet a budget of 256 micropatches per fragment when employing various acceleration structures.

in Figs. 4.10 and 4.11, both the MSSM and the YSM are clearly superior to the HSM. Most notably, they often drastically reduce the number of pixels for which micro-occluders have to be backprojected thanks to being quite effective in identifying umbra and completely lit pixels. Moreover, they typically yield smaller search areas than the HSM. Consequently, fewer micro-occluders have to be processed, or alternatively, a finer shadow map level can be employed for micro-occluder construction while still meeting an imposed budget. On the other hand, the YSM incurs only a marginal overhead concerning construction time and occupied texture space with respect to the HSM, as shown in Table 4.1.

Key to the improved results obtained with the MSSM (and the YSM) is its ability to directly support queries for arbitrarily placed squares of power-of-two size. By contrast, the HSM, essentially being a quadtree constructed over the shadow map, is restricted to rectangles corresponding to quadtree tiles. Therefore, an arbitrary power-of-two-sized square typically has to be grown to the next coarser encompassing tile, quadrupling its size. Consequently, the depth range is looser, often precluding a classification as completely lit or in umbra.

Note that the hierarchical traversal of an HSM as suggested by Dmitriev [93] similarly suffers from supporting only quadtree tiles for classifying pixels. Moreover, this approach involves maintaining a stack, which increases the shader’s register count and consumes additional time.

4.4 Occluder approximations

A shadow map provides a point sampling of the geometry visible from a dedicated point on the light source. Reconstructing an approximation of the captured occluders from this scene information is central to soft shadow mapping algorithms. Although micropatches, obtained by unprojecting shadow map texels into world space, are often employed and simple to create, they are neither the only nor necessarily the best option. Some alternative approximations are shown in Fig. 4.12.

Addressing major shortcomings of micropatches, we developed microquads (Sec. 4.4.1), which feature several desirable properties, like implicitly avoiding light leaks. To further improve quality, we augmented them with microtris (Sec. 4.4.1). Both microquads and microtris are rather challenging to process, and we hence devised both approximate and exact solutions for determining the occlusion caused by them. Other researchers devised different options, which we briefly cover in Sec. 4.4.5.

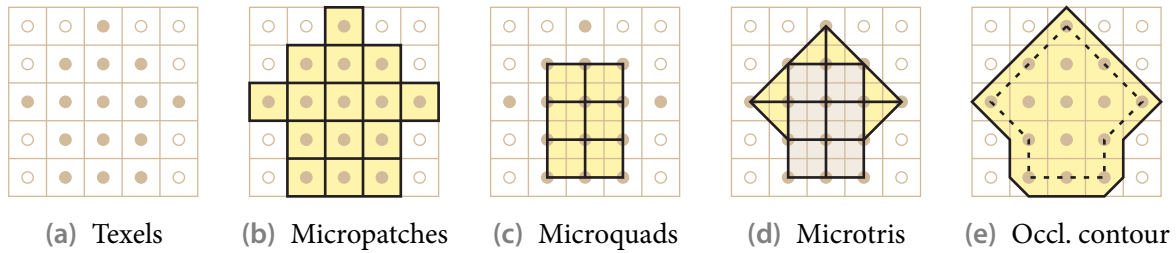


Figure 4.12 Texture-space footprint of various occluder approximations (b–e) constructed from an example shadow map excerpt (a; filled circles indicate texels which are closer to the light than the currently processed point).

4.4.1 Microquads

Micropatches provide a piecewise-constant approximation of the occluders' geometry. Since they are parallel to the light plane such that their backprojections are axis-aligned rectangles, many operations are rather simple and hence fast to execute. However, this simplicity also causes micropatches to suffer from several problems. For instance, gaps are introduced, which require special processing to avoid light leaks. Moreover, occluders are frequently overestimated (cf. Fig. 4.14 a), potentially leading to noticeable enlargements of the penumbra's extent (see Fig. 4.33 on page 66 for an example).

To alleviate these shortcomings, we construct a different kind of micro-occluder from the shadow map data. Instead of considering each texel as a micropatch, its unprojected center is taken as a vertex. A regular quad mesh is then implicitly defined by these points and their texture-space adjacencies. We make each face serve as a micro-occluder, called *microquad* [333] (see Fig. 4.13). It is created from four vertices corresponding to 2×2 neighboring texels (cf.

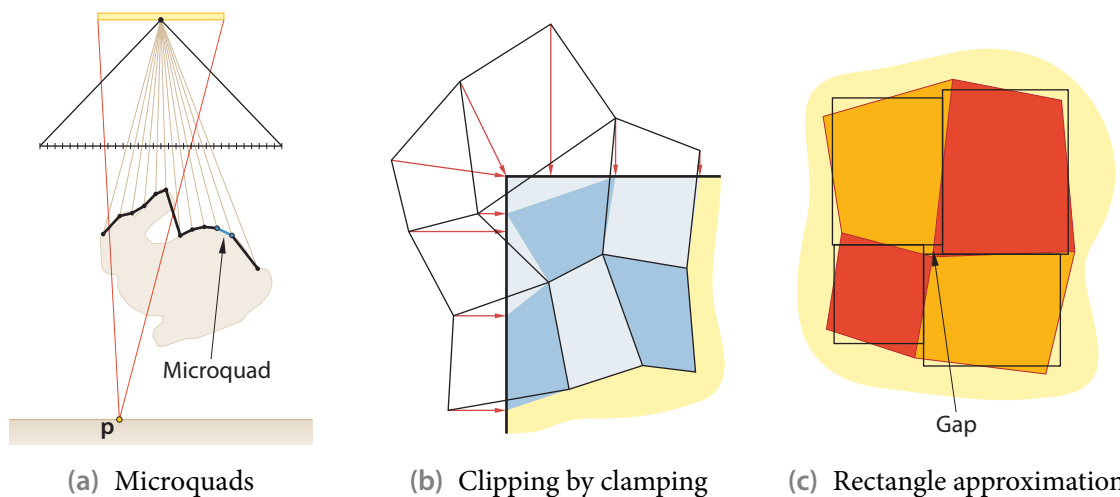


Figure 4.13 (a) Microquads are constructed from unprojected texel centers and provide a piecewise-(bi)linear approximation of the occluders. (b) Backprojected microquads are cheaply clipped to the light area by projecting their vertices onto the nearest boundary edge via clamping. Note that the microquad mesh remains watertight and that no overlaps are introduced. (c) To allow fast bitmask updates, each microquad is approximated with a rectangle defined by the centers of the quad's edges.

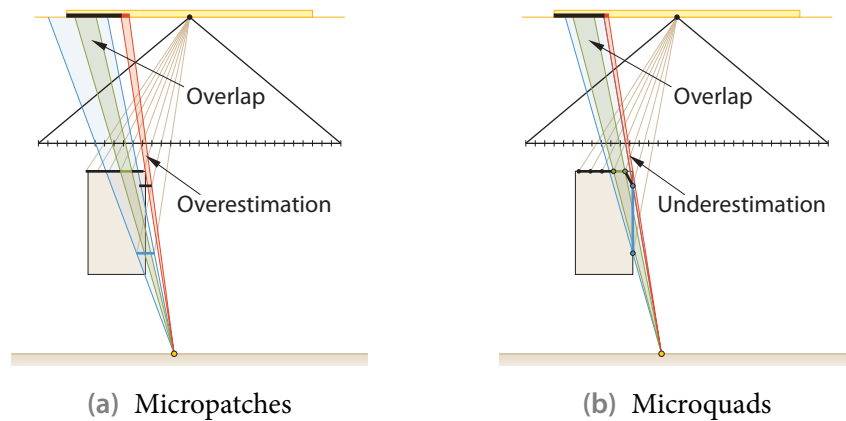


Figure 4.14 While using micropatches (a) often suffers from overestimating the occluders, microquads (b) can lead to some underestimation. Note that with both approximations overlaps can occur as demonstrated by the blue and green primitives.

Fig. 4.12 c), and gets only taken into account during visibility determination if all four vertices are closer to the light source than the point for which light visibility is computed.

Note that this alternative occluder approximation replaces the micropatches' piecewise-constant interpolation with piecewise-(bi)linear interpolation. It is hence not surprising that microquads adapt better to the actual geometry than micropatches (cf. Fig. 4.16). In particular, since adjacent microquads share a common boundary no unwanted gaps occur in the first place, and hence light leaks are avoided.

Another advantage of microquads is that they are less prone to cause surface acne. Referring to Fig. 4.15, consider a plane which is visible from the light's center and not perpendicular to the z axis of the shadow map's frustum. In case of using a micropatch approximation, many points on that plane which don't coincide with a shadow map sample point get partially occluded by the micropatch corresponding to the nearest sample point closer to the light, thus suffering from incorrect self-shadowing. This is especially hard to avoid via biasing (without causing

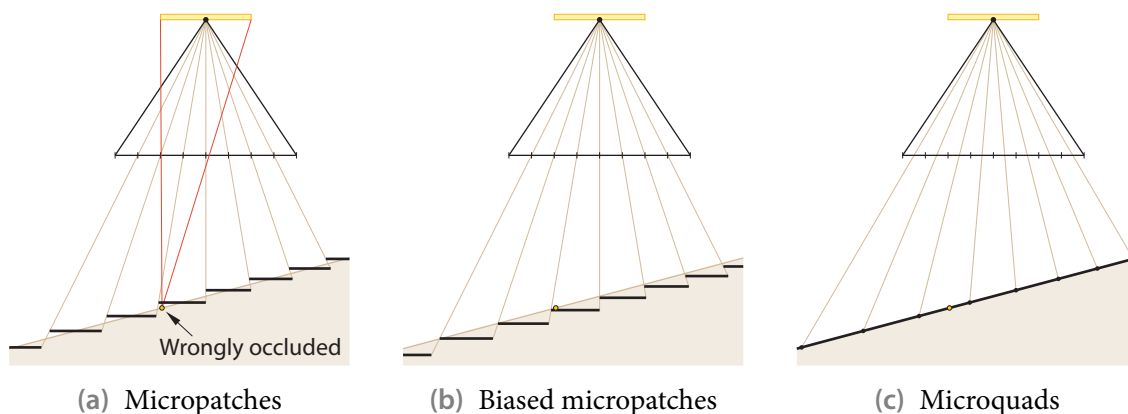


Figure 4.15 Whereas micropatches typically require biasing to avoid self-shadowing artifacts, microquads are less prone to surface acne. They typically provide a better fit to the underlying geometry because they constitute a piecewise-(bi)linear approximation instead of a piecewise-constant one like micropatches.

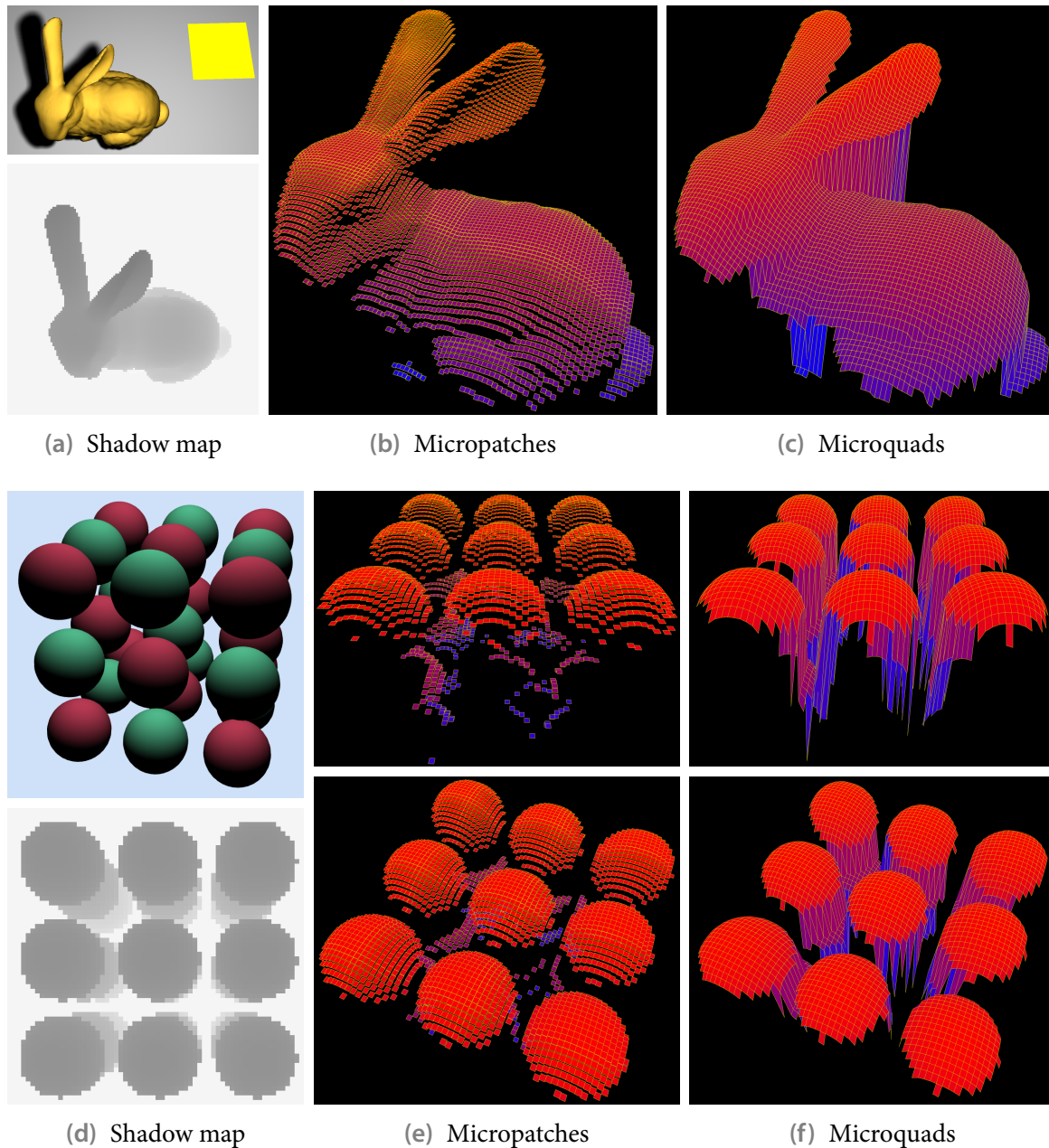


Figure 4.16 Visualization of the micropatches and microquads constructed from a coarse shadow map for two example scenes. Note that micropatches are typically extended on the fly to close gaps between adjacent micropatches.

artifacts in other parts of the image) if patches from coarser hierarchy levels get considered (see also Sec. 4.5.4). Microquads, on the other hand, don't cause any artifacts in this setup.

However, because a quad's backprojection onto the light area does not yield an axis-aligned rectangle in general, correct clipping and area determination as well as occlusion bitmask updates are complicated. Accurate solutions are described below in Sec. 4.4.4, but incur some overhead.

A cheap way to clip the backprojection of a microquad to the light area is simply clamping its projected vertex coordinates against the light's extent (see Fig. 4.13 b). While this approx-

imation can introduce minor errors in the covered area, these imprecisions only occur at the boundary of a mesh of connected microquads. In particular, no gaps or overlaps occur due to this clamping step because the vertices are adapted consistently across microquads.

If light visibility is derived by area accumulation, the area covered by a clipped quad is then easily computed by decomposing the quad into two triangles, calculating their (signed) areas and adding these together.

4.4.2 Approximate occlusion bitmasks for microquads

While employing microquads in lieu of micropatches offers many advantages, artifacts due to overlapping micro-occluders can still occur (cf. Fig. 4.14). Using occlusion bitmasks for visibility determination is hence desirable, and being able to quickly derive bitmasks for backprojected microquads proves important. Because an accurate solution for arbitrary quads is rather expensive, as mentioned before, we developed some fast approximation.

After clipping a microquad's projection via clamping, we approximate the resulting quad by fitting an axis-aligned rectangle to the center points of the quad's edges. While this alleviates area overestimation compared to using the quad's bounding box, it breaks the connectedness of diagonal neighbors, thus introducing minor gaps and overlaps (cf. Fig. 4.13 c). Note that the same problem occurs when filling gaps via micropatch extension (and not accounting for the diagonal neighbor).

Despite these simplifications, dealing with microquads is slightly more expensive than operating with micropatches. This is largely due to more complex computations and not because of an often marginally increased number of required texture fetches. As when using micropatches and performing gap filling, for each microquad, usually only two new texels have to be accessed because the remaining ones are known from the previously processed micro-occluder.

4.4.3 Microtris

Recall that when light visibility for a certain point \mathbf{p} is determined, a microquad is only backprojected if all of its four vertices are closer to the light source than \mathbf{p} . Consequently, the occluder geometry is potentially not that well approximated and its extent probably underestimated if only three of the four considered vertices pass the distance test. To improve results in these cases, the triangle defined by the three closer vertices may be taken as an additional micro-occluder, termed *microtri* [335] (cf. Fig. 4.12 d).

Analogous to microquads, the backprojection of a microtri may cheaply be clipped against the light source by clamping its vertex coordinates. Determining the covered area is then trivial, whereas deriving an according bitmask is less straightforward. In particular, an approximation by an axis-aligned rectangle is not a reasonable option for triangles, unlike with microquads. Therefore, we advocate using the accurate procedure described in the next subsection when employing occlusion bitmasks for visibility determination.

4.4.4 Exact occlusion bitmasks for microquads and microtris

Deriving the exact area or occlusion bitmask for a backprojected microquad raises the question of which light region to take as being actually occluded by the projection. The obvious option of adopting a quad's interior as its occlusion footprint turns out to be problematic. As illustrated in

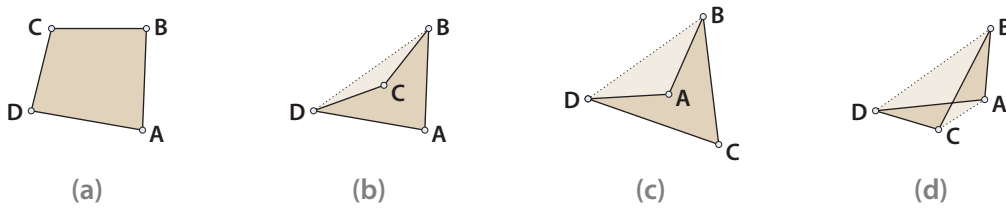


Figure 4.17 Various encountered cases when backprojecting a microquad $\square ABCD$ onto the light source. The dashed lines indicate the convex hulls.

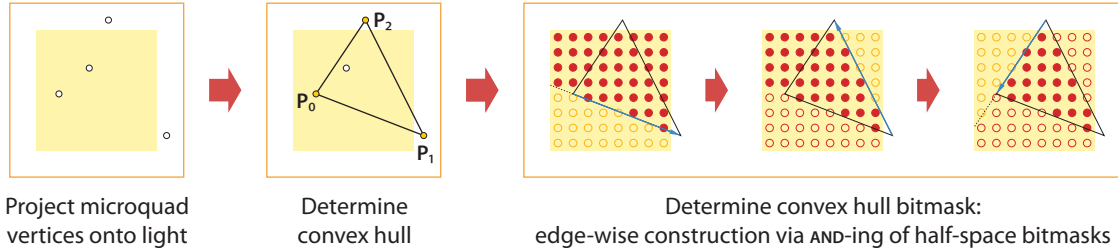


Figure 4.18 Overview of the steps involved in deriving an accurate bitmask for a microquad from its backprojected vertices.

Fig. 4.17, several different configurations can arise when backprojecting a microquad $\square ABCD$ onto the light source. While selecting the dark brown area as the occluded region might seem reasonable in cases a–c⁸, the fold-over setting in Subfig. d questions the suitability of this choice.

Therefore, we settle for the alternative of taking the convex hull of a microquad’s backprojected vertices as occlusion footprint. This is motivated by reasoning that a microquad should have the same footprint as the union of all of its microtris. Most notably, if one vertex is no longer closer to the light than the considered point and hence only a microtri in lieu of a microquad is backprojected, then the region occluded by this micro-occluder should cover no additional parts of the light compared to the footprint of the corresponding microquad.

Our general approach to exactly determine a microquad’s bitmask, outlined in Fig. 4.18, therefore starts with deriving the convex hull in counter-clockwise (CCW) orientation. In order to derive the bits corresponding to occluded light sample points, we employ a *convex hull bitmask* initialized to all bits set. For each oriented edge of the convex hull, we determine the bitmask for the corresponding half-space and perform a bitwise AND with the convex hull bitmask. Finally, the occlusion bitmask is updated to incorporate the microquad’s occlusion footprint by OR-ing it with the convex hull bitmask.

Note that correct clipping against the light source is automatically performed by the edge-wise convex hull bitmask updates. Moreover, microtris are directly supported because their backprojection, a triangle, is just one of the possible convex hull shapes.

Convex hull determination

In the first step, we determine the convex hull of the micro-occluder’s backprojection such that its vertices are enumerated in CCW order. Note that in case of a microquad, the (non-

⁸ Note that splitting the quad into two triangles and adding their signed areas correctly yields the area of this region.

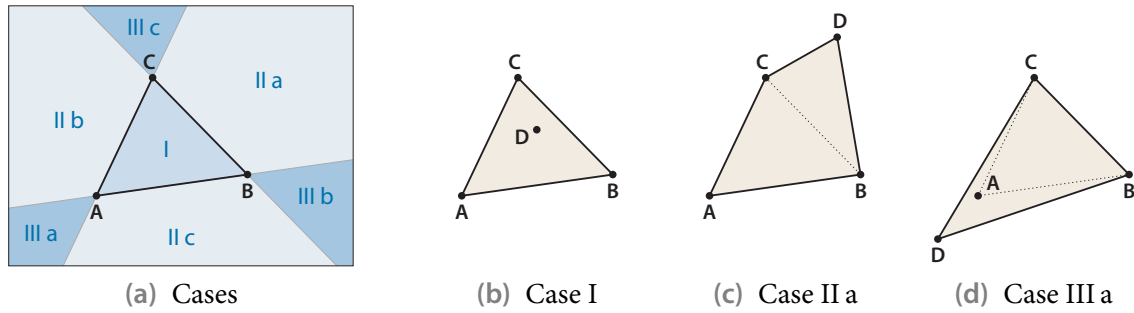


Figure 4.19 (a) Cases for the position of microquad vertex **D** distinguished during convex hull determination. (b–d) An example for each of the three classes is shown.

degenerate) convex hull may be either a quadrilateral or a triangle.

Pursuing a shader-friendly approach, we start with the triangle $\triangle ABC$ made up of the microquad's first three vertices or the microtri, respectively, determine its orientation and swap **B** and **C** if it is clockwise. In case of a microtri, the resulting triangle already constitutes the convex hull and nothing remains to be done. Otherwise, we compute the barycentric coordinates of vertex **D** with respect to the triangle, to identify which of the seven cases depicted in Fig. 4.19 holds, and choose the convex hull accordingly. If necessary, the vertices are finally reordered to ensure a CCW order.

Convex hull bitmask construction via half-space bitmask lookup texture

Subsequently, for each edge of the convex hull, a bitmask for the corresponding half-space is determined; these are combined with bitwise AND, yielding the convex hull bitmask. One possibility for deriving the half-space bitmasks is to employ a lookup texture storing precomputed bitmasks for a number of half-spaces. Recall from Sec. 4.2.1 that such an approach puts no restrictions on the placement of light sample points, and hence naturally supports random distribution patterns.

More precisely, similar to Eisemann and Décorêt [107], we parameterize a half-space by the Hough transform [99] of its defining edge (line), i.e. its angle θ and signed distance r to the origin. In a precomputation step, we regularly sample (θ, r) from the relevant domain $[-\pi, \pi] \times [-\sqrt{2}, \sqrt{2}]$, determine the corresponding bitmask values and store them in a lookup texture. During visibility computation, the pixel shader computes the Hough parameters, queries the lookup texture (with circular wrapping for θ and clamping for r) and updates the convex hull bitmask, for each edge of the convex hull.

Concerning the required resources, a two-channel UINT32 texture suffices for 8×8 light sample points, whereas for 16×16 points a four-channel UINT32 texture array with two array slices is needed, and 32×32 points even necessitate eight array slices and hence eight texture fetches per edge of the convex hull. To keep the shader register count to a minimum, we refrain from maintaining a complete representation of the convex hull bitmask for cases with more than 128 sample points, i.e. where multiple texture fetches are performed per edge. Instead, we only employ a uint4 variable as working set, and by appropriately interleaving instructions both construct the convex hull bitmask and update the occlusion bitmask in groups of 128 sample points.

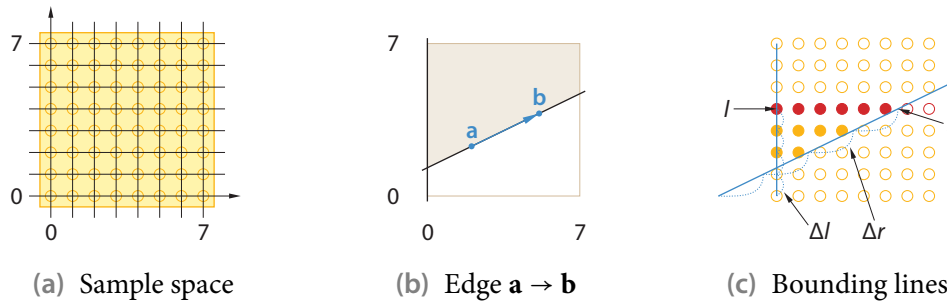


Figure 4.20 Light sample points are placed at integer positions (a) for the exact bitmask computation of the half-space defined by an edge of the convex hull (b). After determining bounding lines (specified by l , Δl and r , Δr) that enclose the region covered by the half-space, the bitmask is created row-wise (c). All bits within the range $[l]$ to $[r]$ (excluded) are set before proceeding to the next row, incrementing l and r by Δl and Δr , respectively.

Exact convex hull bitmask computation

Another option to determine half-space bitmasks is to compute them directly, thus avoiding using a lookup texture and the entailed minor inaccuracies due to discretizing the half-space parameters. Moreover, since the gap between offered computational power and available memory bandwidth is widening, keeping a shader's arithmetic intensity high can help performance, especially in the long run. However, deriving a convex hull's bitmask merely via computations is currently clearly slower than employing a lookup texture. In particular, it is way too expensive to loop over sample positions and check for each whether it is contained in the convex hull. On the other hand, for regular sample point patterns of size $n \times n$, bitmask updates are already feasible at interactive frame rates.

More concretely, if the sample points are located at integer positions (i, j) , $0 \leq i, j \leq n - 1$, it is possible to efficiently determine the half-space bitmask for an edge of the convex hull by processing a whole row of bits at a time (see Fig. 4.20). To this end, we transform the edge vertices into sample space and derive two lines bounding the samples within the half-space, as listed in Fig. 4.21. These lines are specified by inverse slopes Δl and Δr and horizontal coordinates l and r of where they intersect the current row. We then repeatedly set all bits within the range $[l]$ to $[r]$ (excluded) and proceed to the next row by incrementing l and r by Δl and Δr , respectively, until the whole bitmask is determined. Note that horizontal edges are readily supported by taking the parameters for a slightly tilted and shifted edge that yields the same bitmask, thus exploiting the regular sample point placement.

4.4.5 Discussion

Micropatches and microquads (and microtris) are two different approximate reconstructions of the occluder geometry captured in a shadow map. Lacking any further information, both kinds of micro-occluder assume that occluder samples directly adjacent in texture space belong to the same macro-occluder surface, i.e. no holes exist where light can pass through. While micropatches have to be appropriately extended to enforce this and hence avoid light leaking, microquads natively realize this assumption. Thanks to providing a piecewise-(bi)linear approximation, microquads have several further advantages over micropatches, like typically providing a better fit to the actual occluder geometry and thus being less prone to cause sur-

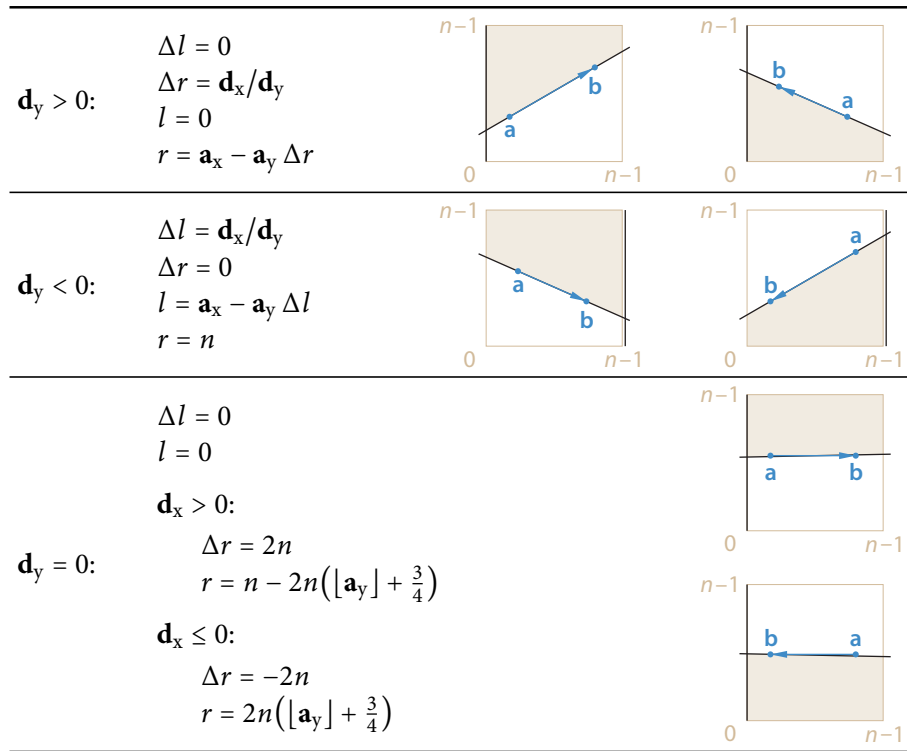


Figure 4.21 Setup of exact bitmask computation parameters for a directed edge $\mathbf{a} \rightarrow \mathbf{b}$ with direction $\mathbf{d} = \mathbf{b} - \mathbf{a}$.

face acne. Since two neighboring microquads are connected by an edge, their backprojections usually don't overlap, which is in strong contrast to the situation with micropatches as these are just isolated primitives. Nevertheless, overlaps can still occur (cf. Fig. 4.14) and hence utilizing occlusion bitmaps is advisable for high quality. While exact processing of microquads is rather expensive, our fast approximations render them competitive to micropatches concerning speed without severely affecting quality.

Because microquads are only backprojected if all four vertices are closer to the light source than the currently shaded point, using them causes considered occluders to be bordered at texel centers in texture space. On the other extreme, micropatches assume sampled occluders to cover the whole texels. Consequently, microquads have a tendency to underestimate occluders, while micropatches suffer from overestimating them. Note that since micropatches are always parallel to the light plane and hence typically do not align well with the approximated occluder surface, the overestimation can be pretty severe but may also turn into an underestimation after backprojection.

On the other hand, potentially overestimating an occluder's extent helps capturing fine structures. By contrast, microquads miss thin geometry like twigs and branches covering only a single shadow map texel in width. The amount of underestimating the size of occluders can be reduced by augmenting microquads with microtris. While they typically produce shadows closer to the reference solution, their visual impact is usually subtle and not easily recognizable, as demonstrated by the example in Fig. 4.22, where smoothness is improved at features lying diagonal in shadow map space.

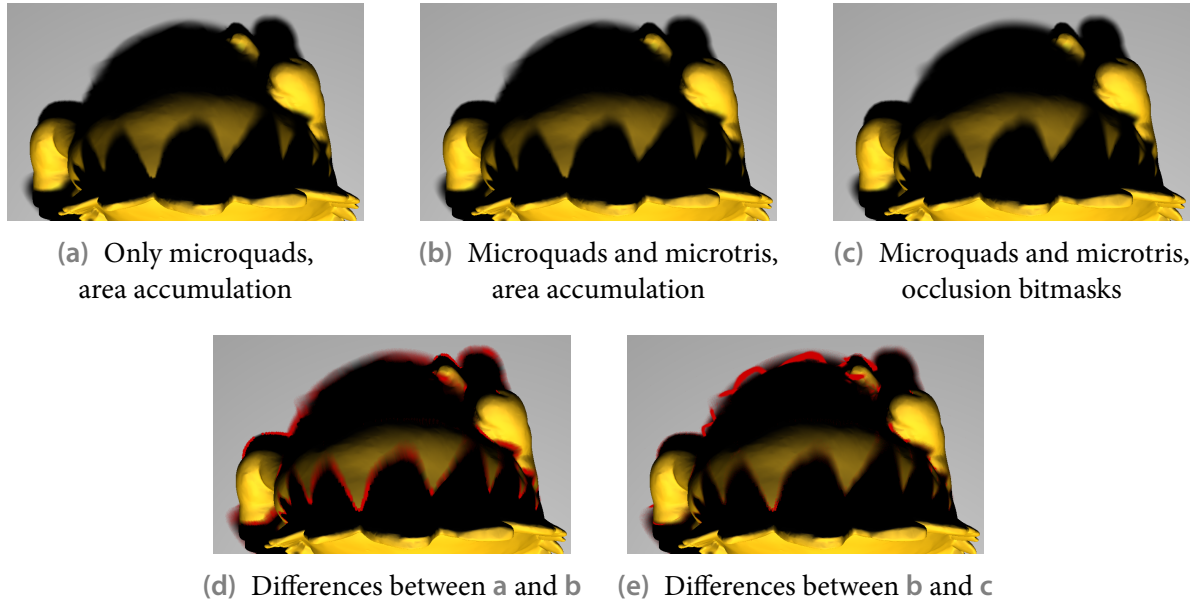


Figure 4.22 Comparison between employing just microquads (a) and additionally using microtris (b), as well as between simple area accumulation (b) and accurate visibility processing with occlusion bitmaps (c). To ease the recognition of differences, affected regions are highlighted in red (d, e) and merely a size of 512^2 was used for the shadow map.

Occluder contours

While microquads and microtris are rather different from micropatches, other occluder approximations devised are directly based on micropatches. For instance, Bavoil and Silva [31] employ the bounding sphere of a micropatch as occluder, and compute the subtended solid angle to determine visibility of a spherical light source.

Guennebaud et al. [148] construct an *occluder contour* for each connected region of shadow map texels passing the depth test (cf. Fig. 4.12 e). To this end, they slide a window of 2×2 adjacent texels across the search area, and for each window position employ the corresponding binary depth test results to consult a lookup texture for deriving a set of oriented edges which together ultimately form the contours. Each edge is backprojected and the signed areas of the resulting radial segments with respect to the light center are accumulated to derive light visibility. Because only contour edges have to be backprojected, and their number is typically smaller than the equivalent micropatch count, some computations are saved. However, all shadow map texels within the search area still have to be accessed, nevertheless.

Since a contour encompasses all neighboring shadow map samples passing the depth test, light leaks are implicitly avoided. However, the rule set suggested by Guennebaud et al. for edge construction treats sub-regions which are only adjacent in diagonal direction as being separate, potentially introducing gaps (see e.g. Fig. 4.23 b, bottom). Furthermore, since contours are extracted in 2D instead of 3D space, occluders recorded in the shadow map may be missed (cf. Fig. 4.23 a). This can lead to noticeable popping artifacts as the depth values at the 2D contour may jump when the light moves relative to the occluder (even if the triggering occluder is captured in both the old and the new shadow map).

Finally, contours can be moved inwards or outwards to adapt the extent of the enclosed region. Most notably, when resorting to coarser shadow map levels for constructing the occluder

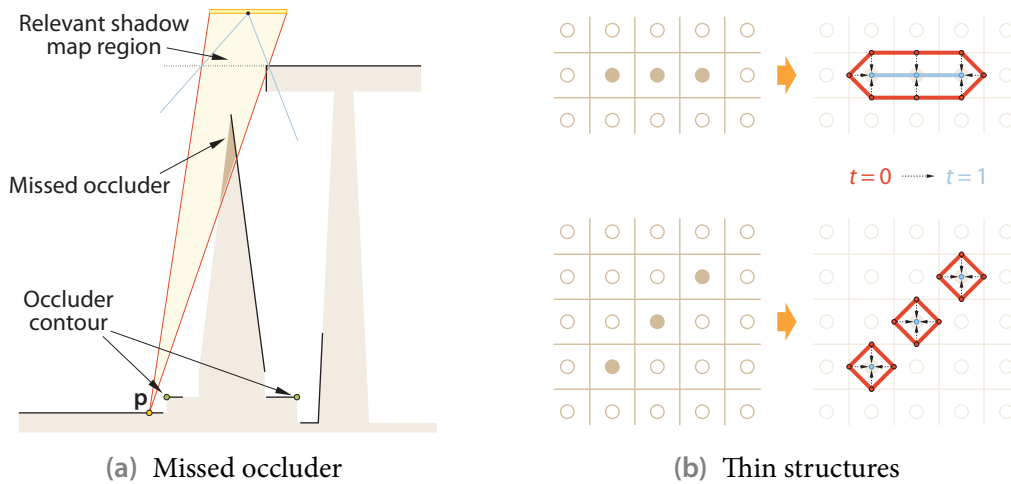


Figure 4.23 Occluder contours are prone to ignoring occluders captured by the shadow map (a), and may miss thin structures if shrunk by setting $t = 1$ (b).

approximation, shrinking them often helps reducing the approximation error (see also the next section). On the downside, this may cause thin structures to be ignored (cf. Fig. 4.23 b).

4.5 Coarser occluder approximations

Since in many scenes the shadow map search area can comprise an excessive number of micro-occluders for a non-negligible amount of pixels, it is reasonable to impose an upper bound on the number of micro-occluders processed per fragment and resort to coarser micro-occluders to satisfy this budget. Recall that the standard approach to derive such coarser approximations is to utilize a coarser shadow map level obtained via minimum reduction (cf. Fig. 4.24). The required multi-resolution representation is readily available from acceleration structures like the HSM, the MSSM or the YSM.

Picking the minimum depth value of 2×2 adjacent shadow map texel as their representative is simple and also conservative in that it ensures that if at least one of the original samples passes the depth test, then the coarser texel does so as well. Generally, this strategy preserves the tendency of micropatches and microquads to over- and underestimate occluders, respectively. In particular, fine structures are implicitly enlarged with micropatches at each coarser level, whereas microquads increasingly miss thin occluders.

Note that while the unprojected center of a level-0 texel actually lies on some occluder surface, this is often no longer the case for coarser texels derived via minimum aggregation. Consequently, vertices of coarser microquads may hover above an occluder instead of being on it, and coarser micropatches can severely protrude the occluder they intersect. This effectively moves the occluder approximations closer to the light source compared to utilizing the finest shadow map level. Depending on the considered shadow-receiving point, the occlusion caused by the corresponding level-0 micro-occluders is hence typically either overestimated or underestimated by coarser micro-occluders (cf. Fig. 4.24 c).

In addition to that, approximation quality is negatively affected by the missing flexibility of the micro-occluders, entailed by their simplicity. Since micropatches have a uniform size in texture space and microquad vertices are uniformly spaced in texture space, with them it is of-

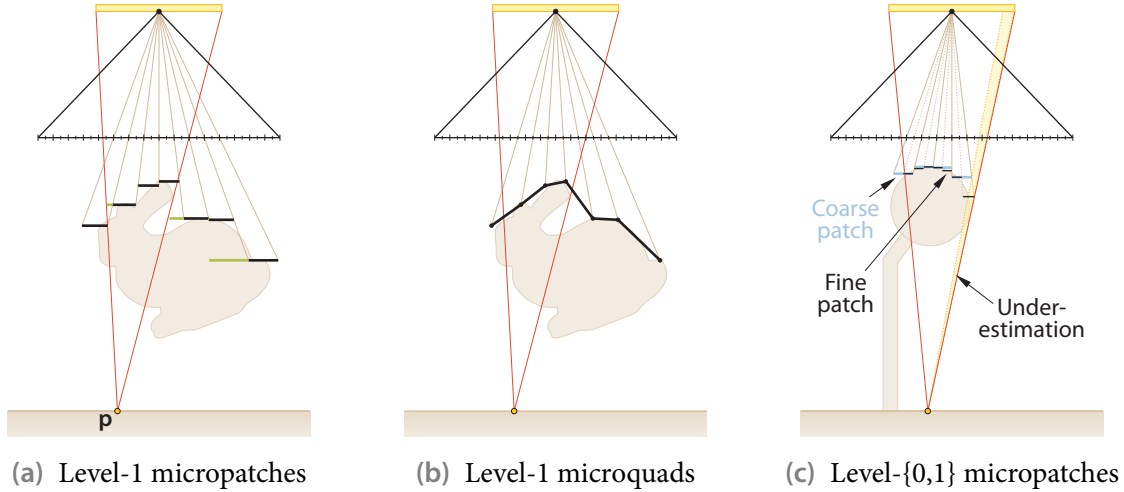


Figure 4.24 Coarser micro-occluders are constructed from a coarser minimum mipmap level of the shadow map. They typically offer a reduced approximation quality.

ten not possible to represent the occluder samples from the original shadow map well at coarser levels. To address this issue and increase the soft shadow quality for a given per-fragment micro-occluder budget, we developed a generalization of the micropatch that enables better fits to the underlying shadow map data at coarser levels. It is covered in the following subsections.

Another source of problems, especially when using a micro-occluder which provides a piecewise-constant approximation, is the depth bias determination for the coarser levels. In particular, simply using the same bias value for all levels can lead to visual artifacts, like surface acne or missing contact shadows. We briefly cover this topic in Sec. 4.5.4.

Finally, note that resorting to a coarser-resolution shadow map level decreases the spatial sampling rate and hence temporal coherence may suffer. Especially due to minimum aggregation, even slightly moving the light relative to the scene objects captured by the shadow map can cause large changes in the derived occluder approximation.

4.5.1 Microrects as a generalization of micropatches

To better represent the sampled occluders at coarser levels, it is crucial to introduce more flexibility in specifying a micro-occluder's extent. As a concrete example, we pick micropatches and generalize them by abandoning the limitation of uniform size in texture space. Since the resulting new kind of micro-occluder can cover varying rectangular regions, it is referred to as *microrect* [336].

Note that each texel of shadow map level i completely defines a micropatch (cf. Fig. 4.25 a): it has a texture-space reference point (the texel center), a fixed size corresponding to one level- i texel (or equivalently $2^i \times 2^i$ level-0 texels), and an associated light depth value. It represents the occluders captured by the related $2^i \times 2^i$ level-0 texels of the shadow map.

Microrects maintain the 1 : 1 relationship between texels and micro-occluders. However, at coarser levels $i > 0$, their texture-space extent is no longer restricted to a single level- i texel but can be any rectangular region of level-0 texels subject to the following two constraints: First, it must contain the microrect's reference point, which is obtained by subsampling the reference points from level $i - 1$. Second, the extent may not contain the reference point of any other level- i microrect, thus imposing a maximum on it as illustrated in Fig. 4.25 b. These two

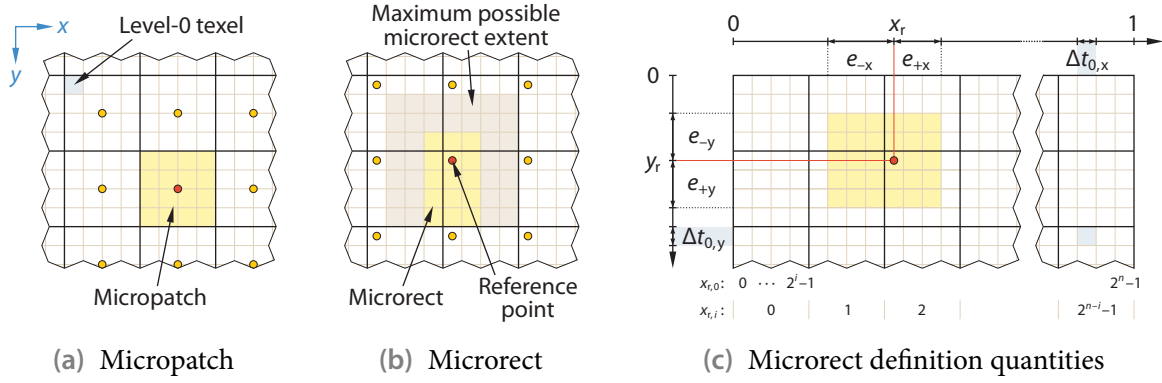


Figure 4.25 Microrects generalize micropatches. The shown examples are from level $i = 2$; for the quantities used in defining a microrect, the one stored in texel $(2, 1)$ is highlighted.

requirements ensure that an acceleration structure can readily be used to identify a search area of relevant micro-occluders.

More formally, for an input shadow map of size $2^n \times 2^n$, a microrect at level i is defined in 2D texture space by a reference point

$$(x_r, y_r) = \frac{1}{2} \Delta \mathbf{t}_0 + (x_{r,i}, y_{r,i}) \cdot 2^i \Delta \mathbf{t}_0 \in [0, 1]^2 \quad \text{with} \quad x_{r,i}, y_{r,i} \in \{0, \dots, 2^{n-i} - 1\}$$

and an extent

$$(e_{-x}, e_{-y}; e_{+x}, e_{+y}) \quad \text{with} \quad e_{\pm\star} \in \left\{ \left(k + \frac{1}{2}\right) \Delta t_{0,\star} \mid k \in \{0, \dots, 2^i - 1\} \right\},$$

where $\Delta \mathbf{t}_0 = (\Delta t_{0,x}, \Delta t_{0,y})$ denotes the texture-space extent of a single texel in the finest shadow map level ($i = 0$). The microrect covers the axis-aligned rectangular region given by the vertices $(x_r - e_{-x}, y_r - e_{-y})$ and $(x_r + e_{+x}, y_r + e_{+y})$, as shown in Fig. 4.25 c. Note that the special case $e_{\pm\star} = 2^{i-1} \Delta t_{0,\star}$ yields a micropatch.

Since the microrects of levels $i > 0$ can have varying extents as well as overlap, both their associated depth values and their extents no longer can directly be obtained from the acceleration structure. We hence store them in two additional textures with full mipmap chains, i.e. the gained flexibility comes along with an increased memory footprint.

4.5.2 Construction of microrects at coarser levels

By construction, the extent and associated depth value of a microrect have to be determined explicitly. To simplify the presentation, we first consider the 1D case, shown in Fig. 4.26. Because microrects at the finest level $i = 0$ are just micropatches, their extent and depth is trivially given by the corresponding level-0 texel. For the remaining levels $i > 0$, the microrect extents and depth values are derived iteratively from the ones of the next finer level $i - 1$. Remember that the reference points of the level- i microrects are obtained by regularly subsampling the ones of the microrects from level $i - 1$. Consequently, we merge every second microrect (e.g. microrect B in Fig. 4.26 b) from level $i - 1$ with one of its neighboring microrects (A and C) to derive the level- i microrects (M and N). We always pick that neighbor as merge partner which features the least depth value difference (here: C). The extent of the resulting level- i microrect (N) is given by the union of the extents of the involved finer-level microrects (B and C); its depth is derived by taking the minimum of these microrects' depth values.

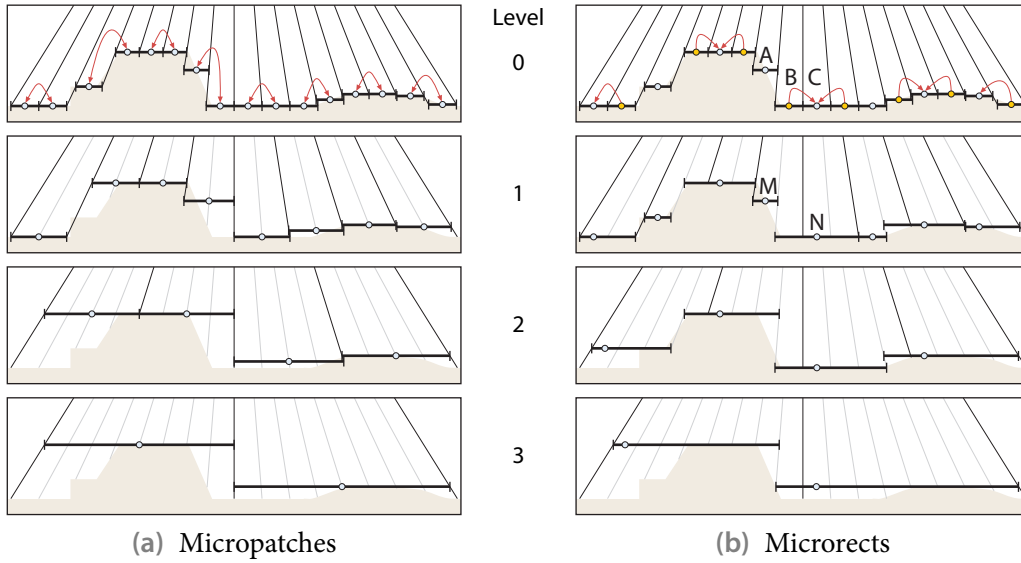


Figure 4.26 Example of 1D micropatches and microrects for successive levels.

Unfortunately, the 2D case is much more involved because a microrect has neighbors in x , y and diagonal x - y direction but a rectangular extent. Therefore, it is in general not possible to merge neighboring microrects such that the union of their extents is still rectangular. To avoid missing any occluder, we hence take the smallest rectangular region encompassing the merged extents as new extent. Note that this can lead to microrects with overlapping texture-space extent. However, disallowing overlaps at all puts a heavy constraint on the microrects' ability to approximate the occluders because many potential microrect merges become prohibited. Consequently, we don't try to avoid overlaps but alleviate their influence on light occlusion by using occlusion bitmasks instead of area accumulation during light visibility determination.

Microrects in level $i + 1$ are derived directly from those in the next finer level i according to a simple and fast greedy approach that is well suited for data-parallel processing and keeps redundant comparisons to a minimum. We basically proceed like in the 1D case for determining the merge partners in x and y direction and treat the diagonal neighbors separately. As before, the depth associated with a microrect is obtained by minimum aggregation. More precisely, the construction comprises the following two passes. Referring to Fig. 4.27, we first consider merg-

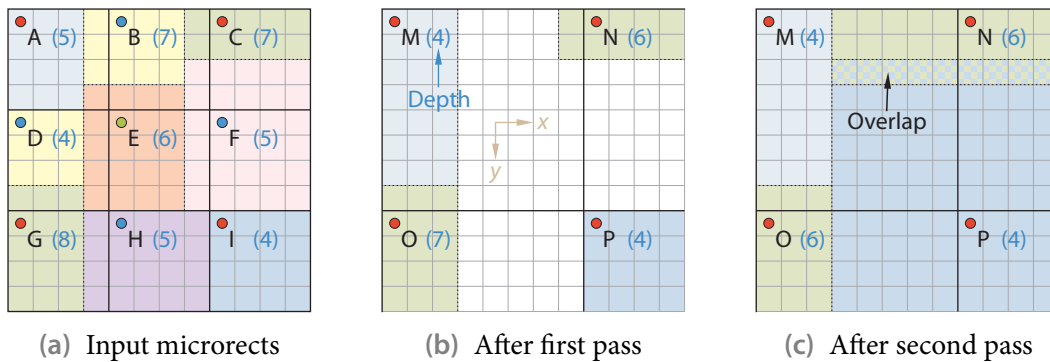


Figure 4.27 Example setup for greedy microrect construction. Each uniformly colored rectangular region corresponds to a microrect, with the contained dot marking its reference point.

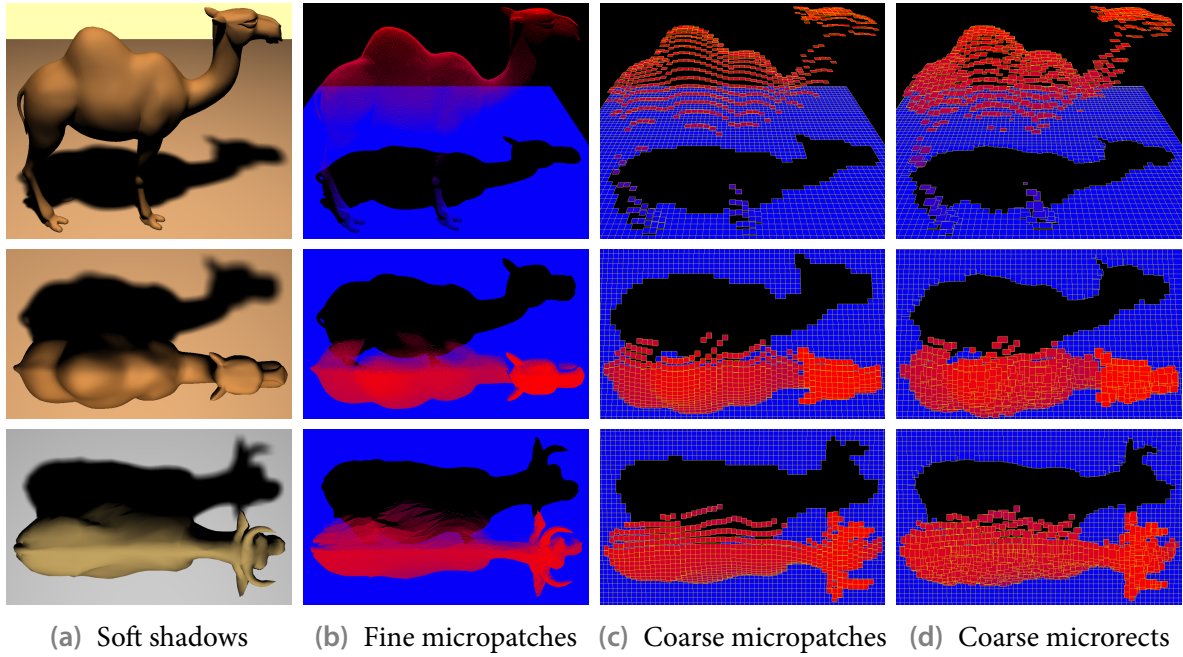


Figure 4.28 Visualization of the micropatches and microrects constructed at level 0 (b) and 4 (c, d) for two example scenes. Note that at the finest level microrects are just micropatches.

ing the microrects retained after subsampling (e.g. microrect A, which becomes M in level $i + 1$) with their neighbors in positive x and y direction (B and D). The decision whether to merge is again based on the depth value differences between the merge candidate and its two involved neighbors (B–A and B–C, as well as D–A and D–G). If a merge is executed and the resulting extent completely contains the diagonal neighbor’s one (E), we further assimilate this microrect. In the second pass, a merge with all those level- i neighbors in negative direction (E, F and H in case of P) is performed which have not been aggregated yet in the first pass.

We note that a global optimization as well as not only taking samples from the next finer but from the finest level may yield better fits than our greedy approach; however, real-time construction becomes rather difficult with more involved methods. Also note that a third pass which locally reduces overlaps is not an option because apart from the extent, the depth obtained via minimum aggregation would also have to be updated.

4.5.3 Discussion of microrects

Microrects vary from and improve on micropatches at coarser levels. The visualizations in Fig. 4.28 nicely demonstrate the main characteristics and differences. Looking at the micro-occluders covering the ground plane (the blue-colored ones), it is obvious that microrects approximate the finest-level samples better than micropatches. In particular, far fewer ground plane samples are aggregated with camel and cow samples in case of microrects, which helps keeping occluder overestimation small. However, while the shadow map region covered by the represented samples is a polyomino [141] (a collection of simply connected squares), a rectangle enclosing this region is used as microrect extent for simplicity. Therefore, the occluder region represented by a microrect may be overestimated, and adjacent microrects often overlap, necessitating the use of occlusion bitmaps for visibility determination. By contrast, neigh-

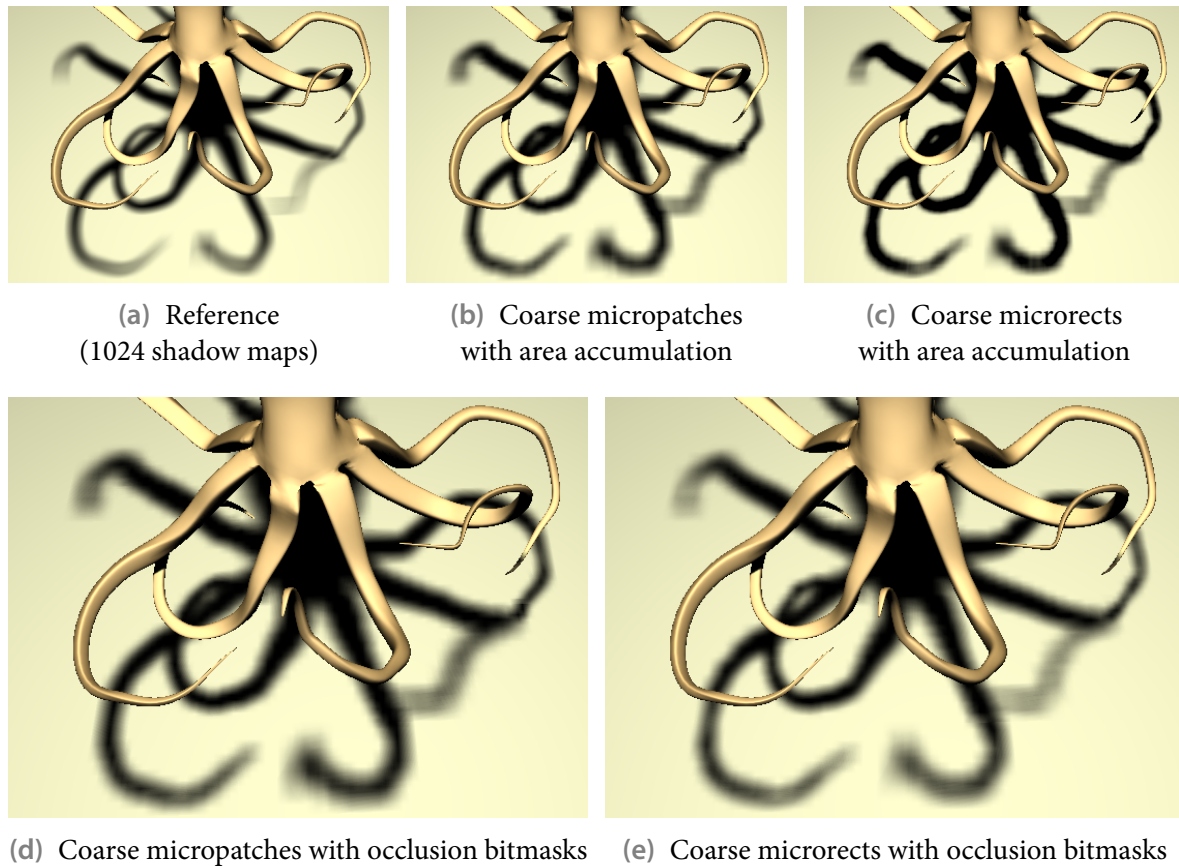


Figure 4.29 When resorting to coarser levels, microrects yield better results than micropatches but necessitate using occlusion bitmaps for visibility determination.

boring micropatches always abut in texture space. Nevertheless, their backprojections often overlap, especially at coarser levels, and hence performing correct micro-occluder fusion is reasonable for them, too.

Thanks to typically more accurately approximating the underlying occluder geometry, microrects yield superior results compared to micropatches when resorting to coarser levels, as shown in Fig. 4.29. We further observe that transition artifacts between pixels using different levels for micro-occluder construction (see also Chapter 5) are often less pronounced than in case of micropatches (cf. Fig. 4.30).

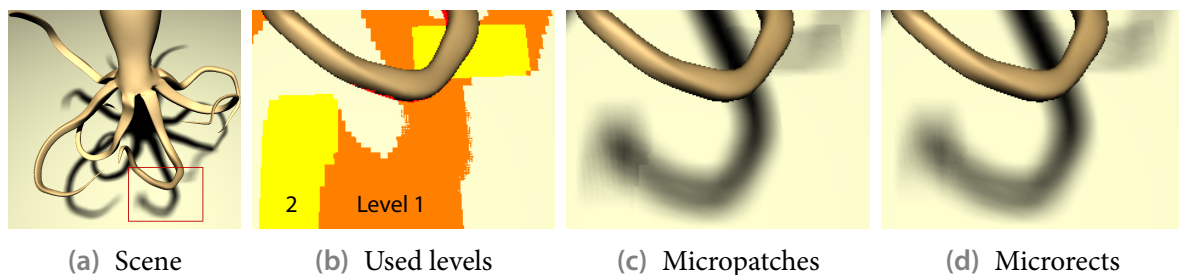


Figure 4.30 When micro-occluders are constructed at different levels across penumbra pixels (b), microrects (d) often cause less noticeable transition artifacts than micropatches (c).

Apart from essentially precluding the use of simple area accumulation for visibility determination, microrects also incur additional overhead with respect to micropatches. Because the extent is not uniform, it has to be derived, stored, and fetched explicitly. Moreover, the associated depth value can no longer be taken from an acceleration structure like the HSM but must be computed and stored separately. As a concrete example, the construction of the two microrect textures (containing extent and depth) consumes 1.89 ms for a 1024^2 -sized shadow map on an NVIDIA GeForce 8800 GTX. Further note that in the employed pixel shader the number of instructions executed per micro-occluder rises slightly and the register count is marginally increased. Consequently, the improved visual quality comes along with a somewhat decreased frame rate. Though depending on the actual scene, graphics hardware and driver, in the worst case simply employing four times as many level- i micropatches may even be almost as fast as preparing microrect textures and using microrects from level $i + 1$. In such settings, completely resorting to micropatches can thus prove the better option.

4.5.4 Biasing problems

When light blockers are approximated with a piecewise-constant reconstruction like micropatches, a bias is usually required to avoid self-occlusion.⁹ It is added to the actual depth of a micropatch, essentially pushing the micro-occluder below the surface part it represents (see Fig. 4.15 on page 47), to ensure that any point on this surface part is considered to be closer to the light source when compared against the micropatch. The amount of bias necessary for a certain micropatch depends on the scene geometry that is visible to the shadow map origin and projects into the shadow map texel corresponding to the micropatch. In practice, only the triangle sampled by the center of this texel is considered and a function of its depth slope is used to derive the bias.

Note that while a biased depth value is required during depth comparison with the point for which light visibility is determined, the unbiased depth should be used for constructing the actual micropatch that gets backprojected. Otherwise, the occluder represented by the micropatch is not correctly accounted for because an offset in depth changes the backprojection on the light source and hence the caused occlusion. Since the needed bias is specific to each micropatch, it has to be stored explicitly in addition to the depth in the shadow map. Alternatively, to avoid this overhead, a constant bias may be employed for all micropatches, accepting that it is typically not appropriate for all of them. At the same time, a certain fixed bias value can be too small for some micropatches, causing surface acne, and too large for other micropatches, resulting in missed close-by occluders and hence making contact shadows appear detached.

The bias problem is further exacerbated when using coarser levels and deriving these via minimum aggregation. Since a coarser micropatch represents a larger surface part and this typically spans a bigger depth range, a larger bias becomes necessary to move the micropatch below the surface. Again, one may simply employ a constant bias, which, however, should be specific to each level. Note that the popular but incorrect approach of adding a bias to the depth value when creating the shadow map precludes adapting the bias at coarser levels (except for adding a constant).

Better results are obtained if a bias is explicitly stored for each micropatch. To derive these bias values for coarser micropatches, a simple heuristic may be employed. We first assume that a level-0 micro-occluder at depth z and with bias b represents a section of a slanted plane covering

⁹For robustness reasons, a constant marginal bias is always advisable because limited numerical accuracy may cause even logically identical points to differ slightly.

the depth range $[z - h, z + b]$, where $h = b$. Then, two neighboring micropatches with depths z_1 and z_2 as well as bias values b_1 and b_2 are considered to belong to the same surface if their depth ranges overlap. In this case, their aggregation with new depth $z = \min_i \{z_i\}$ is assigned a bias $b = \max(|z_1 - z_2| + b_f, b_c)$, where $c = \arg \min_i \{z_i\}$ identifies the closer of the two patches and $f = 3 - c$ the farther one. Since the depth range covered by the new micro-occluder is no longer centered at z , the height $h = \max(h_c, h_f - |z_1 - z_2|)$ must be derived and stored separately. On the other hand, if the depth ranges of the two adjacent micropatches don't overlap, bias $b = b_c$ and height $h = h_c$ are just taken from the closest of them. This essentially enlarges the closer micropatch and causes the farther one to be shadowed by it. In practice, the sketched procedure has to be done with all four level- i micropatches that are aggregated to a micropatch of level $i + 1$.

For simplicity and following the reasoning presented above, we decided to employ just a constant bias in our implementation. However, it is important to be aware of the resulting limitations and to choose the concrete bias carefully.

4.6 Visibility interpolation for multisample support

Overall visual quality of rendered scenes can often profit from using multisample antialiasing (MSAA) [8]. Here, several multisamples are distributed across a pixel, and when rendering a primitive, the rasterizer determines which of them are covered by this primitive as well as the associated depth values. The pixel shader, however, is only invoked for a single sample per pixel (typically the center). The resulting color value is then assigned to all covered multisamples. Since potentially expensive shading computations are hence performed at the same frequency as in a standard single-sample setup, and recent graphics hardware features fast support for deriving and dealing with multisamples, typically almost no performance overhead is incurred by employing MSAA.

Incorporating soft shadows in such multisample settings can, in principle, directly be done by putting both shading and all soft shadow mapping computations into a single big pixel shader that gets executed when rendering the scene. Note that an initial depth-only rendering pass should be performed to avoid carrying out expensive soft shadow computations for finally overdrawn fragments. However, this straightforward approach usually leads to a low utilization of GPU resources because of little effective concurrency, especially in case of highly tessellated scenes, and hence entails significantly increased frame times. Moreover, it is not possible to decompose soft shadow computations into multiple steps and use intermediate values from neighboring pixels, as required by advanced schemes like the one presented in Sec 5.2.

Alternatively, a deferred shading approach may be pursued where inputs to the soft shadow computation, like color and world-space position, are multisampled. A pixel shader then determines light visibility for each (unique) multisample and derives the final pixel color. Compared to a single-sample setup, however, this causes soft shadows to be computed multiple times per pixel, and hence runs counter to one core idea of MSAA, the restriction of shader execution to one sample per fragment. Assuming $4\times$ MSAA, for instance, (up to) four times as much work as in the single-sample case has to be done. We hence would have to resort to the next coarser level when constructing micro-occluders, reducing their number by a factor of about four, to roughly maintain the frame rate achieved in the single-sample setup. Also note that decomposing soft shadow computations into multiple steps, though possible now, implicates similar cost increases.

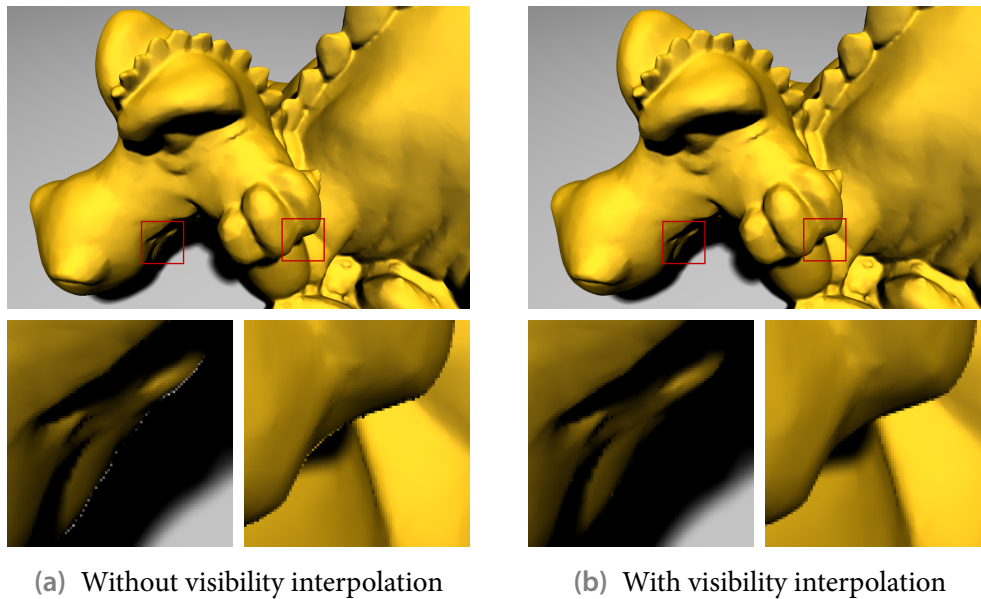


Figure 4.31 Combining multisample rendering with single-sample visibility determination yields good results when visibility is interpolated from neighboring samples.

Since these solutions introduce additional costs far too high to pay for MSAA support, we adopt a different strategy where, in the spirit of MSAA, light visibility is determined only for one sample per pixel and the results are reused for all other multisamples. While we hence accept minor imprecisions at the sub-pixel level, which, however, don't introduce noticeable artifacts, this approach allows maintaining the soft shadow quality known from the single-sample setup without substantially increasing frame time.

At first, the scene is rendered as usual, outputting color and linear (camera-space) depth into multisample buffers.¹⁰ After that, for each pixel, a single multisample value z_c is picked from the multisample camera depth buffer and stored in an ordinary texture. We then run our soft shadow mapping algorithm with these depth values as input, initially reconstructing the corresponding points' world-space positions from them. Note that since exactly one sample gets considered per pixel, this soft shadow computation step is identical to the one in the single-sample rendering case. The resulting light visibility values are written into a visibility buffer, which is subsequently applied to the multisample color buffer with a custom resolve. In this computation of the final pixel color from the multisamples, we interpolate single-sample visibility for those multisamples for which no visibility has been determined explicitly.

More concretely, for all multisamples of a pixel whose camera-space depth difference is within a certain threshold τ with respect to the pixel's multisample chosen for visibility computation, we simply adopt the visibility value of this multisample, assuming they belong to the same surface region. Typically, the vast majority of pixels consist solely of multisamples satisfying this criterion, not least because a pixel's multisamples only differ in value if more than one primitive partially covers the pixel. For the remaining multisamples, which deviate from the pixel's selected z_c value by more than τ , we additionally look at neighboring pixels and take that visibility value whose corresponding camera depth value z_c is closest to the multisample's one.

¹⁰Note that the depth is written by the pixel shader and hence corresponds to the same sample point as for which shading is computed.

As shown by the example in Fig. 4.31, we obtained good visual results when considering a four-neighborhood (direct neighbors in x and y direction) for visibility interpolation and always picking the first multisample of a pixel for the single-sample soft shadow mapping computation. However, note that for extremely fine structures, like twigs with sub-pixel diameter, more sophisticated methods for selecting the representative multisample may be worth exploring.

Concerning performance, measurements over a range of scenes of different complexity suggest that with our scheme multisampling introduces an overhead of less than 9% (on an NVIDIA GeForce 8800 GTX). Note that multisampling itself, i.e. without applying our technique, already accounts for an impact of about 5% on the frame time. Consequently, soft shadows for multisample rendering become possible at low extra cost compared to single-sample setups with our visibility interpolation approach.

4.7 Results and conclusion

Adopting the basic soft shadow mapping approach as starting point, we introduced techniques in this chapter which improve the visual quality and performance of soft shadows. Most notably, we presented occlusion bitmasks as a robust and correct solution to the fundamental occluder fusion problem. Recall that this method is not restricted to soft shadow mapping but is applicable to soft shadow algorithms in general.

As demonstrated in Fig. 4.32, overocclusion artifacts due to overlapping micro-occluders, which are often experienced when deriving visibility with simple area accumulation, are completely avoided by our bitmask soft shadows. Consequently, the visual quality is significantly enhanced, with the produced soft shadows being reasonably close to the reference. Further improvements may be achieved by considering additional shadow maps for micro-occluder construction, which becomes possible thanks to occlusion bitmasks.

Nevertheless, some deviations from the exact soft shadows typically remain owing to the approximations made for performance reasons. First, instead of the actual occluder geometry only micro-occluders extracted from a shadow map (or multiple maps) are employed to derive light visibility. They often merely capture a subset of the real occluders and even these parts are only approximately reconstructed from taken samples. In principle, increasing the shadow map resolution and acquiring additional shadow maps can reduce related approximation artifacts. Another obstacle is the assumption made that micro-occluders adjacent in shadow map texture space belong to the same macro-occluder. Motivated by the lack of further information, this conservative choice ensures that no light leaks occur. But, on the other hand, it may also wrongly fill actual holes between two occluders and hence introduce a non-existent occluder, causing overocclusion. Finally, discretization artifacts may arise, since the visibility is determined by point sampling the light source. Again, increasing the number of samples can ultimately alleviate such problems.

Another ingredient affecting visual quality is the way micro-occluders are constructed from the shadow map. Our microquad approximation is typically superior to the simpler micropatch variant, as shown in Fig. 4.33. Offering a piecewise-(bi)linear interpolation and hence causing the backprojections of adjacent micro-occluders to abut, microquads result in considerably fewer overlaps than micropatches in the first place. Consequently, overocclusion artifacts due to overlapping micro-occluders are greatly reduced compared to micropatches when determining visibility via area accumulation. In case of occlusion bitmasks, microquads are signif-

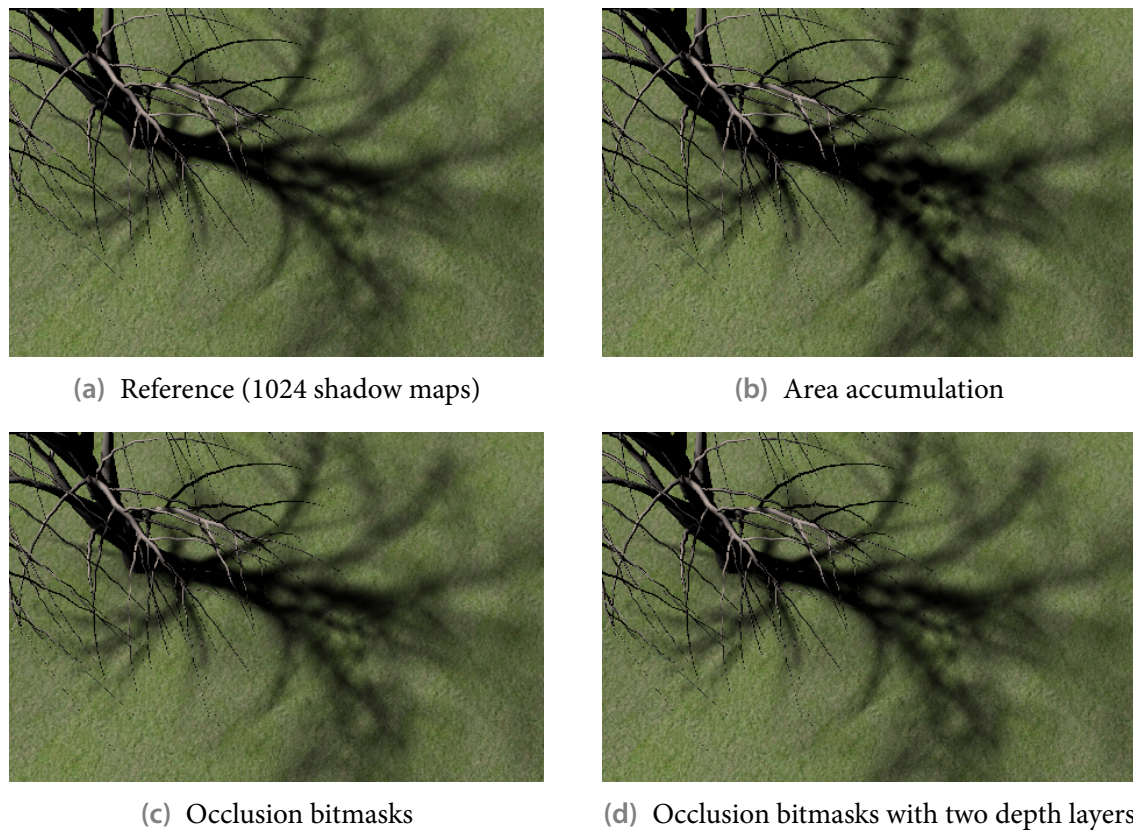


Figure 4.32 Area accumulation causes overocclusion artifacts due to incorrect occluder fusion. By contrast, occlusion bitmaps produce results which are reasonably close to the reference. These can be further improved by using a second shadow map as input to account for previously missed occluders.

icantly more complex to process but our cheap approximations yield results which usually are hardly discriminable from the accurate solutions. Since microquads typically provide a better fit to the underlying occluder geometry, the resulting soft shadows are closer to the reference than in case of micropatches. The tendency of microquads to somewhat underestimate the extent of the represented occluder, contrasting the micropatches' more pronounced bias towards overestimation, can be alleviated by augmenting them with microtris, thus further improving approximation quality.

Concerning performance, note that the actual frame rate not only depends on the graphics hardware, the viewport and the shadow map size, but also on the scene content and camera placement. In particular, the geometric relations of receivers, occluders and light source directly influence the number of micro-occluders that affect light visibility for a certain pixel. If the corresponding search area gets extremely large, comprising tens of thousands of shadow map texels, the attained performance will become very low, eventually dropping even below interactivity. Therefore, to retain real-time frame rates, in practice the number of considered micro-occluders is capped by imposing an upper limit, and in order to meet this budget, a coarser level is employed for constructing fewer but larger micro-occluders. It is also imperative to avoid micro-occluder processing for points which lie in umbra or completely lit regions and to keep the search area tight for the remaining points. Acceleration structures like our YSM are central towards achieving such a high work efficiency.

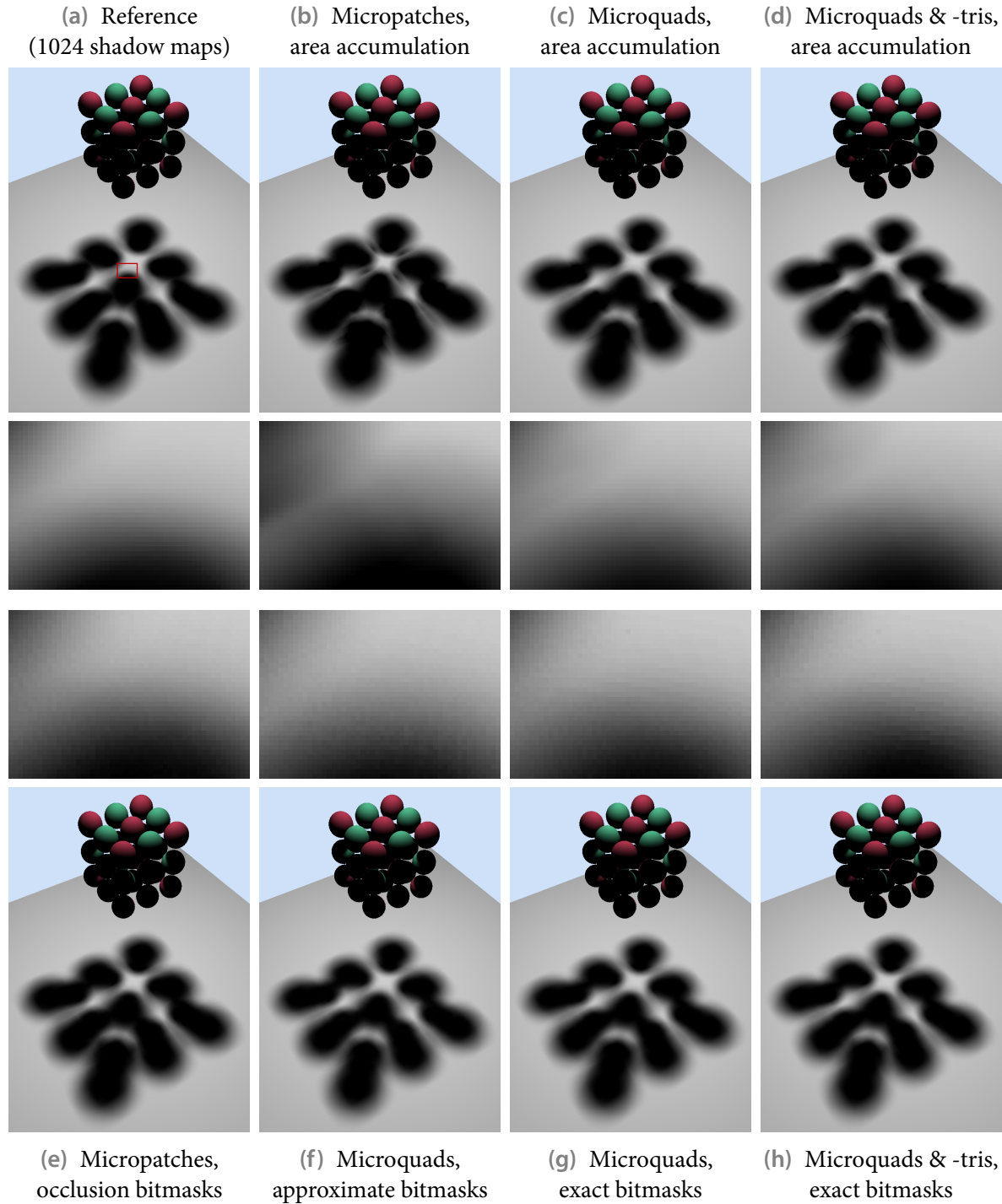


Figure 4.33 A grid of 3^3 spheres and its soft shadows rendered with various methods. While simple area accumulation yields visible artifacts due to overlapping micro-occluders, especially when using micropatches as occluder approximation, these are avoided by occlusion bitmaps (the jittered 16×16 sampling pattern was employed).

To provide some rough idea of the attainable performance, we consider the three example scenes shown and characterized in Table 4.2. For each of them, frame rates for various combinations of employed micro-occluders, used visibility determination approach and utilized ac-

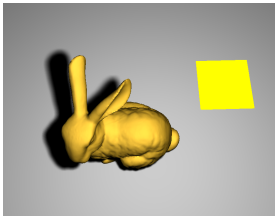
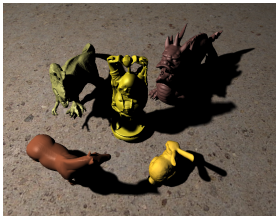

| | | | |
|------------------------|---|--|---|
| |  |  |  |
| | Example I (Fig. 4.16) | Example II | Example III (Fig. 4.10) |
| Viewport | 1024×768 | 1024×768 | 1600×1200 |
| Shadow map size | 1024 ² | 1024 ² | 2048 ² |
| Micro-occluder budget | 256 | 256 | 400 |
| Processed pixels (HSM) | 16.2% | 28.4% | 32.0% |
| Processed pixels (YSM) | 5.1% | 6.4% | 11.6% |

Table 4.2 Three example scenes with soft shadows and the employed configurations. Moreover, the percentage of pixels which are not classified as in umbra or entirely lit and for which hence micro-occluders have to be processed is listed, both for the HSM and the YSM.

| Micro-occluders | Visibility determination | Accel. | Ex. I | Ex. II | Ex. III |
|--------------------|-------------------------------------|--------|--------|--------|---------|
| Micropatches | Area accumulation | HSM | 88 Hz | 56 Hz | 16 Hz |
| Micropatches | Area accumulation | YSM | 122 Hz | 62 Hz | 29 Hz |
| Micropatches | Occlusion bitmasks | YSM | 68 Hz | 38 Hz | 15 Hz |
| Microquads | Area accumulation | HSM | 85 Hz | 53 Hz | 14 Hz |
| Microquads | Area accumulation | YSM | 122 Hz | 62 Hz | 28 Hz |
| Microquads | Occlusion bitmasks (approximate) | YSM | 70 Hz | 39 Hz | 14 Hz |
| Microquads | Occlusion bitmasks (lookup texture) | YSM | 19 Hz | 14 Hz | 4 Hz |
| Microquads & -tris | Area accumulation | YSM | 59 Hz | 36 Hz | 12 Hz |
| Microquads & -tris | Occlusion bitmasks (lookup texture) | YSM | 15 Hz | 11 Hz | 3 Hz |

Table 4.3 Performance for the three example scenes achieved on an NVIDIA GeForce GTX 280. In case of occlusion bitmasks, the jittered 16×16 sampling pattern was used.

celeration structure are reported in Table 4.3. First note that the figures clearly indicate that our YSM is far superior to the HSM. Despite its somewhat larger construction time (cf. Table 4.1 on page 43), the YSM consistently yields a higher frame rate, especially thanks to effectively identifying umbra and completely lit points and thus keeping the number of pixels small for which micro-occluders have to be processed.

As can be expected, using occlusion bitmasks for visibility determination certainly has a negative performance impact. However, it is worth noting that when employed together with the YSM, frame rates are often not that much lower than in case of “classical” soft shadow mapping with HSM-guided area accumulation, while visual quality is considerably improved.

Microquads turn out to be roughly as fast to process as micropatches, not only in case of area accumulation but even with occlusion bitmasks, at least when determining them using the clipping-by-clamping and rectangle approximations. By contrast, deriving the bitmask corresponding to a microquad’s accurate occlusion footprint incurs a significant overhead, mainly

caused by the convex hull determination and the texture lookups for the half-spaces defined by the hull's edges. Given the typically minor resulting improvement in visual quality, we hence strongly advocate always using our approximate scheme unless important reasons justify the additional cost.

Similarly, the augmentation with microtris slightly enhances visual quality compared to solely using microquads, but also considerably reduces performance. Therefore, to maintain real-time frame rates without having to reduce the micro-occluder budget, one usually may refrain from employing this extension. In some cases, however, the gained detail coverage, especially noticeable for small occluders, as well as the improved smoothness of the approximated macro-occluder silhouettes make microtris a viable instrument. Also please note that while it may appear that exact microquad occlusion processing and adding microtris are often a needless luxury, the availability of these options is essential to be able to assess the quality of the approximate occlusion bitmask solution for microquads and of microquads in general, respectively.

CHAPTER 5

Level of quality for soft shadows

As detailed in the last two chapters, the objective of real-time performance currently forces algorithms for rendering soft shadows to impose constraints on the scene that enable simplifications and precomputations, or to introduce approximations. In the latter case, the quality of the produced soft shadows is affected by the kind and degree of the approximation, with its associated cost influencing frame rate.

Ideally, the *level of quality* (LOQ) of the rendered soft shadows is directly related to the spent computational efforts and can be adapted to flexibly meet some prescribed budget like allotted frame time. Since often a lower soft shadow quality is accepted in some regions of a scene than in others, and we typically seek to achieve the highest possible overall perceived visual quality for a given budget, it is further desirable to allow controlling LOQ locally in screen space. Note that to avoid visual artifacts in such settings, the resulting quality transitions have to be sufficiently smooth.

In this chapter, we present one practical LOQ scheme for the concrete approach of soft shadow mapping which addresses these requests. Initially, to explore the solution space and provide motivation for the choices finally made, we discuss possible ways of realizing a LOQ scheme for soft shadows. Subsequently, our actual approach for soft shadow mapping is described. Because this is ultimately based on adapting the shadow map level used for micro-occluder construction, the presented solution also particularly handles the remaining issue of transition artifacts at pixels using different levels (cf. Secs. 4.1 and 4.5.3).

5.1 Possible approaches

Recall from Sec. 2.4 that a common approach taken for geometry is to resort to a coarser level of detail (LOD) in order to meet a given frame time, thus trading visual quality for speed. Most flexibility and adaptability is offered by view-dependent LOD schemes [165, 237], where the geometric detail can be changed locally, allowing finer detail at silhouettes without having to apply the corresponding LOD to the whole model. The transition between two LODs is usually smoothed either by some form of alpha blending [139] or via geomorphing [164]—at least when the difference would be visible otherwise.

While (conceptually) many similarities to such techniques for geometry exist, approaches for adapting the LOQ when rendering soft shadows also face some significant differences. For instance, geometric LOD is associated with and affects only a single scene object or group of objects. By contrast, soft shadows are a global effect that spans and involves multiple scene objects, and thus a corresponding LOQ generally cannot be realized on a per-object basis but

has to be defined in screen space. Moreover, in contrast to geometric LOD methods, which usually heavily depend on them, leveraging offline preprocessing steps is not possible, unless some restrictions are imposed, like requiring a strict partition of the set of scene objects into shadow casters and shadow receivers. Concentrating on the general setting, where such simplifying constraints are foreclosed, we briefly discuss a wide range of possible approaches for soft shadow LOQ in the following subsections.

5.1.1 Multiple algorithms producing different quality

One seemingly straightforward option to tackle soft shadow LOQ is switching the algorithm employed for producing soft shadows, with each LOQ being defined by a distinct algorithm. Unfortunately, such a scheme only allows of discrete LOQ because a smooth transition between the results of different soft shadow algorithms is usually not feasible owing to incompatible assumptions, simplifications and approximations.

For instance, faking soft shadows by just smoothing depth comparison results with percent-age-closer filtering might seem a good candidate for a lower LOQ. Assuming some simple soft shadow mapping variant with micropatches and area accumulation constitutes the next higher LOQ, it becomes important for the PCF approach to spatially vary the filter kernel size to account for non-uniform penumbra widths like encountered in shadow hardening on contact. Although both filtering with screen-space-local kernel sizes and heuristically deriving these sizes have recently become possible at high frame rates [11], a smooth transition to soft shadow mapping is still not feasible in general. Apart from the employed heuristic being far from robust, this is mainly because of fundamentally conflicting ways of attributing occlusion to a shadow map texel. Recall from Sec. 3.2.1 that the PCF approach only takes into account the result of depth comparisons but not whether the used samples actually occlude the light or get projected next to it. Also the implicit assumption that the fraction of the light source that is occluded by a sample is given by its PCF filter weight is incompatible with micro-occluder approximations.

Note that even without these problems, smoothly changing the LOQ would require running both involved algorithms and blending their results. However, we acknowledge that for several applications a discrete LOQ scheme suffices, like in selecting a LOQ during start-up to adapt to different hardware setups or in the special case where all shadow-receiving regions are disjoint and the same LOQ is used throughout a region. But even then, it is unclear how to order the algorithms in question concerning both quality and expected render time. For instance, we observe that one algorithm which excels in situation A might fail miserably for scene B, whereas another algorithm might show average quality for both, i.e. relative orderings may not be consistent across scenes. In addition, even algorithms yielding soft shadows of lower quality, like PCSS [122], can take a considerable amount of time, further limiting the practicability of a multi-algorithm LOQ approach. Therefore, we will only consider LOQ schemes using a single algorithm in the remainder of this chapter.

5.1.2 Geometric occluder LOD

When operating directly on the occluder geometry, another option to provide soft shadow LOQ is to apply a geometric LOD scheme to the occluders. Ultimately, for each pixel in a soft shadow region, a separate view-dependent LOD of the occluder geometry that is consistent across neighboring pixels is desired. Since providing this pixel-wise LOD would incur a tremendous cost, in practice simpler schemes have to be adopted. Unfortunately, this is only

reasonable if objects are classified as either shadow casters or shadow receivers, since otherwise problems exist with self-shadowing and when a caster and a receiver are in contact.

Actually, Ren et al. [313] perform such a kind of occluder LOD when determining soft shadows cast by low-frequency environment maps. They approximate light blockers by a sphere hierarchy and utilize the solid angle subtended by a blocker to select the hierarchy level for deriving the occlusion caused by this blocker at a certain receiver point. However, as mentioned in Sec. 3.2.4, the sphere approximation negatively impacts the attainable soft shadow quality, in particular that of visually important contact shadows.

Note that one might argue that employing different shadow map levels for micro-occluder construction constitutes some kind of LOD scheme for the occluder geometry reconstructed from the shadow map. On the other hand, soft shadow mapping simply chooses a certain level when determining light visibility for a pixel, and hence does not even select an occluder-specific LOD but just picks the same LOD for all occluders.

5.1.3 Sparse visibility sampling

Because light visibility often varies rather smoothly in penumbra regions, one may adapt LOQ by performing the visibility evaluation not per pixel but according to a sparser sampling with a subsequent interpolation step.

A very simple such approach is to use a lower image resolution for determining light visibility with the adopted soft shadow algorithm. For example, DeCoro and Rusinkiewicz [90] first render the unshadowed shaded scene in a full-resolution color buffer, also storing associated normals and depth values. Approximating environmental lighting with many point lights, they subsequently determine light visibility at a reduced resolution for each light, derive the light's shading contribution at shadowed receiver pixels, and accumulate these surplus shading portions (i.e. excessively reflected radiance) in a shadow buffer. This is then resampled to full resolution utilizing normal and position information, and finally subtracted from the unshadowed color buffer to obtain the shadowed and shaded result. Note that, at least in this concrete algorithm, the sparser shadow buffer samples are different from (any subset of) the full-resolution image points. This implies having to render the scene again to determine the receiver sample points for visibility computation, unlike in case of subsampling. A further, more general disadvantage is that the soft shadow LOQ cannot vary across screen but is constant as determined by the chosen lower resolution.

Whereas such a uniform sampling density treats all penumbra regions equally irrespective of their extent and hence may undersample thin penumbrae while at the same time oversampling large ones, more sophisticated techniques provide a better adaptivity to the actual penumbra sizes. For instance, Guennebaud et al. [148] successfully implemented a sparse sampling scheme for their soft shadow mapping algorithm, enabling high speed-ups. They derive an estimate of the penumbra's screen-space footprint and employ it to adjust the sampling density by appropriately skipping visibility computations for some pixels. The resulting subsampled visibility solution is then subjected to a pyramidal pull-push reconstruction to determine the final soft shadows. Since the relative sparseness is guided by a parameter (called s_{\min} by them) that is fixed throughout the screen, the LOQ can only be influenced globally. Note that with increasing sparseness objectionable patterns can appear with this approach, as closer inspection of the published result images demonstrates. Because the underlying sparse sampling pattern is fixed in screen space, these patterns can be expected to become particularly noticeable in animated scenes.

More generally, it is unclear whether smoothly varying relative sparseness—sparseness relative to the (spatially varying) sample density required for a certain quality—suffices to allow smooth LOQ transition or whether blending between the results obtained for two different sampling sparsenesses is required.

5.1.4 Intrinsic algorithm parameters

Another option for realizing soft shadow LOQ is to vary an intrinsic parameter of an algorithm that affects quality. In case of soft shadow mapping, varying the bit field size and adapting the complexity of the bitmask's sampling pattern (completely regular, regularly jittered, or random) are possible choices when using occlusion bitmasks. Again, smooth LOQ transitions would require alpha blending. Employing varying numbers of shadow maps or switching between area accumulation and occlusion bitmasks also constitute quality alteration parameters, which, however, pose problems similar to those in multi-algorithm LOQ schemes concerning blending.

The most promising parameter is the shadow map level and hence the virtual shadow map resolution used, which is somewhat related to the geometric LOD of the occluders.¹ Ideally, we want a hierarchical occluder representation derived from the shadow map which allows smoothly varying soft shadow LOQ by blending between two hierarchy levels and computing light visibility for the resulting intermediate occluder representation instead of having to blend the visibility results for the two discrete hierarchy levels.

The microquad approximation might seem like a natural candidate because it can be considered a geometry image [146] of the occluder/receiver surface as seen from the light that trivially supports geomorphing. However, it turns out that deriving coarser-level vertices cannot be done by mere subsampling but must perform some kind of minimum aggregation of the light depth values. This is also one reason why the seemingly obvious relationship to vertex clustering [318] is actually rather weak. The major obstacle which prevents microquads (and supposedly any occluder representation) from achieving the desired ideal is the binary decision of whether they get backprojected or not. Recall that a microquad is only considered if all four vertices are closer to the light source than the shadow-receiving point \mathbf{p} . Therefore, while geomorphing between two levels causes a smooth transition of the involved microquads, once a vertex crosses the depth level of \mathbf{p} , the corresponding microquad abruptly becomes included or excluded, respectively, from backprojection, i.e. the LOQ is not transitioning smoothly. Consequently, adapting the shadow map level requires alpha blending for smooth LOQ variations.

5.2 Smooth quality variation for soft shadow mapping

Carefully considering the options discussed in the preceding section, we decided to base our LOQ scheme for soft shadow mapping on the shadow map level employed for micro-occluder construction. Recall that choosing a coarser level reduces the number of micro-occluders that

¹Note that adapting the shadow map resolution is a popular choice for trading quality against effort and required resources in case of hard shadows. A nice example are fitted virtual shadow maps [138], which try to ensure that throughout the screen the ratio of a pixel's size projected into shadow map texture space and a shadow map texel satisfies some upper bound. To this end, the scene is decomposed into screen-space tiles, and (conceptually) for each a separate shadow map of optimal resolution is employed. A global quality-vs-performance parameter (called ξ) is offered which affects the highest pixel/texel ratio deemed acceptable and thus controls the effective shadow map resolution, hence the encountered aliasing, and ultimately the quality.

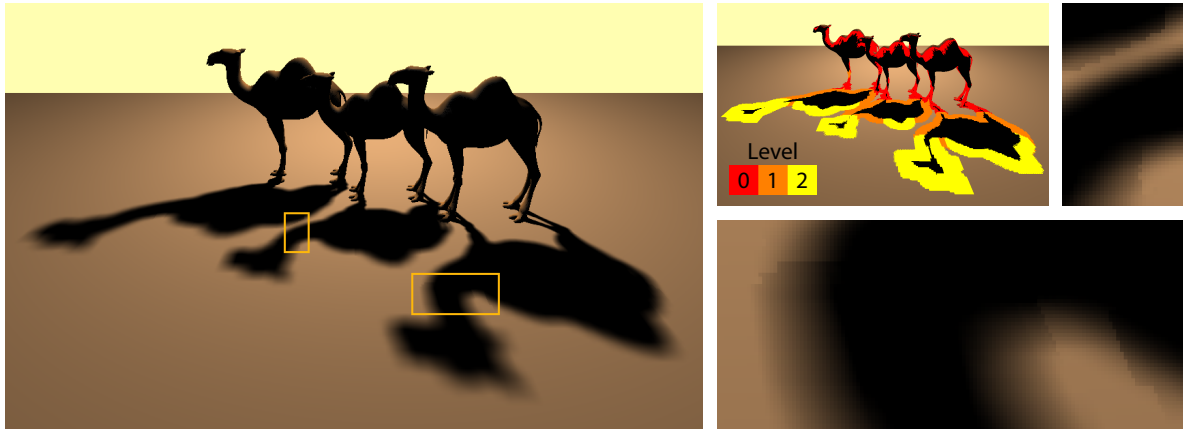


Figure 5.1 Transition artifacts (e.g. see zoom-ins) can occur with soft shadow mapping if varying shadow map levels are employed across pixels for micro-occluder construction and no blending is performed.

need to be processed and hence increases performance. At the same time, soft shadow quality typically degrades because fewer but larger micro-occluders are utilized for representing the actual light blockers, thus often raising the approximation error. Lower quality manifests itself in decreasing smoothness and detail in the shadow shape, as well as in growing or shrinking shadow regions, with the magnitude of change partially depending on whether micropatches or microquads are used.

As detailed in the last chapter, soft shadow mapping algorithms routinely choose the finest shadow map level at which the search area satisfies some user-specified upper bound on the number of encompassed micro-occluders in order to ensure real-time frame rates. However, since the level employed can vary across the pixels in a soft shadow region, transition artifacts may appear due to changing blocker approximations, as exemplified in Fig. 5.1. Consequently, note that the ability to smoothly vary the quality across screen is actually a requirement for the practical employment of soft shadow mapping algorithms, because manually choosing an appropriate micro-occluder budget that renders transitions unnoticeable cannot really be considered a viable alternative.

To realize a smooth variation of quality, we make the shadow map level a continuous quantity, and determine light visibility by considering the two closest integer shadow map levels and combining the results obtained for them via alpha blending. Note that this decision is in line with the observations from the previous section. In particular, as unsatisfying as it may be, alpha blending appears to be unavoidable when smooth soft shadow quality transitions are required.

The main challenge is to determine a real-valued shadow map level ℓ that varies continuously across a soft shadow region, with the fractional part driving the alpha blending. In a previous approach, Guennebaud et al. [148] suggest deriving ℓ from a continuously varying estimate of the closest occluder's depth obtained by linearly filtering the HSM. They hence implicitly assume that neighboring fragments sample the same HSM level when determining the depth range during search area refinement. In general, however, this assumption is not true and fails in particular if a more accurate acceleration structure like the YSM is used. As a consequence, discontinuities concerning ℓ are introduced which can manifest themselves as visible artifacts despite alpha blending.

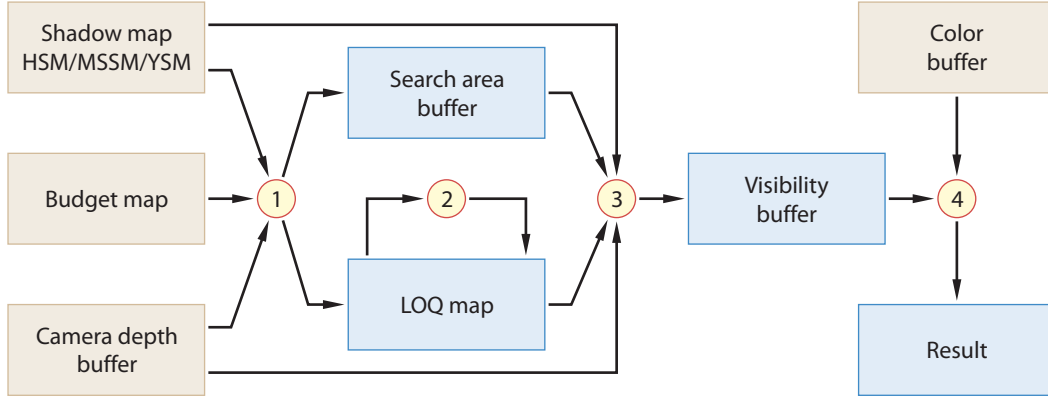


Figure 5.2 Overview of our approach for smooth soft shadow quality variation. ① Taking the local micro-occluder budget into account, at first the search area and the (real-valued) shadow map level ℓ are determined for each pixel. ② The resulting LOQ map is subsequently smoothed before ③ visibility is determined, performing alpha blending if necessary. ④ Finally, the visibility values are applied to the color buffer.

5.2.1 Approach

Our approach [336] for achieving a smooth soft shadow quality variation is summarized in Fig. 5.2. At its core is determining a real-valued level ℓ per pixel for micro-occluder construction such that ℓ and hence quality change continuously within a soft shadow region. Apart from tackling the above-mentioned shortcomings of previous approaches, we particularly also allow adapting the shadow quality according to screen-space-local features, like high texture masking or varying importance as assigned by the user or derived by perceptually motivated algorithms. To this end, a *budget map* is provided as input which stores for each pixel (x, y) the maximum number $b(x, y)$ of micro-occluders that the search area at the finally employed level ℓ should comprise.² By spatially varying the budget $b(x, y)$, soft shadow quality can be locally increased or decreased.

Since ensuring the smoothness of ℓ within a soft shadow region involves considering a pixel's neighborhood, our algorithm is realized as a deferred shading technique, with textures providing both the shading result (in a color buffer) and world-space position (indirectly via a linear camera-space depth buffer) for each pixel. Furthermore, search area pruning and micro-occluder processing are performed in separate passes to enable inserting a non-pixel-local adaptation phase that establishes smoothness.

In a first step, we utilize the chosen acceleration structure for the acquired shadow map to determine for each pixel whether it is in umbra or completely lit and if not, the relevant search area for micro-occluders. Taking the pixel's budget $b(x, y)$ from the input budget map into account, we then derive an appropriate real-valued shadow map level ℓ . The search area and ℓ are stored in two textures, referred to as *search area buffer* and *LOQ map*, respectively, for later use. Because the determined ℓ values don't necessarily vary smoothly across the screen, in a next step a smoothing filter is applied that respects geometric discontinuities. For performance reasons, we resort to a variant of separable bilateral filtering [297], where the smoothing kernel stops at depth discontinuities and at the interface to regions where no backprojection is required.

²In practice, we actually use $\lfloor \sqrt{b(x, y)} \rfloor$ as upper bound along each axis of the rectangular search area.

Subsequently, using the previously derived search area from the same-named buffer and the filtered ℓ value from the LOQ map, we perform the actual backprojection to determine light visibility. Note that unless the blend weight $\alpha(\ell)$ is integer, micro-occluder processing has to be done both for level $\lfloor \ell \rfloor$ and for level $\lceil \ell \rceil$, with the resulting visibility values then being linearly blended according to $\alpha(\ell)$. Finally, the computed light visibility is applied to the input color buffer to incorporate soft shadows into the shaded scene.

5.2.2 Discussion

While the specified micro-occluder budget b is satisfied by the initially derived level ℓ , this may no longer be the case after smoothing the LOQ map. Moreover, not real-valued but integer levels are processed, and when the next finer level $\lfloor \ell \rfloor$ is employed, typically the budget is violated, with up to roughly $4b$ micro-occluders getting considered. In particular, if alpha blending is required, both level $\lceil \ell \rceil$ (where usually at most b micro-occluders get processed) and level $\lfloor \ell \rfloor$ have to be taken into account, further exceeding the budget. Therefore, note that b only serves to guide the level of quality by specifying the size of the micro-occluders relative to the search area extent but it is not a strict upper bound on the number of actually processed micro-occluders.

Since alpha blending is particularly expensive (if $\alpha(\ell) \notin \{0, 1\}$), involving visibility determination for two levels, it is desirable to keep the transition region small where not just either level $\lceil \ell \rceil$ or $\lfloor \ell \rfloor$ is employed but a combination of both of them. For instance, Guennebaud et al. [148] suggest restricting blending to within a region of $\Delta\ell = 0.1$ for practical use. However, our experience shows that while this might be acceptable in static images, the resulting thinner blending regions often become visible in animated settings. Furthermore, severe artifacts can show up in regions where transitions between multiple levels occur within a very small neighborhood if the blending region is chosen too narrow. Hence, for maximum robustness, we always perform alpha blending. Note that consequently, a good classification of umbra and completely lit regions becomes even more important, i.e. using the YSM is highly advisable.

Compared to normal soft shadow mapping, our LOQ scheme features three sources of overhead. First, due to distributing the calculations over several shaders, texture reads and writes to the search area buffer and the LOQ map, as well as accesses to the budget map are introduced. However, we didn't observe any measurable performance penalty (on an NVIDIA GeForce 8800 GTX), probably because the effective parallelism of pixel shader executions and cache utilization are improved by this reorganization. Second, the LOQ map is smoothed in a separate pass. In practice, we use a filter with support 11×11 , which provides a good trade-off between speed and smoothing capability. Our measurements show that with this choice the smoothing step takes less than 2 ms for a 1024×768 viewport (again on a GeForce 8800 GTX). Finally, if the smoothly varying shadow map level ℓ is non-integer, we blend the visibility results for levels $\lceil \ell \rceil$ and $\lfloor \ell \rfloor$ and hence process more micro-occluders per pixel than with normal soft shadow mapping, where only level $\lfloor \ell \rfloor$ is considered. Measurements across many scenes and viewpoints suggest that in practice about 30% more micro-occluders are considered on average. However, in return for these overheads, spatial transition artifacts are successfully alleviated, enabling the practical use of selectively lower soft shadow quality to reduce frame time.

Finally, note that similar to aliasing in standard shadow mapping, depending on the scene configuration and the value of ℓ , the footprint of a single micro-occluder's influence region in screen space may be large, resulting in jaggy-like shadow boundaries. Even worse, this large footprint can lead to swimming artifacts; that is, discontinuities in the temporal domain can

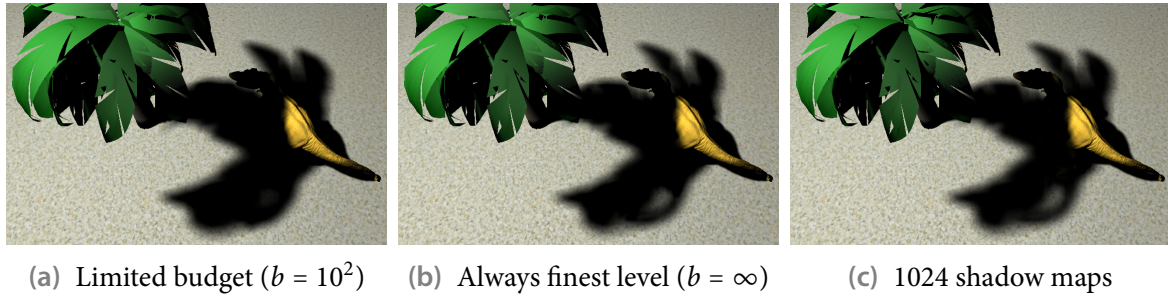


Figure 5.3 Dinosaur under palm tree. Imposing some medium micro-occluder budget reduces soft shadow detail and quality (a) compared to unconstrained soft shadow mapping (b), which comes pretty close to the reference solution (c).

appear once objects move relative to the light source and hence their rasterization in the shadow map changes. As already mentioned in Sec. 4.5, this is a general side effect of employing coarser-level micro-occluders to trade quality for speed. The obvious solution of increasing the effective shadow map resolution used for micro-occluder construction is not a viable option as it runs counter to reducing the number of micro-occluders and keeping the computational effort low. A more reasonable method to alleviate such artifacts is to apply temporal smoothing, for instance with a history buffer approach [328].

5.2.3 Results

We conclude this chapter and the part on soft shadows with some results. For all of them, a YSM of size $1024^2/256^2$ was utilized, micropatches were employed and visibility was determined via area accumulation. Note, however, that these choices are orthogonal to the presented scheme and hence alternatives like microquads and occlusion bitmasks can be readily adopted.

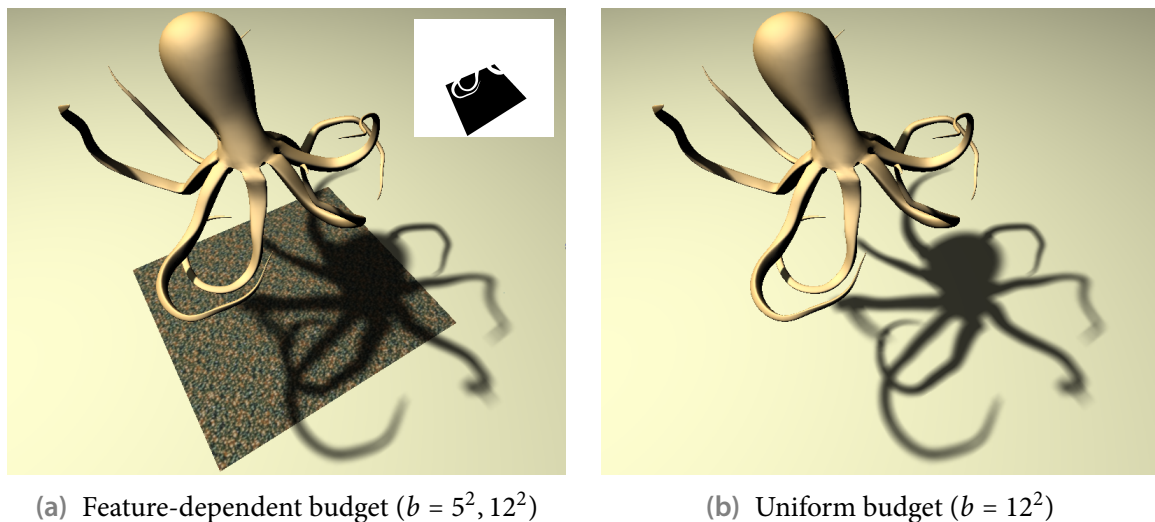


Figure 5.4 The scheme allows selectively decreasing the budget b in regions of high texture masking (a; to $b = 5^2$ in the black region shown in the budget map inset). Compared to retaining a uniformly high budget (b), this perceptually motivated work reduction increases performance, for instance from 32 Hz to 50 Hz on an NVIDIA GeForce 8800 GTX.

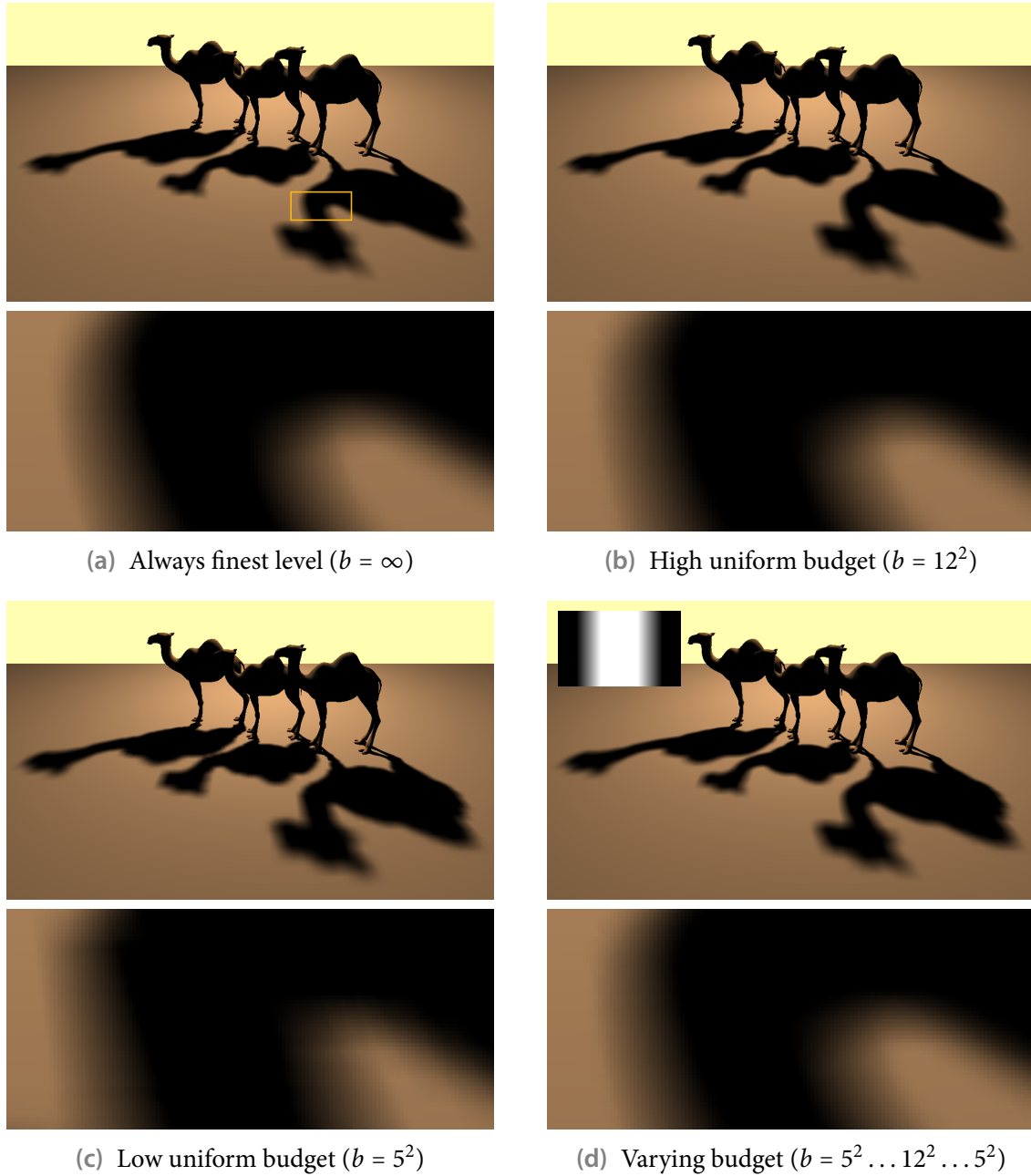


Figure 5.5 The soft shadow quality variation scheme enables not only smooth transitions for screen-uniform budgets (b, c) but also readily supports local budget variations (d: b decreases from 12^2 in the center to 5^2 at the left and right borders as shown in the inset).

Remember that the objective is to allow spatially varying quality degradation without introducing transition artifacts. Although a lower quality usually causes shadow shape smoothness and detail to decrease and shadow regions to grow, the resulting soft shadows remain plausible, as exemplified in Fig. 5.3. In particular, our approach successfully helps avoiding artifacts due to shadow map level transitions inherent to soft shadow mapping with limited budget b . This is further demonstrated in Fig. 5.5, showing the same scene as Fig. 5.1 (where no smoothing is performed and which hence suffers from artifacts). The concept of a budget map additionally allows locally adapting soft shadow quality while retaining smooth transitions. For instance,

| Scene/ result | Micro-occluder budget b | Frame rate | Processed pixels | Micro-occl./pixel | |
|------------------|------------------------------|---------------|---------------------|-------------------|----------|
| | | | | Mean | St. dev. |
| Fig. 5.3 a | 10^2 | 24.8 Hz | 12.7% | 194 | 84.8 |
| Fig. 5.3 b | ∞ | 7.1 Hz | 11.2% | 3,632 | 2,166 |
| Fig. 5.5 c | 5^2 | 82.6 Hz | 15.6% | 33.3 | 13.7 |
| Fig. 5.5 d | 5^2-12^2 | 35.8 Hz | 13.0% | 164 | 138 |
| Fig. 5.5 b | 12^2 | 27.1 Hz | 11.7% | 480 | 313 |
| Fig. 5.5 a | ∞ | 23.9 Hz | 11.0% | 548 | 351 |

Table 5.1 Performance quantities for two example scenes and various budgets, obtained for a viewport of size 1024×768 on an NVIDIA GeForce 8800 GTX. The percentage of pixels for which actual micro-occluder processing is performed, and the mean and standard deviation of the number of processed micro-occluders for these pixels characterize the required computational effort. Note that in case of an unconstrained budget (i.e. $b = \infty$) no LOQ map smoothing or alpha blending is performed.

we can selectively lower soft shadow quality at screen borders (cf. Fig. 5.5 d), which may be considered visually less important. Similarly, as shown in Fig. 5.4, the budget can be reduced in regions of large texture masking, where a high soft shadow quality would not be noticeable, anyway.

Table 5.1 provides some performance data for the two examples scenes in Figs. 5.3 and 5.5. First note again that blending is required to avoid transition artifacts unless all pixels use the same shadow map level i for micro-occluder construction. Since completely resorting to a coarser common shadow map level is usually not a viable option because objectionable artifacts would appear where occluders and receivers touch, we compare against using the finest level $i = 0$ throughout the screen (i.e. $b = \infty$). The reported figures show that selectively lowering the quality and performing blending is indeed faster than always using the finest level. This is mainly due to having significantly fewer micro-occluders to process per pixel on average. On the other hand, the percentage of pixels which are not classified as either completely lit or in umbra and for which hence micro-occluders actually need to be processed increases with decreasing budget b . Being reflected in growing shadow regions, this rise originates in the adaptation of query regions during search area pruning according to b for avoiding “over-precision” artifacts, which causes additional pixels (close to penumbrae that would be obtained for $b = \infty$) to perform actual micro-occluder processing.

PART II

Rendering of curved surfaces

CHAPTER 6

Fundamentals of curved surfaces

In reality, the majority of shapes are visually smooth. While it is possible to mimic this smoothness by a collection of connected planar polygonal facets like triangles, such an approximation only works satisfactorily if the facets are small enough. However, because the facet sizes required for visual smoothness depend on the visual angle covered by the surface, the illusion of visual smoothness will eventually break down when approaching the surface. Consequently, it is highly desirable to describe smooth surfaces directly by curved surface primitives.

A curved surface is precisely described by mathematic expressions and features a compact representation, like a net of control points. In particular, the number of required control points is usually rather low compared to the number of vertices in corresponding polygonal mesh approximations. Since there is also an intuitive relationship between control points and a surface's shape, curved surfaces are easier to work with from a modeling point of view than traditional polygonal meshes. Moreover, many operations become simpler. For instance, it is easier and cheaper to animate a coarse control net than to deal directly with a fine polygonal mesh.

In this chapter, we review the most important representatives of curved surface primitives. Since our primary goal is to increase the realism in real-time rendering applications by utilizing such primitives, a special focus is put on the relevance towards this aim and on related aspects. We note that in practice curved surface primitives are often employed to provide a smooth base surface to which a displacement map is then applied [209]; however, such extensions are beyond the scope of this thesis.

To be of any practical value for real-time rendering applications, we must be able to efficiently render curved surfaces. For current graphics hardware, which has no built-in support for such primitives, tessellation-based approaches are the most suitable ones. Here, a piecewise-linear approximation with triangular meshes is derived and rendered using the standard triangle-centric graphics pipeline. Ideally, the tessellation is adaptive, generating triangles of locally varying and always sufficiently small (but not much smaller) size to achieve visual smoothness. This essentially comes down to creating a view-dependent level of detail.

Raycasting is another relevant rendering technique, especially for implicit surface representations. We close this chapter with a section discussing rendering approaches in more detail. The next chapter is then devoted exclusively to GPU-assisted, real-time, adaptive, on-the-fly tessellation, exploring major variants and aspects in depth.

6.1 Bézier surfaces

The probably most important curved surface primitives for rendering are Bézier surfaces, polynomial surfaces in Bézier form. Not only can many other primitives be decomposed into or well approximated by them, but Bézier surfaces also have several properties that make processing them efficient, simple, intuitive, and also numerically stable.

Since Bézier surfaces are extensions of Bézier curves to two dimensions and inherit many properties and concepts from them, we first briefly review the curve case. A Bézier curve

$$\mathbf{b}(t) = \sum_{i=0}^n \mathbf{b}_i B_i^n(t), \quad t \in [0, 1]$$

is a polynomial curve of degree n expressed as linear combination of $n + 1$ control points \mathbf{b}_i ($0 \leq i \leq n$), with the univariate Bernstein polynomials

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

serving as coefficients. Because the Bernstein polynomials are non-negative (for $t \in [0, 1]$) and sum to unity, the control points are actually combined convexly, yielding numerical stability, especially compared to a monomial representation. Note that since the Bernstein polynomials can be expressed recursively using convex combinations, i.e.

$$B_i^n(t) = (1-t)B_{i-1}^{n-1}(t) + tB_{i-1}^{n-1}(t),$$

a Bézier curve can alternatively be defined by repeated convex combinations of control points (de Casteljau algorithm).

A curve's control points are related to its shape in a meaningful and intuitive way. The endpoints are interpolated, i.e. $\mathbf{b}(0) = \mathbf{b}_0$ and $\mathbf{b}(1) = \mathbf{b}_n$, but in general not the interior control points. Due to the convex combinations in its definition, a Bézier curve is always contained in the convex hull of the control points. Note that this property enables fast bounding box computations. Moreover, Bézier curves are invariant under affine transformations. Consequently, first transforming the control points and then evaluating the curve at some parameter values is equivalent to first performing the evaluation and then transforming the result. Further properties along with a more detailed discussion can be found in standard textbooks like Farin's [116].

In practice, usually only the cubic case ($n = 3$) is considered. Lower-degree curves are unable to describe inflections and hence S-like shapes. Moreover, cubic is the lowest degree where the curve is not necessarily within a plane and where prescribed endpoints and derivatives of them can be interpolated. On the other hand, higher degrees are rarely required for computer graphics applications.

Polynomial curves cannot describe conic sections (ellipse, parabola, hyperbola). If this expressive power is required, one has to resort to rational curves. In case of a rational Bézier curve, each k -D control point \mathbf{p}_i is associated with a weight¹ w_i , yielding homogeneous $(k+1)$ -D control points $\mathbf{b}_i = (w_i \mathbf{p}_i, w_i)^T$. The rational k -D curve $\mathbf{p}(t)$ is obtained by evaluating the $(k+1)$ -D polynomial Bézier curve $\mathbf{b}(t)$ with control points \mathbf{b}_i and performing the dehomogenizing divide, i.e.

$$\mathbf{p}(t) = \frac{\sum_{i=0}^n w_i \mathbf{p}_i B_i^n(t)}{\sum_{i=0}^n w_i B_i^n(t)}.$$

¹In the rest of the thesis, we always implicitly assume all weights to be positive.

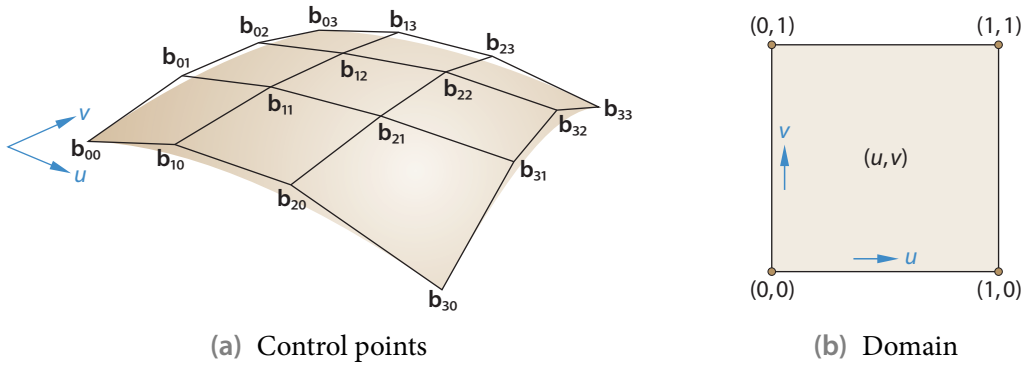


Figure 6.1 Bicubic tensor-product Bézier patch.

Most properties from polynomial Bézier curves, like endpoint interpolation and the convex hull property, also hold in the rational case. Furthermore, in addition to affine invariance, rational Bézier curves are even invariant under projective transformations.

6.1.1 Bézier patches

One way to generalize Bézier curves to surfaces is by building the tensor product of two Bézier curves. That is, a surface is swept out by moving an m -degree Bézier curve with the trajectory of each control point being described by an n -degree Bézier curve. A (*tensor-product*) *Bézier patch* of degree $m \times n$ has a rectangular parameter domain $[0, 1]^2 \ni (u, v)$ and is defined as

$$\mathbf{b}(u, v) = \sum_{i=0}^m \sum_{j=0}^n \mathbf{b}_{ij} B_i^m(u) B_j^n(v) = \sum_{i=0}^m \left(\sum_{j=0}^n \mathbf{b}_{ij} B_j^n(v) \right) B_i^m(u) = \sum_{j=0}^n \left(\sum_{i=0}^m \mathbf{b}_{ij} B_i^m(u) \right) B_j^n(v),$$

with the grid of control points \mathbf{b}_{ij} ($0 \leq i \leq m, 0 \leq j \leq n$) specifying the shape.

By construction, each boundary curve is a Bézier curve of degree m ($\mathbf{b}(u, 0)$ and $\mathbf{b}(u, 1)$) or n ($\mathbf{b}(0, v)$ and $\mathbf{b}(1, v)$), respectively. Properties inherited from the curve case include corner-point interpolation, the convex hull property as well as affine invariance. Again, the bicubic case ($m = n = 3$; cf. Fig. 6.1) is the most relevant one in practice. A famous example of a model composed of bicubic Bézier patches is shown in Fig. 6.2.

The partial derivatives required for calculating the surface normal are given by

$$\begin{aligned} \mathbf{b}_u(u, v) &= m \sum_{i=0}^{m-1} \sum_{j=0}^n (\mathbf{b}_{i+1,j} - \mathbf{b}_{i,j}) B_i^{m-1}(u) B_j^n(v), \\ \mathbf{b}_v(u, v) &= n \sum_{i=0}^m \sum_{j=0}^{n-1} (\mathbf{b}_{i,j+1} - \mathbf{b}_{i,j}) B_i^m(u) B_j^{n-1}(v). \end{aligned}$$

Note that they are themselves Bézier patches of degree $(m-1) \times n$ and $m \times (n-1)$, respectively. Similarly, the second-order partial derivatives are

$$\begin{aligned} \mathbf{b}_{uu}(u, v) &= m(m-1) \sum_{i=0}^{m-2} \sum_{j=0}^n (\mathbf{b}_{i+2,j} - 2\mathbf{b}_{i+1,j} + \mathbf{b}_{i,j}) B_i^{m-2}(u) B_j^n(v), \\ \mathbf{b}_{vv}(u, v) &= n(n-1) \sum_{i=0}^m \sum_{j=0}^{n-2} (\mathbf{b}_{i,j+2} - 2\mathbf{b}_{i,j+1} + \mathbf{b}_{i,j}) B_i^m(u) B_j^{n-2}(v), \end{aligned}$$

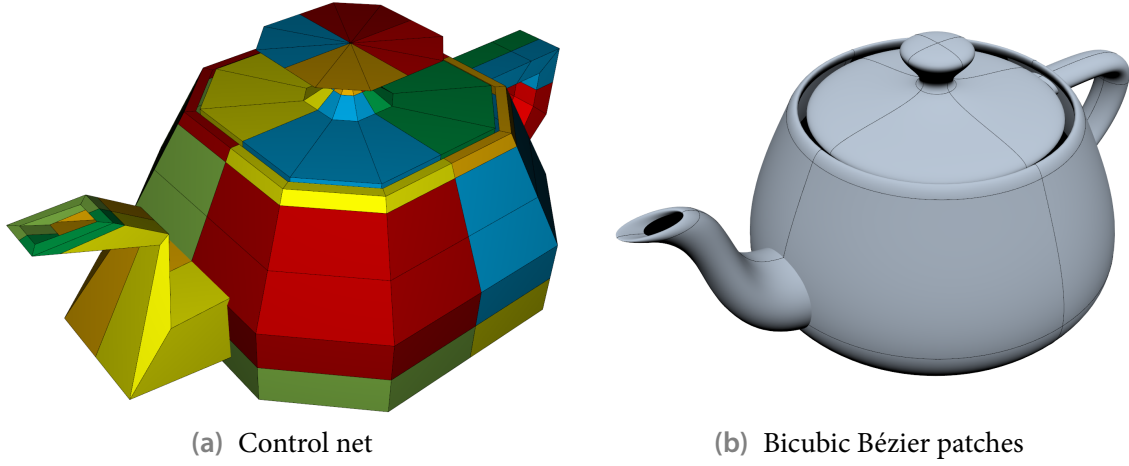


Figure 6.2 Example model Utah/Newell teapot (actually scaled by $10/13$ in height). Control faces belonging to the same patch are colored identically.

and the mixed partial derivative is

$$\mathbf{b}_{uv}(u, v) = mn \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (\mathbf{b}_{i+1,j+1} - \mathbf{b}_{i+1,j} - \mathbf{b}_{i,j+1} + \mathbf{b}_{i,j}) B_i^{m-1}(u) B_j^{n-1}(v).$$

Analogous to the curve case, a rational Bézier patch

$$\mathbf{p}(u, v) = \frac{(\mathbf{b}(u, v))_{xyz}}{(\mathbf{b}(u, v))_w} = \frac{\sum_{i=0}^m \sum_{j=0}^n w_{ij} \mathbf{p}_{ij} B_i^m(u) B_j^n(v)}{\sum_{i=0}^m \sum_{j=0}^n w_{ij} B_i^m(u) B_j^n(v)} = \frac{\mathbf{P}(u, v)}{W(u, v)}$$

can be defined by using homogeneous 4D control points $\mathbf{b}_{ij} = (w_{ij} \mathbf{p}_{ij}, w_{ij})^T$ resulting from enriching each 3D control point \mathbf{p}_{ij} with a weight w_{ij} . The partial derivative

$$\mathbf{p}_u(u, v) = \frac{(\mathbf{b}_u(u, v))_{xyz} - (\mathbf{b}_u(u, v))_w \mathbf{p}(u, v)}{(\mathbf{b}(u, v))_w} = \frac{W(u, v) \mathbf{P}_u(u, v) - W_u(u, v) \mathbf{P}(u, v)}{W(u, v)^2}$$

is obtained by the chain rule; $\mathbf{p}_v(u, v)$ can be determined analogously. The (unnormalized) surface normal is then $\mathbf{n}(u, v) = \mathbf{p}_u(u, v) \times \mathbf{p}_v(u, v)$. Note that $\mathbf{p}_u(u, v)$ is a rational patch of degree $(2m - 1) \times 2n$ in the numerator and degree $2m \times 2n$ in the denominator. The second partial derivative

$$\begin{aligned} \mathbf{p}_{uu}(u, v) &= \frac{(\mathbf{b}_{uu}(u, v))_{xyz} - 2(\mathbf{b}_u(u, v))_w \mathbf{p}_u(u, v) - (\mathbf{b}_{uu}(u, v))_w \mathbf{p}(u, v)}{(\mathbf{b}(u, v))_w} \\ &= \frac{W^2 \mathbf{P}_{uu} - 2W W_u \mathbf{P}_u + (2W_u^2 - W W_{uu}) \mathbf{P}}{W^3} (u, v) \end{aligned}$$

even has an overall degree of $3m \times 3n$. Unlike in the polynomial case, the complexity hence doesn't decline but rises with the number of differentiations. As a consequence, computing bounds on the partial derivatives of a rational Bézier patch is rather expensive.

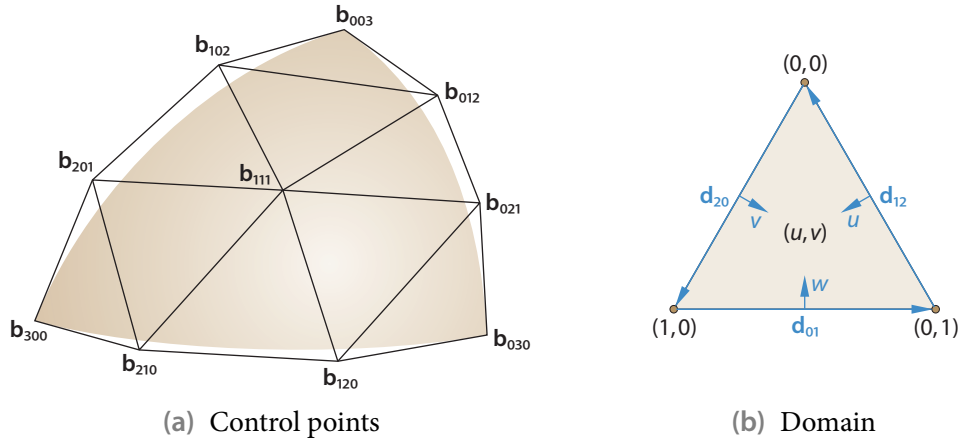


Figure 6.3 Cubic Bézier triangle.

6.1.2 Bézier triangles

Bézier patches (and their spline extensions, see Sec. 6.2) are wide-spread and popular in modeling, not least because the control net's quad layout is well suited to follow parallel feature lines often encountered in nature, architecture and engineering. However, triangular shapes occurring at poles (like at the knob of the teapot's lid in Fig. 6.2) or when feature lines merge require collapsing a boundary curve to a single point, degenerating the patch.

Another surface primitive that generalizes Bézier curves and doesn't suffer from this topological limitation is the *Bézier triangle* [88, 115]. Since it can be defined by repeated barycentric interpolation between three control points, it is the natural extension of Bézier curves to surfaces, recalling that Bézier curves are obtained by repeated linear interpolation between two control points.

The closed-form definition of a triangular Bézier patch of degree n is

$$\mathbf{b}(u, v) = \sum_{i+j+k=n} \mathbf{b}_{ijk} B_{ijk}^n(u, v),$$

where u , v and $w = 1 - u - v$ are barycentric coordinates of the triangular domain and

$$B_{ijk}^n(u, v) = \binom{n}{i, j, k} u^i v^j (1 - u - v)^k = \frac{n!}{i! j! k!} u^i v^j (1 - u - v)^k$$

denotes the bivariate Bernstein polynomials. The Bézier triangle's shape is determined by the $\frac{1}{2}(n+2)(n+1)$ control points \mathbf{b}_{ijk} . For the cubic case ($n = 3$), the most relevant one in practice, Fig. 6.3 shows the control net and visualizes the domain.

Properties like affine invariance, corner-point interpolation, and the convex hull property also hold for Bézier triangles. The boundary curves are Bézier curves of degree n . While partial derivatives can be computed, we are usually interested in the directional derivatives

$$\begin{aligned} D_{\mathbf{d}_{01}} \mathbf{b}(u, v) &= n \sum_{i+j+k=n-1} (\mathbf{b}_{i,j+1,k} - \mathbf{b}_{i+1,j,k}) B_{ijk}^{n-1}(u, v), \\ D_{\mathbf{d}_{12}} \mathbf{b}(u, v) &= n \sum_{i+j+k=n-1} (\mathbf{b}_{i,j,k+1} - \mathbf{b}_{i,j+1,k}) B_{ijk}^{n-1}(u, v), \\ D_{\mathbf{d}_{20}} \mathbf{b}(u, v) &= n \sum_{i+j+k=n-1} (\mathbf{b}_{i+1,j,k} - \mathbf{b}_{i,j,k+1}) B_{ijk}^{n-1}(u, v) \end{aligned}$$

with respect to the (u, v) domain directions

$$\mathbf{d}_{01} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}, \quad \mathbf{d}_{12} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \quad \mathbf{d}_{20} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix},$$

which are parallel to the three boundaries ($w = 0$, $u = 0$, $v = 0$). Note that each directional derivative is itself a Bézier triangle of degree $n - 1$. The second-order directional derivatives, like

$$D_{\mathbf{d}_{01}}^2 \mathbf{b}(u, v) = n(n-1) \sum_{i+j+k=n-2} (\mathbf{b}_{i,j+2,k} - 2\mathbf{b}_{i+1,j+1,k} + \mathbf{b}_{i+2,j,k}) B_{ijk}^{n-2}(u, v),$$

and the mixed directional derivatives, like

$$D_{\mathbf{d}_{01}, -\mathbf{d}_{20}}^{1,1} \mathbf{b}(u, v) = n(n-1) \sum_{i+j+k=n-2} (\mathbf{b}_{i,j+1,k+1} - \mathbf{b}_{i+1,j+1,k} - \mathbf{b}_{i+1,j,k+1} + \mathbf{b}_{i+2,j,k}) B_{ijk}^{n-2}(u, v),$$

are also Bézier triangles (of degree $n - 2$).

Note that a degree- n Bézier triangle can be converted into a degenerate Bézier patch of degree $n \times n$, where one boundary curve is collapsed to a single point [222]. Conversely, a general Bézier patch of degree $n \times m$ can be described by two Bézier triangles of degree $n + m$ [223].

6.1.3 PN triangles

Bézier triangles can be used for scattered data interpolation [247]. Often, given three points \mathbf{P}_i along with associated normals \mathbf{N}_i , one seeks a Bézier triangle that interpolates to this data (at its corners). In the context of real-time rendering with its still prevalent coarse triangle models, this can be adapted to replace each flat triangle by a Bézier triangle to get a smoother surface appearance and, in particular, visually improved silhouettes. One such interpolation scheme, called *PN triangle* (short for curved point-normal triangle) or alternatively *N-patch*, was introduced by Vlachos et al. [387] in the course of equipping Direct3D 8 with support for higher-order surfaces.

A PN triangle is a cubic Bézier triangle $\mathbf{b}(u, v)$, where all control points are derived from the vertex positions and normals of a *base triangle*. In particular, no neighborhood information is required in the construction. As a consequence of this locality (and also of the cubic degree), two abutting PN triangles in general only meet with C^0 continuity. Note that this requires the two corresponding base triangles to share normals at their common vertices. Since the missing tangent continuity results in normal-field discontinuities, shading artifacts appear. To avoid these and obtain at least a C^0 -continuous normal field, the normal component is decoupled from the geometric component and specified by a separate linear or quadratic Bézier triangle $\mathbf{n}(u, v)$.

An example of a base triangle mesh and the resulting PN triangle surface is depicted in Fig. 6.4. Notice the smooth appearance and the curved silhouettes.

Geometric component

Given a base triangle with vertex positions $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ and associated normals $\mathbf{N}_1, \mathbf{N}_2, \mathbf{N}_3$, the cubic Bézier triangle $\mathbf{b}(u, v)$ describing the geometric component is constructed as follows. The *vertex control points* are just the given points:

$$\mathbf{b}_{300} = \mathbf{P}_1, \quad \mathbf{b}_{030} = \mathbf{P}_2, \quad \mathbf{b}_{003} = \mathbf{P}_3.$$

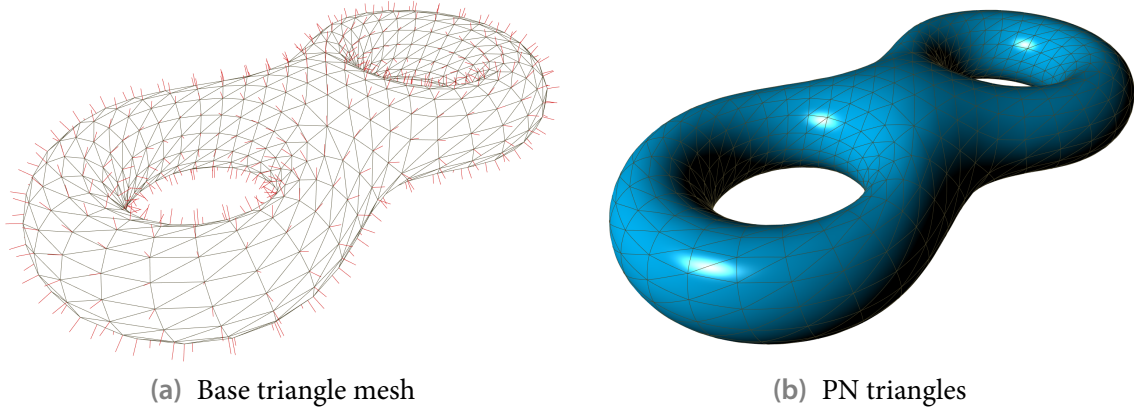


Figure 6.4 Example model double torus. The PN triangles are constructed from the base triangle mesh.

The *tangent control points* are obtained by projecting a linearly interpolated point between \mathbf{P}_i and \mathbf{P}_j onto the tangent plane at \mathbf{P}_i :

$$\begin{aligned}
 \mathbf{b}_{210} &= \left(\frac{2}{3}\mathbf{P}_1 + \frac{1}{3}\mathbf{P}_2 \right) - \left\langle \left(\frac{2}{3}\mathbf{P}_1 + \frac{1}{3}\mathbf{P}_2 \right) - \mathbf{P}_1, \mathbf{N}_1 \right\rangle \mathbf{N}_1 \\
 &= \frac{1}{3} \left(2\mathbf{P}_1 + \mathbf{P}_2 - \langle \mathbf{P}_2 - \mathbf{P}_1, \mathbf{N}_1 \rangle \mathbf{N}_1 \right), & \mathbf{b}_{120} &= \frac{1}{3} \left(2\mathbf{P}_2 + \mathbf{P}_1 - \langle \mathbf{P}_1 - \mathbf{P}_2, \mathbf{N}_2 \rangle \mathbf{N}_2 \right), \\
 \mathbf{b}_{021} &= \frac{1}{3} \left(2\mathbf{P}_2 + \mathbf{P}_3 - \langle \mathbf{P}_3 - \mathbf{P}_2, \mathbf{N}_2 \rangle \mathbf{N}_2 \right), & \mathbf{b}_{012} &= \frac{1}{3} \left(2\mathbf{P}_3 + \mathbf{P}_2 - \langle \mathbf{P}_2 - \mathbf{P}_3, \mathbf{N}_3 \rangle \mathbf{N}_3 \right), \\
 \mathbf{b}_{102} &= \frac{1}{3} \left(2\mathbf{P}_3 + \mathbf{P}_1 - \langle \mathbf{P}_1 - \mathbf{P}_3, \mathbf{N}_3 \rangle \mathbf{N}_3 \right), & \mathbf{b}_{201} &= \frac{1}{3} \left(2\mathbf{P}_1 + \mathbf{P}_3 - \langle \mathbf{P}_3 - \mathbf{P}_1, \mathbf{N}_1 \rangle \mathbf{N}_1 \right).
 \end{aligned}$$

Note that consequently, all control points of a PN triangle's boundary curve are only dependent on the vertex positions and normals of the corresponding edge. This ensures that two PN triangles constructed for two adjacent base triangles with identical vertices at their shared edge have a common boundary curve and hence join C^0 -continuously. Finally, the *center control point* is chosen to achieve quadratic precision (by degree-elevating a quadratic Bézier triangle):

$$\mathbf{b}_{111} = \frac{1}{4}(\mathbf{b}_{210} + \mathbf{b}_{120} + \mathbf{b}_{021} + \mathbf{b}_{012} + \mathbf{b}_{102} + \mathbf{b}_{201}) - \frac{1}{6}(\mathbf{b}_{300} + \mathbf{b}_{030} + \mathbf{b}_{003}). \quad (6.1)$$

The effect of the normals on the PN triangle's shape is demonstrated by an example in Fig. 6.5.

In case of a crease point, i.e. a vertex at which two abutting base triangles have different normals, C^0 continuity mandates to additionally provide the adjacent triangle's normal \mathbf{N}_i^a for the vertex as input. Picking \mathbf{P}_1 and edge $\mathbf{P}_1\mathbf{P}_2$ as example, we first determine a tangent line direction

$$\mathbf{T}_{01} = \mathbf{N}_1 \times \mathbf{N}_1^a.$$

The affected tangent control point is then obtained by projecting onto this tangent line instead of the tangent plane at \mathbf{P}_1 :

$$\mathbf{b}_{210} = \mathbf{P}_1 + \frac{1}{3} \langle \mathbf{P}_2 - \mathbf{P}_1, \mathbf{T}_{01} \rangle \mathbf{T}_{01}.$$

Normal component

The normal field for shading is described by a Bézier triangle $\mathbf{n}(u, v)$. Apart from simple linear interpolation of the base triangle's normals, a quadratic interpolation scheme is proposed by

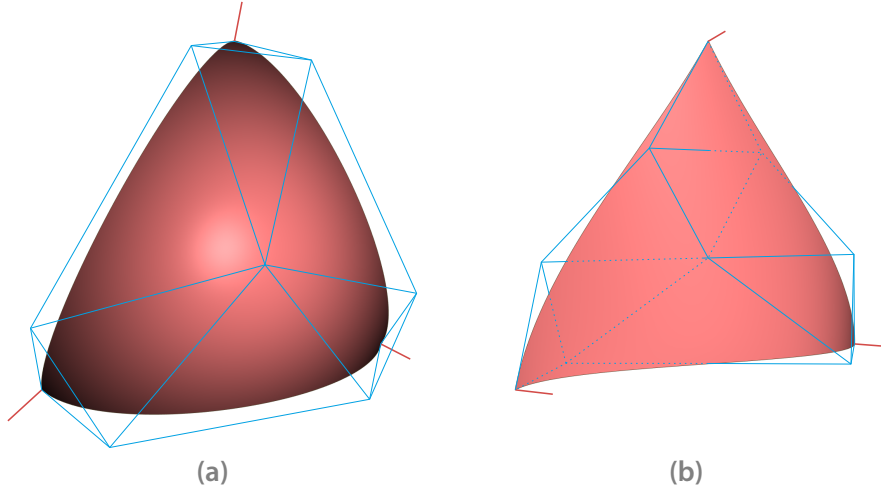


Figure 6.5 Two example PN triangles with identical base triangle vertex positions but different vertex normals. The normal field is constructed according to Farin's variant.

Vlachos et al. [387] to be able to capture inflections (like in Fig. 6.5 b). For this, the vertex control points are directly given by the provided normals:

$$\mathbf{n}_{200} = \mathbf{N}_1, \quad \mathbf{n}_{020} = \mathbf{N}_2, \quad \mathbf{n}_{002} = \mathbf{N}_3.$$

The remaining *mid-edge control points* are derived by reflecting the linearly interpolated normal at the center of each edge across a plane which is normal to this edge:

$$\begin{aligned} \mathbf{n}_{110} &= \frac{\mathbf{h}_{110}}{\|\mathbf{h}_{110}\|}, & \mathbf{h}_{110} &= (\mathbf{N}_1 + \mathbf{N}_2) - 2 \frac{\langle \mathbf{N}_1 + \mathbf{N}_2, \mathbf{P}_2 - \mathbf{P}_1 \rangle}{\|\mathbf{P}_2 - \mathbf{P}_1\|^2} (\mathbf{P}_2 - \mathbf{P}_1), \\ \mathbf{n}_{011} &= \frac{\mathbf{h}_{011}}{\|\mathbf{h}_{011}\|}, & \mathbf{h}_{011} &= (\mathbf{N}_2 + \mathbf{N}_3) - 2 \frac{\langle \mathbf{N}_2 + \mathbf{N}_3, \mathbf{P}_3 - \mathbf{P}_2 \rangle}{\|\mathbf{P}_3 - \mathbf{P}_2\|^2} (\mathbf{P}_3 - \mathbf{P}_2), \\ \mathbf{n}_{101} &= \frac{\mathbf{h}_{101}}{\|\mathbf{h}_{101}\|}, & \mathbf{h}_{101} &= (\mathbf{N}_3 + \mathbf{N}_1) - 2 \frac{\langle \mathbf{N}_3 + \mathbf{N}_1, \mathbf{P}_1 - \mathbf{P}_3 \rangle}{\|\mathbf{P}_1 - \mathbf{P}_3\|^2} (\mathbf{P}_1 - \mathbf{P}_3). \end{aligned}$$

In earlier work, van Overveld and Wyvill [384] propose an almost identical approach, but set $\mathbf{n}_{ijk} = 1/2 \mathbf{h}_{ijk}$ and use a factor of 3 instead of 2 in the expression for \mathbf{h}_{ijk} , resulting from their adopted constraint of the average curvature being as small as possible. Farin [117] suggests the following modifications to guarantee $\mathbf{n}(1/2, 1/2) = \mathbf{h}_{110}$, etc.:

$$\begin{aligned} \mathbf{n}_{110} &= \frac{3}{2} \mathbf{h}_{110} - \frac{1}{4} \mathbf{N}_1 - \frac{1}{4} \mathbf{N}_2 = \frac{5}{4} (\mathbf{N}_1 + \mathbf{N}_2) - 3 \frac{\langle \mathbf{N}_1 + \mathbf{N}_2, \mathbf{P}_2 - \mathbf{P}_1 \rangle}{\|\mathbf{P}_2 - \mathbf{P}_1\|^2} (\mathbf{P}_2 - \mathbf{P}_1), \\ \mathbf{n}_{011} &= \frac{3}{2} \mathbf{h}_{011} - \frac{1}{4} \mathbf{N}_2 - \frac{1}{4} \mathbf{N}_3 = \frac{5}{4} (\mathbf{N}_2 + \mathbf{N}_3) - 3 \frac{\langle \mathbf{N}_2 + \mathbf{N}_3, \mathbf{P}_3 - \mathbf{P}_2 \rangle}{\|\mathbf{P}_3 - \mathbf{P}_2\|^2} (\mathbf{P}_3 - \mathbf{P}_2), \\ \mathbf{n}_{101} &= \frac{3}{2} \mathbf{h}_{101} - \frac{1}{4} \mathbf{N}_3 - \frac{1}{4} \mathbf{N}_1 = \frac{5}{4} (\mathbf{N}_3 + \mathbf{N}_1) - 3 \frac{\langle \mathbf{N}_3 + \mathbf{N}_1, \mathbf{P}_1 - \mathbf{P}_3 \rangle}{\|\mathbf{P}_1 - \mathbf{P}_3\|^2} (\mathbf{P}_1 - \mathbf{P}_3). \end{aligned}$$

However, we observe that with this choice we actually get $\mathbf{n}(1/2, 1/2) = 1/8 \mathbf{N}_1 + 3/4 \mathbf{h}_{110} + 1/8 \mathbf{N}_2$. To really achieve $\mathbf{n}(1/2, 1/2) = \mathbf{h}_{110}$, we have to set $\mathbf{n}_{110} = 2\mathbf{h}_{110} - 1/2 \mathbf{N}_1 - 1/2 \mathbf{N}_2$.

The assumption of quadratic normal variation can yield shading results that are visually less pleasing than those obtained by the simpler linear interpolation. Actually, for many of the

models we worked with this was the case. We then just used the linear normal interpolant and degree-elevated it to a quadratic Bézier triangle to not skew performance measurements. Lee and Jen [211] investigate shading problems with quadratic normal interpolation in detail and discuss improvements. Most notably, they decide heuristically whether to assume linear or quadratic variation of the normals.

Extensions and variants

While one major design goal of PN triangles was simplicity to allow efficient hardware implementation, several extensions have been proposed to increase this primitive's expressive power and improve the visual smoothness of surfaces composed of PN triangles.

To exercise more control over the shape and especially of sharp features, *scalar tagged PN triangles* [47] were devised. Each base triangle vertex is augmented by a normal discontinuity vector Δ , which represents the change in average normal across a crease passing through the vertex ($\Delta = 0$ for smooth vertices). Furthermore, three per-vertex scalar tags (sharpness σ , tension θ and bias β) are introduced. These essentially adapt the well-known continuity, tension, and bias control parameters from spline theory [116, 192] to the PN triangle case.

Visual smoothness of PN triangles is hampered by the missing G^1 continuity across boundary curves. Since achieving such a G^1 continuity is in general not possible with cubic Bézier triangles, a different surface primitive must be utilized to this end. Gruen [144] adopts Nielson's side-vertex scheme [271] to get so-called *smoothed N-patches*. The boundary curves are derived as before. Then, for each boundary curve, a triangular patch is constructed by casting cubic curves from each point of the boundary curve (i.e. the side) to the opposing vertex (each interpolating these endpoints and their normals). In the final step, these patches are blended together, with the blend weights being functions of the barycentric coordinates. Because this construction is purely local and doesn't take adjacent triangles into account, the shape is sometimes unsatisfactory. To improve it, each patch is at first blended with the corresponding original PN triangle, where the employed weights are based both on the vicinity to the patch center and on the dot product of the face normals of the two base triangles sharing the related edge.

A related and more involved, but also more solid approach are *PNG1 triangles* [133], which require the three adjacent triangles as additional input. This information is utilized to construct a surface for each edge which is G^1 -continuous across the boundary curve. Finally, these three surfaces are blended together, yielding a cubic Bézier triangle where each non-vertex control point is a function of (u, v) . This triangular patch is at least G^1 -continuous at the border and G^2 in the interior. PNG1 triangles also support sharp features. Again, a separate quadratic normal field is constructed, where the mid-edge control points are derived from the patch's analytic normals.

Whereas all these techniques aim at enhancing PN triangles, *Phong tessellation* [46] further simplifies them by using a Bézier triangle of only quadratic degree for the geometric component, as well as a linear normal field. The surface is defined by projecting a point on the flat base triangle onto the three tangent planes at the vertices, and then linearly interpolating between the resulting points. The name Phong tessellation is motivated by the resemblance of this evaluation procedure with Phong normal interpolation. In practice, the actual surface is obtained by linearly blending between the flat base triangle and its Phong tessellation to reduce the curvedness of the patch. However, even for the recommended value $\alpha = 3/4$ of the related shape parameter α , we routinely observe surfaces that subjectively are curved too much. Consequently, because a Phong tessellation is only C^0 -continuous, they suffer from unnatural bumps

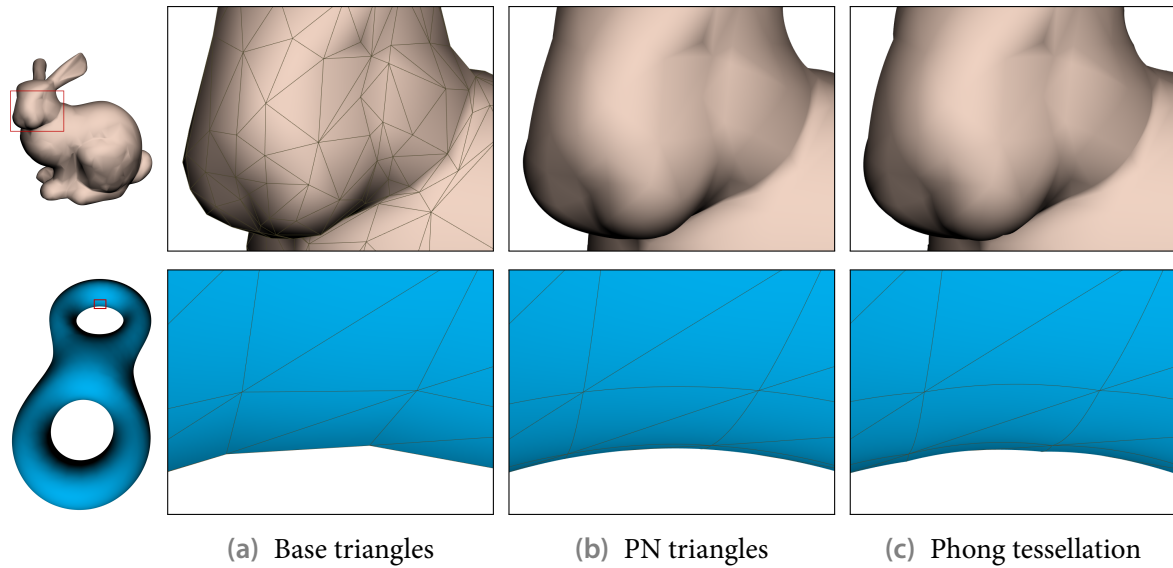


Figure 6.6 While PN triangles (b) yield a smooth appearance, Phong tessellation (c) introduces some unnatural bumps and dents.

and dents (cf. Fig. 6.6). Not least due to this shape deficiency, the practical relevance seems rather limited despite having a somewhat lower computational cost than the already rather cheap PN triangles.

6.2 Spline surfaces

Usually, to describe a surface, we need multiple Bézier patches (or Bézier triangles). It is then often desirable to treat a connected collection of them as a single entity, featuring a continuous global parameterization that spans all member-patches. Considering the underlying curve case again, we seek a piecewise-polynomial curve, a *spline*, where each segment is a Bézier curve. By joining multiple degree- n Bézier curves together at their endpoints, and subjecting each of these curve segments to an affine parameter transformation $\varphi_i(t) = (t - t_i)/(t_{i+1} - t_i)$ in order to get a continuous concatenation of parameter intervals $[t_i, t_{i+1}]$ and hence a global parameterization, we obtain a *Bézier spline* curve. Applying the tensor-product approach, this directly generalizes to Bézier spline surfaces.

Such a composition of Bézier patches is well suited for rendering, since each sub-patch (spanning one segment in u and one segment in v direction) can readily be treated independently from all others. Moreover, each sub-patch is of the same degree and depends only on the same fixed number of control points, enabling a simple and efficient uniform representation and processing of all sub-patches. On the other hand, Bézier splines are less suited for modeling because constraints on the control points have to be manually enforced to achieve a desired degree of continuity.

B-spline surfaces

For such applications, *B-spline* surfaces are a more appropriate representation. They generalize Bézier surfaces to the piecewise-polynomial case. A B-spline curve of degree n

$$\mathbf{d}(t) = \sum_{i=0}^N \mathbf{d}_i N_i^{n,T}(t), \quad t \in [t_n, t_{N+1}]$$

is described by a knot vector $T = \{t_0, t_1, \dots, t_{N+1+n}\}$ ($t_i \leq t_{i+1}$, $i = 0, \dots, N+n$) and control points $\mathbf{d}_0, \dots, \mathbf{d}_N$ (also called de Boor points). The definition makes use of the B-spline basis functions $N_i^{n,T}$, which depend on the curve's knot vector T . Like the Bernstein polynomials used for Bézier curves, they can be defined recursively:²

$$\begin{aligned} N_i^{0,T}(t) &= \begin{cases} 1, & \text{if } t \in [t_i, t_{i+1}) \\ 0, & \text{otherwise} \end{cases} \\ N_i^{k,T}(t) &= \frac{t - t_i}{t_{i+k} - t_i} N_i^{k-1,T}(t) + \frac{t_{i+1+k} - t}{t_{i+1+k} - t_{i+1}} N_{i+1}^{k-1,T}(t), \quad 1 \leq k \leq n. \end{aligned} \quad (6.2)$$

Because the basis functions $N_i^{n,T}$ have local support ($N_i^{n,T}(t) = 0$ for $t \in (-\infty, t_i) \cup [t_{i+n+1}, \infty)$), for a concrete value of $t \in [t_j, t_{j+1})$, it suffices to consider the range $i = j - n, \dots, j$ of control points and basis functions when computing $\mathbf{d}(t)$.

Concerning modeling applications, B-spline curves are superior to simple Bézier spline curves since continuity can be easily controlled via the knot vector. A B-spline curve $\mathbf{d}(t)$ is automatically C^{n-1} -continuous except at parameter values t corresponding to a knot t_i which has multiplicity μ_i greater than one (i.e. $t_i = t_{i+1} = \dots = t_{i+\mu_i-1}$), where it is only $C^{n-\mu_i}$ -continuous. Endpoint interpolation is achieved by setting the multiplicity of the first and of the last knot to $n+1$, each.

Again, the curve case can be generalized to surfaces by the tensor-product approach. A B-spline surface of degree $m \times n$ is completely specified by two knot vectors $U = \{u_0, \dots, u_{M+1+m}\}$ and $V = \{v_0, \dots, v_{N+1+n}\}$, one for each parameter direction, and a grid of control points \mathbf{d}_{ij} , $0 \leq i \leq M$, $0 \leq j \leq N$. The surface is then defined by

$$\mathbf{d}(u, v) = \sum_{i=0}^M \sum_{j=0}^N \mathbf{d}_{ij} N_i^{m,U}(u) N_j^{n,V}(v).$$

Sometimes, the knot vectors U and V are non-uniform, that is, internal knots are spaced non-uniformly (internal knots with multiplicity greater than one are a special case of this). Moreover, B-spline surfaces can be generalized to piecewise-rational surfaces by equipping each de Boor point \mathbf{d}_{ij} with an additional weight w_{ij} . Supporting both non-uniform knot vectors and varying weights results in *NURBS* (**n**on-**u**niform rational **B**-spline) surfaces [298, 316], the primary surface primitive in many computer-aided design and modeling tools and applications.

Circumventing topological constraints of NURBS surfaces

Irrespective of their wide-spread use, NURBS surfaces suffer from several limitations which can render modeling tedious. In particular, the rectangular domain and the regular quad grid

²If one denominator becomes 0, the corresponding basis function of degree $k-1$ also evaluates to zero. In such cases we assume $0/0 = 0$.

of control points, which are a direct consequence of the tensor-product approach, impose topological constraints. For instance, to add more detail within a certain rectangular region of control points, new control points can be introduced by inserting additional knots. However, adding a knot to the knot vector U results in introducing a whole column of new control points, spanning the complete parameter range in v direction and not just the relevant sub-range. Such a global refinement makes modeling harder because modifying a region of less detail has to be done unintentionally at a finer level of control.

Moreover, while the rectangular domain is well suited for following parallel feature lines, it mandates degenerating control net edges or faces to merge feature lines or to split a feature line. In practice, a complex surface is usually composed of multiple adjoining NURBS surfaces. On the one hand, this occurs to keep the number of unwanted but topologically required control points low and hence maintain a reasonable modeling control and effort. On the other hand, such a composition is necessitated when modeling surfaces of higher genus (i.e. with handles). However, it then once again is up to the modeler to enforce constraints that maintain a certain continuity. Furthermore, it happens that two abutting NURBS surfaces are specified with different knot vectors for the parameter direction along the shared boundary curve.³ Without special handling, this may result in gaps during rendering, especially in tessellation-based approaches.

T-splines [343, 346] address many of these limitations. They generalize tensor-product B-spline surfaces, allowing T-junctions in the control grid, which evolves to a so-called *T-mesh*. Abutting B-spline surfaces can be merged into a single T-spline. On the other hand, a T-spline can be converted to a collection of B-spline surfaces.

Another option for circumventing topological constraints of NURBS surfaces is *trimming*. Within the parameter space, trimming regions are defined via NURBS curves to restrict the domain in a non-trivial way. Those parts of the surface to which these parameter regions would map are trimmed, i.e. they are removed from the actual surface. Trimming enables to easily insert holes into a NURBS surface and realize complex boundaries.

Unfortunately, trimming introduces new problems and challenges. Concerning rendering, tessellating trimmed NURBS surfaces is a complex and expensive process. However, it is possible to avoid an explicit tessellation and mimic the parametric trimming process on the GPU with trim textures [150]. They provide a discrete sampling of the un-trimmed domain, storing for each sample whether it is defined or trimmed away. For each fragment, the trim texture is consulted at the associated parametric coordinates. If it indicates trimming, the fragment is discarded. However, the discrete sampling can cause visual problems at the boundary of a trimmed surface region abutting on another surface, necessitating further processing. Moreover, the pixel kill precludes antialiasing at trimmed surface region boundaries.

If two intersecting NURBS surfaces are trimmed at their intersection curve, the trimming curves in parameter space, which correspond to (high-degree) pre-images of this curve, usually suffer from inaccuracies that can lead to gaps. In such cases, a promising option is the conversion to untrimmed T-splines [345], which introduces some perturbations, though.

Direct B-spline surface evaluation on GPU

For rendering, especially for parallel tessellation-based approaches, B-spline surfaces are usually converted to a collection of equivalent Bézier patches [39]. Recall that this has the advantage

³In general, due to the limited numerical accuracy of floating-point numbers, the two boundary curve representations are not exactly equivalent, i.e. the surfaces don't abut completely.

that each patch has a compact description of rather small, uniform and bounded size. Compared to B-spline surfaces, where the number of knots and hence control points is basically unlimited, this representation is highly amenable to processing on graphics hardware. Nevertheless, some approaches were devised for direct evaluation of B-spline surfaces on the GPU without prior conversion to Bézier patches.

Kanai [176] stores knots and control points in textures, allocating one (potentially only partially occupied) row per NURBS surface. In a shader invoked with domain coordinates (u, v) as input, the relevant knot intervals $[u_j, u_{j+1}) \ni u$ and $[v_k, v_{k+1}) \ni v$ are first determined with a binary search. Then, the (non-vanishing) basis functions and their derivatives are computed recursively. To evaluate the surface position at (u, v) , the basis functions are linearly combined with the control points. Finally, the surface normal gets derived from the derivatives, which are computed analogously. Note that to efficiently realize the recursive basis function evaluation and to keep the required temporary space to a minimum, a specialized shader version is employed for each combination of degrees $m \times n$.

An alternative approach is pursued by Krishnamurthy et al. [194], who target the NURBS surface evaluation at sample points on a regular (u, v) grid. Initially, they determine the relevant subsequences of the knot vectors for each sample point on the CPU and store them in a texture. Using the recurrence relations from (6.2), the basis functions are calculated iteratively in multiple passes via render-target ping-pong, first in u and then in v direction. The values for the zero-degree basis functions N_i^0 are provided in a texture prepared on the CPU. In two final rendering passes, the basis functions are first multiplied with their corresponding control points, and then these results are summed up.

The technique was later extended to additionally compute exact derivatives [195]. Analogous to evaluating the surface positions, the derivatives of the basis functions are determined and then utilized together with the basis functions and the control points to compute the surface derivatives and finally the surface normal. While this approach is generic regarding the supported NURBS surfaces, it is not very appealing because it is slow, performs evaluation of vanishing intermediate basis functions, requires many rendering passes, and unnecessarily off-loads work onto the CPU. Moreover, it is only applicable to regular sampling grids.

6.3 Subdivision surfaces

B-spline surfaces can be refined by subdivision, e.g. performed with the Lane-Riesenfeld algorithm [205, 323]. This inserts a new knot in the middle of each knot interval, introduces new control points and adapts the position of old control points. The resulting new grid of control points approximates the B-spline surface closer than the old one. In the limit, successive subdivision yields a control grid that equals the B-spline surface.

As discussed in the previous section, B-spline surfaces suffer from topological restrictions. However, using the subdivision paradigm, they can be generalized to arbitrary topology. A *subdivision surface* is defined by a polygonal base mesh \mathcal{M}^0 and a set of subdivision rules. A one-time application of these rules to the mesh constitutes a subdivision step. This performs a topological refinement of the mesh \mathcal{M}^i and a subsequent smoothing, yielding a new mesh \mathcal{M}^{i+1} . Executing further subdivision steps continuously results in a sequence of meshes that converges to a surface \mathcal{M}^∞ . This so-called limit surface is in general smooth. In particular, note that smoothness doesn't require any constraints on the relative position of vertices in the input mesh \mathcal{M}^0 .

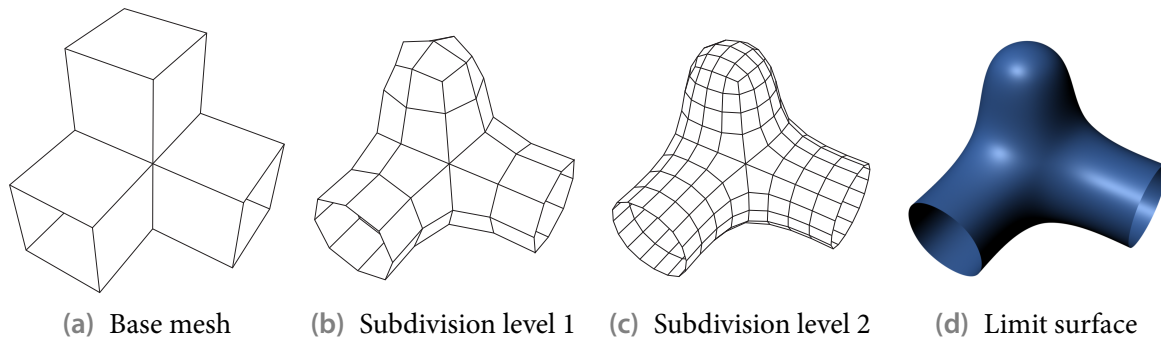


Figure 6.7 Example of Catmull-Clark subdivision.

A certain set of subdivision rules defines a subdivision scheme. Depending on the concrete scheme, the supported facet types of the base mesh \mathcal{M}^0 may be restricted. One of the first and probably the most well-known and most widely used scheme is *Catmull-Clark subdivision* [64]. It generalizes uniform bicubic B-spline surfaces to arbitrary topology and is hence a quadrilateral scheme but supports arbitrary polygonal facets in the base mesh. Catmull-Clark subdivision is widely supported by modeling applications such as BLENDER and MAYA as well as by renderers like PRMAN and MENTAL RAY. An example surface is shown in Fig. 6.7. The subdivision rules for the interior of the surface are illustrated and described in Fig. 6.8.

Depending on the *valence* k of a vertex, i.e. the number of incident edges, we distinguish between regular or ordinary vertices ($k = 4$ in case of quad meshes) and irregular or extraordinary vertices ($k \neq 4$). The limit surface is C^2 -continuous everywhere except at extraordinary vertices, where it is only C^1 -continuous. Since a subdivision step yields solely quad faces, only ordinary vertices are created during subdivision and hence the number of extraordinary vertices remains constant.

There also exist triangular subdivision schemes where each mesh \mathcal{M}^i is only composed of triangular facets. The first and most prominent one is *Loop's* scheme [231], which generalizes three-directional quartic box splines and is C^2 -continuous everywhere except at extraordinary vertices (valence $k \neq 6$), where it is C^1 -continuous. Moreover, subdivision schemes were devised that combine quadrilateral and triangular schemes and operate directly on mixed triangle/quad meshes [294, 324, 365]. For polar configurations, i.e. closed triangle fans with center vertices of high valence and outer vertices of valence 4, often encountered when modeling surfaces of revolution or capping features like fingertips, distinct subdivision schemes exist [178] that can also be combined with standard schemes like Catmull-Clark [261].

All mentioned subdivision schemes are approximating because the base mesh vertices are in general not interpolated by the limit surface. By contrast, interpolating schemes never move existing vertices when performing a subdivision step but only introduce additional vertices. Such schemes, like *modified butterfly subdivision* [421], which is C^1 -continuous everywhere, usually produce surfaces of lower quality and converge slower to the limit surface than approximating ones.

So-called primal schemes are characterized by splitting facets at each subdivision step; they replace a face by new faces which correspond topologically to this face. Besides them, dual schemes, which split vertices, like *Doo-Sabin* [96, 97], and schemes based on Laves or Archimedean tilings of the plane, like $\sqrt{3}$ -subdivision [190, 191], exist. More details on these as well as on the mathematical analysis of subdivision schemes can be found in books [293, 395] and SIGGRAPH course notes [419, 420] about subdivision.

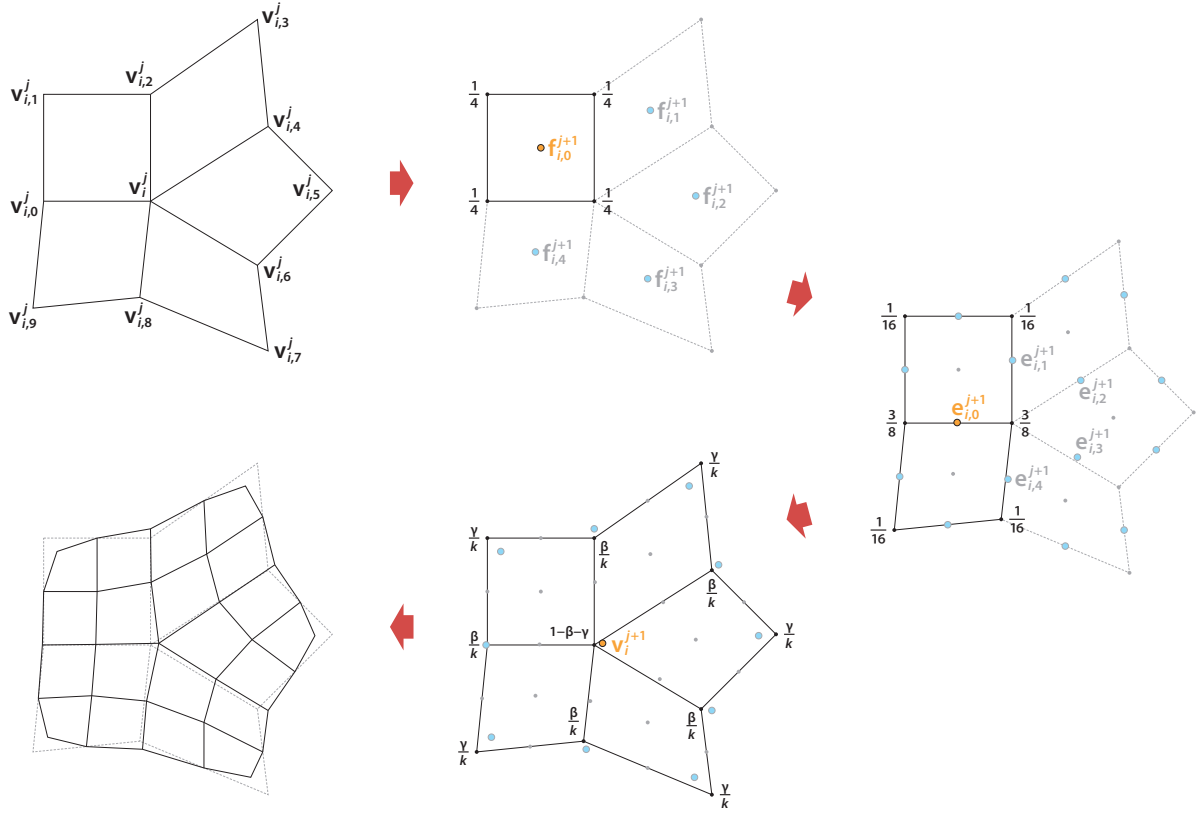


Figure 6.8 One step of Catmull-Clark subdivision. Consider a vertex \mathbf{v}_i^j of valence k . For each incident face, a new face point $\mathbf{f}_{i,l}^{j+1}$ is added at the face's centroid. Subsequently, for each edge, a new edge point $\mathbf{e}_{i,l}^{j+1} = \frac{1}{4}(\mathbf{v}_i^j + \mathbf{v}_{i,2l}^j + \mathbf{f}_{i,l}^{j+1} + \mathbf{f}_{i,l-1 \bmod k}^{j+1})$ is computed as the average of the edge's endpoints and the two adjacent faces' newly created face points. Each old vertex is finally displaced to $\mathbf{v}_i^{j+1} = \frac{k-2}{k} \mathbf{v}_i^j + \frac{1}{k^2} \sum_{\ell=0}^{k-1} \mathbf{v}_{i,2\ell}^j + \frac{1}{k^2} \sum_{\ell=0}^{k-1} \mathbf{f}_{i,\ell}^{j+1}$, i.e. to a weighted average of its current position, the average of its adjacent vertices and the average of the newly added face points. The new points $\mathbf{f}_{i,\ell}^{j+1}$, $\mathbf{e}_{i,\ell}^{j+1}$ and \mathbf{v}_i^{j+1} can also be computed directly from the old points \mathbf{v}_i^j using the weights from the indicated stencils/subdivision masks ($\beta = \frac{3}{2k}$, $\gamma = \frac{1}{4k}$). Note that different rules apply for boundaries.

Piecewise-smooth subdivision

While subdivision surfaces are smooth, they may be augmented by sharp features, like creases, to produce piecewise-smooth surfaces. To this end, each edge in the base mesh can be tagged as sharp. Depending on the number s of sharp incident edges, a vertex can be classified as either smooth ($s = 0$), dart ($s = 1$), crease ($s = 2$) or corner vertex ($s \geq 3$). For each non-smooth configuration, modified subdivision masks were derived for both Loop's scheme [166] and for Catmull-Clark subdivision [92]. Biermann et al. [33] provide improved rules for both schemes and also allow the prescription of normals at base mesh vertices.

Semi-sharp or soft creases may be incorporated by hybrid subdivision, where sharp rules are only applied during a number of initial subdivision steps and then the standard smooth rules are utilized [92].

6.3.1 Direct evaluation

Since subdivision surfaces are defined by repeated subdivision, the most natural method to render them is to perform a sufficiently high number n of subdivision steps, and to utilize the resulting polygonal approximation for rendering. It is possible to directly push the vertices of \mathcal{M}^n onto the limit surface by applying limit masks to the mesh [152]. Analogously, rules exist to derive the corresponding limit tangents directly from \mathcal{M}^n . Several efficient implementations pursuing such an approach have been devised [56, 300, 357]; however, the recursive refinement has multiple drawbacks, like high storage and memory bandwidth requirements because of the explicitly constructed intermediate meshes \mathcal{M}^j ($j < n$), a limited ability to locally adapt the number of performed subdivision steps, and requiring sampled surface points to correspond to vertices of \mathcal{M}^n (see also Sec. 7.2).

Alleviating the necessity to construct intermediate meshes \mathcal{M}^j , Kazakov [180] suggests directly evaluating the surface obtained after a fixed number n of subdivision steps. Considering the case of Catmull-Clark subdivision and $n = 2$ steps, he presents a formulation where the new position \mathbf{v}_i^2 of a vertex from \mathcal{M}^0 is expressed in terms of the vertex valence, the old position \mathbf{v}_i^0 , the average position of the neighboring vertices sharing an edge with the vertex, and the average position of the remaining face-sharing neighboring vertices. The other vertices of \mathcal{M}^2 , which are newly introduced with respect to \mathcal{M}^0 , are derived from a fixed, valence-independent number of neighborhood vertices, the original vertices \mathbf{v}_i^0 , and their updated positions \mathbf{v}_i^1 , which are computed analogously to \mathbf{v}_i^2 . While this evaluation technique was devised for implementation in a geometry shader, it requires support for input primitive topologies with a variable number of vertices, usually more than six (corresponding to triangle with adjacency). Such a direct realization is hence not supported by current PC graphics hardware, but with some minor modifications should be implementable within the pipeline of Direct3D 11 (discussed in Sec. 7.5.5). Nevertheless, because of the approach's restriction to a fixed and rather low subdivision depth and of enabling only parallelism across faces of the base mesh, the practical relevance of the technique appears quite limited.

Bolz and Schröder [42, 43] observe that subdivision is a linear process, where each vertex of the mesh \mathcal{M}^n projected to the limit surface \mathcal{M}^∞ is a linear combination of the base mesh vertices \mathbf{v}_i^0 . The combination coefficients constitute basis functions B_i which have compact support and don't depend on the actual vertices \mathbf{v}_i^0 but only on the local mesh connectivity and edge tags. These basis functions can be precomputed at sample points corresponding to the mesh vertices after n subdivision steps. During runtime, for each face of the base mesh \mathcal{M}^0 , the relevant vertices \mathbf{v}_i^0 in the neighborhood along with the corresponding basis functions B_i are determined. Then, for each sample within the face corresponding to an \mathcal{M}^n -vertex, the related basis function values are looked up, weighted by the vertices and summed. Because storage cost for basis functions is high, especially for higher subdivision levels n , an initial subdivision step is performed to isolate extraordinary vertices, i.e. the algorithm actually operates on \mathcal{M}^1 . Consequently, no face has more than one extraordinary vertex and the basis functions become a function of just valence and edge tags. By exploiting symmetries, the number of distinct basis functions and hence storage requirements can be further reduced. Despite the support for piecewise-smooth subdivision surfaces and good parallelizability, the reliance on sample-specific precomputed data renders this approach somewhat unattractive for implementation on current graphics hardware designed for high arithmetic intensity.

Note that both discussed methods essentially do the same: linearly combining base mesh vertices. The first indirectly derives the influence of each base mesh vertex on the fly with ex-

PLICIT formulae structured for efficient evaluation, whereas the latter determines these weights directly in a preprocess and stores them as basis functions. While both methods thus successfully avoid explicitly executing individual subdivision steps, they can only evaluate the surface at sample points corresponding to the vertices of a mesh \mathcal{M}^n after n steps. Schaefer and Warren [325, 326] lift this limitation, introducing a technique to directly evaluate the limit surface on a uniform grid of arbitrary integer size within each face of \mathcal{M}^0 . Analogous to the tabulated-basis-function approach, they precompute exact evaluation masks for all sample grid points and all supported neighborhood configurations.

For subdivision schemes which generalize piecewise-polynomial surfaces, like Catmull-Clark and Loop, Stam [364] devised a method to directly evaluate the limit surface at arbitrary sample positions. Since Catmull-Clark subdivision is based on bicubic B-spline surfaces, each regular quad face of a mesh \mathcal{M}^i , having only ordinary vertices, converges to a bicubic Bézier patch. The patch control points can be readily derived from the vertices of the face and its one-ring neighborhood. Consequently, within regular faces, the subdivision surface can easily be evaluated at any sample position. On the other hand, one subdivision step splits an irregular quad face with one extraordinary vertex into three regular faces and one irregular one. Therefore, any sample position within an irregular face will eventually be located within a regular face after a certain number n of subdivision steps,⁴ enabling a direct evaluation again.⁵ Since a single subdivision step can be expressed with a subdivision matrix \mathbf{A} , n -times subdivision corresponds to a matrix \mathbf{A}^n derived by repeated multiplication of \mathbf{A} . By utilizing the eigenstructure of \mathbf{A} , this matrix power can be computed in constant time. As a direct consequence, the control points of a Bézier patch corresponding to that regular face of \mathcal{M}^n which contains a certain sample point can be derived from the base mesh \mathcal{M}^0 in constant time and without having to execute any subdivision steps. In practice, for each valence k , the eigenstructure of the subdivision matrix \mathbf{A}_k is precomputed. During runtime, the base mesh vertices are first projected into the eigenspace of \mathbf{A}_k . Then, for each sample point, the required subdivision depth n is determined, and a linear combination of the eigenvalues raised to n and the projected base mesh vertices is computed.

Stam only discussed the method for subdivision surfaces without boundaries, for both Catmull-Clark [364] and Loop subdivision [363]. The technique was later extended to piecewise-smooth subdivision surfaces with boundaries and demonstrated on Loop's scheme [418]. Moreover, for (smooth) Catmull-Clark subdivision surfaces with boundaries another extension exists [201].

6.3.2 Approximation using Bézier surfaces

Although the direct-evaluation methods just presented in the previous subsection, especially the latter ones, are efficient and in principle well suited for GPU-based parallel execution, they are more complex, more expensive and slower than direct-evaluation approaches for Bézier surfaces. For rendering purposes, it hence appears desirable to approximate subdivision surfaces by Bézier surfaces.

⁴The only exception are sample positions which correspond exactly to an extraordinary vertex. However, the limit position of a base mesh vertex can be evaluated utilizing limit masks, anyway. Alternatively, such a critical sample may be moved slightly away from the vertex.

⁵This also gives rise to the following rendering approach [34, 92]: render regular parts and subdivide irregular faces unless they are small enough to just render the face as polygon; process the newly generated faces recursively.

Since the Catmull-Clark scheme is based on B-spline surfaces and is also of high practical relevance, it is a natural candidate for such an approximation. The *PCCM* (**p**atching **C**atmull-**C**lark **m**eshes) approach [292] constructs a bicubic NURBS patch for each quad face of the coarsest quad-only mesh \mathcal{M}^j . The surface described by the composition of the NURBS patches is C^2 -continuous everywhere except near extraordinary vertices, where it is at least C^1 -continuous. These are also the only regions where it differs from the subdivision surface \mathcal{M}^∞ . PCCM requires a sufficient separation of the extraordinary vertices, necessitating one or even two initial subdivision steps.

A different approach targeted specifically towards GPU implementation is pursued by Loop and Schaefer [230] with their *ACC* (**a**pproximating **C**atmull-**C**lark) technique. They give up C^1 continuity across Bézier patches and, similar to the PN triangle technique, separate the geometric from the normal component. This decoupling often results in smaller geometric and normal approximation errors compared to PCCM. ACC constructs a bicubic Bézier patch, called geometry patch, for each quad face of \mathcal{M}^j . In case of regular faces, a geometry patch matches the subdivision surface exactly. To achieve visual smoothness even near extraordinary vertices, two tangent patches of degree 2×3 each are additionally derived. Their cross product gives a shading normal field which is continuous everywhere. Unlike PCCM, ACC doesn't require any initial subdivision steps to isolate extraordinary vertices; only if the base mesh contains any non-quad facets, a single subdivision is needed. Because ACC operates only locally on a face's one-ring neighborhood, the conversion to geometry and tangent patches can be well parallelized and executed on the GPU. Recently, Kovacs et al. [193] extended the ACC scheme to further support piecewise-smooth subdivision surfaces with creases and corners.

By utilizing another primitive in addition to (bicubic) Bézier patches, Ni et al. [269] generate a G^1 -continuous approximation of Catmull-Clark subdivision surfaces. As usual, for regular quad faces a bicubic Bézier patch is constructed. However, an irregular quad face with up to four extraordinary vertices gets converted to a composite patch. Such a *c-patch* has cubic boundaries and is equivalent to a closed fan of four quartic Bézier triangles. It is defined by only 24 control points because for each of the four sub-triangles only the three interior control points as well as the four control points (of which two are shared by two triangles) defining the cubic boundary curve are actually required. The exterior control points can be derived by degree-elevating the boundary curve or by averaging control points from two adjacent triangles, respectively. In general, the component triangles of a c-patch meet only with C^1 continuity, and two c-patches join with G^1 continuity. Note that the approach was explicitly designed for efficient GPU implementation. In particular, the computation of the control points for the Bézier patches and c-patches can be structured into two phases which directly map to the vertex and geometry shader stages. Moreover, the resulting piecewise-polynomial approximation has a compact representation, requiring even fewer control points than ACC.

The technique was later extended to also directly support triangular and pentagonal faces as well as polar configurations [262]. Like non-regular quads, triangles and pentagons are converted to a *P_m-patch*. Generalizing the c-patch, a *P_m-patch* has m cubic boundary curves and is composed of m quartic triangular Bézier patches that form a closed triangle fan. It is described by $6m + 1$ control points analogous to the c-patch, with the difference that it explicitly stores the corner control point common to all component triangles, necessitated by the support for facets with $m \neq 4$ sides. If a triangular face appears in polar configuration, it is not converted to a *P₃-patch* but to a degenerate bicubic Bézier patch where one boundary curve is contracted to the single pole point. The approximation scheme also supports (semi-)sharp features, augmenting each vertex of the base mesh by a sharpness parameter α for each incident edge.

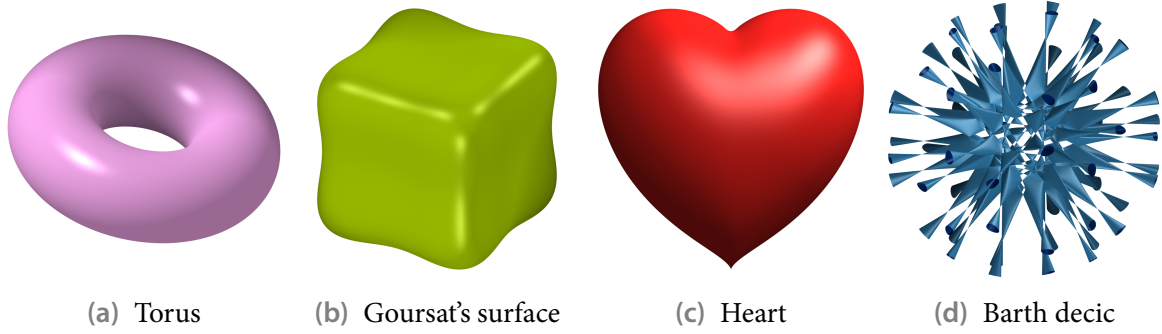


Figure 6.9 Examples of algebraic surfaces. (a) Torus (degree 4): $(x^2 + y^2 + z^2 + R^2 - r^2)^2 - 4R^2(x^2 + y^2) = 0$, $R = 2$, $r = 1$. (b) Goursat's surface (degree 4): $x^4 + y^4 + z^4 + a(x^2 + y^2 + z^2)^2 + b(x^2 + y^2 + z^2) + c = 0$, $a = 0$, $b = -2$, $c = -1$. (c) Heart (degree 6): $(x^2 + \frac{9}{4}y^2 + z^2 - 1)^3 - x^2z^3 - \frac{9}{80}y^2z^3 = 0$. (d) Barth decic [27] (degree 10): $8(x^2 - \tau^4y^2)(y^2 - \tau^4z^2)(z^2 - \tau^4x^2)(x^4 + y^4 + z^4 - 2x^2y^2 - 2x^2z^2 - 2y^2z^2) + (3 + 5\tau)(x^2 + y^2 + z^2 - w^2)^2(x^2 + y^2 + z^2 - (2 - \tau)w^2)^2w^2 = 0$, $w = 1$, $\tau = \frac{1}{2}(1 + \sqrt{5})$.

For triangular subdivision schemes like Loop, a simpler and computationally cheaper approach is QAS (**q**uadratic **a**pproximation of **s**ubdivision surfaces) [50]. Building on the PN triangle technique, each face of the base mesh is approximated by two quadratic Bézier triangles, one for the geometric component and one for the shading normal field. Initially, a single subdivision step is performed, the resulting mesh vertices are projected to the limit surface, and the corresponding limit normals are determined. Taking only the four sub-triangles created for each base mesh face as input, the control points for the two Bézier triangles are computed such that all sub-triangle vertices are interpolated by a corner or a boundary curve, respectively. Note that this straightforward local fitting only guarantees C^0 continuity. Furthermore, not least because of using only a quadratic surface, the approximation in general doesn't match the subdivision surface except at vertices of \mathcal{M}^1 .

6.4 Algebraic surfaces

Apart from defining surfaces via parametric or recursive formulations, they may also be described implicitly by the zero set of some scalar function $f(\mathbf{x})$ of points $\mathbf{x} = (x, y, z)^T$ in model space. More formally, such an *implicit surface* [37, 386]

$$S = \{\mathbf{p} \in U \subseteq \mathbf{R}^3 \mid f(\mathbf{p}) = 0\}$$

is specified by a domain $U \subseteq \mathbf{R}^3$ and a function $f : U \rightarrow \mathbf{R}$. Its normal is given by the gradient:

$$\mathbf{n}_S(\mathbf{x}) = \nabla f(\mathbf{x}) = \left(\frac{d}{dx}f(\mathbf{x}), \frac{d}{dy}f(\mathbf{x}), \frac{d}{dz}f(\mathbf{x}) \right)^T.$$

If f is a polynomial function, the surface S is an algebraic variety of dimension two and is called an *algebraic surface*.

Some examples demonstrating the expressive power of algebraic surfaces are shown in Fig. 6.9. It is also possible to exactly describe (rational) Bézier surfaces as algebraic surfaces, but the degree of f becomes high quickly. In general, a tensor-product patch of degree $m \times n$ has an implicit representation of degree $2mn$, and a triangular patch of degree n results in

an algebraic surface of degree n^2 [244, 341]. Note that the actual algebraic degree is lower in presence of base points, i.e. (possibly complex and infinite) parameter values (u, v) where the homogeneous version of the Bézier surface vanishes, i.e. $\mathbf{b}(u, v) = (0, 0, 0, 0)^T$. As an example, from the 32 bicubic patches defining Newell's teapot [81], only 8 have degree-18 implicit equations; the remaining ones have implicit representations of degree 9 (16 patches), 13 (4 patches) and 15 (4 patches), respectively [344].

Unfortunately, the high degree and especially the non-intuitive relationship of f 's polynomial coefficients to the shape of the surface make general algebraic surfaces inappropriate as a primitive for modeling curved surfaces. These shortcomings are addressed by *piecewise algebraic surfaces* [340], compositions of algebraic surface patches. Such a patch of degree n is defined by a tetrahedron with vertices $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3, \mathbf{P}_4$, and a scalar function of the tetrahedron's barycentric coordinates $(u, v, w, 1 - u - v - w)$ given by an n -degree Bézier tetrahedron

$$b(u, v, w) = \sum_{i+j+k+l=n} b_{ijkl} B_{ijkl}^n(u, v, w),$$

where

$$B_{ijkl}^n(u, v, w) = \binom{n}{i, j, k, l} u^i v^j w^k (1 - u - v - w)^l = \frac{n!}{i!j!k!l!} u^i v^j w^k (1 - u - v - w)^l$$

are the trivariate Bernstein polynomials. The isosurface $b(u, v, w) = 0$ confined to the tetrahedron (i.e. $u, v, w \geq 0$ and $u + v + w \leq 1$) describes the patch's surface. Its shape is controlled in a meaningful way by the $\frac{1}{6}(n+3)(n+2)(n+1)$ control weights b_{ijkl} , where each weight influences the function $b(u, v, w)$ most directly near its corresponding control point

$$\mathbf{p}_{ijkl} = \frac{i}{n}\mathbf{P}_1 + \frac{j}{n}\mathbf{P}_2 + \frac{k}{n}\mathbf{P}_3 + \frac{l}{n}\mathbf{P}_4.$$

Note that the formulation in terms of a Bézier simplex has the nice property that many Bézier techniques for parametric curves and surfaces can easily be adapted. In particular, multiple algebraic surface patches may be joined together with a desired cross-boundary continuity by imposing constraints on the control weights. To simplify such constructions, techniques like macro patches [342] have been developed.

6.4.1 GPU-based raycasting

All previously discussed curved surface primitives can essentially be represented or at least closely approximated by Bézier surfaces. Algebraic surfaces, however, in general defy a direct conversion. Although it is possible to derive a piecewise-polynomial approximation, for instance a piecewise-linear one with the marching cubes algorithm [232], it is usually of low quality, or consists of a huge number of patches, or requires a sophisticated and time-consuming process.

Not least due to such issues, raycasting [155] is the most common and probably the most natural approach for directly rendering algebraic surfaces, albeit a variety of other methods exist [188]. In contrast to parametrically defined surfaces, the ray-surface intersection equation to solve is univariate (ray parameter) and not multivariate (parametric coordinates), facilitating an efficient implementation. Because of this special status of algebraic surfaces, we briefly review GPU-based raycasting techniques achieving real-time frame rates. Note that Sec. 6.5.2 discusses raycasting for the other, non-implicit surface primitives.

For piecewise algebraic surface patches of degree $n \leq 4$, Loop and Blinn [229] render the projected triangular faces of a patch's defining tetrahedron. In the pixel shader, the ray-surface intersection is computed using an analytic root finding method, discarding the fragment if no intersection is found. However, sometimes an existing real root is missed and hence a pixel-sized gap introduced, which leads to visual artifacts. To alleviate these, a numerically more robust root finding algorithm was devised later [36].

Stoll et al. [367] present an optimized method for raycasting of quadratic algebraic surfaces whose spatial domain is bounded by a sphere or a tetrahedron. Arbitrary implicit surfaces are targeted by Knoll et al. [189], who employ interval and affine arithmetic to ensure numerical robustness.

By rendering proxy geometry for the domain of the algebraic surface and performing the actual raycasting in a pixel shader, all these approaches fit into the standard graphics pipeline. In contrast, Reimers and Seland [312] use a CUDA-based method and raycast the complete frame, depicting a single arbitrary algebraic surface of potentially high degree like those in Fig. 6.9. At first, the polynomial f is put into so-called frustum form by compositing it with a trilinear mapping from the unit cube to the view frustum. This allows efficiently deriving a univariate representation $g_{xy}(t)$ in Bézier form for the intersection equation of each ray (corresponding to pixel (x, y)). Its closest root is then computed by repeated refinement of the control polygon of g_{xy} using B-spline knot insertion.

6.5 Rendering approaches

After having given an overview of the most relevant primitives for curved surfaces in the preceding sections, we discuss the three major rendering approaches—tessellation, raycasting, and direct rasterization—in more detail. Recall that except algebraic surfaces basically all considered surface descriptions can be represented as Bézier surfaces or well approximated by them. Bézier surfaces have a compact, uniform representation of small and bounded size, unlike e.g. general B-spline surfaces. Their evaluation is also straightforward and doesn't involve determining relevant sub-parts of the description like a hit knot interval. On the other hand, to describe a complex surface, a whole collection of Bézier surfaces is required. However, for rendering purposes, this is actually an advantage because such a decomposition into independent patches naturally provides a reasonably fine granularity for both locally adapting rendering efforts and efficient parallel execution. For all these reasons, Bézier surfaces are the primary primitive for rendering curved surfaces. We thus mainly focus on them during the following discussion.

6.5.1 Tessellation

The dominating and historically probably first approach for rendering curved surfaces is tessellation. At first, an approximation by a polygonal mesh is derived for the surface. Usually, the mesh vertices are chosen to interpolate the surface and are hence determined by sampling the surface. If the mesh is composed of non-triangular faces, these are subsequently triangulated. The resulting triangular mesh provides a piecewise-linear approximation, both of the geometry and of the normal field. Finally, the mesh is rendered directly. Note that in practice mesh generation and rendering may be interleaved, i.e. once a part of the surface got tessellated, it may be rendered immediately.

Because current standard graphics hardware is optimized for rasterizing triangles, tessellation-based approaches are well suited for real-time rendering. To allow arbitrary changes of viewpoint and scene content, an appropriate approximating triangle mesh must be derived anew each frame, possibly incrementally modifying an existing one to exploit temporal coherence. In general, different parts of the surface, for instance different patches, can be treated independently from each other and hence be processed in parallel. The tessellation rate, i.e. the number of triangles created for a certain part of the surface, can vary locally and should ideally be chosen as small as possible to still achieve visual smoothness, especially at silhouettes. Such an adaptive tessellation keeps memory requirements low and enables a good utilization of the GPU. On the other hand, care must be taken that no cracks in a surface's approximation are introduced if two adjacent parts are tessellated to different levels. Issues like these and possible solutions are covered in the next chapter, which is dedicated to adaptive tessellation.

General methods

One way to approach tessellation is repeated subdivision. Each patch can trivially be approximated by a polygon constructed from the patch's corner control points. If the resulting approximation error is too large, the patch is subdivided into two or more sub-patches. This procedure is applied recursively to the newly generated sub-patches. In early work, Catmull [65] performs subdivision for bicubic polynomial patches until the resulting quads become smaller than one pixel. Such a generation of sub-pixel-sized micropolygons is also central to the Reyes rendering technique [78], where each geometric primitive is successively split or subdivided until it can be diced, i.e. converted to a regular grid of micropolygons of roughly equal size in screen space. Note that one major motivation for (sub-)pixel-level subdivision was the applicability of flat shading without sacrificing visual quality. In the majority of cases, however, it is reasonable to stop subdivision once an acceptable geometric approximation error is reached [75, 129]. As further pointed out in Sec. 7.2, recursive approaches are challenging to realize efficiently on current-generation GPUs. This is also one main reason for recent research efforts of converting subdivision surfaces, for which repeated subdivision is the most natural way of obtaining an approximation for rendering, to Bézier surfaces.

Another approach which avoids recursion is to first determine the tessellation rate for each patch, and then generate a uniform tessellation by regularly sampling the surface. While this allows only for adaptivity on an inter-patch level but not within a patch, it offers a finer-grained control of the sample spacing compared to regular repeated subdivision. To avoid cracks, per boundary curve the tessellation factor, i.e. the number of line segments used for approximating the curve, may be chosen separately. As an example, Rockwood et al. [315] present a modular technique for rendering trimmed tensor-product surfaces. After conversion to Bézier patches, for each patch, sampling step sizes in u and v parameter direction are derived that guarantee that the screen-space approximation error of the resulting triangle mesh satisfies a user-specified tolerance. Skipping the trimming-related steps, the interior of each patch is then uniformly tessellated according to the step sizes. Boundary curves are sampled separately to avoid inter-patch cracks and are connected to the interior with triangles. Algorithms in this spirit are currently the most promising ones for real-time rendering, since they can efficiently be mapped to graphics hardware.

Realizations

Surface tessellation is a long-standing task. Historically, most approaches are CPU-based, although sometimes efforts are made to realize some parts on programmable graphics hardware. Algorithms have been proposed for surface primitives like, for instance, (trimmed) NURBS surfaces [3, 199, 296, 315], Bézier triangles [55, 200] or subdivision surfaces [42, 259, 300, 325, 351]. Many CPU-guided approaches are not that well suited to be directly and fully implemented on the GPU because they are recursive or sequential in nature or require global knowledge. Moreover, they sometimes rely on surface-specific precomputations that limit general applicability, e.g. preventing modifications of the control points. A nice example that deprives a direct and efficient GPU realization and lacks flexibility but also highlights the range of possible approaches is the view-dependent adaptive tessellation algorithm of Chhugani and Kumar [69]. They incrementally precompute a good object-space sampling of each Bézier patch. These domain sample points are sorted according to their influence on reducing the geometric deviation and stored in a list. At runtime, a Delaunay triangulation of the required subset of samples is kept. To adapt to changing views, samples are added or removed incrementally. In case further samples beyond the precomputed ones are required, new samples are generated on the fly by uniform sampling.

With graphics hardware becoming ever more powerful regarding programmability, computational units, as well as memory bandwidth and space, GPU-based tessellation approaches [43, 48, 150, 337] are actively researched and now routinely outperform CPU-based ones if crafted well. Such techniques are discussed in more detail in the next chapter. In particular, we present our technical contributions, like a unified, completely GPU-based framework for high-performance tessellation of curved surfaces.

On the other hand, dedicated hardware solutions were proposed and sometimes even realized for several primitives, including tensor-product Bézier patches [110], PN triangles [71], and subdivision surfaces [9, 34]. Older-generation main-stream graphics hardware also featured some native support for selected primitives. NVIDIA's GeForce 3 [254] assisted the tessellation of Bézier patches, while ATI incorporated TruForm [18] into its products for deriving and tessellating PN triangles.

Such specialized approaches never caught on. Recent concepts aim at augmenting the pipeline by a dedicated tessellation unit which takes a primitive type (line, triangle, quad) and tessellation factors as input and outputs a corresponding generic tessellation of the related unit primitive in parameter space. Together with programmable shaders, this provides a flexible geometric synthesis stage on the graphics hardware. While the latest AMD GPUs like the Xbox 360's Xenos or the members of the Radeon R600 and R700 series [374] already feature such a unit, Direct3D 11 requires future hardware to provide a more versatile tessellation support, introducing two additional programmable pipeline stages (see Sec. 7.5.5).

6.5.2 Raycasting

Tessellation-based approaches seek to derive a piecewise-linear approximation of a surface that is good enough to be visually equivalent to the actual surface. By contrast, raycasting tries to determine an accurate (sub)pixel-wise sampling of the surface. For each (sample point of a) pixel, a ray is cast from the camera through the pixel into the scene. A ray-surface intersection test determines whether and at which position a curved surface is hit. Making this intersection calculation fast and robust is one main challenge, especially for real-time rendering.

Targeting bicubic polynomial patches, Kajiya [174] uses techniques from algebraic geometry, employing Laguerre's method for solving the arising univariate polynomial equations. In contrast, Toth [379] advocates multivariate Newton iteration and utilizes interval analysis techniques to determine safe starting regions in parameter space where Newton iteration will converge to a unique solution. Similarly, Sweeney and Bartels [370] also use Newton iteration but obtain good starting points for the iteration from a precomputed hierarchy of bounding boxes encompassing single vertices of the sub-patches resulting from an initial subdivision of the surface. Improving on this, Barth and Stürzlinger [28] adaptively subdivide the surface until all sub-patches can be well approximated by a planar parallelogram, and build a hierarchy of parallelepipeds tightly bounding the surface. Later, this approach was adapted to raycasting (trimmed) Bézier triangles [368].

An alternative technique is *Bézier clipping* [272], where regions of the parameter domain known to have no intersections are successively clipped away. Unlike Newton iteration and related root finding algorithms, Bézier clipping doesn't require a good initial guess of the intersection point for convergence. On the other hand, it suffers from wrongly reporting intersections [60], and may also report equivalent intersections multiple times, especially near degenerate boundaries [104, 105]. Improvements addressing these shortcomings were suggested [60, 104, 105]. Moreover, Bézier clipping, originally devised for tensor-product surfaces, was adapted for triangular Bézier patches [319, 320].

GPU-based realization

For raycasting (trimmed) Bézier patches, several fast CPU-based approaches exist [1, 32, 135] as part of interactive software raytracing systems. Since graphics hardware offers more computational power and higher memory bandwidth than CPUs, realizing raytracing systems on GPUs seems promising. But because the wide parallelism and the limited flexibility of current graphics hardware impose some challenging restrictions, current implementations [149, 167, 299] for raycasting scenes composed of triangles are not able to distinctly outperform their CPU-based competitors.

At least in the short run, however, most real-time rendering applications don't aim at completely substituting the standard rasterization-based graphics pipeline by a raytracing system. Instead, they seek to augment the current pipeline with curved surfaces possibly rendered via raycasting. As already mentioned in the context of algebraic surfaces in Sec. 6.4.1, the general integration approach is to render proxy geometry bounding the surface, thus generating fragments for all pixels covered by the surface, and to perform the raycasting in the triggered pixel shader.

Currently, such raycasting approaches, like that of Papst et al. [285], are not yet competitive with techniques using tessellation, both in terms of visual quality and particularly speed. One major issue is the convergence of the intersection finding method. Usually Newton iteration is employed, which may not converge if the starting point is too far off from the actual solution. In such cases, an intersection is missed, possibly introducing a visible gap in the surface. Because furthermore the number of required Newton steps depends on the starting point, determining a good initial guess is essential both for speed and quality. While a tight bounding volume hierarchy is well suited to derive a close estimate of the actual intersection point, its construction is far from free, obviating arbitrary per-frame changes to the surface geometry. Note that building such a hierarchy usually entails an adaptive surface subdivision and hence essentially involves deriving a (coarse) tessellation. Moreover, in the graphics-pipeline-based raycasting approach,

the leafs of such a hierarchy are actually rendered to trigger the pixel shader. To avoid incorporating tessellation-based surface rendering as part of the raycasting approach, the bounding proxy geometry should hence be simple and fast to determine on the fly. This, however, is in conflict with providing good starting points. Adopting Bézier clipping instead is not really an alternative because, although it can be implemented on the GPU [177, 285], it is complex and slow. Again, convergence can be sped up by subdividing the surface into smaller patches.

Another limitation of raycasting is that it doesn't directly and cheaply support antialiasing of silhouettes, while multisample antialiasing of rasterized triangles comes essentially for free on current graphics hardware. First, a raycasting pixel shader has to decide whether the whole fragment belongs to the surface or not, discarding it in the latter case. Only with Direct3D 10.1 it becomes possible to output a multisample coverage mask [338]. Second, determining accurate coverage requires casting multiple rays per fragment, which further slows down rendering compared to tessellation-based approaches.

Moreover, to correctly integrate raycast surfaces into the rendered scene, the pixel shader has to explicitly output the actual depth value as this usually differs from the proxy geometry's one. On current graphics hardware, this typically disables both the early-z test and z-culling, and hence can cause superfluous pixel shader executions.

Finally, note that even in case of immediate convergence at least one surface evaluation is required per processed fragment. Comparing this with exactly one surface evaluation per vertex encountered in a tessellation-based approach, raycasting can only win if fewer hitting rays are cast than tessellation vertices are output. Excluding excessive overtessellation, this implies that some parts of the curved surfaces are occluded and only the visible ones are actually raycast. But when just rendering proxy geometry and adapting the z value, such a scenario is not possible. On the other hand, using a raytracing system where the whole image is raycast and hence only the closest surface needs to be processed, raycasting may outperform tessellation-based approaches, at least in the long run.

Example: Raycasting of PN triangles

As a concrete example for GPU-based raycasting within the standard graphics pipeline, we describe our approach for PN triangles [332]. It was among the first presented solutions for raycasting Bézier surfaces on the GPU. We render a bounding proxy geometry for each PN triangle and perform a ray-PN-triangle intersection test for each fragment in the pixel shader.

To allow arbitrary changes of a PN triangle's control points during runtime, the proxy geometry has to be simple enough to be rapidly constructed anew each frame, for instance in the geometry shader stage. In particular, creating and drawing the geometry ought to be faster than the tessellation-based rendering of PN triangles. On the other hand, the proxy geometry should tightly bound the surface to keep the number of fragments low whose corresponding rays miss the PN triangle. We opted for a triangular prism whose base facets are parallel to the plane containing the corner control points. Exploiting the convex hull property of Bézier triangles, the prism is constructed to encompass the PN triangle's control points. To get a reasonably tight bound, we actually perform a 1-to-4 subdivision of the PN triangle and fit the prism to the control points of the resulting four sub-triangles. Note that due to the quadratic convergence of subdivision, further subdivision steps in general don't improve the tightness enough to justify the additional computational effort.

The proxy geometry triggers the actual raycasting process. In the pixel shader, we use Newton iteration to determine the intersection of the ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ originating at the camera

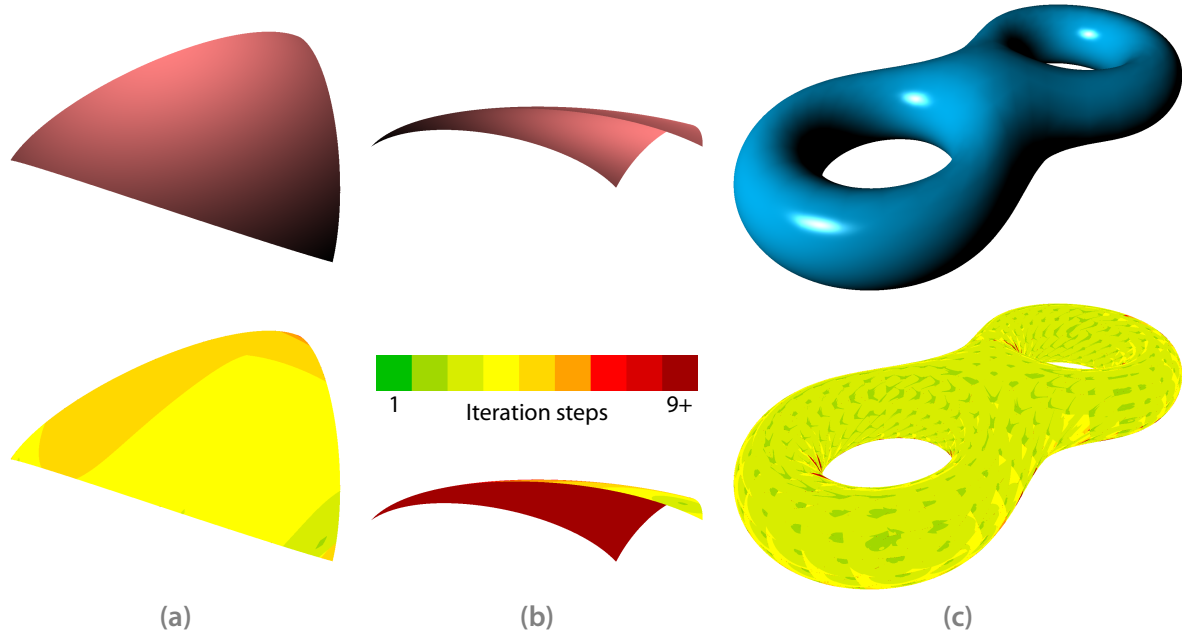


Figure 6.10 Examples of PN triangles rendered with our GPU-based raycasting approach. The lower row visualizes the total number of Newton iteration steps performed for intersection calculation when utilizing our initial guess heuristic.

position and passing through the pixel center with the PN triangle's geometry $\mathbf{b}(u, v)$. More precisely, we first represent the ray $\mathbf{r}(t)$ as intersection of two orthogonal planes [174]

$$\begin{aligned} P_1(\mathbf{x}) &= \langle \mathbf{n}_1, \mathbf{x} \rangle + d_1 = 0, \\ P_2(\mathbf{x}) &= \langle \mathbf{n}_2, \mathbf{x} \rangle + d_2 = 0. \end{aligned}$$

Finding the intersection nearest to the camera is then equivalent to determining the closest root of the distance function

$$\mathbf{D}(u, v) = \begin{pmatrix} P_1(\mathbf{b}(u, v)) \\ P_2(\mathbf{b}(u, v)) \end{pmatrix} = \begin{pmatrix} \langle \mathbf{n}_1, \mathbf{b}(u, v) \rangle + d_1 \\ \langle \mathbf{n}_2, \mathbf{b}(u, v) \rangle + d_2 \end{pmatrix},$$

i.e. the nearest point along the ray which has zero distance to the PN triangle's surface. An initial guess (u_0, v_0) of the intersection point is successively updated by Newton steps

$$\begin{pmatrix} u_{k+1} \\ v_{k+1} \end{pmatrix} = \begin{pmatrix} u_k \\ v_k \end{pmatrix} - \mathbf{J}_{\mathbf{D}}(u_k, v_k)^{-1} \mathbf{D}(u_k, v_k)$$

until either a hit is found within a given tolerance, i.e. $\|\mathbf{D}(u_k, v_k)\| \leq \varepsilon$, or the number of steps exceeds a specified upper bound N_{\max} . Note that the Jacobian of \mathbf{D}

$$\mathbf{J}_{\mathbf{D}}(u, v) = \begin{pmatrix} \langle \mathbf{n}_1, \mathbf{b}_u(u, v) \rangle & \langle \mathbf{n}_1, \mathbf{b}_v(u, v) \rangle \\ \langle \mathbf{n}_2, \mathbf{b}_u(u, v) \rangle & \langle \mathbf{n}_2, \mathbf{b}_v(u, v) \rangle \end{pmatrix}$$

can easily be inverted:

$$\mathbf{J}_{\mathbf{D}}^{-1}(u, v) = \frac{1}{\det \mathbf{J}_{\mathbf{D}}(u, v)} \begin{pmatrix} (\mathbf{J}_{\mathbf{D}}(u, v))_{22} & -(\mathbf{J}_{\mathbf{D}}(u, v))_{12} \\ -(\mathbf{J}_{\mathbf{D}}(u, v))_{21} & (\mathbf{J}_{\mathbf{D}}(u, v))_{11} \end{pmatrix}.$$

For a more detailed exposition of such intersection calculations, see, for instance, the description by Martin et al. [250].

Choosing a good starting point (u_0, v_0) is essential for convergence. However, due to our coarse bounding geometry, it is not possible to reliably estimate the intersection point—in contrast to setups where the leafs of a bounding volume hierarchy represent almost flat parts of the surface. In particular, since the PN triangle may be highly curved, it can happen that a ray barely misses one part of the surface but actually hits it farther away. Moreover, multiple surface intersections require finding the closest one of them, not just any one. To address these issues, we resort to the following heuristic. Taking a nearby parametric position for each of the three corners as well as the center as starting-point candidates, we project the corresponding surface points⁶ onto the ray and sort the candidates according to the ray parameter values t of their projections. Subsequently, a Newton iteration is performed for the first starting-position candidate. If it doesn't converge within N_{\max} iteration steps or if the surface is intersected outside the valid parameter domain, a new Newton iteration is initiated for the next starting-point candidate. Once a hit is reported, the remaining candidates are skipped, the surface normal is evaluated and the lighting is done. In case all four Newton iterations fail, we assume that the ray missed the PN triangle and discard the fragment.

Fig. 6.10 shows some examples rendered with our approach. With an NVIDIA GeForce GTX 280, we achieve frame rates of 230 Hz (a), 154 Hz (b) and 25 Hz (c), respectively, for them at a viewport of size 1600×1200. Note that these figures don't include the time for constructing the bounding prisms, which is done only once in a preprocess but could easily be realized in a geometry shader. We rarely if ever noticed missed intersections during our tests. In particular, even difficult cases like that in Subfig. b are handled well. However, they involve high Newton step counts, slowing down the intersection calculation. On the other hand, even for nicely facing PN triangles like in Subfig. a, we generally require more iteration steps than methods which derive an initial guess from a planar approximation of a leaf node in a bounding volume hierarchy. This is the price to pay for a simple proxy geometry and a heuristic for starting-point selection that quite robustly finds the correct intersection.

To verify the reasonability of our heuristic approach, we alternatively used just one Newton iteration and derived the starting point by taking the fragment's barycentric coordinates with respect to the prism's triangular base facet. Note that this roughly corresponds to the uv-texturing technique [285]. For the simple case in Fig. 6.10 a, this proves faster than our method (660 Hz). The remaining examples, however, suffer from missed intersections. These can be largely alleviated by subdividing the PN triangles multiple times and constructing separate bounding prisms for each sub-patch. But for the reasons outlined above, this is exactly what we want to avoid in the first place.

6.5.3 Direct rasterization

Yet another method to render curved surfaces is to directly rasterize them. Although such algorithms are currently no longer used nor in the focus of active research, they are of significant historical interest, having been among the standard approaches till the beginning of the 1990s.

The first representatives were scan-line algorithms. They progress row-wise and within each pixel row determine and process the line segments resulting from intersecting the surfaces with the scan line, i.e. the plane through the row's pixel centers and the camera point. Three different

⁶To avoid explicit surface evaluations, we actually take the three corner control points instead of the accurate near-corner points.

approaches were published in a joint paper by Lane, Carpenter, Whitted, and Blinn [204]. Supporting a wide range of directly evaluable and differentiable curved surface primitives, Blinn's method makes extensive use of Newton iteration. This numerical root finder is utilized to derive the range of rows covered by a surface, to compute the intersections of boundary curves and silhouette edges with the current scan line, as well as to determine the surface point at the current pixel center. On the downside, the approach is not robust and might fail in many special cases.

Whitted generalizes scan-line algorithms for polygons differently, evolving linear polygon edges to cubic edge curves of bicubic patches. In case of an excessively curved patch, additional edge curves are specified at interior isolines, effectively subdividing the patch. Similarly, to capture a patch's silhouette, (approximating) edge curve segments are introduced. All edge curves are decomposed into segments monotonic in vertical screen direction. Then, the single intersection of the current scan line with a certain edge curve segment is determined via Newton iteration, switching to a brute-force binary search if it doesn't converge. Whitted's method fails to find internal silhouettes, i.e. silhouette curves that do not intersect the boundaries of a patch.

Finally, Lane and Carpenter perform a kind of on-the-fly tessellation via adaptive subdivision. They subdivide bicubic patches overlapping the current scan line until all relevant (sub-) patches can be approximated by a planar quadrilateral. These are then processed like in a standard polygon scan-line algorithm. Since the non-uniform subdivision may introduce cracks, the approximation error threshold must be chosen small enough.

Rendering closely spaced isocurves

While such early scan-line algorithms have a low memory footprint, they are slow and suffer from robustness problems. Later approaches render surfaces as sequences of isocurves spaced close enough to avoid gaps. They thus sample a surface not along parallel scan lines in screen space but along parallel lines in parameter space.

Targeting tensor-product Bézier patches, Rockwood [314] first derives sampling step sizes in u and v parameter direction for each surface from the maximum distance of two successive control points. These patch-global step sizes are chosen such that for every pixel covered by the surface at least one sample is generated that maps to it. Ordinary forward differencing [390] is then used to efficiently emit the sample points and evaluate the surface at them. However, because the step sizes are fixed throughout the patch, some regions may be severely oversampled, causing significant pixel overdraw.

This shortcoming is alleviated by adaptive forward differencing [217]. While rendering an isocurve, the parametric step size is adaptively halved or doubled to maintain a sample spacing of roughly one pixel in screen space. Note that, nevertheless, the distance between two isocurves must be chosen close enough to avoid gaps, and hence oversampling can still occur. Thanks to a unified formulation as linear parameter substitution ($f(t) \rightarrow f(\phi(t))$), adaptive forward differencing directly augments fast traditional forward differencing ($\phi_E(t) = t + 1$) with the adaptivity of subdivision ($\phi_L(t) = \frac{1}{2}t$, $\phi_R(t) = \frac{1}{2}t + \frac{1}{2}$). The original approach for bicubic patches was later extended to arbitrary degrees and non-uniform (rational) B-splines [353]. This version also keeps the distance between two isocurves as a function in Bézier form to efficiently determine the inter-curve spacing. Steketee [366] presents an improved algorithm, observing that previous adaptive forward differencing methods are not completely correct as they may not generate a sample for each pixel overlapped by the surface.

The remaining issue of oversampling due to overlapping isocurves is addressed by Elber and Cohen [109]. Instead of rendering complete isocurves, they adaptively determine partial isocurves sufficient to cover each pixel with a sample. Starting with two boundary curves, two adjacent (partial) isocurves are recursively processed. First, a function of their distance is tested for roots. If none exist, and the two curves are already close enough, we are done. Otherwise, a new curve is introduced in between the two curves, and the new curve pairs are processed recursively. Finally, in case roots are found, the two curves are split at the roots, and the pairs of corresponding curve segments are treated recursively.

A different approach is taken by Rappoport [309] with his hybrid rendering algorithm. If a patch is deemed directly renderable, it is output with forward differencing using constant step sizes for intra- and inter-isocurve sample spacing. Otherwise, the patch is subdivided according to which direct rendering criterion is not satisfied, and the algorithm gets applied recursively. These criteria check for sufficient geometric flatness, uniform spacing of control points, and whether the required sample count is small enough to keep the numerical error accumulated during forward differencing within bounds avoiding artifacts.

Whereas the scan-line methods were not intended for hardware implementation, the ones based on (adaptive) forward differencing are amenable to such a realization and sometimes even were designed for it. However, creating some special hardware seems to make little sense because it would be restricted to a single primitive like bicubic tensor-product Bézier patches. On the other hand, efficient software solutions for such isocurve evaluation schemes are possible [67]. We reckon that on future hardware like Intel's Larrabee [349], where custom graphics pipelines can be crafted, rasterizing isocurves may become an option for some applications. A kind of coarse subdivision scheme may prove useful not only for reducing oversampling but also to generate screen-space-localized, parallelizable work items. The semantics of executing a shader instance per fragment also have to be sorted out. This is challenging because a pixel may be hit by multiple samples of the same surface. Moreover, the pixel-relative positions of the sample points in general vary per pixel, requiring some additional processing to enable consistent texturing.

CHAPTER 7

Adaptive tessellation

The ability to efficiently render curved surfaces in real time and within the standard rasterization-based graphics pipeline is key to enabling direct usage of curved surfaces as output primitive in graphics applications. Recall that this has many advantages, like only having to maintain a compact description or making animation easy and cheap. Tessellation-based approaches, which first convert a curved surface into a piecewise-linear approximation and then render the resulting triangles, are the most suitable and promising methods to achieve this goal.

In this chapter, we discuss adaptive tessellation in more detail. Striving to harness the power of modern graphics hardware to achieve high performance, we primarily focus on GPU-based approaches. Moreover, justified by the reasons outlined in Sec. 6.5, we concentrate on Bézier surfaces. After clarifying the objectives we aim for, like adaptivity, parallelizability, and watertightness, we study methods performing repeated subdivision until the approximation is locally good enough.

The rest of the chapter is dedicated to the competing approach of first determining the sampling density required to well approximate a surface patch and subsequently generating a tessellation by sampling the patch accordingly. We discuss both creating tessellation patterns which satisfy a given sampling density, and how to derive tessellation factors specifying the sample spacing. Concerning the actual rendering, one technique is to use generic tessellation patterns in parameter space, often called refinement patterns, and map them to object space by evaluating the surface at the parametric coordinates associated with the patterns' vertices. After dealing with this method in detail and describing our contributions, we present an alternative technique, our CudaTess framework. Using a surface patch as unit of parallelism, it runs all major steps on the GPU and generates an object-space tessellation on the fly. We conclude the chapter with a discussion and a comparison of these two different methods.

7.1 Objectives

When tessellating a curved surface, we ultimately strive for a triangular approximation that is efficient and fast to both determine and render, and that is visually reasonably close to the actual surface and not suffering from visual artifacts. In practice, this overall goal translates to several criteria and requirements.

Adapting sampling density

During the conversion into a triangular approximation, the sampling density should be high enough to ensure visual smoothness; in particular, silhouettes have to appear curved and not reveal their piecewise-linear approximation. On the other hand, (excessive) oversampling ought to be avoided for performance and quality reasons. A too fine tessellation takes longer to compute than necessary and consumes more memory space and bandwidth. Moreover, rendering the resulting triangles may become slower. Not only can the GPU's triangle setup stage turn into a bottleneck, but the occurrence of many small triangles covering just a single pixel or very few pixels reduces the efficiency of the rasterizer and the effective parallelism in the pixel shader stage. Recall that usually fragments are assigned in blocks of 2×2 to the shader processors to be able to compute derivatives required in selecting a texture mipmap level. In case of triangles resulting in a tiny fragment count, the fraction of the processors that are essentially idle is hence rather large. Finally, if several pixel-sized triangles are mapped to the same pixel, spatial and especially temporal aliasing may occur.

Consequently, the surface tessellation should be adaptive, i.e. the sampling density should locally be varied to both avoid undersampling and keep oversampling to a minimum. Note, however, that we don't seek to find the optimal sampling which has the lowest possible sample count needed to satisfy some error metric. While this may be reasonable in other application domains, our goal is to derive and render a sufficiently fine tessellation as fast as possible. In particular, to allow updating the surface description during runtime, for instance to animate the surface, it must be possible to efficiently and quickly determine the complete tessellation anew each frame.

Good parallelizability

Fundamental to fast execution on the GPU is the ability to parallelize the task. Moreover, dependencies among the parallel work items should be avoided or at least kept to a minimum. Usually, a complex surface is broken into multiple parts that can be processed independently; together the resulting partial tessellations then yield the overall piecewise-linear approximation. Independent processing, however, typically comes at the cost of redundancy, like evaluating the surface twice along curves where two surface parts meet. It is hence important to find a good balance between the granularity of parallelization and the associated overhead due to redundancy but also due to setting up the work items.

Note that since in our assumed setup complex surfaces are described by a collection of abutting Bézier patches or triangles, an initial surface decomposition is readily available. One possible way to achieve the combined objective of parallelizability and providing a reasonable adaptation of the sampling density to avoid excessive oversampling is uniformly sampling each Bézier surface patch but choosing a patch-specific sampling rate.

Avoiding visual artifacts

When treating multiple parts of a surface, for instance individual Bézier patches, independently from each other, care must be taken to avoid visual artifacts. In particular, the triangle mesh implicitly defined by the union of the partial tessellations should be watertight (cf. Fig. 7.1). That is, the two tessellations of two abutting surface parts should have identical vertices along the boundary curve where they meet.

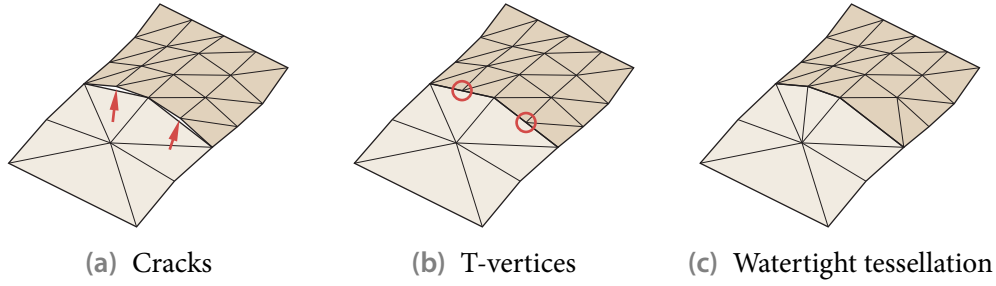


Figure 7.1 A watertight tessellation is free of cracks and T-vertices, both of which typically result in visual artifacts.

If a common boundary curve is sampled at different rates by the two adjoining patches, the two corresponding polyline approximations in general do not coincide and *cracks* are introduced into the surface. These lead to holes in the mesh or cause neighboring triangles to overlap, often resulting in severe visual artifacts.

But even if the two polyline boundary approximations coincide mathematically, i.e. all vertices of one polyline lie on the other one and vice versa, problems may arise. Vertices of one polyline which have no match on the other polyline, so called *T-vertices*, can evoke missing pixels, resulting from the limited numerical precision within the rasterizer stage. While this may be alleviated to a certain degree by increasing the number of coverage samples per pixel, i.e. enabling MSAA, minor artifacts still occur. Furthermore, T-vertices induce shading problems if a non-linear quantity is interpolated along the polylines. Also note that in case a displacement is applied to the tessellation, a T-vertex typically no longer lies on the other polyline and hence transforms into a crack. One way to circumvent missing pixels is to stitch the two polylines together with a strip of triangles, each connecting a vertex of one polyline with an edge of the other polyline. However, such zero-area triangles may cause artifacts when the surface is rendered with alpha blending, don't solve interpolation-related shading problems, and require additional geometry. Note that T-vertices may also result from input surfaces where two patches only partially share a boundary curve. In such settings, it is not enough to choose a consistent sampling rate but further special processing is required.

Finally, if a shared boundary curve is sampled at the same rate by the two neighboring patches but the involved parameterization directions differ, limited numerical precision may still cause mathematically equivalent points to be slightly different, potentially leading to visual artifacts. There are two sources of numerical deviation. First, the order in which terms are evaluated may differ for the two adjoining patches. This can be addressed by enforcing a consistent order of boundary control points across patches. Another option is reorganizing the evaluation computation into a symmetric form, which guarantees that the order in which corresponding non-zero terms are combined for samples on a boundary curve is independent of the parameterization direction. For instance, a bicubic Bézier patch may be computed as [62]

$$\mathbf{b}(u, v) = \left(\left((\mathbf{b}_{00}B_0^3(u) + \mathbf{b}_{10}B_1^3(u)) + (\mathbf{b}_{20}B_2^3(u) + \mathbf{b}_{30}B_3^3(u)) \right) B_0^3(v) + \left(\sum_{i=0}^3 \mathbf{b}_{i1}B_i^3(u) \right) B_1^3(v) \right) \\ + \left(\left(\sum_{i=0}^3 \mathbf{b}_{i2}B_i^3(u) \right) B_2^3(v) + \left((\mathbf{b}_{03}B_0^3(u) + \mathbf{b}_{13}B_1^3(u)) + (\mathbf{b}_{23}B_2^3(u) + \mathbf{b}_{33}B_3^3(u)) \right) B_3^3(v) \right),$$

where parentheses are meant to enforce computation order. Note that such a formulation usually entails an increased arithmetic operation count.

The second source of numerical difference is that corresponding parametric sample coordinates may not match exactly; for instance, $\frac{i}{n}$ is numerically typically not the same as $1 - \frac{n-i}{n}$ for a non-power-of-two sampling rate n . Any numerical inaccuracy problems can be avoided by actually sharing the common boundary vertices, i.e. using the ones computed for one patch for the tessellation of both patches. Note that this requires a synchronization among the two associated parallel work items after having independently performed the patch tessellation. A simple realization just copies the vertex data from one patch's tessellation to the other one's, or merely the positions if other vertex attributes like color don't match.

While cracks but also the occurrence of T-vertices routinely cause artifacts, we never experienced problems due to an inconsistent evaluation order. Therefore, we don't bother to address numerical evaluation inaccuracies in the remainder of this chapter, noting that a remedy could easily be incorporated at the cost of probably marginally reducing performance.

Finally, since long and thin triangles can cause shading problems, it is desirable that the triangles in a tessellation are well shaped, ideally being roughly equilateral. We further note that additional care is typically required to avoid artifacts when applying displacement mapping [62], which is beyond our scope, though.

7.2 Recursive refinement

As mentioned before, one main method for obtaining a tessellation is repeated subdivision. For each patch, a polygon can be constructed from its corner control points. Subsequently, this initial approximation is recursively refined by subdividing the patches and splitting the polygons accordingly until the resulting approximation is considered good enough.¹ Note that we refer to initial input patches as *base patches*, and usually treat the sub-patches resulting from subdivision as individual patches.

7.2.1 Refinement criteria

The decision whether and how to subdivide a patch (and split its polygon) and thus refine the approximation is guided by refinement criteria, which are usually completely local to a patch. They typically check for flatness of the boundary curves of the patch as well as of its interior part. If these are deemed sufficiently linear or planar, respectively, the approximation is considered fine enough and the recursion stops.

Flatness is often determined using the maximum screen-space distance of a boundary curve to its linear approximation, or of a patch to its triangular approximation, respectively, with polygonal faces being triangulated. In practice, it suffices to just compute an estimate of this distance bound. A simple example is to take the parametric midpoint along a boundary curve and determine its distance to the line segment by which the curve currently gets approximated. However, this may underestimate the overall deviation, especially for S-shaped curves. Alternatively, a conservative bound can be derived by considering the distance of the control points to the line segment, thus exploiting the convex hull property of Bézier curves. Analogously, approximation distance estimates can be determined for patch interiors. Note that to simplify

¹In principle, it suffices to only split the polygons and compute the newly introduced vertices from the input patch without subdividing it. While this keeps storage requirements low and allows fast refinement, it often complicates the procedure employed for deciding whether the approximation needs further refinement, since this decision is usually based on properties of the related subpart of the surface, for which now no explicit representation exists. In practice, this approach is hence rarely considered for adaptive tessellation.

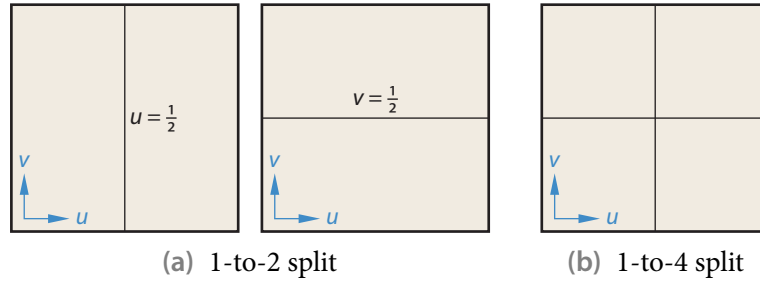


Figure 7.2 Subdivision cases for a rectangular domain.

computations, often the point distances are determined not with respect to a line segment or a triangular face, respectively, but to the whole corresponding line or plane, potentially underestimating the approximation error.

Using just point-to-surface distances to judge flatness may fail in some special cases like loops in boundary curves [129], which, however, can be detected and handled appropriately if desired. Another option is to compute point-to-point distances, for instance of control points to their corresponding points in the linear approximation. In case of a degree- n Bézier triangle, the control point \mathbf{b}_{ijk} would be compared against $1/n(i\mathbf{b}_{n00} + j\mathbf{b}_{0n0} + k\mathbf{b}_{00n})$, which is the corresponding control point of the triangular approximation elevated to degree n . Note that such an approach not only keeps the geometric deviation bounded but also the parameterization approximation error, which is desirable for texturing.

Another way to test for flatness of a boundary curve is to compare the length of the polyline formed by its control points against the length of the approximating line segment [204]. Similarly, a patch may be considered flat if the deviation of the surface area of the control net from the area of the approximating polygon is within a tolerance.

Apart from flatness, which is by far the most widely used criterion, often additional criteria are employed that prevent refinement if the patch is not visible, for instance because it is outside the view frustum. Another extension may choose the deviation threshold below which a flat approximation is considered acceptable to be larger for non-silhouette regions. Some applications also require different or further criteria, like one keeping the maximum screen-space coverage per approximating polygon bounded. Finally, note that in case a displacement map is applied, specific refinement tests exist [94, 257].

7.2.2 Refining a rectangular domain

Depending on the domain shape of a surface patch, different options of how to subdivide the patch and hence split its approximating polygon exist. We first consider patches with a rectangular domain, like tensor-product Bézier patches, and cover the triangular case later in Sec. 7.2.3.

For rectangular domains, the basic refinement is a 1-to-2 split, which subdivides the patch along one of the center isolines $u = 1/2$ or $v = 1/2$.² By compositing two such bisections, one in u and one in v direction, a patch may also directly be split into four sub-patches (see Fig. 7.2).

²To keep the presentation clear and the expressions short and simple, we always implicitly refer to the unit domain $[0, 1]^2$ throughout this chapter. Note that expressions for an arbitrary rectangular domain $[u_0, u_1] \times [v_0, v_1]$ are easily obtained via the mapping $\xi \mapsto (1 - \xi)\xi_0 + \xi\xi_1$, $\xi \equiv u, v$. The analogous holds for the triangular case.

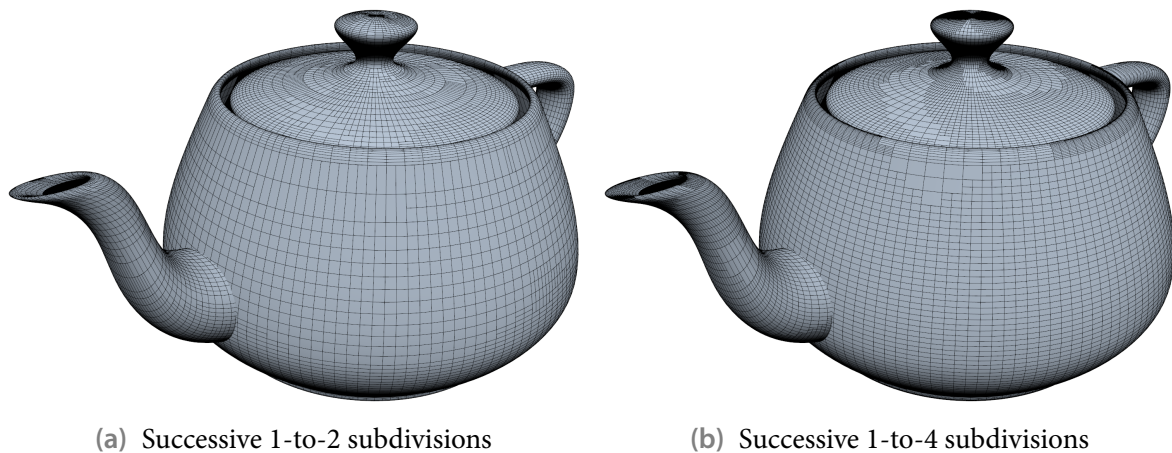


Figure 7.3 Adaptively refined Utah teapot. Recursively subdividing patches into two sub-patches results in fewer primitives (15771 vs. 24413 quads) but requires more subdivision steps (up to 11 per base patch vs. 6) than splitting into four sub-patches.

A common approach [75] is now to first check whether any of the boundary curves along u direction ($v = 0, v = 1$) is not sufficiently flat, in which case the patch gets subdivided at $u = 1/2$. Otherwise, an analogous flatness test is performed for the boundary curves in v direction ($u = 0, u = 1$), splitting the patch along $v = 1/2$ in case of failure. If all boundary curves are considered adequately flat, the patch itself is checked for flatness. In case it passes, the approximation is good enough and the refinement stops; a quad is constructed from the patch's corner control points and emitted for rendering. If, however, the patch is not sufficiently flat yet, it is bisected along either u or v parameter direction. Ideally, that split direction is chosen whose corresponding edges are longest in screen space, resulting in more square-like quads. The whole procedure is applied recursively to the newly generated sub-patches.

To reduce the number of subdivision steps necessary and hence the depth of recursion, one may directly perform a 1-to-4 split if both in u and in v direction at least one boundary curve is not flat. Sometimes, it is also desirable to exclusively use 1-to-4 splits, in which case a single flatness test for the whole patch suffices.

We implemented tessellation by recursive subdivision for bicubic Bézier patches. Flatness is assessed using the distance of the control points to the corresponding points in the related (triangulated) quad approximation. We perform either only 1-to-2 splits or exclusively 1-to-4 subdivisions. Fig. 7.3 shows an example using both variants.

Cracks and T-vertices

Because each (sub-)patch is processed individually and the recursion depth is controlled by flatness and is hence not globally uniform, the union of the resulting quads in general yields a non-watertight tessellation suffering from cracks and T-vertices. Unfortunately, these artifacts cannot completely be avoided unless each patch keeps some information about its neighborhood. Alternatively, some global post-processing step may be run.

Since operating completely patch-locally simplifies implementation and parallelization, one popular class of methods keeps this locality but in consequence only prevents cracks but not T-vertices and thus accepts some remaining artifacts. We detail such crack-avoidance approaches later in Sec. 7.2.4.

Note that the repeated subdivision of a patch implicitly defines a tree. One technique that actually yields a watertight tessellation but requires some global control enforces this tree to be a restricted quadtree [388]. That is, only 1-to-4 splits are allowed and two leaf nodes adjacent in parameter space may differ by at most one tree level. To satisfy this constraint, some nodes may have to be further subdivided despite already being flat enough. Once the quadtree is built, appearing cracks in the corresponding tessellation are efficiently filled. More precisely, whenever a quad of subdivision level i abuts in parameter space along an edge to two quads of level $i + 1$, the coarser-level quad is refined by adding a vertex at the T-junction. Subsequently, the modified quad face is retriangulated, for instance by introducing a central vertex and building a triangle fan.

Fast subdivision

It is worth mentioning that to rapidly perform subdivision, special patch representations have been developed. Catmull [65] proposes the polynomial basis defined by the functions

$$C_0(t) = 1 - t, \quad C_1(t) = -\frac{1}{3}t^3 + t^2 - \frac{2}{3}t, \quad C_2(t) = \frac{1}{3}t^3 - \frac{1}{3}t, \quad C_3(t) = t$$

to represent cubic curves. A midpoint subdivision at $t = 1/2$ can then be realized with only three adds and four shifts.

Equivalently, Clark [75] suggests employing *central differencing*, which we briefly review here due to its wide-spread use. Considering a cubic curve $\mathbf{c}(t)$, its Taylor expansion

$$\mathbf{c}(t + \Delta t) = \mathbf{c}(t) + \Delta t \mathbf{c}'(t) + \frac{1}{2} \Delta t^2 \mathbf{c}''(t) + \frac{1}{6} \Delta t^3 \mathbf{c}'''(t)$$

leads to an evaluation expression as central difference along with a higher-order term:

$$\mathbf{c}(t) = \frac{1}{2}(\mathbf{c}(t - \Delta t) + \mathbf{c}(t + \Delta t)) - \frac{1}{2} \Delta t^2 \mathbf{c}''(t).$$

This can be used to efficiently compute the midpoint at $t = 1/2$ during subdivision from the endpoints $\mathbf{c}(0)$ and $\mathbf{c}(1)$, choosing $\Delta t = 1/2$. Therefore, given a bicubic patch $\mathbf{b}(u, v)$, a split in u direction involves determining

$$\mathbf{b}(\frac{1}{2}, v) = \frac{1}{2}(\mathbf{b}(0, v) + \mathbf{b}(1, v)) - \frac{1}{8} \mathbf{b}_{uu}(\frac{1}{2}, v)$$

for both $v = 0$ and $v = 1$. The required second-order partial derivative

$$\mathbf{b}_{uu}(\frac{1}{2}, v) = \frac{1}{2}(\mathbf{b}_{uu}(0, v) + \mathbf{b}_{uu}(1, v))$$

is also computed by central differencing. Because a later, analogous subdivision in v direction needs the values of $\mathbf{b}_{vv}(u, v)$ at the patch corners, this quantity must be updated, too. Since $\mathbf{b}_{vv}(u, v)$ is cubic in u like $\mathbf{b}(u, v)$, it is computed by

$$\mathbf{b}_{vv}(\frac{1}{2}, v) = \frac{1}{2}(\mathbf{b}_{vv}(0, v) + \mathbf{b}_{vv}(1, v)) - \frac{1}{8} \mathbf{b}_{uuvv}(\frac{1}{2}, v),$$

using central differencing to determine the fourth-order mixed derivative

$$\mathbf{b}_{uuvv}(\frac{1}{2}, v) = \frac{1}{2}(\mathbf{b}_{uuvv}(0, v) + \mathbf{b}_{uuvv}(1, v)).$$

Consequently, a bicubic patch is represented by four corners instead of 16 control points, each storing the quantities $\mathbf{b}(u_{ij}, v_{ij})$, $\mathbf{b}_{uu}(u_{ij}, v_{ij})$, $\mathbf{b}_{vv}(u_{ij}, v_{ij})$ and $\mathbf{b}_{uuvv}(u_{ij}, v_{ij})$.

Note that the derivative terms can be used to quickly decide whether a further 1-to-2 subdivision is required and along which parameter direction. For instance, the boundary curve $u = 0$ may be considered sufficiently flat if all components of both $\mathbf{b}_{vv}(0, 0)$ and $\mathbf{b}_{vv}(0, 1)$ are below a certain magnitude threshold [75].

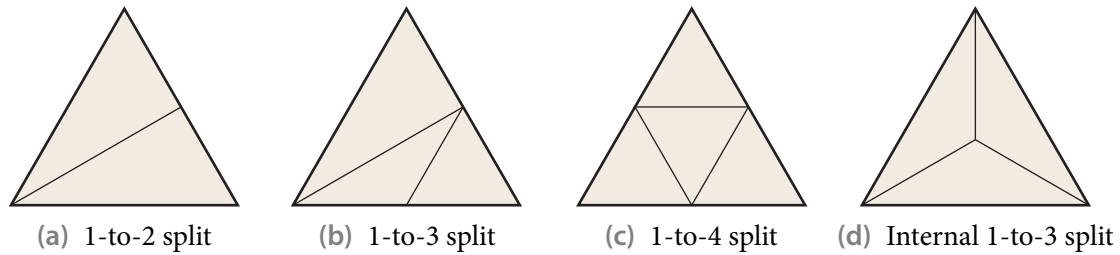


Figure 7.4 Subdivision cases for a triangular domain.

7.2.3 Refining a triangular domain

Compared to the rectangular case, the triangular domain offers a greater variety and hence flexibility concerning topological refinement. As depicted in Fig. 7.4, just a single boundary curve may be subdivided, or two, or all three. Alternatively, only the interior is refined without splitting the boundary curves.

To tessellate a triangular patch via repeated subdivision [129], one hence may first check all boundary curves for sufficient flatness. If at least one curve fails the test, a subdivision is performed that splits all non-flat boundary curves. Otherwise, the patch interior is checked for adequate flatness. In case of failure, the interior is subdivided but not the boundary curves. If all flatness tests are passed, the approximating planar triangle constructed from the patch's corner control points is output and the recursion stops. Note that the resulting tessellation doesn't suffer from cracks or T-vertices, since the subdivision of a boundary curve is guided by the curve alone and not influenced by the rest of the patch.

On the other hand, the resulting shapes of the triangles are often elongated and rather thin. This can be alleviated by using just 1-to-4 splits, producing only triangles equilateral in parameter space. However, such a shape improvement comes at the price of introducing cracks and T-vertices in the tessellation, like in the rectangular case.

An alternative that performs solely quadrisections for refinement but yields a watertight tessellation is *red-green triangulation* [21]. It essentially enforces that the resulting tessellation is restricted such that two adjacent triangles differ by at most one level of subdivision. Finally, all triangles are subdivided along those boundary curves where they adjoin triangles of greater subdivision level, thus stitching any cracks. Concerning terminology, a triangle is green if it is an initial base triangle or results from a 1-to-4 split. Triangles originating from any of the other subdivision cases are called red. A green triangle may be further refined, whereas a red one only serves to terminally make the tessellation conforming by closing cracks and hence must not be subdivided. Instead, the split generating the red triangle is reversed and a green 1-to-4 subdivision performed, possibly triggering further splits of the neighboring triangles. Note that such a scheme requires global control and is amenable to parallelization only to a limited extent.

Choosing PN triangles as example, we implemented recursive refinement using the distance of control points to the matching points on the approximating triangle as flatness measure. Fig. 7.5 shows results obtained both when utilizing all discussed splits as well as just quadrisections.

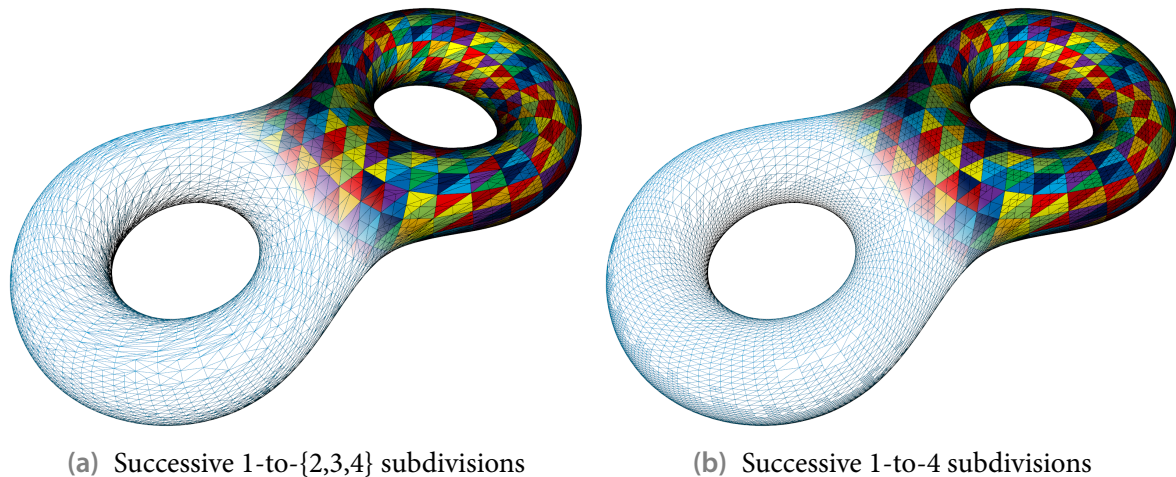


Figure 7.5 Adaptively refined double torus. Compared to just subdividing into four sub-patches, also allowing to split into two or three sub-patches results in fewer primitives (16540 vs. 22581 triangles) and yields a watertight mesh but requires slightly more subdivision steps (up to 4 per base patch vs. 3).

7.2.4 Locally handling cracks

Ideally, the decision whether to subdivide a patch is based only on information about the patch itself, thus allowing utmost parallelism. In particular, to ensure that shared boundary curves are tessellated consistently by both adjoining patches, a boundary curve should only be subdivided if indicated by the curve itself. This, however, is not guaranteed in the rectangular domain, where the decision to split a boundary curve also forces the opposing boundary curve to be subdivided. Moreover, splitting the patch interior necessitates subdividing two boundary curves, too. Similarly, if only quadrisections are used in the triangular setting, boundary curves needing no further refinement are enforcedly split. A consequence of such unwanted but topologically required subdivisions are inconsistencies across patches, manifesting themselves as cracks and T-vertices in the resulting tessellation.

Such artifacts can be alleviated by performing a global stitching after repeated subdivision is completed. Examples of related methods include the presented restricted quadtree approach and red-green triangulation. However, these usually elude an effective parallelization.

An alternative that avoids global bookkeeping but only partially solves the problem is suggested by Clark [75] who exploits that once a boundary curve is considered flat enough, it may be approximated by a straight line segment. Consequently, if the curve is further subdivided, nevertheless, the newly introduced midpoint vertex can safely be forced onto this line segment without violating sufficient flatness. Since this approach ensures that the final piecewise-linear approximation of a base boundary curve depends only on the curve itself, cracks are avoided. On the other hand, T-vertices still exist; in particular, each prevented crack results in a T-vertex.

In his algorithm, Clark actually adapts the control points to make the boundary curve itself a straight line segment. While simple, this modifies the patch and hence introduces an error [22], which visually results in shading discontinuities. These can be alleviated by using the original patch for shading, or alternatively a separate normal field patch as employed by PN triangles.

A better solution is to keep sub-patches unmodified and note which vertices are supposed to lie on a line segment, forcing them onto it only when the recursion is completed and a quad is output. This can be achieved by associating a line segment to a boundary curve that has become flat [22, 129], or by explicitly storing adapted corner points along with flatness flags for each boundary curve [296], or by utilizing edge equations [283].

7.2.5 GPU-based implementations

Executing a single subdivision step for many patches or control faces is easy to parallelize, since each such element can be processed individually. Hence, it is not surprising that several approaches exist which perform the actual subdivision on the GPU. On the other hand, adaptively controlling the subdivision level and especially generating a final watertight tessellation are usually still performed on the CPU.

Losasso et al. [234] perform only uniform subdivision, i.e. the whole surface is subjected to the same number of subdivision steps. Their method operates on a closed surface described by a single smooth geometry image which corresponds to a uniform bicubic B-spline patch. Its control net is stored in a texture. A single subdivision step consists of topological refinement, quadrisecting each control face, and subsequently adapting the control points by averaging. It is realized by two corresponding passes, each time rendering a filling quad into an appropriately sized texture, with each texel storing a control point computed by the triggered pixel shader. After the desired number of subdivision steps have been executed, the control points are projected onto the B-spline surface using limit masks, and the associated analytic normals are derived via limit tangent masks. Finally, the corresponding quad mesh is rendered.

As mentioned earlier, the most natural way to render subdivision surfaces is to perform repeated subdivision and output the resulting tessellation. Shiue et al. [357] present a related GPU-based approach which supports a wide range of subdivision schemes. Concentrating on Catmull-Clark subdivision, the input control mesh has to comprise only quads with at most one extraordinary vertex, possibly necessitating an initial subdivision step executed on the CPU. The input mesh is then decomposed into overlapping fragment meshes, each consisting of a center vertex (of valence k) and two rings of vertices surrounding it. Each fragment mesh is stored in a 1D texture, linearly arranging the vertices via spiral enumeration. A single subdivision step refines and enlarges the fragment mesh and is performed by rendering into an according 1D texture. In the invoked pixel shader, that subdivision mask is applied which yields the vertex corresponding to the current fragment. The masks are obtained from a lookup texture which for each element of the new fragment mesh stores a specific mask, comprising texel indices for accessing old fragment mesh vertices and the mask type. Note that for each supported valence k a separate mask lookup texture is required. For a maximum subdivision depth n_{\max} , this consists of $3\ell_k(n_{\max})$ RGBA texels, where $\ell_k(n) = 1 + k(1 + 2^n)(2 + 2^n)$ denotes the texture size for a valence- k fragment mesh after n subdivision steps. Once subdivision is completed, the vertices are copied into a vertex buffer and normals are computed. While this approach doesn't directly support adaptive subdivision, it is noteworthy that the masks are evaluated in a symmetric fashion to yield a watertight mesh.

In a related method, Bunnell [56] performs coarse-grained adaptive subdivision of Catmull-Clark surfaces. Pursuing a patch-based approach, the input control mesh is decomposed into smaller meshes that can be stored in a 2D texture along with their one-ring. For all patches, flatness is tested on the GPU and the results are read back to the CPU. Patches requiring further refinement are then subdivided on the GPU by drawing quads into new textures, applying the

subdivision masks in the pixel shader. Note that subdividing a patch results in another patch with a larger control mesh instead of individual sub-patches.³ Consequently, the subdivision level can only be varied among the initial patches but not within any of them. Once a patch is subdivided enough, limit surface positions and normals are determined and stored in a global vertex buffer. Finally, an index buffer is generated on the CPU which yields a tessellation free of cracks and T-vertices. This is possible because the subdivision process is controlled by the CPU, and hence global knowledge about the refinement depths and vertex indices exists.

More recently, Patney and Owens [288] utilized CUDA to realize a simplified subset of the surface subdivision stage of the Reyes rendering pipeline [78] on the GPU. They focus on the special case of bicubic Bézier patches and implement only that parts which are straightforward to parallelize. All patches are first repeatedly subdivided until the screen-space footprint of each sub-patch is at most 8×8 pixels; subsequently each sub-patch is diced into a grid of 16×16 micropolygons. The resulting sub-pixel-sized quads can then be rendered. More precisely, all base patches are initially provided in an input buffer. A CUDA kernel is run to process them in parallel, launching a thread for each control point. Note that for the considered bicubic patches such a control-point-parallel approach doesn't require explicit inter-thread synchronization, since the 16 control points of a patch map to a half-warp. The kernel first computes a screen-space bound of each patch. If it is outside the view frustum, the patch gets culled. If the bound exceeds 8×8 pixels, the patch is split into two sub-patches; otherwise the patch is kept unmodified. The result is stored in a buffer with two slots available per input patch. For culled patches both slots remain empty, and an unsubdivided patch leaves the second slot unoccupied. Therefore, the buffer is subsequently compacted to yield a contiguous list of patches. This serves as new input for a further round of subdivision. Once subdivision is completed, dicing is performed using one thread per micropolygon.

Unfortunately, the approach suffers from several problems. First, cracks arising from using different subdivision levels are not dealt with. Instead this challenging issue is postponed for future work. Second, the way subdivision results are stored is far from optimal. Since both unprocessed patches and patches already determined to require no further subdivision are stored mixedly in a common buffer, completed patches are unnecessarily processed repeatedly, wasting both computational resources and memory bandwidth. Third, the compaction step involves copying patch data. This could be avoided and hence memory bandwidth be saved by first determining the number of output slots required per input patch and then writing directly to a contiguous patch list. Note that such an efficient approach is actually pursued in our patch-parallel technique described in Sec. 7.6. Fourth, the performance is not that impressive despite high parallelization and good utilization of resources. This is probably due to efficiency deficits, like the mentioned wasted memory bandwidth and redundant patch processing.

Finally, note that subdivision-based tessellation can also be performed using the geometry shader stage. Patches are input using an appropriate primitive topology, and the geometry shader performs a subdivision or just passes through the patch. The output patches are captured in a stream output buffer that subsequently serves as input for the next subdivision step. Once no further subdivision is required, approximating triangles are generated for each patch using another geometry shader and rendered directly.

³While in principle possible, this would cause a significant overhead because neighboring sub-patches have to overlap in their two outer rings of quad faces. Recall that a patch representing a certain part of the surface not only stores the corresponding mesh but also its one-ring neighborhood to enable further application of the Catmull-Clark subdivision rules.

7.2.6 Discussion

Adaptive subdivision might seem an attractive approach to perform tessellation of curved surfaces. The subdivision itself is easy to parallelize and amenable to an efficient GPU implementation. The sampling density is highly adaptable, varying not only per base patch but also locally within a base patch. Moreover, the sampling rate is typically guided by the actual screen-space flatness instead of some more conservative object-space criterion. Together, this avoids excessive overtessellation.

Closer inspection, however, reveals that recursive refinement suffers from several shortcomings; in particular, it appears to be not the most suitable way to realize tessellation-based rendering of curved surfaces. As already detailed before, one major issue is the avoidance of cracks and T-vertices. For high performance, related methods should ideally operate completely patch-locally to maintain mutual independent work items for parallel execution. While a technique with such characteristics exists for preventing cracks (cf. Sec. 7.2.4), basically transforming them to T-vertices, the avoidance of T-vertices eventually requires some global context. This typically involves some non-trivial bookkeeping and necessitates either inter-patch communication or some predominantly sequential, global post-processing. In particular, a patch that needs no further subdivision may not be directly rendered but has to wait until all its neighboring patches have completed refinement, essentially requiring the explicit storage of all completed patches. Note that just ignoring T-vertices is not really acceptable because of the resulting visual artifacts, like missing pixels or shading discontinuities.

Another issue is that although a subdivision step may be realized efficiently on the GPU and high resource utilizations are achievable, the whole approach is usually not that efficient because redundant work is performed and consumption of memory bandwidth and space is high. For utmost parallelism and adaptivity, each patch is typically processed independently, and the sub-patches resulting from subdivision are treated and stored as individual patches. This, however, implies that boundary curves shared by two sub-patches are processed and stored twice. Similarly, after subdivision has been completed, each patch is represented by an isolated quad or triangle, respectively. This causes high storage costs, since each interior valence- k vertex of the corresponding tessellation is stored k times. Moreover, the vertex shader stage is slowed down because the post-transform vertex cache is of no use when logically identical vertices are redundantly represented by different entries in the vertex buffer. Finally, repeated subdivision runs in multiple passes and hence intermediate storage is required to capture the output of one pass and provide it as input for the next subdivision round. Especially the involved writes and reads to the graphics hardware's device memory entail a significant overhead.

Furthermore, the adaptivity of the sampling density is actually rather restricted because only successive bisections of boundary curves are performed. Consequently, in the worst case each parameter direction may be oversampled by a factor of almost two. For instance, if ten equidistantly spaced samples (including the endpoints) are required for a curve, actually 17 sample points are produced by repeated subdivision. While, on the other hand, adaptive refinement can locally vary the sampling density and hence is not restricted to a uniform sampling, we observe that in practice the variation within a base patch is usually rather low, at least for bicubic Bézier patches and cubic Bézier triangles. This is even the case if the screen-space extent of the patch is quite large, like in the Utah teapot example in Fig. 7.3. Therefore, a uniform sampling with an integer sampling rate will often result in a tessellation with a comparable or even smaller number of faces as when performing adaptive subdivision. Moreover, uniform sampling yields nicely shaped faces in parameter space. Also note that if only 1-to-4 splits are

used during refinement to keep the subdivision depth low, unnecessary oversampling in one parameter direction may occur.

The path-based adaptation algorithm by Velho and de Figueiredo [385] seeks to derive an optimal sampling. Avoiding cracks and T-vertices, this quasi-local approach decouples determining the sampling from the actual subdivision process. Starting with a given domain tessellation consisting only of triangular faces, an appropriate adaptive sampling of the curves corresponding to edges in the tessellation is computed, essentially yielding polyline approximations for them. Subsequently, all patches (corresponding to facets in the tessellation) which feature at least one boundary curve that is approximated by more than one line segment are subdivided, choosing each subdivision point “optimally” from the affected curve’s sample points. Taking newly introduced edges and sub-patches into account, this procedure is then repeated until no further subdivision is required. Since the approach relies on an initial domain tessellation whose edges cover all relevant areas, the method probably fails if the corresponding surface curves are flat and a highly curved part hence gets missed. While still essentially performing recursive refinement, this technique incorporates some characteristics of the alternative approach of first determining the required sampling density and then sampling the patch accordingly to produce its tessellation. We cover this competing approach in the remainder of this chapter.

7.3 Tessellation patterns

The method now widely used for tessellating curved surfaces and also advocated by the upcoming Direct3D 11 first derives the desired sampling rate and subsequently generates an according tessellation. The unit-domain locations of the sample points corresponding to the vertices of such a tessellation as well as its topology are typically described by a *tessellation pattern*. A certain pattern is selected by specifying the sampling rates along the domain boundaries and for the domain interior, usually chosen as either the minimum or the maximum of the boundary sampling rates. Often, these rates are expressed as *tessellation factors*, which refer to the number of line segments along a boundary and equal the inverse of the sampling step size. Several different schemes exist for creating a pattern that satisfies prescribed sampling densities. In this section, we review some of them and detail our method for generating tessellation patterns.

7.3.1 Patterns for rectangular domains

For a rectangular domain, the tessellation pattern is uniform if all factors in u direction are identical as well as all those in v direction. In general, however, the tessellation factors for two opposing domain boundaries differ to ensure a watertight overall tessellation. This flexibility to choose a sampling rate for each boundary curve individually allows a patch’s tessellation to abut seamlessly on those of the neighboring patches. An according tessellation pattern consists of a uniform core and a *transition region* at each boundary where the tessellation factor differs from the corresponding interior factor. Within such a factor transition region, the boundary is stitched to the interior core using a strip of triangles. The concrete method we use is detailed later in Sec. 7.3.3.

Some example patterns are illustrated in Fig. 7.6. Uniform parts are colored yellow, while different shades of brown indicate individual transition regions. Subfigs. a–c are representative of our own scheme. Since we choose the interior tessellation factor for a certain parameter

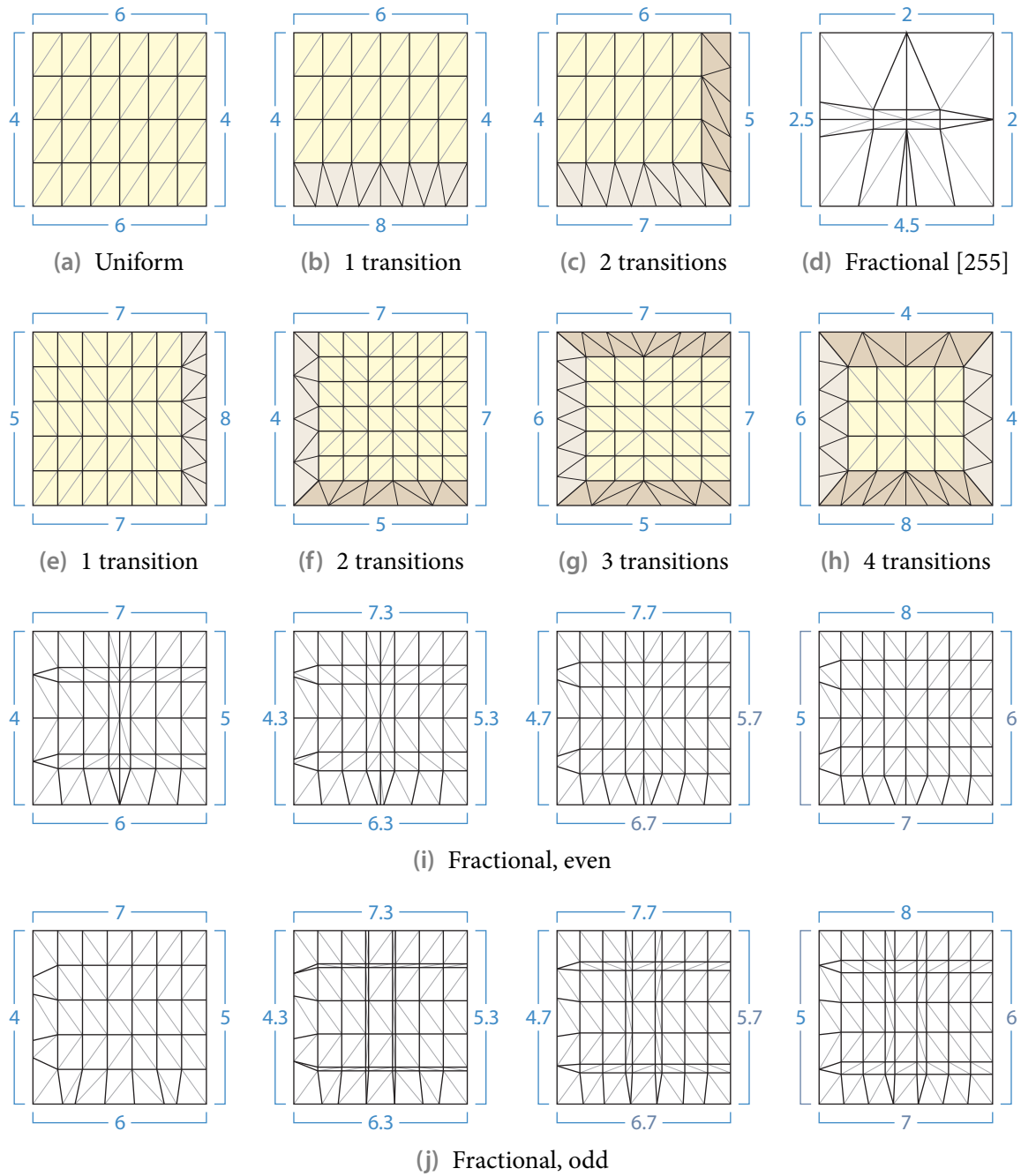


Figure 7.6 Various tessellation patterns for a rectangular domain.

direction as the minimum of the two corresponding boundary factors, the uniform core always extends to at least two boundaries while at most two transition regions occur. This scheme is essentially equivalent to Moreton's [255], who utilizes the maximum operator to derive the internal factors.

By contrast, the tessellation stage of the upcoming Direct3D 11 [252] allows explicitly specifying the two interior tessellation factors. In particular, an internal factor may differ from both of the two related boundary factors, and therefore, up to four transition regions can occur. Subfigs. e–h show patterns for four different classes of tessellation factor configurations.⁴ To

avoid visual clutter, we omitted labeling the internal factors (e: 7, 5; f: 7, 7; g: 6, 7; h: 6, 5). Note that the quad faces in the uniform core regions are triangulated in a quasi-symmetric way such that the diagonal edges point towards the patch center. We, however, choose the diagonals completely uniformly because thus the core can efficiently be described by triangle strips. But when tessellation patterns are directly generated on the fly in hardware, as will be the case with Direct3D 11's scheme, such considerations are of less concern.

Fractional tessellation

The required sampling density for approximating a patch usually increases or decreases when the patch is moved relative to the camera. To allow a smooth adjustment of the tessellation pattern satisfying the sampling requirements and hence a fine-grained continuous level-of-detail, Moreton introduced *fractional tessellation* [255]. In this scheme, tessellation factors are no longer restricted to integer values but can be real numbers. Given a boundary factor m , $\lfloor m \rfloor$ line segments of parameter-space length $1/m$ are generated along the boundary. Additionally, in case m is not an integer, a segment of shortened length $\text{fract}(m)/m$ is created. To avoid any directional bias, in practice a symmetric pattern is constructed by bisecting the boundary and applying the scheme to both halves with a tessellation factor of $1/2 m$, placing the shortened segments towards the center of the boundary. Subfig. d shows an example.

An extended version of this fractional tessellation scheme is available in Direct3D 11. It comes in two flavors, even and odd, denoting the parity of the number of created segments; see Subfigs. i and j. Note that due to the symmetric splitting of the tessellation factor, this parity is independent from the factor's parity and can be specified freely. The even variant corresponds to the original fractional scheme but distributes the shorter-length segments differently. Moreover, the segment lengths are now determined by blending between the two closest integer configurations. For instance, a tessellation factor of 3.5 no longer results in a decomposition in parameter space as $10/35 + 10/35 + 10/35 + 5/35$ but as $7/24 + 7/24 + 7/24 + 3/24$, term-wisely interpolating between $1/3 + 1/3 + 1/3 + 0/3$ (factor 3) and $1/4 + 1/4 + 1/4 + 1/4$ (factor 4).

Fractional tessellation was devised to smoothly vary the tessellation rate without causing popping artifacts when the rendered tessellation geometry changes. However, such artifacts only occur if the maximum approximation error is chosen too large and thus curved surfaces don't appear smoothly curved. But then fractional tessellation may trade popping artifacts for swimming artifacts. A probably better solution for such suprathreshold settings is using only power-of-two tessellation factors and performing geomorphing. This guarantees that when factors are increased or decreased, sample points are only added or removed, respectively, but never moved.

Note that employing fractional tessellation doesn't make sense in our application. First, we always aim at choosing the sampling rate high enough. Second, a fractional tessellation pattern comprises at least as many triangles as the corresponding ordinary tessellation pattern for the next larger integer tessellation factors. Even worse, it is more expensive to create.

For the sake of completeness, we note that non-uniform fractional tessellation [260] subjects the sample points of a fractional tessellation pattern to a view-dependent reverse projection mapping in order to obtain a more uniform distribution of sample points in screen space during rendering. However, the resulting non-uniform sample spacing in parameter

⁴All examples were obtained with the Direct3D 11 reference device of the November 2008 version of the DirectX SDK [252].

space makes it hard to ensure that the sampling guarantees a prescribed approximation error bound. Therefore, even ignoring the entailed overhead, this extension is not suitable for our purposes.

7.3.2 Patterns for triangular domains

Similar to the situation with subdivision options, the triangular domain offers a larger variety than the rectangular case when it comes to tessellation pattern schemes. Typically, the tessellation factors along the boundaries can be chosen freely, while the single internal factor for the uniform patch interior is implicitly set to the maximum of all three boundary factors. Fig. 7.7 depicts some example patterns for various schemes. The internal tessellation factor is labeled in a small blue triangle if it is specified explicitly.

Subfig. a–c illustrate our own scheme. Like in the rectangular case, the pattern consists of a uniform core and up to two transition regions at boundaries where the tessellation factors differ from the internal factor, defined by the maximum of the three boundary factors. The uniform part is constructed by a regular triangular tiling and comprises solely triangles equilateral in parameter space.

By contrast, the scheme pursued by Direct3D 11 (cf. Subfigs. d–g) fills the interior by concentric triangles, successively decreasing the uniform tessellation factor used for each triangle boundary by two towards the center. The boundaries of two concentric triangles are connected by quad faces, triangulated such that the diagonal points to the patch center. Note that the triangular domain is implicitly split into three sectors, as illustrated in Subfig. d. Although the resulting tessellation patterns are highly symmetric, they entail more internal vertices and triangular faces than our scheme. Consequently, more sample points must be processed when generating the approximating tessellation, impacting performance. As an example, a uniform tessellation with factor m results in $\frac{1}{2}m^2 + \frac{3}{2}m + 1$ vertices with our scheme, whereas Direct3D 11's approach produces $\lfloor \frac{1}{4}m^2 \rfloor$ more samples, namely $\frac{3}{4}m^2 + \frac{3}{2}m + 1$ vertices for even values of m and $\frac{3}{4}m^2 + \frac{3}{2}m + \frac{3}{4}$ if m is odd. Moreover, while reasonably shaped, the created triangles are not equilateral as in our scheme. Finally, note that like in the rectangular domain setting, Direct3D 11 also offers even and odd fractional tessellation schemes, shown in Subfigs. h and i, respectively.

Subfig. j exemplifies the method of Chung and Kim [71], which is quite similar to ours, producing the same pattern for uniform factor configurations. If the boundary factors differ, however, they create an inner core triangle uniformly tessellated with a factor equaling the maximum of the boundary factors minus two. This core region is then connected to all three outer boundaries with transition regions. To obtain nicely shaped triangles at the corners of the tessellation, i.e. in the blue and green regions in Fig. 7.7 j, edges connecting an outer corner of the tessellation with a corner of the inner core triangle are flipped if they form the longest edge in both of their adjacent tessellation triangles (green case). Note that this scheme yields more triangles than ours in case of non-uniform tessellation factors.

To reuse the pattern generator for rectangular domains, Moreton [255] treats the domain triangle as a degenerate quad where one edge is collapsed; see Subfigs. k and l. While simple, this scheme imposes a predominant parameter direction (\mathbf{d}_{01}), causes severe oversampling along this direction, and yields long and thin triangles near the singular boundary. Similarly, in the corresponding fractional tessellation variant, shown in Subfig. m, the fractional tessellation scheme for rectangles is reused. To this end, the triangular domain is decomposed into three quadrilateral sectors.

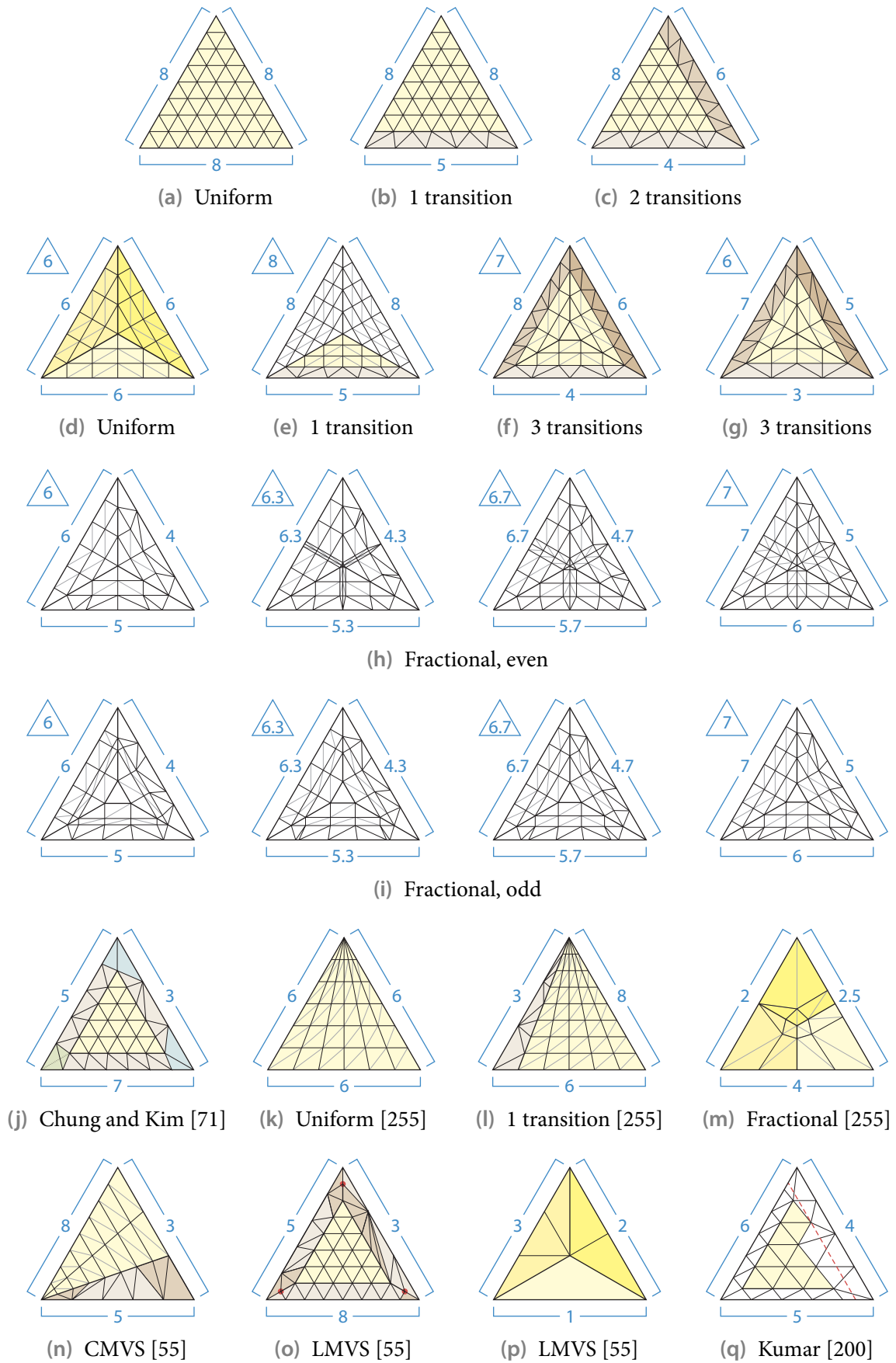


Figure 7.7 Various tessellation patterns for a triangular domain.

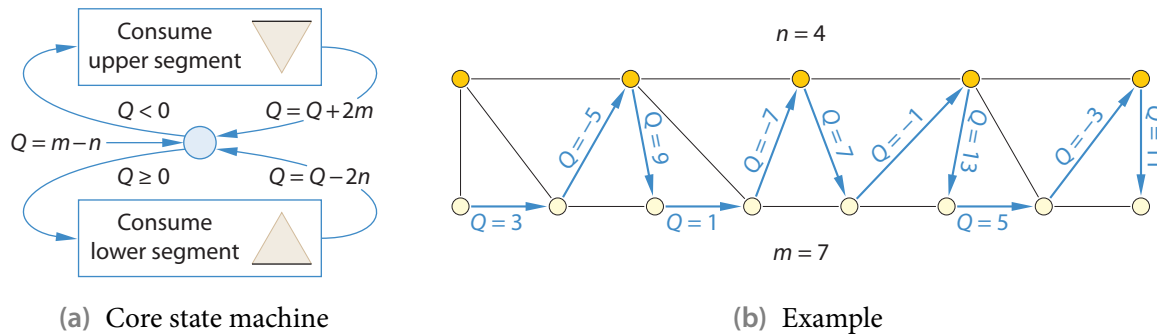


Figure 7.8 Transition strips can be generated with a Bresenham-like algorithm [255].

Bruijns [55] presents three schemes, crafted to dynamically choose the one which generates the least number of triangles. The first one, called fast fixed subdivision (FFS), supports only uniformly-chosen tessellation factors and yields identical results as our approach (cf. Subfig. a). The second scheme, termed constant maximum variable subdivision (CMVS) and depicted in Subfig. n, essentially produces the same output as Moreton's approach of reparameterizing the domain triangle as a quad. However, Bruijns selects that parameter direction as predominant whose corresponding boundary tessellation factor is the smallest one, thus keeping oversampling to a minimum. The last scheme, linear maximum variable subdivision (LMVS), is similar to ours, generating a uniform inner core triangle with FFS and connecting it to the outer boundaries with transition regions; see Subfig. o. To improve the triangle shapes near the corners of the tessellation, an additional sample (colored red) is placed halfway between each corner of the outer triangle and the corresponding corner of the inner core triangle. If the minimum of all boundary tessellation factors is below four, the inner core triangle vanishes. In such cases, the domain is subjected to an internal 1-to-3 split, and for each resulting sector a triangle fan is constructed, as shown in Subfig. p. For filling the transition regions with triangles, Bruijns uses triangle strips (colored in light brown) and fans (darker brown). In one method, the boundary with the smaller number of samples is connected to the central part of the other boundary by a single triangle strip, and triangles fans are used to stitch the remaining parts of the higher-sampled boundary to the endpoints of the lower-sampled one. Alternatively, a triangle fan is constructed for the central part, while a strip is used at each outer part of the boundary with the larger number of samples.

Finally, Subfig. q illustrates Kumar's scheme [200]. First, the two boundaries with the largest tessellation factors are determined.⁵ In the following, we assume these to be $w = 0$ and $v = 0$ with associated factors m and n (corresponding to the lower and the left boundary in Subfig. q). Conceptually, an inner core triangle uniformly tessellated with factor $\max\{m, n\}$ is generated and then stretched such that each edge parallel to $w = 0$ is of length $1/m$ in parameter space and each edge parallel to $v = 0$ is of length $1/n$. Note that consequently, the remaining edges are generally no longer parallel to $u = 0$. This core is placed such that the corner where the two boundaries parallel to $w = 0$ and $v = 0$ meet has position $(v, w) = (1/2m, 1/2n)$. Subsequently, all core faces are removed which are (at least partly) beyond the line $u = 1/2\min(m, n)$ (indicated in red). Finally, the outer edges of the remaining core (colored yellow) are stitched to the outer boundaries.

⁵Kumar actually allows not only for tessellation factors along the boundaries but also for three (possibly non-integer) internal factors, one for each parameter direction. For simplicity, we assume that the boundary factors coincide with the internal ones.

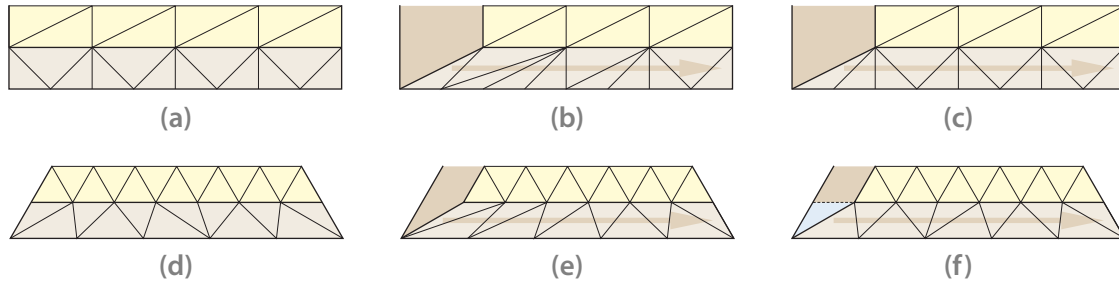


Figure 7.9 The direct application of the stitching algorithm [255] works fine for isolated transition regions (a, d) but yields sub-optimal triangle shapes if two transition regions meet (b, e). In these cases, better results are obtained by temporarily extending the interior core boundary to mirror the isolated setting, running the algorithm with this modified input, and subsequently collapsing the temporarily introduced line segments (c, f).

7.3.3 Transition regions

A transition region serves to link the polyline approximations of two curves possibly sampled at different rates with a filling strip of triangles. Each segment of each polyline is connected to a vertex of the other polyline, creating a triangle. These triangles must not overlap and ideally should not be skinny but well shaped.

An efficient method to generate such triangles for a transition region was proposed by Moreton [255]. It seeks to yield a fair allocation of line segments to vertices and to obtain well-shaped triangles in parameter space by traversing the polyline segments using a state machine similar to Bresenham's line drawing algorithm [54, 158]. As illustrated in Fig. 7.8, a state variable Q is initialized to the difference of the two curve tessellation factors m and n , and subsequently updated. Its sign controls from which of the two polyline approximations the next segment is picked for creating a triangle.

However, we observe that if a transition region is not isolated but meets another one at a corner, then the resulting fill triangles may become badly shaped when the algorithm is directly applied, as depicted in Figs. 7.9 be and 7.10 a.⁶ This is because in such cases the inner polyline does not span the whole tessellation and is also not centered with respect to the outer boundary unless a neighboring transition region abuts at each side. Our solution is to temporarily augment the inner polyline by an additional line segment at each end which is not on the boundary of the tessellation because another transition region adjoins. We then launch the stitching algorithm with the adjusted inner polyline as well as the original outer polyline, and subsequently collapse the segments temporarily introduced to the inner polyline, discarding resulting zero-area triangles. The effectiveness of this approach is demonstrated by the examples in Figs. 7.9 and 7.10. An efficient implementation is discussed in the next subsection.

7.3.4 Fast pattern generation

A main objective in devising our tessellation pattern schemes for rectangular and triangular domains was the ability to rapidly generate patterns for a given tessellation factor configuration. This is crucial for our tessellation approach presented in Sec. 7.6 as it involves creating such

⁶Note that a similar observation was made by Chung and Kim [71]. Moreover, there are some indications that Moreton [255] was aware of the issue, although it is not explicitly addressed.

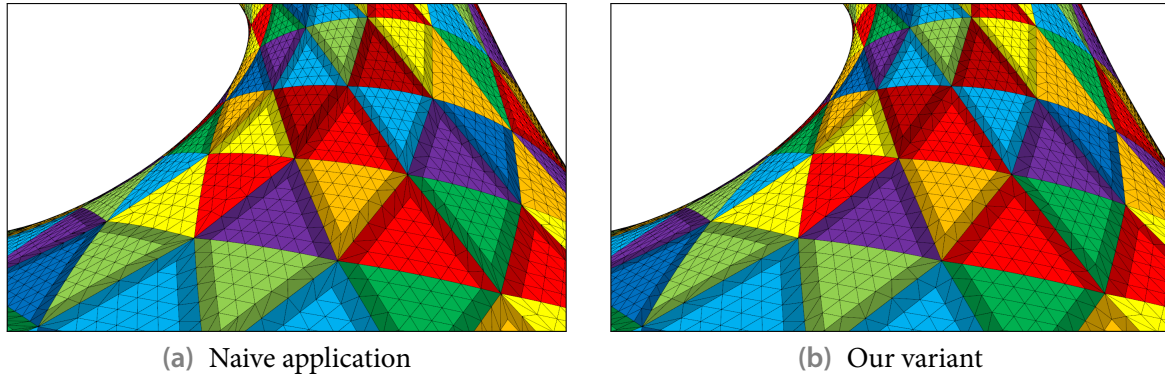


Figure 7.10 Close-up view of the double torus model demonstrating that in case a patch's tessellation pattern features two adjoining transition regions (indicated by the darker shades), the naive application of the transition region filling algorithm often yields badly shaped triangles (a) in contrast to our special treatment (b).

a pattern for each patch on the fly and per frame. In the following, we provide some details concerning our implementation, where a pattern is stored as an ordered list of sample points and a set of triangle strips describing their topology.

Recall that in our scheme for a rectangular domain, each internal tessellation factor is chosen as the minimum of the boundary factors for the same parameter direction. By contrast, in the triangular setting, the internal factor is implicitly set to the maximum of the boundary factors. Therefore, in both cases at most two transition regions exist. Examples of this most general setup are shown in Fig. 7.11; note that the vertices are labeled by their indices.

First, vertices for the uniform core are created, proceeding row-wise. If transition regions exist, subsequently samples are generated along the involved outer boundaries. These vertices are ordered such that their indices can be derived easily and efficiently. The uniform core is then covered by triangle strips, producing one per row. Finally, the transition regions are filled by applying the stitching algorithm described in the previous subsection. Here, each triangle is output as a separate triangle strip because this simplifies exactly predicting the overall storage requirements and also avoids the control overhead of creating optimal-sized strips. Note that the alternative of using just individual triangles throughout instead of strips usually results in a less compact representation, since the efficiency gain for the transition regions is typically outweighed by the storage increase for the uniform core.

Regarding the realization of the linking algorithm for the transition regions, first note that the index increment required to proceed from one vertex of a polyline to an adjacent one is constant for the outer boundaries. In case of a rectangular domain, it is also constant for the internal boundaries, whereas in the triangular setting the increment may vary linearly. Exploiting this allows efficiently determining the indices for the output triangle strips. If two abutting transition regions exist, we initiate the stitching algorithm with the number of segments for the inner polyline increased by one. To efficiently handle the resulting temporary line segment and its collapsing, we pursue a domain-specific approach.

In the triangular case, the direction along which the segments are processed is additionally chosen to point away from where the transition regions meet. This causes the first output triangle to always be made up from the first (only virtual) inner segment because, by construction, the modified inner segment count m is at least as large as the outer segment count n , and hence the state variable Q is initially non-negative ($m - n \geq 0$). Therefore, we only have to skip the

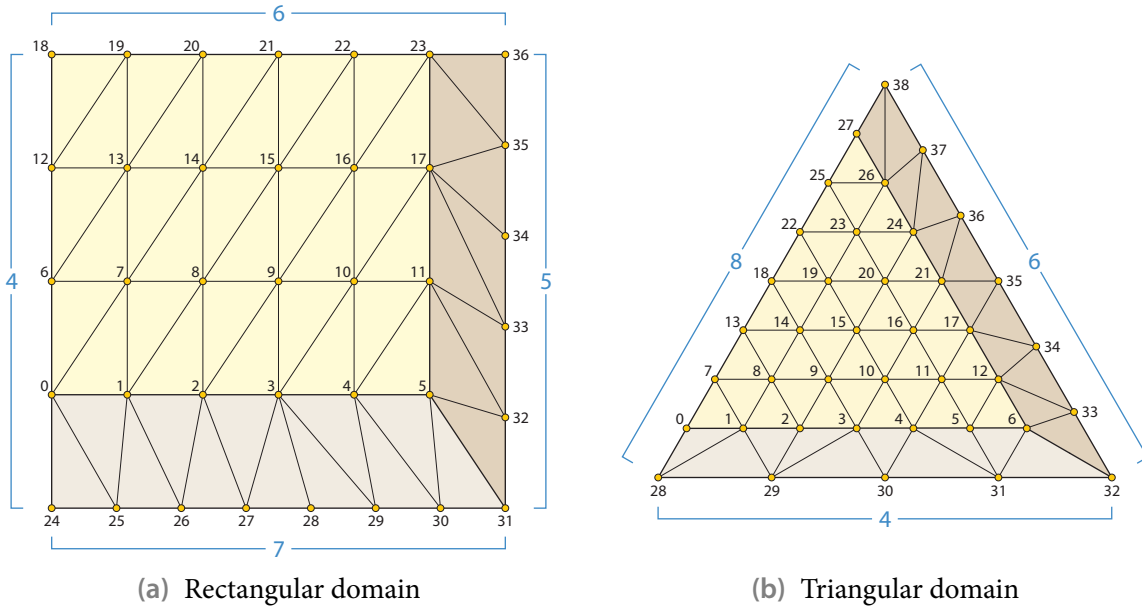


Figure 7.11 Example tessellation patterns from our scheme showing the vertex ordering used in our realization.

first produced triangle (colored blue in Fig. 7.9 f), writing no indices but updating Q , and can then proceed normally with the stitching algorithm.

The rectangular setting is more involved because here the outer segment count n dominates the inner segment count m . If the processing starts from where the neighboring transition region abuts, we first generate triangles made up of segments from the outer polyline as usual until Q becomes non-negative. Then, the next triangle would be constructed from the virtually introduced inner segment and is hence just skipped. Subsequently, the ordinary stitching algorithm is run. Similarly, in case the processing direction points towards where the transition regions meet, we execute the usual linking algorithm and abort it once the last non-temporary vertex of the inner polyline has been used for the first time. A triangle is then constructed for each remaining outer segment by connecting it to this last vertex.

7.4 Determining tessellation factors

When creating a tessellation for a curved surface patch, an appropriate sampling density has to be determined. While this may be achieved implicitly by recursively refining an initial coarse sampling, we now focus on the alternative of directly deriving uniform sampling rates and corresponding tessellation factors from the surface description. These can then be used to select an according tessellation pattern. Note that in most cases, for each parameter direction, only one tessellation factor valid for the whole patch is determined, setting related boundary and internal factors to identical values. The boundary factors may later be increased to match the corresponding factor in the tessellation of an adjacent patch. Another option is to directly determine individual factors for each boundary and the patch interior, avoiding later adjustments. This, however, can result in undersampling close to a boundary when its factor is smaller than the related internal factor because then the maximum sample spacing in the corresponding tessellation pattern transition region grows towards the boundary.

To account for the current viewing setup, tessellation factors are typically calculated such that a prescribed screen-space bound, e.g. an approximation error tolerance ϵ , is satisfied. The computations usually involve some quantities derived from the surface description, like bounds on the magnitudes of the derivatives. These quantities may be directly determined in screen space after transforming the surface description from object space into screen space. Such an approach is often pursued in repeated subdivision techniques to compute the degree of flatness that guides further refinement.

Alternatively, the quantities can be derived in object space and a subsequent transform brings them into a common space with the screen-space bound. Since the quantities are hence view-independent, they can be precomputed for each patch, which is attractive if their calculation is comparatively expensive, as is usually the case. Note that precomputation is not possible or reasonable if the surface description changes every frame or, like in adaptive subdivision methods, patches are dynamically created. On the other hand, the computation in object space typically results in more conservative estimates compared to approaches operating directly in screen space, causing some oversampling. This is because the derived quantities provide only condensed information about a surface and hence the effect of the current view on the surface's screen-space projection cannot be fully taken into account.

While the object-space quantities could be projected into screen space, usually it is easier to transform a scalar screen-space bound into object space. Alternatively, both may be mapped to an intermediate space. For instance, Abi-Ezzi and Shirman [2] advocate using the lighting coordinate space [4] resulting from factoring the view-projection matrix into a rigid factor and a sparse non-rigid factor. By carefully investigating the involved transformation matrices, they get tight scaling factors for transforming object-space derivative bounds and screen-space thresholds to lighting coordinates. On the downside, the approach necessitates matrix factorization and matrix norm computations.

7.4.1 Bounding screen-space triangle sizes

Many approaches determine tessellation factors by computing the sampling rate that is required to make the size of the screen-space projection of each triangle within the resulting tessellation satisfy some bound. This heuristic criterion ensures a certain screen-space sampling density that helps to capture shading variations, especially when using Gouraud shading and glossy materials. On the other hand, without imposing tiny size bounds it cannot be guaranteed that the deviation of the tessellation from the approximated surface is small enough to be invisible. Moreover, severe oversampling may occur because essentially no difference is made whether a surface is highly curved or completely flat.

One very simple method to derive a boundary tessellation factor, for instance pursued by Chung and Kim [71], is to just project the corner points of the related boundary into screen space, determine their distance in pixels, and divide this distance by some user-specified edge length bound ϵ_1 . The corresponding tessellation, however, will yield edges longer than ϵ_1 pixels in case the boundary is curved. Note that if each boundary tessellation factor depends solely on its associated boundary and ignores the patch interior, like in this method, then internal tessellation factors guaranteeing a sufficient sampling cannot be reasonably inferred from the boundary factors but must be specified separately.

Better results are achieved by taking the actual shape of the boundary curve into account. For example, Rockwood [314] suggests deriving the tessellation factor as a function of the maximum distance between two successive Bézier control points. Note that this distance is essen-

tially a bound on the magnitude of the curve's first derivative, obtained by exploiting the convex hull property. Correspondingly, for the general setting of a curve $\mathbf{c}(t)$, $t \in [0, 1]$, and a threshold ε_s , Abi-Ezzi and Shirman [2] determine the tessellation factor $m = \lceil 1/\delta \rceil$ using the mean value theorem to compute the sampling step size δ such that

$$\delta \leq \frac{\varepsilon_s}{\max_{t \in [0,1]} \|\mathbf{c}'(t)\|}$$

holds. However, Kumar [199] argues that the mean value theorem is incorrectly applied because $\mathbf{c}(t)$ is a vector-valued function and not a scalar one. He suggests employing the modified criterion

$$\delta \leq \frac{\varepsilon_s}{\left\| \left(\max_{t \in [0,1]} (\mathbf{c}'(t))_x, \max_{t \in [0,1]} (\mathbf{c}'(t))_y, \max_{t \in [0,1]} (\mathbf{c}'(t))_z \right)^T \right\|} \quad (7.1)$$

instead.

For a Bézier patch $\mathbf{b}(u, v)$, the required tessellation factor $m_u = \lceil 1/\delta_u \rceil$ in u parameter direction is then determined by considering all curves $\mathbf{b}_{[v]}(u) = \mathbf{b}(u, v)$ in u direction and hence ultimately by the (possibly component-wise) maximum magnitude D_u of the partial derivative $\mathbf{b}_u(u, v)$. Utilizing the convex hull property, D_u can be easily bounded by the maximum magnitude of the corresponding control points. Alternatively, one may compute the accurate maximum D_u to get a tighter bound for the step size δ_u [199]. Note that since such a calculation is expensive, it is a good candidate for precomputation in object space.

7.4.2 Bounding the approximation error

However, we are in general not interested in ensuring some minimum screen-space sampling rate but strive to generate a tessellation that well approximates a curved surface within some controllable tolerance. Accordingly, methods that bound the approximation error are more appropriate for determining tessellation factors. They typically yield a fine sampling in presence of high curvature and a coarse sampling for rather flat patches.

A well-known result from approximation theory states that the deviation of a C^2 -continuous curve $\mathbf{c}(t)$, $t \in [a, b]$, from its endpoint-interpolating linear approximation, the linearly parameterized line segment $\mathbf{l}(t)$ with $\mathbf{l}(a) = \mathbf{c}(a)$ and $\mathbf{l}(b) = \mathbf{c}(b)$, is bounded according to [128]

$$\sup_{t \in [a,b]} \|\mathbf{c}(t) - \mathbf{l}(t)\| \leq \frac{1}{8} (b-a)^2 \sup_{t \in [a,b]} \|\mathbf{c}''(t)\|. \quad (7.2)$$

Consequently, to satisfy an error tolerance ε , the sample spacing δ for a Bézier curve $\mathbf{b}(t)$ defined on the unit interval $U = [0, 1] \ni t$ must be chosen such that

$$\delta \leq \sqrt{\frac{8\varepsilon}{\sup_{t \in U} \|\mathbf{b}''(t)\|}}$$

holds. The corresponding tessellation factor is then trivially given by $m = \lceil 1/\delta \rceil$. Since the second derivative $\mathbf{b}''(t)$ is itself a Bézier curve, its norm can easily be bounded by the maximum magnitude of its curve control points. Note the quadratic convergence, that is, doubling the tessellation factor reduces the approximation error by a factor of four. This implies that when zooming in to a curve, the number of samples required to maintain meeting the prescribed error bound grows so slowly that the resulting screen-space sample spacing actually increases. It is hence obvious that generally either over- or undersampling will occur if criteria are used

which choose the tessellation factor such that the screen-space sampling rate is bounded instead of the geometric approximation error.

A similar result exists for surfaces. Consider a C^2 -continuous surface $\mathbf{s}(u, v)$ defined on the right domain triangle $T = \triangle(\mathbf{A}, \mathbf{B}, \mathbf{C}) \ni (u, v)$ with $\mathbf{A} = (u_0, v_0)$, $\mathbf{B} = (u_0 + \delta_u, v_0)$ and $\mathbf{C} = (u_0, v_0 + \delta_v)$, and its corner-interpolating linear approximation, the linearly parameterized triangle $\mathbf{l}(u, v)$ with $\mathbf{l}(\mathbf{A}) = \mathbf{s}(\mathbf{A})$, $\mathbf{l}(\mathbf{B}) = \mathbf{s}(\mathbf{B})$ and $\mathbf{l}(\mathbf{C}) = \mathbf{s}(\mathbf{C})$. Then, according to Filip et al. [128]

$$\sup_{(u,v) \in T} \|\mathbf{s}(u, v) - \mathbf{l}(u, v)\| \leq \frac{1}{8} (M_{uu} \delta_u^2 + 2M_{uv} \delta_u \delta_v + M_{vv} \delta_v^2) \quad (7.3)$$

holds,⁷ where

$$M_{uu} = \sup_{(u,v) \in T} \|\mathbf{s}_{uu}(u, v)\|, \quad M_{uv} = \sup_{(u,v) \in T} \|\mathbf{s}_{uv}(u, v)\|, \quad M_{vv} = \sup_{(u,v) \in T} \|\mathbf{s}_{vv}(u, v)\|.$$

Applied to a tensor-product Bézier patch $\mathbf{b}(u, v)$, $(u, v) \in U = [0, 1]^2$, this means that a uniform tessellation with tessellation factors m_u and m_v in u and v direction, respectively, sampling the patch with step sizes $\delta_u = 1/m_u$ and $\delta_v = 1/m_v$ at parameter points $(i\delta_u, j\delta_v)$, $0 \leq i \leq m_u$, $0 \leq j \leq m_v$, keeps its maximum deviation from the patch within the tolerance ε if

$$D_{uu} \delta_u^2 + 2D_{uv} \delta_u \delta_v + D_{vv} \delta_v^2 \leq 8\varepsilon \quad (7.4)$$

is satisfied, where

$$D_{uu} = \sup_{(u,v) \in U} \|\mathbf{b}_{uu}(u, v)\|, \quad D_{uv} = \sup_{(u,v) \in U} \|\mathbf{b}_{uv}(u, v)\|, \quad D_{vv} = \sup_{(u,v) \in U} \|\mathbf{b}_{vv}(u, v)\|.$$

Each of these bounds on the second-order partial and mixed derivatives can be bounded by the maximum magnitude of the Bézier control points of the corresponding derivative patch.

To infer concrete maximum step sizes δ_u and δ_v from (7.4), and thus ultimately derive tessellation factors m_u and m_v , Filip et al. [128] suggest the following procedure. If $D_{uu} = 0$ and $D_{vv} > 0$, i.e. the surface is linear in u direction, we set $\delta_u = 1$ and hence $m_u = 1$, and then get

$$\delta_v = \frac{\sqrt{D_{uv}^2 + 8\varepsilon D_{vv}} - D_{uv}}{D_{vv}} \quad \text{and thus} \quad m_v = \left\lceil \frac{1}{\delta_v} \right\rceil = \left\lceil \frac{\sqrt{D_{uv}^2 + 8\varepsilon D_{vv}} + D_{uv}}{8\varepsilon} \right\rceil.$$

The case $D_{vv} = 0$ and $D_{uu} > 0$ is treated analogously. If $D_{uu} = D_{vv} = 0$, we choose $\delta_u = \delta_v$ and hence obtain

$$m_u = m_v = \left\lceil \frac{1}{\delta_v} \right\rceil \quad \text{with} \quad \delta_u = \delta_v = \sqrt{\frac{4\varepsilon}{D_{uv}}}$$

unless $D_{uv} = 0$, in which case we trivially set $m_u = m_v = 1$. Finally, if $D_{uu} > 0$ and $D_{vv} > 0$, Filip et al. fix $\delta_v = D_{uu}/D_{vv} \delta_u$, yielding

$$\delta_u = \sqrt{\frac{8\varepsilon D_{vv}}{D_{uu} D_{vv} + 2D_{uu} D_{uv} + D_{uu}^2}} \quad \text{and} \quad m_u = \left\lceil \frac{1}{\delta_u} \right\rceil, \quad m_v = \left\lceil \frac{D_{vv}}{D_{uu} \delta_u} \right\rceil.$$

⁷Note that the bounded approximation error is defined as the distance between a surface point $\mathbf{s}(u, v)$ and the point $\mathbf{l}(u, v)$ on the linear approximation at the same parametric location. If instead the perpendicular distance of the point $\mathbf{s}(u, v)$ to the triangle \mathbf{l} and hence the closest point $\mathbf{l}(u', v')$ is employed to measure deviation, tighter bounds and thus ultimately larger step sizes are possible [378].

Abi-Ezzi and Shirman [2] improve on that by instead choosing $\delta_v = \sqrt{D_{uu}/D_{vv}} \delta_u$, which minimizes the term $2/\delta_u\delta_v$, essentially corresponding to the number of triangles in the tessellation; then

$$\delta_u = \sqrt{\frac{4\varepsilon D_{vv}}{D_{uu}D_{vv} + \sqrt{D_{uu}D_{vv}} D_{uv}}} \quad \text{and} \quad m_u = \left\lceil \frac{1}{\delta_u} \right\rceil, \quad m_v = \left\lceil \sqrt{\frac{D_{vv}}{D_{uu}}} \frac{1}{\delta_u} \right\rceil.$$

A different approach is pursued by Guthe et al. [150], who split the bivariate term from (7.4) into two univariate ones by utilizing the inequality $\delta_u\delta_v \leq 1/2(\delta_u^2 + \delta_v^2)$, and then distribute the error tolerance ε evenly among both parameter directions, yielding

$$(D_{uu} + D_{uv})\delta_u^2 \leq 4\varepsilon \quad \text{and} \quad (D_{uv} + D_{vv})\delta_v^2 \leq 4\varepsilon.$$

While simple and amenable to fast step size determination when the bounds D_{uu} , D_{uv} and D_{vv} are precomputed, this reformulation into two independent curve-like cases generally yields looser bounds on the maximum step sizes and results in a larger number of triangles.

In addition to controlling the maximum geometric deviation of a tessellation from the approximated surface, some applications may desire to further keep the approximation error of the surface normal or some other smooth surface signal bounded.⁸ Guthe et al. [151] present a related method that applies the same approximation-theoretic results as discussed above and augments them by a mapping from signal space to object space to allow combination with the geometric approximation error.

Although beyond our scope, we finally note that bounds for the geometric deviation from a piecewise-linear approximation also exist for subdivision surfaces [403].

7.4.3 Rational Bézier patches

Rational surface patches are of high practical relevance due to their ability to describe quadrics and surfaces of revolution, like ellipsoids and cylinders. For them, too, the condition in (7.4) is sufficient for bounding the approximation error of a uniform tessellation. However, recall from Sec. 6.1.1 that a second-order partial derivative of a rational Bézier patch of degree $n \times n$ has an overall degree of $3n \times 3n$. Consequently, deriving the bounds D_{uu} , D_{uv} and D_{vv} is expensive and hence hampers a rapid on-the-fly computation for dynamically changing patches. While several approaches exist to more cheaply derive bounds on the first-order partial derivatives [169, 414, 417], second-order derivatives are largely unaddressed. But even if the maximum magnitudes of the second-order derivatives are accurately determined, Kumar [199] notes that due to the high degrees involved, the approximation error bound is rather loose and hence unnecessarily large sampling rates get chosen. To alleviate oversampling, he therefore suggests a heuristic which increases the tessellation factor obtained with a screen-space size criterion like (7.1) by the maximum difference in partial derivative values at the resulting samples near the patch corners, scaled by a user-defined constant.

We opt for the more robust method proposed by Zheng and Sederberg [416]. They generalize the results from Sec. 7.4.2 to the rational setting, working directly on the homogeneous

⁸Visual artifacts due to normal deviation are typically not an issue. Note that in the work of Guthe et al. [150] artifacts occur because the bicubic patches used for rendering are only approximations of higher-degree patches which got derived ignoring derivative approximation errors, and because only uniform tessellation patterns are employed and hence abutting patches may be sampled at different rates along their common boundary curve.

representation, which is polynomial. Applied to a rational tensor-product Bézier patch of degree $n \times n$ with 4D homogeneous control points $\mathbf{b}_{ij} = (w_{ij}\mathbf{p}_{ij}, w_{ij})^T$, their theorem states that the approximation error of a uniform tessellation with factors $m_u = 1/\delta_u$ and $m_v = 1/\delta_v$ stays within a tolerance ε if

$$D_{uu}\delta_u^2 + 2D_{uv}\delta_u\delta_v + D_{vv}\delta_v^2 \leq 8\varepsilon \min_{0 \leq i, j \leq n} \{w_{ij}\} \quad (7.5)$$

holds, where

$$\begin{aligned} D_{uu} &= n(n-1) \max_{\substack{0 \leq i \leq n-2 \\ 0 \leq j \leq n}} \left\{ \left\| (\mathbf{b}_{i+2,j} - 2\mathbf{b}_{i+1,j} + \mathbf{b}_{i,j})_{xyz} \right\| + (r - \varepsilon) |w_{i+2,j} - 2w_{i+1,j} + w_{i,j}| \right\}, \\ D_{uv} &= n^2 \max_{0 \leq i, j \leq n-1} \left\{ \left\| (\mathbf{b}_{i+1,j+1} - \mathbf{b}_{i+1,j} - \mathbf{b}_{i,j+1} + \mathbf{b}_{i,j})_{xyz} \right\| \right. \\ &\quad \left. + (r - \varepsilon) |w_{i+1,j+1} - w_{i+1,j} - w_{i,j+1} + w_{i,j}| \right\}, \\ D_{vv} &= n(n-1) \max_{\substack{0 \leq i \leq n \\ 0 \leq j \leq n-2}} \left\{ \left\| (\mathbf{b}_{i,j+2} - 2\mathbf{b}_{i,j+1} + \mathbf{b}_{i,j})_{xyz} \right\| + (r - \varepsilon) |w_{i,j+2} - 2w_{i,j+1} + w_{i,j}| \right\} \end{aligned}$$

with $r = \max_{0 \leq i, j \leq n} \|\mathbf{p}_{ij}\|$. Note that if all weights w_{ij} are identical, this reduces to the general formulation from (7.4) applied to patches of degree $n \times n$.

When an error threshold ε to be satisfied is prescribed, sampling step sizes δ_u and δ_v can be derived from (7.5) like in Sec. 7.4.2, distinguishing four cases based on the positivity of D_{uu} and D_{vv} . The expressions for the bounds D_{uu} , D_{uv} and D_{vv} are now simple enough to allow fast on-the-fly determination. On the other hand, they depend not only on the surface but also on the error tolerance ε , precluding precomputation.

7.4.4 Bézier triangles

While we focused on tensor-product surfaces so far, the results from Sec. 7.4.2 can also be utilized to derive appropriate tessellation factors for Bézier triangles. However, in the triangular setting with its three barycentric domain coordinates u , v and $w = 1 - u - v$, we are primarily not interested in the sampling step sizes along u and v direction but in those along any two parameter directions parallel to the domain boundaries, for instance \mathbf{d}_{01} and $-\mathbf{d}_{20}$ (cf. Sec. 6.1.2). Consequently, D_{uu} , D_{uv} and D_{vv} have to bound the related second-order directional derivatives. This eventually yields two boundary tessellation factors; the internal and the remaining third boundary factor may conservatively be set to their maximum. A minimum overall triangle count could be obtained by considering not just one but all three pairs of boundary-parallel parameter directions emanating from a common domain corner (\mathbf{d}_{01} , $-\mathbf{d}_{20}$; \mathbf{d}_{12} , $-\mathbf{d}_{01}$; \mathbf{d}_{20} , $-\mathbf{d}_{12}$), and choosing the optimal one resulting in the smallest tessellation factors.

We pursue a slightly different approach, individually deriving all three boundary tessellation factors. Similar to Guthe et al. [150], we apply the inequality $\delta_u\delta_v \leq 1/2(\delta_u^2 + \delta_v^2)$ to (7.4) and evenly distribute the error tolerance ε to get univariate inequalities; for all three pairs of boundary-parallel parameter directions, we thus obtain

$$\begin{aligned} (D_{1,1} + D_{1,3})\delta_1^2 &\leq 4\varepsilon, & (D_{2,2} + D_{2,1})\delta_2^2 &\leq 4\varepsilon, & (D_{3,3} + D_{3,2})\delta_3^2 &\leq 4\varepsilon, \\ (D_{1,3} + D_{3,3})\delta_3^2 &\leq 4\varepsilon, & (D_{2,1} + D_{1,1})\delta_1^2 &\leq 4\varepsilon, & (D_{3,2} + D_{2,2})\delta_2^2 &\leq 4\varepsilon, \end{aligned}$$

where for notational convenience we use subscripts 1–3 to refer to the parameter directions \mathbf{d}_{01} , \mathbf{d}_{12} and \mathbf{d}_{20} , respectively. We conservatively combine related inequalities, for instance to

$$(D_{1,1} + \max\{D_{1,3}, D_{2,1}\})\delta_1^2 \leq 4\varepsilon$$

for δ_1 , and then solve for δ_i , yielding the maximum sampling step sizes and the associated tessellation factors. Note that the derivative bounds $D_{p,q}$ are easily obtained from the maximum magnitudes of the related Bézier triangles' control points, for example

$$\begin{aligned} D_{1,1} &= n(n-1) \max_{i+j+k=n-2} \|\mathbf{b}_{i,j+2,k} - 2\mathbf{b}_{i+1,j+1,k} + \mathbf{b}_{i+2,j,k}\| \geq \sup_{\substack{u \in [0,1] \\ v \in [0,1-u]}} \|D_{\mathbf{d}_{01}}^2 \mathbf{b}(u, v)\|, \\ D_{1,3} &= n(n-1) \max_{i+j+k=n-2} \|\mathbf{b}_{i,j+1,k+1} - \mathbf{b}_{i+1,j+1,k} - \mathbf{b}_{i+1,j,k+1} + \mathbf{b}_{i+2,j,k}\| \\ &\geq \sup_{u \in [0,1], v \in [0,1-u]} \|D_{\mathbf{d}_{01}, -\mathbf{d}_{20}}^{1,1} \mathbf{b}(u, v)\| \end{aligned}$$

in case of a degree- n Bézier triangle $\mathbf{b}(u, v)$ with control points \mathbf{b}_{ijk} .

7.4.5 PN triangles

For the special setting of uniformly tessellating a PN triangle according to our tessellation pattern scheme (cf. Fig. 7.7 a), we devised a simple approach for quickly determining the tessellation factor [331].

First, we compute the distance d_{ijk} of each tangent control point \mathbf{b}_{ijk} from its corresponding edge in the base triangle. Considering the boundary curve $\mathbf{c}_1(t) = \mathbf{b}(1-t, t)$, we define an associated distance curve

$$d_1(t) = \sum_{i+j=3} d_{ij0} B_{ij0}^3(1-t, t)$$

with $d_{300} = d_{030} = d_{003} = 0$ for the vertex control points. For any parameter value $t' \in [0, 1]$, $d_1(t')$ provides an upper bound for the deviation of the curve point $\mathbf{c}_1(t')$ from the base triangle. Eventually, however, we are interested in bounding the distance of the boundary curve from its piecewise-linear approximation $\tilde{\mathbf{c}}_1(t)$ obtained by sampling with a step size δ . But simply constructing a corresponding approximation $\bar{d}_1(t)$ of $d_1(t)$ doesn't help because $|d_1(t) - \bar{d}_1(t)|$ may underestimate the geometric deviation if the boundary curve is non-planar or S-shaped.

Therefore, we temporarily employ a signed distance curve

$$s_1(t) = \sum_{i+j=3} s_{ij0} B_{ij0}^3(1-t, t) = 3(1-t)^2 t s_{210} + 3(1-t) t^2 s_{120}$$

with $|s_{ij0}| = d_{ij0}$, implicitly choosing the signs of the s_{ij0} such that the distance $|s_1(t) - \bar{s}_1(t)|$ to its piecewise-linear approximation $\bar{s}_1(t)$ becomes maximum. This distance then provides an estimate for the deviation of $\mathbf{c}_1(t)$ from $\tilde{\mathbf{c}}_1(t)$.⁹ Applying (7.2), we get¹⁰

$$\begin{aligned} \sup_{t \in [t_0, t_0+\delta]} |s_1(t) - \bar{s}_1(t)| &\leq \frac{1}{8} \delta^2 \sup_{t \in [t_0, t_0+\delta]} |s_1''(t)| \\ &\leq \delta^2 \cdot \frac{3}{4} \max_{t \in [t_0, t_0+\delta]} |(s_{120} - 2s_{210} + s_{300})(1-t) + (s_{030} - 2s_{120} + s_{210})t| \\ &\leq \delta^2 \max_{t \in [0,1]} \left| \left(\frac{9}{4}t - \frac{3}{2} \right) s_{210} + \left(\frac{3}{4} - \frac{9}{4}t \right) s_{120} \right| \\ &\leq \delta^2 \max_{t \in [0,1]} \left| \left(\frac{3}{2} - \frac{9}{4}t \right) d_{210} + \left(\frac{3}{4} - \frac{9}{4}t \right) d_{120} \right| \\ &\leq \delta^2 \cdot \frac{9}{4} \max\{d_{210}, d_{120}\}. \end{aligned}$$

⁹Note that $|s_1(t) - \bar{s}_1(t)|$ measures the geometric deviation in the direction perpendicular to the base triangle edge and not perpendicular to the corresponding polyline segment of $\tilde{\mathbf{c}}_1(t)$.

¹⁰We originally obtained this result for the special case $\delta = 2^{-\ell}$ by investigating the error term $4^k e_1(k, i)$, measuring the deviation $e_1(k, i) = s_1(k+1, 2i+1) - \frac{1}{2}(s_1(k, i) + s_1(k, i+1))$ at samples $s_1(k, i) = s_1(2^{-k}i)$.

Consequently, to keep the estimated deviation along all boundary curves below a threshold ε , a uniform sampling step size δ may be chosen such that

$$\frac{9}{4}d_{\max}\delta^2 \leq \varepsilon \quad (7.6)$$

is satisfied, where $d_{\max} = \max\{d_{210}, d_{120}, d_{021}, d_{012}, d_{102}, d_{201}\}$.

Note that because of the construction of the center control point \mathbf{b}_{111} according to (6.1), its distance from the base triangle is bounded by

$$\frac{3}{2}d_{\max} \geq \frac{1}{4}(d_{210} + d_{120} + d_{021} + d_{012} + d_{102} + d_{201}) \geq d_{111}.$$

Therefore,

$$d(u, v) = \sum_{i+j+k=3} d_{ijk} B_{ijk}^3(u, v) \leq d_{\max},$$

that is, d_{\max} actually provides an upper bound for the PN triangle's overall deviation from its base triangle.

Moreover, similar to the curve setting above, we may utilize the signed version of $d(u, v)$ and apply (7.3) to get an estimate of the maximum deviation of the whole patch from its uniform tessellation, yielding the condition

$$\frac{45}{4}d_{\max}\delta^2 \leq \varepsilon$$

for the sampling step size δ . In our experience, however, using the larger step size implied by (7.6) for the boundary curves works extremely well in practice.

Summing up, we hence typically proceed as follows. First, the tangent control point distances d_{ijk} and their maximum d_{\max} are determined, possibly in a precomputation step. To derive a uniform tessellation factor m for satisfying a given error tolerance ε , we then check whether $d_{\max} \leq \varepsilon$. If the test passes, the base triangle constitutes a good enough approximation, and we set $m = 1$. Otherwise, we solve (7.6) for δ and choose $m = \lceil 1/\delta \rceil$.

Note that by design, the uniform tessellation factor can be computed rapidly from the single quantity d_{\max} , which itself can quickly be determined. On the other hand, recall that using the bound (7.6) does not strictly guarantee that the approximation error is always within the tolerance ε , but it keeps the number of tessellation triangles low. These properties make the approach attractive for many real-time rendering applications, trading utmost visual quality for speed and simplicity.

7.5 Rendering of refinement patterns

Once appropriate tessellation factors are determined for a curved surface, an according tessellation can be generated and rendered. Typically, the tessellation follows a tessellation pattern corresponding to the factors, specifying the sample locations and their topological connection. Essentially two options exist for applying such a pattern:

- Either, the tessellation pattern is first explicitly generated in parameter space, thus providing a tessellation of the unit domain, and is then mapped onto the surface. Note that since each created pattern only depends on the tessellation factors and not on the concrete surface, it may be cached or even precomputed and can then be (re)used for all patches with the same factor configuration. Lately, such generic patterns in parameter space have often been referred to as *refinement patterns*.

- Alternatively, the surface is directly sampled as indicated by the pattern, and triangles for the resulting vertices are created. This approach essentially interleaves sample point generation and surface evaluation, and hence allows utilizing efficient computation techniques like forward differencing in the uniformly sampled parts of the pattern. Note that the tessellation pattern is employed in procedural form and not as explicit representation in parameter space.

While our patch-parallel tessellation framework described later in Sec. 7.6 pursues the second method, this section focuses on the first one. An early representative is the modular algorithm for tessellating trimmed NURBS surfaces by Rockwood et al. [315]. After processing the input to get Bézier patches and to simplify dealing with trimming, sampling step sizes are determined and a corresponding tessellation is generated in parameter space. Subsequently, the surface is evaluated at the vertex positions of this domain tessellation, and the resulting object-space triangles are rendered. Note that the parameter-space tessellations depend on the trimming regions and hence cannot reasonably be precomputed without knowledge of the actual surface.

By contrast, if only untrimmed Bézier patches are considered, like in our case, these tessellations depend solely on the required sampling step sizes. Consequently, after discretizing the range of step sizes to inverse integer tessellation factors, related refinement patterns may easily be shared among several patches.

When striving to utilize graphics hardware for mapping a refinement pattern to a surface, the evaluation at the pattern vertices may be performed in parallel, launching a single thread for each sample. Early approaches typically address only the uniform tessellation of rectangular domains. For instance, Bolz and Schröder [43] render a quad, providing the evenly spaced sample positions via interpolated texture coordinates, and evaluate the surface in the pixel shader. Finally, the result is bound as vertex buffer and rendered using a precomputed index buffer that contains the generic topology of the employed tessellation pattern. Note that the detour over the pixel shader stage was motivated by hardware constraints, precluding direct surface evaluation in the vertex shader.

7.5.1 GPU-based methods

Current GPU-based methods for rendering curved surfaces by mapping generic parameter-space tessellation patterns to screen space now typically provide a refinement pattern as triangle mesh input to the vertex stage, and execute the surface evaluation and the subsequent transformation to clip space in a vertex shader.

Targeting only uniform tessellations where the same tessellation factor is used for all patches of a model, and focusing on the triangular setting, Boubekur and Schlick [48] generate tessellation patterns in parameter space for a range of tessellation factors and store the resulting refinement patterns in vertex and index buffers. At runtime, an appropriate pattern is rendered for each surface patch. In the vertex shader, the provided vertex positions, which equal the (u, v) parametric coordinates of the sample points, are used to derive the actual surface points. Note that the same approach was independently proposed by Guthe et al. [150] for the case of a rectangular domain.

It is straightforward to extend this method to non-uniform tessellation factor configurations and thus to adaptive tessellations [49, 51]. However, supporting all possible combinations of factor values up to some maximum factor value may result in a large number of refinement patterns, easily consuming a significant amount of memory (cf. Sec. 7.5.4). Therefore, in prac-

tice often only dyadic tessellation factors, i.e. factors with a power-of-two value, are utilized. Another option is to solely employ uniform refinement patterns and adapt them on the fly in the vertex shader by collapsing vertices on the boundary to yield the desired non-uniform tessellation [102, 375]. This semi-uniform approach is further detailed in Sec. 7.5.4.

A related method is proposed by Dyken et al. [101], who render dyadic uniform refinement patterns and apply a kind of geomorphing during the surface evaluation to geometrically achieve continuity across patches. However, since the tessellation topology itself is not modified, T-vertices and hence rendering artifacts can occur.¹¹

Alternatively, one may refrain from trying to make adjacent patch tessellations consistent and utilize only uniform refinement patterns. The resulting cracks are then filled by rendering some additional geometry [150, 331]. We developed such a technique for the special case of PN triangles, which also closes gaps inherent to PN triangle meshes constructed from coarse base meshes with corner-like features. The next subsection provides further details and presents results.

Note that rendering refinement patterns requires a separate draw call for each patch or at least for each employed pattern. Approaches to reduce this overhead, which is particularly pronounced at low tessellation rates, are discussed in Secs. 7.5.3 and 7.5.4. A rather different solution is proposed by Lorenz and Döllner [233], who specifically target applications with small refinement levels. They store all refinement patterns as a list of individual triangles in a pattern vertex buffer, and then employ a geometry shader to emit a refinement pattern for each patch, copying the triangles from the pattern vertex buffer. The output, comprising a separate refinement pattern instance per patch, is captured in a stream output buffer and subsequently rendered with a single draw call to perform the mapping to the actual surfaces. While this approach indeed keeps the number of API invocations small, it necessitates multiple rendering passes and a considerable amount of intermediate storage. Moreover, since patterns are output as isolated triangles, interior pattern vertices are duplicated and hence redundant surface evaluations are performed.

7.5.2 Connection patterns for dyadic tessellation of PN triangle meshes

The visual smoothness of a coarse triangle model, especially at the silhouettes, can be improved by replacing each flat triangle with its corresponding curved PN triangle. In the following, we present our approach [331] for rendering such PN triangle meshes utilizing refinement patterns. Motivated by the desire to keep the number of different patterns comparatively low, we restrict ourselves to uniform patterns and dyadic tessellation factors.

Typically, however, the tessellation factors derived for the PN triangles of a model are not identical, and hence there are some pairs of adjacent PN triangles which are tessellated to different degrees and whose common boundary curve is thus approximated by different polylines. As a consequence, cracks are introduced in the resulting model tessellation. Referring to Fig. 7.12, basically two situations can be distinguished at such places of discontinuity in sampling density. Either tiny holes appear due to the cracks (a) or the refinement patterns rendered for two adjacent PN triangles overlap slightly (b). While the first case can lead to visible artifacts as single pixels along the common boundary curve may be omitted and reveal the background, the second setting is usually less problematic. In particular, if the screen-space error threshold prescribed when deriving the tessellation factors is chosen small enough, no visible discontinuities appear at all.

¹¹Interestingly, the authors don't discuss this and call their tessellation watertight.

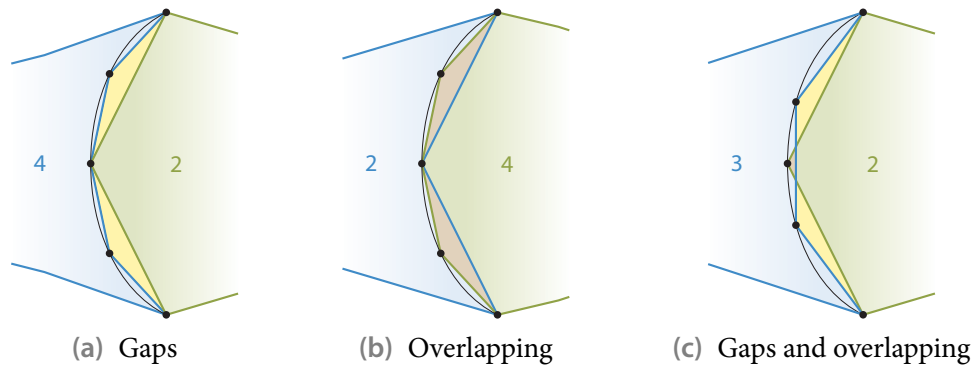


Figure 7.12 If different tessellation factors are used for adjacent PN triangles, often cracks appear. They result in gaps (yellow regions) and partial overlapping (brown regions).

To avoid visible gaps caused by the inconsistent tessellation, we stitch occurring cracks with additional geometry. For each pair of adjacent PN triangles using different refinement patterns, a general triangle strip is rendered that connects the two employed polyline approximations for the affected common boundary curve. The yellow region in Fig. 7.12 a is an example of such a connection strip. Note that the topology of the strip as well as the parameter-space sample points corresponding to its vertices only depend on the two tessellation factors for the boundary curve. Analogous to refinement patterns, we therefore generate generic *connection patterns* for all possible factor configurations. These are then used for rendering the connection strips, with the mapping from parameter space onto the boundary curve being performed in the vertex shader.

In situations where the refinement patterns for adjacent PN triangles overlap, crack stitching results in mesh fold-overs, which could lead to visual artifacts, especially if a large screen-space error threshold is chosen for deriving the tessellation factors. To alleviate such disturbances, we check for these cases and skip rendering those triangles of the connection strips which would cause a fold-over. For reasons of simplicity, consistent numerical precision and speed, we don't perform the necessary tests on the CPU but render the connection patterns with back-face culling enabled, which works fine in practice for closed models.

Note that in case the tessellation factors of two adjacent PN triangles differ by more than a factor of two and the shared boundary curve has an S-like shape, minor imprecisions can occur because then the two involved polyline approximations may overlap at non-vertex positions, potentially causing triangles of the connection strip to both close gaps and to introduce additional overlap. However, thanks to our restriction to dyadic refinement patterns these situations are very rare. By contrast, if non-power-of-two tessellation factors were allowed, such settings would be encountered quite often (cf. Fig. 7.12 c). Moreover, a much larger number of connection patterns would have to be generated, significantly increasing the memory load.

If the coarse base mesh has not been modeled explicitly for being rendered with PN triangles, it might happen that base triangles sharing a single vertex provide different normals for this common vertex to model features like corners. In principle, such a situation could be dealt with by using the tangent line construction detailed in Sec. 6.1.3 for deriving the tangent control points close to this crease vertex, which, however, requires providing normal data from the adjacent base triangles as additional input. If instead the purely local, standard PN triangle construction is pursued, then adjacent base triangles can lead to PN triangles that don't share a common boundary curve. For instance, referring to Fig. 7.13 a, the common edge AC becomes

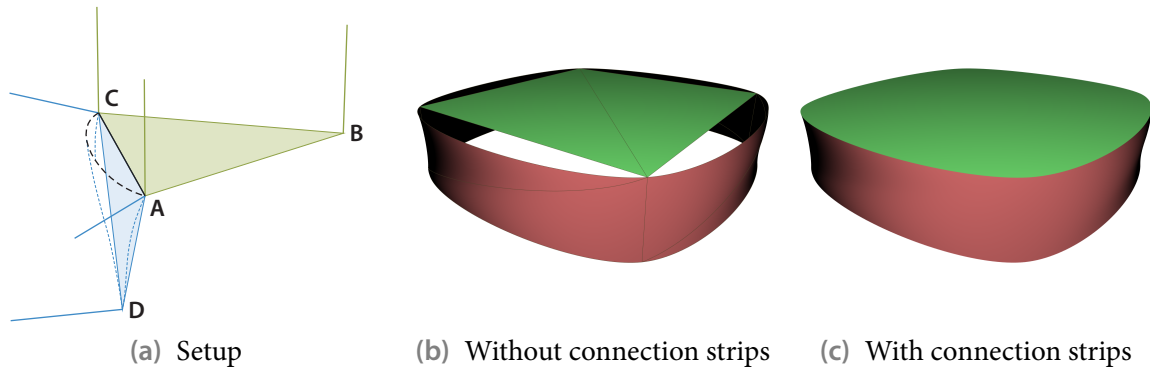


Figure 7.13 Adjacent base triangles with different per-vertex normals at their common edge **AC** (a) can lead to holes in the resulting PN triangle mesh (b). These can be closed with connection strips (c).

a boundary curve of PN triangle **ABC**, whereas it results in the dashed curve in case of PN triangle **DAC**, i.e. a hole appears in the PN triangle mesh not present in the coarse base mesh.

Note, however, that as a side-effect of performing stitching by rendering connection strips, such gaps inherent to the PN triangle mesh get closed (cf. Fig. 7.13 bc). Therefore, if neighboring PN triangles have different per-vertex normals specified, we render a connection pattern even in case the involved tessellation factors are equal. Concerning the pattern's normal field, we adopt the heuristic of using the normals along the boundary curve of that affected PN triangle whose base triangle's face normal deviates less from the average face normal of the connection strip for uniform tessellation factor 2.

Discussion

While our method is simple and fast, it suffers from several shortcomings. First, to avoid artifacts, we require the screen-space error tolerance used for deriving tessellation factors to be chosen small enough, e.g. as half a pixel. However, this is reasonable, anyway, to ensure visual smoothness of the silhouettes. Note that if, nevertheless, the error bound is set to permit deviations of several pixels, overlapping refinement patterns may lead to visible artifacts. Moreover, the connection patterns may become clearly visible as their normal fields stay constant in the surface direction perpendicular to the approximated boundary curves.

Another issue is the limited support for texturing. Since the tessellation is not consistent, artifacts may occur in regions where the two polyline approximations for a shared boundary curve don't abut, at least if a texture with high-frequency content is applied. In particular, severe texture stretching may be noticeable on those connection strips which close a gap inherent to the PN triangle mesh. This special case, however, is a direct consequence of inappropriate input data. Also recall that in general C^1 continuity is required to obtain good-looking texturing, while PN triangles only meet with C^0 continuity.

Further note that the ability to stitch holes inherent to the PN triangle mesh is not a panacea to deal with arbitrary input data and doesn't replace careful modeling. For instance, providing different normals for a vertex shared by two base triangles may not only lead to gaps but to more severe artifacts, like neighboring PN triangles which intersect.

The general approach of filling cracks with additional geometry is long known and can be traced back at least up to Nydegger's 1972 M.Sc. thesis [274]. By carefully restricting the al-

lowed tessellation factors to power-of-two values and alleviating mesh fold-overs via back-face culling, our method both successfully avoids major artifacts and is fast and simple. This is in contrast to fat borders [20], another technique for closing gaps resulting from inconsistent tessellations. Here, for each affected boundary curve approximation, a triangle strip is constructed such that it corresponds to this polyline rendered with screen-space thickness 2ε , where ε denotes the error tolerance used for deriving the tessellation factors. Building and rendering these fat borders is more expensive than our approach; moreover, they are prone to visual artifacts.

Implementation notes

In our example implementation, we use the method described in Sec. 7.4.5 to determine the tessellation factors for the PN triangles. Assuming only static geometry, we compute the control points as well as the quantity d_{\max} for each PN triangle in a preprocess. At the beginning of each frame, d_{\max} is then projected into screen space and subsequently compared with the user-specified screen-space error threshold ε to derive the required dyadic tessellation factor.

The restriction to power-of-two factors implies that the vertices of any refinement pattern for a smaller tessellation factor are a real subset of the ones of the pattern for the largest factor. We exploit this hierarchical structure to keep the memory requirements low, storing only the vertices of the most detailed pattern in a vertex buffer and providing corresponding index buffers for all refinement patterns. An analogous optimization is done for the connection patterns.

When rendering the patterns, the vertex shader evaluates the PN triangle's geometric component to map the vertex to the surface, while the normal component is only computed in the pixel shader, ensuring high visual quality.

Results

Some example scenes rendered with our method are depicted in Figs. 7.14 and 7.15. Note that the employed triangular models were not explicitly modeled with suitability for applying the PN triangle scheme in mind. The involved triangle counts are listed in Table 7.1, and the achieved rendering performance is shown in the “Batched rendering” columns of Table 7.2. All results were obtained with an Intel Core 2 Quad Q9450 processor and an NVIDIA GeForce GTX 280 graphics card at a viewport of size 1600×1200. We used a screen-space error bound of half a pixel for the computation of the tessellation factors, and never observed any visual artifact.

The Venus statue's tessellation in Fig. 7.15 b exemplifies the occurrence of holes when only uniform refinement patterns are rendered, caused either by cracks or by specifying multiple normals at certain vertices (cf. the red-colored regions in Subfig. d, e.g. at the left arm stump). The effectiveness of using connection patterns for closing these gaps is demonstrated in Subfig. c.

Closer inspection of the data in Table 7.1 reveals that in our example scenes 7% to 30% of all pairs of abutting PN triangles employ different refinement patterns and hence necessitate rendering a connection pattern to stitch the cracks along their shared boundary curve. However, the drawn connection strips increase the overall number of rendered triangles by merely 4% to 13% and hence incur only a minor overhead.

Our approach essentially yields an adaptive tessellation for a model's PN triangle mesh, composed of uniformly tessellated PN triangles and crack-filling connection strips. To facilitate

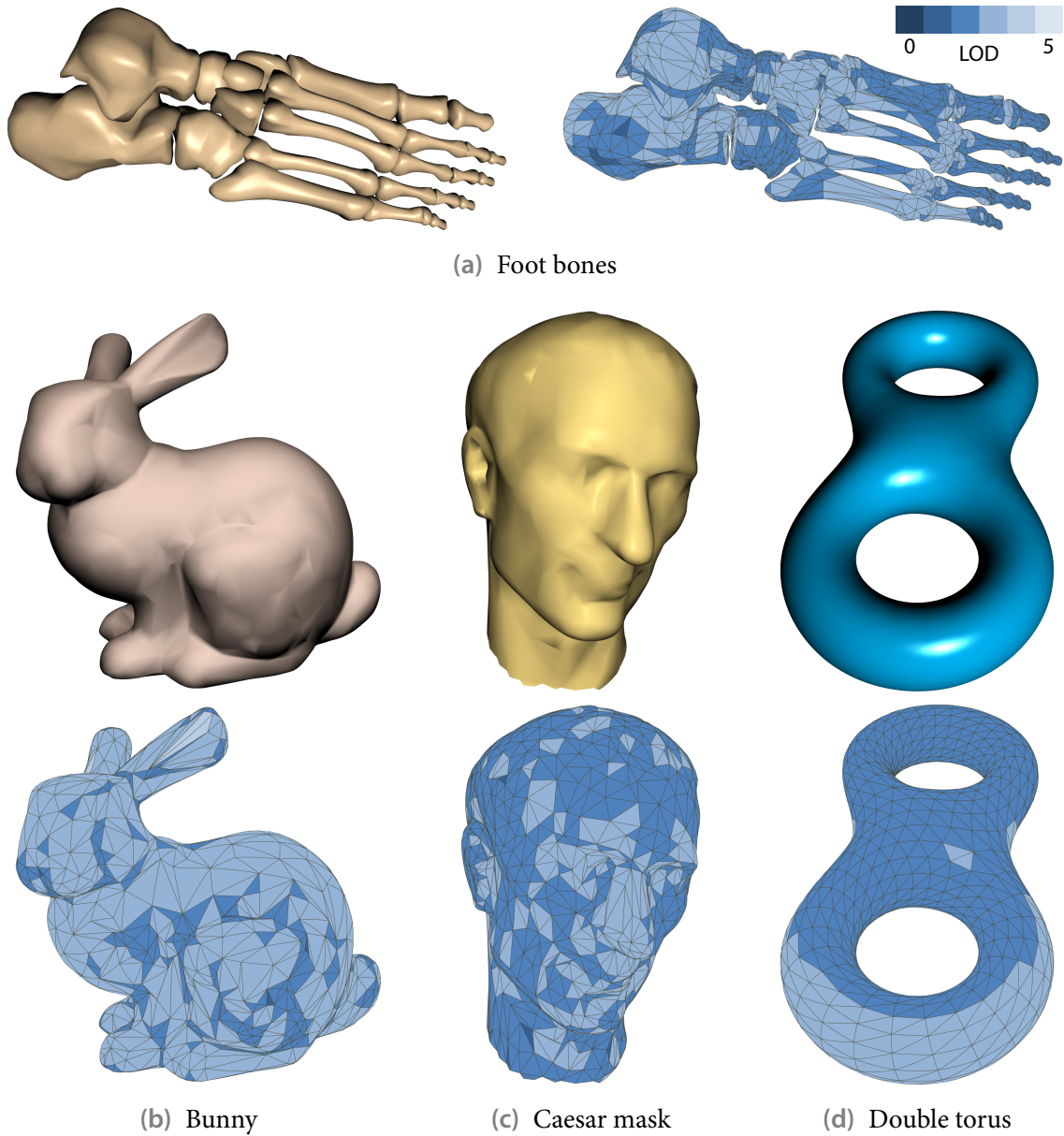


Figure 7.14 Examples rendered with refinement and connection patterns. The color coding visualizes the employed refinement patterns, with LOD value ℓ corresponding to tessellation factor $m = 2^\ell$.

assessing the achieved performance, we also considered the base tessellation, where all tessellation factors are set to one and whose geometry hence equals the coarse base mesh. Since the PN triangles' normal fields are still evaluated on a per-fragment basis, this provides an upper bound on the attainable performance which takes shading into account. Typically, turning a coarse geometry into a visually smooth one with our adaptive tessellation causes a slow-down of less than 50%, which is usually highly acceptable.

On the other hand, we also performed a comparison with the scene-uniform tessellation obtained by using the largest encountered tessellation factor for all PN triangles. Since consequently only one refinement pattern is used throughout, no cracks occur and hence no con-

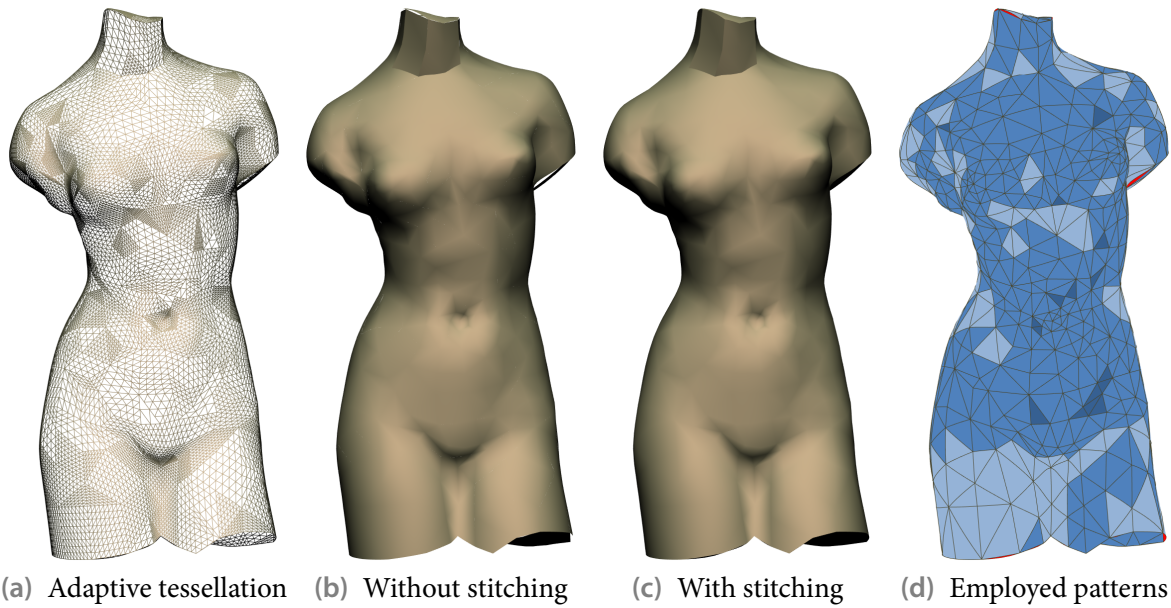


Figure 7.15 Venus statue example rendered with our method. In Subfig. d, blue regions indicate refinement patterns (see Fig. 7.14 for the color key), while red parts are stitched by connection patterns.

| Scene | Base mesh | Triangle count | | | Shared boundary curves | |
|--------------|-----------|-----------------------|-------------------|----------------------|------------------------|-------|
| | | Adaptive tessellation | Connection strips | Uniform tessellation | Stitched | Total |
| Foot bones | 4,204 | 206,722 | 19,110 | 1,076,224 | 1,544 | 6,306 |
| Bunny | 2,000 | 118,818 | 10,675 | 512,000 | 845 | 3,000 |
| Caesar mask | 2,000 | 90,684 | 10,720 | 512,000 | 890 | 2,960 |
| Double torus | 1,536 | 50,976 | 2,016 | 98,304 | 168 | 2,304 |
| Venus statue | 1,418 | 58,816 | 6,887 | 90,752 | 589 | 2,127 |

Table 7.1 Tessellation statistics for the PN triangle example scenes from Figs. 7.14 and 7.15. Note that the triangle count for adaptive tessellation includes the contribution from the connection strips.

| Scene | Batched rendering | | | Immediate Adaptive | Draw calls (adaptive) | |
|--------------|-------------------|----------|---------|--------------------|-----------------------|-----------|
| | Base | Adaptive | Uniform | | Batched | Immediate |
| Foot bones | 973 Hz | 530 Hz | 214 Hz | 143 Hz | 9 | 5,748 |
| Bunny | 1,033 Hz | 571 Hz | 330 Hz | 214 Hz | 13 | 2,845 |
| Caesar mask | 1,293 Hz | 737 Hz | 372 Hz | 187 Hz | 8 | 2,890 |
| Double torus | 1,254 Hz | 891 Hz | 751 Hz | 282 Hz | 3 | 1,704 |
| Venus statue | 1,614 Hz | 1,015 Hz | 943 Hz | 245 Hz | 11 | 2,007 |

Table 7.2 Rendering performance for the base, the adaptive and the uniform tessellation of the PN triangle example scenes from Figs. 7.14 and 7.15. In batched rendering all instances of a pattern are rendered with a single draw call, whereas in immediate rendering a draw call is issued for each patch and each connection strip.

nection patterns need to be rendered. For the listed example scenes, our adaptive tessellation results in 1.5 to 5.6 times fewer triangles being rendered and is about 1.1 to 2.5 times faster than uniform tessellation. This indicates that performing an adaptive tessellation instead of the simpler uniform one is clearly worthwhile despite the crack filling overhead.

7.5.3 Instanced rendering

When pursuing the current approach of rendering refinement patterns as geometry and performing the surface evaluation in the triggered vertex shader, a practical challenge is how to provide the patch-specific data required for surface evaluation, like control points, and how to efficiently issue the refinement patterns at the same time. The first realizations [48, 150] but also more recent extensions [49, 51] don't explicitly address this issue but just process each patch individually, specifying the needed patch data as constant shader input (e.g. as uniform variables in OpenGL) and rendering the refinement pattern as indexed triangle mesh. While simple and easy to implement, this *immediate rendering* approach incurs a high API invocation overhead as a draw call is performed for each patch. This overhead is typically only negligible in case of very high tessellation factors but becomes clearly dominating for smaller factors, noticeably hampering performance.

For the special setting where patches are defined by a coarse triangle mesh and the majority of them needs no refinement, having tessellation factors of value one, Dyken et al. [101] suggest first rendering all coarse triangles, degenerating those tagged for further refinement. Note that this essentially aggregates the rendering of all patches with tessellation factors of one into a single draw call. Subsequently, they process the remaining patches with tessellation factors larger than one individually as described above, rendering a refinement pattern with a separate draw call for each of them.

A better and more general solution, however, is to batch together all patches using the same refinement pattern and employ instancing for rendering. Hence, identical draw calls are grouped into a larger batch, which is then issued with just a single API call. Although quite obvious,¹² and early exploited by Gruen [143], only recently instanced rendering of refinement patterns gained wide interest [102, 375].

To apply instanced rendering, all patches using the same refinement pattern have to be identified and some mapping from the instance id to the actual patch be provided. Typically, this is done by binding a vertex buffer with corresponding per-instance data. As this buffer must potentially be rebuilt every frame, the stored patch-wise data should ideally have a small memory footprint. In our implementation, we hence only write the patch id into this instance vertex buffer. The id is then used in the shaders as an index to access other buffers storing patch-specific data like control points.

Overall, our approach for batched rendering of refinement patterns is as follows. We first determine the tessellation factors for all patches and build a queue for each refinement pattern, containing the ids of all patches with the corresponding factor configuration. Subsequently, we loop over all queues, and for each non-empty queue, we copy the ids to an instance vertex buffer and issue an instanced draw call for the refinement pattern. As the results in Table 7.2 show, this significantly reduces the number of draw calls compared to immediate rendering, where each patch is treated individually. More importantly, batched rendering is significantly faster.

¹²When we originally implemented our scheme from Sec. 7.5.2, we intended to employ instancing. However, back then only Direct3D but not OpenGL, which we used, exposed this feature. We thus resorted to pseudo-instancing [412], specifying the patch data as persistent vertex attributes and issuing a draw call per patch.

| Maximum tess. factor | Adaptive, integer | | Adaptive, dyadic | | Uniform, integer | | Uniform, dyadic | |
|-------------------------|-------------------|-------------|------------------|---------|------------------|---------|-----------------|---------|
| | Patterns | Indices | Patterns | Indices | Patterns | Indices | Patterns | Indices |
| 4 | 64 | 2,028 | 27 | 774 | 4 | 60 | 3 | 42 |
| 8 | 512 | 48,696 | 64 | 5,086 | 8 | 312 | 4 | 130 |
| 16 | 4,096 | 1,153,520 | 125 | 27,770 | 16 | 1,904 | 5 | 434 |
| 32 | 32,768 | 29,034,976 | 216 | 142,902 | 32 | 13,024 | 6 | 1,554 |
| 64 | 262,144 | 790,916,032 | 343 | 725,206 | 64 | 95,680 | 7 | 5,842 |

Table 7.3 Number of different refinement patterns for a triangular domain and corresponding overall number of indices used for specifying their topologies. Resorting to only dyadic tessellation factors and especially using solely (adapted) uniform patterns significantly reduces these numbers.

In our examples, where on average 33 to 49 triangles are used for the tessellation of each PN triangle, indicating medium-sized tessellation factors, we observe frame rate improvements by a factor of 2.7 to 4.1.

While we perform the tessellation factor computation and the grouping of patches into batches for instanced rendering on the CPU, Dyken et al. [102] present a method where these steps are executed on the GPU. Note that building the per-pattern queues still involves CPU control; in particular, the number of enqueued patches must be read back to main memory. Their results suggest that such a GPU-based approach is only reasonable if many patches are rendered and the number of employed refinement patterns is rather small.

7.5.4 Number of refinement patterns

In general, when rendering refinement patterns, we strive to build large batches to amortize the draw call overhead over many patches and thus keep the per-patch overhead to a minimum. However, we have to create at least one batch for each employed refinement pattern. Therefore, the number of batches can become rather high if the allowed tessellation factor configurations are not restricted because the number of tessellation factor combinations grows exponentially as a function of the maximum supported tessellation factor.

This easy explosion of the number of possible refinement patterns, especially for rectangular domains, is a more fundamental problem because the patterns need to be precomputed and stored in vertex and index buffers. To make this more concrete, Table 7.3 lists the number of different patterns in the triangular domain setting for several choices of the maximum tessellation factor. It also includes the number of indices required to specify the topology of these patterns, assuming our tessellation pattern scheme from Sec. 7.3.4 is used. For instance, if all combinations of integer tessellation factors up to a maximum value of 64 are to be supported, then 1.47 GB of 16-bit index buffer data have to be created and stored. In addition, vertex buffer data encoding the sample points is required. Consequently, it is questionable whether providing patterns for all integer tessellation factor configurations is reasonable or possible in practice.

Choosing the maximum supported tessellation factor to be small is in general not a good solution, as this may impact visual quality, leading to large approximation errors when zooming in. A better option is to restrict the tessellation factors to power-of-two values. However, since the resulting dyadic tessellation typically has more sample points than required, this can incur a significant evaluation overhead compared to ordinary integer tessellation. For example, 228%

more vertices need to be processed for a triangular domain if instead of a tessellation factor of 17 the next larger dyadic factor 32 is employed.

Another alternative is to decompose each refinement pattern into patterns for the uniformly tessellated core and the transition regions. But this leads to redundant evaluations and complicates batch building. One may also consider exploiting symmetries among the patterns. Note, however, that this requires a dynamic remapping of the parametric coordinates, for instance swapping u and v or replacing u by $1 - u$, and hence introduces some additional overhead.

Adapted uniform tessellations

An interesting option that greatly reduces the number of precomputed refinement patterns is to use just uniform patterns and translate the sample points along the boundaries during rendering such that they coincide with the sample points for a potentially smaller boundary-specific tessellation factor. If this boundary factor differs from the uniform factor, some adjacent boundary sample points are moved to the same point, thus collapsing the connecting edge and degenerating the related triangle. Overall, this procedure results in an adaptive tessellation which uses the same sample points like the equivalent non-uniform pattern but features some duplicate samples, which lead to redundant surface evaluations.

Note that the approach itself is rather long known. For instance, to close cracks between the dyadic tessellations of two abutting patches, Sharp [355] moves each boundary vertex unique to the finer tessellation to the closest common vertex. Nankervis [267] provides a geometry-shader-based height field tessellation demo program where the samples along the boundaries are smoothly morphed to the positions of the next coarser dyadic tessellation. Specifically for tessellation via rendering of refinement patterns, the concept is applied by Tatarinov [375] as well as by Dyken et al. [102], who call it semi-uniform adaptive tessellation.

Given dyadic boundary tessellation factors $m_i = 2^{e_i}$, $1 \leq i \leq 3$, and an internal factor $m_0 = 2^{e_0}$ for a triangular domain, we first determine the maximum tessellation factor $n = \max_i \{m_i\} = 2^g$. Then, a uniform refinement pattern for factor n is rendered, with the tessellation being dynamically adapted to the boundary factors m_i in the vertex shader before performing the surface evaluation. More precisely, considering the boundary curve $w = 0$ as example, we map a related boundary sample point's barycentric coordinates $(u, 1 - u, 0)$ to $(\sigma_{m_1}(u), 1 - \sigma_{m_1}(u), 0)$ using the snap function [102]

$$\sigma_m(t) = \frac{1}{m} \begin{cases} \lceil mt - \frac{1}{2} \rceil, & t < \frac{1}{2}, \\ \lfloor mt + \frac{1}{2} \rfloor, & t \geq \frac{1}{2}. \end{cases}$$

The case distinction, which we efficiently implement according to the reformulation

$$\sigma_m(t) = \frac{\lfloor mt + \frac{1}{2} - 2^{-\varepsilon-1} + 2^{-\varepsilon}t \rfloor}{m}$$

with integer constant $\varepsilon > g - e$, $e = \log_2 m$, is not strictly necessary but yields a more symmetric pattern. This scheme essentially performs an on-the-fly subsampling of the boundary sample points, moving dropped vertices to the closest remaining boundary vertex.

We observe, however, that this approach is not restricted to dyadic tessellations but can also directly be applied to non-power-of-two values of m_i and n . While it then generally no longer performs subsampling, it again rounds the barycentric coordinates of an input boundary point to those of the closest actual boundary vertex. Note that this rounding yields a fair distribution

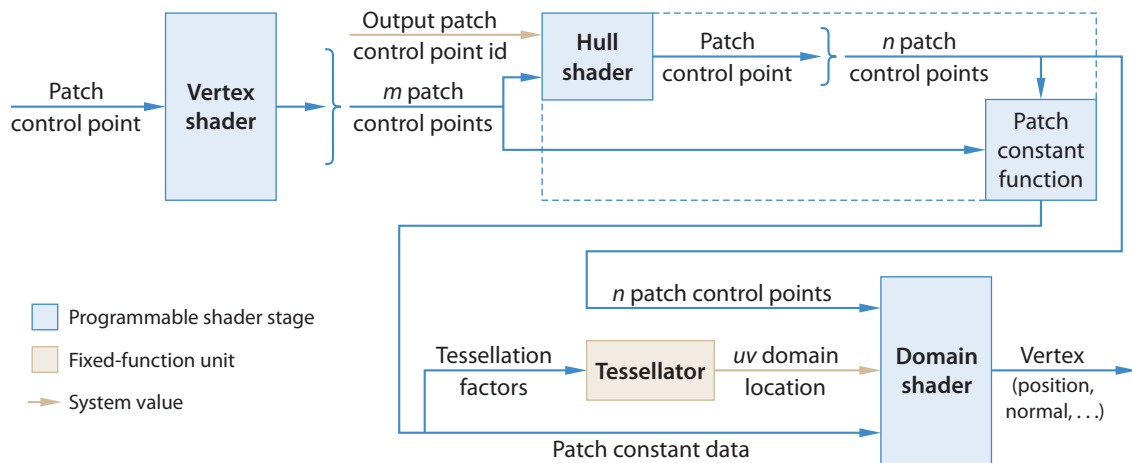


Figure 7.16 Direct3D 11 comes with native tessellation support, introducing three new pipeline stages and patch input primitive topologies (with a fixed number of $m \leq 32$ control points).

of original sample points to actual ones. It hence comes as no surprise that the resulting transition regions are essentially identical to those obtained with the Bresenham-like algorithm [255] detailed in Sec. 7.3.3. Apart from the degenerate triangles, the only difference are occasionally some flipped edges, which, however, actually improve the shapes of the triangles towards equilaterality in parameter space.

7.5.5 Direct hardware support

Recent AMD graphics chips like the Xbox 360's Xenos or the Radeon R600 and R700 [373, 374] feature a tessellation unit that is capable of generating refinement patterns on the fly. Preceding the vertex shader stage, it is invoked by rendering for each patch an input primitive of the same topology as the patch's domain and additionally providing tessellation factors. Each generated tessellation vertex has its parametric sample position as attribute but also the input data for all vertices of the rendered primitive. The tessellator supports lines, triangles and quads as input, and can perform fractional tessellation.

Compared to ordinary rendering of refinement patterns, where a pattern's triangle mesh is explicitly drawn, the tessellator-based approach has several advantages. It admits optimal batching, requiring only a single draw call for the whole model instead of one per employed refinement pattern. The dynamic pattern generation in the tessellation stage also avoids the explicit storage of all supported refinement patterns in vertex and index buffers. It is hence not necessary to restrict the allowed tessellation factor combinations to keep the overall number of patterns low. Moreover, it saves both memory bandwidth and space.

Direct3D 11

Currently, hardware-based tessellation is hardly used in practice (except on the Xbox 360), not least because for some reason AMD didn't make any API for utilizing the tessellator in their PC graphics cards publicly available until recently. However, this situation will probably change significantly once graphics hardware for the upcoming Direct3D 11 [252] is available. Recognizing the importance of curved surfaces and real displacement mapping for high visual quality, Direct3D 11 features native support for tessellation. Most notably, it introduces three

new related pipeline stages: a fixed-function tessellator and the programmable hull and domain shader stages (see Fig. 7.16). These provide a general tool and framework to efficiently and directly perform tasks involved in the tessellation of curved surfaces, like transforming control points, computing tessellation factors, and providing control points for surface evaluation.

In addition to points, lines, triangles, and their adjacency-augmented versions, new input primitive topologies are introduced for patches with a fixed number m of control points ($1 \leq m \leq 32$). That is, m input vertices form a patch primitive with corresponding control points. These can then be transformed individually in the vertex shader, and are subsequently input as an array to the hull shader stage. This stage serves to determine tessellation factors and to transform the input patch into a different representation which simplifies surface evaluation, yielding an output patch with n control points. For instance, when a base mesh facet of a Catmull-Clark subdivision surface along with its one-ring neighborhood constitutes an input patch, it may be converted into an approximating bicubic Bézier patch (cf. Sec. 6.3.2) as this is far easier to process.

The hull shader stage actually consists of two parts, which operate at different granularities. For each of the n output patch control points, one instance of the main hull shader is executed, computing the new control point from the provided input patch control points. Moreover, a patch constant function is run for each patch. Having access to the control points of both the input and the output patch, this function serves to calculate and output the boundary and internal tessellation factors. It may also output further patch-specific data, like color or, in case of a PN triangle, the normal field control points.

The tessellator supports linear, triangular and rectangular unit domains, as well as integer and fractional (even and odd) tessellation schemes (cf. Sec. 7.3). Given the tessellation factors, which may not exceed a value of 64, it generates an according tessellation of the specified unit domain, i.e. it essentially emits a refinement pattern.

Subsequently, a domain shader instance is launched for each sample point of the output tessellation, performing the actual surface evaluation. Note that the patch control points as well as the patch constant data output by the hull shader stage are readily available as shader input. Unlike the situation when the surface evaluation is performed in the vertex shader, it is hence not necessary to explicitly (re)fetch this data from a buffer or texture for each tessellation vertex.

7.6 Patch-parallel on-the-fly tessellation

The approach of rendering refinement patterns performs sample point generation and surface evaluation in two distinct steps, first creating a tessellation pattern in parameter space, rendering it, and then mapping it onto the surface in the vertex shader. By contrast, it is also possible to directly create the surface points when applying a tessellation pattern, immediately evaluating the surface at the current sample position before producing the next sample.

Note that such a merging and interleaving of parameter-space sample point generation and surface evaluation affects the options for parallelization. Since sample generation is typically a sequential process, while the sampling of each surface patch depends only on the patch itself and is guided by its local tessellation factors, the most adequate unit of parallelism for this step is an individual patch. Consequently, the direct approach which immediately evaluates the surface may be executed patch-parallelly. In the refinement pattern method, on the other hand, the surface evaluation step has a mesh of individual samples as input and is hence naturally performed sample-parallelly, launching a vertex shader instance for each pattern vertex. However,

both the up-front generation of all sample points and especially their independent treatment make it hard to exploit inter-sample coherence, caused, for instance, by uniform spacing and accessing the same patch control points. By contrast, such optimizations are trivially possible when pursuing the patch-parallel direct approach.

Concerning GPU utilization, the geometry shader stage with its amplification capability could be employed to realize the patch-wise processing, directly creating and outputting the surface tessellation using one shader instance for each input patch. However, such an implementation approach doesn't seem worthwhile due to several severe restrictions faced. For instance, the geometry shader's output is basically a vertex list with triangle strip connectivity, requiring interior vertices (with all their attributes) to be emitted at least twice. Also, a geometry shader can typically output at most 1024 32-bit floating-point values per input primitive, limiting the maximum tessellation factors, e.g. to 12 for a triangular domain if just positions and normals are emitted. Even worse, in practice, the shader output must be restricted to far less than the possible 1024 values to avoid severe performance drops. Although this limitation may be alleviated by a multi-pass approach, overall performance will be negatively affected by the overhead entailed. Furthermore, available performance data for approaches employing a geometry shader for outputting a tessellation pattern [102, 233] suggest that even in case of small tessellation factors, rendering refinement patterns performs significantly better.

A better alternative is to use a compute API like NVIDIA's CUDA [273], which offers more flexibility and explicit control, and permits output of variable and almost unbounded size per thread via its scatter memory write capability. Utilizing CUDA, we developed a related framework for adaptive tessellation, called *CudaTess* [334, 337]. It runs all major steps, like deriving consistent tessellation factors, determining and evaluating surface sample points, and creating the tessellation topology, completely on the GPU. In particular, we efficiently construct tessellation patterns on the fly, readily employing patch-scale optimization techniques like forward differencing [390], which are not applicable in vertex-parallel settings.

After giving an overview of our generic framework in the next subsection, we demonstrate its potential by applying it to two concrete examples, bicubic rational Bézier patches (Sec. 7.6.2) and PN triangles (Sec. 7.6.3). In both cases, real-time performance is achieved even for large collections of surface patches. Subsequently, we discuss strengths and limitations and provide a comparison with the competing approach of rendering refinement patterns.

7.6.1 CudaTess framework for adaptive tessellation

Although we focus on rendering curved surfaces, our CUDA-based *CudaTess* framework essentially supports arbitrary surface primitives which are parameterized and can be evaluated directly at any parametric sample location. It adaptively tessellates all surface primitive instances in a scene, conveniently referred to as *patches*, in parallel and outputs the resulting triangle meshes into vertex and index buffers for rendering. While the general approach outlined in Fig. 7.17 is the same for all kinds of surface primitives, the actual implementation is usually specific to each kind of primitive.

In the first stage, the tessellation factors are determined using criteria like those covered in Sec. 7.4. Depending on the adopted criterion, the computation is performed on the level of either patches, (patch) edges and boundary curves, or (base mesh) vertices¹³. Each such *element*

¹³For instance, one may derive a per-vertex factor as a function of camera distance, and later—during the actual tessellation generation—choose each boundary tessellation factor as the maximum of the factors of the boundary's two endpoints.

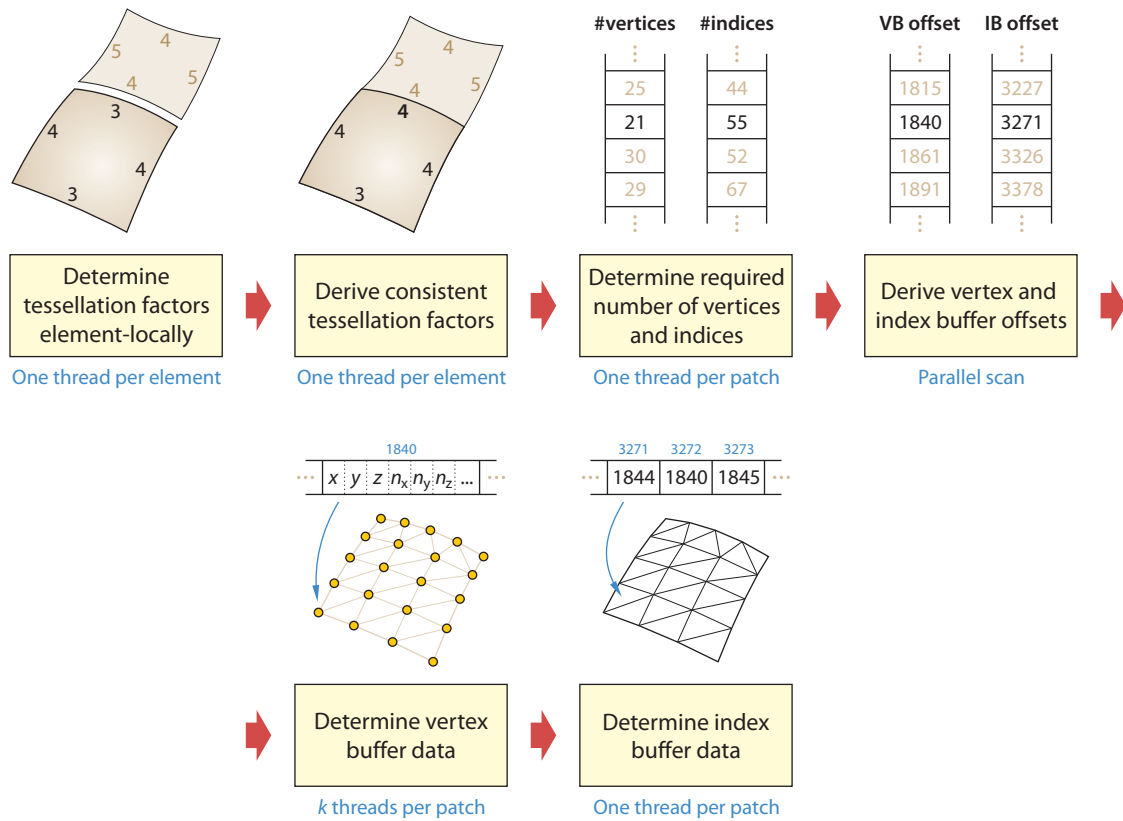


Figure 7.17 Overview of the CudaTess framework for adaptive tessellation.

is usually treated independently of the other elements in a separate thread. It is also possible to perform view-frustum or back-face culling at this stage if patches serve as elements; affected patches can be flagged by setting their tessellation factors to zero.

Recall that to avoid any cracks in the tessellation, boundary curves shared by multiple patches must be sampled consistently. Therefore, we adapt the elements' tessellation factors appropriately in the next step using both neighborhood information provided by the application and the original element tessellation factors from the first stage. Note that such factor modifications are usually only required if the elements are patches.

Once the final tessellation factors have been computed, the actual tessellation of each patch can be carried out. Processing all patches in parallel, we ultimately generate a vertex list of the sampled surface points and a corresponding index list describing the topology for each patch. These variable-sized lists are stored contiguously in one vertex and one index buffer, respectively. To directly populate these buffers, we first derive according optimal-sized slots. Based on the tessellation factors, the number of vertices and the number of indices required for the tessellation of each patch are computed and stored in two arrays. Subsequently, we run an exclusive parallel scan¹⁴ [35, 350] on these arrays to obtain the slot offsets within the vertex and the index buffer for the vertex and index data of each patch. To get the required total buffer sizes, we pad the two input arrays with a zero and then read back the last entry of the scan results. If necessary, the vertex and index buffer are resized appropriately.

After that, a patch's surface is directly evaluated at sample points generated on the fly ac-

¹⁴An exclusive scan computes for each element a_k of a list a_1, a_2, \dots, a_n the sum $\sum_{i=1}^{k-1} a_i$ of all elements preceding it, yielding the prefix sum $0, a_1, a_1 + a_2, \dots, a_1 + a_2 + \dots + a_{n-1}$.

cording to the tessellation pattern implied by the tessellation factors. The resulting vertices are stored sequentially in the patch's slot within the vertex buffer (mapped into global memory). Note that for several kinds of surface primitives, it is advantageous to employ more than one thread per patch for surface evaluation, e.g. one per xyz component ($k = 3$ threads). In particular, loading control points to fast shared memory becomes effective, the register count stays lower (enabling surfaces of rather high degrees) and memory writes are more coherent within a warp. Finally, the index buffer data is written for each patch, thus creating the topology of its tessellation.

The resulting buffers can then be used directly for rendering. Usually, the vertex data features an object id, which allows selecting object-local shading options analogous to instanced rendering. Note that in case of multi-pass rendering, the buffer data can readily be reused without necessitating reevaluation of the patch surfaces.

Since an explicit representation of the tessellation result is available, it is also possible to post-process it before rendering. For instance, assume only dyadic tessellations are created and geomorphing is desired. Then, the surface evaluation may initially be done completely at the finest involved tessellation level. Only in a post-process on the vertex buffer, the vertices to be morphed are adapted. They easily get their coarser-level positions by interpolating between their adjacent vertices, which can readily be accessed, thus avoiding many redundant computations. As another example, recall that even in case control points and sampling parameters are consistent among patches, the numerical results may slightly differ if the involved parameterization directions differ. But with all tessellation vertices at hand, it is possible to use neighborhood information to copy generated vertex position data for boundary curves across adjacent patches in a post-process, ensuring absolute crack-freeness irrespective of numerical inaccuracies.

To provide more insight into the actual realization as well as the flexibility of our framework, the following two subsections describe two concrete examples. These are chosen to often differ significantly in the single steps.

7.6.2 Example: bicubic rational Bézier patches

As first example, we consider bicubic rational tensor-product Bézier patches (cf. Sec. 6.1.1). An overview of our related CudaTess implementation is shown in Fig. 7.18. The application-provided input comprises the control points of all patches in the scene packed contiguously in an array, as well as adjacency data, storing for each boundary curve the abutting patch's id. We first derive a bounding box for each patch's control points and perform view-frustum culling. If a patch doesn't get culled, tessellation factors in u and v direction are computed. Taking neighborhood information into account, we subsequently derive consistent tessellation factors for the four boundary curves of each patch. Based on these factors, patch-wise vertex and index counts are determined, and then buffer offsets are derived. Finally, the actual tessellation is performed by generating vertex and index buffer data.

Regarding the computation of tessellation factors, we employ the deviation-based method detailed in Sec. 7.4.3. For each patch, we therefore first derive the object-space error tolerance ε from a user-specified screen-space error bound and then determine the quantities r , $\min_{0 \leq i, j \leq 3} \{w_{ij}\}$, D_{uu} , D_{uv} and D_{vv} . Recall that the last three of them depend on ε and hence cannot be precomputed. Next, we derive the sampling step sizes δ_u and δ_v and compute the corresponding internal tessellation factors. For each patch, we store one tessellation factor per boundary curve, initialized to the associated internal factor.

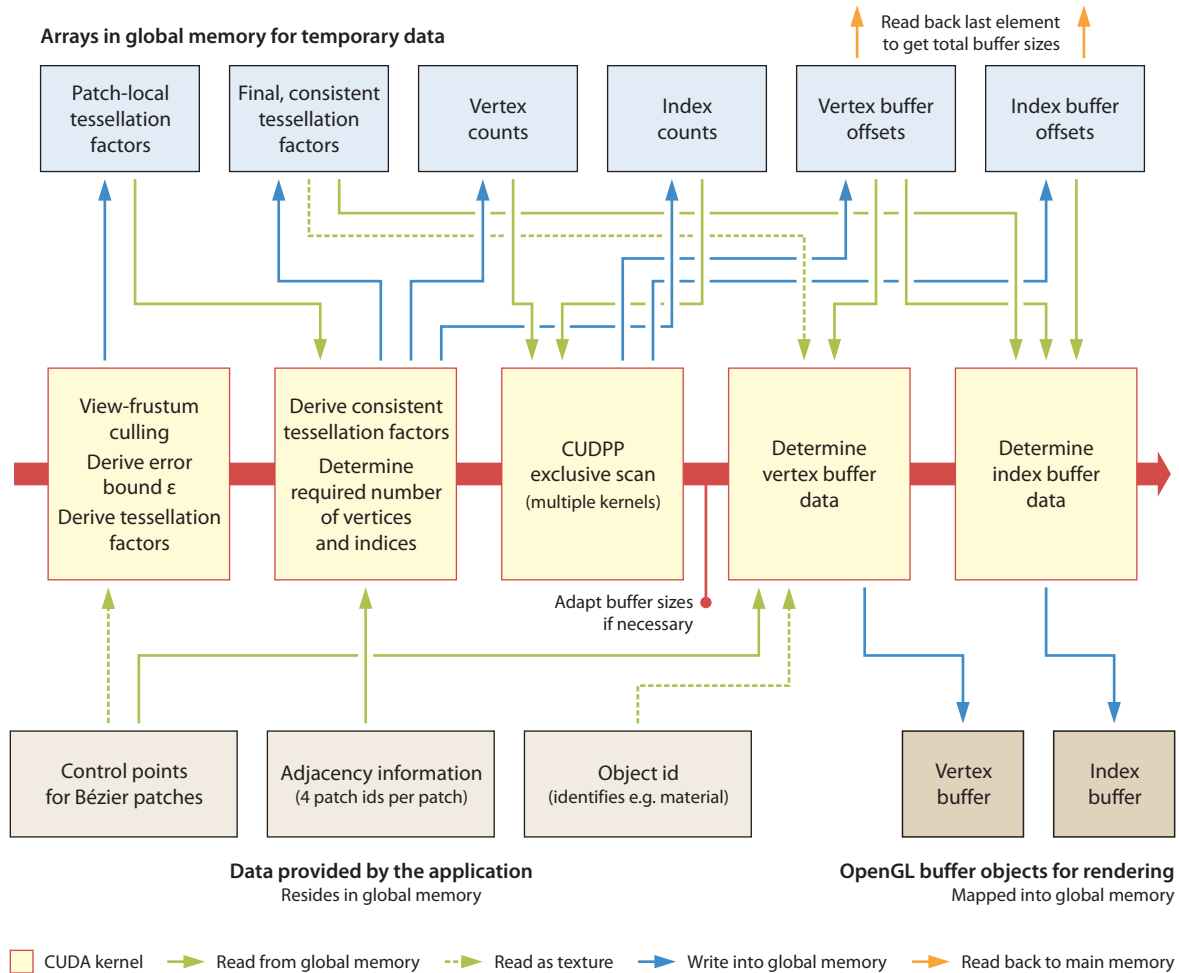


Figure 7.18 Realization of the CudaTess framework for bicubic rational Bézier patches.

To avoid cracks among two adjacent patches, we make the involved factors for the shared boundary consistent in the next step, assigning their maximum value to them. As a consequence, within each patch the minimum of two opposing boundaries' factors provides an upper bound on the corresponding internal factor. Note that if both boundary factors get increased during the adaptation step, this bound is larger than the originally determined internal factor. Picking just the boundary factors' minimum as internal factor may hence result in oversampling. But since, on the other hand, this choice guarantees that at most two transition regions can occur, thus keeping the tessellation pattern simple, we adopt it, nevertheless.

The tessellation created follows our scheme from Sec. 7.3.4. Pursuing the approach described there, the topology is produced in a single thread for each patch, and output as triangle strips separated by a special strip restart index. Note, however, that for simplicity we don't perform the mentioned shape-improving modification for two adjoining transition strips.

The vertex data generation for each patch is distributed across four consecutive threads, one for each component (x, y, z, w). Note that a patch's threads belong to the same warp and hence run in lockstep and can easily communicate via shared memory. First, the control points are collectively loaded into shared memory. Then, vertex data is successively determined according to the tessellation pattern implied by the tessellation factors, and written to the vertex buffer. Each thread first evaluates its component of $\mathbf{b}(u, v)$, $\mathbf{b}_u(u, v)$ and $\mathbf{b}_v(u, v)$. Then, a patch's first

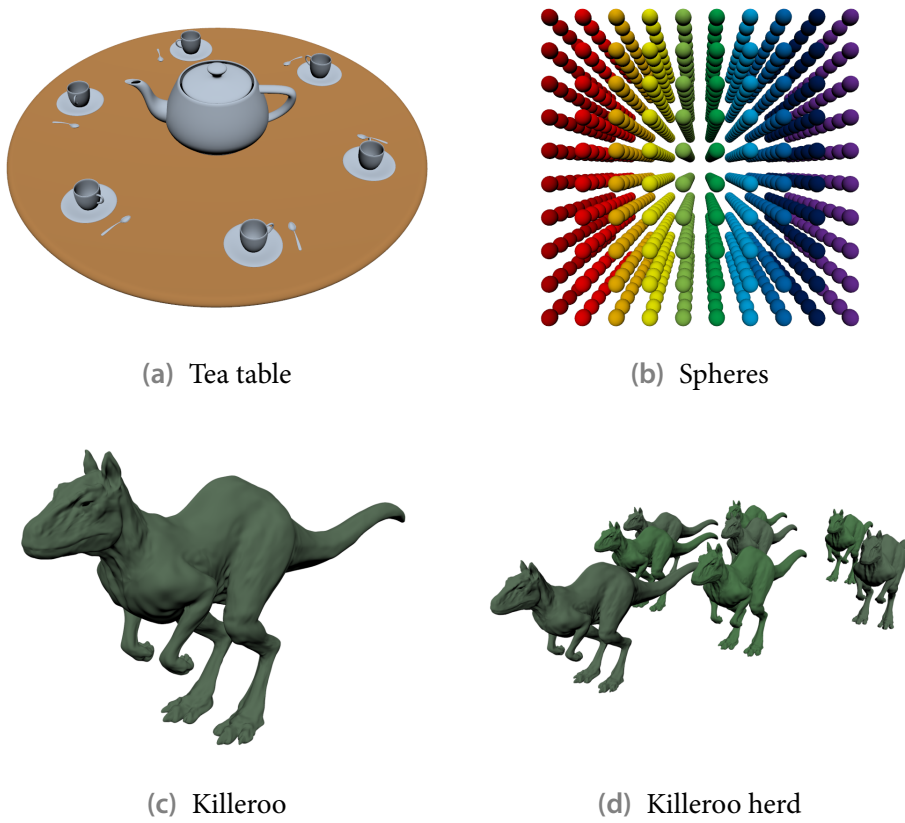


Figure 7.19 Example scenes composed of bicubic rational Bézier patches, adaptively tessellated with CudaTess.

three threads compute the position $\mathbf{p}(u, v)$ and normal $\mathbf{n}(u, v)$, with required quantities like $(\mathbf{b}(u, v))_w$ being exchanged via shared memory. In addition, for each vertex, we emit (u, v) coordinates and an object id obtained from a texture.

Forward differencing

Thanks to performing surface evaluation patch-wise and thus processing a single patch's vertices sequentially instead of in parallel, we are able to employ techniques which reuse results from computations carried out for previous vertices. We exemplarily adopted *forward differencing* [390], which reduces the evaluation of $\mathbf{b}(u, v)$, $\mathbf{b}_u(u, v)$ and $\mathbf{b}_v(u, v)$ to a small number of additions for all vertices with the same v (or u) coordinate but the first.

More generally, when evaluating a cubic function $f(t)$ at evenly spaced sample points $t_i = t_0 + i\Delta t$, the forward difference

$$\Delta^1 f(t) = f(t + \Delta t) - f(t)$$

between two adjacent sample values varies quadratically. Applied iteratively, the difference

$$\Delta^2 f(t) = \Delta^1 f(t + \Delta t) - \Delta^1 f(t)$$

changes linearly, while the difference

$$\Delta^3 f(t) = \Delta^2 f(t + \Delta t) - \Delta^2 f(t)$$

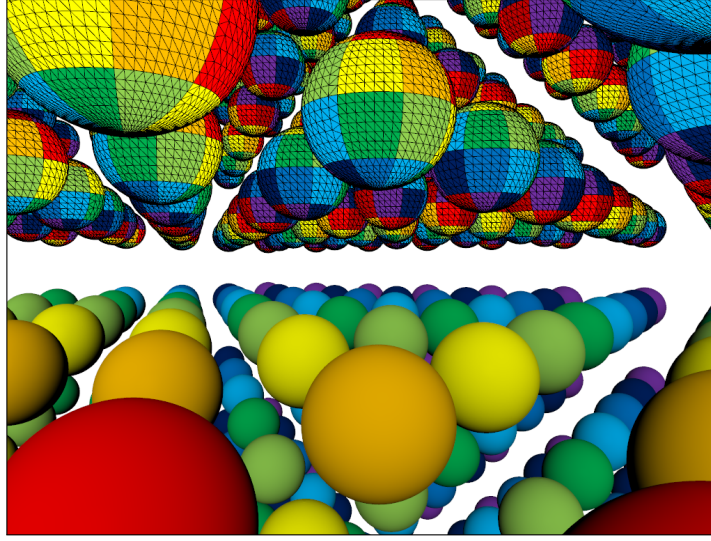


Figure 7.20 Close-up view of the spheres scene from Fig. 7.19 b. The top part reveals the adaptive tessellation, coloring triangles originating from the same patch identically. The resulting visually smooth rendering result is shown in the bottom part.

is constant. Therefore, given values $f(t_k)$, $\Delta^1 f(t_k)$, $\Delta^2 f(t_k)$, $\Delta^3 f(t_k)$ for some sample point t_k , the corresponding values for the successive sample $t_{k+1} = t_k + \Delta t$ can easily and efficiently be evaluated by just three additions:

$$\begin{aligned} f(t_{k+1}) &= f(t_k) + \Delta^1 f(t_k), \\ \Delta^1 f(t_{k+1}) &= \Delta^1 f(t_k) + \Delta^2 f(t_k), \\ \Delta^2 f(t_{k+1}) &= \Delta^2 f(t_k) + \Delta^3 f(t_k), \\ \Delta^3 f(t_{k+1}) &= \Delta^3 f(t_k). \end{aligned}$$

In case of a cubic Bézier curve $\mathbf{b}(t)$ with control points \mathbf{b}_i and the initial sample point $t_0 = 0$, the start-up values are directly calculated as

$$\begin{aligned} \mathbf{b}(t_0) &= \mathbf{b}_0, \\ \Delta^1 \mathbf{b}(t_0) &= \Delta t^3 (\mathbf{b}_3 - 3\mathbf{b}_2 + 3\mathbf{b}_1 - \mathbf{b}_0) + 3\Delta t^2 (\mathbf{b}_2 - 2\mathbf{b}_1 + \mathbf{b}_0) + 3\Delta t (\mathbf{b}_1 - \mathbf{b}_0), \\ \Delta^2 \mathbf{b}(t_0) &= 6\Delta t^3 (\mathbf{b}_3 - 3\mathbf{b}_2 + 3\mathbf{b}_1 - \mathbf{b}_0) + 6\Delta t^2 (\mathbf{b}_2 - 2\mathbf{b}_1 + \mathbf{b}_0), \\ \Delta^3 \mathbf{b}(t_0) &= 6\Delta t^3 (\mathbf{b}_3 - 3\mathbf{b}_2 + 3\mathbf{b}_1 - \mathbf{b}_0), \end{aligned}$$

ideally reusing common expressions.

Note that since computations are performed in finite precision, resulting numerical errors may get accumulated by the involved successive addition. At extremely large tessellation factors, using a single forward differencing sequence to produce the surface samples for a complete isocurve may hence not be accurate enough. However, in such cases, we may easily restart forward differencing every, say, 64 samples, and still achieve a huge saving in arithmetic operations compared to independently evaluating the surface at each sample position.

Results

We applied our implementation to several example scenes; some of those are shown in Figs. 7.19 and 7.20. The related tessellations produced at a viewport of 1024×768 and for a prescribed

| Scene | Fig. | Patches | Triangles | Vertices | Indices |
|--------------------|--------|---------|-----------|----------|---------|
| Tea table | 7.19 a | 332 | 41,372 | 25,798 | 55,316 |
| Killeroo | 7.19 c | 11,532 | 100,930 | 110,515 | 213,709 |
| Killeroo herd | 7.19 d | 92,256 | 345,751 | 514,753 | 827,389 |
| Spheres | 7.19 b | 32,000 | 402,000 | 393,600 | 675,600 |
| Spheres (close-up) | 7.20 | 32,000 | 272,301 | 216,925 | 391,401 |

Table 7.4 Statistics for the rational Bézier patch example scenes, showing the geometric complexity of the adaptive tessellations created with CudaTess.

| Scene | Tea table | Killeroo | Killeroo herd | Spheres | Spheres (close-up) |
|------------------------------------|-----------|----------|---------------|---------|--------------------|
| Adaptive tessellation | 207 Hz | 156 Hz | 48 Hz | 98 Hz | 93 Hz |
| ~ with forward differencing | 261 Hz | 194 Hz | 63 Hz | 109 Hz | 123 Hz |
| Only shading (reusing buffer data) | 758 Hz | 706 Hz | 366 Hz | 362 Hz | 320 Hz |
| Tessellation factors | 0.39 ms | 0.82 ms | 4.18 ms | 1.62 ms | 1.04 ms |
| Final factors & buffer offsets | 0.09 ms | 0.16 ms | 0.42 ms | 0.20 ms | 0.19 ms |
| Vertex buffer data update | 2.40 ms | 3.32 ms | 12.22 ms | 4.54 ms | 5.54 ms |
| ~ with forward differencing | 1.40 ms | 2.04 ms | 7.38 ms | 3.54 ms | 2.92 ms |
| Index buffer data update | 0.56 ms | 0.61 ms | 1.19 ms | 0.96 ms | 0.77 ms |

Table 7.5 Rendering performance and tessellation time break-down for the adaptive tessellations created with CudaTess for the rational Bézier patch example scenes.

screen-space error bound of 0.5 pixels are quantified in Table 7.4. Note that in case of the close-up view of the spheres scene (Fig. 7.20), only those patches get actually tessellated which pass the view-frustum test.

Corresponding performance data obtained on an Intel Pentium IV 3 GHz with an NVIDIA GeForce 8800 GTS (G80) graphics card is listed in Table 7.5. Even for large numbers of patches to tessellate, we attain real-time frame rates. Recall that the adaptive tessellation is generated from scratch each frame, requiring only the patches' control points and their neighborhood relationships as input. Consequently, both the view point and the patch control points can be freely animated without negatively affecting the tessellation performance. Since CudaTess outputs a vertex buffer and an index buffer, the data can be reused for multi-pass rendering without necessitating any recomputations, enabling even higher frame rates. Note that we employ the Oren-Nayar reflectance model [279] for shading.

As the time break-down shows, generating the vertex data and hence evaluating the surface is the most dominating part. Even if all tessellation factors are low, like in the spheres scene, this costly computation can be sped up significantly via forward differencing, underlining the optimizational potential and advantage of a patch-parallel approach. In scenes with a large number of visible patches, like the Killeroo herd scene, determining tessellation factors also consumes a significant share of time, mainly because of having to compute the bounds D_{uu} , D_{uv} and D_{vv} . It is hence beneficial that this time-demanding stage is executed well-parallelized on the GPU by CudaTess.

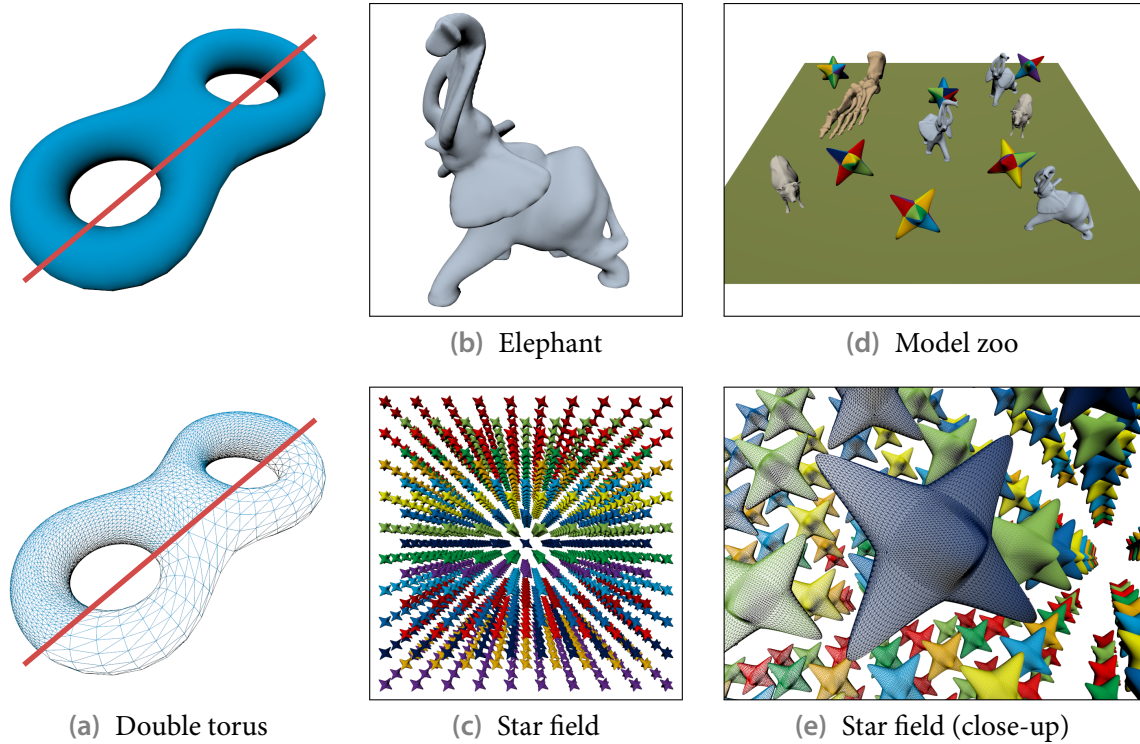


Figure 7.21 Example scenes described by base triangle meshes, adaptively refined according to the PN triangle scheme with CudaTess.

7.6.3 Example: PN triangles

For our second example, we applied the CudaTess framework to PN triangles (see Sec. 6.1.3). A collection of coarse base triangle meshes as well as according neighborhood data for the faces is provided as input. For each corresponding PN triangle, we first derive the control points \mathbf{b}_{ijk} as well as their bounding box and test it against the view frustum. If the patch is potentially visible, we further determine its normal field control points \mathbf{n}_{ijk} and store them along with the \mathbf{b}_{ijk} for the vertex data generation stage, before finally computing the three boundary tessellation factors. Next, we utilize adjacency information for the input triangles to make the factors consistent across patches. After determining the number of vertices and indices required for tessellating each PN triangle, buffer offsets are computed. In a last step, the actual vertex and index buffer data is generated, yielding the final tessellation.

To obtain appropriate tessellation factors guaranteeing a prescribed maximum geometric approximation error, we pursue the approach detailed in Sec. 7.4.4. For each PN triangle we first derive the bounds $D_{p,q}$ on its second-order directional derivatives required for determining the sampling step sizes. In principle, they can be precomputed if the coarse base triangles are only subjected to rigid transformations. Subsequently, we calculate the object-space error threshold ε corresponding to a user-specified screen-space error bound, and derive the three step sizes δ_1 , δ_2 and δ_3 for the parameter directions \mathbf{d}_{01} , \mathbf{d}_{12} and \mathbf{d}_{20} such that the error tolerance is satisfied. From these the related boundary tessellation factors are determined and stored. Similar to the rational Bézier patch realization, we obtain consistent tessellation factors among neighboring PN triangles in the next framework stage by setting related factors of two abutting patches to their maximum.

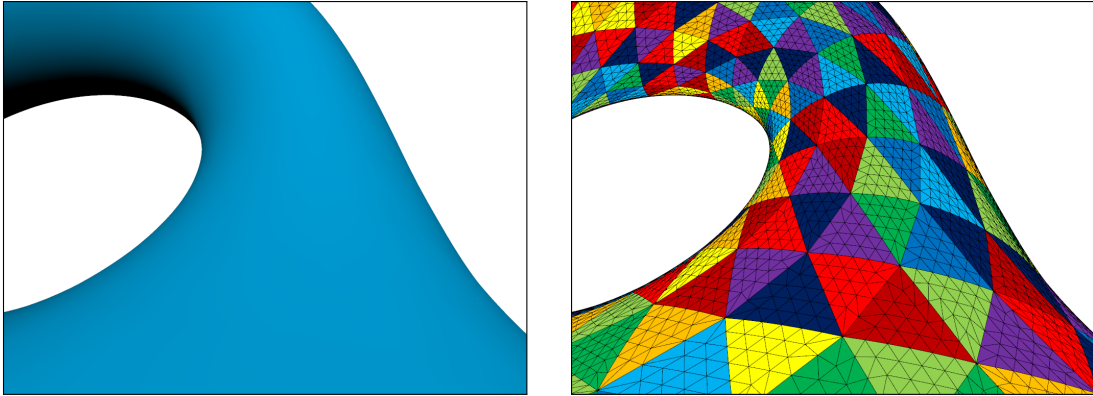


Figure 7.22 A close-up view of the double torus from Fig. 7.21 a.

A PN triangle's tessellation adheres to our pattern scheme from Sec. 7.3.4 and is generated according to the method detailed there. We distribute the vertex data generation for each patch across three consecutive threads, one for each component. At first, the control points corresponding to the vertex positions and normals of the underlying coarse triangle are collectively loaded to shared memory. If all tessellation factors equal one, the coarse triangle is not refined and we just output the corresponding vertex data. Otherwise, the remaining control points are brought into shared memory and vertex data is successively computed according to the tessellation pattern and written to the mapped vertex buffer. In addition to evaluating the position $\mathbf{b}(u, v)$ and the normal $\mathbf{n}(u, v)$, we also output (u, v) coordinates and an object id fetched from a texture.

Concerning thread assignment, we skip every 16th thread and leave it idle, such that the three threads of a patch all belong to the same warp. Despite this underutilization, a higher performance is typically achieved compared to other options like choosing the block size to be a multiple of three and especially to using just one thread per patch.

Results

Some example scenes on which we tested our realization are depicted in Figs. 7.21 and 7.22. They cover a wide range of both coarse triangle counts and generated tessellation triangle counts, as shown in Table 7.6. Note that we employed the same configuration as for the rational Bézier patch results.

The performance figures listed in Table 7.7 document that even large tessellation loads can be coped with in real time. Since both the PN triangles' control points as well as the adaptive tessellation are derived anew each frame, animations are freely supported. Further recall that the tessellation may be reused, for instance in multi-pass rendering, achieving significantly higher frame rates.

Like in the rational Bézier patch case, computing the vertex data proves to be the most time-consuming part. The kernel invoked for deriving tessellation factors, which involves computing PN triangle control points, bounding box determination, culling, and calculation of the bounds on the second-order directional derivatives, also takes a considerable amount of time. However, it can be stream-lined in case of static scenes by precomputing the control points, their bounding box and the derivative bounds. Especially for complex scenes like the model zoo, this optimization yields a measurable speed-up.

| Scene | Fig. | Base triangles | Triangles | Vertices | Indices |
|-----------------------|--------|----------------|-----------|----------|-----------|
| Double torus | 7.21 a | 1,536 | 15,320 | 16,048 | 43,328 |
| Elephant | 7.21 b | 21,540 | 79,208 | 116,399 | 257,894 |
| Star field | 7.21 c | 41,496 | 1,050,688 | 864,928 | 2,547,040 |
| Star field (close-up) | 7.21 e | 41,496 | 379,982 | 249,385 | 692,306 |
| Model zoo | 7.21 d | 80,578 | 160,200 | 311,469 | 548,730 |

Table 7.6 Statistics for the PN triangle example scenes, showing the geometric complexity of the adaptive tessellations created with CudaTess.

| Scene | Double torus | Elephant | Star field | Star field (close-up) | Model zoo |
|------------------------------------|--------------|----------|------------|-----------------------|-----------|
| Adaptive tessellation | 408 Hz | 230 Hz | 60 Hz | 118 Hz | 109 Hz |
| ~ using precomputed data | 409 Hz | 259 Hz | 64 Hz | 122 Hz | 135 Hz |
| Only shading (reusing buffer data) | 1,413 Hz | 1,076 Hz | 186 Hz | 301 Hz | 524 Hz |
| Tessellation factors | 0.39 ms | 0.91 ms | 1.41 ms | 0.75 ms | 2.42 ms |
| ~ using precomputed data | 0.35 ms | 0.43 ms | 0.47 ms | 0.45 ms | 0.65 ms |
| Final factors & buffer offsets | 0.21 ms | 0.26 ms | 0.33 ms | 0.30 ms | 0.50 ms |
| Vertex buffer data update | 0.60 ms | 1.49 ms | 6.81 ms | 2.84 ms | 3.33 ms |
| Index buffer data update | 0.48 ms | 0.68 ms | 2.57 ms | 1.11 ms | 0.93 ms |

Table 7.7 Rendering performance and tessellation time break-down for the adaptive tessellations created with CudaTess for the PN triangle example scenes.

7.6.4 Discussion

Our CudaTess framework not only constitutes a novel approach for efficiently performing adaptive tessellation, but, to the best of our knowledge, is also the first generic method which performs all major steps completely on the GPU without relying on dedicated hardware support. In particular, we offer a CUDA-based solution for the efficient and purely GPU-based dynamic generation of potentially non-uniform geometry that requires no CPU assistance except invoking a few kernels. Compared to the alternative of utilizing geometry shaders, which don't support outputting indexed primitives and hence vertex reuse and which also impose an upper limit on the number of output primitives, our approach to GPU-guided geometry generation provides significantly more flexibility while at the same time typically being considerably faster. Therefore, we reckon that our method is of more general interest than just for tessellation purposes.

On the other hand, our approach suffers from some limitations. While adopting a patch as unit of parallelism is crucial for efficient GPU-guided generation of varying amounts of geometry and enables acceleration techniques like forward differencing, it may prevent utmost utilization of the GPU's processors. First, since each patch is processed by one single thread (or a small number of threads), the overall number of patches must be reasonably high to not leave processors completely idle. Nevertheless, as the tea table scene demonstrates (cf. Tables 7.4 and 7.5), even smaller patch counts with high tessellation rates are handled very well. Second, threads within a warp may diverge and finish at different times if the tessellation patterns of

the warp's patches differ, which impacts the effective parallelism. However, even when manually imposing a kind of worst-case workload, we only observed a reasonably low reduction in throughput compared to a best case for SIMD parallelism.

Since CudaTess requires the tessellation to be stored in a vertex and an index buffer, it may consume a noticeable amount of memory. Especially in case of a large number of patches being excessively tessellated, one hence may consider applying CudaTess sequentially to subsets of the scene. On the other hand, it is the explicit availability of the tessellation result that enables post-processing of the vertex data as well as fast buffer reuse for multi-pass rendering.

Compared to (re)using previously computed patterns (like when rendering refinement patterns), CudaTess exhibits a creation overhead because we generate a tessellation pattern for each patch on the fly. However, such a dynamic creation naturally supports essentially arbitrary tessellation factor configurations and thus prevents having to put restrictions on them, otherwise necessitated to enable precomputation and to cope with combinatorial explosion. As a consequence, potentially fewer samples may be processed, since, for instance, forcing each factor to the next larger dyadic value is not necessary. Furthermore, computing the surface vertices making up the tessellation is typically faster when interleaving sample generation and surface evaluation. Especially techniques made possible by our patch-parallel approach, like caching control points in shared memory or forward differencing, can make a significant impact on performance. Therefore, despite creating a custom tessellation pattern for each patch, CudaTess is usually faster in total, nevertheless, if the surface evaluation is rather expensive. See the next subsection for a concrete comparison with rendering refinement patterns.

Note that it is possible to employ the CudaTess framework for creating a vertex and an associated index buffer containing a refinement pattern instance for each patch, render all these patterns with a single draw call, and perform the surface evaluation only in the vertex shader. However, unlike the normal CudaTess procedure, such a hybrid approach is not able to amortize the pattern creation overhead by a more efficient surface evaluation. By contrast, because the per-sample workload within CudaTess reduces to progressing the current parametric position and hence only a low arithmetic intensity is available to hide the latency of the involved non-coalesced memory writes, the pattern generation overhead is even accentuated.

It is interesting to observe that the upcoming Direct3D 11's tessellation support bears some resemblance with the CudaTess pipeline and also offers some of its acceleration capabilities. For instance, Direct3D 11 organizes determining tessellation factors, creating an according tessellation pattern, and evaluating the surface as successive steps within a single rendering pass. However, unlike in CudaTess, no stage for making factors consistent across patches is provided. It hence becomes necessary to derive separate tessellation factors for the boundaries and the patch interior in the first place. But recall from Sec. 7.4 that this may not guarantee a sufficient sampling close to a boundary whose factor is smaller than the related internal factor. Also note that while customizing stages as well as adding further stages is straightforward in CudaTess, Direct3D 11's according flexibility is rather limited because it fixes the single steps and their properties to enable and facilitate direct hardware implementation. As another characteristic, Direct3D 11 provides the patch control points as input both for the tessellation factor determination and for the vertex-parallel surface evaluation in the domain shader. Like in CudaTess, control points hence don't have to be fetched anew for each surface sample. On the other hand, further patch-scale optimizations like forward differencing are not possible because each surface point is processed independently. Finally, note that it is unclear how efficient the first hardware realizations of the complete Direct3D 11 pipeline with enabled tessellation support will be.

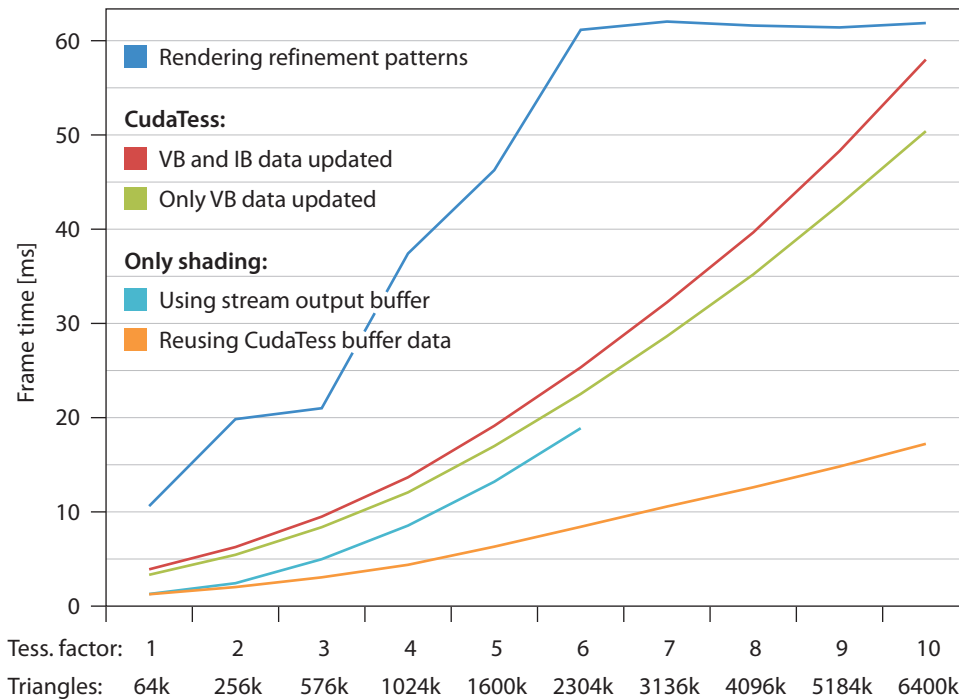


Figure 7.23 Rendering performance comparison between (batched) rendering of refinement patterns and CudaTess for the spheres scene (Fig. 7.19 b). All patches are uniformly tessellated according to the selected global tessellation factor.

Even if they met all expectations right away, we reckon that the flexibility of the CudaTess framework along with the opportunities arising from the pursued patch-parallel approach make our method still relevant and competitive. In particular, adapting CudaTess to the compute shader technology of Direct3D 11 enables a seamless integration into the standard graphics pipeline. We further believe that our approach is well suited for future hardware architectures like Intel’s Larrabee [349] with their increased flexibility and programmability as well as their tendency to omit special-purpose units like a rasterizer or a tessellator and realize their functionality in software.

7.6.5 Comparison to rendering refinement patterns

We close this chapter as well as the part on (real-time) rendering of curved surfaces with a comparison between the two currently most promising approaches, our patch-parallel CudaTess framework and the rendering of refinement pattern. All results were obtained with the same configuration as before in this section (Pentium IV 3 GHz, GeForce 8800 GTS (G80), 1024×768 viewport, 0.5 pixels error tolerance).

As first scenario, we pick the spheres scene and select identical tessellation factors for all patches, varying the global factor to cover a wide range of uniform tessellations. When rendering refinement patterns, we employ instancing to limit API overhead to just one draw call, noting that batch building is trivial because only a single refinement pattern is utilized. Similarly, our CudaTess implementation simplifies to generating vertex and index buffer data, as tessellation factors are prescribed and all buffer slots are of equal size. The performance data in Fig. 7.23 shows that CudaTess is always faster than rendering refinement patterns. This can be attributed to the higher efficiency of executing surface evaluation in a patch-parallel way

and especially the involved caching of control points in shared memory. Note that a further speed-up is possible if forward differencing is employed.

To enable an even more direct comparison of the evaluation performance and also reveal the overhead in CudaTess for creating index data for the tessellation topology, we further include frame times for a CudaTess version where a precomputed index buffer is used and only the vertex data is regenerated every frame. Moreover, we considered reusing a tessellation for the next passes. While employing the compact indexed buffer data readily available in CudaTess results in significant speed-ups, the alternative of capturing the rendered refinement patterns in a stream output buffer and rendering it in subsequent passes doesn't scale that well. Even worse, because only a triangle soup is recorded, causing valence- n vertices to be stored n times, a huge stream output buffer is required which quickly exhausts the available memory.

For assessing adaptive tessellation performance, we applied the rendering of refinement patterns to all our CudaTess example scenes. In a preparation step, we determine the tessellation factors and generate just the actually required refinement patterns. We consider the cases of supporting all tessellation factor configurations, of restricting the factors to dyadic values, and of adapting uniform dyadic tessellations (cf. Sec. 7.5.4). Furthermore, we distinguish between immediate rendering, issuing one draw call per (visible) patch, and batched rendering via instancing, invoking one draw call per pattern. Note that we build the batches and populate a per-instance data buffer with the appropriately arranged patch control points during the preparation step. Table 7.8 lists the measured performance data.

Immediate rendering of refinement patterns turns out to be significantly slower than CudaTess. The only exception is the tea table scene with its small number of patches and hence limited potential for high GPU utilization with a patch-parallel approach like CudaTess. On the other hand, the encountered high tessellation rates make this scene an ideal case for the immediate refinement pattern technique, effectively amortizing the API invocation overhead.

When rendering refinement patterns with instancing, the performance improves significantly for more complex scenes, underlining the importance of building larger batches. Nevertheless, our CudaTess implementation is still clearly faster for all rational Bézier patch examples but the simple tea table scene, especially when utilizing forward differencing.

In case of PN triangles, however, the situation is less obvious. If refinement patterns for arbitrary combinations of integer tessellation factors are employed, CudaTess appears to be often slower. One main reason for this difference to the rational Bézier patch setting is that the evaluation of PN triangles is cheaper than that of Bézier patches, limiting the overall impact of accelerating this step in CudaTess. On the other hand, we perform batch building in the preparation phase for our static viewpoints, while in practice this may reasonably be expected to consume some amount of runtime, thus reducing the achievable frame rate. Moreover, recall that we only created the refinement patterns actually used for one specific viewpoint, circumventing the issue of excessive storage requirements. But if the hence more realistic option of rendering dyadic refinement patterns is pursued, the performance is typically reduced because significantly more surface samples must be processed. By contrast, CudaTess can efficiently deal with any tessellation factor configuration, since it constructs tessellation patterns on the fly.

It is interesting to note that these performance characteristics indicate that—at least in cases where it is rather expensive to evaluate a surface at a given sample position—CudaTess is competitive even against approaches where the refinement patterns are not explicitly rendered but emitted by a hardware tessellation unit. Moreover, CudaTess may have to evaluate fewer sample points, since a dedicated tessellator may output a pattern which features more vertices than our pattern scheme for equivalent tessellation factors, as is the case with Direct3D 11 (cf. Sec. 7.3.2).

| Scene | Refinement patterns, integer tessellation | | | | Refinement patterns, dyadic tessellation | | | |
|-----------------------|--|-----------|--------|---------|---|-----------|--------|---------|
| | #P | #T | Immed. | Batched | #P | #T | Immed. | Batched |
| Tea table | 81 | 41,372 | 595 Hz | 365 Hz | 26 | 80,808 | 515 Hz | 262 Hz |
| Killeroo | 179 | 100,930 | 44 Hz | 146 Hz | 70 | 125,396 | 44 Hz | 133 Hz |
| Killeroo herd | 109 | 345,751 | 5.6 Hz | 32 Hz | 74 | 357,110 | 5.6 Hz | 31 Hz |
| Spheres | 7 | 402,000 | 16 Hz | 49 Hz | 7 | 586,400 | 16 Hz | 38 Hz |
| Spheres (close-up) | 54 | 272,301 | 49 Hz | 86 Hz | 13 | 406,200 | 49 Hz | 63 Hz |
| Double torus | 23 | 15,320 | 307 Hz | 979 Hz | 11 | 23,660 | 307 Hz | 977 Hz |
| Elephant | 65 | 79,206 | 25 Hz | 303 Hz | 31 | 102,542 | 25 Hz | 305 Hz |
| Star field | 4 | 1,050,688 | 13 Hz | 110 Hz | 3 | 2,410,336 | 13 Hz | 60 Hz |
| Star field (close-up) | 57 | 379,982 | 102 Hz | 180 Hz | 7 | 777,472 | 100 Hz | 124 Hz |
| Model zoo | 74 | 160,200 | 6.6 Hz | 86 Hz | 35 | 198,540 | 6.6 Hz | 86 Hz |

| Scene | Refinement patterns, adapted uniform dyadic tessellation | | | | CudaTess | | | |
|-----------------------|---|-----------|--------|---------|----------|-----------|-------------------|------------------|
| | #P | #T | Immed. | Batched | #P | #T | Adaptive tess. | Forward diff. |
| Tea table | 5 | 237,664 | 303 Hz | 51 Hz | 81 | 41,372 | 213 Hz | 271 Hz |
| Killeroo | 4 | 226,518 | 41 Hz | 41 Hz | 179 | 100,930 | 173 Hz | 225 Hz |
| Killeroo herd | 4 | 568,326 | 5.1 Hz | 10 Hz | 109 | 345,751 | 60 Hz | 84 Hz |
| Spheres | 2 | 774,400 | 15 Hz | 14 Hz | 7 | 402,000 | 111 Hz | 124 Hz |
| Spheres (close-up) | 3 | 457,472 | 46 Hz | 26 Hz | 54 | 272,301 | 102 Hz | 139 Hz |
| Double torus | 3 | 25,104 | 285 Hz | 979 Hz | 23 | 15,320 | 431 Hz | |
| Elephant | 4 | 119,226 | 23 Hz | 310 Hz | 65 | 79,206 | 286 Hz | |
| Star field | 2 | 2,540,544 | 12 Hz | 55 Hz | 4 | 1,050,688 | 70 Hz | |
| Star field (close-up) | 3 | 792,320 | 92 Hz | 118 Hz | 57 | 379,982 | 131 Hz | |
| Model zoo | 5 | 224,263 | 6.1 Hz | 86 Hz | 74 | 160,200 | 156 Hz | |

Table 7.8 Performance comparison between various approaches for rendering refinement patterns and CudaTess. The determination of tessellation factors as well as preparations for instanced rendering are not included in the timings. (#P: number of different patterns used/created; #T: number of rendered triangles; Immed.: one draw call per patch; Batched: one draw call per pattern)

Finally, it seems necessary to point out that the reported performance data was obtained with driver version ForceWare 175.19 under Windows XP, because during our tests we observed that the employed driver version often measurably impacts performance. For instance, compared to the listed frame rates, a different driver resulted in a 25% performance gain for rendering the Killeroo herd scene with CudaTess. Another driver showed only small performance effects of using forward differencing. Moreover, the instanced rendering of refinement patterns was up to four times slower with a CUDA-specific driver than the figures reported. On an NVIDIA GeForce GTX 280, we even encountered the situation that rendering refinement patterns with surface evaluation in the vertex shader is faster than rendering a precomputed mesh of the corresponding adaptive tessellation. Consequently, any method like CudaTess which generates an explicit representation of the whole tessellation is doomed with such a driver behavior.

PART III

Perception-aware rendering

CHAPTER 8

Fundamentals of human visual perception

Real-time rendering of scenes produces a sequence of frames, which are output on a display device and are then ultimately viewed by a user. Consequently, algorithms should ideally be aware of human visual perception and its limitations, and only spend additional effort on raising quality if this improvement can actually be perceived. Although this goal is elusive, not least because our understanding of visual perception is far from complete and large variations among humans exist, taking some perceptual considerations into account during rendering is quite possible and worthwhile. It helps reducing excessive quality improvements that eventually remain imperceptible but also spending a limited time or resource budget wisely, i.e. such that roughly the best perceived quality possible under the imposed constraints is achieved. For instance, the geometric level-of-detail selection may be guided by estimates of the actual discriminability or of the perceivable difference between two LODs instead of just their geometric deviation.

In this chapter, we briefly review some fundamentals of human visual perception. After covering perception and its measurement within psychophysics in general, we discuss relevant characteristics of the human visual system. Subsequently, color and its representation in color spaces that are perceptually roughly uniform as well as color appearance are discussed. Finally, attention, which causes normally perceptible changes like quality degradations to go unnoticed, is dealt with.

Building on this background information, the next chapter then describes how some core elements of human visual perception can be modeled and how such perceptually motivated components can be applied to rendering. In particular, we present an approach which incorporates on-the-fly evaluation of a perceptually based metric within the real-time context.

The part is concluded by Chapter 10 on visual popping. There, we introduce a perceptually motivated predictor which estimates whether popping due to a LOD change is perceptible and as how severe this temporal artifact is likely perceived.

8.1 Human perception and psychophysics

Human perception is the complex process of acquiring sensory information about physical stimuli of the environment via sense organs like the eyes and deriving an interpretation from it. This translation typically involves selecting relevant sensory stimuli and factoring in experiences, and may comprise determining higher-level features. For instance, in vision, the scene is analyzed and distinct objects but also shadows are detected. Note that different modalities like vision and auditory perception are not processed completely independently from each other

but usually interact to some degree. Related cross-modal effects may, for example, improve the detection of a visual target [389]. As an aside, even a whole multi-partner research project funded by the European Union (CROSSMOD), in which we participated, was dedicated to leveraging such audio-visual cross-modal interactions for improving rendering quality and efficiency.

In order to harness and account for perception, it is important to investigate relations between physical stimuli and the induced subjective percepts. Such studies are the concern of psychophysics, a branch of psychology which aims at objectively measuring these relationships. A good overview of related experimental methods, targeted specifically to computer graphics researchers, is provided by Ferwerda [125].

One central concept in psychophysics is that of threshold. An *absolute* or *detection threshold* specifies the level of stimulus intensity at which the presence of a stimulus becomes detectable. On the other hand, a *difference* or *discrimination threshold* quantifies the minimum intensity difference between two stimuli that is required to be able to discriminate between them, i.e. to detect that they are different. This change ΔI in intensity I necessary for discriminability is referred to as *just noticeable difference* (JND). According to *Weber's law*, which is actually only an often appropriate approximate rule, the ratio $\Delta I/I = k$ is a constant.¹

Note that a threshold is not an exact quantity and that at the according intensity level no hard transition in detectability occurs. Instead, a threshold corresponds to the intensity at which a stimulus or change in stimulus, respectively, was detected by subjects with a certain probability p in a psychophysical experiment. Typically, a value of $p = 75\%$ is selected for establishing 1 JND in case a 2AFC method² is employed. Finally, note that the inverse of a threshold is commonly called *sensitivity*.

8.2 Human visual system

The human visual system (HVS) is responsible for visual perception and hence deserves further study. In the following, we briefly review some of its characteristics, paying special regard to aspects relevant for computer graphics. More details can be found, for instance, in Wandell's book [393] or Ferwerda's tutorial [123].

Visual stimuli are acquired by the eyes. Light enters through the pupil and is focused by the cornea and the lens onto the retina, where an image is formed (cf. Fig. 8.1). The cornea features a large refractive power, which is however basically fixed, and hence it is the responsibility of the lens to finely adjust the focal length via the process of accommodation. The pupil serves as aperture, with its size being controlled by the iris.

The retina comprises several layers of neurons, including two types of photoreceptors, *rods* and *cones*, which are responsible for capturing incident light. While rods are extremely sensitive to light and hence offer vision at low light levels, cones are active at higher levels, where rods eventually become saturated. Since all rods essentially feature the same spectral responsivity, they can only convey achromatic information. By contrast, three different kinds of cones, termed L, M, and S cones, exist, which have their peak sensitivities in the long (red), medium (green) and short (blue) wavelength range, respectively, of the visible spectrum (from about

¹It was postulated by Ernst Heinrich Weber in 1834 based on experiments with lifting weights.

²Short for two-alternative forced-choice. Here, two stimuli are presented, and the subject is asked whose intensity is larger. As the name indicates, the subject is forced to make a choice. Consequently, in case no difference is perceived, the subject can only guess, resulting in a selection probability of $p = 50\%$.

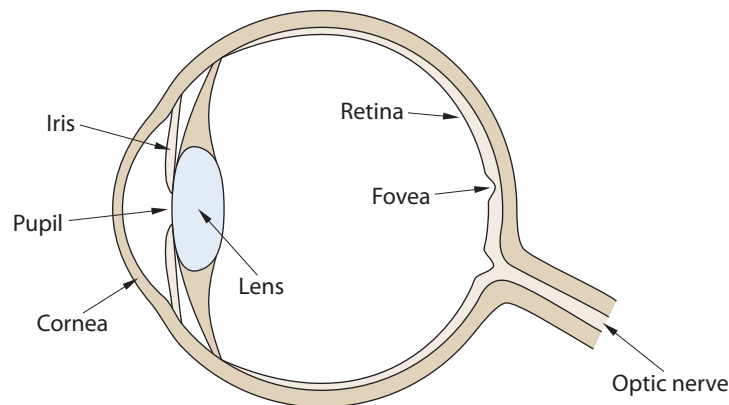


Figure 8.1 Schematic view of the human eye.

380 to 750 nm). Providing an RGB-like trichromatic encoding, they are hence able to capture color (see also Sec. 8.3). In the fovea, subtending about two degrees of visual angle near the center of the retina, spatial resolution is highest, with approximately 120 cones being contained per degree of visual field. Acuity quickly drops substantially with increasing distance from the fovea, and rods, which are completely missing in the central fovea, become dominating in the retina's peripheral areas. Note that apart from photoreceptor density, the effective optical resolution of the HVS is also affected by chromatic aberration resulting from the inability to focus all wavelengths simultaneously. This is directly reflected in the relative number of L, M and S cones (about 12 : 6 : 1), where the sparseness of S cones accounts for the strong defocus of the shorter wavelengths.

The rods and cones connect via bipolar cells to ganglion cells. Each such cell responds to input originating from a certain region in the visual field, called its *receptive field*. It features an antagonistic center-surround organization, where the cell is excited by light hitting the center region of the field and inhibited by light falling on its concentric surround (on-center/off-surround), or vice versa (off-center/on-surround). This effectively corresponds to deriving information about local changes, i.e. computing contrast.

The axons of the retinal ganglion cells form the optic nerve, which goes to the two lateral geniculate nuclei (LGN), where the LGN in one hemisphere (left/right) of the brain processes information from the two eyes for the opposite half of the visual field (right/left). The LGN projects to the primary visual cortex in the occipital lobe. Here, cells exist which are selective to features like spatial frequency, orientation, texture or motion direction. Perceptual processing then continues in further cortical areas.

The HVS developed several techniques to optimize the usage of visual pathways. For instance, recall that color is initially encoded by the responses of the three different cone types. Because their spectral responsivities overlap significantly, however, the resulting signals are later decorrelated to improve representational efficiency. To this end, they are transformed to an achromatic response and two chromatic, opponent-color signals, spanning bipolar axes from red to green and yellow to blue, respectively.

Another example is visual *adaptation* to the prevalent environmental conditions. It serves to support vision over a wide range of illumination levels (covering more than ten orders of magnitude). At any specific level to which the HVS is adapted, however, discrimination ability is limited to a significantly smaller range (of up to three orders of magnitude). The adaptation mechanism involves changing the pupil size and adjusting cell response sensitivities, for

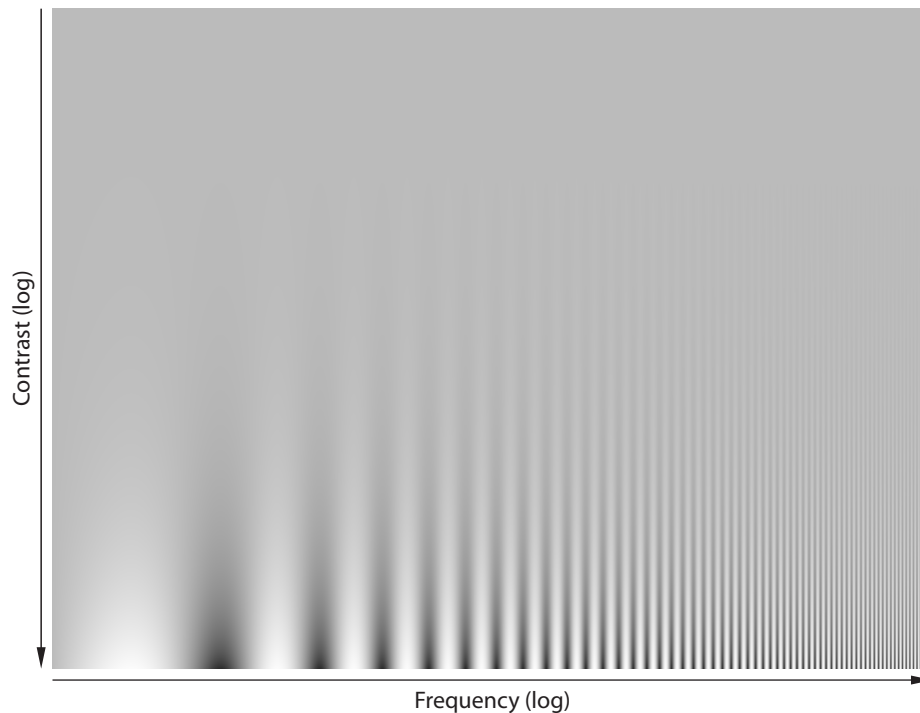


Figure 8.2 Campbell-Robson chart [61, 291]. It nicely demonstrates that contrast detection threshold (the lowest contrast along vertical axis direction that can still be recognized) is a function of spatial frequency (i.e. varies in horizontal axis direction).

instance by shifting the limited operating range of retinal cells. Consequently, the detection threshold for a visual stimulus depends on the surrounding illumination. This non-linear relationship can be described by a *threshold-versus-intensity* (TVI) function. It is typically determined by briefly showing a disk of luminance³ $L + \Delta L$ on a uniform background of luminance L to which the subject is completely adapted.

In addition to coping with varying light levels, the HVS can also adapt to changes in the spectral power distribution of the illumination. This so-called *chromatic adaptation* is partially realized by adjusting the sensitivities of the three cone types independently.

The previously mentioned availability of cells which are sensitive to certain stimulus characteristics suggests that visual stimuli are actually processed in multiple channels by according visual mechanisms. These are tuned, for instance, to bands of spatial frequency and of orientation, with the orientation bandwidths being larger at lower frequencies as well as for chromatic information. Note that the frequency decomposition, which probably originates in the varying sizes of the ganglion cells' receptive fields, may give rise to a multi-resolution representation of the retinal image, where lower-frequency bands are encoded at a lower effective resolution. Apart from spatial mechanisms selective in frequency and orientation, the HVS further features temporal mechanisms. These are often assumed to just comprise a low-pass and a band-pass mechanism, called sustained and transient channel, respectively [132, 198].

Partially caused by the tuning of cells to frequency bands, the ability to detect a stimulus depends on its spatial frequency content (see Fig. 8.2). It is typically quantified by a *contrast*

³Luminance is the photometric equivalent to radiance. In photometry, the wavelength-dependent sensitivity of the HVS is accounted for by weighting the according spectral radiometric quantity with the so-called luminous efficiency function, derived for a standard observer.

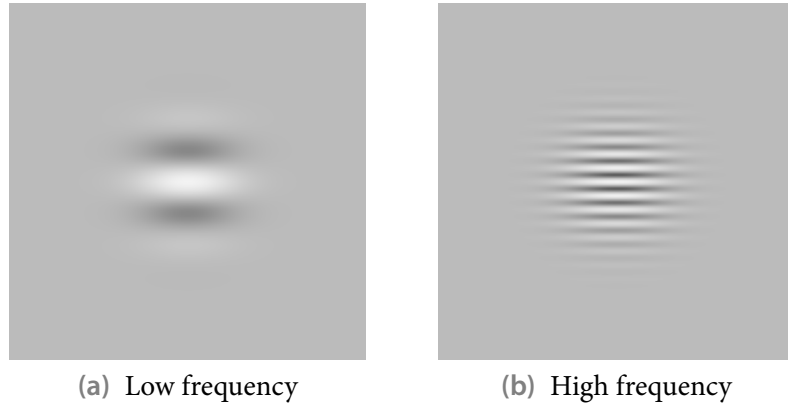


Figure 8.3 Two example Gabor patches (of identical contrast) employed for measuring contrast detection thresholds.

sensitivity function (CSF). This is experimentally obtained by considering periodic stimuli like sine-wave gratings or their convolutions with a Gaussian envelope, so-called Gabor patches (cf. Fig. 8.3). The measured detection threshold is defined in terms of *Michelson contrast*

$$\frac{L_{\max} - L_{\min}}{L_{\max} + L_{\min}} = \frac{\Delta L}{\bar{L}} \quad \text{with} \quad L_{\max} = \bar{L} + \Delta L, \quad L_{\min} = \bar{L} - \Delta L,$$

where ΔL denotes the amplitude and \bar{L} the base-level luminance of the corresponding stimulus. For the achromatic channel, the CSF shows a band-pass behavior, where the peak sensitivity increases with luminance level. In accordance with the optical properties of the eye and the density of the photoreceptor mosaic, the sensitivity becomes zero at frequencies beyond about 60 cycles per degree of visual angle (cpd). The CSFs for the chromatic channels are of low-pass nature, have smaller peak sensitivities, and feature lower cut-off frequencies. This is especially true for the yellow–blue channel owing to the wider spacing of the relatively few S cones.

The threshold for detecting a stimulus is typically affected by the presence of another stimulus, a phenomenon called *visual masking*. A test or target pattern that is just visible by itself is masked by a masking stimulus if their superposition cannot be discriminated from the masking stimulus alone. Consequently, the masking pattern raises the contrast required for the target to be detectable.⁴ On the other hand, at least in case test and masking pattern show similar characteristics, it may happen that the presence of the masking stimulus actually facilitates detection of the test pattern, i.e. although the test pattern alone is not visible, its superposition with the masking stimulus is discriminable from just the masking pattern [213]. Masking is most pronounced for signals within the same channel but may as well occur across channels. Furthermore, note that masking also happens in the temporal domain.

Visual masking is often exploited in computer graphics to hide artifacts and conceal lower rendering quality. For instance, applying a complex texture may mask lower soft shadow quality (recall Fig. 5.4 on page 76), coarse quantizations (like considering only 8×8 regularly spaced light sample points during soft shadow computation), and low tessellation rates (except at silhouettes).

⁴An intuitive way to think about (within-channel) masking is that a masking stimulus already introduces a visible contrast c_m , and hence, to detect a superimposed target stimulus of contrast c_t , it becomes decisive that the resulting “contrast” $((c_m + c_t) - c_m)/c_m = c_t/c_m$ between the overall contrasts c_m and $c_m + c_t$ is above a certain threshold.

8.3 Color and color appearance

In a nutshell, the perception of color comprises three main steps. At first, light hitting the retina is captured by cones, whose responses provide an initial trichromatic encoding. This is then converted to an opponent color representation. Finally, the according signals are subjected to further cognitive processing.

Recall that each of the three cone types has a distinct spectral responsivity. Cone responses to an incident light can hence be determined by computing the integral of this responsivity multiplied with the light's spectral power distribution over all wavelengths. Note that this means that the spectral power distribution gets reduced to just three values. As a consequence, different spectral power distributions may yield identical responses and hence cannot be distinguished. Corresponding color pairs are referred to as metamers.

This is actually exploited to reproduce a certain color by taking a set of three lights with different spectral power distributions (like the phosphors employed in CRT displays), the so-called primaries, and mixing them together, where the power of each is adapted appropriately such that the desired cone responses are evoked. The color may therefore also be specified by the employed amounts of the primaries instead of the corresponding cone responses. Note that the conversion between these different sets of so-called *tristimulus values* can be described by a simple linear transformation. This also enables choosing actually non-existent primaries that yield tristimulus values which have a desired behavior or meaning.

One prominent example is the CIE XYZ color space, which was designed such that the Y value corresponds to the color's luminance. By contrast, Meyer [251] derived his AC_1C_2 space by decorrelating cone responses via a principal component analysis. The achromatic channel A roughly equals CIE Y , while the chromatic channels correspond to red–green (C_1) and yellow–blue (C_2) opponencies. Interestingly, the color space was devised with the aim of computing AC_1C_2 tristimulus values from a given spectral power distribution via Gaussian quadrature using a minimum number of wavelength samples.

Ideally, a color space should be perceptually uniform such that the Euclidean distance between two colors indicates the magnitude of their perceived difference. To approach this goal, more advanced opponent color spaces have been devised which subject tristimulus values to further non-linear transformations. They typically compress high values and augment small values to account for the discrimination threshold rising with stimulus intensity. Moreover, they may perform some form of chromatic adaptation.

One major example is the CIE $L^*a^*b^*$ (or just CIELAB) color space, where L^* denotes lightness and a^* and b^* specify the position with respect to the red–green and yellow–blue opponencies. As it is not completely uniform, several advanced formulae have been proposed for computing color differences, improving on using just Euclidean distance. A reasonably simple and hence popular one is CIE94, which introduces chroma- and hue-dependent weights for achieving a greater uniformity [112]. Another officially endorsed formula is the computationally much more involved CIEDE2000 [354], which features additional improvements, mainly affecting the performance for blue and gray colors. Nevertheless, it still suffers from several problems [197].

Further examples specifically devised for image processing applications include *IPT* [103], which aims at being uniform in perceived hue, as well as Chong et al.'s color space [70], whose parameterization was directly derived from the objectives of being perceptually uniform as well as of color difference vectors being invariant to reillumination.

While these color representations are well suited to quantify perceived color differences,

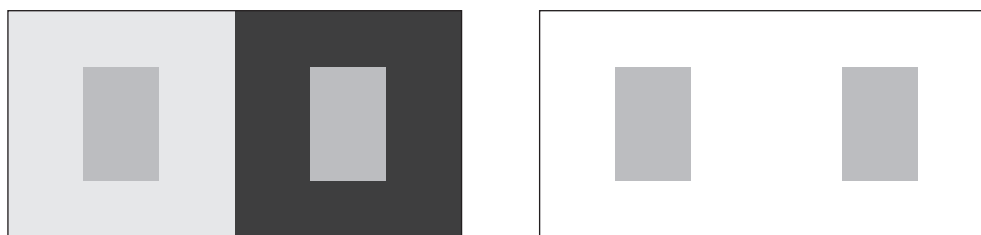


Figure 8.4 Example of simultaneous contrast. Although the two small gray rectangles are of identical shade, as demonstrated in the right image, they appear to be of different brightness in the left image.

they are largely unable to describe a color’s actual appearance because this is influenced by many additional factors that are not accounted for. They hence cannot capture effects like simultaneous contrast (see Fig. 8.4) or that of “discounting the illuminant”, where the influence of the illumination’s color is discounted when perceiving the color of an object.

This shortcoming is addressed by *color appearance models* (CAMs) [112]. They require the specification of additional parameters, like the relative luminance of the surround or the illumination’s color, and utilize these to model mechanisms like chromatic adaptation as well as some appearance effects. A color’s predicted appearance is then usually described in terms of appearance attributes like hue, lightness, brightness, chroma, colorfulness and saturation. Examples of CAMs are CIECAM97s [242] and its simpler but often superior successor CIECAM02 [256]. Also note that with CAM02-SCD and CAM02-UCS [241] two color difference formulae for the CIECAM02 model were developed which perform well for both daylight conditions and incandescent lighting, as often encountered in rooms.

One major limitation of both the perceptually (roughly) uniform color spaces as well as of CAMs is that they were designed for uniform color patches subtending a visual angle of normally two degrees. They are hence ignorant of the spatial structure of more complex stimuli like images, which, however, clearly affects color perception.

To somewhat alleviate this, Zhang and Wandell [415] extended CIELAB to S-CIELAB, introducing a preprocessing step which maps the input tristimulus values into an opponent color space and then performs a spatial filtering with a CSF approximation (a series of Gaussians), before finally transforming into CIELAB space. To exercise more control over the filter kernel, CSF filtering may alternatively be performed in frequency space [173], which, however, necessitates computing Fourier transformations. A completely different approach is adopted by Hong and Luo [163], who weight CIELAB pixel differences according to their importance. To this end, first a histogram of hue values is determined, which is then used to derive a measure of significance. Furthermore, with the modular iCAM framework [113, 114] and its variant iCAM06 [196], first efforts emerged to evolve CAMs to complete image appearance models, which allow assessing image quality and quantifying image differences as well as maintaining image appearance across different media (of potentially varying dynamic range).

8.4 Visual attention

The human’s processing capacity is limited and, in particular, significantly too small to completely process all acquired sensory information. One mechanism to cope with this situation is

attention [171], which serves to select relevant information and to allocate resources accordingly. In case of vision, attention often causes the region of primary importance to be fixated on, such that it is captured by the fovea, where the eye's spatial resolution is highest. Interestingly, when a stimulus is completely unattended, it may go unnoticed despite being clearly visible and even if it is recorded by the fovea, a phenomenon called *inattention blindness*.

Attention is guided by a low-latency bottom-up process, which is purely stimulus-driven and attracted by salient image features, and a top-down process, which is task-dependent. A well-known computational model for the first component is the *saliency map* [170, 172], which derives for each pixel of a given image its saliency relative to the whole image. At first, visual features like color (using four channels: red, green, blue, yellow), intensity and orientation are determined. Then, according multi-scale feature maps are derived by applying difference-of-Gaussians filters corresponding to center-surround receptive fields at different scales. Finally, these maps are combined to a saliency map. Several improvements for, extensions to and variants of this basic model have been devised. Walther [392], for instance, points out problems with the original definition of color opponencies and incorporates motion as an additional feature. A simplified GPU-based implementation is described by Longhurst et al. [228], which apart from color and luminance takes edges, motion, depth and also habituation into account. We note, however, that their employed feature map combination strategy [227] didn't yield satisfactory results in tests we conducted. Furthermore, the concept of a saliency map was also transferred to the geometric domain to quantify regional importance on meshes [210] as well as to the auditory modality [179].

The top-down attentional process is significantly harder to model, as it depends on the context of viewing and the person's experiences. An often-cited study by Yarbus [407], which was later repeated [221], corroborating many original results, shows that the gaze pattern when viewing a painting is strongly influenced by a given task, suggesting that attention is guided by such a task. This motivated researchers like Cater et al. [63] to encode task relevance in a task map analogous to the saliency map. However, automatically deriving such task maps remains an open challenge. Perhaps recent results utilizing eye tracking [295, 369] may eventually lead to a practical way to achieve this, though, at least for some applications. But even then, further issues like combining bottom-up and top-down predictions or accounting for the viewer's memory remain to be solved.

CHAPTER 9

Perceptually motivated rendering

The general overview of human visual perception and according aspects relevant to computer graphics provided in the last chapter suggests a huge potential that exploiting perception offers for improving rendering efficiency and selecting the lowest quality actually required to evoke a realistic appearance. But it also raises the question how these perceptual concepts and characteristics can concretely be utilized for rendering.

This chapter is dedicated to address that issue. At first, we present realizations of core properties like varying contrast sensitivity and visual masking, which can be used as building blocks in perceptually based algorithms. They are also employed in more complex components, like complete computational vision models and visual difference metrics, which we briefly review subsequently. After that, a short overview of existing perceptually-motivated applications is given, which demonstrate the concrete utilization of perceptual results in domains like image synthesis.

Disproving the often-voiced complaint that perceptually based image metrics are too time-consuming to evaluate for being suitable for on-the-fly usage in real-time rendering, we then present a rapid GPU-based implementation of threshold maps. These encode for each pixel of an input image the threshold below which a change in luminance will not be noticed. Serving as an enabling core component, they are key to a subsequently described interactive perceptual rendering pipeline which exploits inter-object, scene-level visual masking for geometric LOD control. Finally, we conclude with highlighting some general problems encountered when applying perceptual results.

9.1 Building blocks for computational models

Human visual perception is a complex process, which involves many interacting mechanisms and is far from being completely understood. To cope with this complexity, it is reasonable to try to concentrate on individual aspects and investigate and model them in isolation. Note that such a procedure is also required by psychophysical experimental practice to keep the number of parameters low. The resulting components can then be combined appropriately to implement a more complete computational model that takes several aspects into account simultaneously.

In the following, we describe some relevant example realizations of such building blocks, focusing on contrast sensitivity functions, the effect of visual masking, and the decomposition of an image into multiple channels, for instance tuned to certain bands of spatial frequency.

9.1.1 Contrast sensitivity functions

Owing to the optics of the eye¹ and the subsequent neuronal processing (starting with the photoreceptors), the contrast threshold for detecting a stimulus is not constant but depends on many factors like its spatial frequency content and the *adaptation luminance*, i.e. the luminance level the HVS is currently adapted to. As detailed in Sec. 8.2, this threshold variation is typically quantified in terms of a contrast sensitivity function.

Luminance CSFs

The vast majority of contrast sensitivity measurements conducted are concerned with luminance contrast. Taking the data from one such study, Barten [23, 24] derived the following simple CSF:

$$\text{csf}_{\text{B89}}(\rho, L) = a\rho \exp(-b\rho) \sqrt{1 + c \exp(b\rho)} \quad (9.1)$$

with

$$a = 440 (1 + 0.7/L)^{-0.2}, \quad b = 0.3 (1 + 100/L)^{0.15} \quad \text{and} \quad c = 0.06,$$

where ρ denotes spatial frequency in cpd and L is the adaption luminance in cd/m^2 . As Fig. 9.1 a shows, the CSF features a band-pass behavior, and peak sensitivity rises with adaption luminance. Note that an extension exists which additionally takes the effect of display size into account [23, 25].²

Later, Barten [26] also devised a complex physical contrast sensitivity model that accounts for many influences like neural noise. A simplified version for a typical viewer is given by

$$\text{csf}_{\text{B03}}(\rho, L, A) = \frac{5200 \exp(-0.0016\rho^2 (1 + 100/L)^{0.08})}{\sqrt{\left(1 + \frac{144}{A} + 0.64\rho^2\right) \left(\frac{63}{L^{0.83}} + \frac{1}{1 - \exp(-0.02\rho^2)}\right)}}, \quad (9.2)$$

where A denotes the angular display area in square degrees of visual angle. Furthermore, Barten describes extensions to account for orientation (sensitivity is highest for horizontal and vertical gratings) and the effect of surround illumination.

Several other CSFs have been derived, ranging from pretty sophisticated ones, like Daly's function [84, 85] of spatial frequency, orientation, adaptation luminance, display size, viewing distance and eccentricity, to simple ones, like Martin et al.'s fit [249] to measurements by van Nes and Bouman [383],

$$\text{csf}_{\text{M92,L}}(\rho, L) = 10^{S(\log_{10} \rho, \log_{10} L)}$$

with

$$S(f, l) = (2.094 + 0.6019f - 0.9730f^2) + (0.2218 + 0.6828f - 0.3656f^2)l \\ - (0.10258 - 0.07854f + 0.05234f^2)l^2 - (0.01557 + 0.02197f - 0.03920f^2)l^3.$$

¹The blurring caused by the optical system (e.g. by cornea and lens) can be described by a *point spread function* (PSF). Its convolution with an input image then yields the appropriately defocused “retinal” image. Equivalently, an *optical transfer function* (OTF) in Fourier domain may be utilized.

²The only change is actually to set $a = 540 (1 + 0.7/L)^{-0.2} / \left(1 + \frac{12}{w(1 + \frac{1}{3}\rho)^2}\right)$, where w denotes the angular display size in degrees.

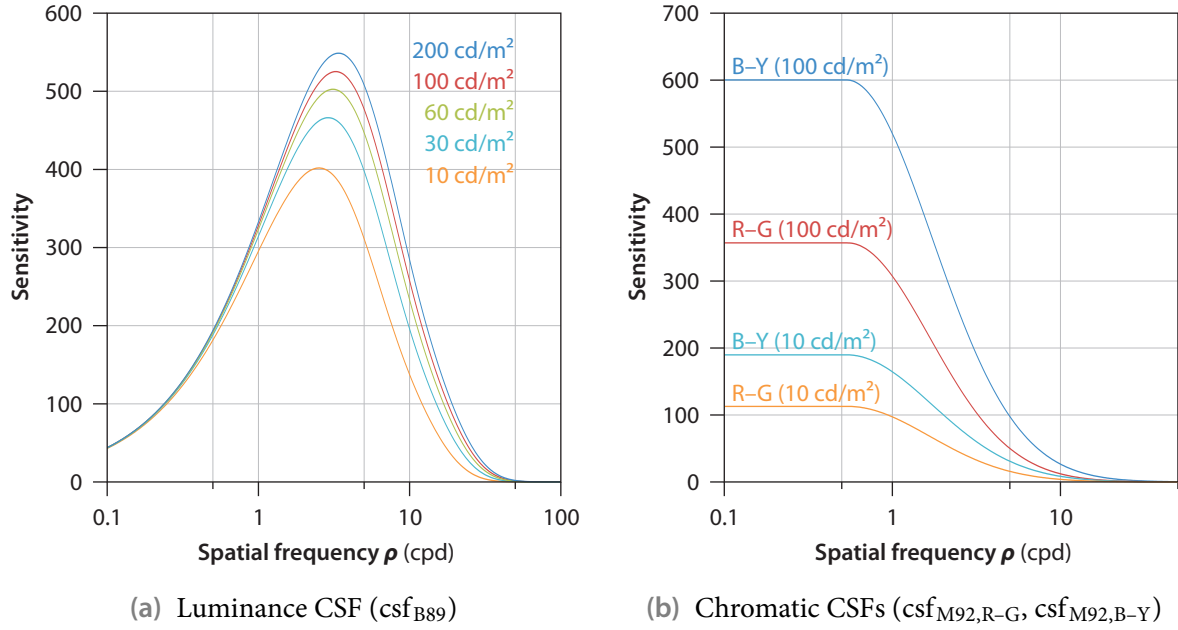


Figure 9.1 Luminance and chromatic CSFs at different adaptation luminance levels.

Chromatic CSFs

By contrast, little data is available for chromatic contrast sensitivity. Often, the results of measurements made by Mullen [258] are employed. Martin et al. [249], for instance, provide the following fits:

$$\begin{aligned} \text{csf}_{\text{M92,R-G}}(\rho, L) &= S(\log_{10} \rho, \log_{10} L, -0.58800, 2.1406, 0.9153, -0.2664), \\ \text{csf}_{\text{M92,B-Y}}(\rho, L) &= S(\log_{10} \rho, \log_{10} L, -0.16111, 1.9395, 0.8347, -0.2728), \end{aligned}$$

where

$$\log_{10} S(f, l, a, b, c, d) = a + \left\{ \begin{array}{ll} b, & f \leq d \\ b - c(f - d)^2, & \text{otherwise} \end{array} \right\} + \left\{ \begin{array}{ll} 0.5l, & l \leq 2.3 \\ 1.15, & \text{otherwise} \end{array} \right\}.$$

Note that due to the absence of according data, the dependency on adaptation luminance L is simply modeled by scaling the sensitivity according to the square root of L . The two CSFs are depicted in Fig. 9.1 b, demonstrating their low-pass nature.

As pointed out by Bolin and Meyer [40, 41], Mullen's experiments corrected for chromatic aberration, and hence this effect should be accounted for when applying the CSFs, effectively reducing sensitivity for the blue–yellow channel (csf_{M92,B-Y}).

Spatio-velocity CSFs

Furthermore, CSFs have been derived for moving stimuli. One such spatio-velocity CSF is given by Kelly [183]:

$$\text{csf}_{\text{K79}}(\rho, v) = (6.1 + 7.3|\log_{10}(v/3)|^3) v (2\pi\rho)^2 \exp(-4\pi\rho(v + 2)/45.9), \quad (9.3)$$

where v denotes retinal velocity in degrees of visual angle per second (deg/s). Note that due to drift eye movements, the retinal velocity of static stimuli fixated on is typically non-zero (about 0.15 deg/s).

Introducing parameters c_0 , c_1 and c_2 for adapting aspects like the frequency of peak sensitivity, this model was later refined by Daly [86] to

$$\text{csf}_{\text{D98}}(\rho, \nu) = \left(6.1 + 7.3 \left| \log_{10}(c_2 \nu / 3) \right|^3\right) c_0 c_2 \nu (2\pi c_1 \rho)^2 \exp(-4\pi c_1 \rho (c_2 \nu + 2)/45.9). \quad (9.4)$$

For a luminance level of about 100 cd/m², he suggests using the values $c_0 = 1.14$, $c_1 = 0.67$ and $c_2 = 1.7$.

A different approach is taken by Burbeck and Kelly [57], who express the CSF as a linear combination of two space-time-separable low-pass-like functions modeling an excitatory center and an inhibitory surround component:

$$\text{csf}_{\text{B80}}(\rho, \nu) = E(\rho, \nu) - I(\rho, \nu)$$

with

$$E(\rho, \nu) = \frac{S_E(\rho) T_E(\rho \nu)}{S_E(0.5 \text{ cpd})} \quad \text{and} \quad I(\rho, \nu) = \frac{S_I(\rho) T_I(\rho \nu)}{S_I(0.5 \text{ cpd})},$$

where both the spatial terms

$$S_E(\rho) = \begin{cases} \frac{\text{csf}_{\text{K79}}(10 \text{ cpd}, 0.1 \text{ deg/s})}{\text{csf}_{\text{K79}}(10 \text{ cpd}, 1.9 \text{ deg/s})} \cdot \text{csf}_{\text{K79}}(\rho, 19 \text{ cy/s}/\rho), & \rho \leq 10 \text{ cpd} \\ \text{csf}_{\text{K79}}(\rho, 1 \text{ cy/s}/\rho), & \text{otherwise} \end{cases}$$

$$S_I(\rho) = S_E(\rho) - \text{csf}_{\text{K79}}(\rho, 1 \text{ cy/s}/\rho)$$

and the temporal ones

$$T_E(\omega) = \begin{cases} \frac{\text{csf}_{\text{K79}}(0.5 \text{ cpd}, 38 \text{ deg/s})}{\text{csf}_{\text{K79}}(10 \text{ cpd}, 1.9 \text{ deg/s})} \cdot \text{csf}_{\text{K79}}(10 \text{ cpd}, \omega/10 \text{ cpd}), & \omega \leq 19 \text{ cy/s} \\ \text{csf}_{\text{K79}}(0.5 \text{ cpd}, \omega/0.5 \text{ cpd}), & \text{otherwise} \end{cases}$$

$$T_I(\omega) = T_E(\omega) - \text{csf}_{\text{K79}}(0.5 \text{ cpd}, \omega/0.5 \text{ cpd})$$

are defined using Kelly's CSF from (9.3).

Later, Kelly [184] observed that by combining the center and surround components additively instead of subtractively a chromatic CSF is obtained:

$$\text{csf}_{\text{K83}}(\rho, \nu) = \frac{1}{30} (E(\rho, \nu) + I(\rho, \nu)). \quad (9.5)$$

Caveats

Given this variety of CSFs, it is worth recalling that they are typically the result of some function fitting to one or more data sets. Hence, different sensitivities may be reported depending on the underlying measurements. In particular, note that experimental stimuli and conditions as well as the subjects have a large influence on the results. As a concrete example, consider the luminance CSFs shown in Fig. 9.2. Although they all exhibit a similar band-pass behavior, they differ in both the magnitude of the peak sensitivity and the spatial frequency where it occurs. Further note that while a CSF can be evaluated at arbitrary parameter values, it may only provide reasonable results for a limited range, depending on the fitting procedure employed in its derivation and on the parameter range covered by the base data set. Consequently, CSF values should always be considered a rough estimate and not a ground truth. This analogously applies to other perceptual functions.

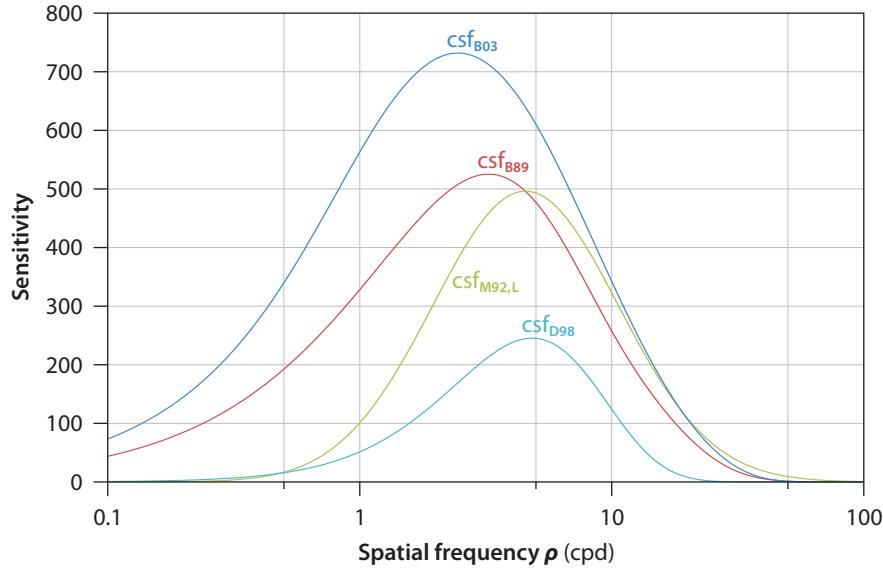


Figure 9.2 Different luminance CSFs at adaptation luminance level $L = 100 \text{ cd/m}^2$: Barten's $\text{csf}_{B89}(\rho, 100 \text{ cd/m}^2)$ and $\text{csf}_{B03}(\rho, 100 \text{ cd/m}^2, 11^\circ \cdot 17^\circ)$, Martin et al.'s $\text{csf}_{M92,L}(\rho, 100 \text{ cd/m}^2)$ and Daly's $\text{csf}_{D98}(\rho, 0.15 \text{ deg/s})$.

Threshold versus intensity

A CSF quantifies the detection threshold for a periodic pattern of a certain spatial frequency. By determining the minimum threshold across all frequencies for a given luminance level, a threshold-versus-intensity function can be derived:

$$\text{tvi}(L) = \frac{L}{\max_{\rho} \text{csf}(\rho, L)}.$$

However, since CSF data sets typically only consider a small number of luminance levels L , and hence luminance dependency is usually modeled by simple expressions, such TVI functions are not very accurate.

Better results are obtained by deriving a function directly from dedicated TVI measurements (cf. Sec. 8.2). For instance, Ward et al. [394] give the following TVI function:

$$\log_{10} \text{tvi}_W(L) = \begin{cases} -2.86, & \log_{10} L < -3.94, \\ (0.405 \log_{10} L + 1.6)^{2.18} - 2.86, & -3.94 \leq \log_{10} L < -1.44, \\ \log_{10} L - 0.395, & -1.44 \leq \log_{10} L < -0.0184, \\ (0.249 \log_{10} L + 0.65)^{2.7} - 0.72, & -0.0184 \leq \log_{10} L < 1.9, \\ \log_{10} L - 1.255, & 1.9 \leq \log_{10} L. \end{cases} \quad (9.6)$$

Note that it is just a combination of rod- and cone-specific TVI functions provided by Ferwerda et al. [126], with rods featuring a higher sensitivity than cones for luminance levels below $\log_{10} L = -0.0184$.

Fig. 9.3 depicts both Ward et al.'s TVI function as well the one implied by Barten's CSF csf_{B89} . They are additionally shown using the alternative representation as *contrast-versus-intensity* (CVI) function

$$\text{cvi}(L) = \frac{\text{tvi}(L)}{L}.$$

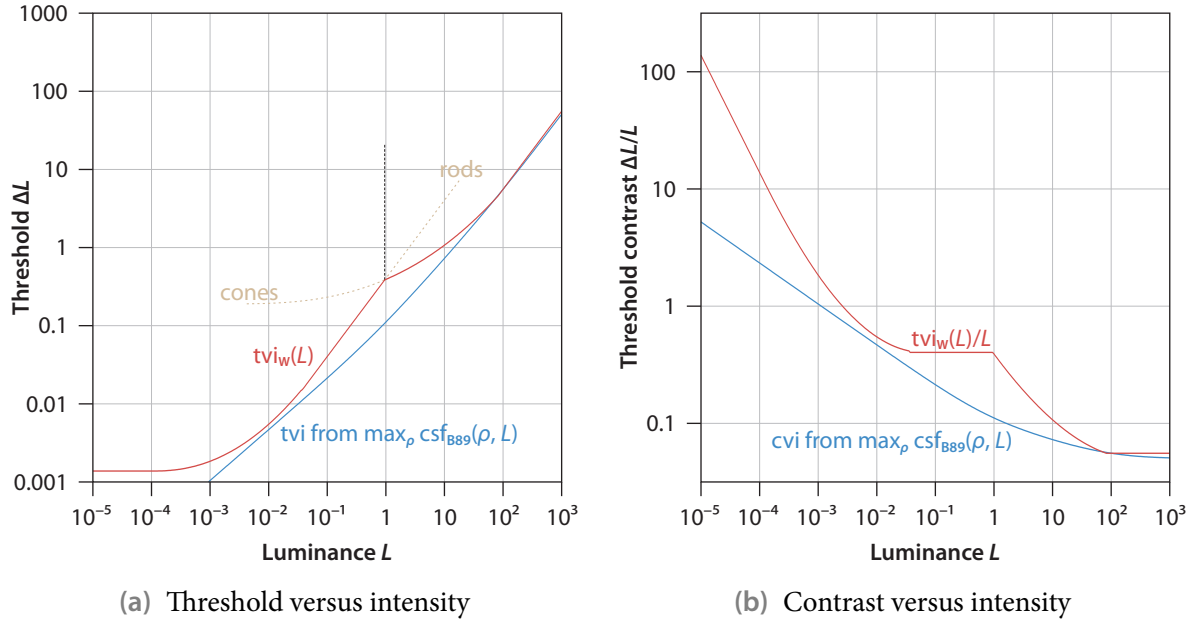


Figure 9.3 TVI functions and their corresponding CVI functions. The TVI function derived from Barten's CSF csf_{B89} is scaled such that it coincides with Ward et al.'s TVI function tvi_w at luminance level $L = 100 \text{ cd/m}^2$.

9.1.2 Visual masking

Recall from Sec. 8.2 that visual masking causes the detection threshold for a stimulus to be affected by the presence of another stimulus, that is, the threshold is elevated (or reduced in case of facilitation). Ignoring cross-channel masking effects, one approach to model masking is to consider the superposition of target and masking stimulus and compute a threshold elevation factor for the combined contrast signal. An according function was devised by Daly [84, 85]:

$$T_D(c) = \left(1 + \left(k_1 (k_2 c)^s \right)^b \right)^{1/b}, \quad (9.7)$$

where c denotes normalized contrast (contrast multiplied by the according contrast sensitivity), i.e. $c = 1$ corresponds to 1 JND. He advocates using the constants $k_1 = 0.0153$, $k_2 = 392.498$, $b = 4$ and $s = 0.7$. Note that T_D converges to $c^s = c^{0.7}$ for increasing c .

Elevating a detection threshold effectively compresses the according contrast signal. One may hence equivalently subject the contrast to a so-called *transducer function*, where the relationship

$$\text{transducer}(c) = \frac{c}{\text{elevation}}$$

holds. An example is Lubin's transducer [236]

$$T_L(c) = \frac{2c^n}{c^{n-w} + 1}, \quad (9.8)$$

where $n \in [2, 2.5]$ and $w = 0.2$ are reasonable parameter choices [215].

Furthermore, some mechanisms for modeling masking are offered by JPEG 2000 as part of its visual optimization tools [87, 413]. They employ a simple power function c^α ($0 < \alpha \leq 1$) as transducer, where a value of $\alpha = 0.7$ is used instead of the typically adopted choice $\alpha = 0.3$

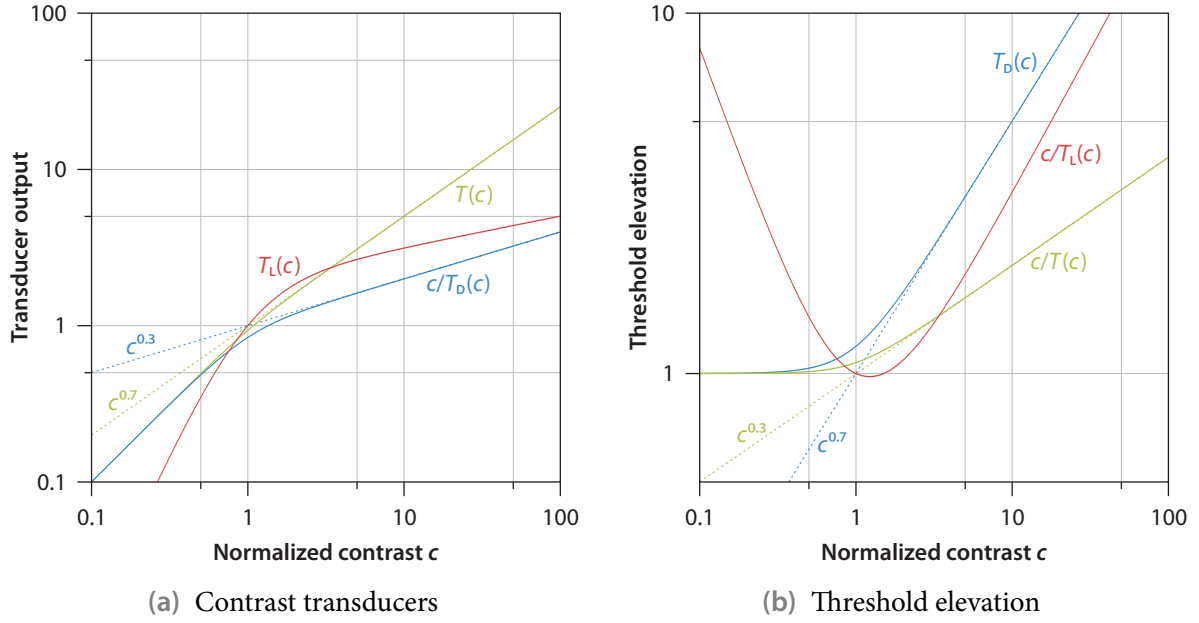


Figure 9.4 Visual masking functions. For Lubin’s transducer T_L , values of $n = 2.2$ and $w = 0.2$ are employed.

(like in T_D ’s asymptotic behavior), yielding better results. An extension of this basic so-called *self-masking* is *point-wise extended masking*, which performs an additional normalization by a neighborhood masking factor, taking also a pixel’s vicinity into account. This enables capturing further masking due to complex texture.

Finally, note that within our vision model, detailed later in Sec. 10.3.3, we obtained good results with the transducer

$$T(c) = \text{sign}(c) \cdot \frac{|c|}{\left(1 + (|c|^{0.3})^{10}\right)^{0.1}}, \quad (9.9)$$

which corresponds to Daly’s threshold elevation function from (9.7) with constant values $k_1 = k_2 = 1$, $b = 10$ and $s = 0.3$. It is depicted in Fig. 9.4, together with the other presented masking functions.

9.1.3 Multi-channel decomposition

Another key concept in modeling human visual perception is the decomposition of the input image into multiple channels, tuned to specific characteristics like certain bands of spatial frequency and orientation. Most important is typically the transformation of the spatial frequency content into individual frequency bands, each normally spanning roughly one octave. One approach to accomplish this is to filter the input image with the difference of two Gaussians whose spreads differ by a factor of two. More precisely, first a Gaussian stack is built, where each level G_i results from the convolution of the image with a Gaussian, with the employed spread being doubled every level. By subtracting two of these band-limited levels, a certain frequency band of the image can then be extracted. A spatial frequency decomposition is hence obtained by constructing a so-called Laplacian stack, where level i is computed from the Gaussian stack as $G_i - G_{i+1}$.

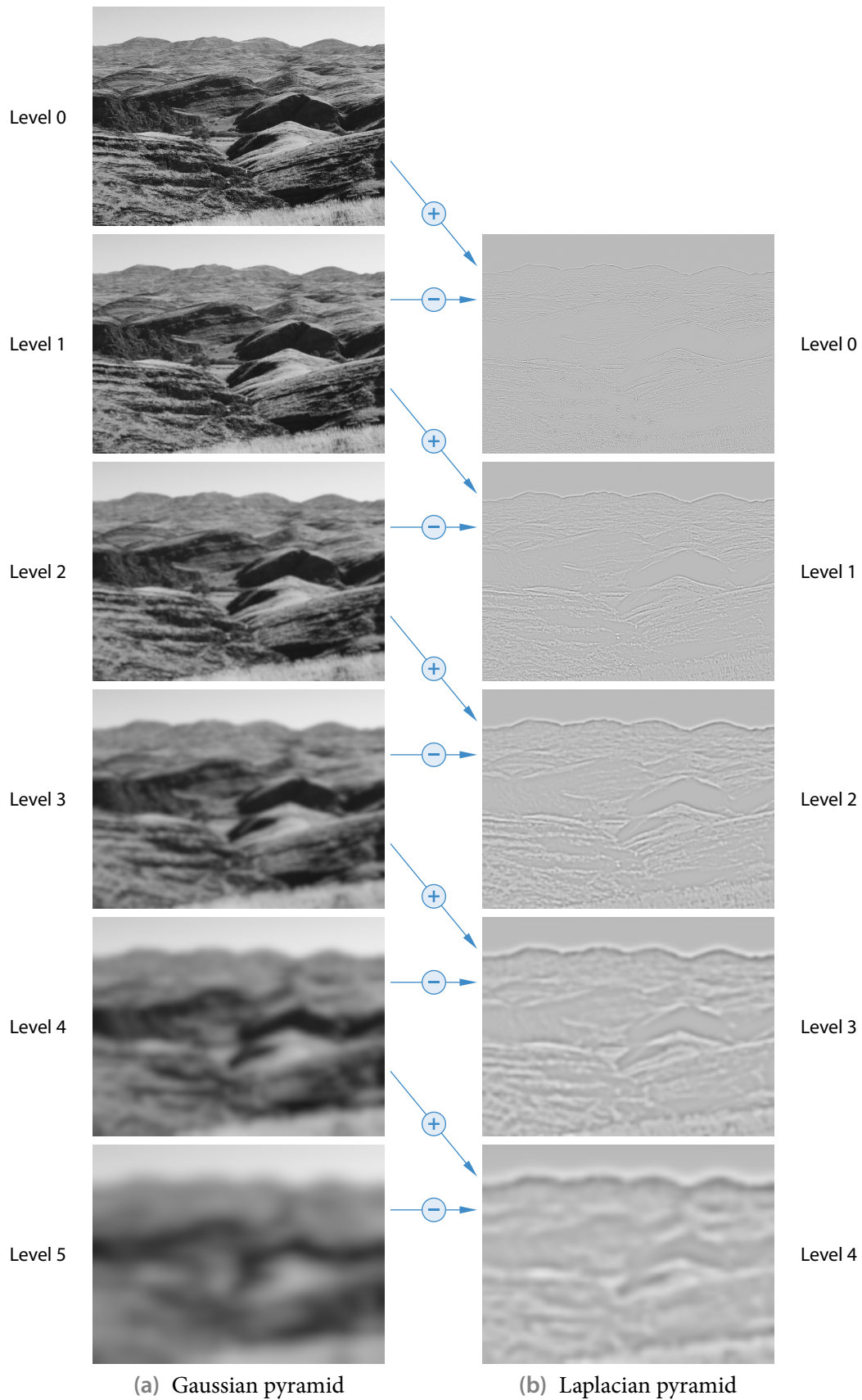


Figure 9.5 First levels of a Gaussian pyramid and the corresponding Laplacian pyramid. Solely for visualization purposes, all levels were upsampled to the same size.

An efficient approximate realization is the *Laplacian pyramid* [58], which maintains lower-frequency levels at a reduced resolution. Initially, a Gaussian pyramid is constructed, where the input image constitutes the finest level $i = 0$. Each next coarser level $i + 1$ is obtained by filtering the preceding level i with a fixed-size Gaussian kernel and downsampling the result to half its horizontal and vertical resolution. Note that this downsizing effectively corresponds to a doubling of the Gaussian's spread at the next level. The Laplacian pyramid's level i is then defined by the difference between level i and the appropriately upsampled level $i + 1$ of the Gaussian pyramid. An example of this multi-resolution representation is shown in Fig. 9.5.

Several further decomposition methods have been devised, often additionally providing some orientation selectivity. Examples include simple Haar wavelets [41] and the sophisticated cortex transform [396].

9.2 Vision models and visual difference metrics

Utilizing building blocks like the ones discussed in the previous section, complex vision models have been devised which output a prediction of the visual system's response to an input image. By comparing the responses for two images, an estimate of the perceivable differences between them can be derived. Apart from corresponding visual difference metrics, several others exist which are not directly based on a computational vision model.

A well-known and quite complex metric designed for image fidelity assessments is Daly's visual differences predictor (VDP) [84, 85], which works solely on luminance values and returns a map of detection probabilities. The underlying vision model incorporates an amplitude non-linearity accounting for light adaptation, a filtering with an anisotropic CSF, as well as a cortex-transform-based decomposition into 31 channels tuned to different spatial frequencies and orientations. For visual masking, the threshold elevation function T_D from (9.7) is utilized. Note that the VDP's application involves Fourier transformations, since most computations are performed in frequency space. Targeting HDR images, Mantiuk et al. [248] later modified and extended the metric, resulting in the so-called HDR VDP.

Another sophisticated metric is the Sarnoff visual discrimination model (VDM) [236], which, unlike Daly's VDP, operates entirely in the spatial domain. After an optics and a re-sampling stage, a contrast pyramid is constructed and subjected to orientation filtering. Subsequently, the energy responses are determined and weighted by a CSF, before the transducer function T_L from (9.8) is applied. Finally, a JND map is calculated, quantifying the predicted differences in JND units. At least for small differences, these relate linearly to the perceived difference magnitudes.

An evaluation [215, 216] of Daly's VDP and the Sarnoff VDM suggests that both perform rather well but also suffer from problems, with the Sarnoff model yielding slightly better results, overall. Furthermore, it was noted that the Sarnoff VDM is faster. Nevertheless, even a GPU-accelerated version reportedly takes more than one second for images of size 512^2 on a (now outdated) NVIDIA GeForce FX 5900 Ultra [401].

Bolin and Meyer [40, 41] simplify the Sarnoff model by using a Haar wavelet transform for multi-channel decomposition, which significantly improves execution time but introduces some blocking artifacts. Moreover, they extend the VDM to deal with color instead of just luminance. In another variant of the Sarnoff VDM, Lindstrom [219] drops the orientation-dependent processing completely. He further replaces the CSF by a TVI function to lift the frequency dependency and hence decouple the results from the solid angle the images subtend at a viewer's eyes.

A rather complex color vision model targeting primarily visual masking was introduced by Ferwerda et al. [127]. It was successfully applied to assess the ability of textures of various contrast, spatial frequency content and orientation to mask artifacts due to flat shading. Pattanaik et al. [289] developed another quite complete vision model, which targets tone mapping and hence accounts for a wide range of illumination levels as well as chromatic adaptation, but doesn't incorporate any orientation-dependent processing. Their model was later also applied as foundation of an image metric [119]. More recently, Tolhurst et al. [377] developed a multi-scale color vision model for predicting the visual discriminability of two images. They also extended their model [235] to yield a rating of how large a visible difference is perceived.

Further metrics, introduced mainly to control image synthesis, include Gaddipatti et al.'s image comparison metric, based on Daubechies wavelets [134], and Farrugia et al.'s color image metric [118], which restricts itself to single-scale processing for performance reasons. Interestingly, an adaptive method is suggested where the image is decomposed via a quadtree and only a representative number of pixels in each cell get compared. A rather different approach is taken by Neumann et al. [268] who employ rectangles sized and distributed quasi-randomly. The difference between two images is then computed as a CSF-weighted combination of the average-color differences between corresponding rectangles in the images, using a modified difference formula in CIELUV space (another perceptually roughly uniform color space similar to CIELAB).

Moreover, metrics exist which also take motion into account. Myszkowski et al. [264, 265], for instance, devised an animation quality metric (AQM), which extends Daly's VDP to the temporal domain, incorporating the spatio-velocity CSF from (9.4). The same CSF is also employed by Yee et al. [409] in the computation of an error tolerance map, termed aleph map.

Threshold maps

A special variant of an image difference metric is a *threshold map*. It is derived from just one image and specifies for each pixel the threshold below which (additive) changes in the pixel's value are predicted to go unnoticed. Hence, to determine whether two images can be discriminated, their pixel-wise difference is computed and compared against the threshold map for one of the images. Any differences are then assumed to be only perceivable at pixels where the deviation between the images is above the according threshold.

Threshold maps were introduced by Ramasubramanian et al. [308], focusing on tolerable luminance errors during image synthesis. At first, they construct a Gaussian pyramid with levels G_i from the input luminance image, and then derive a contrast pyramid, where level i is determined as $(G_i - G_{i+1})/G_{i+2}$, upsampling coarser levels accordingly. For a certain pixel, the luminance threshold

$$\Delta L = 0.2 \text{ tvi}_W(L) \cdot S$$

is computed by modulating the threshold indicated by Ward et al.'s TVI function tvi_W from (9.6) by a spatial threshold elevation factor

$$S = \frac{\sum_i c_i \cdot F_i \cdot T_D(c_i \cdot \text{csf}_{B89}(\rho_i, L))}{\sum_i c_i}, \quad (9.10)$$

which accounts for a reduction in sensitivity due to spatial frequency content and masking. The adaptation luminance L may be taken from that Gaussian pyramid level where one pixel roughly corresponds to one degree of visual angle. For S , summation occurs over all contrast

pyramid levels, each representing a different spatial frequency band with peak frequency ρ_i . The contrast from level i is denoted by c_i , and

$$F_i = \frac{\max_{\rho} \text{csf}_{\text{B89}}(\rho, 100 \text{ cd/m}^2)}{\text{csf}_{\text{B89}}(\rho_i, 100 \text{ cd/m}^2)} \quad (9.11)$$

quantifies the CSF-based threshold elevation relative to peak sensitivity, using Barten's csf_{B89} from (9.1). Note that the computation of F_i ignores the actual adaptation luminance and conservatively assumes a value of 100 cd/m^2 , enabling precomputation.³

Several further methods to derive a pixel's elevation factor S have been suggested. Walter et al. [391], for instance, employ JPEG luminance quantization matrices to quickly determine according elevation maps for textures. By contrast, Qu et al. [304] adopt the visual optimization tools of the JPEG 2000 standard to this end. Moreover, Yee [408] devised a color variant of threshold maps, which adds a final color comparison step but still performs the spatial processing solely for the luminance channel. Reportedly, it was successfully employed for testing of rendering software in the movie industry.

9.3 Overview of perceptually motivated applications

Many computer graphics algorithms exist which take some perceptual results into account. In the following, we provide a brief overview of applications which harness vision models, visual difference metrics or just some of the perceptually based building blocks covered in the preceding sections. Note that we solely focus on image synthesis and level-of-detail control and generation, and hence don't cover further domains where perception is exploited, like tone mapping or attention-guided selective rendering. For a broader review of perceptually motivated computer graphics applications, see, for instance, the report by O'Sullivan et al. [280].

Image synthesis

Many of the perceptually based difference metrics have been applied and often even primarily been developed to speed up off-line realistic image synthesis systems. Here, even rather high computational costs of a metric can easily amortize if the rendering process itself is quite expensive and the resulting savings are large enough. In algorithms operating sample-wise, like path tracing, computed difference or threshold values are typically employed as termination or refinement criterion. For example, Myszkowski et al. [263] use Daly's VDP, Bolin and Meyer [40] employ their variant of Sarnoff's VDM, and Ramasubramanian et al. [308] utilize their threshold map towards this end. Moreover, motion-aware metrics are employed by Myszkowski et al. [266] and Yee et al. [409] in global illumination computations for animation sequences.

Apart from image-space rendering systems, perceptual guidance was also employed for view-independent radiosity solutions. Gibson and Hubbold [137], for instance, utilize a simple perceptually-based metric to drive adaptive patch refinement, reduce the number of rays in occlusion testing, and optimize the resulting mesh. An overview of further radiosity methods exploiting human perception is given by Přikryl [301].

³To avoid threshold overestimation for lower luminance levels, F_i is actually set to 1.0 if $\rho_i < 4 \text{ cpd}$.

Level-of-detail control

Numerous geometric LOD algorithms have been developed which try to take perception into account during simplification or runtime LOD selection. In his extensive work, Reddy [310] determines the perceptual attributes of each LOD by rendering it from various directions and extracting the spatial frequency profile. To this end, a feature segmentation in image space is performed, and the lowest frequency of each feature is determined. Note this approach is supposedly rather inappropriate once complex shading is employed. During runtime, Reddy then utilizes the frequency data along with information about an object's size, velocity and eccentricity to select the appropriate LOD.

Luebke and Hallen [238, 239] introduced a framework for interactive view-dependent triangular model simplification where each simplification operation is mapped to a worst-case estimate of induced contrast and spatial frequency. This estimate is then subjected to a simple, empirical CSF to determine whether the operation causes a visually detectable change. On the down side, the approach assumes Gouraud shading and is overly conservative due to missing image-space information. However, subsequent work [400] tackles these shortcomings and achieves several improvements, among them support for textures and dynamic lighting.

Aiming for the highest visual quality attainable within a given resource budget, Dumont et al. [100] suggest a decision-theoretic framework where simple and efficient perceptually-motivated metrics are evaluated on the fly to drive the selection of the resolution at which textures are uploaded. Since a texture's masking properties are computed off-line and the approach necessitates multiple rendering passes and a frame buffer readback to obtain image-space information, the applicability of the used perceptually motivated metrics for less restricted setups is limited, though.

Finally, utilizing our GPU-based threshold maps (described in the next section), a perceptual rendering pipeline [98] was devised which exploits inter-object, scene-level visual masking for LOD selection. It is covered in more detail in Sec. 9.5.

Level-of-detail generation

Perceptual considerations are also sometimes employed when generating levels of detail. In their remeshing algorithm, Qu and Meyer [302] determine the perceptual properties of textures with the Sarnoff VDM. These are then used to guide the local vertex density in the remeshing result, thus trying to exploit masking effects. However, since the metric evaluation is performed in 2D texture space, it is unclear how well the spatial structure encountered there matches the one in actual renderings, where the texture is applied to the model's surface and hence can be expected to appear distorted in the general case—especially in silhouette regions. The method was later extended [303] to alternatively use a variant of the point-wise extended masking technique of JPEG 2000. Moreover, a mesh simplification algorithm is presented where for each vertex an importance weight is derived from the predicted masking effects.

One approach which enables using perceptually based image metrics instead of geometric measures to guide LOD generation was introduced by Lindstrom and Turk [220]. They suggest driving the geometric simplification process of a model by the magnitude of deviations in images rendered, from various viewpoints placed around the model, before and after the application of a certain simplification operation.⁴ While originally a simple L^2 metric was utilized,

⁴Lindstrom and Turk use a fixed screen-space size for the models regardless of their LOD and adopt a headlight illumination for each viewpoint, which is not really adequate. For instance, experimental results [317] suggest that lighting variation should be taken into account during simplification.

Lindstrom [219] later successfully employed his simple variant of the Sarnoff VDM. Moreover, a related image-driven mesh optimization algorithm was developed by him.

9.4 Real-time threshold maps

Perceptually based image metrics often involve many computations due to modeling complex procedures like a multi-channel decomposition and are hence rather expensive to evaluate. It is therefore expedient to leverage the computational power of graphics hardware to accelerate such metrics. Picking threshold maps (cf. Sec. 9.2) as a concrete example, we developed an according GPU-based variant that allows rapid evaluation. Actually, our realization is so fast that it is not only useful as a module within off-line algorithms, like for image synthesis and LOD generation, but, in particular, also enables the on-the-fly computation of this metric in real-time rendering. An according application which harnesses our GPU-based threshold maps for LOD control is described in the next section.

Given an input image, assumed to be in sRGB space, we initially compute the according CIE Y luminance values, outputting them into a floating-point texture. With this luminance image constituting the finest level, a Gaussian pyramid is subsequently determined and stored in the texture's mipmap chain. Each coarser level $i + 1$ is obtained by first filtering the preceding level i with a Gaussian kernel of size 5×5 and then bilinearly downsampling the result. In the employed pixel shader, we actually perform both steps together, directly computing the according linear combination of the involved 6×6 level- i texels. This requires merely nine texture fetches thanks to utilizing bilinear texture interpolation. More precisely, for each block of 2×2 texels, a single fetch is issued, choosing the sample point such that the resulting interpolation weights for the texels match their relative contributions. Note that no special treatment is necessary for images of non-power-of-two size, where the width or height of one mipmap level is not always exactly twice as large as the next coarser one's.

After that, we compute the luminance threshold in a single rendering pass. At first, the adaptation luminance L is determined by sampling that level of the Gaussian pyramid where

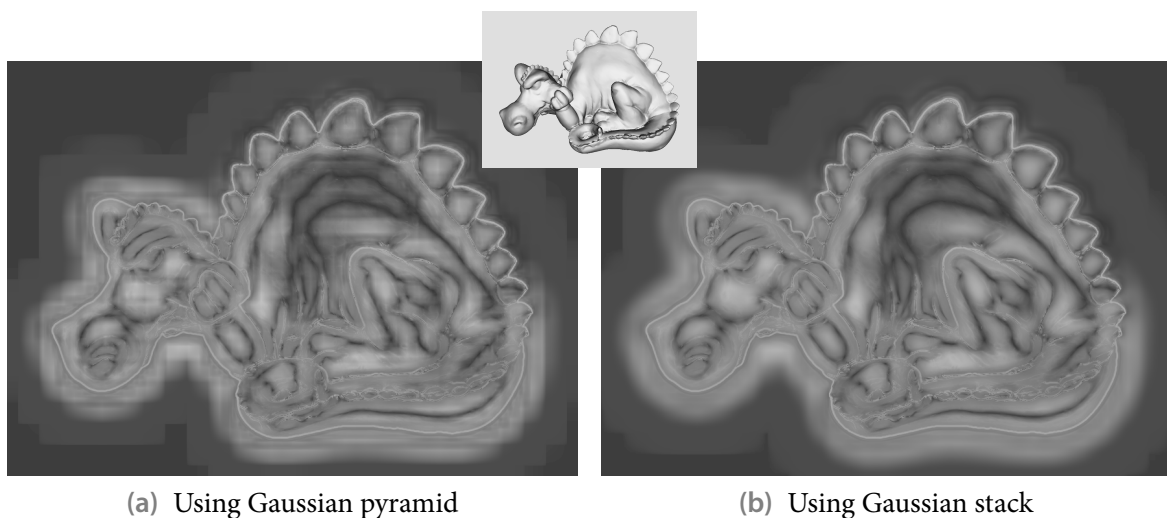


Figure 9.6 Example threshold map. Note the blocky artifacts that occur when directly sampling the Gaussian pyramid.

| Resolution | 16-bit RTs | | 32-bit RTs | |
|------------|------------|---------|------------|---------|
| | Pyramid | Stack | Pyramid | Stack |
| 512×512 | 0.24 ms | 0.33 ms | 0.26 ms | 0.36 ms |
| 1024×768 | 0.63 ms | 0.76 ms | 0.66 ms | 0.85 ms |
| 1024×1024 | 0.82 ms | 0.98 ms | 0.85 ms | 1.10 ms |
| 1600×1200 | 1.44 ms | 1.72 ms | 1.51 ms | 1.94 ms |

Table 9.1 Generation times for threshold maps of various sizes on an NVIDIA GeForce GTX 280, using seven levels for the Gaussian pyramid. Note that the higher quality resulting from using a Gaussian stack instead of directly sampling the Gaussian pyramid incurs an overhead of about 20% to 40%. On the other hand, a speed-up of up to 13% can be achieved by using render targets (RTs) with a floating-point precision of just 16 instead of 32 bits.

one pixel covers roughly one degree of visual field. Subsequently, the spatial threshold elevation factor S is computed according to (9.10). A look-up texture is employed to get the CSF values $\text{csf}_{\text{B89}}(\rho_i, L)$, whereas the masking function T_D is evaluated on the fly. Moreover, unlike in the original formulation by Ramasubramanian et al. [308] in (9.11), we use the actual adaptation luminance L for determining the CSF-based elevation factors F_i , again utilizing a (different channel of the same) look-up texture. Finally, the TVI function is evaluated by consulting another look-up texture, and the obtained threshold is multiplied with S .

Note that when a coarser level of the Gaussian pyramid is accessed, an implicit upsampling to the finest-level resolution is performed via bilinear texture interpolation. While being extremely fast, this approach is not equivalent, however, to successively upsampling by one level a time, which yields smoother results and does not suffer from blocky artifacts. For higher quality, we hence optionally construct a Gaussian stack, stored in a texture array, from the Gaussian pyramid and employ that during threshold computation. To this end, each pyramid level $i \geq 2$ is successively upsampled to the next finer mipmap slice and ultimately to its respective slice of the Gaussian stack. We actually maintain the stack only at the resolution of pyramid level $i = 1$, thus exploiting hardware texture interpolation for the final upsampling step.

As demonstrated by the performance data listed in Table 9.1, our approach enables a rapid threshold map computation, even for large input images and when using a Gaussian stack. In particular, it is well suited to be incorporated in the per-frame workload of real-time rendering applications. An example of computed threshold maps is shown in Fig. 9.6.

9.5 Interactive perceptual rendering pipeline

Recall that in real-time rendering, employing LOD techniques is often essential for achieving a sufficiently high frame rate. Traditionally, the appropriate LOD for a certain object is determined based solely on information about this object, ignoring the potential influence of further scene content. Such an approach is also pursued by the perceptually motivated control methods outlined in Sec. 9.3. However, larger savings in rendering effort are possible by accounting for scene-level effects, like shadows and partial occlusion caused by complex objects such as trees, fences or grids. These can strongly affect discriminability between two LODs of an object and hence often allow significantly lowering the employed LOD without affecting perceived rendering quality.

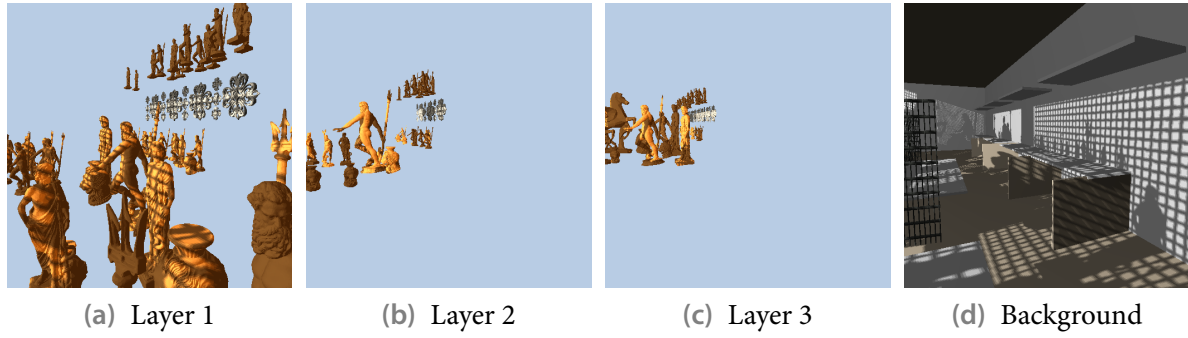


Figure 9.7 Decomposition of the example scene from Fig. 9.8 into separate layers.

A threshold map is an appropriate tool to capture such masking effects. In particular, using our GPU-based variant covered in the preceding section, it can be derived at low costs. This rapid evaluability enabled the development of a perceptual rendering pipeline [98] which actually takes interactions among scene objects into account. To this end, the scene is dynamically decomposed into several layers, as exemplified in Fig. 9.7. While objects for which different LODs are available get assigned to layers based on distance, all remaining scene content is put into a special background layer. Each frame, the scene is at first rendered layer-wise, recording the objects of a certain layer i in an according separate render target O_i . Moreover, a shadow mask is stored for each layer, indicating the pixels where an object of this layer is shadowed by objects from other layers. Then, for each non-background layer i , a combination of all other layers' O_j is computed, incorporating also layer i 's shadow mask. Note that this aggregation C_i essentially represents the scene context for objects of layer i . To quantify the degree of masking introduced by it, a threshold map TM_i is subsequently derived for each such combination C_i .

This information is leveraged to control the employed LODs, aiming for selecting the coarsest LOD which is predicted to be indistinguishable from a fine reference LOD. For an object in level i , its reference LOD is rendered and compared against the currently used LOD, captured in O_i . At each pixel where the object is visible, the determined difference between the two LOD renderings is tested against the threshold map TM_i , counting the number n_{vis} of pixels where the deviation is above threshold via an occlusion query. This quantity is then used to decide whether to maintain, increase or decrease the employed LOD. Given two user-specified threshold δ_l and δ_u , the next lower- or higher-quality LOD is adopted if $n_{\text{vis}} < \delta_l$ or $n_{\text{vis}} > \delta_u$, respectively. Note that each frame, only a subset of all objects is tested. An example of a resulting LOD selection is shown in Fig. 9.8.

To verify that the adopted prediction of difference visibility between two LODs roughly matches a viewer's perception, a small user study was conducted, whose results indicate a reasonably high correlation. This perceptually motivated pipeline for LOD control hence nicely demonstrates the potential of both rapid GPU-based threshold map computations and of exploiting visual perception for scene-level optimizations. Note, however, that the approach suffers from some shortcomings. Most prominently, the thresholds used for comparing the two LODs of an object are derived from a rendering of the scene without this object (barring shadows cast on it). Consequently, both the frequency content and the luminance levels assumed when deriving the thresholds typically don't accurately match the actually encountered ones. Furthermore, temporal aspects of switching the LOD are ignored. Not only is the method prone to causing popping artifacts, but it also does not take the influence of object motion on threshold magnitude into account.

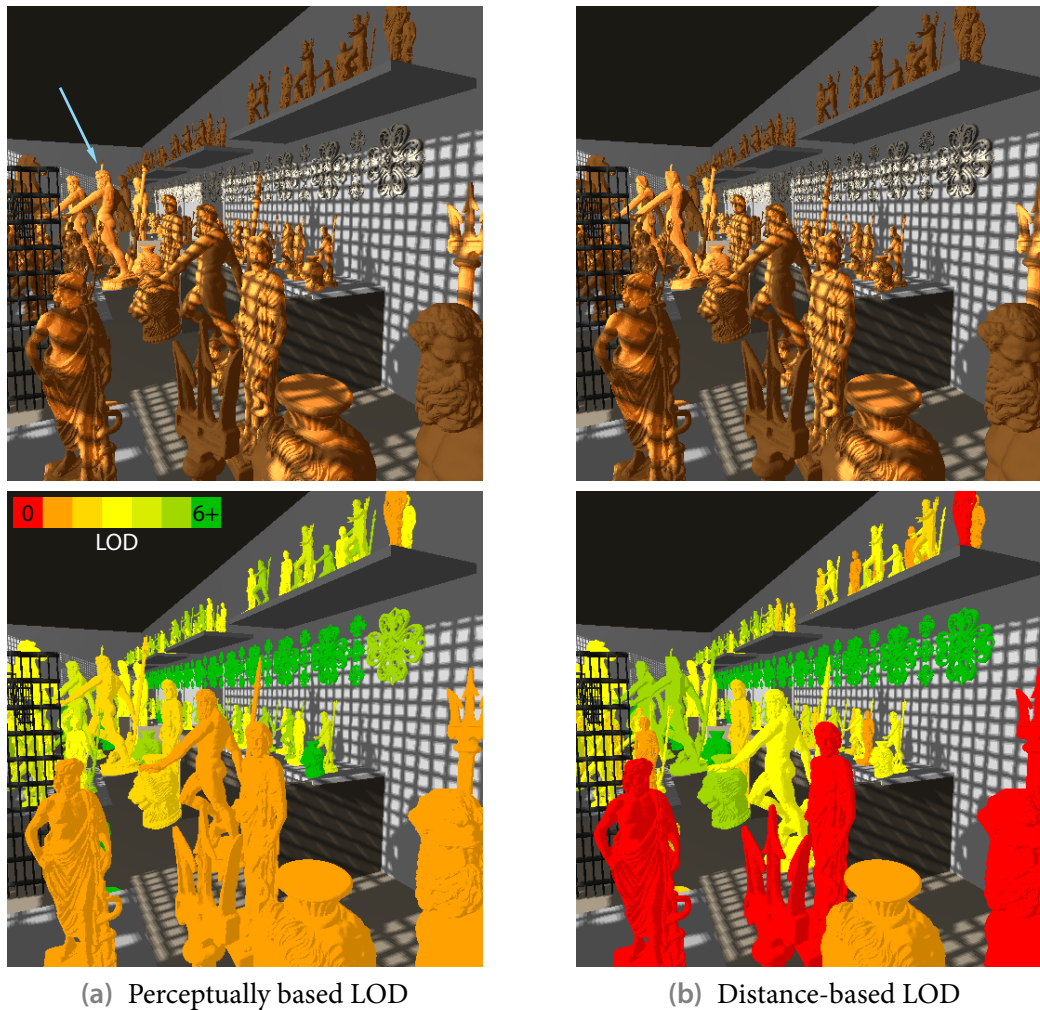


Figure 9.8 Example scene rendered using (a) the perceptual rendering pipeline for LOD control and (b) alternatively a simple distance-based metric, with each approach resulting in roughly the same frame rate. Note that the perceptually motivated variant maintains a higher quality for well-visible statues in the back (like the one indicated by the arrow), enabled by reducing the LOD for front objects where strong visual masking occurs.

9.6 Problems in applying perceptual results

Taking human visual perception into account is a worthwhile goal that promises to spend rendering efforts wiser and hence to achieve a higher realism for a given time and resource budget. On the other hand, as already hinted at throughout this chapter, several problems arise when applying perceptual results.

One major issue that frequently limits their usage, especially in real-time rendering, is that modeling perception introduces a certain overhead, which is often non-negligible. Consequently, the gained savings must be large enough to at least amortize this extra cost. Coming up with efficient and fast GPU-based realizations, like our threshold map variant, is hence one significant step towards lowering the bar for applicability.

Another problem is that models of basic characteristics, like a CSF, result from certain isolated experiments with simple stimuli under distinct artificial conditions. When adopting an

engineering approach and just combining such potentially unrelated modules to construct a more complex perceptual model, as is normally done, inconsistencies in the perceptual foundation may thus easily arise. Note that due to the lack of according experimental data, this situation is usually hardly avoidable. Furthermore, it is unclear how well the models generalize to conditions and stimuli beyond the ones covered in the underlying data. In particular, complex patterns may result in stronger masking effects than observed with superpositions of simple gratings. Therefore, predicted detection thresholds may be way too conservative.

While graphics algorithms strive for exact visibility statements about deviations, a universal point of change in detectability does not exist, thanks to inter- and intra-viewer variations. Hence, recall that established perceptual thresholds only correspond to a certain probability of detection. Moreover, such thresholds depend on the actual viewing setup. Note, for instance, that the absolute spatial frequencies of an image are a function of viewer distance. Although such parameters can be controlled during an experiment, it is typically impractical to determine and specify them accurately in real-world applications, generally forcing some conservative approximations.

Warning examples

Issues like these often make it hard to verify whether perceptual results are correctly applied. Consequently, observed improvements attributed to following perceptual principles may actually turn out to lack such a foundation. For instance, in his threshold map variant, Yee [408] computes a Gaussian stack exactly like a Gaussian pyramid but without downsampling, i.e. level i results from successively applying a fixed-size Gaussian kernel for i times to the finest level. This, however, is not even roughly equivalent to doubling the Gaussian's spread each level. As a consequence, the corresponding Laplacian stack exhibits a severely different frequency selectivity than usual. Nevertheless, Yee assumes the peak frequencies of an ordinary Laplacian stack when evaluating the CSF and hence wrongly applies perceptual results. Despite this deficiency, the metric was reportedly used successfully in the movie industry.⁵

Another example where a realization mishap occurred is the threshold map implementation by Ramasubramanian et al. [307, 308]. They intend to determine the luminance level for the TVI function at a certain pixel by averaging pixel values from a neighborhood covering one degree of visual angle. To this end, a formula from Ward et al. [394] is applied which for a given field of view yields the resolution of an image where one pixel subtends one degree. However, they actually take this quantity as size of the averaging filter kernel, which in their setup causes the luminance level to be determined from a region covering significantly less than one degree. Note that this effect is quite noticeable in the published visualization examples of the TVI thresholds, where blurring is decidedly too weak.

Perceptual results can also easily become void by misinterpreting their specification, which may be hard to discover. As an example, when adopting Kelly's CSF [183] from (9.3), which is originally given in terms of a parameter $\alpha = 2\pi\rho$ rather than of spatial frequency ρ , Reddy [240, 310] confused α with ρ , thus dropping the factor of 2π . Commendably, he noticed that the resulting expression does not match the published data. To alleviate this, Reddy modified the formula accordingly, essentially reintroducing the omitted factor of 2π .

Finally, trying to model some perceptual phenomenon but doing so poorly can yield worse results than not accounting for it at all. For instance, the Nayatani color appearance model [112]

⁵Our own test using the publicly available implementation (<http://pdiff.sourceforge.net/>) was a complete failure, though.

is able to predict both the Stevens and the Hunt effect, i.e. a gain in brightness and chromatic contrast, respectively, with increasing luminance. However, Fairchild [112] reports that for data sets testing these effects the Nayatani model only performs as good as or even worse than simpler models which don't account for the effects at all.

CHAPTER 10

Visual popping

In real-time rendering, level-of-detail techniques are prominently used and often prove central to achieve high frame rates. However, adapting an entity's LOD may not go unnoticed and give rise to visual popping artifacts. This is highly problematic because their occurrence can severely degrade the perceived degree of realism. While approaches exist for several LOD schemes to alleviate popping artifacts or even to suppress them by conservatively selecting the employed LOD, they are ignorant of the actual perceptibility of these artifacts. It is hence desirable for LOD control algorithms to be able to infer whether an envisaged change in LOD can be perceived or not, and also as how disturbing such a switch appears. An important step towards this ideal is a perceptually based metric which predicts the occurrence of popping, indicating its perceived magnitude.

In this chapter, we present a first solution [339] towards the elusive goal of devising such a predictor. After initially reviewing popping and related approaches employed to deal with it, we discuss several aspects affecting popping perception, pointing out its complexity. Subsequently, a practical perceptually motivated predictor for popping artifacts is introduced, which tackles some of the involved issues. Leveraging several simplifying assumptions, it makes heavy use of a spatio-velocity color vision model and condenses the model output in a meaningful way, yielding popping regions (see Fig. 10.1). Examples demonstrating the predictor's concrete application are presented, too. Finally, we describe a user study comprising two experiments which was conducted to evaluate the predictor's performance. The results indicate that our approach makes predictions which are well in line with the subjects' perception.

10.1 Popping and related treatment approaches

When rendering dynamic scenes and hence the image content changes over time, several artifacts can arise in the temporal domain. Apart from aliasing and its most disturbing manifestation, flickering, many of these artifacts are due to *popping*. Popping occurs if the renderer uses two different representations or parameter sets, referred to as levels of detail (cf. Sec. 2.4), for at least one scene entity in two consecutive frames and this results in an abrupt change that gets noticed by the user. Often, such switches are adaptations of an object's geometric LOD, which in general not only influences outer and inner silhouettes but also shading. Popping may also be caused by transitioning from a geometric to an image-based or a point-based representation and vice versa, or by updating an impostor once parallax error or sampling density mismatch exceeds a respective threshold [327]. Other examples comprise changing the complexity of

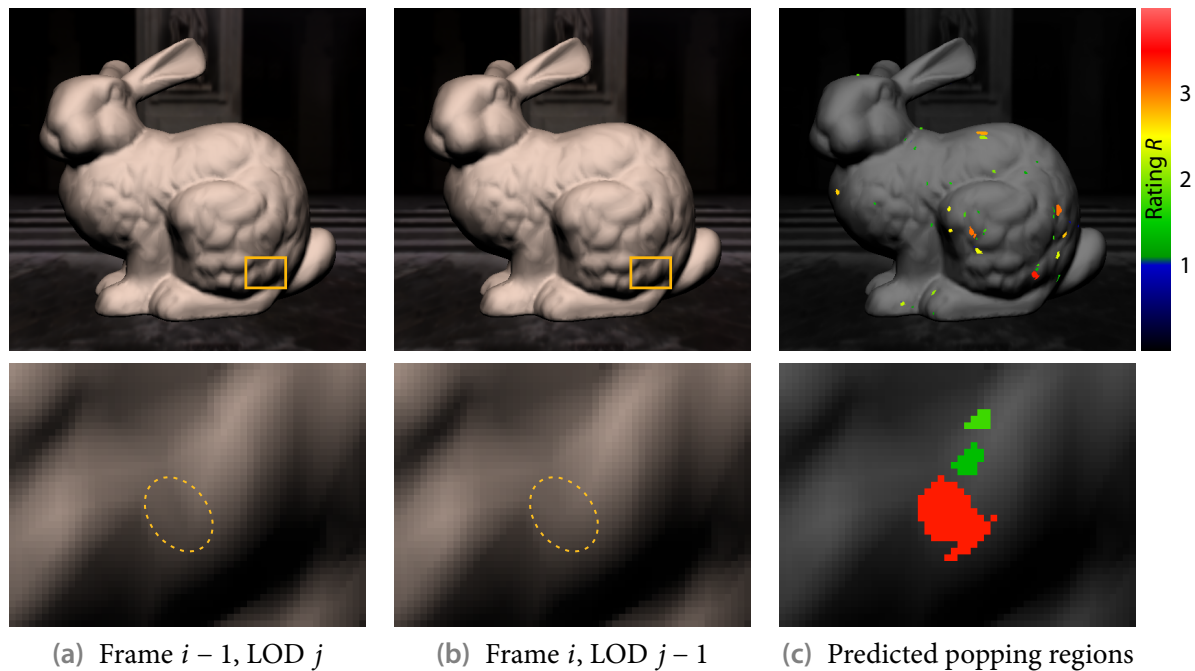


Figure 10.1 When an object is moving and the employed LOD changes, some screen regions may exhibit popping of various severities. For instance, two consecutive frames of an approaching bunny using different geometric LODs are shown (a, b). Popping occurs, among others, in the zoomed-in region. Our perceptually motivated predictor, detailed in Sec. 10.3, identifies such regions suffering from popping artifacts (c).

employed shaders and picking a different set of virtual point lights for approximating indirect illumination.

Many techniques have evolved over time to avoid popping or at least alleviate its severity. One class combats popping by smoothing the transition over several frames, with image-space blending [139] and geomorphing [164] being the most notable approaches. On the downside, however, they introduce additional overhead, often countering the motivation for changing the LOD, increasing performance. Even more problematic, these approaches may cause the transition itself or some of its intermediate states to be perceived as unnatural or even disturbing, essentially trading one problem for a different one.

Another well-adopted option is to utilize a deviation metric and only switch LOD if the predicted deviation stays below a threshold considered acceptable. For geometric LOD, many of these metrics operate on bounds derived in object space and then project them into screen space. The resulting error bound is often considered appropriate in order to avoid popping if it is at most half a pixel. Examples include the texture deviation metric [77] for mesh simplification and the geometric approximation error for adaptive tessellation of higher-order surfaces (see Sec. 7.4.2). Most of these metrics only consider an object's geometry and basically ignore its surface signal. Consequently, they may be too conservative because, for instance, even a large texture-space distortion often remains imperceptible if the texture is uniformly colored or in shadow. On the other hand, small changes in geometry may result in major shading variations, like jumping or vanishing highlights.

To overcome some of these shortcomings, several researchers suggested metrics motivated by perceptual considerations (cf. Sec. 9.3). For instance, the contrast and the change in spa-

tial frequency content induced by a geometric LOD change were employed to predict visual detectability, also accounting for texture content [400]. Using a more involved and computationally expensive vision model, the visual masking potential of an object's surface signal may additionally be regarded [302]. Unfortunately, such approaches mainly target best-effort simplification and don't directly account for popping. On the other hand, Hamill et al. [153] conducted psychophysical experiments for models of buildings and humans, deriving thresholds for the pixel-to-texel ratio at which a change from impostor to geometric representation can be carried out without (disturbing) popping.

In most cases, a LOD switch potentially causing popping is executed because the affected scene entity is moving relative to the viewer. Depending on how fast and in which direction an entity moves, the perceptibility of the LOD change it is subjected to can significantly differ. However, this temporal aspect of popping is basically ignored by all metrics for choosing an appropriate LOD. Although few approaches exist which take object movement into account to select coarser geometric LODs for fast-moving objects [310], they don't consider the switch among two LODs.

Because of the practical importance of popping and the absence of reliable solutions which are not over-conservative, there is a certain need for a perceptually based computational model for predicting whether and where popping occurs in dynamic scenes. One potential application is the derivation of optimal LOD transition points for prerecorded paths in walk-through and fly-over scenarios. Moreover, such an automatic metric may serve as oracle when optimizing parameters or testing LOD schemes. It could also help identifying screen regions where popping is likely to be perceivable. Note, however, that it is not suitable for a per-frame on-the-fly application to guide LOD selection in real-time rendering settings. This is due to the overhead entailed by a perceptually motivated metric that operates in image space to account for the exact context. It hence necessarily requires, among others, a rendering of the current frame using the considered new LOD. Consequently, if the LOD is eventually not changed because it would evoke popping deemed too severe, the scene must actually be rendered a second time for the same frame. Even ignoring further overhead, this already imposes an often unacceptable cost for real-time rendering, countering the hoped-for gain in efficiency by using a coarser LOD.

We developed such a perceptually motivated (off-line) predictor for popping artifacts. Before presenting it in Sec. 10.3, we first review and discuss several aspects involved in perceiving popping in the next section, highlighting the complexity of this phenomenon and why a reliable prediction is extremely hard to achieve.

10.2 Aspects of perceiving popping

The perception of popping turns out to be a very complex phenomenon that is influenced by several factors, many of which are far from being completely understood. In general, popping is perceived if a temporal discontinuity in the image signal occurs that is large enough to be captured by the human visual system and that is then actually detected by the viewer.

Consequently, attention plays a significant role in perceiving popping. Even strong popping may go unnoticed if the viewer's attention is not directed towards the region where it occurs. There is experimental evidence [330] that in case a moving object is pursued, the attention is both focused on this target and its movement direction, causing a loss of sensitivity for both peripheral objects and motion opposite to the pursuit direction. Popping itself may be highly salient and hence attract attention; however, this is mainly true for large-scale popping

involving multi-pixel geometric deviations, which can usually easily be identified by classic non-perceptual metrics. Recall from Sec. 8.4 that while computational models for visual attention exist, incorporating the viewer's experiences and identifying and modeling her adopted task remains a challenge.

Motion perception involves higher-level visual mechanisms and depends partly on more abstract image features like surfaces and objects [393]. Motion introduces some spatial uncertainty about the future location of such features and may also occlude and reveal scene elements, which constitutes another source of uncertainty. Moreover, the HVS has only limited resources and hence each of its receptive fields is sensitive to a range of spatial and temporal frequencies, causing an uncertainty in measuring spatio-temporal signals [136]. Spatial and temporal integration is performed, i.e. each receptive field computes a kind of weighted average of local image signals over a small space-time region, effectively causing a blurring [393]. When the motion flow field is processed, these uncertainties are factored in and the higher-level feature information is taken into account and gets updated. In case of inconsistencies of or temporal discontinuities in the flow field, popping may be detected.

Concerning vision, the sensitivity of contrast detection and discrimination shows both intra- and inter-observer variations and degrades with age [156]. Hence, like with most perception-based aspects, a perfect prediction that applies to everybody is impossible. Only if the sensitivity is high enough, a luminance change or a chromatic shift due to popping can be noticed. Regarding modeling this sensitivity, the higher the desired accuracy, the more dependencies have to be considered. However, too many parameters make a model hard to apply, as several parameter values are difficult to provide. Moreover, experimental data is usually only acquired for a small number of parameters (cf. also Sec. 9.6).

Other artifacts, like aliasing and in particular flickering, can also influence the perception of popping. Not only may they attract the viewer's attention and hence divert it from a region where popping occurs, but they may also mask the actual popping. That is, despite noticing the popping, it is not perceived as popping but attributed to another artifact.

Finally, the display device impacts popping perception. Most notably, the now ubiquitous LCD displays typically suffer from motion blur, mainly because of the employed sample-and-hold technique but also due to their response times [286]. While techniques like flashing backlights are able to alleviate this problem [121], they are not widely utilized yet. Other display characteristics, like the chosen white level, as well as light reflected off of the screen influence visual sensitivity, thus affecting the perceptibility of popping artifacts, too.

10.3 Perceptually motivated popping predictor

Considering all the factors involved in popping perception, an accurate prediction appears to be a goal extremely hard to attain. In particular, higher-level mechanisms play an important role but are challenging to account for. While their influence might be modeled in general, doing so reliably at the pixel level essentially is an open problem, presumably requiring a lot of further vision research.

To make the prediction task more tractable, we hence introduce several simplifying assumptions. Most notably, we ignore temporal integration and consider only the single frame where the popping-prone switch of LOD occurs. To detect temporal discontinuities, we compare the actually rendered frame against a prediction of what the user might expect for this frame by means of a vision model. Differences above a certain magnitude then indicate pop-

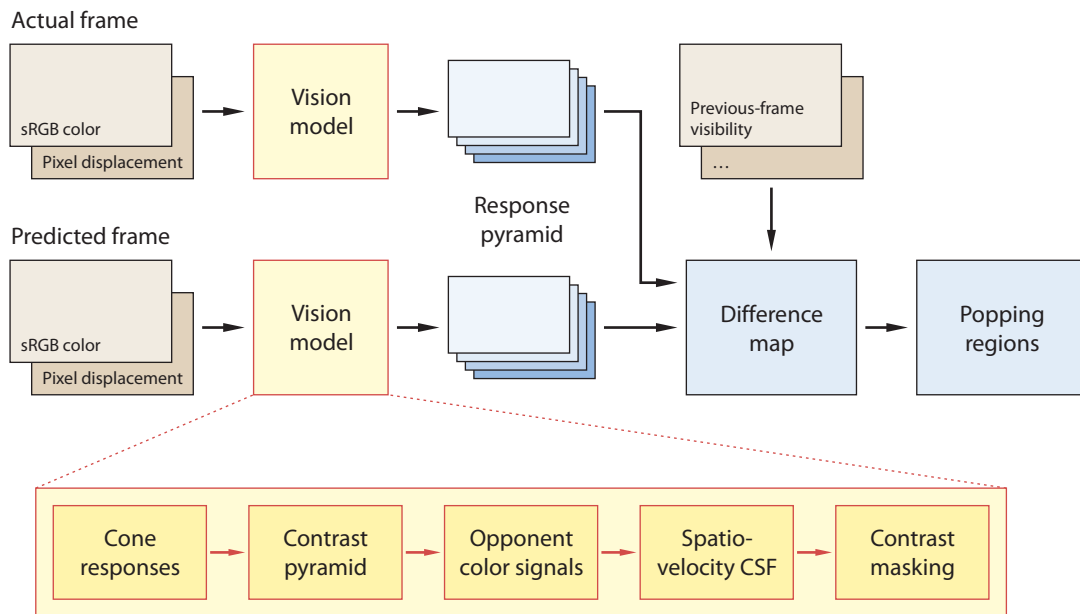


Figure 10.2 Overview of the perceptually motivated popping predictor.

ping. The predicted frame content is obtained by rendering the frame again but utilizing the previous LOD. We hence assume that this way of extrapolating the image content and motion of the previous frames is a good enough approximation for identifying perceived temporal discontinuities resulting in popping artifacts.

10.3.1 Overview

An overview of our predictor is shown in Fig. 10.2. As input, both the actually rendered frame and the predicted frame are provided. Each frame input comprises a color image in sRGB space and a map storing the screen-space displacement of each pixel center with respect to the previous frame. The frame data is subjected to a color vision model, detailed in Sec. 10.3.3, which takes retinal velocity derived from the pixel displacement into account. The model outputs a contrast response pyramid, with its levels corresponding to the spatial frequency decomposition performed by the vision model. Next, the pixel-wise difference between the two input frames' response pyramids is determined across levels and color channels, yielding a difference map. During its computation, additionally provided input for down-weighting differences, like the previous-frame visibility of each pixel, which allows identifying disoccluded pixels, is processed. Finally, connected regions where popping may be perceived, referred to as *popping regions*, are extracted. The whole model output aggregation scheme and its predictive utility are further elaborated on in Sec. 10.3.4.

10.3.2 Discussion

Although higher-level visual mechanisms are not explicitly modeled due to their complexity, the rather simple approach of comparing the actual with the predicted frame accounts for them to a certain degree by indirectly factoring in shape and shading information. Nevertheless, our approach is clearly not appropriate in all cases. For instance, regarding impostor updates, using the previous impostor texture usually doesn't correspond to the user's expectation of how the

previous frame evolved; in contrast, it will probably be a worse match than the new impostor texture due to its larger distortion. On the other hand, transitions from one geometric LOD to another one are rather well captured by our approximation. We believe that while such LOD switches which are amenable to our approach constitute only a subset of all popping-prone LOD changes, they still form a large class of practical importance.

Since we are not modeling most of the uncertainty involved in motion perception, our predictor is slightly too conservative and sometimes wrongly reports a temporal discontinuity which actually gets smoothed out by the visual system. For instance, imagine an object with a curved horizontal silhouette that is approaching the viewer, where every few frames the number of pixels in a scan line covered by the silhouette increases. If this increase is postponed by one frame, often no popping is perceived, while the comparison of our input frames may suggest a popping artifact.

Even though attention is of high importance for perceiving popping, we are not accounting for it. Our algorithm just outputs screen regions where popping is predicted to be perceptible if attention is directed towards them. Note, however, that in principle these regions can easily be checked against the output of a computational attention model. Similarly, we refrain from regarding motion blur inherent to LCD displays, which may lead to some erroneously predicted popping artifacts.

10.3.3 Spatio-velocity color vision model

A computational vision model processes the visual input and yields a response that scales roughly with the perceptibility of the visual contrast stimuli. By comparing the responses for two different inputs, visual differences can be determined. As reviewed in Sec. 9.2, a multitude of vision models have been developed for static images, operating either only on luminance [85, 236] or also on color [41, 235, 289]. Some models for dynamic images which in addition to luminance (but not color) take motion speed into account were also devised [264, 409]. Our vision model is influenced by these approaches and extends them as required by our problem domain, while being comparably cheap to execute.

As input, the model expects a color image in sRGB space as well as a pixel displacement map. First, the color image is converted to absolute CIE XYZ tristimulus values, taking the display's black and white level luminances into account. Then, a transformation to Hunt-Pointer-Estévez cone responses¹ is performed [112]:

$$\begin{pmatrix} L \\ M \\ S \end{pmatrix} = \begin{pmatrix} 0.40024 & 0.70760 & -0.08081 \\ -0.22630 & 1.16532 & 0.04570 \\ 0.0 & 0.0 & 0.91822 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}.$$

Next, we construct a Gaussian pyramid (cf. Sec. 9.1.3) with levels G_i , utilizing a binomial filter kernel of size 5×5 . From this, a contrast pyramid is built which stores local band-limited contrast [236], i.e. level i is computed as $(G_i - G_{i+1})/G_{i+2}$, where coarser levels are appropriately upsampled. Subsequently, the contrast values are converted to Hunt's opponent color space [112]:

$$\begin{pmatrix} A \\ a \\ b \end{pmatrix} = \begin{pmatrix} 2 & 1 & \frac{1}{20} \\ 1 & -\frac{12}{11} & \frac{1}{11} \\ \frac{1}{9} & \frac{1}{9} & -\frac{2}{9} \end{pmatrix} \begin{pmatrix} L \\ M \\ S \end{pmatrix}.$$

¹These correspond to one of the several existing estimates of the cones' spectral responsivities.

A represents the achromatic response; a and b correspond to the red–green and yellow–blue opponent signals, respectively.

The contrast pyramid is then normalized by multiplication with the spatio-velocity CSF. Since sensitivity for fast-moving contrast stimuli is often lower than for static ones, this stage accounts for the observation that visual differences leading to popping artifacts are usually harder to spot for moving objects. The employed CSFs, which are further detailed below, depend on the spatial frequency ρ , the retinal velocity v , and the local adaptation luminance L . For each contrast pyramid level, which essentially represents a spatial frequency band, we take its peak frequency for ρ . The raw velocity v_s is computed from the input pixel displacement map, using parameters of the viewing setup like viewing distance, screen size and resolution. It is then subjected to Daly’s model [86] of unconstrained eye movements, which accounts for the eye’s tracking behavior, to obtain a conservative estimate of the retinal velocity:

$$v = |v_s - \min\{0.82 v_s + 0.15 \text{ deg/s}, 80.0 \text{ deg/s}\}|.$$

Recall that due to drift eye movements, the minimum retinal velocity is in general non-zero. Finally, the adaptation luminance is derived from the Gaussian pyramid level where one pixel roughly corresponds to one degree of visual field.

In a last step, we account for visual masking by applying the transducer function T from (9.9) to the normalized contrast values. Note that T converges to a simple power law for sub-Weber behavior² at suprathreshold contrast levels [212].

Regarding calibration, vision models are usually applied for predicting visual differences among input images shown side by side, whereas in our scenario differences are to be identified when switching between two frames, thus leading to much lower detectability thresholds. We adopted values of 0.5%, 1.0% and 1.2% for the peak contrast sensitivities of the channels A , a , and b , respectively.

Achromatic CSF

For the achromatic channel A , we employ Daly’s spatio-velocity CSF from (9.4). As shown in Fig. 10.3, for increasing velocities the CSF’s band-pass shape moves towards lower spatial frequencies and the peak sensitivity eventually drops.

Recall that the CSF doesn’t directly model the dependence on the adaptation luminance level L , but merely features three parameters c_i for fine-tuning, where Daly suggests setting $c_0 = 1.14$, $c_1 = 0.67$ and $c_2 = 1.7$ for $L = 100 \text{ cd/m}^2$. To remedy this, we make both the peak sensitivity scale factor

$$c_0(L) = 1.14 \cdot \frac{\max_{\rho} \text{csf}_{\text{B03}}(\rho, L, A)}{\max_{\rho} \text{csf}_{\text{B03}}(\rho, 100 \text{ cd/m}^2, A)}$$

and the spatial frequency scale factor

$$c_1(L) = 0.67 \cdot \frac{\arg \max_{\rho} \text{csf}_{\text{B03}}(\rho, 100 \text{ cd/m}^2, A)}{\arg \max_{\rho} \text{csf}_{\text{B03}}(\rho, L, A)},$$

which controls the shift of peak sensitivity along the frequency axis, a function of L by utilizing Barten’s spatial CSF $\text{csf}_{\text{B03}}(\rho, L, A)$ from (9.2).

²That is, the output compressed contrast rises slower than if Weber’s law applied.

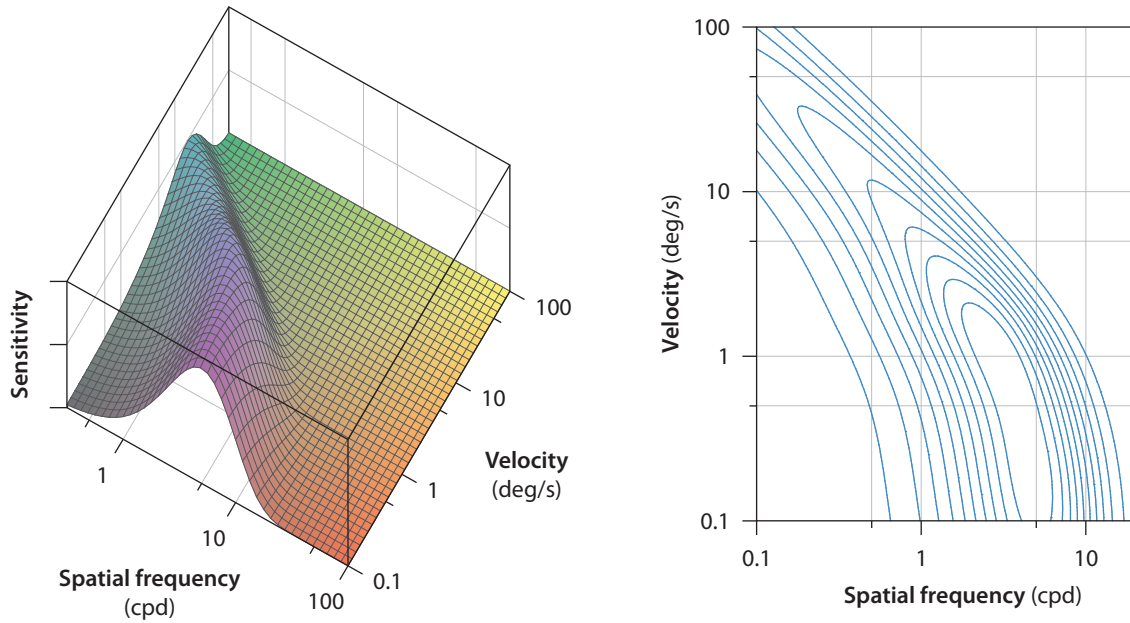


Figure 10.3 Achromatic spatio-velocity CSF at adaptation luminance level $L = 100 \text{ cd/m}^2$.

Chromatic CSF

For both chromatic channels a and b , we adopt Kelly's CSF from (9.5). With increasing velocity, its low-pass nature becomes more pronounced and its peak sensitivity rises, as depicted in Fig. 10.4.

Again, the CSF doesn't model dependence on the luminance level. However, experiments indicate that the threshold contrast decreases inversely proportionally to the square root of the retinal illuminance [381], with retinal illuminance being related to luminance by the pupil's area. To be consistent with the assumptions in Barten's luminance CSF, used to adapt the achromatic CSF to varying light levels, we compute the pupil's diameter by Le Grand's approximation [208]:

$$d(L) = 5 - 3 \tanh(0.4 \log_{10} L).$$

The chromatic CSF is then scaled by the square root of the ratio of the retinal illuminances corresponding to L and the reference luminance (roughly 35 cd/m^2).

Contrast pyramid levels

Each level of the contrast pyramid is tuned to a certain band of spatial frequencies, which results from subtracting two band-limited levels of the Gaussian pyramid. Note, however, that the repeated filtering with a fixed-size Gaussian and downsampling does not exactly yield the frequency response of Gaussian filtering with a spread being doubled every level. Therefore, and especially because the finest level of the Gaussian pyramid is only band-limited by the sampling frequency, the common assumption that the peak frequencies of the contrast pyramid levels halve with every coarser level is not true. Most notably, the finest level's peak frequency is almost four times as high as that of the second-finest level. In our implementation, we account for both this irregularity as well as the amplitude loss due to filtering. We choose the number of levels such that the coarsest level has a peak frequency of at least 0.5 cpd , which results in a five-level contrast pyramid for our viewing setup.

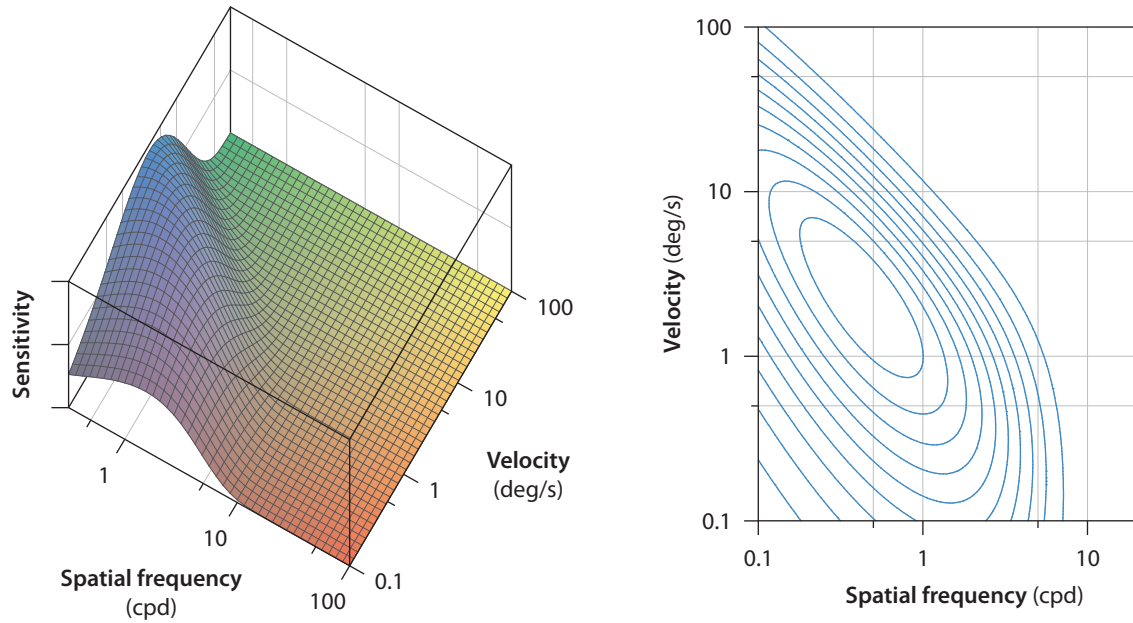


Figure 10.4 Chromatic spatio-velocity CSE.

10.3.4 Popping regions

For both the actual and the predicted frame, the vision model yields a contrast response pyramid. To derive visual differences, we subtract these pyramids and collapse the resulting difference pyramid, combining differences across levels and channels (A , a , b) by Minkowski summation with an exponent of 2.4 [236, 397]. The obtained difference map indicates for each pixel the probability of being able to detect a difference in units of JNDs.

Recall that factors like disocclusion introduce uncertainty and hence differences between the actual frame and its prediction that occur at pixels affected by such uncertainties are less likely to be detected. To account for this, we take a practical approach and scale down the corresponding values in the difference map by weights provided as additional input to our predictor. For instance, the previous-frame visibility of the current frame's pixel centers may constitute one such weight.

While a difference map is of certain utility itself, the contained information should be aggregated in a meaningful way for further analysis. Standard measures like number of pixel differences above threshold, maximum difference, average and variance are usually of limited use because they are too coarse-grained. We hence adopt a different approach, which is based on the observation that not only difference magnitude but also spatial context is important for detection [44]. Intuitively, even smaller visual differences may be easily detected if the affected pixels are clustered together and cover a larger screen region. On the other hand, if a visual difference occurs at an isolated pixel, its magnitude must be rather large to spot the difference.

To model this, we first identify all pixels where the two input color images differ and the visual difference map reports a value of at least 2 JNDs. We then start growing regions around these seed pixels, successively considering all eight direct neighbor pixels and including those with visual difference values of again at least 2 JNDs. The empirically chosen threshold of 2 JNDs accounts for the fact that differences are harder to detect in complex images than in case of simple gratings (typically employed in vision experiments for determining sensitivity). This procedure finally yields a number of popping regions, identifying those parts of the

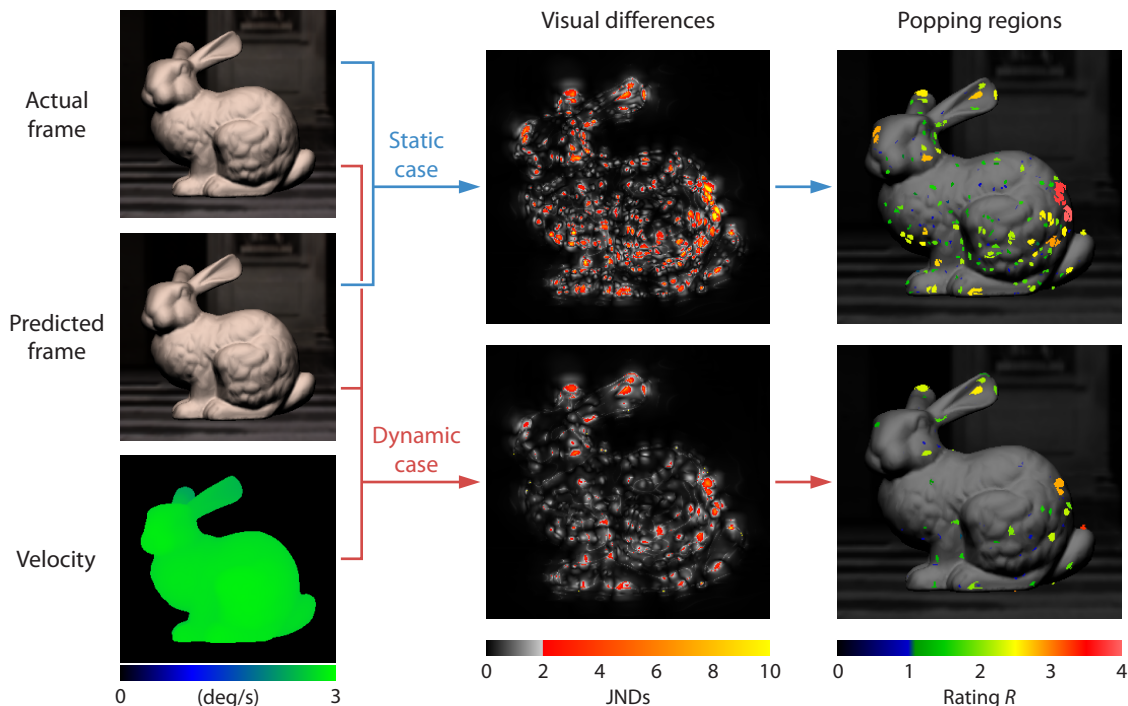


Figure 10.5 The popping predictor applied to a concrete example. From the visual difference map, more meaningful popping regions are extracted.

image where popping artifacts can be expected. For each region, we acquire statistics, like its size in number of pixels, and determine the Minkowski sum (with an exponent of 2.4) of the visual difference values at its pixels. The magnitude of this sum is a good indicator of how severe popping occurs in the region. If we further subject it to the empirically derived mapping $R(\Sigma) = \ln(0.375\Sigma)/\ln(2.25)$, we obtain a simple rating R , where values of $R < 1$ predict rarely visible popping and $R > 3$ suggests easily detectable popping.

An in-context visualization of the popping regions colored according to their rating values (see Fig. 10.5 for an example) allows fast identification of where popping artifacts of which degree can be expected. Moreover, the popping region information is well suited for further automatic processing. For instance, given a screen region of high importance, possibly provided by a computational attention model, it could be checked whether any popping regions are located in this screen region and, if so, how many pixels they cover and what their ratings are. Based on this, an informed decision whether the potential popping artifacts can be considered acceptable or not for the given application can automatically be made.

Fig. 10.5 shows a concrete example. Please recall that differences are harder to spot when viewing images side by side. Thanks to the selective aggregation performed, popping regions are a useful tool for analyzing the visual difference map and for identifying and rating popping artifacts. Moreover, note that the visual differences' magnitude is clearly affected by fast motion.

10.3.5 Examples

We applied our popping predictor to two different examples, chosen to be representative of possible real-world applications: an object-wise geometric LOD and a simple terrain LOD (see Fig. 10.6). In the first example, we constructed coarser concrete LODs via the progressive mesh

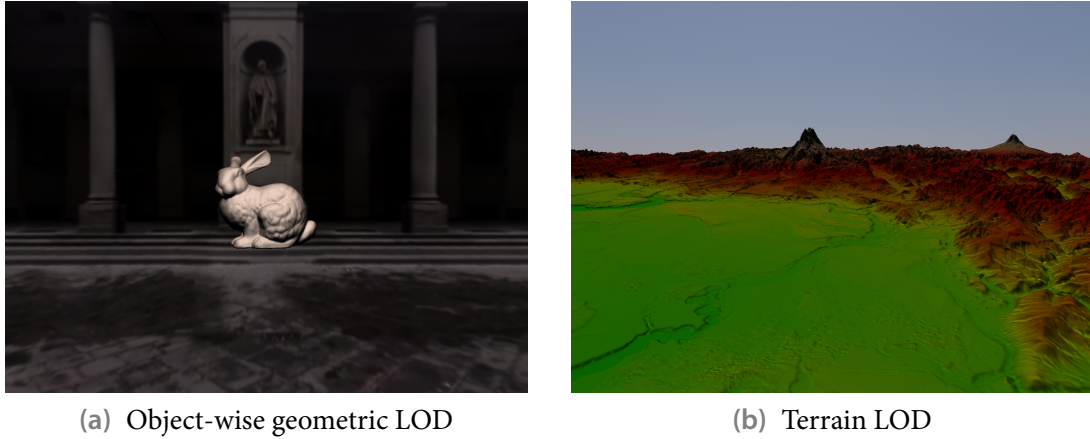


Figure 10.6 Screenshots of two examples to which we applied our predictor.

implementation of Direct3D 9 and manually specified distances at which to switch LODs. To obtain the required input for our predictor, we render the frame in question twice, once with the new and once with the previous LOD. Apart from the pixel color, we further derive for each fragment the previous-frame screen location of its corresponding point and store the resulting screen-space displacement. To account for disocclusion, for both of the involved LODs we determine the depth map for the previous frame setting and perform a depth comparison with percentage-closer filtering to derive a real-valued previous-frame visibility factor per pixel. Finally, being conservative, we take the pixel-wise maximum of these factors and provide the resulting weight map as further input to the predictor.

For the terrain application, we adopted a simple chunk LOD approach [380], where coarser-level terrain tiles are generated by regular subsampling. LOD switches are controlled by the screen-projected maximum height deviation from the finest-level terrain geometry. For shading, we resort to ambient aperture lighting [275], which allows easy adaptation to various times of day and hence sun positions. The input data for our popping predictor is obtained like in the first example.

However, since terrain fly-overs often suffer from strong flickering mainly at distant mountain ranges that can mask popping, we additionally incorporate a map for weighting down visual differences in flicker-affected regions. To detect flickering, we resort to a simple heuristic. For each pixel, we compute its shading for the previous-frame setup and compare it against the color obtained by sampling the previous frame's color image, taking depth discontinuities into account. If these two colors have a CIELAB ΔE_{94}^* difference value (computed using the CIE94 formula [112]) that is roughly as large as or even larger than the color difference between the two LOD renderings of the current frame, we assume flickering to occur, unless the pixel's inter-frame screen-space displacement is large.

10.4 User study

Given the simplifying assumptions and empirical choices made, we considered it important to conduct a user study to investigate the plausibility of our approach and its predictions. However, experimental validation turns out to be challenging for multiple reasons. In practice, a LOD change usually results in several popping regions. But because popping occurs at a single

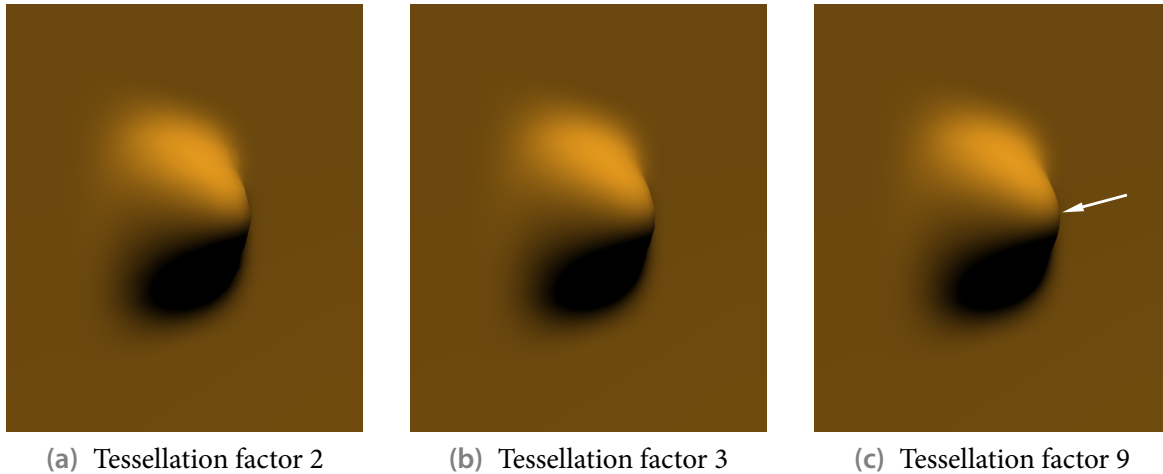


Figure 10.7 The three LODs of the patch employed in the first experiment. The arrow indicates the tip to which the subject is asked to attend.

point in time, a subject can only spot and attend to at most one region (or maybe a few small and closely clustered ones), but misses processing all the other ones. Moreover, it is hard to determine where a subject directed its attention to. On the other hand, attention can only be guided to a certain degree and accuracy, especially in case of complex stimuli. Consequently, validating all predicted popping regions directly in ecological settings is an elusive task.

Another major obstacle is the huge space of possible LOD transitions that could result in popping, and its high dimensionality. In particular, perceptibility of popping artifacts is influenced by the involved objects (shape and its complexity, material), their environment (lighting, complexity), the LODs used, the chosen transition point (e.g. certain distance), and the kind, direction and speed of the object's movement relative to the camera. Therefore, any test necessarily has to concentrate on few samples of the LOD transition space.

We address these challenges by two different experiments. In the first one (Sec. 10.4.1), we seek to directly evaluate the predictive power of single popping regions. To this end, we focus on a simple object that allows directing a participant's attention to a specific region. Testing all combinations of two LOD sets, multiple transition points and two movement speeds while fixing all other degrees of possible variation, we densely sample a small subspace of all LOD transitions. In contrast, the second experiment (Sec. 10.4.2) deals with a larger subspace of the LOD transition space, but samples it only sparsely. Utilizing our two example applications, it also considers more natural and complex situations. Since attention cannot really be controlled in the setup, no validation of single popping regions is possible this time. Instead, an indirect evaluation of the overall prediction of all popping regions is performed.

10.4.1 Experiment I: direct evaluation with simple object

For the first experiment, we adapted our geometric LOD example, using a simple bicubic B-spline patch (see Fig. 10.7) instead of a mesh. Different LODs are obtained by varying the tessellation level. Normals are computed per pixel by directly evaluating the patch's derivatives. We show 4.5-second sequences of the camera approaching the patch, during each of which the LOD is changed exactly once. Two LOD sets (tessellation factor $2 \rightarrow 3$ and $3 \rightarrow 9$) and seven distances at which the LOD is switched are considered. In addition to showing these scenarios

as an animation (dynamic case), we also present just the frame where the LOD transition occurs, initially with the old and then with the new LOD (static case). Each sequence is shown three times, as well as one extra time without any change in LOD to verify that a subject indeed perceives popping and is not giving random answers.

The stimuli are presented on a dark-grey background in the central 1024×768 region of a 20" FSC P20-2 LCD display with a resolution of 1600×1200. The participant, being sat at a viewing distance of 60 cm, is instructed to attend to the tip of the patch. After each sequence, he is asked whether popping was noticed. In total, 112 sequences are presented in random order, with the dynamic scenarios preceding the static ones; a whole session lasts less than 20 minutes. To make the subject familiar with the task and the voting interface, an exercise session is run before the actual experiment.

Eight subjects with normal or corrected-to-normal vision, all of them members of the Computer Graphics Group of the University of Erlangen-Nuremberg, participated in the experiment. The mean of the subjects' average popping detection rates is 67.71% (stdev: 12.01%) when LOD switches occurred but only 4.46% (4.58%) in absence of a LOD transition (and hence popping), indicating the plausibility of the answers. A repeated-measures ANOVA applied to the cases where the LOD was changed shows a main effect of the employed LOD set ($F(1, 7) = 98.926$; $p < 0.0001$) and of the switch distance ($F(6, 42) = 30.226$; $p < 0.0001$), as well as an interaction of these two factors ($F(6, 42) = 3.139$; $p = 0.012$). Moreover, there is also a main effect of whether the stimulus is static or dynamic ($F(1, 7) = 29.377$; $p < 0.001$).

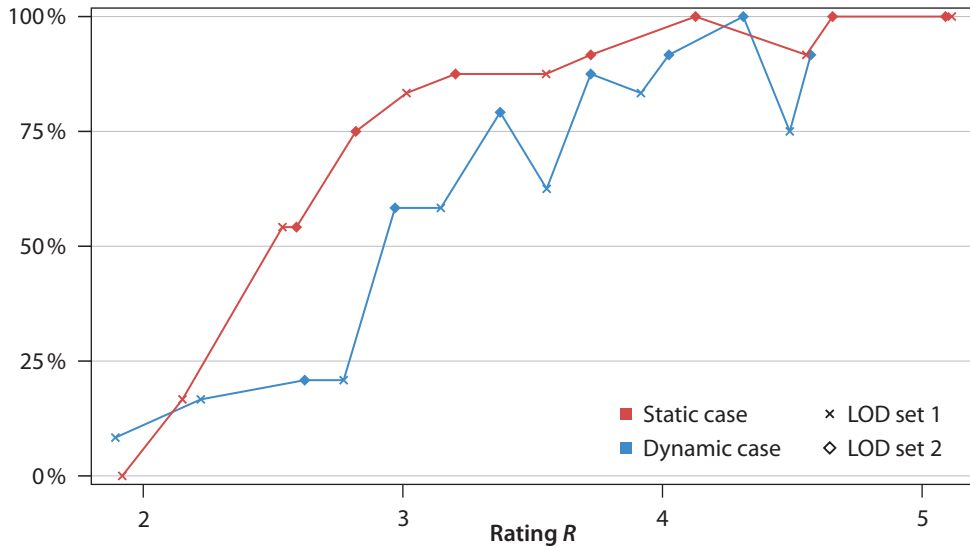
To evaluate our predictor, we consider the single popping region at the tip of the patch. For the static case, this region's rating value R shows an almost monotonic relationship with the average popping detection rate (cf. Fig. 10.8). This is also reflected in a high Kendall rank correlation coefficient³ $\tau_c = 0.933$. To compute the Pearson product-moment correlation coefficient⁴ r , we first clamp R to the lowest value where a 100% detection performance is encountered, accounting for the detection rate's upper bound of 100%. The value of $r = 0.915$ indicates a rather highly linear relationship.

As a sanity check, a comparison with another metric's performance is desirable. In particular, it is advisable to test whether the complexity of the predictor is justified and the obtained results are really superior compared to those of a much simpler and cheaper approach. However, being not aware of any alternative popping predictor, the best we can do is to adopt our approach of comparing the actual frame with its prediction using the previous LOD and employ some image-space metric to compute their difference. Unfortunately, it is also not obvious how to reasonably aggregate such a metric's pixel-wise output. Therefore, opting for simplicity, we chose the maximum CIELAB ΔE_{94}^* difference value. It shows a weaker correlation ($\tau_c = 0.723$, $r = 0.806$) to the subjects' average detection rate. In particular, unlike our predictor, this metric only reasonably orders scenarios which use the same LOD set but fails to correctly rank them across LOD sets (cf. Fig. 10.8).

In the dynamic case, our predictor's output still shows a high correlation to the average detection rate ($\tau_c = 0.808$, $r = 0.932$). It also performs far better than maximum CIELAB ΔE_{94}^* ($\tau_c = 0.561$, $r = 0.747$), which doesn't account for object motion. Compared to the static case,

³A rank correlation coefficient is a statistical measure, which quantifies the association between two rankings of the same data set. In our case, these rankings result from sorting the scenarios in ascending order according to rating R and to detection rate, respectively. A value of one indicates perfect agreement among the two rankings, while zero corresponds to complete independence.

⁴It measures the strength of linear relationship between two interval-scaled quantities X and Y , where $r = 1$ means that Y is a linear function of X , and $r = 0$ indicates lack of any correlation.



(a) Popping predictor

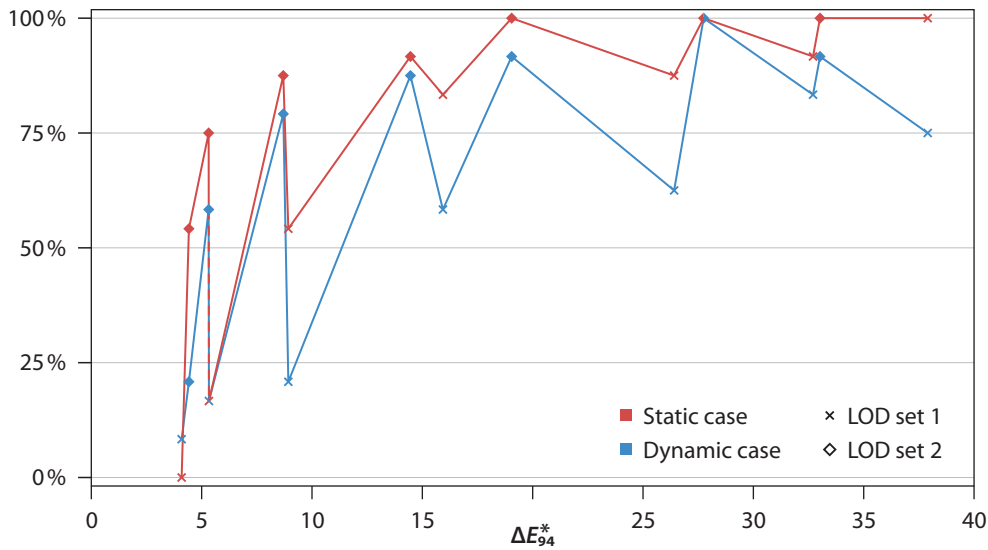
(b) Maximum CIELAB ΔE_{94}^*

Figure 10.8 Average observed popping detection rate in the first experiment as a function of (a) the predictor's rating R , and of (b) the maximum CIELAB ΔE_{94}^* value.

however, our predictor slightly overestimates the subjects' detection performance. We partially attribute this to attentional effects; tracking of the patch's tip has to be performed, which complicates focusing attention. Also, LCD motion blur, which we don't account for, might lower the perceptibility. Nevertheless, when considering both static and dynamic cases together, correlation is still reasonably high ($\tau_c = 0.790$, $r = 0.902$), and better than in case of maximum CIELAB ΔE_{94}^* ($\tau_c = 0.589$, $r = 0.786$).

10.4.2 Experiment II: indirect evaluation with real-world examples

In the second experiment, we show four-second sequences of the two example applications (cf. Sec. 10.3.5), where exactly one LOD switch occurs in each. Unless otherwise noted, we use the same setup and procedure as in the first experiment. Twelve subjects with normal or corrected-to-normal vision, again all members of the Computer Graphics Group of the University of Erlangen-Nuremberg, participated. In total, each of them is shown 64 stimuli in randomized order; a whole session lasts about 25 minutes.

Object-wise geometric LOD

In the first part of the experiment, we use the Stanford bunny for the object-wise geometric LOD. We consider eight different scenarios, varying the movement velocity and the employed LODs (the coarser LOD features between 3% and 88% fewer triangles than the finer LOD). Moreover, different initial locations and movement directions are chosen. The bunny moves either horizontally across screen or towards the user. The LOD is switched after a certain horizontal distance has been covered or the distance to the camera has fallen below a threshold, respectively. As in experiment I, for each scenario, we additionally include the corresponding static case and, for testing for subject reliability, consider each sequence also without changing the LOD. Altogether, each scenario is hence presented to a subject in four instances (dynamic/static, with/without LOD change).

The participant is essentially freely viewing the object and not told a specific region to which to direct its attention. To increase the chance that the subject attends a region where popping occurs, we show each dynamic scenario instance three times and each static one twice, with a one-second gray interval between the repetitions. After all repetitions of a stimulus have been presented, the subject is asked to vote whether popping was perceived and, if yes, to rate the strongest detected popping artifact on a three-level scale (hardly ... clearly visible).

The mean of the subjects' average popping detection rates is 61.46% (stdev: 16.61%) when the LOD was changed and 4.17% (4.87%) in case of no LOD switches, suggesting that popping artifacts were indeed perceived by all subjects. For the cases with LOD changes, a repeated-measures ANOVA shows a main effect of whether the stimulus is static or dynamic ($F(1, 11) = 5.337$; $p = 0.041$) and of the scenario ($F(7, 77) = 18.222$; $p < 0.0001$) on detection performance.

Concerning the evaluation of our predictor's output, comprising several popping regions for each scenario, note that we cannot predict whether one of these regions is attended to and especially not which one. However, assuming that our predictor works, we can reasonably expect that the chance of attending to any of the predicted popping regions increases as the object's coverage with popping regions grows. Moreover, the larger the ratings R of the predicted regions, the higher the chance of detecting popping when attending to a popping region. Adopting this reasoning, we assign to each output of our predictor both a coverage score (four levels: tiny, small, large, huge) and a rating score (four levels: very low, low, high, very high), reflecting the average rating R of the most highly rated popping regions. We then derive an integer detection score (1 ... 5) according to a rule table (cf. Table 10.1).

For the obtained detection scores, we observe a high rank correlation to the subjects' average detection rates for the static case ($\tau_c = 0.833$), the dynamic case ($\tau_c = 0.917$), as well as both cases together ($\tau_c = 0.830$). Treating the score as interval-scaled, Pearson's r suggests a highly linear relationship (static: $r = 0.922$; dynamic: $r = 0.929$; both: $r = 0.927$).

| | | Coverage score | | | |
|-------------------------|-----------|----------------|-------|-------|------|
| | | tiny | small | large | huge |
| Rating score | very low | 1 | 1 | 2 | 2 |
| | low | 1 | 2 | 3 | 3 |
| | high | 2 | 3 | 4 | 5 |
| | very high | 3 | 3 | 5 | 5 |

Table 10.1 Rule table employed for mapping coverage and rating score to a detection score.

Further analysis shows that there is also a lower, but still distinct correlation between coverage score and detection rate (static: $\tau_c = 0.750$, $r = 0.816$; dynamic: $\tau_c = 0.792$, $r = 0.860$; both: $\tau_c = 0.750$, $r = 0.848$). In addition, the rating score correlates well to the subjects' average rating of how strong they perceived a detected popping artifact (static: $\tau_c = 0.938$, $r = 0.916$; dynamic: $\tau_c = 0.750$, $r = 0.829$; both: $\tau_c = 0.781$, $r = 0.878$). Overall, we reckon that these distinct relationships are an encouraging indication that our predictor works well.

Terrain LOD

In the second part of the experiment, our terrain LOD example is used, showing a fly-over. We again consider eight scenarios with varying flying speeds; in each, a different terrain region is subjected to a LOD switch. Moreover, shading is altered by imposing different times of day. The LOD is changed after a certain distance has been covered, affecting only a single terrain tile. As before, each scenario is presented in four different instances (dynamic/static, with/without LOD change). We also adopt the same stimulus presentation and voting procedure as in the first part. However, since the terrain sequences are much more complex and popping can only occur at a rather small region (a single tile), the likelihood that the subject directs its attention towards the occurrence of popping is far too low, as also indicated by a pretest. To address this, we highlight a circular region, with a radius of about 50 pixels on average, to which the user should attend to for two seconds before each trial.

The mean of the subjects' average detection rates is 67.71% (stdev: 33.59%) for the instances with a LOD change and 3.13% (4.21%) otherwise, again indicating that no random answers were given. A repeated-measures ANOVA for the cases with LOD changes shows a main effect of whether the stimulus is static or dynamic ($F(1, 11) = 5.337$; $p = 0.004$) on detection rate, but not of the scenario ($p = 0.170$).

In the static case, our predictor's output always yields a coverage score of at least "large" and a rating score of at least "high". On the other hand, the mean of the scenarios' average detection rates is 75.0% (stdev: 8.33%) and the mean of the average severity ratings on the three-level scale is 2.20 (stdev: 0.24). That is, well-visible popping didn't get noticed by everyone, which we attribute to inattentional blindness. For instance, even a whole mountain tip popping in got missed by two subjects. Therefore, we feel that the overall voting result is well captured by our predictor's output.

In the dynamic case, we observe a larger variation of coverage and rating scores as well as in subject response. However, given that attention was guided to the affected terrain region and that there are three repetitions to detect popping, we expect coverage score to play a minor role. It is thus not surprising that coverage score is only weakly correlated to the subjects' average

detection rate ($\tau_c = 0.234$, $r = 0.285$), whereas, on the other hand, the rating score shows a distinct relationship with the detection rate ($\tau_c = 0.750$, $r = 0.811$).

10.4.3 Conclusion

Overall, both the presented indirect evaluation of our predictor for the second experiment and the direct evaluation of a single popping region in the first experiment indicate that our approach yields plausible and useful predictions of popping perceptibility. In particular, we consider the good correlations between our predictions and the subjects' votings to be very encouraging. This is especially true given the simplifying assumptions made and the influence of attentional effects. Not least due to them, our predictor is however still far from offering a complete and general solution, but constitutes a promising first step.

CHAPTER 11

Conclusion

In this thesis, we have considered three different topics that have a high potential for enhancing realism in real-time rendering. At first, soft shadows were covered. Adopting the general soft shadow mapping algorithm with occluder backprojection as basis, we presented several new approaches and techniques concerning diverse aspects, like acceleration structures and occluder approximations, improving on attainable visual quality and performance. In particular, occlusion bitmasks were introduced, which provide a robust, sample-based solution to the long-standing occluder fusion problem. Moreover, we discussed soft shadow level of quality and proposed a practical scheme for smooth quality variation.

Subsequently, we have been concerned with curved surfaces and according rendering methods, concentrating mainly on adaptive tessellation. Besides investigating recursive refinement, we discussed and presented contributions to tessellation patterns, deriving tessellation factors, and rendering refinement patterns. Most notably, however, the patch-parallel CudaTess framework, which allows efficiently running all major steps on the GPU, was introduced.

Finally, human visual perception and leveraging its characteristics and limitations have been dealt with. We presented a rapid GPU-based variant of threshold maps, which enables the use of this perceptually based image metric, and hence accounting for perceptual sensitivity, during rendering, as demonstrated by a pipeline for LOD control that exploits scene-level visual masking. Furthermore, a perceptually motivated predictor for visual popping, which employs a vision model to identify popping-prone screen regions and estimate the severity of the occurring popping artifacts, was proposed and evaluated in a user study.

Outlook

Note that attainable realism in real-time rendering is not only a matter of visual quality and the accuracy of the employed models and approximations but always also of performance. Achievable performance, however, depends on the available raw computational power, the hardware features and the offered possibilities to map a given algorithm to them. But graphics hardware keeps advancing rapidly regarding provided computational power, and functionality offered and exposed to the programmer is growing continuously, with Direct3D 11 [252], for example, introducing new pipeline stages and compute shaders. Furthermore, if Intel's upcoming Larrabee chip [349] is any indication, future generations of graphics hardware will offer significantly more flexibility and programmability.

Consequently, methods and strategies currently not feasible may eventually offer simpler and better solutions than the presented GPU-based techniques, which, however, provide a good foundation for further improvements. For example, our approach of sampling light visibility

and tracking it with an occlusion bitmask is equally appropriate for determining accurate soft shadows by working directly with the real occluder geometry instead of with a shadow map approximation. Similarly, our method for dynamically generating non-uniform amounts of geometry purely on the GPU, introduced within the CudaTess framework, is well suited for near-future graphics hardware and also useful for other applications beyond tessellation.

Regarding future challenges, providing accurate and not just physically plausible soft shadows for complex scenes in real time is one major open problem, whose solution probably necessitates some future graphics hardware. Another issue is predicting and avoiding popping artifacts reliably, with our perceptually motivated predictor constituting a promising first step. Furthermore, while individual solutions exist for many aspects of improving realism, combining and integrating them to one comprehensive rendering approach, especially to one that still achieves real-time performance, remains a significant challenge.

Bibliography

- [1] Abert, O., Geimer, M., and Müller, S. 2006. Direct and fast ray tracing of NURBS surfaces. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pp. 161–168.
- [2] Abi-Ezzi, S. S. and Shirman, L. A. 1991. Tessellation of curved surfaces under highly varying transformations. In *Proceedings of Eurographics '91*, pp. 385–397.
- [3] Abi-Ezzi, S. S. and Subramaniam, S. 1994. Fast dynamic tessellation of trimmed NURBS surfaced. *Computer Graphics Forum (Proceedings of Eurographics 1994)*, 13 (3), 107–126.
- [4] Abi-Ezzi, S. S. and Wozny, M. J. 1990. Factoring a homogeneous transformation for a more efficient graphics pipeline. In *Proceedings of Eurographics '90*, pp. 159–175.
- [5] Advanced Micro Devices, Inc. 2007. *AMD Close-To-The-Metal Programming Guide*. Version 1.1 Alpha.
- [6] Advanced Micro Devices, Inc. 2009. ATI Stream computing: technical overview.
▷ http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf
- [7] Agrawala, M., Ramamoorthi, R., Heirich, A., and Moll, L. 2000. Efficient image-based methods for rendering soft shadows. In *Proceedings of ACM SIGGRAPH 2000*, pp. 375–384.
- [8] Akenine-Möller, T., Haines, E., and Hoffman, N. 2008. *Real-Time Rendering*. 3rd ed. A K Peters.
- [9] Amor, M., Bóo, M., Strasser, W., Hirche, J., and Doggett, M. 2005. A meshing scheme for efficient hardware implementation of butterfly subdivision using displacement mapping. *IEEE Computer Graphics and Applications*, 25 (2), 46–59.
- [10] Anderson, S. E. 2009. Bit twiddling hacks.
▷ <http://graphics.stanford.edu/~seander/bithacks.html>
- [11] Annen, T., Dong, Z., Mertens, T., Bekaert, P., Seidel, H.-P., and Kautz, J. 2008. Real-time, all-frequency shadows in dynamic scenes. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2008)*, 27 (3), art. 34.
- [12] Annen, T., Mertens, T., Bekaert, P., Seidel, H.-P., and Kautz, J. 2007. Convolution shadow maps. In *Proceedings of Eurographics Symposium on Rendering 2007*, pp. 51–60.
- [13] Annen, T., Mertens, T., Seidel, H.-P., Flerackers, E., and Kautz, J. 2008. Exponential shadow maps. In *Proceedings of Graphics Interface 2008*, pp. 155–161.

- [14] Arvo, J., Hirvikorpi, M., and Tyystjärvi, J. 2004. Approximate soft shadows with an image-space flood-fill algorithm. *Computer Graphics Forum (Proceedings of Eurographics 2004)*, 23 (3), 271–280.
- [15] Assarsson, U. and Akenine-Möller, T. 2003. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22 (3), 511–520.
- [16] Assarsson, U., Dougherty, M., Mounier, M., and Akenine-Möller, T. 2003. An optimized soft shadow volume algorithm with real-time performance. In *Proceedings of Graphics Hardware 2003*, pp. 33–40.
- [17] Aszódi, B. and Szirmay-Kalos, L. 2006. Real-time soft shadows with shadow accumulation. In *Eurographics 2006 Short Papers*, pp. 53–56.
- [18] ATI Technologies Inc. 2001. Truform. White paper.
▷ <http://ati.amd.com/products/pdf/truform.pdf>
- [19] Atty, L., Holzschuch, N., Lapierre, M., Hasenfratz, J.-M., Hansen, C., and Sillion, F. X. 2006. Soft shadow maps: efficient sampling of light source visibility. *Computer Graphics Forum*, 25 (4), 725–741.
- [20] Balázs, Á., Guthe, M., and Klein, R. 2004. Fat borders: gap filling for efficient view-dependent LOD NURBS rendering. *Computers & Graphics*, 28 (1), 79–85.
- [21] Bank, R. E., Sherman, A. H., and Weiser, A. 1983. Some refinement algorithms and data structures for regular local mesh refinement. In *Scientific Computing: Applications of Mathematics and Computing to the Physical Sciences* (edited by R. Stepleman, M. Carver, R. Peskin, W. F. Ames, and R. Vichnevetsky), pp. 3–17. North-Holland.
- [22] Barsky, B. A., DeRose, T. D., and Dippé, M. D. 1987. An adaptive subdivision method with crack prevention for rendering beta-spline objects. Tech. Rep. UCB/CSD-87-348, University of California, Berkeley.
- [23] Barten, P. G. J. 1989. The effects of picture size and definition on perceived image quality. *IEEE Transactions on Electron Devices*, 36 (9), 1865–1869.
- [24] Barten, P. G. J. 1989. The square root integral (SQRI): a new metric to describe the effect of various display parameters on perceived image quality. In *Proceedings of SPIE*, vol. 1077 (Human Vision, Visual Processing, and Digital Display), pp. 73–82.
- [25] Barten, P. G. J. 1990. Evaluation of subjective image quality with the square-root integral method. *Journal of the Optical Society of America A*, 7 (10), 2024–2031.
- [26] Barten, P. G. J. 2003. Formula for the contrast sensitivity of the human eye. In *Proceedings of SPIE*, vol. 5294 (Image Quality and System Performance), pp. 231–238.
- [27] Barth, W. 1996. Two projective surfaces with many nodes, admitting the symmetries of the icosahedron. *Journal of Algebraic Geometry*, 5 (1), 173–186.
- [28] Barth, W. and Stürzlinger, W. 1993. Efficient ray tracing for Bezier and B-spline surfaces. *Computers & Graphics*, 17 (4), 423–430.

- [29] Bavoil, L. 2008. Advanced soft shadow mapping techniques. Presentation, *Game Developers Conference 2008*.
▷ http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_SoftShadowMapping.pdf
- [30] Bavoil, L., Callahan, S. P., and Silva, C. T. 2008. Robust soft shadow mapping with back-projection and depth peeling. *Journal of Graphics Tools*, 13 (1), 19–30.
- [31] Bavoil, L. and Silva, C. T. 2006. Real-time soft shadows with cone culling. In *ACM SIGGRAPH 2006 Sketches*.
- [32] Benthin, C., Wald, I., and Slusallek, P. 2004. Interactive ray tracing of freeform surfaces. In *Proceedings of AFRIGRAPH 2004*, pp. 99–106.
- [33] Biermann, H., Levin, A., and Zorin, D. 2000. Piecewise smooth subdivision surfaces with normal control. In *Proceedings of ACM SIGGRAPH 2000*, pp. 113–120.
- [34] Bischoff, S., Kobbelt, L. P., and Seidel, H.-P. 2000. Towards hardware implementation of Loop subdivision. In *Proceedings of Workshop on Graphics Hardware 2000*, pp. 41–50.
- [35] Blelloch, G. E. 1990. *Vector Models for Data-Parallel Computing*. MIT Press.
- [36] Blinn, J. 2007. How to solve a cubic equation, Part 5: Back to numerics. *IEEE Computer Graphics and Applications*, 27 (3), 78–89.
- [37] Bloomenthal, J. (ed.). 1997. *Introduction to Implicit Surfaces*. Morgan Kaufmann.
- [38] Blythe, D. 2006. The Direct3D 10 system. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2006)*, 25 (3), 724–734.
- [39] Böhm, W. 1981. Generating the Bézier points of B-spline curves and surfaces. *Computer-Aided Design*, 13 (6), 365–366.
- [40] Bolin, M. R. and Meyer, G. W. 1998. A perceptually based adaptive sampling algorithm. In *Proceedings of ACM SIGGRAPH 98*, pp. 299–309.
- [41] Bolin, M. R. and Meyer, G. W. 1999. A visual difference metric for realistic image synthesis. In *Proceedings of SPIE*, vol. 3644 (Human Vision and Electronic Imaging IV), pp. 106–120.
- [42] Bolz, J. and Schröder, P. 2002. Rapid evaluation of Catmull-Clark subdivision surfaces. In *Proceedings of Web3D 2002*, pp. 11–17.
- [43] Bolz, J. and Schröder, P. 2003. Evaluation of subdivision surfaces on programmable graphics hardware.
▷ <http://multires.caltech.edu/pubs/GPUSubD.pdf>
- [44] Bonnef, Y. and Sagi, D. 1998. Effects of spatial configuration on contrast detection. *Vision Research*, 38 (22), 3541–3553.
- [45] Borgeat, L., Godin, G., Blais, F., Massicotte, P., and Lahanier, C. 2005. GoLD: interactive display of huge colored and textured models. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2005)*, 24 (3), 869–877.

- [46] Boubekur, T. and Alexa, M. 2008. Phong tessellation. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2008)*, 27 (5), art. 141.
- [47] Boubekur, T., Reuter, P., and Schlick, C. 2005. Scalar tagged PN triangles. In *Eurographics 2005 Short Presentations*, pp. 17–20.
- [48] Boubekur, T. and Schlick, C. 2005. Generic mesh refinement on GPU. In *Proceedings of Graphics Hardware 2005*, pp. 99–104.
- [49] Boubekur, T. and Schlick, C. 2007. Generic adaptive mesh refinement. In *GPU Gems 3* (edited by H. Nguyen), chap. 5, pp. 93–104. Addison Wesley Professional.
- [50] Boubekur, T. and Schlick, C. 2007. QAS: real-time quadratic approximation of subdivision surfaces. In *Proceedings of Pacific Graphics 2007*, pp. 453–456.
- [51] Boubekur, T. and Schlick, C. 2008. A flexible kernel for adaptive mesh refinement on GPU. *Computer Graphics Forum*, 27 (1), 102–113.
- [52] Boyd, C. 2008. Direct3D 11 compute shader: more generality for advanced techniques. Presentation, *Gamefest 2008*.
 ▶ <http://www.microsoft.com/downloads/details.aspx?FamilyId=9f943b2b-53ea-4f80-84b2-f05a360bfc6a>
- [53] Brabec, S. and Seidel, H.-P. 2002. Single sample soft shadows using depth maps. In *Proceedings of Graphics Interface 2002*, pp. 219–228.
- [54] Bresenham, J. E. 1965. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4 (1), 25–30.
- [55] Bruijns, J. 1998. Quadratic Bezier triangles as drawing primitives. In *Proceedings of Workshop on Graphics Hardware 1998*, pp. 15–24.
- [56] Bunnell, M. 2005. Adaptive tessellation of subdivision surfaces with displacement mapping. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (edited by M. Pharr), chap. 7, pp. 109–122. Addison Wesley Professional.
- [57] Burbeck, C. A. and Kelly, D. H. 1980. Spatiotemporal characteristics of visual mechanisms: excitatory-inhibitory model. *Journal of the Optical Society of America*, 70 (9), 1121–1126.
- [58] Burt, P. J. and Adelson, E. H. 1983. The Laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31 (4), 532–540.
- [59] Cai, X.-H., Jia, Y.-T., Wang, X., Hu, S.-M., and Martin, R. R. 2006. Rendering soft shadows using multilayered shadow frons. *Computer Graphics Forum*, 25 (1), 15–28.
- [60] Campagna, S., Slusallek, P., and Seidel, H.-P. 1997. Ray tracing of spline surfaces: Bézier clipping, Chebyshev boxing, and bounding volume hierarchy – a critical comparison with new results. *The Visual Computer*, 13 (6), 265–282.

- [61] Campbell, F. W. and Robson, J. G. 1964. Application of Fourier analysis to the modulation response of the eye. *Journal of the Optical Society of America*, 54, 581A.
- [62] Castaño, I. 2008. Tessellation of displaced subdivision surfaces in DX11. Presentation, *Gamefest 2008*.
▷ <http://www.microsoft.com/downloads/details.aspx?FamilyId=a484d275-9360-41dd-abd4-86d4f1218d0c>
- [63] Cater, K., Chalmers, A., and Ledda, P. 2002. Selective quality rendering by exploiting human inattentional blindness: looking but not seeing. In *Proceedings of ACM Symposium on Virtual Reality Software and Technology 2002*, pp. 17–24.
- [64] Catmull, E. and Clark, J. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10 (6), 350–355.
- [65] Catmull, E. E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.D. thesis, University of Utah.
- [66] Chan, E. and Durand, F. 2003. Rendering fake soft shadows with smoothies. In *Proceedings of Eurographics Symposium on Rendering 2003*, pp. 208–218.
- [67] Chang, S.-L., Shantz, M., and Rocchetti, R. 1988. Rendering cubic curves and surfaces with integer adaptive forward differencing. *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, 22 (4), 157–166.
- [68] Chen, Y.-C. and Chang, C.-F. 2008. A prism-free method for silhouette rendering in inverse displacement mapping. *Computer Graphics Forum (Proceedings of Pacific Graphics 2008)*, 27 (7), 1929–1936.
- [69] Chhugani, J. and Kumar, S. 2001. View-dependent adaptive tessellation of spline surfaces. In *Proceedings of ACM Symposium on Interactive 3D Graphics 2001*, pp. 59–62.
- [70] Chong, H. Y., Gortler, S. J., and Zickler, T. 2008. A perception-based color space for illumination-invariant image processing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2008)*, 27 (3), art. 61.
- [71] Chung, K. and Kim, L.-S. 2003. Adaptive tessellation of PN triangle with modified Bresenham algorithm. In *Proceedings of SOC Design Conference 2003*, pp. 448–452.
- [72] Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R. 2004. Adaptive TetraPuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2004)*, 23 (3), 796–803.
- [73] Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R. 2005. Batched multi triangulation. In *Proceedings of IEEE Visualization 2005*, pp. 207–214.
- [74] Clark, J. H. 1976. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19 (10), 547–554.
- [75] Clark, J. H. 1979. A fast algorithm for rendering parametric surfaces. *Computer Graphics*, 13 (2), 7–12.

- [76] Claustres, L., Barthe, L., and Paulin, M. 2007. Wavelet encoding of BRDFs for real-time rendering. In *Proceedings of Graphics Interface 2007*, pp. 169–176.
- [77] Cohen, J., Olano, M., and Manocha, D. 1998. Appearance-preserving simplification. In *Proceedings of ACM SIGGRAPH 98*, pp. 115–122.
- [78] Cook, R. L., Carpenter, L., and Catmull, E. 1987. The Reyes image rendering architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21 (4), 95–102.
- [79] Cook, R. L., Halstead, J., Planck, M., and Ryu, D. 2007. Stochastic simplification of aggregate detail. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2007)*, 26 (3), art. 79.
- [80] Cook, R. L., Porter, T., and Carpenter, L. 1984. Distributed ray tracing. *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 18 (3), 137–145.
- [81] Crow, F. 1987. The origins of the teapot. *IEEE Computer Graphics and Applications*, 7 (1), 8–19.
- [82] Crow, F. C. 1977. Shadow algorithms for computer graphics. *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, 11 (2), 242–248.
- [83] Crow, F. C. 1984. Summed-area tables for texture mapping. *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 18 (3), 207–212.
- [84] Daly, S. J. 1992. The visible differences predictor: an algorithm for the assessment of image fidelity. In *Proceedings of SPIE*, vol. 1666 (Human Vision, Visual Processing, and Digital Display III), pp. 2–15.
- [85] Daly, S. J. 1993. The visible differences predictor: an algorithm for the assessment of image fidelity. In *Digital Images and Human Vision* (edited by A. B. Watson), chap. 14, pp. 179–206. MIT Press.
- [86] Daly, S. J. 1998. Engineering observations from spatiovelocity and spatiotemporal visual models. In *Proceedings of SPIE*, vol. 3299 (Human Vision and Electronic Imaging III), pp. 180–191.
- [87] Daly, S. J., Zeng, W., Li, J., and Lei, S. 2000. Visual masking in wavelet compression for JPEG2000. In *Proceedings of SPIE*, vol. 3974 (Image and Video Communications and Processing 2000), pp. 66–80.
- [88] de Casteljau, P. 1959. Outillages méthodes calcul. Tech. rep., S.A. André Citroën.
- [89] Décoret, X. 2005. N-buffers for efficient depth map query. *Computer Graphics Forum (Proceedings of Eurographics 2005)*, 24 (3), 393–400.
- [90] DeCoro, C. and Rusinkiewicz, S. 2008. Subtractive shadows: a flexible framework for shadow level of detail. *Journal of Graphics Tools*, 13 (1), 45–56.
- [91] Deering, M., Winner, S., Schediwy, B., Duffy, C., and Hunt, N. 1988. The triangle processor and normal vector shader: a VLSI system for high performance graphics. *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, 22 (4), 21–30.

- [92] DeRose, T., Kass, M., and Truong, T. 1998. Subdivision surfaces in character animation. In *Proceedings of ACM SIGGRAPH 98*, pp. 85–94.
- [93] Dmitriev, K. 2007. Soft shadows. White Paper WP-03016-001_v01, NVIDIA Corporation.
▷ <http://developer.download.nvidia.com/whitepapers/2007/SDK10/SoftShadows.pdf>
- [94] Doggett, M. and Hirche, J. 2000. Adaptive view dependent tessellation of displacement maps. In *Proceedings of Workshop on Graphics Hardware 2000*, pp. 59–66.
- [95] Donnelly, W. and Lauritzen, A. 2006. Variance shadow maps. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2006*, pp. 161–165.
- [96] Doo, D. and Sabin, M. 1978. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10 (6), 356–360.
- [97] Doo, D. W. H. 1978. A subdivision algorithm for smoothing down irregular shaped polyhedrons. In *Proceedings on Interactive Techniques in Computer Aided Design*, pp. 157–165.
- [98] Drettakis, G., Bonneel, N., Dachsbacher, C., Lefebvre, S., Schwarz, M., and Viaud-Delmon, I. 2007. An interactive perceptual rendering pipeline using contrast and spatial masking. In *Proceedings of Eurographics Symposium on Rendering 2007*, pp. 297–308.
- [99] Duda, R. O. and Hart, P. E. 1972. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15 (1), 11–15.
- [100] Dumont, R., Pellacini, F., and Ferwerda, J. A. 2003. Perceptually-driven decision theory for interactive realistic rendering. *ACM Transactions on Graphics*, 22 (2), 152–181.
- [101] Dyken, C., Reimers, M., and Seland, J. 2008. Real-time GPU silhouette refinement using blended Bézier patches. *Computer Graphics Forum*, 27 (1), 1–12.
- [102] Dyken, C., Reimers, M., and Seland, J. 2009. Semi-uniform adaptive patch tessellation. *Computer Graphics Forum*, 28 (8), 2255–2263.
- [103] Ebner, F. and Fairchild, M. D. 1998. Development and testing of a color space (IPT) with improved hue uniformity. In *Proceedings of IS&T/SID 6th Color Imaging Conference*, pp. 8–13.
- [104] Efremov, A. 2004. *Efficient Ray Tracing of Truncated NURBS Surfaces*. Master's thesis, University of Saarland.
- [105] Efremov, A., Havran, V., and Seidel, H.-P. 2005. Robust and numerically stable Bézier clipping method for ray tracing NURBS surfaces. In *Proceedings of Spring Conference on Computer Graphics 2005*, pp. 127–135.
- [106] Eisemann, E. and Décoret, X. 2006. Plausible image based soft shadows using occlusion textures. In *Proceedings of SIBGRAPI 2006*, pp. 155–162.
- [107] Eisemann, E. and Décoret, X. 2007. Visibility sampling on GPU and applications. *Computer Graphics Forum (Proceedings of Eurographics 2007)*, 26 (3), 535–544.

- [108] Eisemann, E. and Décoret, X. 2008. Occlusion textures for plausible soft shadows. *Computer Graphics Forum*, 27 (1), 13–23.
- [109] Elber, G. and Cohen, E. 1996. Adaptive isocurve-based rendering for freeform surfaces. *ACM Transactions on Graphics*, 15 (3), 249–263.
- [110] Espino, F. J., Bóo, M., Amor, M., and Bruguera, J. D. 2007. Hardware support for adaptive tessellation of Bézier surfaces based on local tests. *Journal of Systems Architecture*, 53 (4), 233–250.
- [111] Everitt, C. 2001. Interactive order-independent transparency. Tech. rep., NVIDIA Corporation.
 ▶ http://developer.nvidia.com/object/Interactive_Order_Transparency.html
- [112] Fairchild, M. D. 2004. *Color Appearance Models*. 2nd ed. John Wiley & Sons.
- [113] Fairchild, M. D. and Johnson, G. M. 2002. Meet iCAM: a next-generation color appearance model. In *Proceedings of IS&T/SID 10th Color Imaging Conference*, pp. 33–38.
- [114] Fairchild, M. D. and Johnson, G. M. 2004. iCAM framework for image appearance, differences, and quality. *Journal of Electronic Imaging*, 13 (1), 126–138.
- [115] Farin, G. 1986. Triangular Bernstein-Bézier patches. *Computer Aided Geometric Design*, 3 (2), 83–127.
- [116] Farin, G. 2002. *Curves and Surfaces for CAGD: A Practical Guide*. 5th ed. Morgan Kaufmann.
- [117] Farin, G. 2003. PN patches.
 ▶ <http://www.farinhansford.com/gerald/classes/cse578/additions/pnormals.pdf>
- [118] Farrugia, J.-P., Albin, S., and Péroche, B. 2004. A perceptual adaptive image metric for computer graphics. In *WSCG 2004 Poster Proceedings*, pp. 49–52.
- [119] Farrugia, J.-P. and Péroche, B. 2004. A progressive rendering algorithm using an adaptive perceptually based image metric. *Computer Graphics Forum (Proceedings of Eurographics 2004)*, 23 (3), 605–614.
- [120] Fatahalian, K. and Houston, M. 2008. A closer look at GPUs. *Communications of the ACM*, 51 (10), 50–57.
- [121] Feng, X.-F. 2006. LCD motion-blur analysis, perception, and reduction using synchronized backlight flashing. In *Proceedings of SPIE*, vol. 6057 (Human Vision and Electronic Imaging XI), pp. 213–226.
- [122] Fernando, R. 2005. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*.
- [123] Ferwerda, J. A. 2001. Elements of early vision for computer graphics. *IEEE Computer Graphics and Applications*, 21 (5), 22–33.
- [124] Ferwerda, J. A. 2003. Three varieties of realism in computer graphics. In *Proceedings of SPIE*, vol. 5007 (Human Vision and Electronic Imaging VIII), pp. 290–297.

- [125] Ferwerda, J. A. 2008. Psychophysics 101: how to run perception experiments in computer graphics. *ACM SIGGRAPH 2008: Classes*.
- [126] Ferwerda, J. A., Pattanaik, S. N., Shirley, P., and Greenberg, D. P. 1996. A model of visual adaptation for realistic image synthesis. In *Proceedings of ACM SIGGRAPH 96*, pp. 249–258.
- [127] Ferwerda, J. A., Pattanaik, S. N., Shirley, P., and Greenberg, D. P. 1997. A model of visual masking for computer graphics. In *Proceedings of ACM SIGGRAPH 97*, pp. 143–152.
- [128] Filip, D., Magedson, R., and Markot, R. 1986. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3 (4), 295–311.
- [129] Filip, D. J. 1986. Adaptive subdivision algorithms for a set of Bézier triangles. *Computer-Aided Design*, 18 (2), 74–78.
- [130] Forest, V., Barthe, L., and Paulin, M. 2006. Realistic soft shadows by penumbra-wedges blending. In *Proceedings of Graphics Hardware 2006*, pp. 39–47.
- [131] Forest, V., Barthe, L., and Paulin, M. 2008. Accurate shadows by depth complexity sampling. *Computer Graphics Forum (Proceedings of Eurographics 2008)*, 27 (2), 663–674.
- [132] Fredericksen, R. E. and Hess, R. F. 1998. Estimating multiple temporal mechanisms in human vision. *Vision Research*, 38 (7), 1023–1040.
- [133] Fünfzig, C., Müller, K., Hansford, D., and Farin, G. 2008. PNG1 triangles for tangent plane continuous surfaces on the GPU. In *Proceedings of Graphics Interface 2008*, pp. 219–226.
- [134] Gaddipatti, A., Machiraju, R., and Yagel, R. 1997. Steering image generation with wavelet based perceptual metric. *Computer Graphics Forum (Proceedings of Eurographics 1997)*, 16 (3), 241–251.
- [135] Geimer, M. and Abert, O. 2005. Interactive ray tracing of trimmed bicubic Bézier surfaces without triangulation. In *WSCG 2005 Full Papers Conference Proceedings*, pp. 71–78.
- [136] Gepshtein, S., Tyukin, I., and Kubovy, M. 2007. The economics of motion perception and invariants of visual sensitivity. *Journal of Vision*, 7 (8), art. 8.
- [137] Gibson, S. and Hubbard, R. J. 1997. Perceptually-driven radiosity. *Computer Graphics Forum*, 16 (2), 129–141.
- [138] Giegl, M. and Wimmer, M. 2007. Fitted virtual shadow maps. In *Proceedings of Graphics Interface 2007*, pp. 159–168.
- [139] Giegl, M. and Wimmer, M. 2007. Unpopping: solving the image-space blend problem for smooth discrete LOD transitions. *Computer Graphics Forum*, 26 (1), 46–49.
- [140] Gobbetti, E. and Marton, F. 2005. Far voxels: a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2005)*, 24 (3), 878–885.

- [141] Golomb, S. W. 1954. Checker boards and polyominoes. *The American Mathematical Monthly*, 61 (10), 675–682.
- [142] Greene, N., Kass, M., and Miller, G. 1993. Hierarchical z-buffer visibility. In *Proceedings of ACM SIGGRAPH 93*, pp. 231–238.
- [143] Gruen, H. 2005. Efficient tessellation on the GPU through instancing. *Journal of Game Development*, 1 (3).
- [144] Gruen, H. 2006. Smoothed N-patches. In *ShaderX⁵: Advanced Rendering Techniques* (edited by W. Engel), pp. 5–22. Charles River Media.
- [145] Grün, H. 2008. Approximate cumulative distribution function shadow mapping. In *ShaderX⁶: Advanced Rendering Techniques* (edited by W. Engel), pp. 239–256. Charles River Media.
- [146] Gu, X., Gortler, S. J., and Hoppe, H. 2002. Geometry images. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2002)*, 21 (3), 355–361.
- [147] Guennebaud, G., Barthe, L., and Paulin, M. 2006. Real-time soft shadow mapping by backprojection. In *Proceedings of Eurographics Symposium on Rendering 2006*, pp. 227–234.
- [148] Guennebaud, G., Barthe, L., and Paulin, M. 2007. High-quality adaptive soft shadow mapping. *Computer Graphics Forum (Proceedings of Eurographics 2007)*, 26 (3), 525–533.
- [149] Günther, J., Popov, S., Seidel, H.-P., and Slusallek, P. 2007. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of IEEE/EG Symposium on Interactive Ray Tracing 2007*, pp. 113–118.
- [150] Guthe, M., Balázs, Á., and Klein, R. 2005. GPU-based trimming and tessellation of NURBS and T-spline surfaces. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2005)*, 24 (3), 1016–1023.
- [151] Guthe, M., Balázs, Á., and Klein, R. 2006. GPU-based appearance preserving trimmed NURBS rendering. *Journal of WSCG*, 14 (1–3), 1–8.
- [152] Halstead, M., Kass, M., and DeRose, T. 1993. Efficient, fair interpolation using Catmull-Clark surfaces. In *Proceedings of ACM SIGGRAPH 93*, pp. 35–44.
- [153] Hamill, J., McDonnell, R., Dobbyn, S., and O’Sullivan, C. 2005. Perceptual evaluation of impostor representations for virtual humans and buildings. *Computer Graphics Forum (Proceedings of Eurographics 2005)*, 24 (3), 623–633.
- [154] Han, C., Sun, B., Ramamoorthi, R., and Grinspun, E. 2007. Frequency domain normal map filtering. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2007)*, 26 (3), art. 28.
- [155] Hanrahan, P. 1983. Ray tracing algebraic surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, 17 (3), 83–90.

- [156] Hardy, J. L., Delahunt, P. B., Okajima, K., and Werner, J. S. 2005. Senescence of spatial chromatic contrast sensitivity. I. Detection under conditions controlling for optical factors. *Journal of the Optical Society of America A*, 22 (1), 49–59.
- [157] Hargreaves, S. and Harris, M. 2004. Deferred shading. Presentation, *6800 Leagues Under the Sea*.
▷ http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf
- [158] Harris, M. A. and Reingold, E. M. 2004. Line drawing, leap years, and Euclid. *ACM Computing Surveys*, 36 (1), 68–80.
- [159] Hasenfratz, J.-M., Lapierre, M., Holzschuch, N., and Sillion, F. 2003. A survey of real-time soft shadows algorithms. *Computer Graphics Forum*, 22 (4), 753–774.
- [160] Heckbert, P. S. and Hanrahan, P. 1984. Beam tracing polygonal objects. *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 18 (3), 119–127.
- [161] Heckbert, P. S. and Herf, M. 1997. Simulating soft shadows with graphics hardware. Tech. Rep. CMU-CS-97-104, Carnegie Mellon University.
- [162] Heidmann, T. 1991. Real shadows real time. *IRIS Universe*, 18, 28–31.
- [163] Hong, G. and Luo, M. R. 2002. Perceptually based colour difference for complex images. In *Proceedings of SPIE*, vol. 4421 (9th Congress of the International Colour Association), pp. 618–621.
- [164] Hoppe, H. 1996. Progressive meshes. In *Proceedings of ACM SIGGRAPH 96*, pp. 99–108.
- [165] Hoppe, H. 1997. View-dependent refinement of progressive meshes. In *Proceedings of ACM SIGGRAPH 97*, pp. 189–198.
- [166] Hoppe, H., DeRose, T., Duchamp, T., Halstead, M., Jin, H., McDonald, J., Schweitzer, J., and Stuetzle, W. 1994. Piecewise smooth surface reconstruction. In *Proceedings of ACM SIGGRAPH 94*, pp. 295–302.
- [167] Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. 2007. Interactive k-d tree GPU raytracing. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2007*, pp. 167–174.
- [168] Hu, L., Sander, P. V., and Hoppe, H. 2009. Parallel view-dependent refinement of progressive meshes. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2009*, pp. 169–176.
- [169] Huang, Y. and Su, H. 2006. The bound on derivatives of rational Bézier curves. *Computer Aided Geometric Design*, 23 (9), 698–702.
- [170] Itti, L. and Koch, C. 2000. A saliency-based search mechanism for overt and covert shifts of visual attention. *Vision Research*, 40 (10–12), 1489–1506.
- [171] Itti, L. and Koch, C. 2001. Computational modelling of visual attention. *Nature Reviews Neuroscience*, 2 (3), 194–203.

- [172] Itti, L., Koch, C., and Niebur, E. 1998. A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20 (11), 1254–1259.
- [173] Johnson, G. M. and Fairchild, M. D. 2003. A top down description of S-CIELAB and CIEDE2000. *Color Research & Application*, 28 (6), 425–435.
- [174] Kajiya, J. T. 1982. Ray tracing parametric patches. *Computer Graphics (Proceedings of ACM SIGGRAPH 82)*, 16 (3), 245–254.
- [175] Kajiya, J. T. 1986. The rendering equation. *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, 20 (4), 143–150.
- [176] Kanai, T. 2007. Fragment-based evaluation of non-uniform B-spline surfaces on GPUs. *Computer-Aided Design & Applications*, 4 (1–4), 287–294.
- [177] Kanamori, Y., Szego, Z., and Nishita, T. 2008. GPU-based fast ray casting for a large number of metaballs. *Computer Graphics Forum (Proceedings of Eurographics 2008)*, 27 (2), 351–360.
- [178] Karčiauskas, K. and Peters, J. 2007. Bicubic polar subdivision. *ACM Transactions on Graphics*, 26 (4), art. 14.
- [179] Kayser, C., Petkov, C. I., Lippert, M., and Logothetis, N. K. 2005. Mechanisms for allocating auditory attention: an auditory saliency map. *Current Biology*, 15 (21), 1943–1947.
- [180] Kazakov, M. 2007. Catmull-Clark subdivision for geometry shaders. In *Proceedings of Afrigraph 2007*, pp. 77–84.
- [181] Keating, B. and Max, N. 1999. Shadow penumbras for complex objects by depth-dependent filtering of multi-layer depth images. In *Proceedings of Eurographics Workshop on Rendering 1999*, pp. 197–212.
- [182] Keller, A. and Heidrich, W. 2001. Interleaved sampling. In *Proceedings of Eurographics Workshop on Rendering 2001*, pp. 269–276.
- [183] Kelly, D. H. 1979. Motion and vision. II. Stabilized spatio-temporal threshold surface. *Journal of the Optical Society of America*, 69 (10), 1340–1349.
- [184] Kelly, D. H. 1983. Spatiotemporal variation of chromatic and achromatic contrast thresholds. *Journal of the Optical Society of America*, 73 (6), 742–750.
- [185] Khronos OpenCL Working Group. 2009. *The OpenCL Specification*. Version 1.0.
- [186] Kirsch, F. and Döllner, J. 2003. Real-time soft shadows using a single light sample. *Journal of WSCG*, 11 (2), 255–262.
- [187] Knill, D. C., Mamassia, P., and Kersten, D. 1997. Geometry of shadows. *Journal of the Optical Society of America A*, 14 (12), 3216–3232.
- [188] Knoll, A. 2007. A survey of implicit surface rendering methods, and a proposal for a common sampling framework. *Second Annual Workshop of IRTG 1131*.
 ▷ <http://www.cs.utah.edu/~knolla/impsurvey.pdf>

- [189] Knoll, A., Hijazi, Y., Kensler, A., Schott, M., Hansen, C., and Hagen, H. 2009. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. *Computer Graphics Forum*, 28 (1), 26–40.
- [190] Kobbelt, L. 2000. $\sqrt{3}$ -subdivision. In *Proceedings of ACM SIGGRAPH 2000*, pp. 103–112.
- [191] Kobbelt, L. P., Labsik, U., and Seidel, H.-P. 1999. $\sqrt{3}$ -subdivision and forward adaptive refinement. In *Proceedings of Korea-Israel Bi-National Conference on Geometrical Modeling and Computer Graphics in the World Wide Web Era*, pp. 245–252.
- [192] Kochanek, D. H. U. and Bartels, R. H. 1984. Interpolating splines with local tension, continuity, and bias control. *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 18 (3), 33–41.
- [193] Kovacs, D., Mitchell, J., Drone, S., and Zorin, D. 2009. Real-time creased approximate subdivision surfaces. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2009*, pp. 155–160.
- [194] Krishnamurthy, A., Khardekar, R., and McMains, S. 2007. Direct evaluation of NURBS curves and surfaces on the GPU. In *Proceedings of ACM Solid and Physical Modeling Symposium 2007*, pp. 329–334.
- [195] Krishnamurthy, A., Khardekar, R., McMains, S., Haller, K., and Elber, G. 2008. Performing efficient NURBS modeling operations on the GPU. In *Proceedings of ACM Solid and Physical Modeling Symposium 2008*, pp. 257–268.
- [196] Kuang, J., Johnson, G. M., and Fairchild, M. D. 2007. iCAM06: a refined image appearance model for HDR image rendering. *Journal of Visual Communication and Image Representation*, 18 (5), 406–414.
- [197] Kuehni, R. G. 2002. CIEDE2000, milestone or final answer? *Color Research & Application*, 27 (2), 126–127.
- [198] Kulikowski, J. J. and Tolhurst, D. J. 1973. Psychophysical evidence for sustained and transient detectors in human vision. *The Journal of Physiology*, 232 (1), 149–162.
- [199] Kumar, S. 1996. *Interactive Rendering of Parametric Spline Surfaces*. Ph.D. thesis, University of North Carolina.
- [200] Kumar, S. 1998. Interactive visualization of triangular Bézier surfaces. In *Proceedings of Indian Conference on Computer Vision, Graphics and Image Processing 1998*.
- [201] Lacewell, D. and Burley, B. 2007. Exact evaluation of Catmull-Clark subdivision surfaces near B-spline boundaries. *Journal of Graphics Tools*, 12 (3), 7–15.
- [202] Laine, S. and Aila, T. 2005. Hierarchical penumbra casting. *Computer Graphics Forum (Proceedings of Eurographics 2005)*, 24 (3), 313–322.
- [203] Laine, S., Aila, T., Assarsson, U., Lehtinen, J., and Akenine-Möller, T. 2005. Soft shadow volumes for ray tracing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2005)*, 24 (3), 1156–1165.

- [204] Lane, J. M., Carpenter, L. C., Whitted, T., and Blinn, J. F. 1980. Scan line methods for displaying parametrically defined surfaces. *Communications of the ACM*, 23 (1), 23–34.
- [205] Lane, J. M. and Riesenfeld, R. F. 1980. A theoretical development for the computer generation and display of piecewise polynomial functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2 (1), 35–46.
- [206] Lauritzen, A. 2007. Summed-area variance shadow maps. In *GPU Gems 3* (edited by H. Nguyen), chap. 8, pp. 157–182. Addison Wesley Professional.
- [207] Lauritzen, A. and McCool, M. 2008. Layered variance shadow maps. In *Proceedings of Graphics Interface 2008*, pp. 139–146.
- [208] Le Grand, Y. 1968. *Light, Colour and Vision*. 2nd English ed. Chapman and Hall.
- [209] Lee, A., Moreton, H., and Hoppe, H. 2000. Displaced subdivision surfaces. In *Proceedings of ACM SIGGRAPH 2000*, pp. 85–94.
- [210] Lee, C. H., Varshney, A., and Jacobs, D. W. 2005. Mesh saliency. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2005)*, 24 (3), 659–666.
- [211] Lee, Y.-C. and Jen, C.-W. 2001. Improved quadratic normal vector interpolation for realistic shading. *The Visual Computer*, 17 (6), 337–352.
- [212] Legge, G. E. 1981. A power law for contrast discrimination. *Vision Research*, 21 (4), 457–467.
- [213] Legge, G. E. and Foley, J. M. 1980. Contrast masking in human vision. *Journal of the Optical Society of America*, 70 (12), 1458–1471.
- [214] Lehtinen, J., Laine, S., and Aila, T. 2006. An improved physically-based soft shadow volume algorithm. *Computer Graphics Forum (Proceedings of Eurographics 2006)*, 25 (3), 303–312.
- [215] Li, B. 1997. *An Analysis and Comparison of Two Visual Discrimination Models*. Master's thesis, University of Oregon.
- [216] Li, B., Meyer, G. W., and Klassen, R. V. 1998. A comparison of two image quality models. In *Proceedings of SPIE*, vol. 3299 (Human Vision and Electronic Imaging III), pp. 98–109.
- [217] Lien, S.-L., Shantz, M., and Pratt, V. 1987. Adaptive forward differencing for rendering curves and surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21 (4), 111–118.
- [218] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. 2008. NVIDIA Tesla: a unified graphics and computing architecture. *IEEE Micro*, 28 (2), 39–55.
- [219] Lindstrom, P. 2000. *Model Simplification using Image and Geometry-Based Metrics*. Ph.D. thesis, Georgia Institute of Technology.
- [220] Lindstrom, P. and Turk, G. 2000. Image-driven simplification. *ACM Transactions on Graphics*, 19 (3), 204–241.

- [221] Lipps, M. A. and Pelz, J. B. 2006. Influence of high-level cognitive goals on gaze patterns: Yabus revisited. In *ACM SIGGRAPH 2006 Research Posters*.
- [222] Lischinski, D. 1992. Converting Bézier triangles into rectangular patches. In *Graphics Gems III* (edited by D. Kirk), chap. V.11, pp. 256–261. Academic Press.
- [223] Lischinski, D. 1994. Converting rectangular patches into Bézier triangles. In *Graphics Gems IV* (edited by P. S. Heckbert), chap. IV.5, pp. 278–285. Academic Press.
- [224] Lischinski, D. and Rappoport, A. 1998. Image-based rendering for non-diffuse synthetic scenes. In *Proceedings of Eurographics Workshop on Rendering 1998*, pp. 301–314.
- [225] Lloyd, D. B., Govindaraju, N. K., Quammen, C., Molnar, S. E., and Manocha, D. 2008. Logarithmic perspective shadow maps. *ACM Transactions on Graphics*, 27 (4), art. 106.
- [226] Lokovic, T. and Veach, E. 2000. Deep shadow maps. In *Proceedings of ACM SIGGRAPH 2000*, pp. 385–392.
- [227] Longhurst, P. 2005. *Rapid Saliency Identification for Selectively Rendering High Fidelity Graphics*. Ph.D. thesis, University of Bristol.
- [228] Longhurst, P., Debattista, K., and Chalmers, A. 2006. A GPU based saliency map for high-fidelity selective rendering. In *Proceedings of AFRIGRAPH 2006*, pp. 21–29.
- [229] Loop, C. and Blinn, J. 2006. Real-time GPU rendering of piecewise algebraic surfaces. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2006)*, 25 (3), 664–670.
- [230] Loop, C. and Schaefer, S. 2008. Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics*, 27 (1), art. 8.
- [231] Loop, C. T. 1987. *Smooth Subdivision Surfaces Based on Triangles*. Master's thesis, University of Utah.
- [232] Lorensen, W. E. and Cline, H. E. 1987. Marching cubes: a high resolution 3D surface construction algorithm. *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21 (4), 163–169.
- [233] Lorenz, H. and Döllner, J. 2008. Dynamic mesh refinement on GPU using geometry shaders. In *WSCG 2008 Full Papers Proceedings*, pp. 97–104.
- [234] Losasso, F., Hoppe, H., Schaefer, S., and Warren, J. 2003. Smooth geometry images. In *Proceedings of Eurographics Symposium on Geometry Processing 2003*, pp. 138–145.
- [235] Lovell, P. G., Párraga, C. A., Troscianko, T., Ripamonti, C., and Tolhurst, D. J. 2006. Evaluation of a multiscale color model for visual difference prediction. *ACM Transactions on Applied Perception*, 3 (3), 155–178.
- [236] Lubin, J. 1995. A visual discrimination model for imaging system design and evaluation. In *Vision Models for Target Detection and Recognition* (edited by E. Peli), pp. 245–283. World Scientific Publishing.
- [237] Luebke, D. and Erikson, C. 1997. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of ACM SIGGRAPH 97*, pp. 199–208.

- [238] Luebke, D. and Hallen, B. 2001. Perceptually driven interactive rendering. Tech. Rep. CS-2001-01, University of Virginia.
- [239] Luebke, D. and Hallen, B. 2001. Perceptually driven simplification for interactive rendering. In *Proceedings of Eurographics Workshop on Rendering 2001*, pp. 223–234.
- [240] Luebke, D., Reddy, M., Cohen, J. D., Varshney, A., Watson, B., and Huebner, R. 2002. *Level of Detail for 3D Graphics*. Morgan Kaufmann.
- [241] Luo, M. R., Cui, G., and Li, C. 2006. Uniform colour spaces based on CIECAM02 colour appearance model. *Color Research & Application*, 31 (4), 320–330.
- [242] Luo, M. R. and Hunt, R. W. G. 1998. The structure of the CIE 1997 colour appearance model (CIECAM97s). *Color Research & Application*, 23 (3), 138–146.
- [243] Ma, W.-C., Chao, S.-H., Tseng, Y.-T., Chuang, Y.-Y., Chang, C.-F., Chen, B.-Y., and Ouhyoung, M. 2005. Level-of-detail representation of bidirectional texture functions for real-time rendering. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2005*, pp. 187–194.
- [244] Macaulay, F. S. 1902. Some formulæ in elimination. *Proceedings of the London Mathematical Society*, 35, 3–37.
- [245] Mamassian, P., Knill, D. C., and Kersten, D. 1998. The perception of cast shadows. *Trends in Cognitive Sciences*, 2 (8), 288–295.
- [246] Mammen, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9 (4), 43–55.
- [247] Mann, S., Lounsbery, M., Loop, C., Meyers, D., Painter, J., DeRose, T., and Sloan, K. 1992. A survey of parametric scattered data fitting using triangular interpolants. In *Curve and Surface Design* (edited by H. Hagen), pp. 145–172. SIAM.
- [248] Mantiuk, R., Daly, S., Myszkowski, K., and Seidel, H.-P. 2005. Predicting visible differences in high dynamic range images – model and its calibration. In *Proceedings of SPIE*, vol. 5666 (Human Vision and Electronic Imaging X), pp. 204–214.
- [249] Martin, R. A., Ahumada, A. J., Jr., and Larimer, J. O. 1992. Color matrix display simulation based upon luminance and chromatic contrast sensitivity of early vision. In *Proceedings of SPIE*, vol. 1666 (Human Vision, Visual Processing, and Digital Display III), pp. 336–342.
- [250] Martin, W., Cohen, E., Fish, R., and Shirley, P. 2000. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools*, 5 (1), 27–52.
- [251] Meyer, G. W. 1988. Wavelength selection for synthetic image generation. *Computer Vision, Graphics, and Image Processing*, 41 (1), 57–79.
- [252] Microsoft Corporation. 2008. DirectX software development kit (November 2008).
 ▷ <http://www.microsoft.com/downloads/details.aspx?FamilyID=5493f76a-6d37-478d-ba17-28b1cca4865a>

- [253] Mo, Q., Popescu, V., and Wyman, C. 2007. The soft shadow occlusion camera. In *Proceedings of Pacific Graphics 2007*, pp. 189–198.
- [254] Moreton, H. 2001. Higher order surfaces. Presentation.
▷ http://developer.nvidia.com/object/higher_order_surfaces.html
- [255] Moreton, H. 2001. Watertight tessellation using forward differencing. In *Proceedings of Workshop on Graphics Hardware 2001*, pp. 25–32.
- [256] Moroney, N., Fairchild, M. D., Hunt, R. W. G., Li, C., Luo, M. R., and Newman, T. 2002. The CIECAM02 color appearance model. In *Proceedings of IS&T/SID 10th Color Imaging Conference*, pp. 23–27.
- [257] Moule, K. and McCool, M. D. 2002. Efficient bounded adaptive tessellation of displacement maps. In *Proceedings of Graphics Interface 2002*, pp. 171–180.
- [258] Mullen, K. T. 1985. The contrast sensitivity of human colour vision to red–green and blue–yellow chromatic gratings. *The Journal of Physiology*, 359, 381–400.
- [259] Müller, K. and Havemann, S. 2000. Subdivision surface tessellation on the fly using a versatile mesh data structure. *Computer Graphics Forum (Proceedings of Eurographics 2000)*, 19 (3), 151–159.
- [260] Munkberg, J., Hasselgren, J., and Akenine-Möller, T. 2008. Non-uniform fractional tessellation. In *Proceedings of Graphics Hardware 2008*, pp. 41–45.
- [261] Myles, A., Karčiauskas, K., and Peters, J. 2007. Extending Catmull-Clark subdivision and PCCM with polar structures. In *Proceedings of Pacific Graphics 2007*, pp. 313–320.
- [262] Myles, A., Ni, T., and Peters, J. 2008. Fast parallel construction of smooth surfaces from meshes with tri/quad/pent facets. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Geometry Processing 2008)*, 27 (5), 1365–1372.
- [263] Myszkowski, K. 1998. The Visible Differences Predictor: applications to global illumination problems. In *Proceedings of Eurographics Workshop on Rendering 1998*, pp. 223–236.
- [264] Myszkowski, K., Rokita, P., and Tawara, T. 1999. Perceptually-informed accelerated rendering of high quality walkthrough sequences. In *Proceedings of Eurographics Workshop on Rendering 1999*, pp. 5–18.
- [265] Myszkowski, K., Rokita, P., and Tawara, T. 2000. Perception-based fast rendering and antialiasing of walkthrough sequences. *IEEE Transactions on Visualization and Computer Graphics*, 6 (4), 360–379.
- [266] Myszkowski, K., Tawara, T., Akamine, H., and Seidel, H.-P. 2001. Perception-guided global illumination solution for animation rendering. In *Proceedings of ACM SIGGRAPH 2001*, pp. 221–230.
- [267] Nankervis, A. 2007. Geometry shader tessellation demo.
▷ <http://www.naixela.com/alex/>

- [268] Neumann, L., Matkovic, K., and Purgathofer, W. 1998. Perception based color image difference. *Computer Graphics Forum (Proceedings of Eurographics 1998)*, 17 (3), 233–241.
- [269] Ni, T., Yeo, Y. I., Myles, A., Goel, V., and Peters, J. 2008. GPU smoothing of quad meshes. In *Proceedings of IEEE International Conference on Shape Modeling and Applications 2008*, pp. 3–9.
- [270] Nickolls, J., Buck, I., Garland, M., and Skadron, K. 2008. Scalable parallel programming with CUDA. *ACM Queue*, 6 (2), 40–53.
- [271] Nielson, G. M. 1987. A transfinite, visually continuous, triangular interpolant. In *Geometric Modeling: Algorithms and New Trends* (edited by G. E. Farin), pp. 235–246. SIAM.
- [272] Nishita, T., Sederberg, T. W., and Kakimoto, M. 1990. Ray tracing trimmed rational surface patches. *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, 24 (4), 337–345.
- [273] NVIDIA Corporation. 2008. *NVIDIA CUDA Programming Guide 2.0*.
- [274] Nydegger, R. W. 1972. *A Data Minimization Algorithm of Analytical Models for Computer Graphics*. Master's thesis, University of Utah.
- [275] Oat, C. and Sander, P. V. 2007. Ambient aperture lighting. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2007*, pp. 61–64.
- [276] Olano, M. and Kuehne, B. 2002. SGI OpenGL shader level-of-detail. White paper, Silicon Graphics, Inc.
- [277] Olano, M., Kuehne, B., and Simmons, M. 2003. Automatic shader level of detail. In *Proceedings of Graphics Hardware 2003*, pp. 7–14.
- [278] Oliveira, M. M. and Policarpo, F. 2005. An efficient representation for surface details. Tech. Rep. RP-351, Universidade Federal do Rio Grande do Sul.
- [279] Oren, M. and Nayar, S. K. 1994. Generalization of Lambert's reflectance model. In *Proceedings of ACM SIGGRAPH 94*, pp. 239–246.
- [280] O'Sullivan, C., Howlett, S., McDonnell, R., Morvan, Y., and O'Connor, K. 2004. Perceptually adaptive graphics. In *Eurographics 2004 State of the Art Reports*, pp. 141–164.
- [281] Overbeck, R., Ramamoorthi, R., and Mark, W. R. 2007. A real-time beam tracer with application to exact soft shadows. In *Proceedings of Eurographics Symposium on Rendering 2007*, pp. 85–98.
- [282] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. 2008. GPU computing. *Proceedings of the IEEE*, 96 (5), 879–899.
- [283] Owens, J. D., Khailany, B., Towles, B., and Dally, W. J. 2002. Comparing Reyes and OpenGL on a stream architecture. In *Proceedings of Graphics Hardware 2002*, pp. 47–56.

- [284] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26 (1), 80–113.
- [285] Pabst, H.-F., Springer, J. P., Schollmeyer, A., Lenhardt, R., Lessig, C., and Froehlich, B. 2006. Ray casting of trimmed NURBS surfaces on the GPU. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pp. 151–160.
- [286] Pan, H., Feng, X.-F., and Daly, S. 2005. LCD motion blur modeling and analysis. In *Proceedings of IEEE International Conference on Image Processing 2005*, pp. II–21–24.
- [287] Parker, S., Shirley, P., and Smits, B. 1998. Single sample soft shadows. Tech. Rep. UUCS-98-019, University of Utah.
- [288] Patney, A. and Owens, J. D. 2008. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2008)*, 27 (5), art. 143.
- [289] Pattanaik, S. N., Ferwerda, J. A., Fairchild, M. D., and Greenberg, D. P. 1998. A multiscale model of adaptation and spatial vision for realistic image display. In *Proceedings of ACM SIGGRAPH 98*, pp. 287–298.
- [290] Pellacini, F. 2005. User-configurable automatic shader simplification. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2005)*, 24 (3), 445–452.
- [291] Pelli, D. G. 1987. Programming in PostScript: techniques and applications in vision research, plus a survey of PostScript hardware and software. *BYTE*, 12 (5), 185–202.
- [292] Peters, J. 2000. Patching Catmull-Clark meshes. In *Proceedings of ACM SIGGRAPH 2000*, pp. 255–258.
- [293] Peters, J. and Reif, U. 2008. *Subdivision Surfaces*. Springer-Verlag.
- [294] Peters, J. and Shiue, L.-J. 2004. Combining 4- and 3-direction subdivision. *ACM Transactions on Graphics*, 23 (4), 980–1003.
- [295] Peters, R. J. and Itti, L. 2007. Beyond bottom-up: incorporating task-dependent influences into a computational model of spatial attention. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition 2007*.
- [296] Peterson, J. W. 1994. Tessellation of NURB surfaces. In *Graphics Gems IV* (edited by P. S. Heckbert), chap. IV.6, pp. 286–320. Academic Press.
- [297] Pham, T. Q. and van Vliet, L. J. 2005. Separable bilateral filtering for fast video preprocessing. In *Proceedings of IEEE International Conference on Multimedia and Expo 2005*, pp. 454–457.
- [298] Piegl, L. and Tiller, W. 1997. *The NURBS Book*. 2nd ed. Springer-Verlag.
- [299] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. 2007. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum (Proceedings of Eurographics 2007)*, 26 (3), 415–424.

- [300] Pulli, K. and Segal, M. 1996. Fast rendering of subdivision surfaces. In *Proceedings of Eurographics Workshop on Rendering 1996*, pp. 61–70.
- [301] Přikryl, J. 2001. *Radiosity Methods Driven by Human Perception*. Ph.D. thesis, Vienna University of Technology.
- [302] Qu, L. and Meyer, G. W. 2006. Perceptually driven interactive geometry remeshing. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2006*, pp. 199–206.
- [303] Qu, L. and Meyer, G. W. 2008. Perceptually guided polygon reduction. *IEEE Transactions on Visualization and Computer Graphics*, 14 (5), 1015–1029.
- [304] Qu, L., Yuan, X., Nguyen, M. X., Meyer, G. W., Chen, B., and Windsheimer, J. 2006. Perceptually guided rendering of textured point-based models. In *Proceedings of Eurographics Symposium on Point-Based Graphics 2006*, pp. 95–102.
- [305] Ramanarayanan, G., Bala, K., and Ferwerda, J. A. 2008. Perception of complex aggregates. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2008)*, 27 (3), art. 60.
- [306] Ramanarayanan, G., Ferwerda, J., Walter, B., and Bala, K. 2007. Visual equivalence: towards a new standard for image fidelity. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2007)*, 26 (3), art. 76.
- [307] Ramasubramanian, M. 2000. *A Perceptually Based Physical Error Metric for Realistic Image Synthesis*. Master's thesis, Cornell University.
- [308] Ramasubramanian, M., Pattanaik, S. N., and Greenberg, D. P. 1999. A perceptually based physical error metric for realistic image synthesis. In *Proceedings of ACM SIGGRAPH 99*, pp. 73–82.
- [309] Rappoport, A. 1991. Rendering curves and surfaces with hybrid subdivision and forward differencing. *ACM Transactions on Graphics*, 10 (4), 323–341.
- [310] Reddy, M. 1997. *Perceptually Modulated Level of Detail for Virtual Environments*. Ph.D. thesis, University of Edinburgh.
- [311] Reeves, W. T., Salesin, D. H., and Cook, R. L. 1987. Rendering antialiased shadows with depth maps. *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21 (4), 283–291.
- [312] Reimers, M. and Seland, J. 2008. Ray casting algebraic surfaces using the frustum form. *Computer Graphics Forum (Proceedings of Eurographics 2008)*, 27 (2), 361–370.
- [313] Ren, Z., Wang, R., Snyder, J., Zhou, K., Liu, X., Sun, B., Sloan, P.-P., Bao, H., Peng, Q., and Guo, B. 2006. Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2006)*, 25 (3), 977–986.
- [314] Rockwood, A. 1987. A generalized scanning technique for display of parametrically defined surfaces. *IEEE Computer Graphics and Applications*, 7 (8), 15–26.

- [315] Rockwood, A., Heaton, K., and Davis, T. 1989. Real-time rendering of trimmed surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, 23 (3), 107–116.
- [316] Rogers, D. F. 2000. *An Introduction to NURBS With Historical Perspective*. Morgan Kaufmann.
- [317] Rogowitz, B. E. and Rushmeier, H. E. 2001. Are image quality metrics adequate to evaluate the quality of geometric objects? In *Proceedings of SPIE*, vol. 4299 (Human Vision and Electronic Imaging VI), pp. 340–348.
- [318] Rossignac, J. and Borrel, P. 1993. Multi-resolution 3D approximations for rendering complex scenes. In *Geometric Modeling in Computer Graphics*, pp. 455–465.
- [319] Roth, S. H. M., Diezi, P., and Gross, M. H. 2000. Triangular Bézier clipping. In *Proceedings of Pacific Graphics 2000*, pp. 413–414.
- [320] Roth, S. H. M., Diezi, P., and Gross, M. H. 2001. Ray tracing triangular Bézier patches. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, 20 (3), 422–430.
- [321] Salvi, M. 2008. Rendering filtered shadows with exponential shadow maps. In *ShaderX⁶: Advanced Rendering Techniques* (edited by W. Engel), pp. 257–274. Charles River Media.
- [322] Sander, P. V. and Mitchell, J. L. 2005. Progressive buffers: view-dependent geometry and texture LOD rendering. In *Proceedings of Eurographics Symposium on Geometry Processing 2005*, pp. 129–138.
- [323] Schaefer, S. and Goldman, R. 2009. Non-uniform subdivision for B-splines of arbitrary degree. *Computer Aided Geometric Design*, 26 (1), 75–81.
- [324] Schaefer, S. and Warren, J. 2005. On C^2 triangle/quad subdivision. *ACM Transactions on Graphics*, 24 (1), 28–36.
- [325] Schaefer, S. and Warren, J. 2007. Exact evaluation of non-polynomial subdivision schemes at rational parameter values. In *Proceedings of Pacific Graphics 2007*, pp. 321–330.
- [326] Schaefer, S. and Warren, J. 2008. Exact evaluation of limits and tangents for non-polynomial subdivision schemes. *Computer Aided Geometric Design*, 25 (8), 607–620.
- [327] Schaufler, G. 1995. Dynamically generated imposters. In *Proceedings of GI Workshop on Modeling – Virtual Worlds – Distributed Graphics*, pp. 129–135.
- [328] Scherzer, D., Jeschke, S., and Wimmer, M. 2007. Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In *Proceedings of Eurographics Symposium on Rendering 2007*, pp. 45–50.
- [329] Scherzer, D. and Wimmer, M. 2008. Frame sequential interpolation for discrete level-of-detail rendering. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2008)*, 27 (4), 1175–1181.
- [330] Schütz, A. C., Delipetkos, E., Braun, D. I., Kerzel, D., and Gegenfurtner, K. R. 2007. Temporal contrast sensitivity during smooth pursuit eye movements. *Journal of Vision*, 7 (13), art. 3.

- [331] Schwarz, M., Staginski, M., and Stamminger, M. 2006. GPU-based rendering of PN triangle meshes with adaptive tessellation. In *Proceedings of Vision, Modeling, and Visualization 2006*, pp. 161–168.
- [332] Schwarz, M. and Stamminger, M. 2006. Pixel-shader-based curved triangles. In *ACM SIGGRAPH 2006 Research Posters*.
- [333] Schwarz, M. and Stamminger, M. 2007. Bitmask soft shadows. *Computer Graphics Forum (Proceedings of Eurographics 2007)*, 26 (3), 515–524.
- [334] Schwarz, M. and Stamminger, M. 2008. CudaTess: fast GPU-based adaptive tessellation with CUDA. In *Poster Proceedings of Pacific Graphics 2008*, pp. 59–66.
- [335] Schwarz, M. and Stamminger, M. 2008. Microquad soft shadow mapping revisited. In *Eurographics 2008 Annex to the Conference Proceedings (Short Papers)*, pp. 295–298.
- [336] Schwarz, M. and Stamminger, M. 2008. Quality scalability of soft shadow mapping. In *Proceedings of Graphics Interface 2008*, pp. 147–154.
- [337] Schwarz, M. and Stamminger, M. 2009. Fast GPU-based adaptive tessellation with CUDA. *Computer Graphics Forum (Proceedings of Eurographics 2009)*, 28 (2), 365–374.
- [338] Schwarz, M. and Stamminger, M. 2009. Multisampled antialiasing of per-pixel geometry. In *Eurographics 2009 Annex (Short Papers)*, pp. 21–24.
- [339] Schwarz, M. and Stamminger, M. 2009. On predicting visual popping in dynamic scenes. In *Proceedings of Symposium on Applied Perception in Graphics and Visualization 2009*, pp. 93–100.
- [340] Sederberg, T. W. 1985. Piecewise algebraic surface patches. *Computer Aided Geometric Design*, 2 (1–3), 53–59.
- [341] Sederberg, T. W. 1990. Techniques for cubic algebraic surfaces. Part one. *IEEE Computer Graphics and Applications*, 10 (4), 14–25.
- [342] Sederberg, T. W. 1990. Techniques for cubic algebraic surfaces. Part two. *IEEE Computer Graphics and Applications*, 10 (5), 12–21.
- [343] Sederberg, T. W., Cardon, D. L., Finnigan, G. T., North, N. S., Zheng, J., and Lyche, T. 2004. T-spline simplification and local refinement. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2004)*, 23 (3), 276–283.
- [344] Sederberg, T. W. and Chen, F. 1995. Implicitization using moving curves and surfaces. In *Proceedings of ACM SIGGRAPH 95*, pp. 301–308.
- [345] Sederberg, T. W., Finnigan, G. T., Li, X., Lin, H., and Ipson, H. 2008. Watertight trimmed NURBS. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2008)*, 27 (3), art. 79.
- [346] Sederberg, T. W., Zheng, J., Bakenov, A., and Nasri, A. 2003. T-splines and T-NURCCs. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22 (3), 477–484.

- [347] Segal, M., Korobkin, C., van Widenfelt, R., Foran, J., and Haeberli, P. 1992. Fast shadows and lighting effects using texture mapping. *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, 26 (2), 249–252.
- [348] Segovia, B., Iehl, J.-C., Mitanchey, R., and Péroche, B. 2006. Non-interleaved deferred shading of interleaved sample patterns. In *Proceedings of Graphics Hardware 2006*, pp. 53–60.
- [349] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2008)*, 27 (3), art. 18.
- [350] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. 2007. Scan primitives for GPU computing. In *Proceedings of Graphics Hardware 2007*, pp. 97–106.
- [351] Settgast, V., Müller, K., Fünfzig, C., and Fellner, D. 2004. Adaptive tessellation of subdivision surfaces. *Computers & Graphics*, 28 (1), 73–78.
- [352] Shade, J., Gortler, S., He, L., and Szeliski, R. 1998. Layered depth images. In *Proceedings of ACM SIGGRAPH 98*, pp. 231–242.
- [353] Shantz, M. and Chang, S.-L. 1988. Rendering trimmed NURBS with adaptive forward differencing. *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, 22 (4), 189–198.
- [354] Sharma, G., Wu, W., and Dalal, E. N. 2005. The CIEDE2000 color-difference formula: implementation notes, supplementary test data, and mathematical observations. *Color Research & Application*, 30 (1), 21–30.
- [355] Sharp, B. 1999. Optimizing curved surface geometry. *Game Developer Magazine*, 6 (7), 40–48.
- [356] Shirley, P., Ashikhmin, M., Gleicher, M., Marschner, S. R., Reinhard, E., Sung, K., Thompson, W. B., and Willemsen, P. 2005. *Fundamentals of Computer Graphics*. 2nd ed. A K Peters.
- [357] Shiue, L.-J., Jones, I., and Peters, J. 2005. A realtime GPU subdivision kernel. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2005)*, 24 (3), 1010–1015.
- [358] Sintorn, E., Eisemann, E., and Assarsson, U. 2008. Sample based visibility for soft shadows using alias-free shadow maps. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2008)*, 27 (4), 1285–1292.
- [359] Sloan, P.-P., Govindaraju, N. K., Nowrouzezahrai, D., and Snyder, J. 2007. Image-based proxy accumulation for real-time soft global illumination. In *Proceedings of Pacific Graphics 2007*, pp. 97–105.
- [360] Snyder, J. and Nowrouzezahrai, D. 2008. Fast soft self-shadowing on dynamic height fields. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2008)*, 27 (4), 1275–1283.

- [361] Soler, C. and Sillion, F. X. 1998. Fast calculation of soft shadow textures using convolution. In *Proceedings of ACM SIGGRAPH 98*, pp. 321–332.
- [362] St-Amour, J.-F., Paquette, E., and Poulin, P. 2005. Soft shadows from extended light sources with penumbra deep shadow maps. In *Proceedings of Graphics Interface 2005*, pp. 105–112.
- [363] Stam, J. 1998. Evaluation of Loop subdivision surfaces. In *CD-ROM Proceedings of ACM SIGGRAPH 98*.
- [364] Stam, J. 1998. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of ACM SIGGRAPH 98*, pp. 395–404.
- [365] Stam, J. and Loop, C. 2003. Quad/triangle subdivision. *Computer Graphics Forum*, 22 (1), 79–85.
- [366] Steketee, J. J. 1991. Step length in forward differencing for curved surface display. *The Visual Computer*, 8 (1), 35–46.
- [367] Stoll, C., Gumhold, S., and Seidel, H.-P. 2006. Incremental raycasting of piecewise quadratic surfaces on the GPU. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pp. 141–150.
- [368] Stürzlinger, W. 1998. Ray tracing triangular trimmed free form surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 4 (3), 202–214.
- [369] Sundstedt, V., Stavakis, E., Wimmer, M., and Reinhard, E. 2008. A psychophysical study of fixation behavior in a computer game. In *Proceedings of Symposium on Applied Perception in Graphics and Visualization 2008*, pp. 43–50.
- [370] Sweeney, M. A. J. and Bartels, R. H. 1986. Ray tracing free-form B-spline surfaces. *IEEE Computer Graphics and Applications*, 6 (2), 41–49.
- [371] Tan, P., Lin, S., Quan, L., Guo, B., and Shum, H.-Y. 2005. Multiresolution reflectance filtering. In *Proceedings of Eurographics Symposium on Rendering 2005*, pp. 111–116.
- [372] Tan, P., Lin, S., Quan, L., Guo, B., and Shum, H.-Y. 2008. Filtering and rendering of resolution-dependent reflectance models. *IEEE Transactions on Visualization and Computer Graphics*, 14 (2), 412–425.
- [373] Tatarchuk, N. 2007. Real-time tessellation on GPU. In: *ACM SIGGRAPH 2007: Courses*, course 28 (Advanced Real-Time Rendering in 3D Graphics and Games).
- [374] Tatarchuk, N. 2008. Advanced topics in GPU tessellation: algorithms and lessons learned. Presentation, *Gamefest 2008*.
▷ [http://developer.amd.com/gpu_assets/Tatarchuk-Tessellation\(Gamefest2008\).pdf](http://developer.amd.com/gpu_assets/Tatarchuk-Tessellation(Gamefest2008).pdf)
- [375] Tatarinov, A. 2008. Instanced tessellation in DirectX10. Presentation, *Game Developers Conference 2008*.
▷ http://developer.download.nvidia.com/presentations/2008/GDC/Inst_Tess_Compatible.pdf

- [376] Thompson, W. B., Smits, B., Shirley, P., Kersten, D. J., and Madison, C. 1998. Visual glue. Tech. Rep. UUCS-98-007, University of Utah.
- [377] Tolhurst, D. J., Ripamonti, C., Párraga, C. A., Lovell, P. G., and Troscianko, T. 2005. A multiresolution color model for visual difference prediction. In *Proceedings of Symposium on Applied Perception in Graphics and Visualization 2005*, pp. 135–138.
- [378] Tookey, R. and Cripps, R. 1997. Improved surface bounds based on derivatives. *Computer Aided Geometric Design*, 14 (8), 787–791.
- [379] Toth, D. L. 1985. On ray tracing parametric surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*, 19 (3), 171–179.
- [380] Ulrich, T. 2002. Rendering massive terrains using chunked level of detail control. In: *ACM SIGGRAPH 2002 Course Notes*, course 35 (Super-Size It! Scaling Up to Massive Virtual Worlds).
- [381] van der Horst, G. J. C. and Bouman, M. A. 1969. Spatiotemporal chromaticity discrimination. *Journal of the Optical Society of America*, 59 (11), 1482–1488.
- [382] Van Horn, R. B., III and Turk, G. 2008. Antialiasing procedural shaders with reduction maps. *IEEE Transactions on Visualization and Computer Graphics*, 14 (3), 539–550.
- [383] Van Nes, F. L. and Bouman, M. A. 1967. Spatial modulation transfer in the human eye. *Journal of the Optical Society of America*, 57 (3), 401–406.
- [384] van Overveld, C. W. A. M. and Wyvill, B. 1997. Phong normal interpolation revisited. *ACM Transactions on Graphics*, 16 (4), 397–419.
- [385] Velho, L. and de Figueiredo, L. H. 1996. Optimal adaptive polygonal approximation of parametric surfaces. In *Anais do SIBGRAPI 96*, pp. 127–133.
- [386] Velho, L., Gomes, J., and de Figueiredo, L. H. 2002. *Implicit Objects in Computer Graphics*. Springer-Verlag.
- [387] Vlachos, A., Peters, J., Boyd, C., and Mitchell, J. L. 2001. Curved PN triangles. In *Proceedings of ACM Symposium on Interactive 3D Graphics 2001*, pp. 159–166.
- [388] Von Herzen, B. and Barr, A. H. 1987. Accurate triangulations of deformed, intersecting surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21 (4), 103–110.
- [389] Vroomen, J. and de Gelder, B. 2000. Sound enhances visual perception: cross-modal effects of auditory organization on vision. *Journal of Experimental Psychology: Human Perception and Performance*, 26 (5), 1583–1590.
- [390] Wallis, B. 1990. Tutorial on forward differencing. In *Graphics Gems* (edited by A. S. Glassner), chap. XI.5, pp. 594–603. Academic Press.
- [391] Walter, B., Pattanaik, S. N., and Greenberg, D. P. 2002. Using perceptual texture masking for efficient image synthesis. *Computer Graphics Forum (Proceedings of Eurographics 2002)*, 21 (3), 393–399.

- [392] Walther, D. 2006. *Interactions of Visual Attention and Object Recognition: Computational Modeling, Algorithms, and Psychophysics*. Ph.D. thesis, California Institute of Technology.
- [393] Wandell, B. A. 1995. *Foundations of Vision*. Sinauer Associates.
- [394] Ward Larson, G., Rushmeier, H., and Piatko, C. 1997. A visibility matching tone reproduction operator for high dynamic range scenes. *IEEE Transactions on Visualization and Computer Graphics*, 3 (4), 291–306.
- [395] Warren, J. and Weimer, H. 2001. *Subdivision Methods for Geometric Design: A Constructive Approach*. Morgan Kaufmann.
- [396] Watson, A. B. 1987. The cortex transform: rapid computation of simulated neural images. *Computer Vision, Graphics, and Image Processing*, 39 (3), 311–327.
- [397] Watson, A. B. and Ahumada, A. J., Jr. 2005. A standard model for foveal detection of spatial contrast. *Journal of Vision*, 5 (9), 717–740.
- [398] Williams, L. 1978. Casting curved shadows on curved surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 78)*, 12 (3), 270–274.
- [399] Williams, L. 1983. Pyramidal parametrics. *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, 17 (3), 1–11.
- [400] Williams, N., Luebke, D., Cohen, J. D., Kelley, M., and Schubert, B. 2003. Perceptually guided simplification of lit, textured meshes. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics 2003*, pp. 113–121.
- [401] Windsheimer, J. E. and Meyer, G. W. 2004. Implementation of a visual difference metric using commodity graphics hardware. In *Proceedings of SPIE*, vol. 5292 (Human Vision and Electronic Imaging IX), pp. 150–161.
- [402] Woo, A., Poulin, P., and Fournier, A. 1990. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10 (6), 13–32.
- [403] Wu, X. and Peters, J. 2005. An accurate error measure for adaptive subdivision surfaces. In *Proceedings of International Conference on Shape Modeling and Applications 2005*, pp. 51–56.
- [404] Wyman, C. and Hansen, C. 2003. Penumbra maps: approximate soft shadows in real-time. In *Proceedings of Eurographics Symposium on Rendering 2003*, pp. 202–207.
- [405] Xia, J. C. and Varshney, A. 1996. Dynamic view-dependent simplification for polygonal models. In *Proceedings of IEEE Visualization '96*, pp. 327–334.
- [406] Yang, L., Sander, P. V., and Lawrence, J. 2008. Geometry-aware framebuffer level of detail. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2008)*, 27 (4), 1183–1188.
- [407] Yarbus, A. L. 1967. Eye movements during perception of complex objects. In *Eye Movements and Vision*, pp. 171–211. Plenum Press.

- [408] Yee, H. 2004. A perceptual metric for production testing. *Journal of Graphics Tools*, 9 (4), 33–40.
- [409] Yee, H., Pattanaik, S., and Greenberg, D. P. 2001. Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments. *ACM Transactions on Graphics*, 20 (1), 39–65.
- [410] Yoon, S., Gobbetti, E., Kasik, D., and Manocha, D. 2008. *Real-Time Massive Model Rendering*. Synthesis Lectures on Computer Graphics and Animation. Morgan & Claypool.
- [411] Yoon, S.-E., Salomon, B., Gayle, R., and Manocha, D. 2004. Quick-VDR: interactive view-dependent rendering of massive models. In *Proceedings of IEEE Visualization 2004*, pp. 131–138.
- [412] Zelsnack, J. 2004. GLSL pseudo-instancing. Tech. rep., NVIDIA Corporation.
▷ http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/OpenGL/src/glsl_pseudo_instancing/docs/glsl_pseudo_instancing.pdf
- [413] Zeng, W., Daly, S., and Lei, S. 2002. An overview of the visual optimization tools in JPEG 2000. *Signal Processing: Image Communication*, 17 (1), 85–104.
- [414] Zhang, R.-J. and Ma, W. 2006. Some improvements on the derivative bounds of rational Bézier curves and surfaces. *Computer Aided Geometric Design*, 23 (7), 563–572.
- [415] Zhang, X. and Wandell, B. A. 1996. A spatial extension of CIELAB for digital color image reproduction. *SID International Symposium Digest of Technical Papers*, 27, 731–734.
- [416] Zheng, J. and Sederberg, T. W. 2000. Estimating tessellation parameter intervals for rational curves and surfaces. *ACM Transactions on Graphics*, 19 (1), 56–77.
- [417] Zhongke, W., Feng, L., Soon, S. H., and Yun, C. K. 2004. Evaluation of difference bounds for computing rational Bézier curves and surfaces. *Computers & Graphics*, 28 (4), 551–558.
- [418] Zorin, D. and Kristjansson, D. 2002. Evaluation of piecewise smooth subdivision surfaces. *The Visual Computer*, 18 (5–6), 299–315.
- [419] Zorin, D. and Schröder, P. 1999. Subdivision for modeling and animation. *ACM SIGGRAPH 99 Course Notes*.
- [420] Zorin, D. and Schröder, P. 2000. Subdivision for modeling and animation. *ACM SIGGRAPH 2000 Course Notes*.
- [421] Zorin, D., Schröder, P., and Sweldens, W. 1996. Interpolating subdivision for meshes with arbitrary topology. In *Proceedings of ACM SIGGRAPH 96*, pp. 189–192.