Towards a High Quality Real-Time Graphics Pipeline

Jacob Munkberg
Department of Computer Science
Lund University

LUND INSTITUTE OF TECHNOLOGY
Lund University

ISBN 978-91-976939-3-6
ISSN 1404-1219
Dissertation 34, 2011
LU-CS-DISS:2011-1

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: jacob@cs.lth.se
WWW: http://www.cs.lth.se/home/Jacob_Munkberg

Typeset using LaTeX2ε
Printed in Sweden by Tryckeriet i E-huset, Lund, 2011

Abstract

Modern graphics hardware pipelines create photorealistic images with high geometric complexity in real time. The quality is constantly improving and advanced techniques from feature film visual effects, such as high dynamic range images and support for higher-order surface primitives, have recently been adopted.

Visual effect techniques have large computational costs and significant memory bandwidth usage. In this thesis, we identify three problem areas and propose new algorithms that increase the performance of a set of computer graphics techniques. Our main focus is on efficient algorithms for the real-time graphics pipeline, but parts of our research are equally applicable to offline rendering.

Our first focus is *texture compression*, which is a technique to reduce the memory bandwidth usage. The core idea is to store images in small compressed blocks which are sent over the memory bus and are decompressed on-the-fly when accessed. We present compression algorithms for two types of texture formats. *High dynamic range images* capture environment lighting with luminance differences over a wide intensity range. *Normal maps* store perturbation vectors for local surface normals, and give the illusion of high geometric surface detail. Our compression formats are tailored to these texture types and have compression ratios of $6:1$, high visual fidelity, and low-cost decompression logic.

Our second focus is *tessellation culling*. Culling is a commonly used technique in computer graphics for removing work that does not contribute to the final image, such as completely hidden geometry. By discarding rendering primitives from further processing, substantial arithmetic computations and memory bandwidth can be saved. Modern graphics processing units include flexible tessellation stages, where rendering primitives are subdivided for increased geometric detail. Images with highly detailed models can be synthesized, but the incurred cost is significant. We have devised a simple remapping technique that allows for better tessellation distribution in screen space. Furthermore, we present programmable tessellation culling, where bounding volumes for displaced geometry are computed and used to conservatively test if a primitive can be discarded *before* tessellation. We introduce a general tessellation culling framework, and an optimized algorithm for rendering of displaced Bézier patches, which is expected to be a common use-case for graphics hardware tessellation.

Our third and final focus is forward-looking, and relates to efficient algorithms for *stochastic rasterization*, a rendering technique where camera effects such as depth of field and motion blur can be faithfully simulated. We extend a graphics pipeline with stochastic rasterization in spatio-temporal space and show that stochastic motion blur can be rendered with rather modest pipeline modifications. Furthermore, backface culling algorithms for motion blur and depth of field rendering are presented, which are directly applicable to stochastic rasterization. Hopefully, our work in this field brings us closer to high quality real-time stochastic rendering.

Acknowledgements

First of all, I would like to thank my main supervisor Tomas Akenine-Möller for his enthusiasm, intuition and invaluable support throughout my thesis work. I also wish to thank my assistant supervisors Lennart Ohlsson, Jacob Ström, and Michael Doggett for their support and insights.

I would like to extend my gratitude to my colleagues at the Lund University Graphics Group for all the great discussions and collaborations; Jon Hasselgren, Petrik Clarberg, Jim Rasmusson, Magnus Andersson, Björn Johnsson and Calle Lejdfors. My sincere thanks to Robert Toth, the Advanced Rendering Technology team at Intel, everyone at former Illuminate Labs, Masahiro Takatsuka at University of Sydney, David Bonner and Loïc Le Feuvre at Dassault Systèmes, and all the masters students that I have supervised.

The work presented in this thesis was carried out within the Computer Graphics group at the Department of Computer Science, Lund University. It was partly funded by Stiftelsen för Strategisk Forskning (SSF) and Intel Corporation.

Finally, I owe my deepest gratitude to Jodi Kelly, my family, and my friends for their encouragement, inspiration, and for always believing in me.

Preface

This thesis summarizes my research in high-quality real-time rendering.
The following papers are included:

I. Jacob Munkberg, Petrik Clarberg, Jon Hasselgren and
Tomas Akenine-Möller,
"High Dynamic Range Texture Compression for Graphics Hardware",
in *ACM Transactions on Graphics*, 25(3):698–706, 2006.

II. Jacob Munkberg, Petrik Clarberg, Jon Hasselgren and
Tomas Akenine-Möller,
"Practical HDR Texture Compression",
in *Computer Graphics Forum*, 27(6):1664–1676, 2008.

III. Jacob Munkberg, Tomas Akenine-Möller and Jacob Ström,
"High-Quality Normal Map Compression",
in *Graphics Hardware*, pages 95–102, 2006.

IV. Jacob Munkberg, Ola Olsson, Jacob Ström and Tomas Akenine-Möller,
"Tight Frame Normal Map Compression",
in *Graphics Hardware*, pages 37–40, 2007.

V. Tomas Akenine-Möller, Jacob Munkberg and Jon Hasselgren,
"Stochastic Rasterization using Time-Continuous Triangles",
in *Graphics Hardware*, pages 7–16, 2007.

VI. Jacob Munkberg, Jon Hasselgren, Masahiro Takatsuka
and Tomas Akenine-Möller,
"Non-Uniform Fractional Tessellation using Edge Blending",
A short version of this paper is published as:
"Non-Uniform Fractional Tessellation",
in *Graphics Hardware*, pages 41–45, 2008.

VII. Jon Hasselgren, Jacob Munkberg and Tomas Akenine-Möller,
"Automatic Pre-Tessellation Culling",
in *ACM Transactions on Graphics*, 28(2):19, 2009.

VIII. Jacob Munkberg, Jon Hasselgren, Robert Toth and
Tomas Akenine-Möller,
"Efficient Bounding of Displaced Bézier Patches",
in *High Performance Graphics*, pages 153–162, 2010.

IX. Jacob Munkberg and Tomas Akenine-Möller,
"Backface Culling for Motion Blur and Depth of Field",
in *journal of graphics, gpu and game tools* (to appear).

The following papers are also published but are not included in this thesis:

- Erik Månsson, Jacob Munkberg and Tomas Akenine-Möller
  "Deep Coherent Ray Tracing"
  in *IEEE - Symposium on Interactive Ray Tracing 2007*, pages 79–85, 2007

- Jacob Ström, Per Wennersten, Jim Rasmusson, Jon Hasselgren,
  Jacob Munkberg, Petrik Clarberg and Tomas Akenine-Möller,
  "Floating-Point Buffer Compression in a Unified Codec Architecture",
  in *Graphics Hardware*, pages 75–84, 2008.

Contents

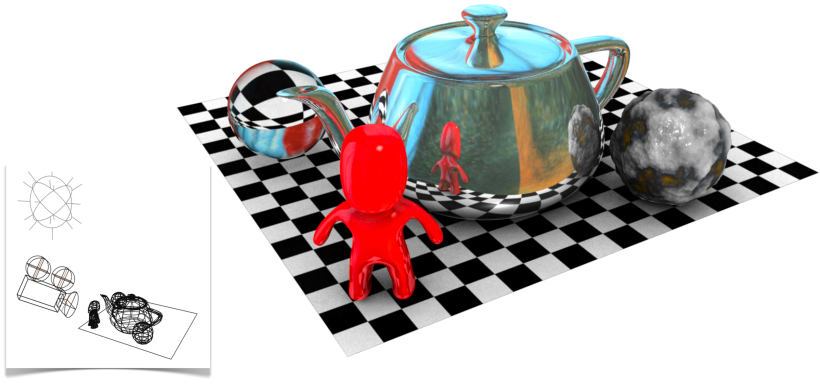


Figure 1: The left image shows a simple three-dimensional scene with a set of geometric models, a camera, and a light source. On the right side is the rendered image from the camera's point of view, with materials applied to all the objects. By carefully computing the interaction between lights and materials, a realistic image can be obtained.

1 Introduction

In the field of computer graphics, collections of three-dimensional geometric models are transformed into realistic-looking images. A scene description is created by positioning models into the scene, assigning materials and adding light sources. The scene is viewed from a virtual camera and the interactions between lights and materials are computed from that viewing position. The result is a computer-generated image. Figure 1 shows a simple scene with five objects, each with a unique material. The process of computing the color of each pixel on screen from a three-dimensional scene description is often referred to as *rendering* an image.

The ability to visualize three-dimensional objects by rendering realistic images has widespread use. Medical visualizations help doctors diagnose symptoms and prepare for surgery. In biology, complex three-dimensional structures of proteins and other substances are better understood when visualized. In computer-aided geometric design, virtual prototypes of new products, such as cars and houses, can be presented to the client with the option to interactively replace the materials and colors. This enables more control and reduced costs during product development. Several geographic map applications allow for three-dimensional navigations directly in a browser, and with the HTML5 standard, low-level 3D graphics APIs, such as WebGL [36] are under development, which support interactive rendering of three-dimensional scenes in web browsers. The most widespread use of computer graphics, however, is in the entertainment industry, which includes computer games and visual effects in feature films and advertising. Most recent feature films include computer graphics to some degree. These effects are often obvious, such as when an extinct dinosaur attacks a car, but can be more subtle, as when a camera,

capturing an ordinary scene in a room, suddenly moves through a key hole.

In this introductory chapter, we will first introduce computer graphics rendering. Thereafter, an overview of the graphics pipeline is presented in Section 2. In Section 3, we summarize our contributions and research methodology. Section 4 presents texture compression and the first four papers of this thesis. Section 5 discusses culling in graphics pipelines and introduces the following three papers. Section 6 describes stochastic rasterization, which is related to the two remaining papers. Finally, in Section 7, we conclude the thesis and present a few directions for future work.

Rendering can be broadly divided in two categories, namely *offline* and *real-time* rendering. The focus of this thesis is on the latter, but we will give a brief summary of both aspects to give more context to our work. Furthermore, the two categories are converging, as faster processors and better algorithms allow for more advanced visual effects at interactive frame rates.

1.1 Offline Rendering

In *offline* rendering, hours or even days are spent on rendering a single high quality computer generated image. Visual quality is the top priority, and realistic camera models and advanced simulations of light and material interaction are used to create stunning, often photorealistic images. Use cases include, for example, feature films, high-end commercials, architectural prototypes, and product visualizations.

Visual effects departments at the big production houses often work for years on each feature film to create spectacular visual effects, using massive computational resources. Big render farms with thousands of processors [43] are used, with rendering times on the order of hours per frame. Great care is taken to match computer generated content with live action footage.

To generate photorealistic images, one needs to solve the so called *rendering equation* [30]:

$$L_o(x,\omega) = L_e(x,\omega) + \int_\Omega f_r(x,\omega',\omega) L_i(x,\omega')(\omega' \cdot \mathbf{n}) d\omega'. \quad (1)$$

A thorough discussion of this equation is beyond the scope of this thesis. Please refer to Jensen's book [29] for a detailed discussion of the involved terms. At a high level, this equation states that the outgoing radiance,[1] L_o, from one surface point x, in the direction ω, is the sum of the emitted radiance, L_e, from x in the direction ω and the total radiance which is scattered towards x from all directions ω' over the hemisphere Ω. The term $f_r(x,\omega',\omega)$ is the *bidirectional reflectance distribution function* (BRDF), which returns the proportion of light reflected from the direction ω' towards the direction ω at position x. Loosely speaking, the BRDF determines the surface material properties.

[1] Radiance is the amount of light that passes through or is emitted from a particular area, and falls within a given solid angle in a specified direction.

Solving this equation involves computing the interaction of light for all primitives in the scene, which is a daunting task. A large collection of algorithms has been proposed to approximate the rendering equation, often involving tracing a large number of rays that bounce around in the scene or by computing the light transfer between pairs of surface elements. Typically, the algorithms require efficient sampling of the entire scene from any point within its bounds. An important area of computer graphics research concerns data structures for holding and querying the scene hierarchy and efficient scene traversal strategies. For a detailed overview of the field of *physically based rendering*, please refer to Pharr and Humphreys' book [44]. The memory access patterns of graphics sampling algorithms are often incoherent. For example, Monte Carlo methods that estimate hemispherical integrals (e.g. the integral over Ω in Equation 1) may trace a set of randomly distributed rays from a large set of surface points. Efficient geometry caching schemes are difficult to implement, which makes rendering of scenes with high geometric complexity a challenging task.

To circumvent the need of building and storing traversal structures of the scene geometry in memory, the REYES architecture [13] approaches the problem somewhat differently. The initial design goals included managing massive amounts of highly detailed geometry, high-quality anti-aliasing, and camera effects like motion blur and depth of field. By sacrificing support for secondary ray effects, such as true reflection rays and indirect illumination, each geometric primitive is only processed once, and can be streamed through the pipeline. Furthermore, each geometric primitive is recursively subdivided into grids of pixel-sized primitives, called *micropolygons*. These are then displaced and shaded in wide SIMD-batches. Each generated micropolygon is visibility tested at a rate decoupled from the shading rate. Such a renderer is denoted a *micropolygon* renderer. Pixar's RenderMan [2, 13] renderer was designed around these concepts, and is highly popular in the visual effects industry, partly due to the ability to handle vast amounts of geometric complexity, but also due to a flexible shading system with programmable materials. Movies such as Toy Story were rendered using a micropolygon renderer, and although the geometric detail is high, the absence of highly reflective materials and advanced lighting effects is rather obvious.

Currently, most offline visual effects use a combination of micropolygon and physically based renderers depending on the look of the particular shot. Each renderer outputs large sets of layers which are composited together in post-processing passes. One could easily conclude that visual effects can be created more efficiently today. A large number of algorithms for efficient offline rendering have been published over the last decades [44]. Furthermore, for more than 40 years, the number of transistors per unit area in computer hardware has doubled approximately every two years, following Moore's law [40]. Many computer graphics techniques can indeed be generated in shorter amount of time today, but in general, the additional processor resources and algorithmic improvements have instead largely been invested in the creation of even more advanced visual effects within the same time-budget. The total production time remains approximately

constant [24]. Looking forward, we anticipate a continuing strong demand for high quality computer graphics. Currently, 3D cinema and television are gaining momentum and display resolutions are still increasing. Both these trends will require even more computational resources and efficient rendering algorithms.

1.2 Real-Time Rendering

In *real-time* rendering, interactivity is critical, because it allows the end user to move the camera, change material parameters or move objects around in the scene, for example. Applications include computer games, scientific visualizations, flight simulators, 3D in web browsers, and pre-visualizations for feature films. To meet these needs, coarser approximations to the rendering equation (Equation 1) are used, where effects such as reflections and soft shadows are approximated instead of being sampled with a large number of rays. Similar to the REYES architecture discussed above, the requirement of holding the entire scene hierarchy in memory has been removed and the geometry is instead *streamed* through a graphics pipeline. Each primitive is processed in turn and then discarded. A streaming model is more amenable for a hardware implementation as complex hierarchical scene representations are avoided. For performance reasons in real-time rendering, primitives are typically not subdivided to pixel-sized polygons as in the REYES architecture, and camera effects such as motion blur and depth of field are not supported directly. However, many effects can be approximated by rendering multiple passes and combining them. A thorough overview of real-time rendering is given in the book by Akenine-Möller et al. [1].

In offline rendering, visual fidelity is the end goal, where each individual image may take hours to render. In real-time rendering, an interactive frame rate is a requirement, which limits the output to the best possible rendered image that can be obtained in less than, say, 40 ms. To make the problem more constrained, the form factors of consumer computational devices are decreasing. Power consumption and heat dissipation are often the limiting factors.

To attack this problem, dedicated hardware has been designed for real-time rendering, where billions of triangles are streamed through the pipeline every second. Great effort is put into reducing memory bandwidth usage and power, by, for example, using compressed data formats and fixed function processing units for some tasks. To retain flexibility, the graphics hardware pipeline also contains a number of programmable stages, similar to REYES, which allow for geometry displacements, animations, and a wide range of user-defined materials. With dedicated hardware and a graphics pipeline programming model that extracts the inherit parallelism in computer graphics rendering, visualizations that a decade ago required a server rack the size of a refrigerator can now be rendered in real time on a smartphone.

2 The Graphics Pipeline

In this section, we discuss the graphics pipeline in order to provide context to the new compression, culling, and stochastic rasterization algorithms presented in this thesis. We start by motivating the need for dedicated graphics hardware and the concept of a graphics pipeline, followed by a description of each stage of a modern graphics pipeline. In upcoming sections, we will focus on performance improvements for this model with further details regarding our contributions.

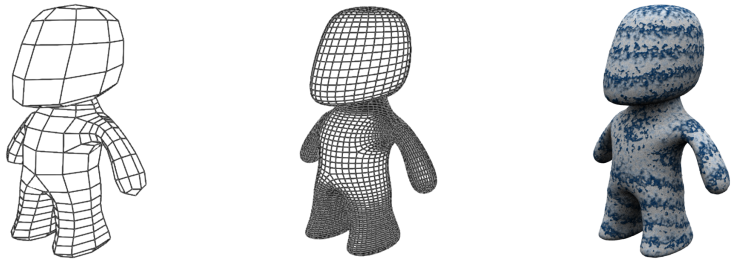
As an initial example, consider increasing the contrast of a full screen photograph displayed on a computer display. The display contains a large number of picture elements, *pixels*, each representing a unique color value at one point on the display. The task of increasing the contrast of the photograph is equivalent to increasing the contrast of each individual pixel. In computer science terminology, this translates into executing a short program, that takes the pixel's color value as input and outputs a modified color value, for every pixel on screen. A modern display contains more than two million pixels, which makes even a simple operation expensive when computed over all pixels.

Note that many instances of the same program are executed independently over different data (in this case, pixels). Hence, this is a massively parallel problem. In general, computer graphics workloads have similar characteristics to our initial example, where one program, denoted a *shader*, is executed over, e.g., a large number of pixels or geometric primitives. Geometric primitives are often represented by meshes of triangles, and each point in such a mesh, describing a point in space, is denoted a *vertex*. Vertex shaders and pixel shaders are the most commonly used shader programs in real-time graphics pipelines.

Dedicated graphics hardware is designed for these large parallel workloads. Instead of having one, or a few, complex processors, large sets of simpler processors are packed together in wide SIMD units, where each set share instruction fetch and decode logic [18]. Within each set of processors, the *same* shader program is executed with different data for each pixel. Similar computations are executed for large blocks of pixels, and at one extreme, one processing unit can be attached to each pixel [20].

The set of packed SIMD-processors are called *shader cores*, which are designed for high arithmetic throughput and to cope with long latencies. Each shader core has a small amount of local memory and high bandwidth to the graphics processor's memory system. Shader programs commonly request values from images, which are denoted *textures*, stored in the graphics card's memory. Textures are presented in further detail in Section 4.

The *graphics pipeline* abstracts the underlying hardware details from the graphics programmer, while providing a restricted programming model that expresses the inherent parallelism in rendering. For example, each vertex of a geometric model can be transformed in parallel and each pixel can be colored independently. The current graphics pipeline has evolved from a fixed-function pipeline to a pro-



Input model Tessellated model Shaded model

Figure 2: The graphics pipeline takes three-dimensional models as input, positions the models, adds geometric detail, and applies materials and lighting.

grammable pipeline, where many of the stages execute user-provided shader programs [7, 22, 49].

In a simplified form of the pipeline, the graphics programmer writes a shader that determines how *one* individual vertex should be transformed and another shader that contains instructions of how the color of *one* pixel should be computed. The underlying hardware then schedules and dispatches all instances of vertex and pixel shader programs. This scheduling process is completely hidden from the user. The different shader programs are applied to a large set of primitives, and have clearly defined frequencies of operation, such as geometry transforms on vertex level and shading at pixel level.

At a coarse level, the graphics pipeline can be divided into three main stages: *geometry processing*, where three-dimensional models are positioned and animated. Additionally, the input geometry can be refined if needed. *Rasterization* follows next, where the visibility of each primitive for a specific camera position is determined. Finally, in the *pixel processing* phase, materials are applied to the visible parts of the geometry and the pixel color is computed and written into the resulting image. Figure 2 shows this process for a single mesh.

In the next subsection, we will give a brief overview of the stages in a modern graphics processing unit (GPU).

2.1 The Graphics Hardware Pipeline

In the presentation below, we use a simplified version of the recently introduced Direct 3D 11 graphics pipeline [14] as a basis for our discussion. For clarity of description, we have omitted the *geometry shader* [7] and the *compute shader* [10], as those stages are not directly relevant for the work presented in this thesis.

The pipeline consists of both fixed-function and programmable stages. All programmable stages of the pipeline can access texture images through the GPU's memory system. Figure 3 shows a schematic overview of this pipeline. We start

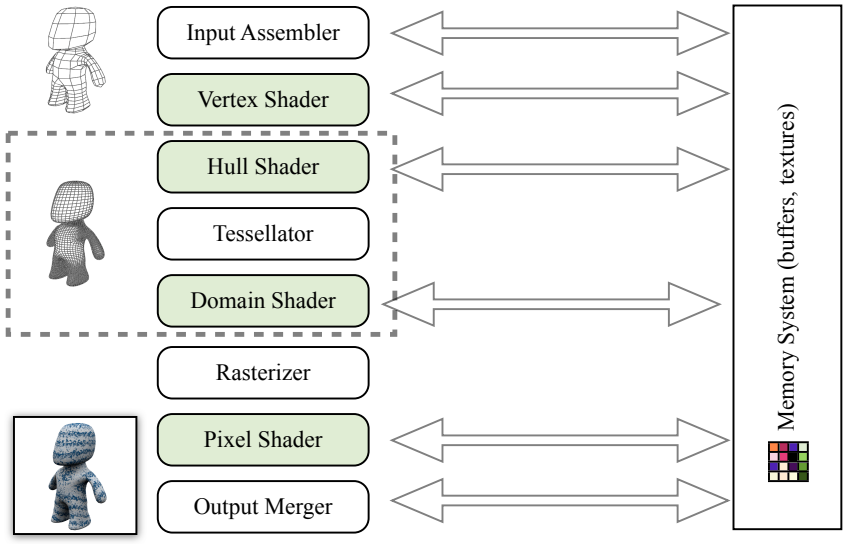


Figure 3: A simplified GPU pipeline. At the top, three-dimensional models are fed to the pipeline. These are positioned in space, and optionally geometric detail is added in the tessellation stages (dashed box). On-screen visibility is determined in the rasterizer. Materials and lighting are then applied in the pixel shader. The green stages represent programable pipeline stages. Note that many stages interact with the memory system.

from the top, where geometry data from the application is received at the graphics processor. Three-dimensional models are represented as collections of triangles, quadrilaterals or higher order primitives, where the latter are parametric surface patches. In addition to geometric primitives, the application uploads shader programs for the programmable stages and buffers holding constant values and specific rendering state, such as tessellation method, light positions, and camera parameters. A set of texture images needed for the shader programs are also uploaded to the GPU's memory system. With this information available from the application, we are ready to step through the pipeline stages.

Input Assembler The first stage of the pipeline reads arrays containing vertex data, such as arrays of positions, normal vectors and texture coordinates, and assembles vertex structures from the individual arrays. The members of the vertex structure are called *vertex attributes*, Often, a separate index list with offsets into the vertex arrays is used for more compact storage, and to avoid transforming the same vertex more than once. The assembled arrays of vertex structures are then sent to the next stage of the pipeline, the *vertex shader*. Consecutive indices in the arrays form rendering primitives, typically triangles.

Vertex Shader The vertex shader is a user-provided program executed once for each vertex structure in the assembled vertex array. The program outputs a transformed position and all attributes needed by shader programs later in the pipeline. A minimal vertex shader takes a three-dimensional position, expressed in a coordinate system local to the geometric object, and multiplies it with a matrix that transforms the position into the camera *clip space*. This is a coordinate system independent of the scene scale, convenient for clipping operations against the camera's view volume. The vertex shader program can, as all programmable shader stages, access texture images through the graphics processing unit's memory system and use those values in the computation of the transformed vertex position. Other vertex attributes, such as normal vectors and texture coordinates can also be modified in this stage. If the tessellation stages are enabled, the vertex structures do not represent geometric positions, but animated patch control points, which are passed on to the hull shader stage. If tessellation is disabled, the vertices are sent directly to the rasterizer.

Hull Shader The following three pipeline stages interact to support flexible geometry amplification. These are the *hull shader*, a fixed-function *tessellator*, and the *domain shader*. The hull shader interprets the transformed positions from the vertex shader as control points for a parametric patch. In the graphics pipeline, parametric patches are represented by a set of control points and a parametric surface domain. They are evaluated and tessellated into a large set of small triangles in the two downstream tessellation stages. The hull shader is applied to the control points of an input patch, and has two tasks. First, a user-provided program is executed per control point of the input patch, typically to change the patch's control point basis. The transformed control points are passed directly to the domain shader. The second task is to execute a user-provided program that computes tessellation factors for the edges of the patch, which are provided to the next stage, the tessellator.

Tessellator This is a fixed-function unit that, given a patch, tessellation factors from the preceding hull shader, and state parameters, generates a set of barycentric coordinates and connectivity in a planar domain inside the patch. In modern GPU pipelines, the tessellator works on triangular and quad patches and has a set of modes, including uniform and fractional tessellation, where the latter tessellation algorithm enables smoothly varying surface detail. The barycentric positions are fed into the next pipeline stage.

Domain Shader This program is executed once for each generated barycentric position from the tessellator and outputs a displaced vertex. Figure 4 shows the domain shader executed on all points within a triangular patch. The shader has access to the transformed control points from the hull shader, and typically evaluates a parametric surface, such as a bi-degree Bézier surface, at a certain parametric

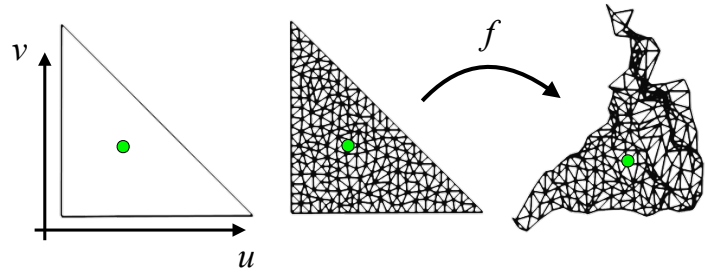


Figure 4: A triangular surface patch (left) is tessellated into a mesh of triangles (middle), and a domain shader (f) is executed at each vertex of the mesh to create a highly detailed surface (right). The green point shows one barycentric position within the patch, where one instance of the domain shader is executed.

coordinate. Additionally, the shader may use texture images to add local surface detail. When the tessellation stages are enabled, the domain shader takes the role of the vertex shader, and transforms the geometric positions into the camera clip space. The transformed vertices are then passed on to the rasterizer.

The three tessellation stages are not as flexible as the adaptive split-dice tessellation algorithm [32] used in REYES offline rendering. However, recent research has shown that the tessellation stages can be combined to represent advanced surface representations, such as animated approximate subdivision surfaces [33, 34].

Rasterizer The first task of the rasterizer is to set up triangles from consecutive indices in the vertex arrays. After triangle setup, the next task is to determine which pixels on screen that are covered by each triangle. At this point in the pipeline, the triangles have been transformed by the vertex or domain shader into the camera clip space. The rasterization stage handles clipping of primitives to the view volume and performs culling operations, which are further described in Section 5. After clipping and culling, the clip space coordinates are projected on screen and visibility is determined. There are several traversal strategies for the visibility test. One simple approach is to, for each triangle, compute a bounding box on screen and for each screen space visibility sample within this bounding box, test if the sample is covered by the triangle. This process is called *scan conversion* or *rasterization*. Figure 5 shows the rasterization process for two overlapping triangles in screen space.

To determine if the triangle covers a certain screen space sample position, a common technique is to represent each triangle edge by a linear function, which has a value greater than zero on one side of the edge and less than zero on the other. By evaluating these three *edge functions* for a screen space sample position and looking at the resulting signs, the inside status can be determined [45].

More efficient rasterization approaches include hierarchical visibility testing, where

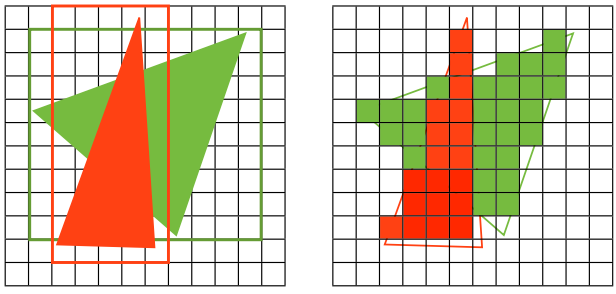


Figure 5: The rasterization process for two triangles. All pixels in the bounding box of each triangle are tested. If the triangle covers the pixel center, a fragment is generated. In this example, only the fragment closest to the camera is stored for each pixel.

tiles of sample positions are coverage tested. Rasterization can be performed in screen space after clipping [38], or directly in homogeneous coordinates [37, 41]. If a sample is covered, a *fragment* is generated. For each covered fragment, the vertex attributes are interpolated using the triangle's barycentric coordinates at the sample's hit point. The interpolated vertex attributes, including the depth at the hit point, are stored in the fragments. The fragments are then sent to the next pipeline stage, the pixel shader. In most graphics hardware pipelines to date, the rasterizer is a fixed-function unit.

Pixel Shader The rasterizer outputs a set of covered sample positions for each triangle, and the next step is to determine the color of these fragments. For each fragment generated by the rasterizer, the final color is computed by a user-defined program, representing a surface material, that takes the fragment's interpolated attributes as input. These pixel shader programs often contain several texture map lookups and advanced material descriptions, that take the local surface normal and the light sources in the scene into account.

Returning to Equation 1, the bi-directional reflection distribution function (BRDF), $f_r(x, \omega', \omega)$, is an important building block in pixel shaders, as it describes how the surface interacts with light and is a function of the incoming (ω') and outgoing (ω) light direction for each surface point x. To determine the radiance at a specific surface point, the contribution from all incoming light directions are accumulated to determine the total outgoing radiance in the viewing direction determined by the current pixel and the camera origin. This radiance is stored as the pixel color. Figure 6 shows a set of pixel shaders with different characteristics applied to five spheres.

Pixel shader programs request textures from memory using dedicated hardware for accelerated filtered texture lookups. The pixel shader program is often the main performance bottleneck in terms of both execution cycles and memory bandwidth usage in graphics workloads, mainly due to the vast number of fragments that are

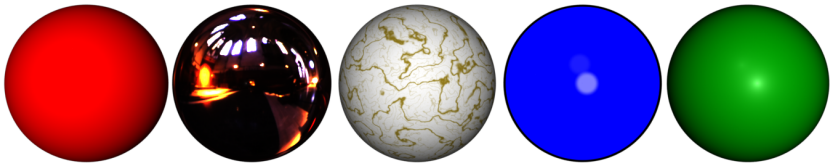


Figure 6: A set of different pixel shaders applied to five spheres.

needed to generate an image.

The pixel shader is followed by a *depth test*, and if the pixel shader is guaranteed not to modify the depth interpolated in the rasterizer, this test can sometimes be moved before the pixel shader. For fully opaque fragments overlapping the same sample, the depth values of the currently processed sample is compared to the value stored in a *depth buffer*. The depth test is configurable, and a commonly used test is *less than*: only if the current fragment's depth is closer than the depth buffer value, the current color value is written to that sample position in the frame buffer, and the corresponding entry in the depth buffer is updated. The same test, but on a coarser depth buffer, can be performed earlier in the pipeline, and is called *occlusion culling*. This culling technique will be further described in Section 5.

Output Merger In the final stage of the GPU pipeline, shaded fragments are blended into the frame buffer, taking color and transparency of each fragment into account. Once the entire scene has been fully processed, the resulting color buffer is displayed on screen. The output merger is typically a fixed-function stage, with a user-configurable blending mode.

3 Contributions and Methodology

This thesis presents a collection of algorithms that reduces the computations and bandwidth requirements needed to render high quality images. The end goal is higher visual quality within the same computational budget, or consistent visual quality with computational savings. The algorithms we have developed are designed to fit in modern graphics pipelines, harnessing their strengths and reducing some of the bottlenecks, both in terms of computational requirements, memory bandwidth usage, and power consumption. Our contributions can be divided into three subcategories: *texture compression*, *tessellation culling*, and *stochastic rasterization*. All three categories aim to improve the graphics pipeline, and in Figure 7, we show which parts of the pipeline the different algorithms apply to.

To reduce memory bandwidth usage, we present texture compression formats, designed to respect graphics hardware constraints such as fast random access and minimal decompression complexity. We present block-based compression for-

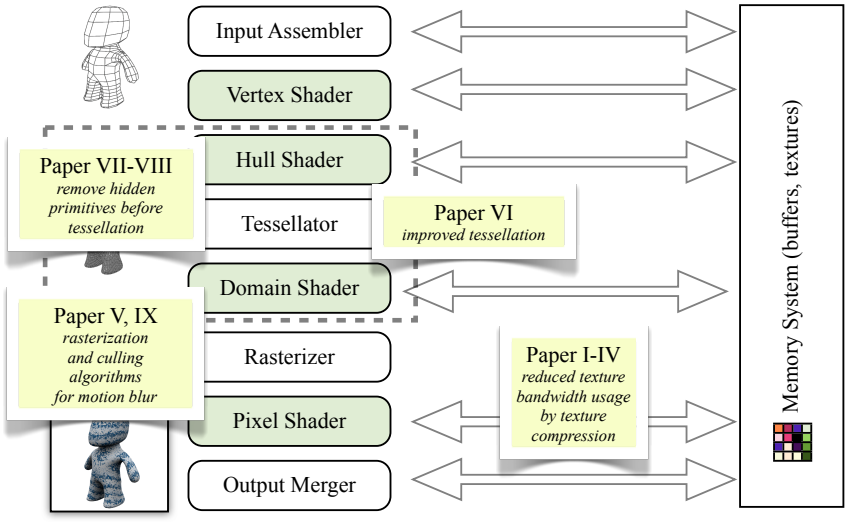


Figure 7: The graphics pipeline with notes summarizing, for each paper in this thesis, which aspect of the GPU pipeline they improve.

mats for *high dynamic range textures* and *normal maps*, which are two texture types with extended bit depths compared to standard digital images. Texture compression formats developed for standard images are not directly applicable. If texture formats with high bit rates are used uncompressed, they consume a large amount of memory bandwidth. Our research addresses this problem by presenting new compressed formats for these texture types, with compression ratios of 6 : 1, simple decompression logic, and high visual quality.

For this research project, we have constructed a framework where floating-point images can be compressed and decompressed. Here, new compression algorithms can easily be prototyped and carefully evaluated with a set of image quality metrics. Visual quality evaluation of high dynamic range image compression techniques is an active field of research [35], and we present a new error metric for comparing high dynamic range textures. More details about texture mapping in general and our compression algorithms are given in Section 4.

To approach the problem of tessellation culling, we used a software graphics pipeline simulator framework developed in the graphics group at Lund University by Jon Hasselgren and Tomas Akenine-Möller. We extended this simulator with tessellation stages, allowing us to implement tessellation culling algorithms and measure their efficiency on relevant workloads.

Here, the core contribution is two culling algorithms that, by careful analysis of the programmable tessellation shaders, can discard rendering primitives even before they reach the fixed-function tessellator of a GPU pipeline. Primitives are

discarded only if they are guaranteed not to contribute to the final image. The culling algorithms are therefore conservative, which means that the resulting image, rendered with culling enabled, is identical to the image rendered with culling disabled. The evaluation focus in this part of the research was therefore on the incurred cost of executing the culling, versus the costs saved by removing rendering primitives from downstream processing. The efficiency of the culling algorithms for various workloads was also measured. The presented culling techniques apply both to real-time graphics pipelines and to offline REYES-like renderers.

During this research project, we also devised a remapping technique to obtain better tessellation quality given a fixed tessellation rate. By warping the barycentric coordinates output from the tessellator, a more uniform tessellation distribution in screen space is obtained. Tessellation culling techniques for the graphics pipeline and our algorithms are discussed further in Section 5.

With a graphics hardware pipeline simulated in software at hand, we also investigated how to extend this pipeline to handle stochastic rasterization. The visibility test in a standard rasterizer was generalized to handle moving primitives, with generalized *time-dependent* edge equations for visibility testing. Furthermore, new texture formats storing unique samples for each time interval were developed, which allowed for motion blurred shadows and reflections. Finally, we developed generalized, conservative backface culling tests for the case of motion blur and depth of field.

To evaluate image quality, the stochastic rasterizer was compared with *accumulation buffering*, which is a workaround to obtain high quality motion blur on current GPUs, where the fixed-function rasterizer cannot be configured to handle the temporal dimension. The evaluation focused on visual quality and memory bandwidth usage. For the generalized backface culling tests, we investigated the need of a truly conservative backface test in the presence of motion blur and depth of field, and estimated the computational overhead of correct backface culling. More details about stochastic rasterization and our algorithms are given in Section 6.

My main supervisor, Tomas Akenine-Möller, has been actively involved in all the research projects presented in this thesis, including algorithmic design and in writing the articles.

In summary, the goal of the research presented in this thesis is to allow for more advanced visual effects in real-time rendering, by cleverly compressing data, removing unnecessary work, and adapting stochastic rasterization to fit into modern graphics pipelines. In the next sections, each of these topics is discussed in further detail, starting with texture compression, followed by culling techniques, and stochastic rasterization. In each focus area, the individual papers are introduced and briefly discussed.

4 Texture Compression for Graphics Hardware

As mentioned in Section 2, the programmable shader stages of the graphics pipeline often need access to textures in memory. Our first focus is to reduce the memory bandwidth usage between the shader cores and the GPU's memory system (see Figure 7 and Figure 8).

A single pixel shader program may access many different textures, and as these programs are executed over millions of pixels, the memory bandwidth usage caused by the texture accesses can become a bottleneck in the graphics pipeline. For example, an advanced skin shader may use more than fifteen different texture maps [17].

Texture compression is a widely used technique to reduce this bandwidth usage. The first part of this thesis presents a set of new compressed texture formats targeting textures with floating-point values per channel. Before discussing these algorithms in detail, we give a brief overview of texture mapping.

4.1 Texture Mapping

In general, a set of attributes is attached to each triangle vertex, containing, for example, a normal vector, color values, and texture coordinates. For a fragment generated by the rasterizer, texture coordinates specified at each vertex are interpolated to give a unique position within the texture for that fragment. The interpolated texture coordinates are used to access a small set of texels in a texture image stored in memory. This technique can, for example, be used to paste "decals" onto a three-dimensional model.

Texture maps are used in many places in the graphics pipeline. As seen in Figure 7, all programmable pipeline stages can access texture maps. Pixel shaders often use textures to determine the diffuse and specular albedos of the surface material. Other textures contain perturbation vectors to the shading normals (*normal mapping*). Environment maps can be represented either by set of textures forming a cube (*cube map*) or by a photograph of a reflective sphere, capturing the entire view of the environment (*sphere map*). These maps can be used to simulate highly reflective materials or to light a scene using a photograph. In Figure 1, an environment map is used to simulate the chrome material of the teapot.

Texture maps are often stored as three-channel RGB images with $3 \times 8 = 24$ bits per pixel (bpp). This gives $2^8 = 256$ values per color channel, which is sufficient to represent decals, diffuse, and specular albedos. However, some texture formats require higher bit rates for sufficient quality, such as textures representing slowly varying normal vectors. Another example is floating-point environment maps, which are commonly used in offline rendering to carefully match real footage with rendered images [16, 47]. Vertex and domain shaders use texture maps with higher bit rates for a technique called *displacement mapping*, where the geometry is locally displaced according to a value stored in a texture image.

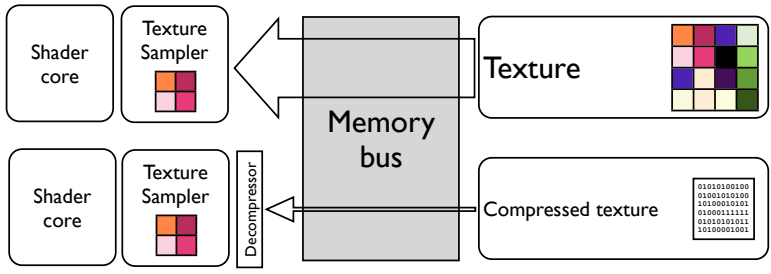


Figure 8: A schematic illustration of a texture request from a shader, with and without compression. Textures can be stored in compressed form in memory. When a region of the texture is requested, a compressed block is sent over the memory bus, and is decompressed by dedicated logic close to the shader cores. With this approach, the memory bandwidth usage can be significantly reduced.

Altogether, texture mapping is a powerful technique that adds flexibility and control when designing a shader. It is very frequently used, and is an integral part in both real-time and offline rendering to approach photorealistic rendering. The main drawback is the added memory bandwidth usage. The gap between computing power and memory bandwidth is increasing for standard CPUs, and the compute versus memory access ratio is even higher for dedicated graphics hardware [42]. In terms of energy, a memory operation from external DRAM requires about $20-2000\times$ the power of an arithmetic operation [15, 21]. Hence, it is critical to reduce the memory bandwidth usage as much as possible, which is the main motivation for compressed texture representations.

4.2 Texture Compression

By storing the textures in compressed form in memory and decompressing them on-the-fly when they are accessed, memory bandwidth usage can be traded for additional computations. This is well-known by graphics hardware designers, and in modern GPUs, texture maps are accessed through dedicated hardware units called texture samplers. To allow for fast access at any position within the texture image, textures are commonly divided into small blocks of, say 4×4 texels, which are individually compressed and stored in the GPU memory in blocks of a fixed size. This is in contrast to commonly used image compression techniques, such as JPEG, where variable encoding is a key to achieve high compression rates. The blocks are decompressed on-the-fly by dedicated hardware [4, 31, 50]. Figure 8 illustrates this high-level concept. Note that texture compression may in some cases increase the level of detail, as larger, compressed texture maps can be stored in the same amount of off-chip memory as smaller uncompressed textures.

To produce a filtered texture value, the color values of four to eight individual texels are typically needed. High quality anisotropic texture filtering requires an

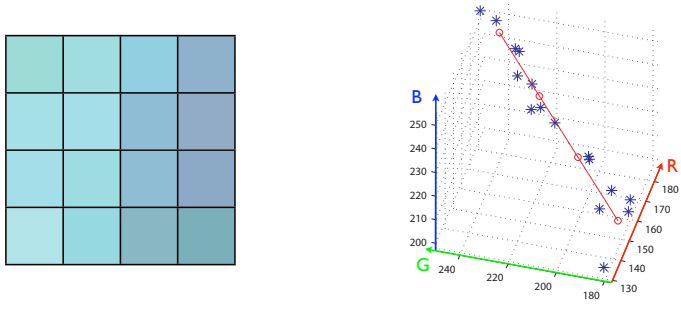


Figure 9: S3 texture compression (S3TC/DXTC) illustrated. The color values in RGB triples from a block of 4×4 *texels are plotted in a Cartesian coordinate system, and a line is fitted to the points. In the compression stage, each texel (blue stars) is assigned to one of four quantized positions (red circles) along this line.*

even larger set of texels. Hence, texture caches that exploit the spatial locality in texture requests are commonly used to reduce the number of individual memory transfers [25, 27].

Texture accesses are read-only, so the encoding of a texture into a compressed representation is a one-time process, that can be performed offline. In contrast, the texture will be decompressed many times during rendering, which motivates dedicated decompression hardware and compression formats with fast decompression algorithms. The design criteria for texture compressions are: minimal quality loss, high compression rates, and low-complexity decompression.

S3 texture compression [28] (S3TC, also called DXTC) is one commonly used scheme for compressing 24 bpp textures. It uses blocks of 4×4 pixels that are compressed from $16 \times 24 = 384$ bits to 64 bits, giving a compression ratio of $6:1$. Two base colors are stored in 16 bits (RGB565) each. These base colors form a line segment in RGB-space. Two additional colors are placed along the line, in between the base colors, creating a local color map of four colors. Figure 9 shows an example of this technique. For each of the 4×4 pixels within the block, a two-bit index is stored, assigning one of the four colors to the pixel. Finding the best line for a set of 16 pixels in the encoding phase requires some computations, but the decompression algorithm is very simple: decode the base colors, recreate the four-color palette from the two base colors, and assign a value to each pixel. The quality of the reconstructed texture is acceptable in most cases for 24 bpp textures. Variants of this technique with more colors along the line, or more than one line are also commonly used [14].

The first four papers of this thesis present new compression algorithms for two types of textures used in graphics hardware pipelines; *normal maps* and *high dynamic range textures*. In the next two subsections, we discuss these texture types and present our compression algorithms.

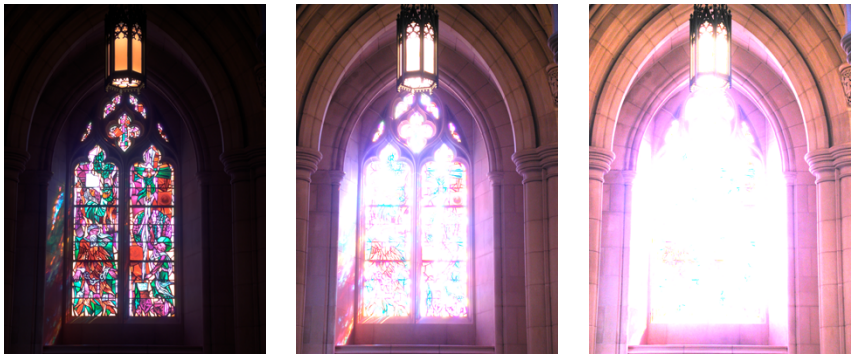


Figure 10: A single high dynamic range (HDR) image displayed at three different exposures. Note the different details appearing in each exposure. Due to the increased bit depth, more image information is retained. This image is Copyright ©2004, Industrial Light & Magic, a division of Lucasfilm Entertainment Company Ltd.

4.3 High Dynamic Range Textures

Standard texture maps use 8 bits per color channel, which is not always enough when capturing large lighting variations. Modern SLR camera sensors use 14 bits or more per channel in their RAW formats, and to faithfully capture large lighting variations, a 16 or 32 bit floating-point number per color channel may be needed. A texture with a floating-point representation per color channel is called a high dynamic range (HDR) image. These image formats are convenient in computer graphics, as they can accurately capture the light of an environment. Therefore, HDR images are commonly used to faithfully light computer generated models [47]. Furthermore, with this additional information, the exposure of one HDR image can be modified over a large range of values. Figure 10 shows three different exposures of an HDR image with a large range of luminance values.

The main drawback is again the additional storage requirements. Compared to a three-channel 8-bit texture with a storage cost of 24 bpp, a three channel 16-bit HDR texture requires 48 bpp, which is a $2\times$ increase in storage cost and bandwidth usage. Compression formats for standard textures with lower bit rates are not designed for HDR and give poor quality when naïvely applied.

The first two papers in this thesis, Paper I and Paper II, address this problem, and present texture compression algorithms designed especially for high dynamic range textures. The compression ratio is $6:1$, which is the same ratio as standard compression schemes for 24 bpp texture maps, such as S3TC. Our algorithm uses a color space transform to separate the luminance and chrominance information of the images, and introduces a set of shapes, that inexpensively capture chrominance information within 4×4 texel blocks. The luminance is logarithmically encoded and stored as quantized values along a one-dimensional line, similar to the S3TC

format, but with higher precision and more quantized values. The decompressor is of low complexity and the visual quality is very high over a large exposure range. Furthermore, we present new error metrics for visual quality evaluation of HDR textures. The dynamic range of the texture allows for a wider range of possible viewing conditions with different exposures, which makes standard image quality metrics unsuitable.

Paper I was, together with a similar HDR texture scheme developed independently by Roimela et al. [48], one of the two first published HDR texture compression algorithms. Both formats were presented at the ACM SIGGRAPH conference in 2006. Their algorithm has a strong focus on minimal decompression complexity, while our main focus is on high image quality, with a slightly higher decompression complexity. Paper II extends the compression algorithm with an additional mode with higher chrominance quality, and discusses a set of practical details with regards to HDR texture compression. We also show that the compression algorithm can be used as a high quality mode for standard 24 bpp textures.

More recently, the Direct3D 11 graphics API included a block compression format for HDR textures called BC6H [14].

The research presented in Paper I and Paper II was a joint project between myself, Petrik Clarberg, Jon Hasselgren and Tomas Akenine-Möller. I was the primary author of the two papers and contributed to the conceptual design of the algorithms, implementation, evaluation and in writing the articles. My major implementation focus was on the chrominance shape transform algorithms and the framework used for evaluation of texture compression algorithms.

4.4 Normal Mapping

High geometric detail is often desired for rendering, but it comes with a high cost. For most rendering and animation algorithms, the performance degrades significantly when the geometric detail is increased. Geometric complexity is therefore one important knob in trading rendering speed for quality. For small geometric features, a commonly used approximation is to store a perturbed shading normal in a texture image, which changes the local appearance of the surface's interaction with light [6]. Figure 11 illustrates this technique. This is a convincing approximation in the interior of a mesh, but as the true geometry is unmodified, the object's silhouette is left unchanged. Also, local self-shadowing is not captured. Still, this is an extremely wide-spread technique for real-time graphics.

Similarly to HDR textures, the main drawbacks with normal mapping are again the storage costs and added texture bandwidth usage. Normal maps stored at 24 bpp are not always sufficient to represent smoothly varying normals fields, and offline rendering and modeling applications support normal maps with 16 and 32 bits per vector component for higher quality. Furthermore, texture compression formats designed for standard color texture maps do not perform well on normal data [3].

In Paper III, we introduce a normal map compression format that extends ATI's

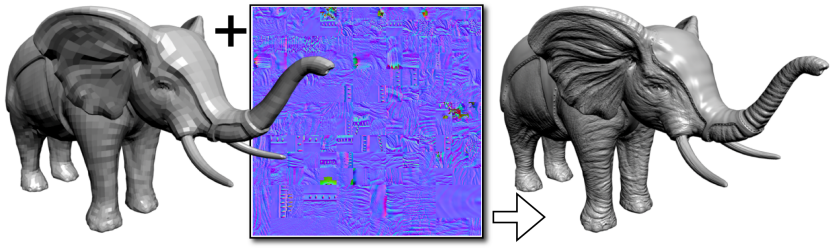


Figure 11: Normal mapping. Left: a coarsely tessellated mesh is rendered with a plastic material using the face normal of each facet of the mesh for shading. Right: the local surface normal is perturbed with a vector fetched from a texture at every pixel, which gives the illusion of increased geometric detail. The elephant model and normal map are courtesy of Pixologic Inc.

normal map compression format 3Dc [3] with a set of new compression modes for higher quality, but with the same compression rate and retained backwards compatibility. The 3Dc format encodes the *xy*-components of the normal as two separate channels. A one-dimensional S3TC-like algorithm is used for each channel, so the algorithm can be seen as creating a grid of uniformly distributed sample points in the *xy*-plane. The third normal component, z, is reconstructed from the *xy* coordinates by using the constraint that normal vectors have unit length. In our algorithm, we exploit the correlation between the x and y channels for higher quality. Our additional modes use rotated coordinate frames, variable point distribution, and differential encoding which results in a compression format with higher image quality on our set of test images. The additional cost is a slightly more expensive decompression algorithm.

Paper IV presents a similar normal map compression algorithm which is more robust for directed features in normal maps. Inspired by the *shape transform* encoding from our HDR texture format (Paper I), we use two points and a scale factor to encode a local two-dimensional coordinate frame that tightly enclose the normals in a 4×4 texel block. It is equivalent to, or better than, previous algorithms on a large set of normal maps. However, it comes with a slightly higher decompression cost and is no longer backwards compatible with 3Dc. Both papers include visual quality analyses using appropriate error metrics and high-level hardware decompression proposals.

This was a joint research project with Jacob Ström, Ola Olsson and Tomas Akenine-Möller. I was the primary author of both papers and contributed in writing the paper and in all parts of the algorithm design, implementation, and evaluation, except for the hardware decompressor proposals.

5 Geometry Culling Techniques

The second focus of this thesis concerns geometry culling for the graphics pipeline. Returning to Figure 7, the task at hand is to reduce the geometry amplification introduced by the tessellation stages by carefully removing primitives that do not contribute to the rendered image. In the tessellation pipeline stages, each input primitive, typically a parametric triangular or rectangular surface patch, may be subdivided into thousands of small triangles. This drastically increases the geometry workload, as each generated triangle must be processed through the remainder of the pipeline. Culling techniques can potentially remove many of the generated triangles, which translates into computational savings.

Our research in this field is based on bounding the output values of domain shaders, given a restricted input interval, so that spatial bounds for all positions within the patch *after* domain shading can be computed. This allows for conservative culling of patches even before the tessellator is invoked. We first give a brief overview of culling techniques in the graphics pipeline, and then describe our algorithms in detail.

5.1 Culling in Graphics Pipelines

In general, if a geometric primitive can be removed early in the pipeline, without affecting the final image quality, a substantial amount of downstream work can be avoided. As a result, total performance may increase. Culling techniques are used in a graphics pipeline to remove work from further processing. Figure 12 shows the three most common culling techniques applied to geometric primitives in a rasterization pipeline. *View frustum culling* removes objects that are entirely outside the camera's view volume. *Occlusion culling* removes objects completely hidden by already drawn objects. *Backface culling* discards triangles whose face normals point away from the camera.

Culling in graphics processors are typically applied in the fixed-function rasterization unit, *after* the geometry has been transformed by the programmable vertex and domain shader stages. In the rasterizer, geometric primitives are expressed in camera clip space, where view frustum culling and backface culling tests have simple expressions. Occlusion culling is commonly performed by testing the primitive against a hierarchical coarse depth buffer [23] prior to inside testing.

Coarse view frustum and occlusion culling can also be performed on the application side, where the programmer can avoid sending primitives, which are guaranteed to be outside the camera frustum, or occluded, to the graphics processor. Furthermore, if tessellation is enabled, we can get substantial gains if a patch can be culled before the tessellator, which avoids unnecessary domain shader evaluations. However, as discussed in Section 2, many of the pipeline stages are programmable, which means that in practice, a user-provided program may displace each vertex to an arbitrary position. This makes conservative culling early in the pipeline, or

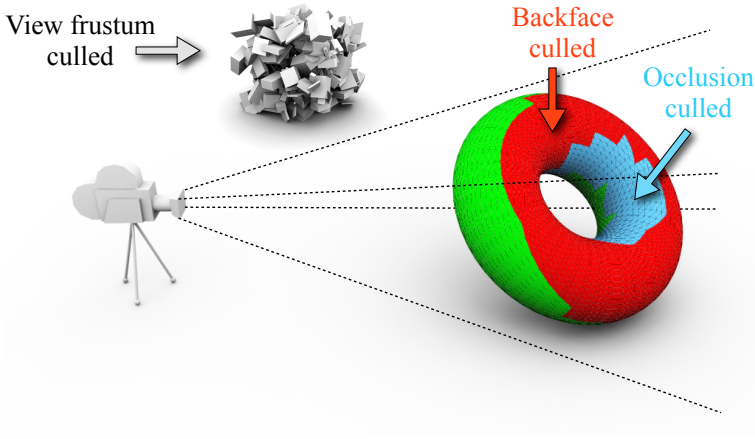


Figure 12: Culling in a rasterization pipeline. From the given camera position, the only visible parts of the scene are the triangles in the green area of the torus. All other geometry could potentially be removed from further processing. The red region of the torus can be backface culled, as all triangles in that region have face normals pointing away from the camera. From the camera's point of view, triangles in the blue region are completely covered by geometry closer to the camera, and can be occlusion culled if the triangles are rendered in front-to-back order. The pile of white cubes in the upper left corner can be view frustum culled, as it is entirely outside the camera view frustum (dashed lines).

on the application side, very difficult in many cases.

5.2 Pre-Tessellation Culling

Paper VII introduces a technique which carefully analyzes the shader programs that compute the final vertex positions. It generates conservative bounds that can be used to cull primitives *before* tessellation is applied.

Figure 13 shows a triangular patch that is tessellated and displaced in a domain shader. The input to one particular instance of the domain shader is a barycentric coordinate, and it outputs a displaced point. If the domain shader is applied to all barycentric positions from the tessellator, the result is a displaced patch, illustrated to the right.

Our technique translates the domain shader into a *bounding shader* by expressing each shader instruction in a bounded arithmetic. The input to the bounding shader is a barycentric domain, which in this case is the entire barycentric domain within the triangular patch. The bounding shader outputs a bounding box, conservatively enclosing the entire displaced patch. This is shown in the lower part of Figure 13. This bounding box is then used for view frustum and occlusion culling, and similar bounds can be obtained for the surface normal's variation over the patch, allowing

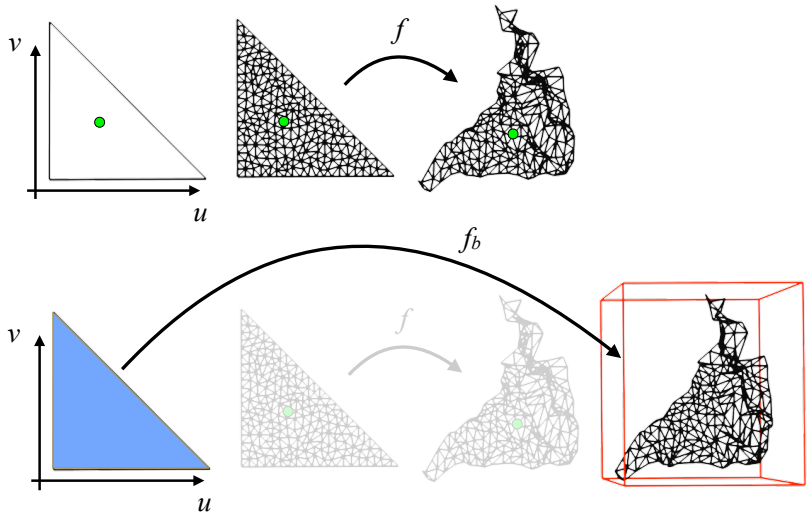


Figure 13: The upper row shows a triangular patch (left) that is tessellated (middle), and each tessellated point is displaced in a domain shader, denoted f. The final displaced surface (right) is obtained by applying the domain shader to each barycentric point in the tessellation of the patch's domain. One particular instance of the domain shader is highlighted in green. The lower row shows the result of our bounding shader, denoted f_b, that takes the entire barycentric domain as input and outputs a bounding box, which is guaranteed to contain the displaced surface.

a per-patch backface culling test. The bounding shader is more complex than the corresponding domain shader, and this technique is expected to give performance improvements if the total cost of executing the bounding shaders is less than the total cost of all domain shader evaluations multiplied by the culling rate.

Our algorithm is based on previous work on programmable culling techniques for pixel shaders [26], where conservative estimates for the range of output values for a block of pixels were derived. For tessellation culling, we use a more elaborate bounded arithmetic, called Taylor models [5], that is adapted to the problem of bounding domain shaders expressing higher-order surface evaluations. Our work was evaluated in an instrumented graphics pipeline simulator framework, where culling rates and costs of the new bounding shader stages were measured. When the input patches are highly tessellated, the proposed technique showed significant gains on most test scenes, making it suitable not only for GPU tessellation workloads but especially for micropolygon offline renderers, where all primitives are diced into pixel-sized primitives (See Section 1.1).

The automatic tessellation culling project was joint work with Jon Hasselgren and Tomas Akenine-Möller. I was not the primary author of this paper, but was involved in all aspects of the research project. My implementation focus was on

the tessellation system of the pipeline simulator and the support for approximate subdivision surfaces.

The technique in Paper VII handles a large variety of domain shaders, but may come at a high cost due to its generality. In particular, obtaining accurate bounds for the normal vector variation over the patch can be expensive. A common use case for the tessellation stages of a modern GPU pipeline is *displaced Bézier patches*. These are often bi-cubic patches, represented by 16 control points and a displacement texture image that stores fine-grained surface detail. This can be seen as a compact representation of high geometric detail, and simplifies animation, as it can be performed on the control point level. Displaced Bézier surfaces are also popular approximations of displaced subdivision surfaces [33, 34].

Paper VIII presents a bounding technique fine-tuned for this common use case. We obtained higher culling rates at a lower cost than the general technique, mostly due to a more efficient bounding algorithm for the surface normal. We also designed and evaluated a Direct3D 11 hull shader implementation of the algorithm that runs interactively on a modern GPU, and improves performance on a complex tessellation example, representing approximate subdivision surfaces.

This was joint work with Jon Hasselgren, Robert Toth and Tomas Akenine-Möller. I was primary author and involved in all aspects of the paper. My implementation focus was on algorithmic design and evaluation.

The tessellator in a modern GPU is flexible and contains a set of tessellation modes that allow for smooth transitions from densely to coarsely tessellated regions. One drawback is that each tessellated patch is, except for the outer row of vertices, near-uniformly tessellated in object space. This can be sub-optimal for large patches seen in perspective, where the triangle distribution is denser further away from the camera. In Paper VI, we present a simple remapping approach that warps the distribution of points, generating a tessellation that is often more uniform in screen space.

This was a joint project with Jon Hasselgren, Tomas Akenine-Möller and Masahiro Takatsuka. I was primary author of the paper and involved in all aspects of the work.

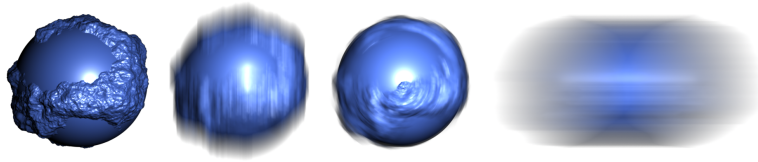
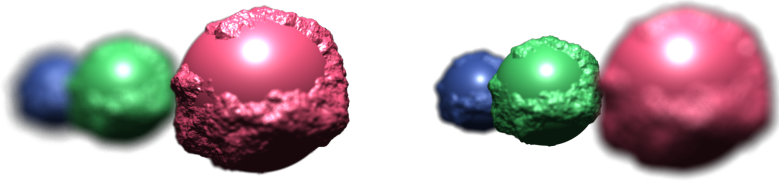


Figure 14: A simple object (left) is animated and rendered using motion blur by stochastic rasterization. As can be seen, the blur gives a hint of how the object is moving.



Focus at near object Focus at middle object

Figure 15: Depth of field gives an important visual cue to direct the attention of the viewer, and is frequently used by photographers and directors. These images were rendered with stochastic rasterization.

6 Stochastic Rasterization

For increased levels of realism, effects such as motion blur and depth of field are needed. These effects are integral parts of offline rendering systems to create smooth animations (decrease temporal aliasing) and to direct the viewers' attention. The REYES architecture [13] used in RenderMan [2] supports these effects by using *stochastic sampling*. This is a technique where samples are randomly distributed over, for example, the lens area and in the time interval when the camera shutter is open. With enough samples, stochastic sampling faithfully renders effects needed for high quality rendering, such as motion blur, depth of field, and glossy reflections. The images in Figure 14 and Figure 15 were rendered using this technique, using samples distributed in time and over the lens respectively. High quality depth of field and motion blur through stochastic rasterization are computationally expensive techniques, due to the high sampling rates needed for noise-free images. There is no direct support for stochastic sampling in current graphics hardware pipelines, so these effects are often approximated, or rendered using multi-pass techniques.

Paper V presents a graphics pipeline, targeting real-time rendering, with added support for stochastic sampling in the temporal dimension. The main difference is a modified rasterizer, that stochastically rasterizes moving triangles. A *time-continuous* triangle is shown in Figure 16, where each vertex moves linearly in the

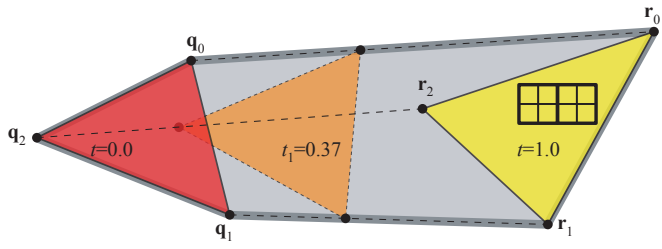


Figure 16: A time-continuous triangle is a moving triangle with linear vertex motion in world space. In this example, the triangle sweeps out a volume in clip space, moving from left to right during the exposure interval. As can be seen, the two drawn quads are only covered by the moving triangle in a small temporal interval towards the end of the exposure interval.

interval when the camera shutter is open.

A modified inside test is executed for each sample covered by a tight bounding box of the time-continuous triangle. For this purpose, *time-dependent* edge equations were introduced, that extend standard homogeneous rasterization [37, 41] with a temporal dimension. These time-dependent edge equations determine if the primitive overlaps with a screen space sample position at a given time. The graphics pipeline requires that pixel shaders are executed over 2×2 blocks of pixels in order to estimate shader derivatives based on finite differences. We derive stochastic sampling patterns that respect this constraint. Furthermore, we introduce *time-dependent* texture maps that can be queried for a certain time interval, which enables stochastic shadow maps and reflection maps.

Our work has been followed by related research on stochastic rasterization [9, 11, 19, 39, 46].

Paper V was a joint project with Tomas Akenine-Möller and Jon Hasselgren, and my main contribution was in the algorithmic design and implementation of time-dependent texture maps and in the evaluation of the proposed algorithms.

Stochastic rasterization can be seen as a generalization of rasterization from two to five dimensions, where time and two coordinates for the camera lens add three new dimensions to the visibility query in screen space. These additional dimensions make culling tests more complex. For example, a backface culling test for a moving triangle must guarantee that the triangle is backfacing during the entire exposure time. An overly conservative solution is to disable the backface test for moving primitives. In many scenes, about 50% of the primitives can be backface culled, which implies that without backface culling, the rasterizer has to perform about two times more inside tests. Therefore, in many cases, disabling the backface culling test is very likely to result in a performance degradation. Common practice for motion blur is to backface test the moving triangle only at the start and end of the motion [19, 39]. In Paper IX, we first show that this is not always cor-

rect, and derive *conservative* backface culling tests for triangles undergoing motion blur and depth of field for the case of linear and polynomial vertex motion. We also introduce Bézier edge equations for moving triangles, which are numerically more robust reformulations of the *time-continuous* edge equations from Paper V.

I was involved in all aspects of the backface culling for motion blur and depth of field research project, which was a joint project with Tomas Akenine-Möller. I was the primary author of the paper.

7 Conclusions and Future Work

Our research concerns algorithms which exploit domain-specific characteristics for increased quality and performance. The graphics pipeline has served as a foundation, but instead of designing algorithms that run with maximum performance on current GPU generations, we propose modifications to the pipeline itself, to incorporate further optimizations and alternative rendering techniques. A recurring theme in our research is to trade an increase in arithmetic operations for reduced memory bandwidth usage, which we strongly believe is an important tradeoff and will continue to be so in the near future.

The geometric complexity will most likely increase as more developers start using the tessellation stages for real-time graphics workloads. The performance characteristics of tessellated workloads are still mostly unknown for real-time graphics, and the domain shader stage may become a performance bottleneck. Tessellation culling is a powerful tool for optimizing these workloads, and we hope that our research may inspire new optimized culling algorithms, tailored for specific tessellation use cases.

It is exciting to see that HDR texture compression has recently been added to graphics APIs [14, 49]. The latest generation of GPUs have hardware support for decompression of HDR textures, five years after our first paper on the topic. Today, stochastic rasterization is applied mainly in offline rendering, but there is substantial activity in the research community around real-time stochastic rasterization. This makes us hopeful for added support for these effects in future generations of graphics APIs and graphics processors.

A natural extension to our work is programmable tessellation culling for stochastic rasterization, taking the time dimension, camera lens parameters, and varying shader programs parameters into account. Paper IX takes a first step in providing backface culling for a single moving and defocused triangle, but generalized pretessellation culling for moving *patches* may be very useful if the incurred cost of computing the bounds is low enough. The Taylor model bounding technique presented in this thesis is applicable, at least from a theoretical point of view. It can be generalized to handle additional varying input variables, including time and lens parameters, but the curse of dimensionality may be a severe problem in practice. When introducing additional variables, the computations involved in evaluating and bounding the Taylor model quickly becomes intractable, even for a modest

number of dimensions. Optimizing compiler techniques for shader bounding [12] applied to Taylor models is a related direction for future work.

The OpenEXR image format [8] is a commonly used image format in offline rendering systems, as it can represent textures with high bit rates, including HDR images, normal maps, and displacement maps. OpenEXR recently announced support for *multi-view images*, and most likely, someone will present a use-case where these types of images can be used as textures in the graphics pipeline. There is generally a significant amount of coherence between the different views of a multi-view image, making it amenable for a compressed representation. This is another interesting area for future research.

In conclusion, I hope that the algorithms presented in this thesis help to bring more visually pleasing effects to a wider audience.

Bibliography

[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. A. K. Peters Ltd., 3^{rd} edition, 2008.

[2] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000.

[3] ATI. Radeon X800: 3Dc White Paper. Technical report, 2005.

[4] A.C. Beers, M. Agrawala, and Navin Chadda. Rendering from Compressed Textures. In *Proceedings of ACM SIGGRAPH 96*, pages 373–378, 1996.

[5] Martin Berz and Georg Hoffstätter. Computation and Application of Taylor Polynomials with Interval Remainder Bounds. *Reliable Computing*, 4(1):83–97, 1998.

[6] Jim Blinn. Simulation of Wrinkled Surfaces. In *Proceedings of SIGGRAPH*, pages 286–292, 1978.

[7] David Blythe. The Direct3D 10 System. *ACM Transactions on Graphics*, 25(3):724–734, 2006.

[8] R. Bogart, F. Kainz, and D. Hess. OpenEXR Image File Format. In *ACM SIGGRAPH Sketches & Applications*, 2003.

[9] Solomon Boulos, Edward Luong, Kayvon Fatahalian, Henry Moreton, and Pat Hanrahan. Space-Time Hierarchical Occlusion Culling for Micropolygon Rendering with Motion Blur. In *High Performance Graphics*, pages 11–18, 2010.

[10] Chas Boyd. Direct3D Compute Shader. ACM SIGGRAPH Course, Beyond Programmable Shading, 2008.

[11] John S. Brunhaver, Kayvon Fatahalian, and Pat Hanrahan. Hardware Implementation of Micropolygon Rasterization with Motion and Defocus Blur. In *High Performance Graphics*, pages 1–9, 2010.

[12] Petrik Clarberg, Robert Toth, Jon Hasselgren, and Tomas Akenine-Möller. An Optimizing Compiler for Automatic Shader Bounding. *Computer Graphics Forum (Proceedings of EGSR 2010)*, 29(4):1259–1268, 2010.

[13] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21(4):95–102, 1987.

[14] Microsoft Corporation. Windows DirectX Graphics Documentation. Available at `http://msdn.microsoft.com/en-us/library/ee663301(v=VS.85).aspx`, 2010.

[15] William Dally. Power Efficient Supercomputing. Presented at the Accelerator-based Computing and Manycore Workshop, 2009.

[16] Paul E. Debevec and Jitendra Malik. Recovering High Dynamic Range Radiance Maps from Photographs. In *Proceedings of ACM SIGGRAPH*, pages 369–378, 1997.

[17] Eugene d'Eon and David Luebke. Advanced Techniques for Realistic Real-Time Skin Rendering. In *GPU Gems 3*, pages 293–348, 2007.

[18] Kayvon Fatahalian. From Shader Code to a Teraflop: How a Shader Core Works. ACM SIGGRAPH Course, Beyond Programmable Shading, 2010.

[19] Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. Data-parallel Rasterization of Micropolygons with Defocus and Motion Blur. In *High Performance Graphics*, pages 59–68, 2009.

[20] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-planes 5: A Heterogeneous Multiprocessor Graphics System using Processor-Enhanced Memories. *Proceedings of ACM SIGGRAPH*, 23(3):79–88, 1989.

[21] Michael Garland and David B. Kirk. Understanding Throughput-Oriented Architectures. *Communications of the ACM*, 53(11):58–66, 2010.

[22] Kevin Gee. Introduction to the Direct3D 11 Graphics Pipeline. nVision conference course, 2008.

[23] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-buffer Visibility. In *Proceedings of ACM SIGGRAPH*, pages 231–238, 1993.

[24] Larry Gritz. Production Perspectives on High Performance Graphics. Keynote - High Performance Graphics, 2009.

[25] Ziyad S. Hakura and Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *24th International Symposium of Computer Architecture*, pages 108–120, 1997.

[26] Jon Hasselgren and Tomas Akenine-Möller. PCU: The Programmable Culling Unit. *ACM Transactions on Graphics*, 26(3):92.1–92.10, 2007.

[27] Homan Igehy, Matthew Eldridge, and Pat Hanrahan. Parallel Texture Caching. In *Graphics Hardware*, pages 95–106, 1999.

[28] Konstantine Iourcha, Krishna Nayak, and Zhou Hong. System and Method for Fixed-Rate Block-Based Image Compression with Inferred Pixel Values. US Patent 5,956,431, 1999.

[29] Henrik Wann Jensen. *Realistic Image Synthesis using Photon Mapping*. AK Peters, 2001.

[30] James T. Kajiya. The Rendering Equation. *Proceedings of ACM SIGGRAPH*, 20:143–150, August 1986.

[31] Günter Knittel, Andreas G. Schilling, Anders Kugler, and Wolfgang Straßer. Hardware for Superior Texture Performance. *Computers & Graphics*, 20(4):475–481, 1996.

[32] Jeffrey M. Lane, Loren C. Carpenter, Turner Whitted, and James F. Blinn. Scan line methods for displaying parametrically defined surfaces. *Communications of the ACM*, 23:23–34, 1980.

[33] Charles Loop and Schott Schaefer. Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. *ACM Transactions on Graphics*, 27(1):1–11, 2008.

[34] Charles Loop, Scott Schaefer, Tianyun Ni, and Ignacio Castaño. Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation. *ACM Transactions on Graphics*, 28(5):1–9, 2009.

[35] Rafał Mantiuk, Scott Daly, Karol Myszkowski, and Hans-Peter Seidel. Predicting Visible Differences in High Dynamic Range Images – Model and its Calibration. In *Human Vision and Electronic Imaging X*, pages 204–214, 2005.

[36] Chris Marrin. WebGL Specification - Working Draft 03 November 2010. `http://www.khronos.org/webgl/`, 2010.

[37] Michael D. McCool, Chris Wales, and Kevin Moule. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. In *Graphics Hardware*, pages 65–72, 2002.

[38] Joel McCormack and Robert McNamara. Tiled Polygon Traversal Using Half-plane Edge Functions. In *Graphics Hardware*, pages 15–21, 2000.

[39] Morgan McGuire, Eric Enderton, Peter Shirley, and David Luebke. Real-Time Stochastic Rasterization on Conventional GPU Architectures. In *High Performance Graphics*, pages 173–182, 2010.

[40] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, page 4, 1965.

[41] Marc Olano and Trey Greer. Triangle Scan Conversion Using 2D Homogeneous Coordinates. In *Graphics Hardware*, pages 89–96, 1997.

[42] John Owens. Streaming Architectures and Technology Trends. In *GPU Gems 2*, pages 457–470, 2005.

[43] Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes. In *ACM Transactions on Graphics*, pages 1–10, 2010.

[44] Matt Pharr and Greg Humphreys. *Physically Based Rendering*. Morgan Kaufmann, 2^{nd} edition, 2010.

[45] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. In *Proceedings of ACM SIGGRAPH*, pages 17–20, 1988.

[46] Jonathan Ragan-Kelley, Jaakko Lethinen, Jiawen Chen, Michael Doggett, and Fredo Durand. Decoupled Sampling For Real-Time Graphics Pipelines. *To appear in ACM Transactions on Graphics*.

[47] Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann, 1^{st} edition, 2005.

[48] Kimmo Roimela, Tomi Aarnio, and Joonas Itäranta. High Dynamic Range Texture Compression. *ACM Transactions on Graphics*, 25(3):707–712, 2006.

[49] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A specification (Version 4.0). `http://www.opengl.org/documentation/specs/`, 2010.

[50] Jay Torborg and Jim Kajiya. Talisman: Commodity Real-time 3D Graphics for the PC. In *Proceedings of ACM SIGGRAPH*, pages 353–364, 1996.

Paper I

High Dynamic Range Texture Compression for Graphics Hardware

Jacob Munkberg Petrik Clarberg Jon Hasselgren Tomas Akenine-Möller

Lund University

ABSTRACT

In this paper, we break new ground by presenting algorithms for fixed-rate compression of high dynamic range textures at low bit rates. First, the S3TC low dynamic range texture compression scheme is extended in order to enable compression of HDR data. Second, we introduce a novel robust algorithm that offers superior image quality. Our algorithm can be efficiently implemented in hardware, and supports textures with a dynamic range of over 10^9:1. At a fixed rate of 8 bits per pixel, we obtain results virtually indistinguishable from uncompressed HDR textures at 48 bits per pixel. Our research can have a big impact on graphics hardware and real-time rendering, since HDR texturing suddenly becomes affordable.

ACM Transactions on Graphics 25(3):698–706, 2006.

1 Introduction

The use of high dynamic range (HDR) images in rendering [6, 7, 18, 27] has changed computer graphics forever. Prior to this, only low dynamic range (LDR) images were used, usually storing 8 bits per color component, i.e., 24 bits per pixel (bpp) for RGB. Such images can only represent a limited amount of the information present in real scenes, where luminance values spanning many orders of magnitude are common. To accurately represent the full dynamic range of an HDR image, each color component can be stored as a 16-bit floating-point number. In this case, an uncompressed HDR RGB image needs 48 bpp.

In 2001, HDR images were first used in real-time rendering [4], and over the past years, we have observed a rapidly increasing use of HDR images in this context. Game developers have embraced this relatively new technique, and several recent games use HDR images as textures. Examples include Unreal Engine 3, Far Cry, Project Gotham Racing 3, and Half-Life 2: Lost Coast.

The disadvantage of using HDR textures in real-time graphics is that the texture bandwidth usage increases dramatically, which can easily limit performance. With anisotropic filtering or complex pixel shaders, it can become even worse. A common approach to reduce the problem is *texture compression*, introduced in 1996 [1, 11, 23]. By storing textures in compressed form in external memory, and sending compressed data over the bus, the bandwidth is significantly reduced. The data is decompressed in real time using special-purpose hardware when it is accessed by the pixel shader. Several formats use as little as 4 bpp. Compared to 24 bpp RGB, such techniques can potentially reduce the texture bandwidth to only 16.7% of the original.

Texels in textures can be accessed in any order during rasterization. A fixed-rate texture compression (TC) system is desirable, as it allows random addressing without complex lookup mechanisms. Hence, JPEG and similar algorithms do not immediately qualify as reasonable alternatives for TC, since they use an adaptive bit rate over the image. The fixed bit rate also implies that all realistic TC algorithms are lossy. Other characteristics of a TC system are that the decompression should preferably be fast and relatively inexpensive to implement in hardware. However, we expect that increasingly complex decompression schemes can be accepted by the graphics hardware industry, since the available bandwidth grows at a much slower pace than the computing power [15]. A difference between LDR and HDR TC is that for HDR images, we do not know in advance what range of luminance values will be displayed. Hence, the image quality must remain high over a much larger range of luminance.

We present novel HDR TC schemes, which are inexpensive to implement in hardware. Our algorithms compresses tiles of 4×4 pixels to only 8 bpp. The compressed images are of very high quality over the entire range of input values, and are essentially indistinguishable from uncompressed 48 bpp data. An example of a compressed image is shown in Figure 1.

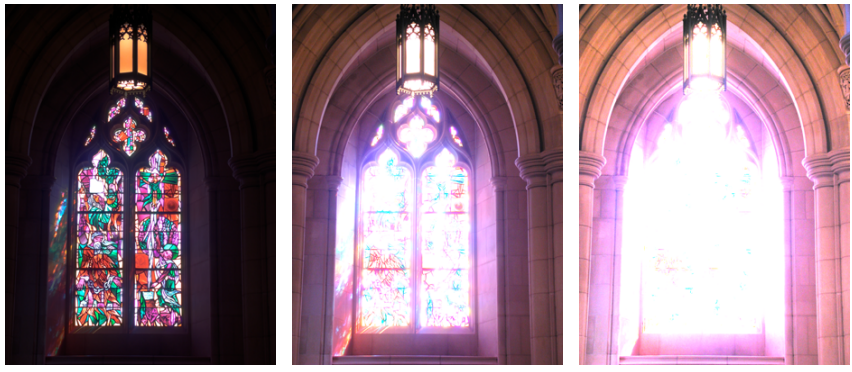


Figure 1: Example of a high dynamic range image, here shown at three different exposures, compressed with our algorithm to a fixed rate of 8 bits per pixel. Our algorithm gives excellent image quality over a large dynamic range, and is fast to decompress in hardware.

2 Related Work

Here, we first present research in LDR texture compression (TC) for graphics hardware that is relevant to our work. For a more complete overview, consult Fenney's paper [9]. Second, some attention is given to existing HDR compression systems.

LDR Texture Compression Vector quantization (VQ) techniques have been used by Beers et al. [1] for TC. They presented compression ratios as low as one or two bpp. However, VQ requires an access in a look-up table, which is not desirable in a graphics hardware pipeline. The S3TC texture compression scheme [10] has become a de facto standard for real-time rendering. Since we build upon this scheme, it is described in more detail in Section 4.

Fenney [9] presents a system where two low-resolution images are stored for each tile. During decompression, these images are bilinearly magnified and the color of a pixel is obtained as a linear blend between the magnified images. Another TC scheme assumes that the whole mipmap pyramid is to be compressed [16]. Box filtering is used, and the luminance of a 4×4 tile is decomposed using Haar wavelets. The chrominance is subsampled, and then compressed. The compression ratio is 4.6 bpp.

In a TC system called iPACKMAN [22], 4×4 tiles of pixels are used, and each tile encodes two base colors in RGB444 and a choice of modifier values. The color of a pixel is determined from one of the base colors by adding a modifier value.

HDR Image and Video Compression To store an HDR image in the RGBE format, Ward [24] uses 32 bits per pixel, where 24 bits are used for RGB, and the remaining 8 bits for an exponent, E, shared by all three color components. A

straightforward extension would be to compress the RGB channels using S3TC to 4 bits per pixel, and store the exponent uncompressed as a separate 8 bpp texture, resulting in a 12 bits per pixel format supported by current graphics hardware. However, RGBE has a dynamic range of 76 orders of magnitude and is not a compact representation of HDR data known to reside in a limited range. Furthermore, as both the RGB and the exponent channel contain luminance information, chrominance and luminance transitions are not separated, and artifacts similar to the ones in Figure 13 are likely to occur. Ward also developed the LogLuv format [28], where the RGB input is separated into luminance and chrominance information. The logarithm of the luminance is stored in 16 bits, while the chrominance is stored in another 16 bits, resulting in 32 bits per pixel. A variant using 24 bpp was also presented.

Ward and Simmons [26] use the possibility of storing an extra 64 kB in the JPEG image format. The file contains a tone mapped image, which can be decompressed using standard JPEG decompressors. In the 64 kB of data, a ratio image of the luminance in the original and the tone mapped image is stored. A loader incapable of handling the format will display a tone mapped image, while capable loaders will obtain the full dynamic range. Xu et al. [29] use the wavelet transform and adaptive arithmetic coding of JPEG 2000 to compress HDR images. They first compute the logarithm of the RGB values, and then use existing JPEG 2000 algorithms to encode the image. Impressive compression ratios and a high quality is obtained. Mantiuk et al. [14] present an algorithm for compression of HDR video. They quantize the luminance using a non-linear function in order to distribute the error according to the luminance response curve of the human visual system. Then, they use an MPEG4-based coder, which is augmented to suppress errors around sharp edges. These three algorithms use adaptive bit rates, and thus cannot provide random access easily.

There is a wide range of tone mapping operators (cf. [18]), which perform a type of compression. However, the dynamic range is irretrievably lost in the HDR to LDR conversion, and these algorithms are therefore not directly applicable for TC. Li et al. [12] developed a technique called companding, where a tone mapped image can be reconstructed back into an HDR image with high quality. This technique is not suitable for TC since it applies a global transform to the entire image, which makes random access extremely slow, if at all feasible. Still, inspiration can be obtained from these sources.

In the spirit of Torborg and Kajiya [23], we implemented a fixed-rate HDR DCT encoder, but on hardware-friendly 4×4 tiles at 8 bits per pixel. The resulting images showed severe ringing artifacts near sharp luminance edges and moderate error values. The decompressor is also substantially more complex than the algorithms we present below.

3 Color Spaces and Error Measures

In this section, we discuss different color spaces, and develop a small variation of an existing color space, which is advantageous in terms of hardware decompression and image quality. Furthermore, we discuss error metrics in Section 3.2, where we also suggest a new simple error metric.

3.1 Color Spaces

The main difficulty in compressing HDR textures is that the dynamic range of the color components can be very large. In natural images, a dynamic range of 100,000:1 is not uncommon, i.e., a factor 10^5 difference between the brightest and the darkest pixels. A 24-bit RGB image, on the other hand, has a maximum range of 255:1. Our goal is to support about the same dynamic range as the OpenEXR format [2], which is based on the hardware-supported *half* data type, i.e., 16-bit floating-point numbers. The range of representable numbers with full precision is roughly $6.1 \cdot 10^{-5}$ to $6.5 \cdot 10^4$, giving a dynamic range of 10^9:1. We aim to support this range directly in our format, as a texture may undergo complex image operations where lower precision is not sufficient. Furthermore, the dynamic range of the test images used in this paper is between $10^{2.6}$ and $10^{7.3}$. An alternative is to use a tighter range and a per-texture scaling factor. This is a trivial extension, which would increase the quality for images with lower dynamic ranges. However, this requires global per-texture data, which we have opted to avoid. We leave this for future work.

To get consistently good quality over the large range, we need a color space that provides a more compact representation of HDR data than the standard RGB space. Taking the logarithm of the RGB values gives a nearly constant relative error over the entire range of exposures [25]. Assume we want to encode a range of 10^9:1 in 1% steps. In this $\log[RGB]$ space, we would need $k = 2083$ steps, given by $1.01^k = 10^9$, or roughly 11 bits precision per color channel. Because of the high correlation between the RGB color components [19], we need to store all three with high accuracy. As we will see in Section 4, we found it difficult to reach the desired image quality and robustness when using the $\log[RGB]$ space.

In image and video compression, it is common to decorrelate the color channels by transforming the RGB values into a luminance/chrominance-based color space [17]. The motivation is that the luminance is perceptually more important than the chrominance, or *chroma* for short, and more effort can be spent on encoding the luminance accurately. Similar techniques have been proposed for HDR image compression. For example, the LogLuv encoding [28] stores a log representation of the luminance and CIE (u', v') chrominance. Xu et al. [29] apply the same transform as in JPEG, which is designed for LDR data, but on the logarithm of the RGB components. The OpenEXR format supports a simple form of compression

based on a luminance/chroma space with the luminance computed as:

$$Y = w_r R + w_g G + w_b B, \tag{1}$$

and two chroma channels, U and V, defined as:

$$U = \frac{R-Y}{Y}, \qquad V = \frac{B-Y}{Y}. \tag{2}$$

Lossy compression is obtained by subsampling the chroma components by a factor two horizontally and vertically.

Inspired by previous work, we define a simple color space denoted $\log Y\bar{u}\bar{v}$, based on log-luminance and two chroma components. Given the luminance Y computed using Equation 1, the transform from RGB is given by:

$$(\bar{Y}, \bar{u}, \bar{v}) = \left(\log_2 Y, w_b \frac{B}{Y}, w_r \frac{R}{Y} \right). \tag{3}$$

We use the Rec. 601 [17] weights (0.299, 0.587, 0.114) for w_r, w_g and w_b. With non-zero, positive input RGB values in the range $[2^{-16}, 2^{16}]$, the log-luminance $\bar{Y}$ is in the range $[-16, 16]$, and the chroma components are in the range $[0, 1]$ with $\bar{u} + \bar{v} \leq 1$.

In our color space, the HDR luminance information is concentrated to the $\bar{Y}$ component, which needs to be accurately represented, while the $(\bar{u}, \bar{v})$ components only contain chrominance information normalized for luminance. These can be represented with significantly less accuracy.

3.2 Error Measures

In order to evaluate the performance of various compression algorithms, we need an image quality metric that provides a meaningful indication of image fidelity. For LDR images, a vast amount of research in such metrics has been conducted [3]. Perceptually-based metrics have been developed, which attempt to predict the observed image quality by modeling the response of the human visual system (HVS). The prime example is the *visible differences predictor* (VDP) introduced by Daly [5].

Error measures for HDR images are not as thoroughly researched, and there is no well-established metric. The image must be tone-mapped before VDP or any other standard image quality metric, designed for LDR data, can be applied. The choice of tone mapping operator will bias the result, which is unfortunate. In our application, another difficulty is that we do not know how the HDR textures will be used or what the display conditions will be like. For example, a texture in a 3D engine can undergo a number of complex operations, such as lighting, blending and multi-texturing, which change its appearance.

Xu et al. [29] compute the *root-mean-square error* (RMSE) of the compressed image in the $\log[RGB]$ color space. Their motivation is that the logarithm is a

conservative approximation of the HVS luminance response curve. However, we argue that this error measure can be misleading in terms of visual quality. The reason is that an error in a small component tends to over-amplify the error measure even if the small component's contribution to the final pixel color is small. For example, consider a mostly red pixel, $\mathbf{r} = (1000, 1, 1)$, which is compressed to $\mathbf{r}^* = (1000, 1, 8)$. The $\log[RGB]$ RMSE is then $\log_2 8 - \log_2 1 = 3$, but the log-luminance RMSE, to which the HVS is most sensitive, is only 0.004. Still, we include the $\log[RGB]$ RMSE error because it reflects the relative, per-component error of the compressed image. It is therefore well suited to describe the expected error of the aforementioned image operations: blending, lighting etc.

To account for all normal viewing conditions, we propose a simple error metric, which we call *multi-exposure peak-signal-to-noise ratio*, or *mPSNR* for short. The HDR image is tone mapped to a number of different exposures, uniformly distributed over the dynamic range of the image. See Figure 2 for an example. For each exposure, we compute the *mean square error* (MSE) on the resulting LDR image, and then compute the peak-signal-to-noise ratio (PSNR) using the mean of all MSEs. As a tone mapping operator, we use a simple gamma-adjustment after exposure compensation. The tone mapped LDR image, $T(I)$, of the HDR image, I, is given by:

$$T(I) = \left[255\,(2^c I)^{1/\gamma}\right]_0^{255}, \tag{4}$$

where c is the exposure compensation in f-stops, γ is the display gamma, and $[\cdot]_0^{255}$ indicates clamping to the integer interval $[0, 255]$. The mean square error over all exposures and over all pixels is computed as:

$$\text{MSE} = \frac{1}{n \times w \times h} \sum_c \sum_{x,y} \left(\Delta R_{xy}^2 + \Delta G_{xy}^2 + \Delta B_{xy}^2\right), \tag{5}$$

where n is the number of exposures, c, and $w \times h$ is the image resolution. The error in the red component (similar for green and blue) at pixel (x, y) is $\Delta R_{xy} = T_R(I) - T_R(C)$, where I is the original image, and C is the compressed image. Finally, mPSNR is computed as:

$$\text{mPSNR} = 10 \log_{10} \left(\frac{3 \times 255^2}{\text{MSE}}\right). \tag{6}$$

The obtained mPSNR over all exposures gives us a prediction of the error in the compressed HDR image. The PSNR measure has traditionally been popular for evaluating the performance of TC schemes, and although no other HDR texture compression techniques exist, the use of mPSNR makes our results more easily interpreted.

Recently, Mantiuk et al. [13] have presented a number of modifications to the visual differences predictor, making it possible to predict the perceived differences over the entire dynamic range in real scenes. This novel HDR VDP takes into account a number of complex effects such as the non-linear response and local

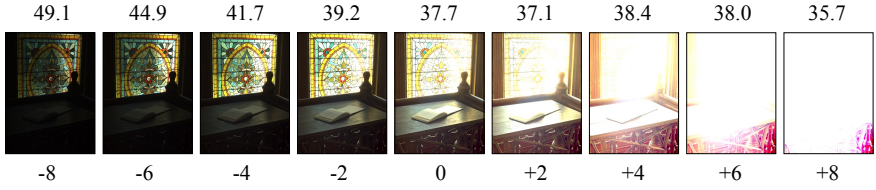


Figure 2: In our multi-exposure PSNR *error measure, the image is tone mapped to a number of different exposures to account for all normal viewing conditions, and the PSNR is computed from the average MSE. In this case, the mPSNR is 39 dB. The top row shows the standard PSNRs, and the bottom row shows the exposure compensation, c.*

adaptation of the HVS. However, their current implementation only works on the luminance, and does not take the chroma error into account.

As there is no established standard for evaluating HDR image quality, we have chosen to use a variety of error metrics. We present results for our algorithm using the mPSNR, the $\log[RGB]$ root-mean-square error, and the HDR VDP.

4 HDR S3 Texture Compression

The S3 texture compression (S3TC) method [10] is probably the most popular scheme for compressing LDR textures. It is used in DirectX and there are extensions for it in OpenGL as well. S3TC uses tiles of 4×4 pixels that are compressed to 64 bits, giving a compression rate of 4 bpp. Two base colors are stored in 16 bits (RGB565) each, and every pixel stores a two-bit index into a local color map consisting of the two base colors and two additional colors in between the base colors. This means that all colors lie on a straight line in RGB space.

A natural suggestion for an HDR TC scheme is to adapt the existing S3TC algorithm to handle HDR data. Due to the increased amount of information, we double the rate to 8 bpp. We also apply the following changes. First, we transform the linear RGB data into a more compact color space. Second, we raise the quantization resolution and the number of per-pixel index bits. In graphics hardware, the memory is accessed in bursts of 2^n bits, e.g., 256 bits. To simplify addressing, it is desirable to fetch 2^m pixels per burst, which gives 2^{n-m} bits per pixel (e.g., 4,8,16,...). Hence, keeping a tile size of 4×4 pixels is a reasonable choice, as one tile fits nicely into 128 bits on an 8 bpp budget. In addition, a small tile size limits the variance across the tile and keeps the complexity of the decompressor low.

The input data consists of three floating-point values per pixel. Performing the compression directly in linear RGB space, or in linear YUV space, produces extremely poor results. This is due to the large dynamic range. Better results are obtained in the $\log[RGB]$ and the $\log Y\bar{u}\bar{v}$ color spaces (Section 3.1). Our tests show that 4-bit per-pixel indices are needed to accurately capture the luminance

variations. We call the resulting algorithms *S3TC RGB* (using $\log[RGB]$), and *S3TC YUV* (using $\log Y\bar{u}\bar{v}$). The following bit allocations performed best in our tests:

Color space	Base colors	Per-pixel indices
$\log[RGB]$	$2\times(11+11+10)=64$	$16\times 4=64$
$\log Y\bar{u}\bar{v}$	$2\times(12+10+10)=64$	$16\times 4=64$

Even though these S3TC-based approaches produce usable results in some cases, they lack the robustness needed for a general HDR TC format. Some of the shortcomings of S3TC RGB and S3TC YUV are clearly illustrated in Figure 13. As can be seen in the enlarged images, both algorithms produce serious block artifacts, and blurring of some edges. This tends to happen where there is a chroma and a luminance transition in the same tile, and there is little or no correlation between these. The reason is that all colors must be located on a straight line in the respective 3D color space for the algorithms to perform well. In Figure 13, we also show the results of our new HDR texture compression scheme. As can be seen, the image quality is much higher. More importantly, our algorithm is more robust, and rarely generates tiles of poor quality.

5 New HDR Texture Compression Scheme

In the previous section, we have seen that building a per-tile color map from a straight line in some 3D color space does not produce acceptable results for S3TC-based algorithms. To deal with the artifacts, we decouple the luminance from the chrominance and encode them separately in the $\log Y\bar{u}\bar{v}$ space defined in Equation 3. By doing this, difficult tiles can be handled much better. In the following, we describe how the luminance and chrominance can be accurately represented on an 8 bpp budget, i.e., 128 bits per tile.

5.1 Luminance Encoding

In the $\log Y\bar{u}\bar{v}$ color space, the log-luminance values $\bar{Y}$ are in the range $[-16,16]$. First, we find the minimum and maximum values, $\bar{Y}_{\min}$ and $\bar{Y}_{\max}$, in a tile. Inspired by S3TC, we then quantize these linearly and store per-pixel indices indicating which luminance step between $\bar{Y}_{\min}$ and $\bar{Y}_{\max}$ that is to be used for each pixel.

As we have seen, we need approximately 16 steps, i.e., 4-bit per-pixel indices, for an accurate representation of HDR luminance data. If we use 12-bit quantization of $\bar{Y}_{\min}$ and $\bar{Y}_{\max}$ as in S3TC YUV, a total of $2\times 12+16\times 4=88$ bits are consumed, and only 40 bits are left for the chroma encoding. This is not enough. By searching in a range around the quantized base values, it is very often possible to find a combination that gives a significantly reduced error. Thus, we manage to encode the base luminances with only 8 bits each without any noticeable artifacts, even on slow gradients.

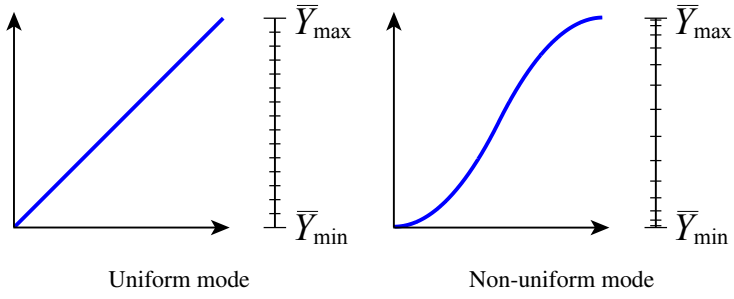


Figure 3: The two luminance quantization modes. The non-uniform mode is used for better handling tiles with sharp luminance transitions, such as edges.

Another approach would be to use spatial subsampling of the luminance. Recent work on HDR displays by Seetzen et al. [20, 21] suggests that the human eye's spatial HDR resolution is lower than its LDR resolution. However, the techniques developed for direct display of HDR images are not directly applicable to our problem as they require high-precision per-pixel LDR data to modulate the subsampled HDR luminance. We have tried various hierarchical schemes, but the low bit budget made it difficult to obtain the required per-pixel precision. Second, our compression scheme is designed for textures, hence we cannot make any assumptions on how the images will be displayed on screen. The quality should be reasonable even for close-up zooms. Therefore, we opted for the straightforward solution of storing per-pixel HDR luminance.

The most difficult tiles contain sharp edges, e.g., the edge around the sun in an outdoor photograph. Such tiles can have a very large dynamic range, but at the same time, both the darker and the brighter areas must be represented accurately. For this, a uniform quantization between the min/max luminances is not ideal. To better handle such tiles, we add a mode using non-uniform steps between the $\bar{Y}_{min}$ and $\bar{Y}_{max}$ values. Smaller quantization steps are used near the base luminances, and larger steps in the middle. Thus, two different luminance ranges that are far apart can be accurately represented in the same tile. In our test images (Figure 10), the non-uniform mode is used for 11% of the tiles, and for these tiles, the log-luminance RMSE is decreased by 12.0% on average. The two quantization modes are illustrated in Figure 3.

To choose between the two modes, we use the mutual ordering[1] of $\bar{Y}_{min}$ and $\bar{Y}_{max}$. In decoding, if $\bar{Y}_{min} \leq \bar{Y}_{max}$, then the uniform mode is used. Otherwise, $\bar{Y}_{min}$ and $\bar{Y}_{max}$ are reversed, and we use the non-uniform mode. Hence, no additional mode bit is necessary, and the luminance encoding uses a total of $2 \times 8 + 16 \times 4 = 80$ bits, leaving 48 bits for the chrominance. The bit allocation is illustrated in Figure 4.

[1] Similar ordering techniques are used in the S3TC LDR texture compression format.

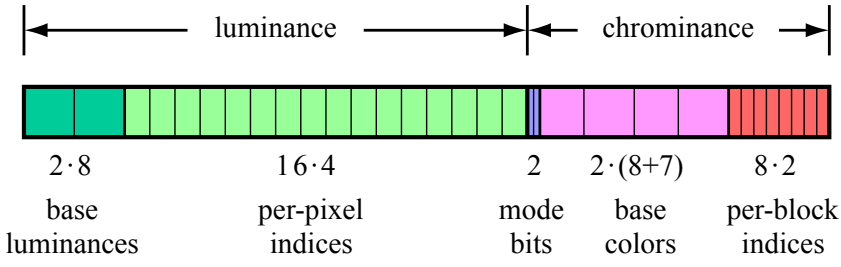


Figure 4: The bit allocation we use for encoding the luminance and chrominance of a 4×4 *tile in 128 bits (8 bpp).*

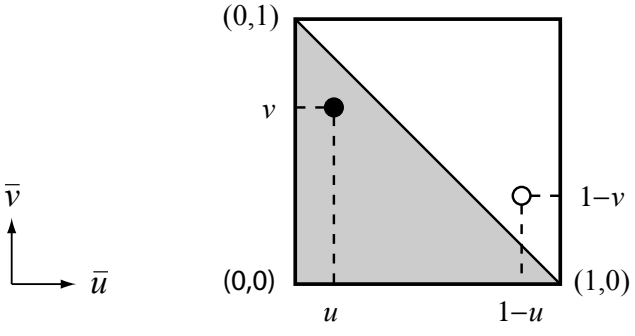


Figure 5: Illustration of the flip trick. *By mirroring the coordinates for a base color, we exploit our triangular chrominance space in order to obtain another bit.*

5.2 Chrominance Line

Our first approach to chrominance compression on a 48-bit budget, is to use a line in the $(\bar{u}, \bar{v})$ chroma plane. Similar to the luminance encoding, each tile stores a number of indices to points uniformly distributed along the line.

In order to fit the chroma line in only 48 bits, we sub-sample the chrominance by a factor two, either horizontally or vertically, similar to what is used in DV encoding [17]. The sub-sampling mode that minimizes the error is chosen. To simplify the following description, we define a *block* as being either 1×2 (horizontal) or 2×1 (vertical) sub-sampled pixels. The start and end points of the chroma line, each with 2×8 bits resolution, and eight 2-bit per-block indices, gives a total cost of $4 \times 8 + 8 \times 2 = 48$ bits per tile.

In our color space, the normalized chroma points $(\bar{u}_i, \bar{v}_i)$, $i \in [0,7]$, are always located in the lower triangle of the chroma plane. If we restrict the encoding of the end points of the line to this area, we can get two extra bits by a *flip trick* described in Figure 5. If a chrominance value $\mathbf{c} = (\bar{u}_i, \bar{v}_i)$ is in the upper (invalid) triangle,

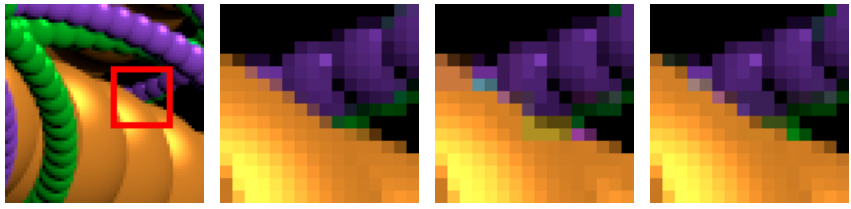


Figure 6: A region where a line in chroma space is not sufficient for capturing the complex color. With shape transforms, we get a much closer resemblance to the true color, although minor imperfections exist due to the sub-sampling.

this indicates that the extra bit is set to one, and the true chroma value is given by $\mathbf{c}' = (1 - \bar{u}_i, 1 - \bar{v}_i)$, otherwise the bit is set to zero. We can use one of these extra bits to indicate whether to use horizontal or vertical sub-sampling. The other bit is left unused.

5.3 Chroma Shape Transforms

A line in chroma space can only represent two chrominances and the gradient between them. This approximation fails for tiles with complex chroma variations or sharp color edges. Figure 6 shows an example of such a case. One solution would be to encode $\bar{u}$ and $\bar{v}$ separately, but this does not easily fit in 48 bits.

To better handle difficult tiles, we introduce *shape transforms*; a set of shapes, each with four discrete points, designed to capture chroma information. In the classic game *Tetris*, the optimal placement of a shape in a grid is found by rotating and translating it in 2D. The same idea is applied to the chrominances of a tile. We allow arbitrary rotation, translation, and uniform scaling of our shapes to make them match the eight sub-sampled chrominance values of a tile as closely as possible. The transformation of the shape can be retrieved by storing only two points.

During compression, we select the shape that most closely covers the chroma information of the tile, and store its index along with two base chrominances (start & end) and per-block indices. This allows each block in the tile to select one of the discrete positions of the shape. The shape fitting is illustrated in Figure 7 on one of the difficult tiles from the image in Figure 6. Using one of the transformed shapes, we get much closer to the actual chroma information.

The space of possible shapes is very large. In order to find a selection of shapes that perform well, we have analyzed the chrominance content in a set of images (a total of 500,000 tiles), different from our test images. First, clustering was done to reduce the chroma values of a tile to four chroma points. Second, we normalized the chroma points for scale and rotation, and then iteratively merged the two closest candidates until the desired number of shapes remained. Figure 8

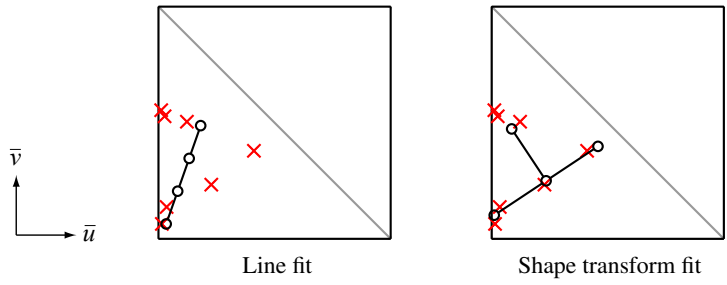
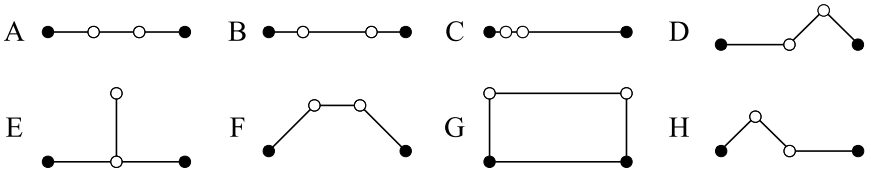


Figure 7: In tiles with difficult chrominance, such as in this example taken from Figure 6, a line in chroma space has difficulties representing the $(\bar{u},\bar{v})$ points accurately (left). Our algorithm based on shape transforms *generates superior chroma representations, as it is often possible to find a shape that closely matches the chrominance points (right).*

shows our selection of shapes after a slight manual adjustment of the positions. See Appendix B for the exact coordinates. Shapes A through C handle simple color gradients, while D–H are optimized for tiles with complex chrominance. Also, by including the uniform line (shape A), we make the chrominance line algorithm (Section 5.2) a subset of the shape transforms approach. Note that the set of shapes is fixed, so no global per-texture data is needed.

Compared to the chrominance line, shape transforms need three bits per tile to indicate which of the eight shapes to use. We exploit the unused bit from the chrominance line, and the other two extra bits are taken from the quantization of the start and end points. We lower the $(\bar{u},\bar{v})$ quantization from $8+8$ bits to $8+7$ bits. Recall from Section 3.1, that in the $\log Y\bar{u}\bar{v}$ to *RGB* transform, the *R* and *B* components are given as $R = \bar{v}Y/w_r$ and $B = \bar{u}Y/w_b$, with $w_r = 0.299$ and $w_b = 0.114$. As w_b is about three times smaller than w_r, it makes the reconstructed color more sensitive for quantization errors in the $\bar{u}$ component, and therefore more



Shape	A	B	C	D	E	F	G	H
Freq(%)	5.8	10.8	6.3	18.3	17.4	16.8	6.6	18.0

Figure 8: The set of shapes we use for the shape transform *algorithm and their frequencies for our test images. The points corresponding to base colors are illustrated with solid black circles.*

bits are spent there.

With these modifications, shape transforms with eight shapes fit in precisely 48 bits. The total bit allocation for luminance and chrominance is illustrated in Figure 4. We evaluated both the chrominance line and the shape transform approach, combined with the luminance encoding of Section 5.1, on our test images. On average, the mPSNR was about 0.5 dB higher using shape transforms, and the resulting images are more visually pleasing, especially in areas with difficult chrominance.

The shape fitting step of our new algorithm is implemented by first clustering the eight sub-sampled input points to four groups. Then we apply Procrustes analysis [8] to find the best orientation of each shape to the clustered data set, and the shape with the lowest error is chosen. This is an approximate, but robust and efficient approach. We achieved somewhat lower errors by using an extensive search for the optimal shape transform, but this is computationally much more expensive.

6 Hardware Decompressor

In this section, we present a decompressor unit for hardware implementation of our algorithm. We first describe how the chrominance, $(\bar{u}, \bar{v})$, is decompressed for a single pixel. In the second part of this section, we describe the color space transformation back to RGB-space. A presentation of log-luminance decompression is omitted, since it is very similar to S3TC LDR decompression. The differences are that the log-luminance is one-dimensional (instead of three-dimensional), and more bits are used for the quantized base values and per-pixel indices. In addition, we also have the non-uniform quantization, but this only amounts to using different constants in the interpolation.

The decompression of chrominance is more complex than for luminance, and in Figure 9, one possible implementation is shown. To use shape transforms, a coordinate frame must be derived from the chroma endpoints, $(\bar{u}_0, \bar{v}_0)$ and $(\bar{u}_1, \bar{v}_1)$, of the shape. In our case, the first axis is defined by $\mathbf{d} = (d_u, d_v) = (\bar{u}_1 - \bar{u}_0, \bar{v}_1 - \bar{v}_0)$. The other axis is $\mathbf{d}^\perp = (-d_v, d_u)$, which is orthogonal to $\mathbf{d}$. The coordinates of a point in a shape are described by two values α and β, which are both fixed-point numbers in the interval $[0,1]$, using only five bits each. The chrominance of a point, with coordinates α and β, is derived as:

$$\begin{pmatrix} \bar{u} \\ \bar{v} \end{pmatrix} = \alpha\mathbf{d} + \beta\mathbf{d}^\perp + \begin{pmatrix} \bar{u}_0 \\ \bar{v}_0 \end{pmatrix} = \begin{pmatrix} \alpha d_u - \beta d_v + \bar{u}_0 \\ \beta d_u + \alpha d_v + \bar{v}_0 \end{pmatrix}. \tag{7}$$

The diagram in Figure 9 implements the equation above. As can be seen, the hardware is relatively simple. The α and β constants units contain only the constants which define the different chrominance shapes. Only five bits per value are used, and hence the four multipliers compute the product of a 5-bit value and an 8 or 9-bit value. Note that $(\bar{u}, \bar{v})$ are represented using fixed-points numbers, so integer arithmetic can be used for the entire chroma decompressor.

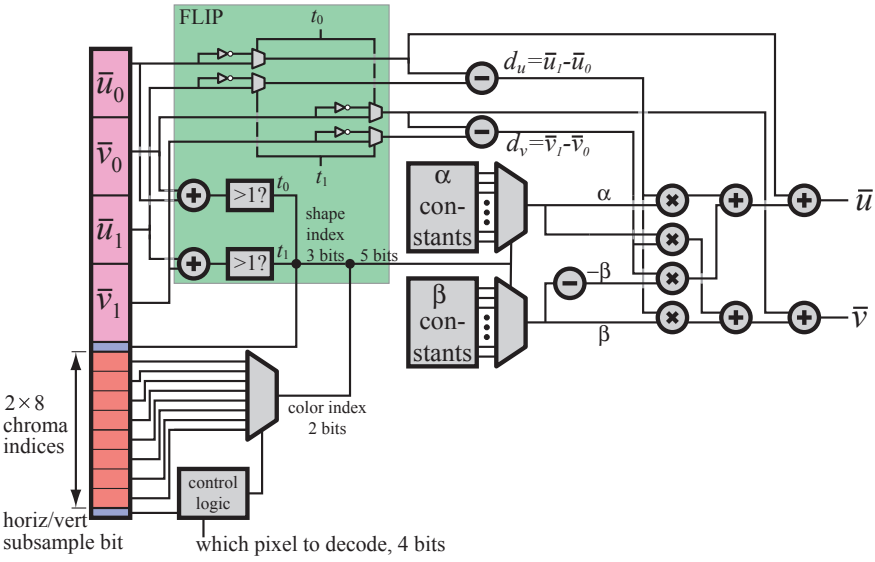


Figure 9: Decompression of chrominance, $(\bar{u},\bar{v})$, for a single pixel. The indata is the 48 bits for chrominance (left part of the figure), and a 4-bit value indicating which pixel in a tile to decode. The outdata is $(\bar{u},\bar{v})$ for one pixel. The green box contains the logic to implement the flip trick*, where inverters have been used to compute the $1-x$ terms.*

At this point, we assume that $\bar{Y}$, $\bar{u}$ and $\bar{v}$ have been computed for a certain pixel in a tile. Next, we describe our transform back to linear RGB space. This is done by first computing the floating-point luminance: $Y = 2^{\bar{Y}}$. After that, the red, green, and blue components can be derived from Equation 3 as:

$$(R,G,B) = \left(\frac{1}{w_r}\bar{v}Y, \frac{1}{w_g}(1-\bar{u}-\bar{v})Y, \frac{1}{w_b}\bar{u}Y\right). \tag{8}$$

Since the weights, $(w_r,w_g,w_b) = (0.299,0.587,0.114)$, are constant, their reciprocals can be precomputed. We propose using the hardware-friendly constants:

$$(1/w'_r, 1/w'_g, 1/w'_b) = \frac{1}{16}(54,27,144). \tag{9}$$

This corresponds to

$$(w'_r,w'_g,w'_b) \approx (0.296,0.593,0.111). \tag{10}$$

Using our alternative weights makes the multiplications much simpler. This comes with a non-noticeable degradation in image quality.

In summary, our color space transform involves one power function, two fixed-point additions, three fixed-point multiplications, and three floating-point times fixed-point multiplications. The majority of color space transforms include at least

a 3×3 matrix/vector multiplication, and in the case of HDR data, we have seen that between 1–3 power functions are also used. Ward's LogLuv involves even more operations. Our transform involves significantly fewer arithmetic operations compared to other color space transforms, and this is a major advantage of our decompressor.

The implementation shown above can be considered quite inexpensive, at least when compared to using other color spaces. Still, when compared to popular LDR TC schemes [10, 22], our decompressor is rather complex. However, we have attempted to make it simpler by designing a hardware-friendly color space, avoided using too many complex arithmetic operations, and simplified constants. In addition, we believe that in the near future, graphics hardware designers will have to look into more complex circuitry in order to reduce bandwidth, and the technological trend shows that this is the way to go [15].

7 Results

To evaluate the algorithms, we use a collection of both synthetic images and real photographs, as shown in Figure 10. Many of these are well-known and widely used in HDR research. In the following, we refer to *our algorithm* as the combination of our luminance encoding (Section 5.1) and shape transforms (Section 5.3). The results using mPSNR, $\log[RGB]$ RMSE, and HDR VDP are presented in Figure 11. After that follows visual comparisons.

In terms of the mPSNR measure, our algorithm performs substantially better over the entire set of images, with an average improvement of about 3 dB over the S3TC-based approaches. The mPSNR measure simulates the most common use of an HDR image in a real-time application, where the image is tone mapped and displayed under various exposures, either as a decal or as an environment map. The range of exposures used in mPSNR is automatically determined by computing the min and max luminance over each image, and mapping this range to exposures that give a nearly black, and a nearly white LDR image respectively. See Appendix A for the exact numbers.

The $\log[RGB]$ RMSE measures the relative error per component over the entire dynamic range of the image. In this metric, the differences between the algorithms are not as obvious. However, our algorithm gives slightly lower error on average.

The HDR VDP chart in Figure 11 shows just noticeable luminance differences. The 75%-value indicates that an artifact will be visible with a probability of 0.75, and the value presented in the chart is the percentage of pixels above this threshold. Our test suite consists of a variety of both luminance-calibrated and relative luminance HDR images. To compare them, we multiply each image with a luminance factor so that all HDR VDP tests are performed for a global adaptation level of approximately 300 cd/m^2. Although we quantize the base luminances to only 8 bits, our algorithm shows near-optimal results, except for image **(a)**. VDP difference images for image **(i)** are shown in Figure 12. Increasing the global adaptation level

increases the detection rates slightly, but the relationship between the algorithms remains.

Our algorithm is the clear winner both in terms of robustness and perceived visual quality, as can be seen in Figure 13. In general, luminance artifacts are more easily detectable, and both S3TC YUV and our algorithm handle these cases better due to the luminance focus of the $\log Y\bar{u}\bar{v}$ color space. However, the limitation of S3TC YUV to a line in 3D makes it unstable in many cases. The only errors we have seen using our algorithm are slight chrominance leaks due to the sub-sampling, and artifacts in some images with high exposures, originating from the quantization of the base chrominances. Such a scenario is illustrated in Figure 14. Overall, our algorithm is much more robust since it generates significantly fewer

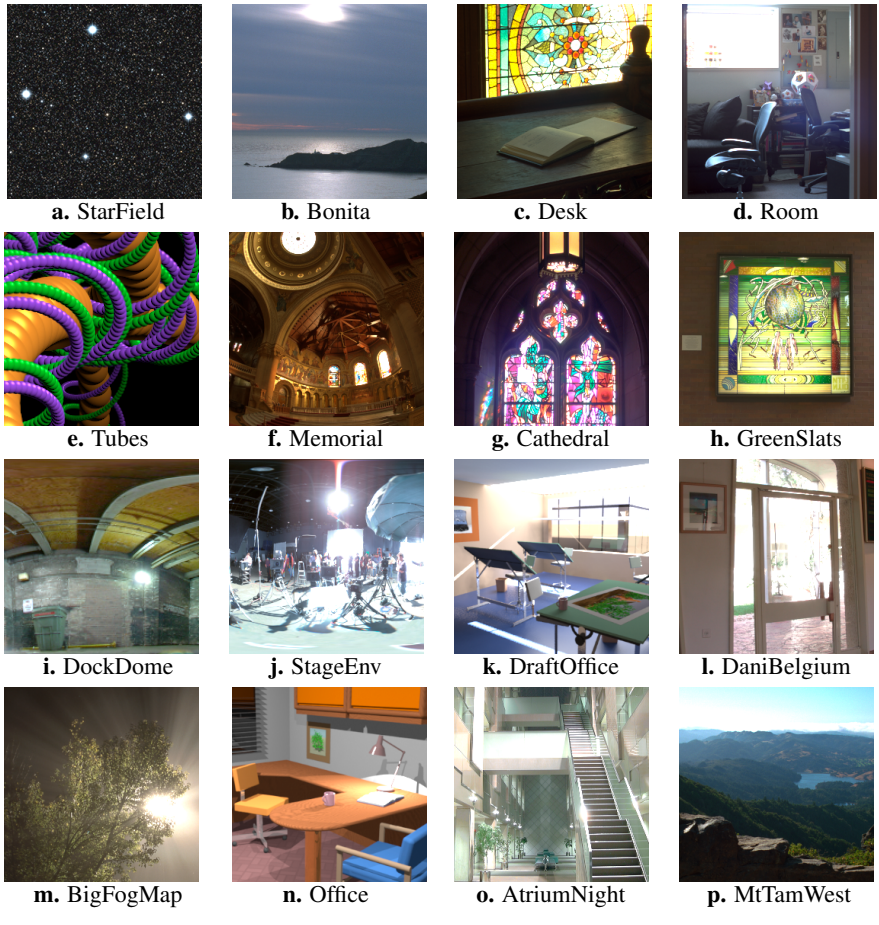
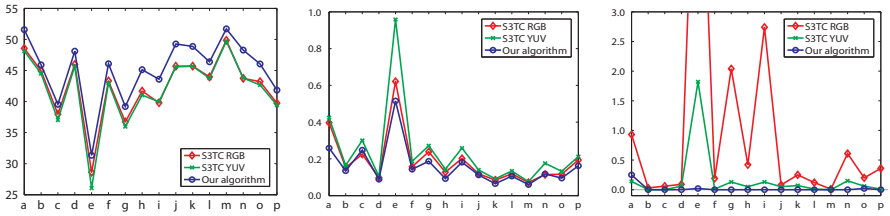
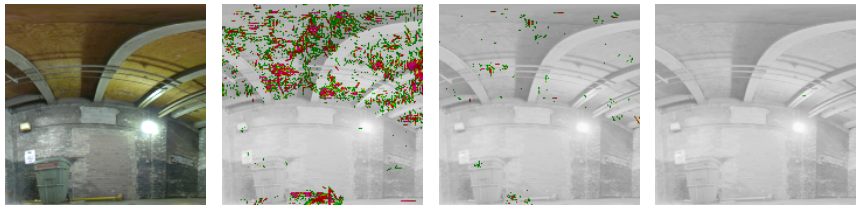


Figure 10: The HDR test images we use for evaluating our algorithms. The figure shows cropped versions of the actual test images. ***(e)****,* ***(k)****, and* ***(n)*** *are synthetic.*



mPSNR (dB) log[*RGB*] RMSE HDR VDP 75% error

Figure 11: These diagrams show the performance of our algorithm compared to the S3TC RGB and S3TC YUV algorithms for each of the images ***(a)–(p)*** *in Figure 10. The mPSNR measure gives consistently better values (left), while the* log[*RGB*] *RMSE (middle) is lower on nearly all of the test images. The HDR VDP luminance measure (right) indicates a perceivable error very close to 0.0% for most images with our algorithm, clearly superior to both S3TC-based algorithms.*



Original S3TC RGB S3TC YUV Our algorithm

Figure 12: HDR VDP difference images. Green indicates areas with 75% chance of detecting an artifact, and red indicates areas with 95% detection probability.

tiles with visible errors, and this is a major strength.

It is very important that a TC format developed for real-time graphics handle mipmapping well. Figure 15 shows the average error from all our test images for the first 7 mipmap levels. The average errors grow at smaller mipmap levels due to the concentration of information in each tile, but our algorithm is still very robust and compares favorably to the S3TC-based techniques. In both error measures, our approach is consistently much better.

8 Conclusions

In this work, we have presented the first low bit rate HDR texture compression system suitable for graphics hardware. In order to accurately represent the wide dynamic range in HDR images, we opted for a fixed rate of 8 bpp. Although it would have been desirable to further reduce the bandwidth, we found it hard to achieve an acceptable image quality at 4 bpp, while preserving the full dynamic range. For future work, it would be interesting to incorporate an alpha channel in the 8 bpp budget.

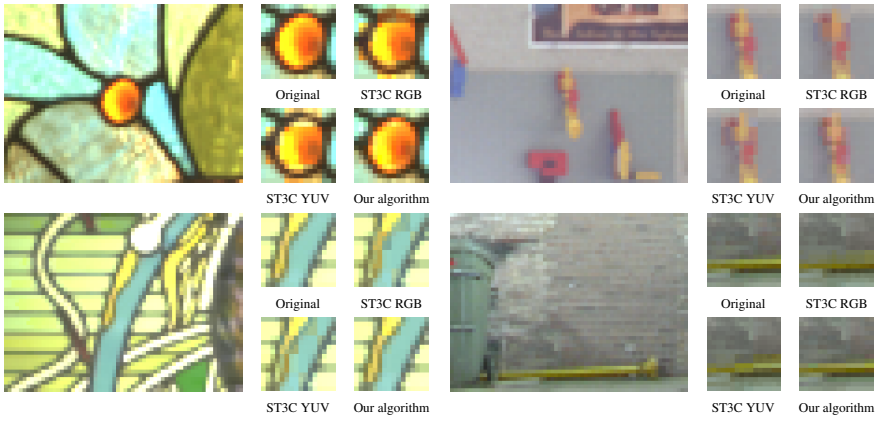


Figure 13: This figure shows magnified parts of the test images ***(c)****,* ***(d)****,* ***(h)****, and* ***(i)****, compressed with our algorithm and the two S3TC-based methods. In difficult parts of the images, the S3TC algorithms sometimes produce quite obvious artifacts, while this rarely happens with our algorithm due to its separated encoding of luminance and chrominance.*

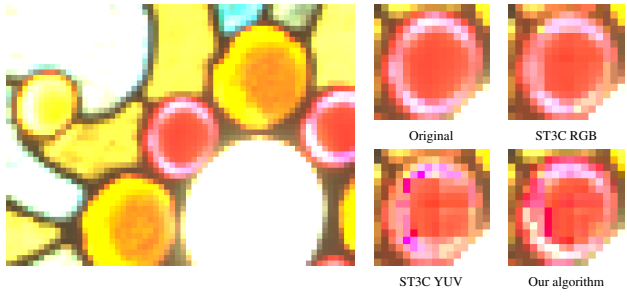


Figure 14: A difficult scenario for our algorithm is an over-exposed image ***(c)*** *with sharp color transitions. S3TC YUV does, however, handle this case even worse. Surprisingly, S3TC RGB performs very well here.*

Our algorithm performs very well, both visually and in the chosen error metrics. However, more research in meaningful error measures for HDR images is needed. The HDR VDP by Mantiuk et al. [13] is a promising approach, and it would be interesting to extend it to handle chroma as well. We hope that our work will further accelerate the use of HDR images in real-time rendering, and provide a basis for future research in HDR texture compression.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research and Vetenskapsrådet. Thanks to Calle Lejdfors & Christina Dege for proof-reading,

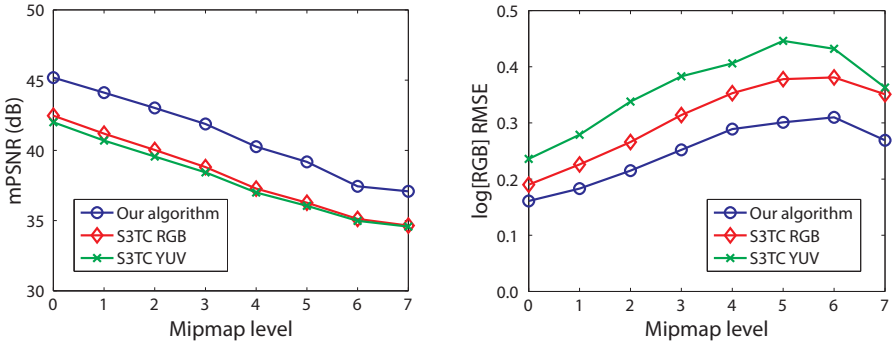


Figure 15: Here, the average mPSNR and $\log[RGB]$ *RMSE values are presented for various mipmap levels of our test images, where 0 is the original image and higher numbers are sub-sampled versions.*

Rafal Mantiuk for letting us use his HDR VDP program, and Apple for the temporary Shake license. Image (**f**) is courtesy of Paul Debevec. (**g**) and (**l**) are courtesy of Dani Lischinski. (**h**) was borrowed from the RIT MCSL High Dynamic Range Image Database. The image (**i**) was created using HDRI data courtesy of HDRIMaps (`www.hdrimaps.com`) from the LightWorks HDRI Starter Collection (`www.lightworkdesign.com`). (**k**) and (**n**) are courtesy of Greg Ward. (**m**) is courtesy of Jack Tumblin, Northwestern University. (**o**) is courtesy of Karol Myszkowski. The remaining images were taken from the OpenEXR test suite.

A mPSNR Parameters

In this appendix, we summarize the parameters used when computing the mPSNR quality measure for the images in Figure 10 (**a**–**p**). For all mPSNR computations, we have computed the mean square error (MSE) only for the integers between the start and stop exposures (as shown in the table below). For example, if the start exposure is -10, and the stop exposure is $+5$, then we compute the MSE for all exposures in the set: $\{-10, -9, \ldots, +4, +5\}$.

Test image	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**
Start exposure	-9	-8	-8	-9	-3	-9	-4	0
Stop exposure	+4	+2	+5	+3	+7	+3	+7	8
Test image	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**
Start exposure	-6	-12	-7	-6	-8	-6	-12	-4
Stop exposure	+3	+1	+2	+5	+2	+3	+1	+5

B Shape Transform Coordinates

Below we present coordinates, (α, β), for each of the template shapes in Figure 8.

Shape	p1	p2	p3	p4
A	(0, 0)	(11/32, 0)	(21/32, 0)	(1, 0)
B	(0, 0)	(1/4, 0)	(3/4, 0)	(1, 0)
C	(0, 0)	(1/8, 0)	(1/4, 0)	(1, 0)
D	(0, 0)	(1/2, 0)	(3/4, 1/4)	(1, 0)
E	(0, 0)	(1/2, 0)	(1/2, 1/2)	(1, 0)
F	(0, 0)	(11/32, 11/32)	(21/32, 11/32)	(1, 0)
G	(0, 0)	(0, 1/2)	(1, 1/2)	(1, 0)
H	(0, 0)	(1/4, 1/4)	(1/2, 0)	(1, 0)

Bibliography

[1] A.C. Beers, M. Agrawala, and Navin Chadda. Rendering from Compressed Textures. In *Proceedings of ACM SIGGRAPH 96*, pages 373–378, 1996.

[2] R. Bogart, F. Kainz, and D. Hess. OpenEXR Image File Format. In *ACM SIGGRAPH Sketches & Applications*, 2003.

[3] Alan Chalmers, Ann McNamara, Scott Daly, Karol Myszkowski, and Tom Troscianko. Image Quality Metrics. In *ACM SIGGRAPH Course Notes*, 2000.

[4] Jonathan Cohen, Chris Tchou, Tim Hawkins, and Paul Debevec. Real-Time High Dynamic Range Texture Mapping. In *Eurographics Workshop on Rendering*, pages 313–320, 2001.

[5] Scott Daly. The Visible Differences Predictor: An Algorithm for the Assessment of Image Fidelity. In *Digital Images and Human Vision*, pages 179–206. MIT Press, 1993.

[6] Paul E. Debevec. Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography. In *Proceedings of ACM SIGGRAPH 98*, pages 189–198, 1998.

[7] Paul E. Debevec and Jitendra Malik. Recovering High Dynamic Range Radiance Maps from Photographs. In *Proceedings of ACM SIGGRAPH 97*, pages 369–378, 1997.

[8] I.L. Dryden and K.V. Mardia. *Statistical Shape Analysis*. Wiley, 1998.

[9] Simon Fenney. Texture Compression using Low-Frequency Signal Modulation. In *Graphics Hardware*, pages 84–91, 2003.

[10] Konstantine Iourcha, Krishna Nayak, and Zhou Hong. System and Method for Fixed-Rate Block-Based Image Compression with Inferred Pixel Values. US Patent 5,956,431, 1999.

[11] Günter Knittel, Andreas G. Schilling, Anders Kugler, and Wolfgang Straßer. Hardware for Superior Texture Performance. *Computers & Graphics*, 20(4):475–481, 1996.

[12] Yuanzhen Li, Lavanya Sharan, and Edward H. Adelson. Compressing and Companding High Dynamic Range Images with Subband Architectures. *ACM Transactions on Graphics*, 24(3):836–844, 2005.

[13] Rafał Mantiuk, Scott Daly, Karol Myszkowski, and Hans-Peter Seidel. Predicting Visible Differences in High Dynamic Range Images – Model and its Calibration. In *Human Vision and Electronic Imaging X*, pages 204–214, 2005.

[14] Rafal Mantiuk, Grzegorz Krawczyk, Karol Myszkowski, and Hans-Peter Seidel. Perception-Motivated High Dynamic Range Video Encoding. *ACM Transactions on Graphics*, 23(3):733–741, 2004.

[15] John D. Owens. Streaming Architectures and Technology Trends. In *GPU Gems 2*, pages 457–470. Addison-Wesley, 2005.

[16] Anton Pereberin. Hierarchical Approach for Texture Compression. In *Proceedings of GraphiCon '99*, pages 195–199, 1999.

[17] Charles Poynton. *Digital Video and HDTV Algorithms and Interfaces*. Morgan Kaufmann Publishers, 2003.

[18] Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec. *High Dynamic Range Imaging: Acquisition, Display and Image-Based Lighting*. Morgan Kaufmann Publishers, 2005.

[19] S. J. Sangwine and R. E. N. Horne, editors. *The Colour Image Processing Handbook*. Chapman and Hill, 1998.

[20] Helge Seetzen, Wolfgang Heidrich, Wolfgang Stuerzlinger, Greg Ward, Lorne Whitehead, Matthew Trentacoste, Abhijeet Ghosh, and Andrejs Vorozcovs. High Dynamic Range Display Systems. *ACM Transactions on Graphics*, 23(3):760–768, 2004.

[21] Helge Seetzen, Lorne A. Whitehead, and Greg Ward. A High Dynamic Range Display Using Low and High Resolution Modulators. *Society for Information Display Internatiational Symposium Digest of Technical Papers*, pages 1450–1453, 2003.

[22] Jacob Ström and Tomas Akenine-Möller. iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. In *Graphics Hardware*, pages 63–70, 2005.

[23] Jay Torborg and Jim Kajiya. Talisman: Commodity Real-time 3D Graphics for the PC. In *Proceedings of SIGGRAPH*, pages 353–364, 1996.

[24] Greg Ward. Real Pixels. In *Graphics Gems II*, pages 80–83. Academic Press, 1991.

[25] Greg Ward. High Dynamic Range Image Encodings, `http://www.anyhere.com/`, 2005.

[26] Greg Ward and Maryann Simmons. Subband Encoding of High Dynamic Range Imagery. In *Proceedings of APGV '04*, pages 83–90, 2004.

[27] Gregory J. Ward. The RADIANCE Lighting Simulation and Rendering System. In *Proceedings of ACM SIGGRAPH 94*, pages 459–472, 1994.

[28] Gregory Larson Ward. LogLuv Encoding for Full Gamut High Dynamic Range Images. *Journal of Graphics Tools*, 3(1):15–31, 1998.

[29] Ruifeng Xu, Sumanta N. Pattanaik, and Charles E. Hughes. High-Dynamic-Range Still-Image Encoding in JPEG 2000. *IEEE Computer Graphics and Applications*, 25(6):57–64, 2005.

Paper II

Practical HDR Texture Compression

Jacob Munkberg Petrik Clarberg Jon Hasselgren Tomas Akenine-Möller

Lund University

ABSTRACT

The use of high dynamic range (HDR) textures in real-time graphics applications can increase realism and provide a more vivid experience. However, the increased bandwidth and storage requirements for uncompressed HDR data can become a major bottleneck. Hence, several recent algorithms for HDR texture compression have been proposed. In this paper, we discuss several practical issues one has to confront in order to develop and implement HDR texture compression schemes. These include improved texture filtering and efficient offline compression. For compression, we describe how Procrustes analysis can be used to quickly match a predefined template shape against chrominance data. To reduce the cost of HDR texture filtering, we perform filtering prior to the color transformation, and use a simple trick to reduce the incurred errors. We also introduce a number of novel compression modes, which can be combined with existing compression schemes, or used on their own.

Computer Graphics Forum 27(6):1664–1676, 2008.

1 Introduction

A general trend in computer architecture is that computing power growth is much faster than the corresponding growth in memory access speeds [10]. This implies that the available memory bandwidth should be regarded as a scarce resource and be exploited as best as possible. One way is to use different *compression* techniques to reduce the required memory bandwidth. For graphics processing units (GPUs), there are many types of compression, such as buffer compression (e.g., depth, color, and stencil), vertex compression, and texture compression.

The focus in this paper is on *texture compression* (TC), where a texture is simply a read-only image. A texture is stored in compressed form in memory, and when the GPU requests access to a small part of the texture, the desired part is sent in compressed form over the bus. The GPU then decompresses the data. The pixels in a texture can be accessed in any order, any number of times, and hence it is of uttermost importance to provide random access in constant time. This has a number of implications, including that the majority of TC schemes operate on blocks of pixels (e.g., 4×4), and compress such a block to a fixed number of bits (e.g., 4 bits per pixel). In general, this also means that TC schemes are *lossy*.

High dynamic range (HDR) images [12] have had a big impact on the computer graphics community. Lately, these are used as textures in real-time rendering as well, and therefore, methods for HDR texture compression [9, 15, 19] have been developed. Roimela et al. [15] try to minimize the hardware cost, and propose a simplified color space. They subsample chrominance, and use a quick floating-point trick to convert to approximately logarithmic luminance. In a recent paper [14], they extend the algorithm with a more compact chrominance encoding and higher luminance fidelity.

Munkberg et al. [9] also develop a new compression algorithm, which uses an S3TC-like [4] encoding for the luminance, and introduce *shape transforms* for flexible chrominance encoding. All of these methods compress 4×4 pixel blocks down to eight bits per pixel (bpp). Wang et al. [19] propose an algorithm based on existing texture compression hardware, rather than targeting new hardware mechanisms, but only achieve a compression rate of 16 bpp. Some details of these schemes will be discussed in later sections. In this paper, we bring to light several practical issues when dealing with HDR texture compression.

To evaluate the quality of a compression algorithm, an error metric has to be used. However, standard error metrics are not suitable for HDR data due to the larger dynamic range and higher precision of floating-point numbers. A few HDR error metrics have been proposed, although more work needs to be done in this area. Section 2 gives a brief review of existing metrics. Similarly, color spaces must be defined with HDR data in mind so that efficient compression is achieved, which often means that a non-linear color space is preferred. In Section 3, we review the color spaces currently used for HDR texture compression. The use of non-linear color spaces complicates texture filtering, as each fragment must be converted to RGB-space prior to filtering for correct results. In Section 4, we introduce a novel

algorithm for texture filtering in non-linear color spaces. This is important as it reduces the cost of hardware filtering.

To improve the compression quality, Section 5 introduces a new algorithm with better quality than previous algorithms at 8 bpp. Our algorithm builds on the method of Munkberg et al. [9], but adds a novel compression mode focused on improving the chrominance precision. In addition, several other ideas on HDR texture compression algorithms are described in Section 6. The implementation of our codec is discussed in Section 7, where the details of how we use Procrustes analysis and clustering are presented for the first time. Furthermore, we use a non-linear optimization trick for improved chrominance quality and we analyze the desired precision for luminance. We compare all existing algorithms, including our new HDR TC scheme in Section 8, and finally we offer some conclusions in Section 9.

2 HDR Error Metrics

When you cannot foresee or predict the usage of an HDR texture, you need to make sure the accuracy of every value, regardless of its absolute magnitude, is preserved as well as possible. For example, a very dark region can become bright and details can appear after tone-mapping. Similarly, the details of a very bright region can become visible if the exposure is low. Standard LDR error metrics are therefore not suitable. In this paper, we use three HDR error metrics: the logarithmic error, multi-exposure PSNR, and HDR-VDP.

Xu et al. [20] compute the *root mean square error* (RMSE) of the compressed image in the log[RGB] color space. More formally, if $(\hat{r},\hat{g},\hat{b})$ denotes the compressed texel color and (r,g,b) is the original color, the error is defined as:

$$\sqrt{\frac{1}{N}\sum\left(\log_2\left(\frac{\hat{r}}{r}\right)\right)^2+\left(\log_2\left(\frac{\hat{g}}{g}\right)\right)^2+\left(\log_2\left(\frac{\hat{b}}{b}\right)\right)^2}, \tag{1}$$

where N is the number of pixels in the texture.

The *multi-exposure PSNR* (mPSNR) error measure computes the standard mean square error for a range of tone-mapped exposures of the HDR image, and averages them together. Then the standard formula for computing peak signal-to-noise ratio (PSNR) is applied. For details, see Munkberg et al. [9]. Roimela et al. [14] compute the PSNR on the L^* and a^*b^* channels in the CIE 1976 $L^*a^*b^*$ color space with the motivation that it is perceptually linear.

HDR-VDP [6] is an extension of the *visual difference predictor*, which finds perceived differences over the dynamic range of the image. The current implementation only works on the luminance channel, so perceived chrominance artifacts are not detected.

3 HDR Color Spaces

In this section, we will present the color spaces used in previous HDR texture compression schemes, and shortly discuss their characteristics.

3.1 LogYuv

Munkberg et al. [9] use a *logYuv* color space with logarithmic luminance and two chrominance channels:

$$\begin{aligned} Y &= w_r R + w_g G + w_b B \\ (\bar{Y}, \bar{u}, \bar{v}) &= \left(\log_2 Y, w_b \frac{B}{Y}, w_r \frac{R}{Y} \right) \\ (R, G, B) &= \left(\frac{1}{w_r} \bar{v} 2^{\bar{Y}}, \frac{1}{w_g} (1 - \bar{u} - \bar{v}) 2^{\bar{Y}}, \frac{1}{w_b} \bar{u} 2^{\bar{Y}} \right) \end{aligned} \tag{2}$$

The luminance channel is encoded as a weighted combination of the RGB channels, with weights according to the Rec. 601 [11] standard. By storing log-luminance, $\bar{Y} = \log_2 Y$, the maximum *relative* luminance error can be effectively bounded over the entire dynamic range. The two chrominance channels are constructed to be in the $[0,1]$ interval. This gives a decorrelated and compact representation of HDR texture data, with HDR information concentrated to the luminance channel. This color space is used in the proposed format in Section 5 of this paper.

A minor issue with logarithm-based luminance encodings is that entirely black pixels, RGB=(0,0,0), need to be treated with care, as $\log x \to -\infty$ when $x \to 0^+$ A simple solution is to clamp the log-luminance to the smallest representable value, $\bar{Y}_{\min}$. However, in some cases it may be desirable to have a true $(0,0,0)$ black level. A possible solution, which we use, is to define the inverse luminance transform as:

$$Y = \begin{cases} 0, & \text{if } \bar{Y} = \bar{Y}_{\min}, \\ 2^{\bar{Y}}, & \text{otherwise.} \end{cases} \tag{3}$$

However, it should be noted that true black pixels rarely occur in natural images, and only sometimes in artificial images.

3.2 Roimela et al.

Roimela et al. [15] use a different transform in order to provide very efficient decoding. The forward and inverse transforms are shown below:

$$\begin{aligned} \tilde{Y} &= \frac{R + 2G + B}{4} \\ (\tilde{Y}, \tilde{u}, \tilde{v}) &= \left(\tilde{Y}, \frac{R}{4\tilde{Y}}, \frac{B}{4\tilde{Y}} \right) \\ (R, G, B) &= \tilde{Y}(4\tilde{u}, 2(1 - \tilde{u} - \tilde{v}), 4\tilde{v}) \end{aligned} \tag{4}$$

Compared with logYuv of Munkberg et al. [9], the luminance is here encoded with simplified weights with hardware-friendly constants. Again, the color space decorrelates the HDR data in one luminance and two chrominance channels.

3.3 LUVW

Wang et al. [19] use a color space called *LUVW* with four channels, where the luminance information is concentrated to the L channel as follows:

$$\begin{aligned} L &= \sqrt{R^2+G^2+B^2} \\ (U,V,W) &= \left(\frac{R}{L},\frac{G}{L},\frac{B}{L}\right) \end{aligned} \tag{5}$$

The L channel is the length of the RGB vector, and the UVW channels are divided by L to get three values in the range $[0,1]$. This is not as compact as the previous approaches, as four channels are stored, but allows for simplified texture filtering on existing hardware.

3.4 Log[RGB]

The log[RGB] color space [20] is simply defined as the logarithm of the R, G and B components:

$$(R',G',B') = (\log_2 R, \log_2 G, \log_2 B) \tag{6}$$

The motivation is that quantization gives a constant, or nearly constant, relative error over the entire dynamic range. This makes it a good choice for measuring the error in each color channel. However, this transform often fails to decorrelate the image data, so all channels needs to be stored with equal resolution, which makes it less suitable for compression algorithms.

4 Texture Filtering

Texture filtering is crucial in real-time graphics, and all modern GPUs support fast bilinear and trilinear filtering. As the most widely used color space on GPUs is RGB, a filtered texture lookup is expected to perform a linear weighting of each of the R, G, and B channels. In the one-dimensional case, we have:

$$r = (1-\alpha)r_1 + \alpha r_2, \tag{7}$$

for linear interpolation between the red components, r_1 and r_2, of two texels (similarly for green and blue).

For compressed HDR textures using a non-RGB space, texture filtering can be performed before *or* after converting to RGB. In post-conversion filtering, we have to perform multiple color transforms for each filtered lookup, e.g., four $\bar{Y}\bar{u}\bar{v} \rightarrow RGB$

transforms in a bilinear two-dimensional lookup, which may harm performance, or increase the cost in terms of extra hardware for duplicated color space transform units. However, filtering before conversion gives deviating results, as the currently used HDR color spaces (Section 3) involve a non-linear transform, i.e., division by the luminance. According to Wang et al. [19], this is a minor issue as most blocks have a small dynamic range. However, in areas with sharp luminance transitions, we have observed clearly visible artifacts (color shifts). See Figure 2 for an example. We propose a simple way to reduce the problem.

Consider the logYuv color space in Section 3.1. The red channel is reconstructed as $r = \bar{v}Y/w_r$, and correct post-transform filtering yields:

$$r = (1-\alpha)r_1 + \alpha r_2 = \frac{1}{w_r}\left[(1-\alpha)\bar{v}_1 Y_1 + \alpha \bar{v}_2 Y_2\right]. \tag{8}$$

If we instead interpolate the Y, $\bar{u}$, and $\bar{v}$ components prior to the color transform, i.e., $Y = (1-\alpha)Y_1 + \alpha Y_2$ and similarly for $\bar{u}$ and $\bar{v}$, we get:

$$\begin{aligned} r = \frac{1}{w_r}\bar{v}Y &= \frac{1}{w_r}\left((1-\alpha)\bar{v}_1 + \alpha\bar{v}_2\right)\left((1-\alpha)Y_1 + \alpha Y_2\right) \\ &= \frac{1}{w_r}\left[(1-\alpha)\bar{v}_1 Y_1 + \alpha\bar{v}_2 Y_2\right] \; + \; \varepsilon, \end{aligned} \tag{9}$$

where the error term, ε, depends on the difference between the two luminance values, $\Delta Y = Y_2 - Y_1$, as follows:

$$\varepsilon = \frac{1}{w_r}\alpha(\alpha-1)(\bar{v}_2 - \bar{v}_1)\Delta Y. \tag{10}$$

This error is due to the non-linearity of the color transform. However, we note that the error goes to zero as ΔY gets smaller. We can exploit this to improve the interpolation quality by normalizing the luminance values so that $Y_1 \approx Y_2$. By shifting the bits of Y_2 to make it roughly the same magnitude as Y_1, we drastically reduce the error. At the same time, we must also shift the bits of $\bar{v}_2$ in the opposite direction to keep the term $\bar{v}_2 Y_2$ constant. The net effect is a change of variables to $Y_2' = Y_2/c$ and $\bar{v}_2' = c\bar{v}_2$, where $c = 2^k$. The number of bits to shift, k, is chosen by comparing the magnitudes of Y_1 and Y_2.

Consider the example of interpolating between two texels with $\bar{v}_1 = 0.25$, $\bar{v}_2 = 0.1$, $Y_1 = 1$, and Y_2 varying between 0.1 and 10. The absolute error in the red component with α arbitrarily set to 0.5, grows from roughly -0.1 to over 1.1 as the luminance Y_2 increases. This is shown as the blue line in Figure 1. With our luminance normalization, the error is much smaller (red line).

With our color transform (Section 3.1), the green component is computed by subtraction of the two chrominance channels: $G = (1-\bar{u}-\bar{v})Y/w_g$. This causes a minor difficulty, as this formula is no longer valid if we rescale Y, $\bar{u}$, and $\bar{v}$. One solution is to replace the constant 1 with the bit shift factor, $c = 2^k$, in the transform, i.e., $G = (c-\bar{u}'-\bar{v}')Y'/w_g$. Note, as c may differ for the two pixels, we

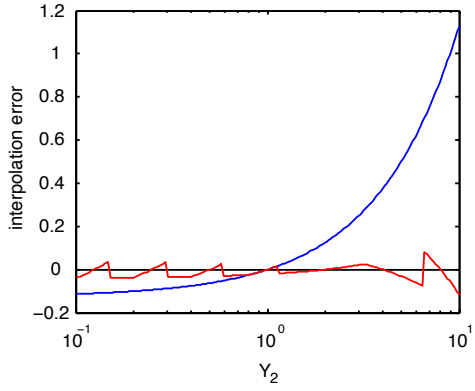


Figure 1: Example of the interpolation error (blue line) introduced by performing texture filtering in the log-luminance color space prior to color transformation. By normalizing the luminance values with simple bit shifts, we effectively reduce the error (red line).

need to interpolate its value: $c = (1-\alpha)c_1 + \alpha c_2$. This adds a small cost, but the savings compared to performing a full color transform for all pixels prior to filtering should be significant. In this discussion, we have studied the one-dimensional case, but the theory extends naturally to bilinear and trilinear filtering, as well as higher dimensions. We show comparison images with and without this pre-filter correction in Figure 2. These images were first compressed using our algorithm and then bilinearly filtered with and without the texture filtering correction discussed above.

5 Improved Shape Transform Compression

In this section, we extend the HDR compression algorithm based on shape transforms [9] (see also Section 7.1) to allow for more freedom in the chrominance representation. The original algorithm quantizes the chrominance information aggressively to allocate enough space for a high quality luminance encoding. The ratio between bits spent on luminance and chrominance is 5 : 3. In most cases, this is a preferred bit layout, as luminance artifacts are more visually disturbing. However, in regions with smaller luminance range but large chrominance variations, a different bit layout may be more appropriate.

In the original encoder, each luminance value was encoded with 4 bits selecting a value linearly interpolated between two 8-bit end points, resulting in 80 bits for a 4×4 block of pixels ($16 \times 4 + 2 \times 8 = 80$). Reducing the luminance resolution to 3 bits, we can encode the luminance for a 4×4 block in $16 \times 3 + 16 = 64$ bits. On a 128-bit budget, this leaves 64 bits to encode chrominance, resulting in a ratio 1 : 1 between luminance and chrominance. Figure 3 shows the bit layout of the two modes. In the original paper, one source of artifacts is the chrominance

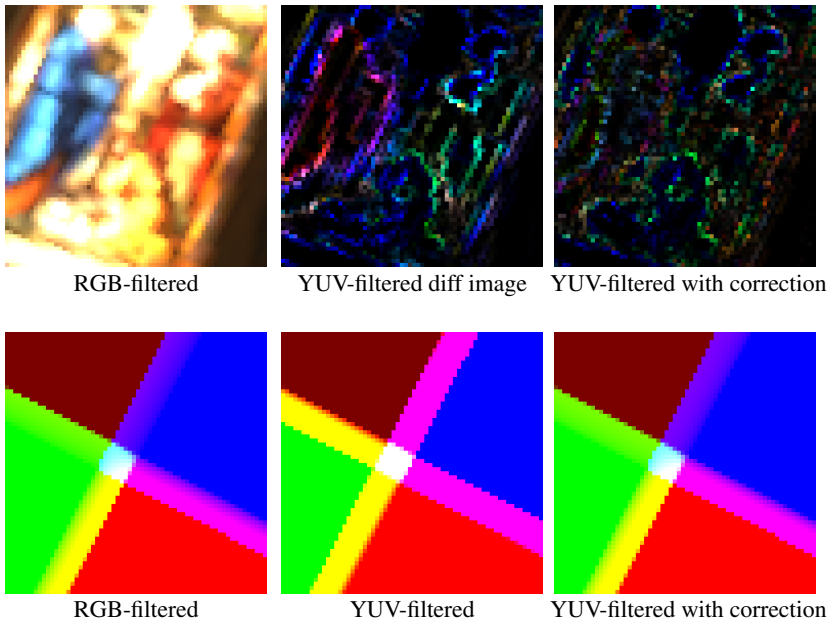


Figure 2: Bilinear filtering examples. The upper row shows a filtered cutout from the 'memorial' image with difference images to highlight the errors. In natural images, pixel-sharp edges are rare, and filtering artifacts are not very visually disturbing. In the example, there are no clearly visible differences, so we present only difference images ($\times 10$) for this example. The lower row shows a worst-case bilinear filtering scenario (after tone-mapping) where there are sharp color and luminance transitions. Here the artifacts are clearly visible. The four colors are (0.1,0,0), (0,10,0), (0,0,1000) and (1,0,0). As can be seen, our texture filtering correction reduces the errors for both examples.

subsampling, forcing nearby texels to have the same encoded chrominance. With an increased chrominance bit budget, we avoid subsampling and allow for higher chrominance detail.

By combining these two modes, we have an algorithm that is more flexible and handles difficult chrominance regions better than before. In practice, we need one bit for indicating which of these two modes is used per block. We remove the option of non-linear luminance distribution from the format [9], and use that bit. The hardware cost of our algorithm is very modest as many parts of the decoder can be shared between the two modes. It is mostly a matter of reallocating the existing resources required for the original algorithm. Our new format is also more suitable for LDR textures, as they have a limited luminance range. A standard test suite of images is evaluated in Figure 10.

A remaining difficult case for the combined algorithm is slow chrominance gradients. Each block has only four chrominance values to choose from, resulting in a limited color resolution. It can be argued that most gradients in natural images are due to luminance changes from shadow/occlusion and these are covered by the

high luminance resolution of the algorithm. However, one can construct examples where the limited chrominance resolution is obvious. To handle these cases, one approach is to include a high resolution variant of S3TC, using a line in RGB or logYuv space with higher end point resolution and more values along the line. This is further described below, but this mode is not included in our results presented in Section 8.

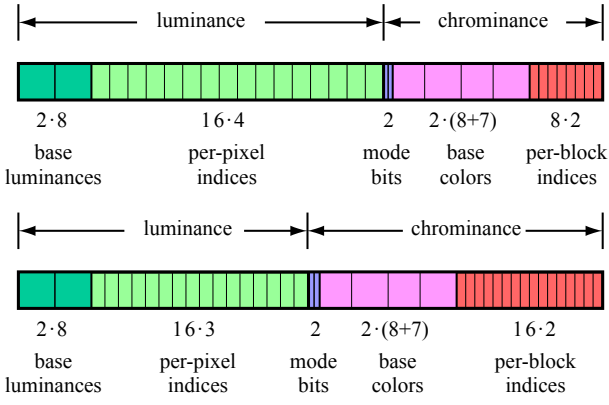


Figure 3: The bit allocation for the two modes included in our new format. The upper figure shows the bit layout used by Munkberg et al. [9], and the lower shows the bit layout for our new mode, allocating more bits for chrominance.

6 Rejected Compression Modes

Our algorithm presented in Section 5 includes two compression modes based on shape transforms [9], with different bit allocations between the luminance and chrominance channels. If more resources are available, it may be beneficial to include other compression modes in the format. In this section, we discuss a number of alternative compression modes that were tried during the development of Munkberg et al.'s format. Although not included in our current format, we believe these new modes present many valueable insights and may be of use to other people in the field.

During the encoding of a block, each of the compression modes is tested, and the one with the smallest error in the chosen metric (we use log[RGB] error) is selected. Hence, the inclusion of alternative compression modes can ideally only improve the result, never increase the error. This assumes we have one or more spare bits to indicate which compression mode a block uses. If no extra bits are available, we have to "steal" them from the encoded data, e.g., by quantizing harder, which may reduce the quality.

An illustration of the usage frequency of the presented additional modes is given in

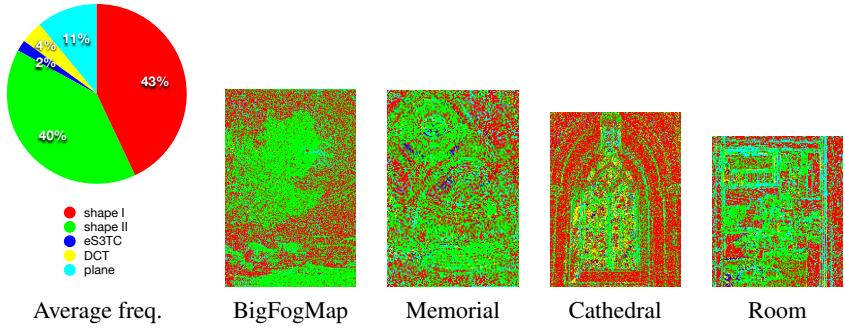


Figure 4: The leftmost diagram shows the average usage frequency of each of our presented algorithms on our suite of test images. ***shape I*** *refers to the new mode (Section 5) with increased chrominance precision,* ***shape II*** *to Munkberg et al. [9],* ***S3TC*** *is an S3TC line with 32 levels in RGB space,* ***DCT*** *a fixed-rate DCT codec, and* ***plane*** *encodes luminance with two planes and chrominance as in the shape I mode. The color-coded images to the right illustrate the usage of the algorithms in different image regions. As seen in the figures, the proposed format (shape I + shape II) is used in more than 80% of the blocks, with the two modes complementing each other well.*

Figure 4. In total, the new modes are only used in on average 17% of the blocks. Thus, we do not believe the added hardware complexity is motivated by a large enough increase in quality.

6.1 Extended S3TC

It is easy to extend the S3 texture compression (S3TC) format [4] to handle HDR images. Traditional S3TC compresses a 4×4 pixel block by storing two reference colors with 16 bits (RGB565) each, and creates a color palette from the reference colors and two additional colors computed through linear interpolation. Each pixel in the block is given a 2-bit index, which is used to select one color from the palette. The format thus requires $16 \times 2 + 4 \times 4 \times 2 = 64$ bits per block.

We extend S3TC by using 2×24 bits (RGB888) for the reference colors and a color palette with 32 entries, computed using linear interpolation between the base colors just like traditional S3TC. As a consequence, a 5-bit index is needed for each pixel, resulting in $2 \times 24 + 16 \times 5 = 128$ bits per block, which was our target bit-rate. We also experimented with a palette of 16 colors and 2×32 bits for the reference colors.

Our extended S3TC works remarkably well for blocks with smooth gradients, and blocks with mostly luminance features. However, the linear approach fails for blocks with three or more distinct colors. This often shows as block artifacts in regions with complex chrominance.

6.2 Fixed-rate DCT-based Compression

The *discrete cosine transform* (DCT) is an energy compaction transform that is popular in image and video compression. It is, for example, used in the JPEG and MPEG standards [11]. These formats use a variable bit rate to allow for a higher precision in important transform coefficients.

In our application, where we require a fixed rate and no global per-texture data, the best we can do is to design an algorithm that works well on average. We work in the logYuv space, and allocate 64 bits to the luminance channel and 32 bits to each of the chrominance channels. To select the bit allocation within each channel, we estimated the variance of each DCT coefficient over all 4×4 blocks in a set of 18 HDR images of both natural and synthetic origin. These estimated variances were used to find a bit allocation table that minimizes the average reconstruction error using Lagrange minimization [1, 16].

The quantization method plays a vital role in the performance of the algorithm. For the DC-components, we used uniform quantization over the range of possible values, as we do not want to favor any particular luminance or chrominance range. Consistent with previous work [13, 17], we found that the AC-components approximately follow a Laplacian distribution. Therefore, we applied non-uniform midtread quantization [16] optimized for the Laplacian distributions given by the estimated variances. The additional cost of using non-uniform quantization as opposed to uniform quantization was well motivated by the increase in quality. In hardware, the reconstruction translates to simple table lookups.

The fixed-rate DCT-based approach works well for a large number of blocks. However, it has a number of drawbacks. First, block artifacts between adjacent blocks are relatively common. These look like typical JPEG-artifacts, and are rather disturbing. To some extent, this could be remedied by taking a larger neighborhood into account when computing the quantization levels. For blocks with a large dynamic range, it proved difficult to find quantization levels that work well. In addition, the decompression is relatively expensive. In summary, the best option is probably to limit the use of DCT-based compression to blocks with smooth gradients and other low-frequency features.

6.3 Plane Encoders for Luminance

The luminance values in a block can be seen as a height-field, $z = f(x,y)$, where x,y is the pixel coordinate of a point, and the z-value is the luminance. One option for encoding luminance is to store the equation of the plane that approximates the z-values as closely as possible: $f(x,y) = z_0 + x \cdot \Delta x + y \cdot \Delta y$, where z_0 is a constant offset, and Δx and Δy are the slopes. Finding the optimal plane parameters is a simple linear optimization problem. However, an encoder based on a single plane is rather restricted, as only linear luminance gradients can be represented.

A more general approach is to store *two* (or more) plane equations for each block, and a per-pixel index to choose between them. If more bits are available, it is also

possible to add intermediate levels in between the planes. For example, a 2-bit per-pixel index can be used to select between the two planes and two additional levels at $1/3$ and $2/3$ between the planes, similar to the work by Fenney [3].

We have implemented a simple plane encoder for luminance information, storing two plane equations with a 10-bit offset and two 7-bit deltas each. A 16-bit mask was used to select which of the two planes each pixel belongs to. This gives a total bit count of $2\times(10+7+7)+16=64$ for the luminance encoder. To find the plane equations, we used exhaustive search and regression, but more intelligent methods can be developed. Our luminance plane encoder was combined with the chrominance encoding using shape transforms as described in Section 5.

Our experiments with the plane encoders yielded promising results. Many blocks contain relatively smooth gradients, and the two-plane encoder is good at handling the case of two partially overlapping surfaces of different luminance. However, areas with complex high-frequency features are poorly represented. The main drawback of the method is that the restriction of the luminance to linear planes can lead to visually noticeable block artifacts. Hence, the method must be combined with other, more flexible, encodings.

7 Implementation

In this section, we describe the details of the compression algorithms based on shape transforms (Section 5).

7.1 Shape Transforms

Shape transforms [9] is a compact way of encoding the 2D chrominance points of a block. Each block stores a scaled and rotated template shape chosen out of the set of pre-defined shapes shown in Figure 5, and a 2-bit index is used to indicate the nearest chrominance point for each pixel or group of pixels. The transform parameters, i.e., scale and rotation, are implicitly encoded by storing the location of two base colors, marked as black dots in the figure.

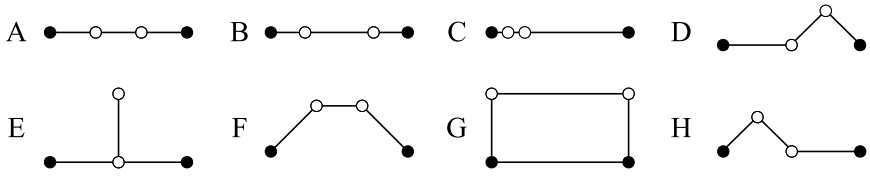


Figure 5: The set of template shapes used for representing chrominance information of a block.

To find the transform parameters that give the best match for the chrominance

information, we have to minimize the shape fitting error. We use the squared distance between the original and the compressed chrominance points:

$$E = \sum_{i=1}^{16} \left[(\bar{u}_i - \bar{u}'_i)^2 + (\bar{v}_i - \bar{v}'_i)^2 \right], \tag{11}$$

where $(\bar{u}_i, \bar{v}_i)$ is the original chrominance point for pixel i, and $(\bar{u}'_i, \bar{v}'_i)$ is the corresponding point after compression. The error is evaluated for the best fit of each template shape, and we select the shape with the smallest overall error.

In practice, minimizing the shape fitting error (Equation 11) is a relatively hard problem. The error function has four degrees of freedom (the location of two base colors), and 16 different $(\bar{u}_i, \bar{v}_i)$-pairs for a 4×4 block, that each snaps to the nearest compressed chrominance point. Exhaustive search is possible, but impractical. If the base colors are quantized to 8 bits per component, there are 2^{32} combinations to try for *each* block. For a quick approximate solution, it is possible to use *Procrustes analysis*, which is further described in the next section. Other alternatives we have tried include *simulated annealing* and *exhaustive search* (see below).

After shape fitting, the position of the two base colors are encoded as two fixed-point values. The limited precision introduces some additional compression error. To further improve the solution, we search in a small radius of quantized values around each base point to minimize the mean square error for the reconstructed chrominance block.

Procrustes Analysis and Clustering

A landmark is a specific feature of an object, in our case represented as 2D coordinates. The idea behind Procrustes analysis [2] is to compare the shapes of objects, represented as sets of landmarks, by removing translation, rotation and scaling. More formally, this analysis finds the similarity transformations to be applied to one set of landmarks, X_1 (template shape coordinates), which minimize its Euclidean distance from a second set, X_2 (chrominance values). These are: b (uniform scaling), $\mathbf{R}$ (rotation) and $\mathbf{v}$ (translation), which minimize the functional:

$$\|X_2 - bX_1\mathbf{R} - \mathbf{1}_k\mathbf{v}^T\|^2. \tag{12}$$

The problem of finding the parameters that minimize this functional has an exact, fast solution: First, center X_1 and X_2 by subtracting the average from each coordinate. $\mathbf{v}$ is given as the average of X_2 prior to centering. Form the matrix $\mathbf{A} = {X_2}^T X_1$, and apply a singular value decomposition $\mathbf{A} = \mathbf{VSU}^T$. The transform parameters that minimize the functional above are given by (where trace is the sum of the diagonal elements of a square matrix):

$$\mathbf{R} = \mathbf{UV}^T,$$

$$b = \frac{\text{trace}(\mathbf{AR})}{\text{trace}({X_1}^T X_1)}. \tag{13}$$

With 2D points, the matrix $\mathbf{A}$ is of size 2×2, so the SVD decomposition is lightweight.

We use blocks of 4×4 texels, containing 16 chrominance points, so the problem is to fit a shape with four landmarks to 16 chrominance points $(\bar{u}, \bar{v}) \in [0,1]^2$. To set up the point correspondences, we create up to four clusters of the chrominance points using *pnn*-clustering and the *k-means* algorithm [16]. Clustering the points is an approximation of the optimal solution to the fitting problem, but it allows us to compare blocks against template shapes in constant time.

Procrustes analysis needs consistent ordering of the two sets of landmark points. Therefore, each cluster is linked to a point on the template shape. With four landmarks per shape, the number of unique mappings is $4! = 24$, and we test all combinations. It is worth noting that for a problem with a larger set of landmarks per shape, this is obviously not a feasible approach. Numbering schemes based on the template shape geometry can be developed to avoid this brute-force solution.

Once the point correspondences are set up, each cluster $k \in \{1 \ldots 4\}$, is assigned a gravity point and a weight $w_k = n_k/16$, where n_k is the number of points in cluster k. In order to take the number of points per cluster into account, we multiply the functional above with a diagonal matrix:

$$\mathbf{W} = \begin{bmatrix} w_1 & 0 & 0 & 0 \\ 0 & w_2 & 0 & 0 \\ 0 & 0 & w_3 & 0 \\ 0 & 0 & 0 & w_4 \end{bmatrix}, \tag{14}$$

containing the cluster weights w_k. Our new functional is:

$$\|\mathbf{W}(X_2 - bX_1\mathbf{R} - \mathbf{1}_k\mathbf{v}^T)\|^2, \tag{15}$$

which favors solutions with close fits for clusters containing many points. Another possibility is to duplicate template points according to the number of points in the matching cluster, avoiding the need of cluster weights altogether. We have evaluated both approaches, and they give equal quality in our tests. We use the former approach in our implementation as it is slightly faster.

The shape fitting routines were implemented in C++. The Procrustes step is fast, as we are only interested in a 2D fit, and the most complex step is the singular value decomposition of a 2×2 matrix. As previously discussed, we need to set up point correspondences in order to use Procrustes analysis, and we tested both ordering schemes and a brute-force solution. The latter was selected as the many special cases of the former made it more error-prone and inflexible when adding new template shapes. A 1024×1024 image is encoded in about a minute, using non-optimized code. The same image is decompressed in a fraction of a second using our software decompressor.

Simulated Annealing and Exhaustive Search

Our shape fitting algorithm based on Procrustes analysis and clustering works very well in practice, but it should be noted that it is a heuristic and is not guaranteed to yield optimal results. We have also examined two other approaches for finding shape parameters, based on simulated annealing and exhaustive search.

Simulated annealing (SA) [5] is a probabilistic optimization algorithm that finds some minimum (not necessarily the global minimum) through a series of small random steps of decreasing length. Unlike greedier methods, spurious "uphill" moves are allowed, which makes the algorithm less prone to getting stuck at local minima. In our case, we wish to minimize the four-dimensional function $f(u_1, v_1, u_2, v_2)$, which takes the two base colors used to represent a shape as input and computes the error according to Equation 11.

When compared to Procrustes analysis, we found that SA is prone to generate a few bad blocks per image due to its probabilistic nature. This problem can be reduced by running more iterations for blocks with higher errors, but it is very hard to reach the same level of stability and performance as we got with Procrustes analysis. SA can potentially generate better shape fits, but only if a very large number of iterations are used, leading to excessive compression times.

Our exhaustive search used the same functional representation as for simulated annealing, but here we redefine the problem using interval arithmetic [8]. That is, the function takes intervals as parameters and computes interval bounds of the error. We search for the global minimum by evaluating the function over the entire search space, and then recursively split the search space into two halves. The traversal of splits are sorted by error intervals, and when we find a solution, it can be used to cull further traversal. This search is exhaustive since we continue the recursion until we reach the resolution used to store the base colors.

Although exhaustive search provides optimal results, it is not practically useful due to the extreme compression times. Even images of moderate sizes take over a day to compress. However, exhaustive search can be used as a reference solution, against which other optimization algorithms can be compared.

7.2 Optimizing with a Non-linear Error Function

The shape transform algorithm enables a compact representation of chrominance information. However, the quantization of the base colors and the limited set of shapes introduce small errors. We have noticed that the $(\bar{u}, \bar{v})$ values in natural images are typically concentrated to a small region close to the $\bar{u}$ and $\bar{v}$ axes. Figure 6 (left) shows an example of this. Therefore, it is often better to use a non-linear error function that gives more weight to small chrominance values.

We employ a transform, $f(x) = x^{\alpha}$, to the $(\bar{u}, \bar{v})$ points prior to measuring the shape fitting error. By choosing the constant α in the range $[0, 1]$, we effectively "stretch" the $(\bar{u}, \bar{v})$ space, as illustrated in Figure 6 (right). The drawback is that errors in

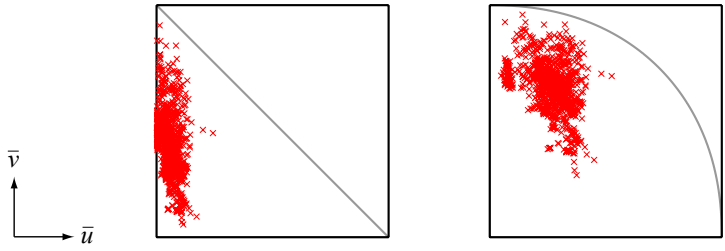


Figure 6: The chrominance information in the test image 'desk'. The left image shows all $(\bar{u},\bar{v})$ pairs in the image, and the right shows the same points after stretching the $(\bar{u},\bar{v})$-space.

the green component, $1-\bar{u}-\bar{v}$, get slightly smaller weights. We use a value $\alpha = 0.455$, which is inspired by the gamma-adjustment step typically included in tone-mapping operators. This works well in practice, but it should be noted that other transforms may perform better, and we have not performed extensive experiments with this.

Note that this trick of using a non-linear error function when minimizing the shape fitting error is only applied during the compression step. Hence, the shapes are stored exactly as before, and no modifications to the decompression hardware are needed. The drawback is slightly longer compression times. As seen in Table 1, the log[RGB] error decreases by 20% on average, compared to performing shape optimization using the original linear error function (Equation 11).

Image	**Non-linear**	**Linear**	**Improvement (%)**
bigFogMap	0.06	0.07	12
cathedral	0.17	0.21	18
memorial	0.13	0.18	27
room	0.08	0.09	11
desk	0.22	0.53	59
tubes	0.28	0.31	10

Table 1: log[RGB] error comparison with non-linear vs *linear error functions for the chrominance points.*

7.3 Luminance Precision

In this section, we discuss the necessary precision for encoding logarithmic luminance, as used in the color space defined in Section 3.1. Assume we work with values in the range $\bar{Y} \in [\bar{Y}_{\min}, \bar{Y}_{\max}]$, and uniformly quantize $\bar{Y}$ using k bits. Then,

the maximum absolute quantization error, $|\Delta\bar{Y}|$, is:

$$|\Delta\bar{Y}| = \frac{1}{2}\frac{\bar{Y}_{\max} - \bar{Y}_{\min}}{2^k}. \tag{16}$$

In linear luminance values, this quantization error translates to a maximum relative luminance error, $|\Delta Y|$, equal to:

$$|\Delta Y| = \max\left|\frac{2^{\bar{Y}\pm|\Delta\bar{Y}|}}{2^{\bar{Y}}} - 1\right| = 2^{|\Delta\bar{Y}|} - 1 \tag{17}$$

As an example, consider the dynamic range supported by the 16-bit `half` type, which is approximately $[2^{-16}, 2^{16}]$. The maximum relative error after quantization to $k = 6\ldots16$ bits are presented in Figure 7. It is widely accepted that a relative luminance error of about 1% is the smallest visually detectable difference [18]. Accordingly, a log-luminance precision of 10 bits (i.e., 1.09% relative error) should be sufficient. Mantiuk et al. [7] came to the same conclusion after a similar reasoning based on measured luminance threshold curves of the human visual system. Note that they use a perceptual quantization of luminance, so the results are not directly comparable.

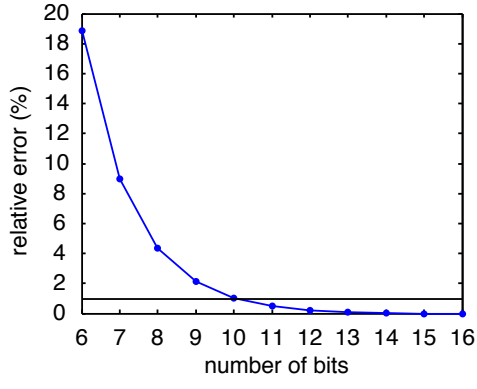


Figure 7: Maximum relative luminance error with uniform quantization of the log-luminance.

Munkberg et al. [9] encode luminance values by quantizing the minimum and maximum log-luminance over a 4×4 block to 8 bits precision. Then, a 4-bit per-pixel index is used to choose between 16 intermediate luminance levels, uniformly distributed between the block's minimum and maximum. In practice, this gives a variable luminance precision, where blocks with a small dynamic range will have the best precision.

Figure 8 shows a histogram over the difference between the maximum and minimum log-luminance over all blocks in our test images (Figure 11). With our luminance encoding, the histogram shows that we get 10 bits precision or better

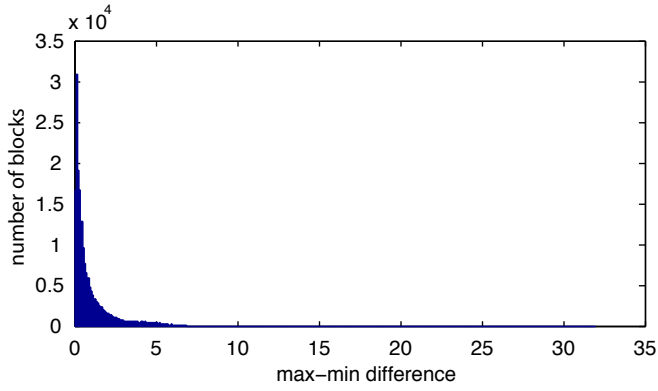


Figure 8: Histogram over the difference between the smallest and the largest log-luminance values over all 4×4 blocks in our suite of test images (Figure 11).

for 51.0% of the blocks (maximum difference 0.5), 84.7% falls within 8 bits precision, and 99.8% within 6 bits. However, as described by Munkberg et al. [9], we search in a small neighborhood around the end-points of the luminance range to minimize the error. In many cases, it is possible to find a combination that better fits the data, as illustrated in Figure 9.

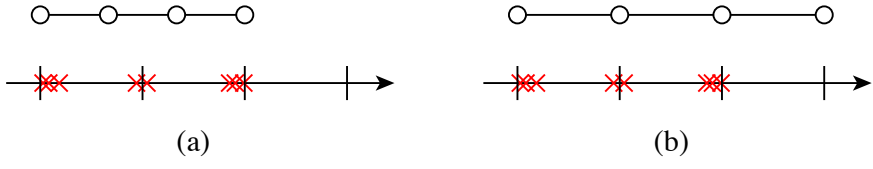


Figure 9: By searching around the end-points of the luminance range, we can often find a better match (b) for the luminance values in a block, rather than just picking the min and max as in (a). The example shows a quantization to 4 distinct levels for clarity, although our format supports 16 different levels.

8 Results

Here, we compare our combined compression mode (Section 5) against state-of-the-art HDR texture compression schemes. We also show that our algorithm works well on regular LDR textures.

8.1 Comparison with Other Approaches

We compare our algorithm with the recently published HDR texture compression formats [9, 15, 19]. We would like to point out that the approaches have different goals: Roimela et al.'s algorithm is very fast and designed for a simple hardware implementation, while Wang et al. present a format that can be directly implemented in DX9/10 without any hardware changes. Munkberg et al. focus on image quality, while keeping the decompression hardware simple. Our algorithm extends this by including a chrominance mode without subsampling.

A set of six test images was used, as shown in Figure 11. One image ('tubes') is artificial, while the others are natural images commonly used in the research community. Table 2 shows the log[RGB] error (Section 2) for the test images. Our combined mode shows slightly better or equal results for all images. Not surprisingly, the largest improvement is in the 'tubes' image, which contains sharp chrominance transitions. The combined mode handles this better by avoiding chrominance subsampling. Table 3 shows that the mPSNR error (Section 2) follows a similar pattern.

Table 4 shows the HDR-VDP (Section 2) error at 75% detection probability at an adaptation luminance manually adjusted per image so that it is close to 300 cd/m^2. We found that the implementation of the HDR-VDP we used (v1.6) had problems with true zeros in images, indicating errors even in totally black areas, e.g., in the 'tubes' image. The results for this image are therefore overly conservative. Our algorithm has one mode with lower luminance resolution, and as the encoder selects the best mode based on the log[RGB] error, the HDR-VDP scores are somewhat higher than with Munkberg et al.'s algorithm. This is because HDR-VDP only measures perceived luminance, so chrominance artifacts are not captured.

	Our	**Munkberg**	**Roimela**	**Wang**
image	8 bpp	8 bpp	8 bpp	16 bpp
bigFogMap	0.06	0.06	0.10	0.14
cathedral	0.17	0.20	0.33	0.36
memorial	0.13	0.14	0.26	0.69
room	0.08	0.09	0.23	0.71
desk	0.22	0.25	1.14	2.92
tubes	0.28	0.43	0.85	0.81

Table 2: Log[RGB] error (smaller is better).

A visual comparison is presented in Figure 11, where the compressed images are diagonally split, showing the squared log differences in the upper left triangle, and the compressed result in the lower right. As can be seen, the images obtained using Wang et al.'s algorithm (fourth column) often (1st, 2nd, 4th, and 5th row) have relatively large errors in the luminance channel. This can be seen in that the error images contain gray regions. In addition, there are also often larger chrominance errors than for the other algorithms (except for the last row, where the method

image	Our 8 bpp	Munkberg 8 bpp	Roimela 8 bpp	Wang 16 bpp
bigFogMap	51.9	51.7	47.1	46.3
cathedral	40.0	38.9	34.2	35.8
memorial	46.5	46.1	41.3	38.0
room	48.6	48.1	41.6	34.1
desk	40.3	39.7	31.1	21.3
tubes	35.7	32.2	26.6	29.1

Table 3: Multi-exposure PSNR (larger is better).

image	Our 8 bpp	Munkberg 8 bpp	Roimela 8 bpp	Wang 16 bpp
bigFogMap	0.00	0.00	0.01	7.18
cathedral	0.02	0.00	0.19	0.04
memorial	0.01	0.01	0.15	15.4
room	0.02	0.02	0.64	26.4
desk	0.03	0.00	2.58	4.34
tubes	1.59	0.66	3.35	2.00

Table 4: HDR-VDP error with 75% detection probability at an adaptation luminance of 300 cd/m^2 (smaller is better). Note that the HDR-VDP only measures luminance errors, while our format improves the chrominance precision. Hence the higher scores.

of Roimela et al. seems to produce the largest errors). Our algorithm and that of Roimela et al. reproduce the luminance quite accurately. However, Roimela et al.'s subsampling strategy for the chrominance gives a higher error than our algorithm in all test images. We believe it is clear from these images that our algorithm is more robust and accurate than previous methods.

8.2 LDR Measures

We have also compared the quality of our algorithm on a set of standard low dynamic range (RGB888) images. Here we have used a standard RGB to YUV color transform [11] instead of the HDR color spaces discussed earlier. All other parts of the algorithm were left unchanged. Figure 10 shows the results for a set of standard test images, and it is clear that our format performs significantly better than the industry standard (S3TC), though at a higher bit-rate.

9 Conclusion

The HDR texture compression algorithm by Wang et al. [19] focuses on reusing existing hardware for texture compression, and therefore arrive at an algorithm using 16 bits per pixel with rather low image quality. However, the algorithm can

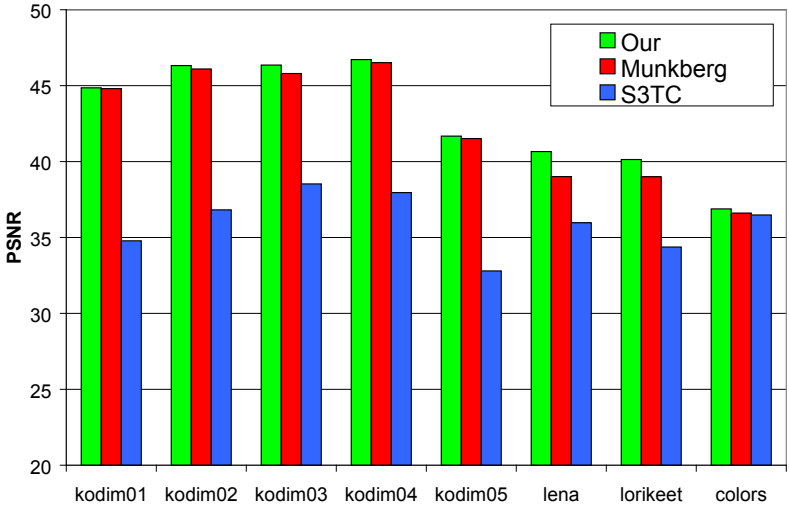


Figure 10: PSNR for a set of standard 24-bit LDR images. We compare our algorithm against S3TC and Munkberg et al.'s algorithm. Our new format gives up to 10 dB improvement over S3TC, but note that S3TC uses only 4 bpp (compared to 8 bpp for the other formats). The 'colors' image consists entirely of color gradients along directed lines, which is well captured by S3TC.

be used today on all DX9 hardware, which is a major advantage. The algorithm by Roimela et al. [15] is a proposal for new hardware, and their focus was to provide very simple decompression hardware, and still the image quality is rather high.

In contrast, our focus has been to increase the image quality as much as possible, as we think this is very important for content creators. We have introduced a new mode for blocks of pixels with difficult chrominance, and combined that with the algorithm of Munkberg et al. [9]. To make this usable, we provide an inexpensive texture filtering method. In addition, the details of our compressor using Procrustes analysis and k-means clustering have been described. We hope that all this information will be useful to many when developing new HDR TC schemes.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research and Vetenskapsrådet. Thanks to Rafal Mantiuk for sharing the HDR-VDP implementation, Kimmo Roimela and Xi Wang for sharing their respective codecs.

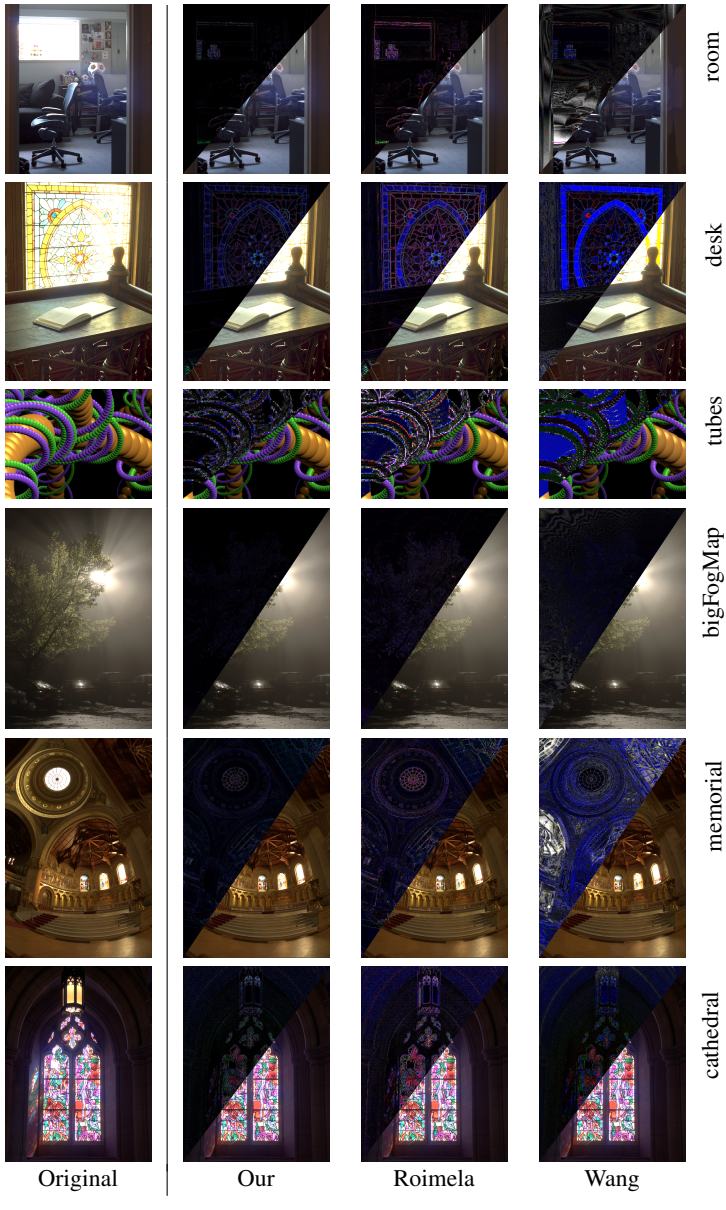


Figure 11: Image comparison. The upper left triangles in the compressed images show the squared log differences.

Bibliography

[1] M. M. Denn. *Optimization by Variational Methods*. McGraw-Hill, 1969.

[2] I.L. Dryden and K.V. Mardia. *Statistical Shape Analysis*. Wiley, 1998.

[3] S. Fenney. Texture Compression using Low-Frequency Signal Modulation. In *Graphics Hardware*, pages 84–91, 2003.

[4] Konstantine Iourcha, Krishna Nayak, and Zhou Hong. System and Method for Fixed-Rate Block-Based Image Compression with Inferred Pixel Values. US Patent 5,956,431, 1999.

[5] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220, 4598:671–680, 1983.

[6] Rafał Mantiuk, Scott Daly, Karol Myszkowski, and Hans-Peter Seidel. Predicting Visible Differences in High Dynamic Range Images – Model and its Calibration. In *Human Vision and Electronic Imaging X*, pages 204–214, 2005.

[7] Rafal Mantiuk, Grzegorz Krawczyk, Karol Myszkowski, and Hans-Peter Seidel. Perception-Motivated High Dynamic Range Video Encoding. *ACM Transactions on Graphics,*, 23(3):733–741, 2004.

[8] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[9] Jacob Munkberg, Petrik Clarberg, Jon Hasselgren, and Tomas Akenine-Möller. High Dynamic Range Texture Compression for Graphics Hardware. *ACM Transactions on Graphics,*, 25(3):698–706, 2006.

[10] John D. Owens. Streaming Architectures and Technology Trends. In *GPU Gems 2*, pages 457–470. Addison-Wesley, 2005.

[11] Charles Poynton. *Digital Video and HDTV Algorithms and Interfaces*. Morgan Kaufmann, 2003.

[12] Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec. *High Dynamic Range Imaging: Acquisition, Display and Image-Based Lighting*. Morgan Kaufmann, 2005.

[13] Randall C. Reininger and Jerry D. Gibson. Distrubutions of the Two-Dimensional DCT Coefficients for Images. *IEEE Transactions on Communications*, 31(6):835–839, 1983.

[14] K. Roimela, T. Aarnio, and J. Itäranta. Efficient High Dynamic Range Texture Compression. In *Proceedings of I3D*, pages 207–214, 2008.

[15] Kimmo Roimela, Tomi Aarnio, and Joonas Itäranta. High Dynamic Range Texture Compression. *ACM Transactions on Graphics,*, 25(3):707–712, 2006.

[16] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 1996.

[17] S. Smoot and L. Rowe. Study of DCT Coefficient Distributions. In *Proceedings of the SPIE Symposium on Electronic Imaging*, volume 2657, pages 403–441, 1996.

[18] B.A Wandell. *Foundations of Vision*. Sinauer Associates, 1995.

[19] Lvdi Wang, Xi Wang, Peter-Pike Sloan, Li-Yi Wei, Xin Tong, and Baining Guo. Rendering from Compressed High Dynamic Range Textures on Programmable Graphics Hardware. In *Proceedings of I3D*, pages 17–24, 2007.

[20] Ruifeng Xu, Sumanta N. Pattanaik, and Charles E. Hughes. High-Dynamic-Range Still-Image Encoding in JPEG 2000. *IEEE Computer Graphics and Applications,*, 25(6):57–64, 2005.

Paper III

High-Quality Normal Map Compression

Jacob Munkberg[†] Tomas Akenine-Möller[†] Jacob Ström[‡]

[†]Lund University [‡]Ericsson Research

ABSTRACT

Normal mapping is a widely used technique in real-time graphics, but so far little research has focused on compressing normal maps. Therefore, we present several simple techniques that improve the quality of ATI's 3Dc normal map compression algorithm. We use varying point distributions, rotation, and differential encoding. On average, this improves the peak-signal-to-noise-ratio by 3 dB, which is clearly visible in rendered images. Our algorithm also allows us to better handle slowly varying normals, which often occurs in real-world normal maps. We also describe the decoding process in detail.

Graphics Hardware 2006, pages 37–40

1 Introduction

Bump mapping [3] is a widespread technique which adds the illusion of detail to geometrical objects in an inexpensive way. More specifically, a texture, called a *bump map* or *normal map*, is used at each pixel to perturb the surface normal. A common approach to generate normal maps is to start with a high polygon count model and create a low complexity model using some geometrical simplification algorithm (see, for example, Cohen et al's work [4]). The "difference" between these two models is then "baked" into a normal map. For real-time rendering, the normal map is applied to the low complexity model, giving it a more detailed appearance. These techniques are heavily used in recent games.

A possible disadvantage is that the savings in transform and rendering due to the lower vertex count is translated into an increase in bandwidth usage of textures (normal maps). A traditional technique to alleviate this problem is lossy *texture compression* (TC), which was introduced in 1996 [2, 8, 9]. TC developed primarily for color can also be applied to normal maps [6], but the quality can be higher if specialized algorithms are developed. One such technique, called 3Dc, has been proposed by ATI [1].

However, little effort has been spent on developing new algorithms for *normal map compression*. One problem with 3Dc is that it cannot handle slowly varying normal maps well. This is illustrated in Figure 10. In this paper, we develop several variations and extensions of 3Dc that perform much better on average, and handle slowly varying data particularly well. We present visual proof showing that our normal mapping algorithms give higher quality renderings, and we also show that the peak-signal-to-noise ratio (PSNR) is improved.

2 Previous Work

The first example of normal compression in graphics that we know of is described in the context of geometry compression [5], i.e., it was not targeted towards normal map compression. Deering presents a method for compressing surface normals, arguing that about 100,000 normals distributed over the unit sphere would give sufficient quality. These normals can be represented by a single 17-bit index, and by exploring symmetries on the sphere, only a 1/48 of the sphere needs to be represented. A regular grid in the angular space of one such patch is used as sample distribution. Nearby normals are encoded differentially. With these techniques he manages to compress a normal to about 12 bits. However, the decompression step includes a number of trigonometric operations and is quite costly compared to the schemes described below.

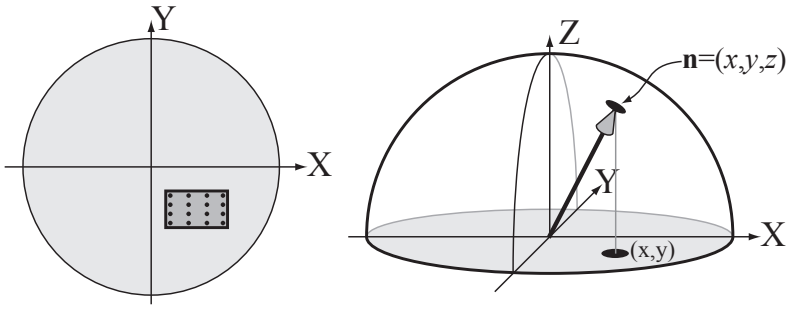


Figure 1: 3Dc selects a rectangle in the xy-plane (left), and places 8×8 *points uniformly over this rectangle (in this figure, only* 4×4 *points were placed to make the illustration clearer). These points can be seen as a "palette" of xy pairs, and each texel in a* 4×4 *tile can select one of these pairs. To the right, one such* (x,y)*-point has been used to generate a normal,* $\mathbf{n} = (x,y,z)$*. This is done by requiring that we use unit normals.*

2.1 3Dc Normal Compression

Next, we will review ATI's normal map compression scheme called *3Dc* [1]. As far as we know, this is the only format dedicated to this purpose alone. In the majority of cases today, bump mapping is performed in local tangent space, (X,Y,Z), of each rendering primitive (e.g. a triangle). Since the length of the normal is not of interest, 3Dc uses units normals, and hence it suffices to compress the x- and y-components. The third component is obtained through normalization:

$$z = \sqrt{1 - x^2 - y^2}, \tag{1}$$

and this computation can either be done in the pixel shader, or by special purpose hardware.

The x- and y-components are compressed independently using a variant of the S3TC/DXTC [7] format. A block of 4×4 texels (a.k.a. a tile) is compressed into 128 bits, i.e., at eight bits per pixel (bpp). The x-coordinates are encoded in the following way. Two eight-bit values, x_{start} and x_{stop}, representing an interval enclosing the x-values in the tile, are found. Each texel can select from eight different x-values: $x_k = x_{start} + k(x_{stop} - x_{start})/7$, $k = 0 \ldots 7$, which are thus spread uniformly over the interval. This requires three bits per texel. To encode the x-values of a tile, we need 2×8 bits for x_{start} and x_{stop}, and 16×3 bits for the per-pixel indices. This sums up to 64 bits. The y-components are encoded in the same way, and the total cost per tile is 128 bits. An illustration of 3Dc is shown in Figure 1.

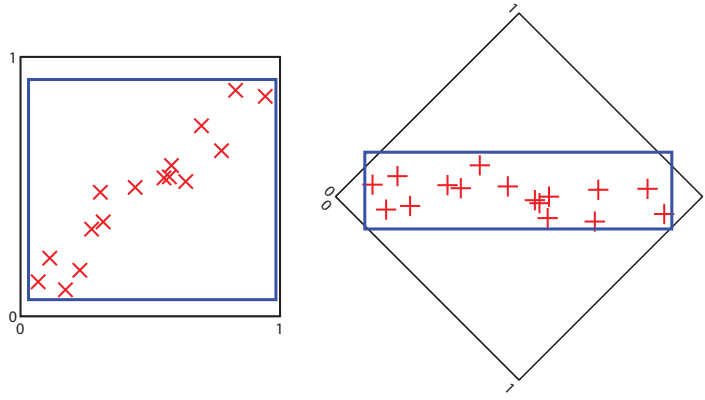


Figure 2: By rotating the coordinate frame, we can often find a much tighter bounding box. This will improve the encoding precision.

3 Improved Normal Compression

In the following three subsections, we present three simple general techniques for improving the quality of the 3Dc normal compression scheme. These are combined into a single compression format in Section 4, while keeping a bit budget of 8 bits per pixel (bpp). Compared to 3Dc, the extra cost is a more expensive decompression phase (Section 4.1).

First, however, we will explain how we can incorporate three new modes into 3Dc. It stems from the fact that swapping the values x_{start} and x_{stop} will produce exactly the same reconstruction levels $x_0 \ldots x_7$, albeit in the reversed order. Since these two representations are equivalent, it is possible to signal one extra bit, b: If $x_{start} < x_{stop}$, then $b \leftarrow 0$, else $b \leftarrow 1$. The same trick is used in DXT1 to signal whether a block is RGB or RGBA, and we call this trick the *ordering technique*. In 3Dc, the ordering technique can be used on both x and y, and hence two extra bits can be used.

3.1 Rotation Compression

When the major axis of a minimal box around the (x,y) points of a tile do not coincide with either the x- or the y-axis, the quality of 3Dc decreases. By rotating the coordinate frame, a much tighter fit can be obtained, and the extra storage cost is only an angle per block. Figure 2 illustrates this scenario. For example, using a single extra bit, one can select to use an angle in the set $\{0, \pi/4\}$, and two bits increase the set to $\{0, \pi/8, \pi/4, 3\pi/8\}$. Note that the standard 3Dc case is included, thus, this technique can only achieve results equivalent to or better than 3Dc. As seen in Figure 3, the peak-signal-to-noise-ratio (PSNR) improves with

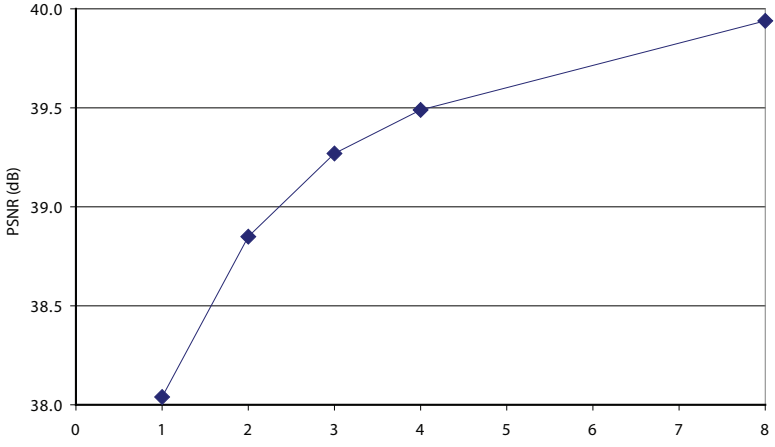


Figure 3: The average PSNR for a set of 20 normal maps as a function of the number of angles in the compressor. Angle count 1 represent no rotation, 2 represent the two angles $\{0, \frac{\pi}{4}\}$ *and generally, for an angle count a, the set of angles is* $\{0, \frac{\pi}{2a}, ..., \frac{\pi(a-1)}{2a}\}$.

more than a decibel on average, already with a set of three angles. Visual results are shown in Section 5.

3.2 Variable Point Distribution

Normally, the 3Dc technique places the sample points uniformly in a grid over the axis-aligned box defined by (x_{min}, y_{min}) and (x_{max}, y_{max}), where

$$x_{min} = \min(x_{start}, x_{stop}), \qquad x_{max} = \max(x_{start}, x_{stop}),$$

and ditto for y_{min} and y_{max}. However, other distributions may allow for better compression. A simple way of altering the sample distribution is to use different distributions depending on the aspect ratio of the box. For example, if the box is more than twice as wide as it is high, then it could be beneficial to use a 16×4-distribution rather than the standard 8×8-distribution. See Figure 4. No extra bits are needed to signal this, since the point distribution is automatically triggered by the aspect ratio, $a = \frac{y_{max} - y_{min}}{x_{max} - x_{min}}$, of the box. For 3Dc, the per-texel indices are encoded in six bits ($3+3$ bits for an (x,y) pair). However, if the aspect ratio triggers, say, the distribution 2×32, we simply move two bits, $3+3 \rightarrow 1+5$. It should be noted that this approach cannot guarantee higher quality in all cases. We have tested this technique on a set of 20 normal maps, with improved PSNR values on all maps. The bounds for selecting a distribution were chosen empirically and are presented in Table 1. The distributions 1×64 and 64×1 did not improve the quality, and are not used in our compressor.

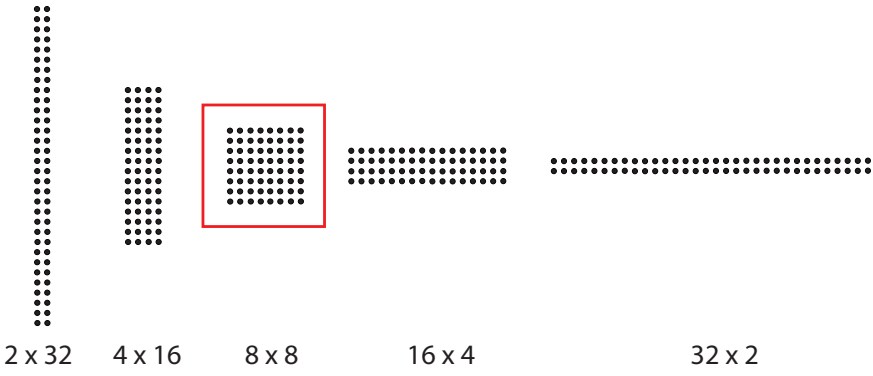


Figure 4: Different point distributions are triggered automatically dependent on the aspect ratio, $a = \frac{y_{max}-y_{min}}{x_{max}-x_{min}}$, of the bounding box.

aspect ratio ($a = \frac{y_{max}-y_{min}}{x_{max}-x_{min}}$)	distribution ($d_x \times d_y$)
$a < 1/8$	32×2
$1/8 \leq a < 1/2$	16×4
$1/2 \leq a \leq 2$	8×8
$2 < a \leq 8$	4×16
$a > 8$	2×32

Table 1: The bounding box aspect ratio automatically selects a point distribution.

3.3 Differential Encoding

One of the case where it is easy to detect compression artifacts is in areas that have a slight curvature, for example, on a car hood. The smoothness of the surface makes it easy for the viewer to predict what the image "should" look like, which is not as simple for a rough surface.

Compressing such slow varying normals with 3Dc poses two problems. First, the smallest representative interval is too wide. Since the quantized resolution is only eight bits, an interval of 1/255 of the range might be to coarse for representing nearly constant normals (see Figure 5a and b). Second, the smallest interval cannot be placed accurately enough, as the interval limits must coincide with the quantization steps. Thus, if values of a block are present on both sides of a quantized step (Figure 5c), the smallest interval covering all values will be at least twice the minimum interval (Figure 5d) . In this section, we will present a technique to make the precision higher in order to solve these two problems.

Our idea is to use the 32 bits that are normally used for storing x_{start}, x_{stop}, y_{start} and y_{stop} in a different way, with an encoding that is specialiced for representing small intervals accurately. However, we must be able to flag this mode of encoding,

*Figure 5: The x-axis is shown with quantized values marked with bold vertical lines. Left: **a** is the desired interval, but the smallest interval representable in 3Dc is **b**. Right: With values on both sides of a quantized value, the smallest interval in 3Dc that covers the desired interval **c** is **d**, twice the size of the smallest representable interval **b**.*

		x_{start}							
		0	1	2	3	4	5	6	7
x_{stop}	0	0	1	2	3	4	5	6	7
	1	■	9	10	11	12	13	14	15
	2	■	■	18	19	20	21	22	23
	3	■	■	■	27	28	29	30	31
	4	■	■	■	■	x	26	25	24
	5	■	■	■	■	■	x	17	16
	6	■	■	■	■	■	■	x	8
	7	■	■	■	■	■	■	■	x

Table 2: By mirroring the positions for number 8, 16, 17, 24, 25 and 26, it is possible to fit the numbers 0 through 31 without using positions where $x_{start} < x_{stop}$ (marked with black).

so some bits are irretrievably lost. Using a simple mapping technique described in the next paragraph, we can exploit 30 bits for a differential mode that handles slowly varying normals. In this mode, we use eleven bits each to encode x_{min} and y_{min} using 8.3 (eight bits for the integer part and three bits for the fractional part), and we spend four bits each on two delta values, Δ_x and Δ_y, using 2.2 bits. x_{max} is calculated as $x_{max} = x_{min} + \Delta_x$, and ditto for y_{max}. Due to the differential coding, we call this mode the *differential* mode, and it addresses both problems identified above: the smallest representable interval is now four times smaller, and since the precision of the location of the interval (3 fractional bits) is twice that of the smallest length (2 fractional bits), we can handle values on both sides of a border as in Figure 5c without doubling the interval.

In the following, we will present a general method useful when exploiting the ordering technique (see beginning of Section 3). Assume that we have detected a special mode signaled by $x_{start} \geq x_{stop}$. Unfortunately, we cannot set the bits of x_{start} and x_{stop} arbitrarily, since x_{stop} must be less than or equal to x_{start}. We thus want to solve the problem of exploiting a maximum number of the sixteen bits occupied by x_{start} and x_{stop}, while preserving $x_{start} \geq x_{stop}$. This can be solved by a simple mapping, illustrated in Table 2, where x_{start} and x_{stop} are 3-bit values instead of 8-bit values for simplicity. Here, we have entered the numbers 0 through 31 into the table, while avoiding the black boxes where $x_{start} < x_{stop}$. The numbers are entered row-by-row, except for the numbers which would have fallen in the

forbidden positions, namely numbers 8, 16, 17, 24, 25 and 26. The positions for these numbers are therefore mirrored both in the vertical and horizontal direction relative to the center of the table. As can be seen, we have stored 32 numbers, and we can therefore extract five bits. This is the maximum number of bits we can obtain since roughly half the values are marked with black.

Decoding this 5-bit number is especially simple for the upper half (rows 0 through 3) using

$$value = (x_{stop} << 3) \text{ OR } x_{start},$$

where $<<$ represents a left shift and OR is the bit-wise logical OR operator. For the lower half (rows 4 through 7), we have to mirror x_{start} and x_{stop} first to $(7 - x_{start})$ and $(7 - x_{stop})$, which is the same as inverting their bits, and we can use

$$value = (\text{NOT}(x_{stop}) << 3) \text{ OR } \text{NOT}(x_{start}),$$

where $\text{NOT}(\cdot)$ denotes bit-wise inversion. For eight bit x-values, we shift with 8 instead of 3, and we can store 15 bits in *value*. Encoding is straightforward—we use the lower part of *value* for x_{start} and the upper part for x_{stop}, and invert both if $x_{stop} > x_{start}$ according to the pseudocode below:

```
xstart = value AND 0xff
xstop = (value >> 8) AND 0x7f
if xstop > xstart
  xstart = NOT(xstart)
  xstop = NOT(xstop)
end
```

where NOT operates on all eight bits.

4 Proposed Scheme

In this section, we will combine the three techniques described above into a format that fits in an 8 bpp budget. The foundation for our combined mode is 3Dc, but we exploit redundancy in its encoding to allow for more modes. Next, we will describe how these two extra bits can be used to improve the quality of 3Dc substantially.

We allow two rotations and limit the differential mode to tiles where both the x- and the y-components can be encoded differentially. Altogether, we have four different modes: I) the standard 3Dc mode, II) a rotation with 30 degrees, III) a rotation with 60 degrees, and IV) a differential mode, encoded with $8.3 + 2.2$ bits. As seen in Figure 3, using three angles gives a significant improvement in quality. It would be possible to add yet another angle, but that mode is more wisely spent on the differential mode in terms of PSNR. The variable point distribution is applied to all modes except the differential one where it did not increase quality. Table 4 shows the quality contribution that each technique adds on a test series. The usage of each mode is further illustrated in Figure 6, showing how often the different

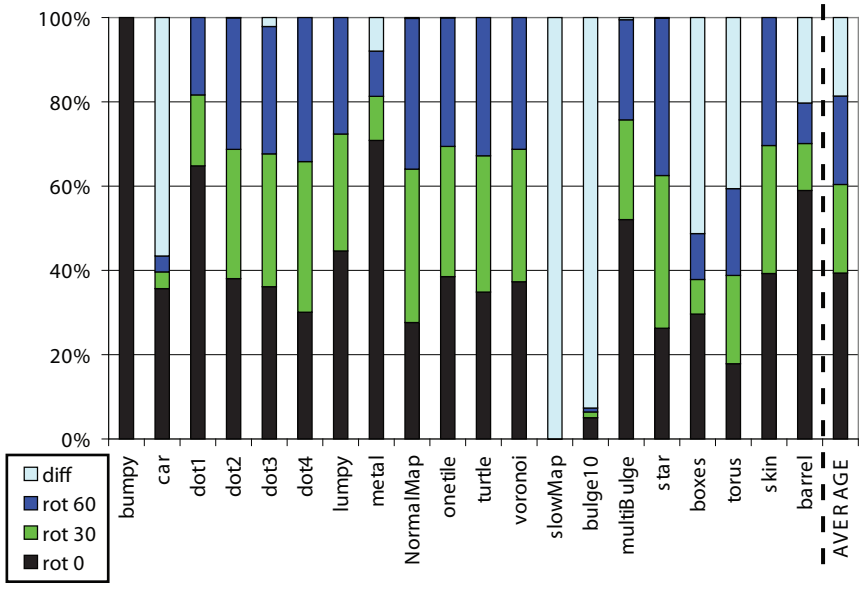


Figure 6: The frequencies of the different algorithms for the images used in the test.

modes are used for each test image. All modes are used quite frequently, which indicates a balanced algorithm.

Note that mode I differs slightly from 3Dc in that it uses variable point distribution. Alternatively, it is possible to avoid using variable point distribution in mode I.

mode	X	Y	bits	vpd
I: rot 0°	$x_{start} < x_{stop}$	$y_{start} < y_{stop}$	8+8	yes
II: rot 30°	$x_{start} \geq x_{stop}$	$y_{start} < y_{stop}$	8+8	yes
III: rot 60°	$x_{start} < x_{stop}$	$y_{start} \geq y_{stop}$	8+8	yes
IV: diff	$x_{start} \geq x_{stop}$	$y_{start} \geq y_{stop}$	8.3+2.2	no

Table 3: The encoding modes for the combined normal compressor. vpd indicates "variable point distribution."

mode	$\overline{PSNR}$ (dB)
3Dc	36.4
3Dc + Point Distr.	37.5
3Dc + Point Distr. + Rot	38.8
3Dc + Point Distr. + Rot + Diff	39.4

Table 4: The average PSNR for the normal maps presented in Figure 8.

This would mean that existing 3Dc hardware designs could be reused to decode this mode. Maybe more important, it would allow existing 3Dc textures to be transcoded to our new format without loss, by swapping x_{start} and x_{stop} if $x_{start} > x_{stop}$ (and performing bit-wise NOT on the per-pixel indices to reflect the inverted ordering). However, this backward compatibility would come at a cost: On the test images of Section 5, the average PSNR for this alternative solution is about 1.3 dB lower than the proposed scheme.

4.1 Decoding

The decoding of a block is performed as follows:

1. First, x_{start}, x_{stop}, y_{start} and y_{stop} are tested to see which mode the block belongs to, according to Table 3. For instance, if $x_{start} < x_{stop}$ and $y_{start} \geq y_{stop}$, then mode III is selected.

2. The next step is to calculate x_{min} and x_{max}. For modes I through III, this is simply done using $x_{min} = \min(x_{start}, x_{stop})$ and $x_{max} = \max(x_{start}, x_{stop})$, and likewise for y_{min} and y_{max}. All resulting numbers will be between 0 and 255. For mode IV, the 15-bit *value* is first calculated from x_{start} and x_{stop} as described in Section 3.3. Then, the first eleven bits of *value* are used to decode x_{min} in format 8.3, i.e., with eight bits for the integer part and three for the fractional part, resulting in a number between 0 and 255.875. The last four bits of *value* are decoded as an offset, Δ_x, in fixed-point format 2.2, resulting in a number between 0 and 2.75. x_{max} is finally calculated as $x_{min} + \Delta_x$. Similar computations are performed for y_{min} and y_{max}.

3. The aspect ratio $a = \frac{y_{max} - y_{min}}{x_{max} - x_{min}}$ is computed, and a point distribution is selected according to Table 1. Denote the distribution $d_x \times d_y$. For mode IV, the distribution is always 8×8.

4. The reconstruction levels are calculated using $x_k = x_{min} + \frac{k}{d_x - 1}(x_{max} - x_{min})$, $k = 0, \ldots, d_x - 1$, and likewise for y_k.

5. The pixel indices are now used to determine which reconstruction level to use. For instance, a value of 010_{bin} selects reconstruction level x_2 for x. The y-value is obtained analogously.

6. For modes II and III, we will also rotate the coordinates using $\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{M} \begin{pmatrix} x \\ y \end{pmatrix}$, where $\mathbf{M} = \begin{pmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{pmatrix}$ is a rotation matrix and ϕ is $-\pi/6$ or $-\pi/3$. See Section 4.2 for an efficient implementation. For modes I and IV, we just use $x' = x$ and $y' = y$.

7. Division by 255, and remapping to $[-1, 1]$ follows: $x'' = 2x'/255 - 1$ and $y'' = 2y'/255 - 1$. In the differential mode, clamping the values to the interval $[-1, 1]$ can also be necessary.

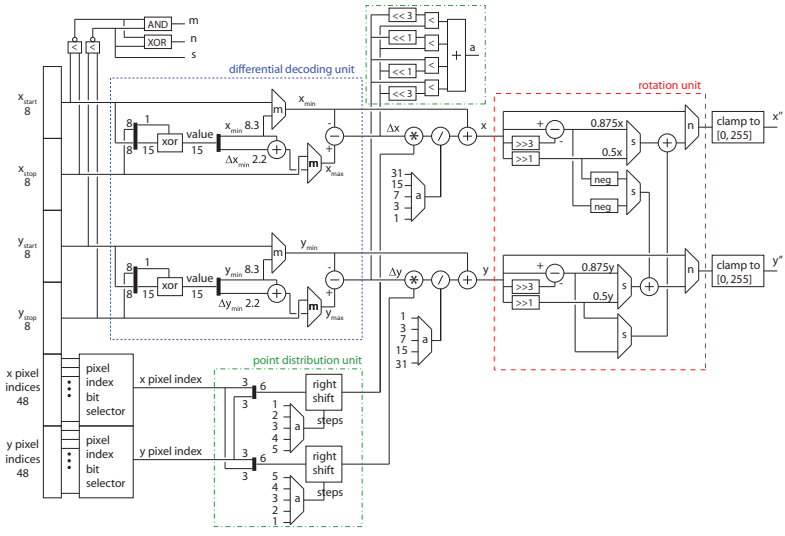


Figure 7: A hardware decompressor unit for our normal map compression algorithm. To the left, 128 bits of data are shown, and these are used to decode one of the 16 normals in a 4×4 tile. As can be seen, our three techniques have been clearly marked. The remaining parts is basically 3Dc (except that 3Dc only divides by 7).

8. Finally, the z coordinate is calculated as $z'' = \sqrt{1 - x''^2 - y''^2}$. The decompressed normal for the pixel is (x'', y'', z'').

The last two steps can be performed in the pixel shader.

4.2 Efficient Rotation

In this section, we suggest a hardware-friendly rotation. For modes II and III of our algorithm, the decompressor needs to rotate a two-dimensional point by -30 and -60 degrees. In the following, we develop an inexpensive, approximate rotation for -30°. The case with -60° uses the same constants, but at different locations in the matrices, so this is omitted from our description. The matrix for rotating -30 degrees is:

$$\mathbf{M} = \begin{pmatrix} \cos(-\pi/6) & -\sin(-\pi/6) \\ \sin(-\pi/6) & \cos(-\pi/6) \end{pmatrix} = \begin{pmatrix} 0.86602... & 0.5 \\ -0.5 & 0.86602... \end{pmatrix}. \quad (2)$$

The 0.5-terms above are not expensive to implement, but multiplication by $\sqrt{3}/2 \approx 0.86602$ is. To that end, we suggest that the hardware-friendly matrix $\tilde{\mathbf{M}}$ is used instead:

$$\mathbf{M} \approx \tilde{\mathbf{M}} = \begin{pmatrix} 1 - \frac{1}{8} & 0.5 \\ -0.5 & 1 - \frac{1}{8} \end{pmatrix} = \begin{pmatrix} 0.875 & 0.5 \\ -0.5 & 0.875 \end{pmatrix}, \quad (3)$$

where multiplication by 0.875 can be implemented as a shift by three and a subtraction. Note that $\tilde{\mathbf{M}}$ is *not* an orthogonal matrix, i.e., $\tilde{\mathbf{M}}\tilde{\mathbf{M}}^{\mathrm{T}} \neq \mathbf{I}$. Therefore, we emphasize that we cannot use $\mathbf{M}^{\mathrm{T}}$ during compression, because it also holds that $\tilde{\mathbf{M}}\mathbf{M}^{\mathrm{T}} \neq \mathbf{I}$. Instead, we must use the inverse of $\tilde{\mathbf{M}}$ during compression:

$$\tilde{\mathbf{M}}^{-1} = \frac{64}{65}\begin{pmatrix} 0.875 & -0.5 \\ 0.5 & 0.875 \end{pmatrix} \approx \begin{pmatrix} 0.8615... & -0.4923... \\ 0.4923... & 0.8615... \end{pmatrix}. \tag{4}$$

If $\tilde{\mathbf{M}}^{-1}$ is used to transform a rectangle, the result will be different from the rectangle obtained by using $\mathbf{M}^{-1} = \mathbf{M}^{\mathrm{T}}$. In fact, when using $\tilde{\mathbf{M}}^{-1}$, the rectangle will get a slight skew due to the fact that the transform is not orthogonal. However, the average PSNR for all our test images was only reduced by 0.03 dB on average, which is not significant.

See Figure 7 for a possible hardware implementation.

5 Results

To evaluate the visual quality of our compressor, we have tested several normal maps, taken from the set in Figure 8, in a real-time shader development application, in order to mimic a typical user scenario. We have also rendered images using a high-end renderer, with anisotropic mipmap filtering, HDR environment mapping and screen space anti-aliasing. When compressing with 3Dc, we perform exhaustive search for the base values in the x- and y-direction separately, to ensure that our 3Dc compressor is near-optimal. A full exhaustive search over x and y simultaneously was too costly.

In Figure 10, we show visual results obtained using a normal map with slowly varying normals. The pixel shader implemented simple environment mapping in order to better show the quality. As can be seen, our technique provides superior results compared to ATI's 3Dc technique. For this particular map, we have observed an increase of 10 dB in PSNR compared to 3Dc.

Figure 11 illustrates a test with a typical game normal map [6] with sharp edges. Our algorithm handles many difficult tiles better due to the flexibility offered by the extra rotation and variable point distribution. We rendered the images in Figure 10 and 11 using an NVIDIA GeForce FX 6800 graphics card. In the tests, we use RGB*fp16* textures, which are supported by the GPU.

Another visual test is shown in Figure 12, which was rendered using a high-quality offline renderer.

In addition to obtaining visual results, we also used the *mean square error* (MSE), which is computed as a summation over all normals in the image:

$$MSE = \frac{1}{w \times h}\sum(\hat{x}-x)^2 + (\hat{y}-y)^2 + (\hat{z}-z)^2, \tag{5}$$

where w and h are the width and the height of the image, $x \in [-1,1]$ is the x-component of the uncompressed normal and $\hat{x} \in [-1,1]$ is the corresponding com-

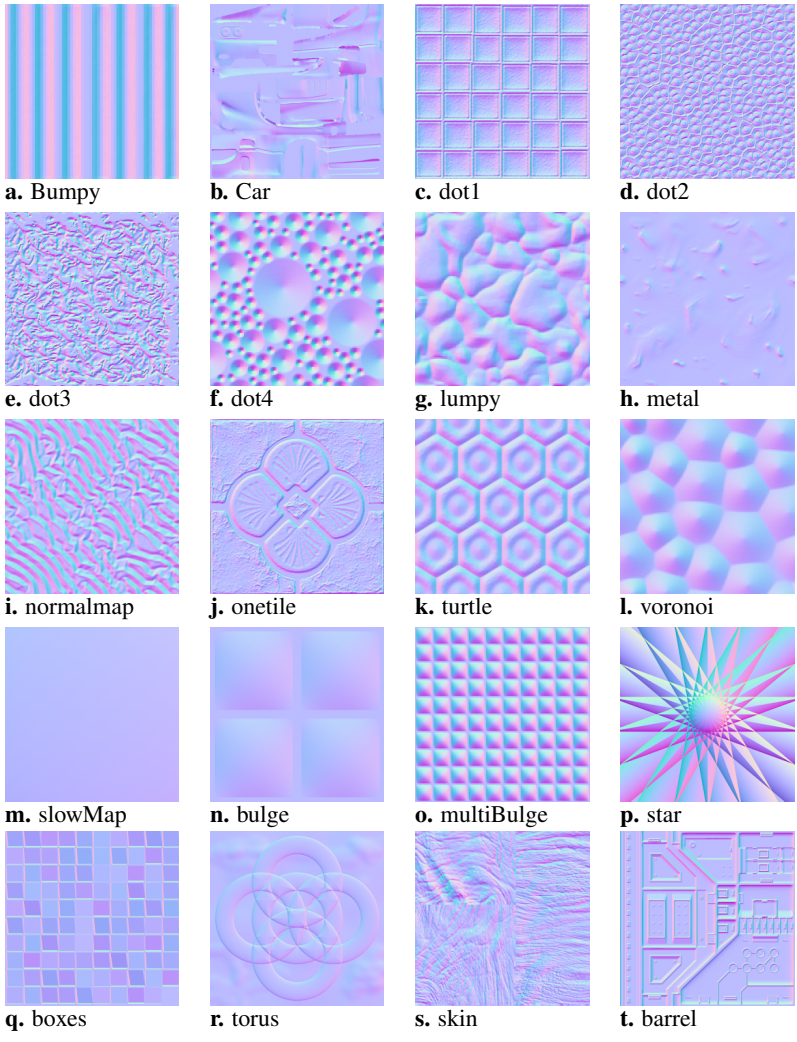


Figure 8: The set of normal maps used for evaluating our compression algorithm. ***m****,* ***n****,* ***o****,* ***p****,* ***q****, and* ***r*** *are 32 bit/channel maps, all other maps are 8 bit/channel.*

pressed x-component, and similar for y and z. For normal values, we use the *Peak Signal to Noise Ratio* (PSNR):

$$PSNR = 10\log_{10}\left(\frac{1}{MSE}\right), \tag{6}$$

where the nominator is one, since the *peak signal* for a normal of unit length will always be equal to one, by construction. PSNR values for all images tested, for 3Dc and our combined algorithm are presented in Figure 9, with improved values

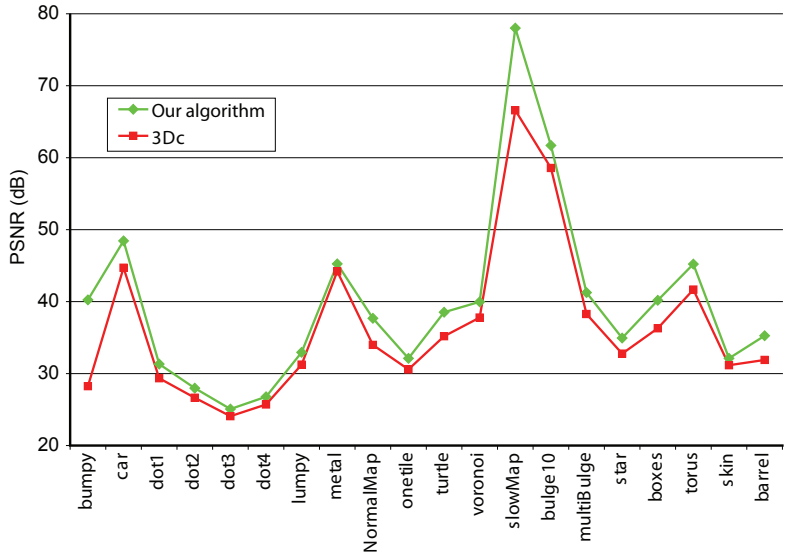


Figure 9: This chart shows the PSNR values for the images in Figure 8 for 3Dc and our algorithm. Our algorithm is the combined algorithm, using a standard 3Dc mode, rotations (30 and 60 degrees), a differential mode and variable point distribution.

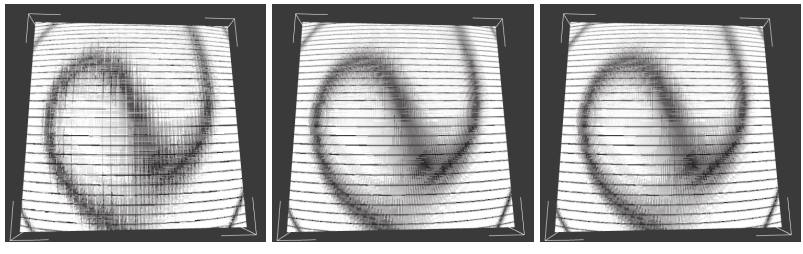


*Figure 10: A grid cube-map environment is used for these images. The normal map is a very slowly varying map (**m**) from Figure 8. Left: normal map compressed with ATI's 3Dc technique. Middle: rendered using original normal map. Right: normal map compressed with our algorithm.*

on all maps. The average improvement is about 3 dB.[1] We see large differences on slowly varying maps and maps with sharp egdes.

[1] As discussed in Paper IV, PSNR values should not be directly averaged, since PSNR is a non-linear operator. Averaging the MSE and computing the PSNR of the result avoids this pitfall, and the average improvement is then about 2 dB.

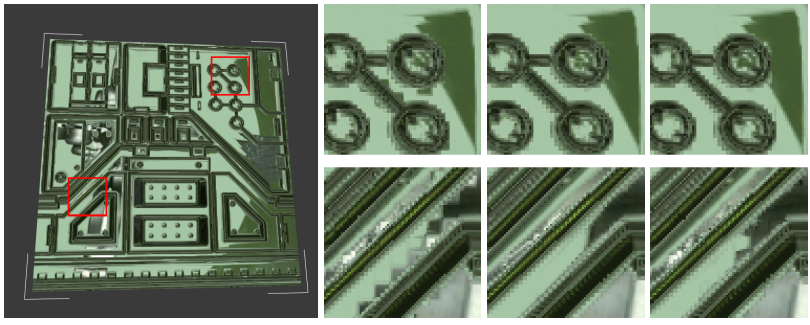


*Figure 11: A typical game normal map (**t**), rendered in a real-time shader development application, with a cube reflection map. Left: normal map compressed with ATI's 3Dc technique. Middle: rendered using original normal map. Right: normal map compressed with our technique.*

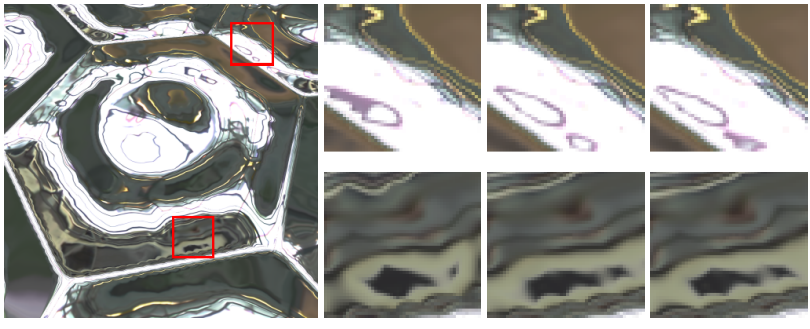


*Figure 12: The normal map (**k**), rendered in a high-end off-line renderer, with HDR environment mapping, texture filtering and advanced anti-aliasing. Left: 3Dc. Middle: uncompressed map. Right: our algorithm. As can be seen in the images, 3Dc shows more "wobbling" artifacts, and some features even disappear. Our new algorithm shows higher quality, even though some artifacts remain.*

6 Conclusions

We have designed three new techniques which can be used in conjunction to the 3Dc normal compression format. As shown in our paper, the combination of these handles many of 3Dc's weaknesses much better. Our techniques are combined into a scheme that still fits into a bit budget of 8 bpp and requires only small additions to a hardware decompressor. The new format is more flexible, with 3Dc as a subset, and we have obtained better results on all normal maps tested, both visually and in the PSNR error measure. For a series of 20 normal maps, the average PSNR increased with 3 dB.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research and Vetenskapsrådet.

Bibliography

[1] ATI. Radeon X800: 3Dc White Paper. Technical report, 2005.

[2] A.C. Beers, M. Agrawala, and Navin Chadda. Rendering from Compressed Textures. In *Proceedings of SIGGRAPH*, pages 373–378, 1996.

[3] Jim Blinn. Simulation of Wrinkled Surfaces. In *Proceedings of SIGGRAPH*, pages 286–292, 1978.

[4] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving Simplification. In *Proceedings of SIGGRAPH*, pages 115–122. ACM Press, 1998.

[5] Michael Deering. Geometry Compression. In *Proceedings of SIGGRAPH*, pages 13–20. ACM Press, 1995.

[6] Simon Green. Bump Map Compression. Technical report, NVIDIA, 2004.

[7] Konstantine Iourcha, Krishna Nayak, and Zhou Hong. System and Method for Fixed-Rate Block-based Image Compression with Inferred Pixels Values. In *US Patent 5,956,431*, 1999.

[8] Günter Knittel, Andreas G. Schilling, Anders Kugler, and Wolfgang Straßer. Hardware for Superior Texture Performance. *Computers & Graphics*, 20(4):475–481, 1996.

[9] Jay Torborg and Jim Kajiya. Talisman: Commodity Real-time 3D Graphics for the PC. In *Proceedings of SIGGRAPH*, pages 353–364, 1996.

Paper IV

Tight Frame Normal Map Compression

Jacob Munkberg[†] Ola Olsson[†] Jacob Ström[‡] Tomas Akenine-Möller[†]

[†]Lund University [‡]Ericsson Research

ABSTRACT

We present a new powerful and flexible fixed-rate normal map compression algorithm with higher quality than existing schemes on a test suite of normal maps. Our algorithm encodes a tight box with uniform normals inside the box, and in addition, a special mode is introduced for handling slowly varying normals. We also discuss several error measures needed to understand the qualities of different algorithms. We believe the high quality of our technique makes it a potential candidate for inclusion in OpenGL ES.

Graphics Hardware 2007, pages 37–40

1 Introduction

Normal maps, also called bump maps [3], allow for significant geometry savings, while preserving the illusion of geometric detail. Therefore, they are very popular in the latest generation of games. Texture bandwidth is a limiting factor, and to allow heavy use of normal maps in a real-time engine, there is a need of a compact representation of these textures. The focus of this article is twofold. First, we will discuss error measures for evaluating the quality of normal maps, and second, we will present a new compression algorithm. We argue that it is important to study not only the PSNR of the resulting maps, but also the *maximum pixel error*, and the *error distribution* over the images alongside with rendered results of the maps in use. Governed by our error measures, we present a new high quality compression algorithm, suitable for hardware implementation. Our technique supports very fast decompression, and robust behavior for a large range of input data.

2 Previous Work

A number of algorithms have been suggested for normal map compression. Most of these are fixed rate algorithms, which allows for fast random access without index tables, palettes or traversal trees.

Standard color texture compression techniques are not well suited for normal maps, which often contain many sharp features. To the best of our knowledge, Deering was the first to present compression of normals [4]. By using symmetries on the sphere, and encoding the "sextants" of the octants, each normal could be stored in 12 bits. Note that this work was targeted for geometry compression. Fenney and Butler [5] also encode by the octants, but select one of four octant-pairs, each parameterized with 7+7 bits. Each normal uses 16 bits.

The 3Dc format [2] is a dedicated normal map compression technique, which compresses blocks of 4×4 pixels. The 16 (unit length) normals in a block are projected onto the unit circle, and the axis-aligned bounding box of the projected values is quantized into an 8×8 grid, giving 64 positions to choose from inside the box. Four values are encoded to determine the size of the box, and $3 + 3$ bits are encoded per normal in the block to determine which point in the grid to select. This results in a total of 128 bits per block of 16 pixels, or 8 bits per pixel. By exploiting unused encoding combinations, and using them as additional compression modes, an enhanced 3Dc (here abbreviated e3Dc) algorithm was defined [6]. This algorithm handles very slowly varying normal maps (e.g., car hoods), rotated frames and more uniform reconstruction point distributions. We have borrowed techniques for better point distributions and bit extraction from this work.

Normal map encodings with adaptive bit rates [9, 12] achieve better compression rates than fixed-rate approaches with comparable quality, but rely on complex addressing for decompression along with more memory accesses to index tables, which can make a hardware implementation significantly more complex. Vector

quantization allows for more compact normal map compression and achieves impressive quality and compression rates [10]. However, the approach is limited to 8-bit normals, which is shown to be insufficient for slowly varying normal maps.

An error analysis for normal maps based on unity condition [11] discussed the impact of the popular elimination of the z-component while compressing normal maps. An interesting conclusion is that as long as the normals have small x, y and small errors in those components, the z-error will be even smaller.

3 Error Analysis

As normal maps are not viewed directly, but rather used in shaders to define the local normal vector, standard image quality metrics are not directly applicable. It can be argued that the *mean square error* (MSE), is a good measure, as it gives an (averaged) error that indicates the quality of the normal map. However, it does not tell us whether there is a constant small error over all pixels or a small set of pixels with large errors. An excellent discussion of the limitations of the MSE is described in Wang et. al's paper about *structural similarity* [8], where different distortions are added to an image, all with equal MSE. A smooth normal map with a few isolated divergent normals will often look unacceptable as the divergent normals will give rise to cracks in the smooth surface. Therefore, we also use the *max error*, and histograms of the angle error (defined below) per image together with MSE values, to ensure that the algorithms behave robustly.

MSE is computed as a summation over all normals in the image:

$$MSE = \frac{1}{w \times h} \sum (\hat{x} - x)^2 + (\hat{y} - y)^2 + (\hat{z} - z)^2, \tag{1}$$

where w and h are the width and the height of the image, $x \in [-1,1]$ is the x-component of the uncompressed normal and $\hat{x} \in [-1,1]$ is the corresponding compressed x-component, and similar for y and z. For normals, we use the *Peak Signal to Noise Ratio* (PSNR):

$$PSNR = 10 \log_{10} \left(\frac{1}{MSE} \right), \tag{2}$$

where the nominator is one, since the *peak signal* for a normal of unit length will always be equal to one by construction.

There are mainly three components which will be affected by the precision of the normal in real-time graphics: diffuse shading, specular shading, and specular reflection. The errors in a rendered image due to the diffuse and specular shading are relatively small compared to that of the specular reflection. Even a small angular error in a normal may result in a different texture access in the environment map. Therefore, it is important to look at the direct angle difference between the compressed and original normal, as well as studying bump mapped images with environment mapping.

We propose using the angular deviation [1], denoted E_{ad}, defined as:

$$E_{ad} = \arccos\left(\mathbf{n}_o \cdot \mathbf{n}_c\right), \tag{3}$$

which measures the difference in angle between the uncompressed normal ($\mathbf{n}_o$) and the compressed one ($\mathbf{n}_c$).

In addition, we will show false color images of the errors in the normals maps, and also render images with environment mapped and bump mapped surfaces. For these, we will compute the structural similarity [8] quality measure.

4 New Algorithm

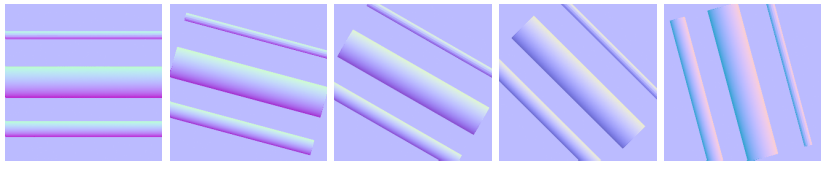


Figure 1: An example with strong directed features. PSNR values are listed for 3Dc / e3Dc / Tight Frame respectively.

Let us start with a simple motivating example. Imagine we have a normal map, as in Figure 1, consisting mainly of parallel lines. If the lines are axis-aligned, 3Dc will handle this example pretty well, as a tight axis-aligned bounding box (AABB) would capture the details. If the lines are rotated, however, the projected values will be more spread out. Thus, the AABB will inevitably grow, resulting in less accurate encoding. The enhanced 3Dc (e3Dc) algorithm handles this by including a small set of angles, thus making the encoder less sensitive to directed features. However, we would like generalize this. The artist should not need to try out the "best" initial position before baking the texture for best compressed quality. We also note that texture atlases contain many small maps, which are packed into a single texture. This is often an automatic process, and can create arbitrarily oriented small texture pieces. This is another case where a *rotation-invariant* normal map compression scheme would be desired.

4.1 Tight Frame Encoding

Here, we describe our rotation-invariant normal map compression algorithm. Instead of creating a bounded interval for our x- and y-values, we express a bounding box in a new coordinate frame using only two points, $\mathbf{p} = (p_x, p_y)$ & $\mathbf{q} = (q_x, q_y)$, and the aspect ratio, $a = \frac{height}{width}$, where $width$ is $||\mathbf{p} - \mathbf{q}||$, and $height$ is the height of

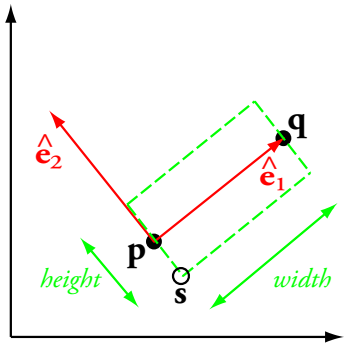


Figure 2: The coordinate system of our tight frame (TF) coding algorithm.

the rotated box. Figure 2 shows this setup. The two axes of this coordinate frame are simply $\hat{\mathbf{e}}_1 = \mathbf{q} - \mathbf{p}$, and $\hat{\mathbf{e}}_2 = (-\hat{e}_{1_y}, \hat{e}_{1_x})$. The lower left point in this frame is $\mathbf{s} = \mathbf{p} - 0.5a\hat{\mathbf{e}}_2$. It should be noted that a similar setup has been discussed in HDR texture compression [7].

Once we have defined this oriented bounding box (OBB), we distribute points uniformly in the box, using the aspect ratio to select the number of divisions along the two axes. For example, in the case of a very wide OBB, it makes more sense to use more points along the widest axis. This *variable point distribution* (VPD) [6] becomes more powerful in our algorithm, as it is easier to find a compact OBB than a compact AABB (3Dc), or fix-rotation AABB (e3DC). See Figure 3 for an illustration of the benefits of VPD.

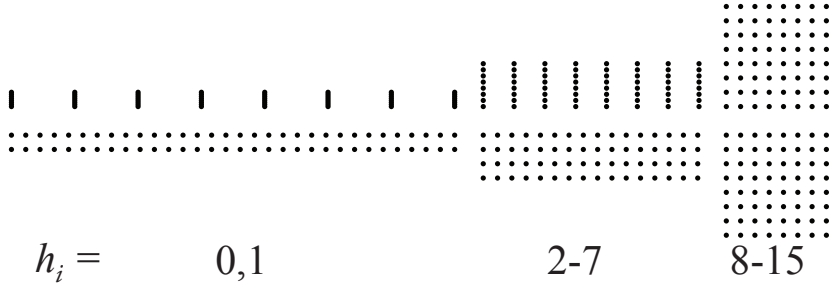


Figure 3: Without (top) and with (bottom) variable point distribution (VPD). By adapting the point distribution to the aspect ratio of the bounding box, the area is more evenly sampled. h_i is a four bit number, as described below.

The flexibility of the OBB combined with the redistribution of sample points (VPD) makes for a simple, yet powerful algorithm which gives high quality compression when there is correlation to exploit between the *x*- and *y*-channels. Hereafter, this technique is called *tight frame* (TF) coding. The target of our algorithm

is 8 bits per pixel (bpp), i.e., 128 bits for 4×4 pixels. Similar to 3Dc and e3Dc, we use six bpp for indices. This leaves 32 bits for encoding the bounding box. The information needed to reconstruct the bounding box comprises the two points **p** & **q** and the aspect ratio a. To stay in the bit budget of 32 bits, **p** and **q** are quantized to $7+7$ bits per point, leaving four bits to a. Note that the points **p** and **q** can always be oriented so that a is a number between zero and one. Being able to encode $a = 1.0$ means that there are two ways of expressing the same bounding box (rotate the first box 90 degrees). In order to avoid this redundancy, we use a maximum value of a which is somewhat smaller than 1.0. In addition, $a = 0.0$ is not particularly useful. For these reasons, we use the following reconstruction levels: $a = \frac{1}{32} + h_i \frac{1}{16}$, where h_i is the 4-bit number stored. Since a increases in steps of $\frac{1}{16}$, the height can be inexpensively calculated from the width using shifts, additions, and integer multiplication with h_i only.

4.2 Differential Mode

Similarly to e3Dc, we include a special mode for handling slowly varying normals inside a block. This is mode is triggered when $p_x \geq q_x$ and $p_y \geq q_y$ [6], and the same trick is used to recover the payload bits for this mode. However, our encoding is slightly different. To increase the accuracy of the bounding box positions (**p** and **q**) of this mode, we encode normals inside a (non-rotated) *square*. We encode the lower-left corner of the square using 2×11 bits, and the length of the square side is coded using 8 bits. Inside the square, we use 8×8 uniformly distributed points, which costs $3+3$ index bits per pixel. All in all, this mode costs $22+8+16 \times 6 = 126$ bits per block. Since we target slowly varying normals with this mode, we limit the square's side length for added precision. As an example, we can use a maximum length of $1/4$. This implies that the minimum side of the square is $\frac{1}{4 \times 2^8} = \frac{1}{1024}$. If we select a smaller maximum size, say $1/32$, we get square sizes in $[\frac{1}{32768}, \frac{1}{32}]$. For the test series used in this paper, a max length of $1/4$ worked well. For comparison, e3Dc uses a differential mode with 2×11 bits for positions and 2×4 bits for a differential vector. This implies a length of the differential vector in the smaller interval $[\frac{1}{512}, \frac{1}{32}]$, but the mode is not restricted to squares, making it a bit more flexible, where applicable.

4.3 Decompression

A proposal for a hardware decompressor is illustrated in Figure 4. The two vectors spanning the bounding box, $\hat{v} = a\hat{e}_2$ and $\hat{e}_1$, as well as the lower left point s, are calculated by the green part. The red part calculates the same values for the differential version of the coder. The blue part assigns the right bits for the variable point distribution.

Without implemeting 3Dc, e3Dc and TF in VHDL, it is hard to estimate relative gate counts for the different algorithms. However, comparing Figure 4 with the

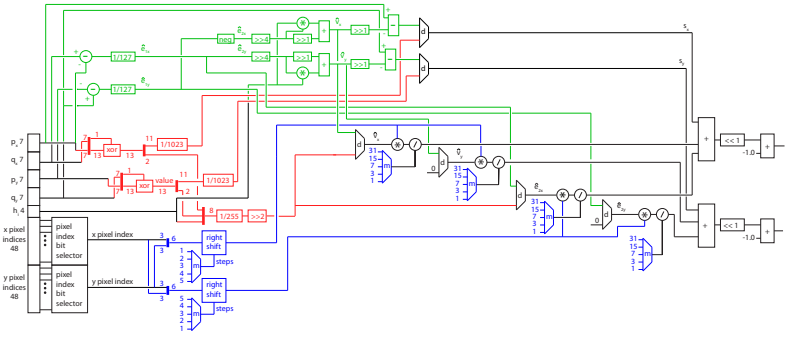


Figure 4: Diagram of a proposal for a hardware implementation of the decoder. ***Green****: calculates* $\hat{\mathbf{v}} = a\hat{\mathbf{e}}_2$ *and* $\hat{\mathbf{e}}_1$ *which are the two vectors spanning the bounding box. It also calculates* $\mathbf{s}$*, which is the lower left point of the bounding box.* ***Red****: calculates,* $\hat{\mathbf{v}}$*,* $\hat{\mathbf{e}}_1$ *and* $\mathbf{s}$ *for the differential version of the decoder. Note that, if* $p_x \geq q_x$ *and* $p_y \geq q_y$*, differential data is stored in* p_x, p_y, q_x *and* q_y*.* ***Blue****: assigns the right bits for the variable bit distribution using the multiplexors marked with m. Likewise, the multiplexors marked with d choose between regular and differential input. Note that multiplication with* $1/127$ *can be approximated efficiently as 1/128 + 1/16384 which is implementable with shifts and additions (not shown). Likewise for 1/255 and 1/1023.*

diagram of e3Dc [6], we see that TF has twice the number of "multiply and divide" units compared to e3Dc, plus two extra smaller multipliers in the green area. Thus a fair guess would be that TF is up to twice as complex as e3DC, which in turn is slightly more complex than 3Dc.

5 Results

To evaluate the visual quality of our compressor, we have used 20 representative normal maps, which are the same ones used previously in normal map compression research [6].

In Figure 8, we present both individual PSNR and maximum angle deviation for the test suite. As can be seen, our algorithm has slightly better scores than e3Dc for the majority of the normal maps, and significantly better scores than 3Dc for all maps. For the “bumpy”-map, e3Dc is better due to that our algorithm uses 7+7 bits for the endpoints, while e3Dc uses 8+8. Further, as all normals in that image are essentially along a horizontal line, there is no gain from being able to rotate the boxes. In the table to the right, we present PSNR values obtained by first averaging the MSE values for all the normal maps. PSNR is then computed on this accumulated MSE using Equation 2. Note that it is not correct to simply average the PSNR scores of the individual images, since this is not a linear operator. In the extreme — if one image would get zero error, it would get infinite PSNR

3Dc	e3Dc	TF
30.87	32.74	33.50

and the aggregate PSNR figure would also be infinite, irrespectively of the errors in the other images. Averaging the MSE and then calculating the PSNR avoids this pitfall. As can be seen, our algorithm has better scores than both 3Dc and e3Dc.

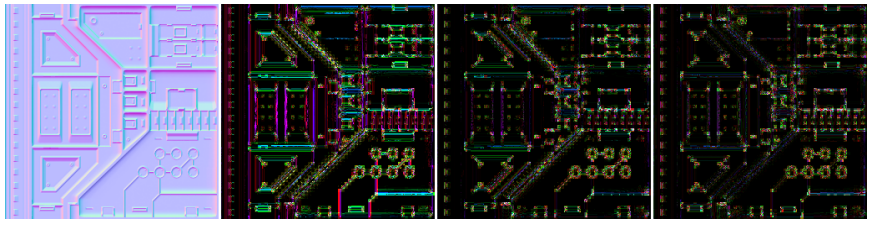


Figure 5: False color images of the pixel errors: From left to right: Original map, 3Dc, e3Dc and TF. We can clearly see improved performance of the TF algorithm over the two 3Dc compressor variants.

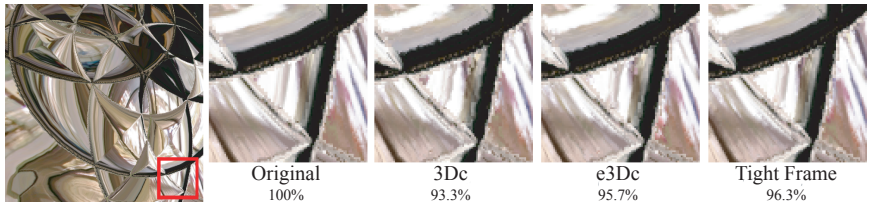


Figure 6: Rendered quality in a real-time engine. Note that the figures below the zoomed images are Structural Similarity values for the entire screenshot.

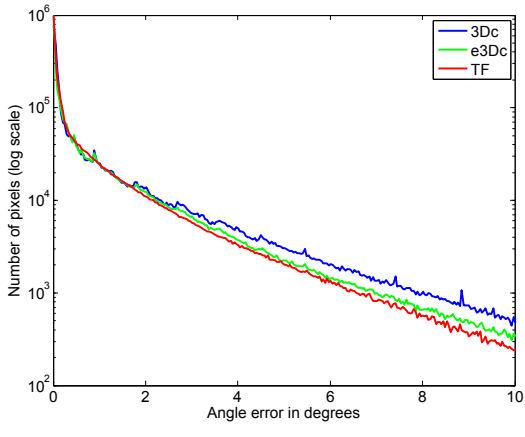


Figure 7: The error distribution of the algorithms.

The maximum angle error (bottom part of Figure 8) indicates that our algorithm is more robust than the other algorithms in all but one image. In Figure 7, we show the histograms over the angular error. Intuitively, it is better to have less area to

the right, and more area to the left. As can be seen, our TF algorithm consistently performs a bit better in this respect.

To further illustrate the improvement of our algorithm, we show false color images of the compressed normal maps in Figure 5, and zoomed-in renderings in Figure 6.

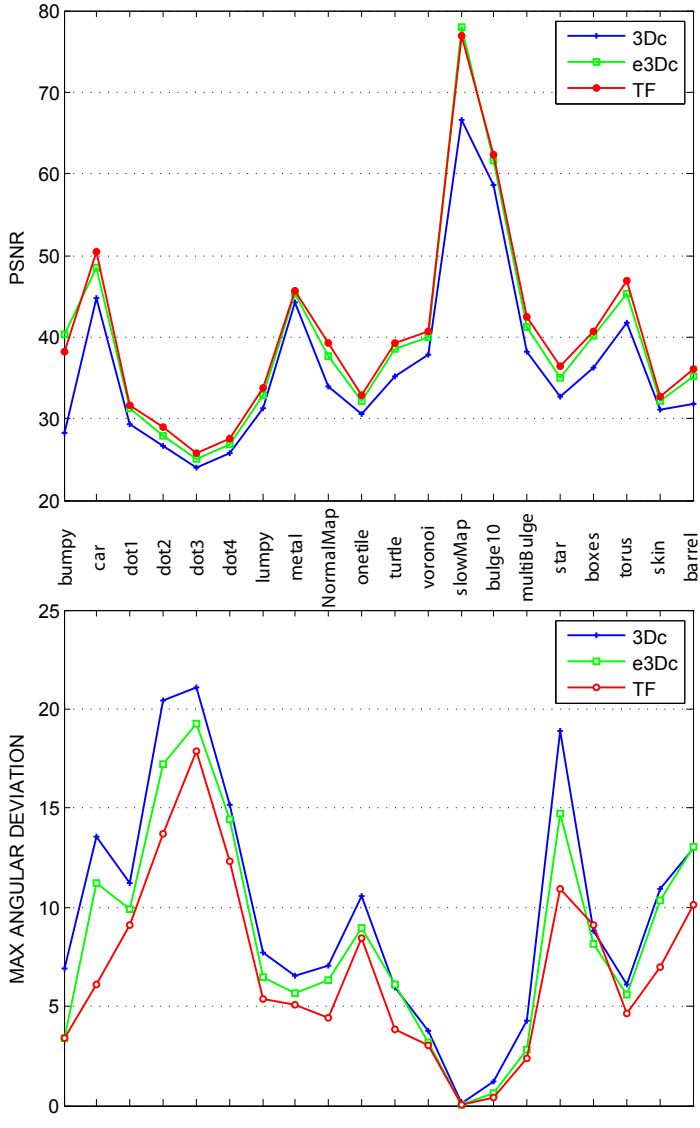


Figure 8: The PSNR (top) and the maximal angular error (bottom) of all images in the test. We can clearly see a more robust behavior for our tight frame (TF) algorithm in both error measures. Please note that all encoders are optimized for MSE.

6 Conclusion

In a sense, our work here is quite incremental, since we have basically put together building blocks from other texture & normal map compression research. However, we have shown that this novel combination gives a powerful normal map compression algorithm with high quality under a wide set of error/quality measures. Furthermore, for mobile devices, compression algorithms are very important, and we hope that our technique can be considered for inclusion in OpenGL ES.

Acknowledgments

We acknowledge support from the Swedish Foundation for Strategic Research and Vetenskapsrådet. Thanks for the NVIDIA fellowship. Thanks to Jon Hasselgren for proofreading.

Bibliography

[1] Andrea F. Abate, Michele Nappi, Stefano Ricciardi, and Gabriele Sabatino. Fast 3D Face Recognition Based On Normal Map. In *Proceedings of ICIP*, volume 2, pages 946–949, 2005.

[2] ATI. Radeon X800: 3Dc White Paper. Technical report, 2005.

[3] Jim Blinn. Simulation of Wrinkled Surfaces. In *Proceedings of SIGGRAPH*, pages 286–292, 1978.

[4] Michael Deering. Geometry Compression. In *Proceedings of SIGGRAPH*, pages 13–20. ACM Press, 1995.

[5] Simon Fenney and Mark Butler. Method and Apparatus for Compressed 3D Unit Vector Storage and Retrieval. Patent WO 2004/008394 A1, 2004.

[6] Jacob Munkberg, Tomas Akenine-Möller, and Jacob Ström. High Quality Normal Map Compression. In *Graphics Hardware*, pages 95–101, 2006.

[7] Jacob Munkberg, Petrik Clarberg, Jon Hasselgren, and Tomas Akenine-Möller. High Dynamic Range Texture Compression for Graphics Hardware. *ACM Transactions on Graphics,*, 25(3):698–706, 2006.

[8] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing,*, 13(4):600–612, 2004.

[9] Alexander Wong and William Bishop. Adaptive Normal Map Compression for 3D Video Games. In *Proceedings of Future Play*, 2006.

[10] Toshihiko Yamasaki and Kiyoharu Aizawa. Fast and Efficient Normal Map Compression Based on Vector Quantization. In *Proceedings of ICASSP*, volume 2, pages 2–12, 2006.

[11] Toshihiko Yamasaki, Kazuya Hayase, and Kiyoharu Aizawa. Mathematical Error Analysis of Normal Map Compression Based on Unity Condition. In *Proceedings of ICIP*, volume 2, pages 253–269, 2005.

[12] Bailin Yang and Zhigen Pan. A Hybrid Adaptive Normal Map Texture Compression Algorithm. In *International Conference on Artificial Reality and Telexistence*, pages 349–354. IEEE Computer Society, 2006.

Paper V

Stochastic Rasterization using Time-Continuous Triangles

Tomas Akenine-Möller Jacob Munkberg Jon Hasselgren

Lund University

ABSTRACT

We present a novel algorithm for stochastic rasterization, which can rasterize triangles with attributes depending on a parameter, t, varying continuously from $t = 0$ to $t = 1$ inside a single frame. This can be used to render motion blur. We develop efficient techniques for rasterizing our primitive, and specialized sampling and filtering algorithms for improved image quality. Our algorithm needs some new hardware mechanisms implemented on top of today's graphics hardware pipelines. However, our algorithm can leverage on much of the already existing hardware units in contemporary GPUs, which makes the implementation fairly inexpensive. By using time-dependent textures, we show that motion blurred shadows and motion blurred reflections can be handled in our framework. In addition, we also present new techniques for efficient rendering of depth of field and glossy planar reflections using our stochastic rasterizer.

Graphics Hardware 2007, pages 7–16

1 Introduction

If objects in the field of view of the camera, or the camera itself move, and the shutter of the camera is open for a finite amount of time, an image with motion blur is obtained. Real photographs and video often contain motion blur, and therefore, this effect is commonly and heavily used in the movie industry using offline rendering tools. In contrast, most real-time graphics applications assume the shutter is open only for an infinitesimal amount of time, which means that motion blur is absent. However, it is our impression that motion blur is a highly desirable feature even for real-time games.

Rendering motion blur is a hard problem to attack since it involves solving visibility in the spatio-temporal domain, i.e., both in screen space and in time. Currently there exists only a few algorithms capable of rendering this effect in real time. However, they usually only solve the problem for a limited domain, e.g., only the textures of the objects are blurred and not the geometrical objects themselves, and consequently, visibility is solved incorrectly.

Cook et al. [8] concluded the following on rendering correct motion blur, and this appear to hold true even today:

> "*Point sampling seems to be the only approach that offers any promise of solving the motion blur problem.*"

Therefore, we introduce an algorithm for rasterization-based point sampling in time using a time-continuous triangle representation. This makes it possible to render motion blurred images with sufficient quality for real-time graphics at only four samples per pixel. Since current GPUs already support spatial supersampling with that amount of samples, we can integrate our algorithm into an existing GPU without increasing the number of samples. In addition, some parts of our algorithm can be executed using geometry and pixel shaders. Only a small portion of our algorithm needs new hardware mechanisms on top of the existing units already available in contemporary GPUs.

This introduction and the entire description of our our algorithms (Section 3) focus on rendering the motion blur effect only. The reason for this is that it greatly simplifies the presentation. However, in our results (Section 4), we show that the exact same framework can be used to render depth of field and glossy reflections as well.

2 Previous Work

An excellent overview of previous work in motion blur research is presented by Sung et al. [31]. In the following, we will review related work that is of particular interest to our research. This means, for example, that we avoid discussing algorithms that produce motion blur only as a post-process, as these cannot solve the problem properly.

Several analytical models for motion blur have been developed [5, 14, 20] for scanline renderers. Due to the evolution of the GPUs into stream processors, these algorithms are not directly well suited for hardware implementation in their current state, since they require a sorting pass to resolve time-space visibility per pixel.

Rendering motion blur using graphics hardware can be done by rendering n images at different points in time, and then averaging these using an accumulation buffer [10, 15]. It should be noted that strobing artifacts appear unless many images are used. However, the final image converges to the correct result when more images are added. These algorithms are expensive in terms of geometry processing, since the entire scene needs to be sent to the graphics pipeline n times.

A variant of these accumulation buffer techniques is *practically frameless rendering* (PFR) [35], which is a rasterization-based version of the original frameless rendering algorithm for ray tracing [4]. In PFR, less than one sample per pixel is generated per frame. For example, one can choose to render to only one fourth of the pixels every frame. After four frames, a complete image has been rendered. A variant of this, called *temporal anti-aliasing*, is supported by some ATI graphics cards [3].

In the REYES rendering architecture [7], primitives are diced until they reach subpixel size, then shading is computed, and finally the primitives are sampled. This is basically a high-quality rasterization engine. However, motion blurred shading cannot be handled correctly since shading is done before sampling. Furthermore, in this original approach, shadows appear to lack motion blur.

For offline high-quality rendering, Wexler et al. [32] conclude that accumulation buffering works well when many images are used, and so they use that approach in their Gelato renderer. However, they also investigate whether a specialized shader can be used to sample stochastically in time. This approach degrades more gracefully than uniform sampling when decreasing the number of samples. They abandon this technique due to inefficient rasterization and because early Z-culling cannot be used, since they write to the depth buffer in the shader. Our work was inspired by Wexler et al's stochastic sampling, but instead of focusing on using only existing hardware, we also develop new hardware mechanisms suitable for implementation on top of today's pipelines for potentially much higher performance.

The remaining motion blur algorithms which we will describe are targeted for real-time graphics. A common disadvantage for these is that the rendered images do not converge to the correct result even if more computations or more samples are used. Some algorithms compute the silhouette of motion, extend the silhouette geometry in the direction of motion and then render semi-transparent primitives [18, 34]. These algorithms cannot correctly handle shaded and textured objects, and so in practice, they are not very useful.

In contrast, Loviscach [23] has presented an algorithm that deals with motion blurred textures. However, blurring takes place only in texture space, and hence spatio-temporal visibility is not solved at all. Another approach is to render an object once into a texture, and at the same time create a vector field of the per-pixel

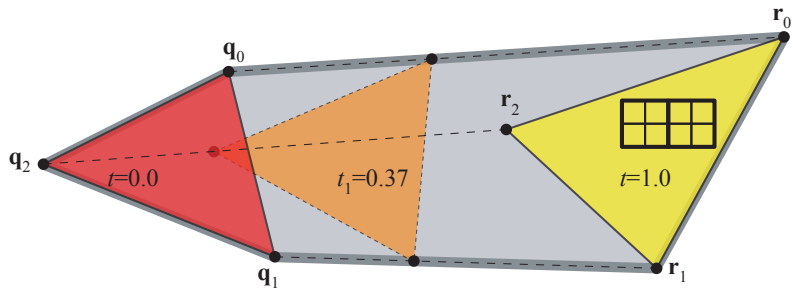


Figure 1: A time-continuous triangle (TCT) defined by a starting triangle, $\Delta\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$, at $t = 0$, and an ending triangle, $\Delta\mathbf{r}_0\mathbf{r}_1\mathbf{r}_2$, at $t = 1$. The TCT is simply the continuous set of linearly interpolated triangles between $t = 0$ and $t = 1$.

motion [29]. In a final pass, the texture is blurred according to the vector field. Again, spatio-temporal visibility is not handled correctly.

Depth of field (DOF) is the effect in which objects outside some distance range appear out of focus. A good survey of techniques to simulate DOF is presented by Demers [11]. Correct DOF can be rendered by distributing rays stochastically over the camera lens, rather than shooting a ray from a single point, or equivalently, render the scene from multiple cameras and accumulate the results. However, for acceptable quality, these approaches require many rays or render passes and are currently too costly for real-time graphics. Faster methods using depth layers, point splatting and variable blur kernels exist, but they cannot resolve visibility correctly.

3 Stochastic Rasterization

In this section, we present our algorithm for stochastic rasterization. As a high-level overview, we rasterize one *time-continuous triangle* (TCT) at a time, and sample it both spatially and in time on a per-tile basis. The design choice of processing one TCT at a time was simple as we would otherwise break the feed-forward principle of contemporary GPUs. Note again that our presentation focuses on rendering motion blur, however in Section 4, we will show that the same algorithm can be used to render other effects, such as depth of field and glossy reflections.

We assume that a TCT is defined at two different instants, $t = 0$ and $t = 1$. See Figure 1. This basically adds another "dimension" to a triangle. If the instants are interpreted as *different times* at a beginning and end of a frame, we can render images with motion blur, for example. The vertices in homogeneous clip space, i.e., after application of the projection matrix (but before division by w), at $t = 0$ are denoted $\mathbf{q}_k$, and at $t = 1$ they are denoted $\mathbf{r}_k$, $k \in \{0,1,2\}$. Furthermore, we

assume that the vertices are interpolated linearly in this space,[1] which is equivalent to linear interpolation in world space. For a certain instant, $t \in [0,1)$, the vertices are: $\mathbf{p}_k(t) = (1-t)\mathbf{q}_k + t\mathbf{r}_k$. This is illustrated in Figure 1. All vertex attributes are linearly interpolated as well for different values of t. A major advantage of the TCT is that we only need to perform geometry processing once, which enables sampling of a triangle at *arbitrary* t-values, $t \in [0,1)$.

3.1 Overview

The basic algorithm works as follows for each time-continuous triangle (with respect to Figure 1), where each pixel is sampled at n different times, t_i, $i \in \{0,\ldots,n-1\}$:

1. Find tight bounding volume (BV) of time-continuous triangle (Section 3.3).
2. Compute time-dependent edge functions (Section 3.3).
3. For each *quad* (2×2 pixels) that overlaps the BV, fetch (or compute) the times, t_i, for the samples in that quad.
4. For each time, t_i, compute edge functions for the triangle $\Delta\mathbf{p}_0(t_i)\mathbf{p}_1(t_i)\mathbf{p}_2(t_i)$ using the time-dependent edge functions. Check whether the quad overlaps this triangle.
5. If overlap from previous step, linearly interpolate vertex attributes using t_i, and execute the pixel shader for the current quad.

Next, we present the details of our algorithm. We start by describing an inexpensive sampling strategy, and continue by developing robust and efficient rasterization of a TCT with Zmin/Zmax-culling. Finally, we introduce time-dependent textures, which can be used for shadow mapping, for example.

3.2 Sampling Strategy

In this section, we will describe our sampling strategy that makes it possible to use as few as four samples per pixel to get usable motion blur. However, our algorithm is not limited by this, and can be generalized to using more samples per pixel. Today, most GPUs have spatial antialiasing schemes with 4–8 samples per pixel or more, and each sample can even execute the pixel shader separately for higher quality. To keep the cost low, we simply want to add the time dimension to each of the samples for such hardware.

[1] This is in contrast to the approach taken by Sung et al. [31], where interpolation takes place in screen space. As a consequence, they cannot handle perspective foreshortening of moving primitives correctly.

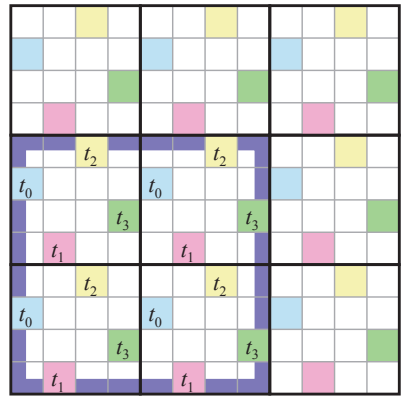


Figure 2: 3×3 *pixels with RGSS sampling. One spatio-temporal sample lies in each colored subpixel. For the lower left quad (outlined in purple), the time samples,* t_0*,* t_1*,* t_2*, and* t_3*, all appear once in each pixel. This gives rise to a undesired quad-sized "pixelation" effect. Notice that all samples that belong to the same time interval,* T_i*, have the same color. For example, all samples in* T_0 *are light blue.*

Our approach is to use n spatio-temporal samples, $\mathbf{s}_i = (x_i, y_i, t_i), i \in \{0, n-1\}$ per pixel, where (x_i, y_i) is the spatial position and t_i is the sample time. Contemporary GPUs always rasterize one *quad*, i.e., 2×2 pixels, at a time, since the GPU then can compute derivatives based on differences in x and y. Our algorithm clearly needs to comply with that requirement. Therefore, a certain time sample, t_i, must occur in *each* of the 2×2 pixels in a quad. Adjacent quads may preferably have a different set of times. Note that each sample has its own depth value, just as in super/multi-sampling.

All our spatio-temporal sampling patterns are completely deterministic, and do not change from frame to frame. In general, if a pixel uses n samples, we let each sample use a predetermined random time, such that $t_i \in T_i$, where T_i is the interval $\left[\frac{i}{n}, \frac{i+1}{n}\right)$ and $i \in \{0, \ldots, n-1\}$. This set will be used in one quad and gives us jittered sampling in time. For an adjacent quad, a new set of time samples $t_i' \in T_i$ is used.

Virtually all contemporary GPUs have some form of rotated grid supersampling (RGSS) implemented. This scheme fulfils the N-rooks requirement [30], and it is illustrated in Figure 2. It is generally accepted that it gives good quality at a cost of only four samples per pixel. In the following, we describe an example of our sampling scheme that uses RGSS. Note that our algorithm is not at all limited to this particular pattern, nor the number of samples. We focus instead on temporal sampling, while allowing different spatial sampling schemes. When adding time to each of these samples, the quad requirement makes the samples share four different times in each quad, and this basically means that the "pixels in time" will appear to have a size of 2×2 instead of the ideal case of 1×1 pixel.

To avoid this problem, we offset the quads depending on which time interval, T_i, the sample belong to. See Figure 3. For all samples within the time interval T_0, we use the standard quads, but for T_1 we offset the quad by one pixel to the right. For samples in T_2, we offset the quad one pixel upwards, and finally for T_3, the quad is offset one pixel to the right, and one pixel upwards. As can be seen, this guarantees that the set of time samples inside a pixel is different from neighboring pixels, which reduces the previously mentioned pixelation effect.

A common strategy to improve the quality of spatial antialiasing is to use larger filter kernels when computing the final color of a pixel. When increasing the kernel for spatio-temporal filtering, we would ideally like to include samples with times different from the times inside the pixel, in order to improve the sampling resolution in the time dimension. In the following, we extend RGSS so that another four samples are used in the filter kernel, and we simply choose the four spatially closest samples. Note that our reasoning applies with minor modifications to any number of samples.

Assume we want to compute the final pixel color of the center pixel in Figure 3 by weighting together the samples with these times: t_0, t_1, t_2, t_3, t_0', t_1'', t_2'', and t_3'. From the figure, we notice that $t_0 \in T_0$ shares subpixel row with $t_0' \in T_0$, and $t_1 \in T_1$ shares subpixel column with $t_1'' \in T_1$, and so on. This is not ideal, at least not from an N-rooks perspective.

To remove this disadvantage, and thus improve sampling and filtering quality, we have devised a solution, which is shown in Figure 4. It is a straightforward task to verify that our sampling scheme gives eight different times for the eight spatio-temporal samples used for computing the final color of a pixel. As a final improvement of the time samples, consider two time samples belonging to the same time interval, T_i, inside the filter kernel. An example consists of t_0 and t_0'' (Figure 4), which both belong to the time interval $T_0 = [0, 0.25)$. To further improve the sam-

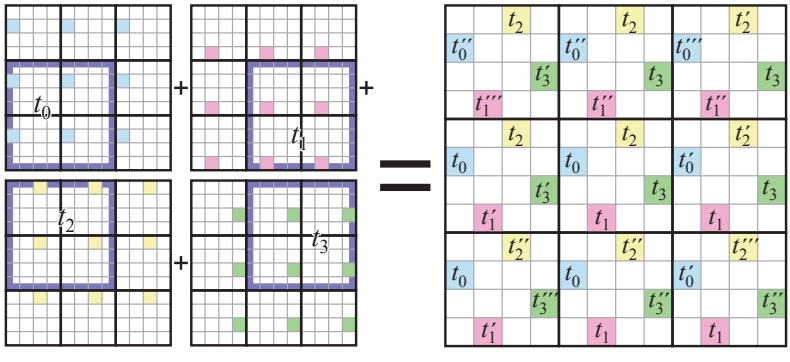


Figure 3: By offsetting the quads (purple) for the different samples in time, we obtain a sampling scheme where neighboring pixels have different sets of times.

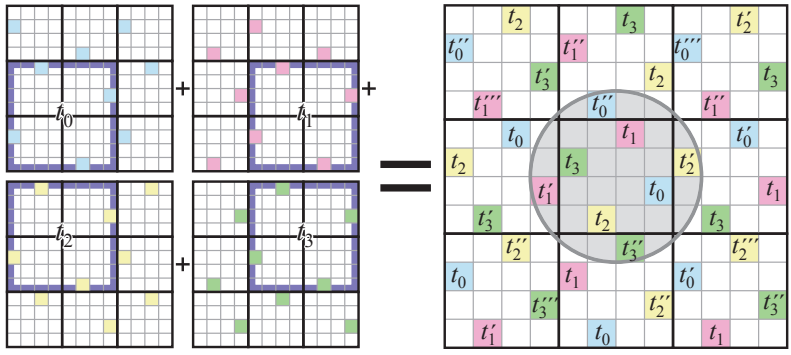
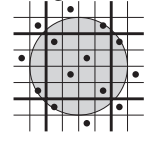


Figure 4: We have redistributed the time samples inside the quads in order to avoid two samples with same color (i.e., belonging to the same time interval, T_i) being on the same subpixel column or row. To the right, this is clearly so for the samples inside the gray filter kernel. Note that only one quad is shown, while they in reality repeat over the entire pixel grid. Furthermore, the spatial sampling pattern repeats after 2×2 pixels, but the times of the samples have a longer period (typically, a 32×32 random table is used).

pling quality, we make certain that $t_0 \in T_0^-$ and $t_0'' \in T_0^+$, where $T_0^- = [0, 0.125)$ and $T_0^+ = [0.125, 0.25)$. In general, we split T_i in the middle into T_i^- and T_i^+. This can be ensured when the sampling pattern is generated. The result is a sampling scheme with four generating samples per pixel, and with the larger filter kernel we obtain eight jittered time samples per pixel. Compared to RGSS, the added cost is essentially only more expensive filtering, which is done only once per pixel when the image has been rendered.

Note that the actual spatial positions can easily be redistributed to form another pattern. For example, we could use the pattern, inspired by Laine and Aila [21], shown to the right instead. In our experience, the *spatial* anti-aliasing would change a little bit compared to RGSS, but the *temporal* anti-aliasing remains very close to constant due to that we still get eight jittered time samples. Recall that the focus of our paper is not on the spatial sampling pattern.



Next, we describe how the filtering of the samples is done. Assume the colors of the samples inside a pixel are denoted, $\mathbf{c}_l^0$, where $l \in \{0,1,2,3\}$, and the colors of the four closest samples in the neighboring pixels by $\mathbf{c}_l^1$, again with $l \in \{0,1,2,3\}$. We use a *low-pass filter* to compute the final pixel color:

$$\mathbf{C} = w_0 \sum_{l=0}^{3} \mathbf{c}_l^0 + w_1 \sum_{l=0}^{3} \mathbf{c}_l^1 \tag{1}$$

For all our tests, we use $w_0 = 5/32$ and $w_1 = 3/32$. This gives a good trade-off between spatial and temporal blurring. Naturally, it is simple to change the weights according to the purpose. We attempted to use another four samples from

the neighboring pixels, but this did not give much of an effect on the quality.

It should be noted that the spatial positions can be jittered inside the subpixel using, for example, multi-jittering [26]. By using a smaller grid of such spatial samples, we basically get a spatial interleaved sampling scheme [19]. Extending the ideas of this subsection to schemes with more samples per pixel is straightforward, and is therefore omitted.

3.3 Traversal of Time-Continuous Triangles

In this section, we describe how a time-continuous triangle (TCT) can be traversed, i.e., how the pixels inside a TCT can be found efficiently. Notice that the quadrilateral sides of a TCT are, in general, bilinear patches, and hence not necessarily planar. This makes clipping a TCT against the canonical view volume a complex procedure. Instead we decided to use edge functions [28] derived directly from the homogeneous coordinates [24, 27], $\mathbf{q}_k$ and $\mathbf{r}_k$ with $k \in \{0,1,2\}$, of the TCT. This avoids clipping altogether. Using the two-dimensional axis-aligned bounding box of the TCT to limit the rasterization can make the traversal algorithm visit an excessive amount of pixels that are outside the TCT [32].

Therefore, we propose a two-level algorithm for efficient rasterization of a TCT. First, a tight three-dimensional oriented bounding box (OBB) around the TCT is rasterized (Section 3.3). Second, for fragments inside the OBB, per-pixel evaluation of time-dependent edge functions (Section 3.3) follows. For samples inside the time-dependent edge functions, the pixel shader is executed.

OBB Traversal

We decided to use oriented bounding boxes (OBBs) around our TCTs to limit the number of pixels visited during traversal. To robustly handle cases where a TCT moves from in front of the viewer to behind the viewer, we rasterize only the *backfaces* of the OBB without any depth testing (which is done in the next stage of our algorithm). This is similar to how shadow volume rendering [9] handles the case when the viewer is inside a shadow volume and when a shadow volume intersects the near plane. For pixels covered by the OBB backfaces, we proceed to testing with time-dependent edge functions (next section).

Our method for computing a tight OBB is simple and gives very good results in the majority of all cases. All computations are done before division by w, and so we use the (x,y,w)-coordinates of the vertices of the TCT. The major axis of the OBB is computed as the difference between the center of the starting and ending triangle of the TCT. If this vector is near zero, an axis-aligned bounding box (AABB) is computed instead. Otherwise, we project the edges of the TCT onto the plane whose normal is the major axis. For the second axis of the OBB, we use the longest projected edge. Again, if there is no such non-zero vector, we revert to using an AABB. The third axis is obtained with a cross product. This algorithm

can be implemented in a geometry shader.

Discussion Several different possibilities for this stage of the algorithm were explored. We tried using the convex hull of the homogeneous coordinates of the TCT, and we devised a hardware-friendly algorithm for this. However, it is very difficult to obtain a robust algorithm without handling a large set of special cases. In addition, the starting triangle of the TCT may be behind the camera, and in such situations, it is not even clear what the definition of the convex hull using homogeneous edge functions is. Another possibility is to use *bounding prisms* (BP) as used for caustic primitives [13], for example. The construction algorithm for BPs works well for typical caustic rendering, but for more general settings, we have found that BPs with infinite size can result. In addition, the computation of BPs was more costly than OBBs. Hence, using OBBs is a good trade-off in terms of robustness, speed, and simplicity.

Time-Dependent Edge Functions

Due to the traversal from the previous section, we know that a quad overlaps with the OBB of the TCT. Now, we need to determine whether the samples, $\mathbf{s}_i$ (see beginning of Section 3.2 for the definition of samples), overlaps with the TCT. To be able to do this efficiently, we introduce *time-dependent edge functions*.

First, recall that the vertices, $\mathbf{q}_k$ and $\mathbf{r}_k$, $k \in \{0,1,2\}$, are in homogeneous clip space after application of the projection matrix (but before division by w), and that the camera is located in $(0,0,0)$. Furthermore, let us introduce a "truncated" variant of a vector $\mathbf{v}$ as $\hat{\mathbf{v}} = (v_x, v_y, v_w)$. This simply means that we create a three-dimensional vector from a four-dimensional by skipping the z-coordinate.[2] The edge function through two vertices, say $\hat{\mathbf{p}}_0$ and $\hat{\mathbf{p}}_1$, is then [24, 27]:

$$e(x,y,w) = (\hat{\mathbf{p}}_1 \times \hat{\mathbf{p}}_0) \cdot (x,y,w) = ax + by + cw, \tag{2}$$

where $(\hat{\mathbf{p}}_1 \times \hat{\mathbf{p}}_0) = (a,b,c)$. Now, since the vertices are functions of time, $\hat{\mathbf{p}}_k(t) = (1-t)\hat{\mathbf{q}}_k + t\hat{\mathbf{r}}_k$, we simplify the expression for the edge function parameters:

$$\begin{aligned}(a,b,c) = (\hat{\mathbf{p}}_1 \times \hat{\mathbf{p}}_0) &= ((1-t)\hat{\mathbf{q}}_1 + t\hat{\mathbf{r}}_1) \times ((1-t)\hat{\mathbf{q}}_0 + t\hat{\mathbf{r}}_0) \\ &= t^2\hat{\mathbf{f}} + t\hat{\mathbf{g}} + \hat{\mathbf{h}},\end{aligned} \tag{3}$$

where:

$$\begin{aligned}\hat{\mathbf{m}} &= \hat{\mathbf{q}}_1 \times \hat{\mathbf{r}}_0 + \hat{\mathbf{r}}_1 \times \hat{\mathbf{q}}_0 \\ \hat{\mathbf{h}} &= \hat{\mathbf{q}}_1 \times \hat{\mathbf{q}}_0, \\ \hat{\mathbf{f}} &= \hat{\mathbf{h}} - \hat{\mathbf{m}} + \hat{\mathbf{r}}_1 \times \hat{\mathbf{r}}_0, \\ \hat{\mathbf{g}} &= -2\hat{\mathbf{h}} + \hat{\mathbf{m}},\end{aligned} \tag{4}$$

This means that we have simple expressions for all the edge function parameters, (a,b,c). For example, we have: $a(t) = f_x t^2 + g_x t + h_x$. Note that $\hat{\mathbf{f}}$, $\hat{\mathbf{g}}$, and $\hat{\mathbf{h}}$ can be

[2] Note that due to the projection matrix (e.g., OpenGL or DirectX), this vector is in a scaled and translated camera space. This can be verified by examining the elements of the projection matrix.

computed in the triangle setup. For a specific time, t_i, and spatial sample position, (x_i, y_i), we now arrive at the time-dependent edge function:

$$e(\mathbf{s}_i) = e(x_i, y_i, t_i) = a(t_i)x_i + b(t_i)y_i + c(t_i), \tag{5}$$

where $w = 1$ since we now are dealing with screen space (x, y)-coordinates.

Once the three edge functions, $e_j(\mathbf{s}_i)$, have been computed, we can determine whether a sample, $\mathbf{s}_i$, is inside the TCT at time t_i. If this is true, we linearly interpolate the vertex attributes of the starting and ending triangle of the TCT with respect to t_i, and pass them on downwards the pipeline.

Note that since each time-dependent edge function is defined by four vertices, cracks "in time" between two TCTs sharing an edge can be avoided using a simple tie-breaker rule [24]. However, to avoid small numerical inaccuracies when evaluating the expressions in Equation 4, we also make sure that two TCTs sharing an edge always compute the parameters $\hat{\mathbf{f}}$, $\hat{\mathbf{g}}$, $\hat{\mathbf{h}}$ in exactly the same way. This is done by swapping $\mathbf{q}_1$ and $\mathbf{q}_0$ so that the first point is always the one with smallest x-value before calculation of the parameters starts. If the x's are equal, testing continues with y, and so on.

Discussion Another possible solution would be to interpolate edge functions in *screen space*. Consider one edge function, $e_0(x, y) = a_0x + b_0y + c_0$, for the first triangle, $\Delta\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$, and the corresponding edge function, $e_1(x, y) = a_1x + b_1y + c_1$ for triangle $\Delta\mathbf{r}_0\mathbf{r}_1\mathbf{r}_2$. To find the edge function for a specific time, $t \in [0, 1)$, one could interpolate the edge functions parameters, e.g., $a(t) = (1-t)a_0 + ta_1$, and so on. However, this does not take perspective foreshortening into account, and in addition, it requires the TCT to be clipped, which we also want to avoid.

For simplicity, we have limited ourselves to linear interpolation of vertex positions and attributes. To get curved motion blur, we can use our technique together with an accumulation buffer for faster performance. Higher-order interpolation, such as quadratic or cubic Bézier curves, is of course also possible. Besides the actual interpolation, only the OBB computation need to be altered, since more vertices need to be processed.

3.4 Zmin/Zmax-Culling

Zmin- and Zmax-culling [2, 25] are crucial for good depth buffer and texture access performance. Therefore, one of our goals has been to make stochastic rasterization work with this type of algorithms. Hence, a conservative estimate of minimum and maximum depth inside a tile (often 8×8 pixels) for a time-dependent triangle is needed.

We limit our discussion here to Zmax-culling, where a conservative estimate of the minimum depth value, denoted z_{min}^{tri}, of a triangle inside a tile is needed. The maximum of the depth values inside a tile is denoted z_{max}. If $z_{min}^{tri} > z_{max}$ we can avoid processing the triangle in that tile. Extending this to Zmin-culling is straightforward.

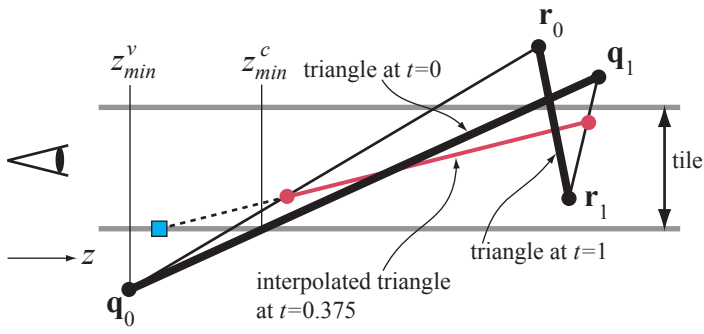


Figure 5: A time continuous triangle (TCT) defined by a starting triangle, $\Delta\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$, and an ending triangle, $\Delta\mathbf{r}_0\mathbf{r}_1\mathbf{r}_2$, here shown in two dimensions. Due to that these triangles do not have the same orientation, problems in Zmax-culling can occur. Normally, we compute $z^{tri}_{min} = \max(z^v_{min}, z^c_{min})$. In this case, this is not correct, since at, e.g., $t = 0.375$, the true depth at the tile corner (blue square) is smaller than z^c_{min} which is computed using the plane equations of the two triangles of the TCT. Our solution is simply to use $z^{tri}_{min} = z^v_{min}$ when the orientation of the triangles changes. This gives a conservative estimate.

A conservative estimate of the minimum depth value of a triangle inside a tile is simply the minimum of the vertices of the triangle being rendered. Let us denote this value by z^v_{min}, where the superscript indicates *vertices*. However, this can become overly conservative, for example, when rendering a large triangle with a normal almost perpendicular to the view direction. To improve this, one can also compute the depth at the tile corners using the plane equation of the triangle, and computing the minimum of these. Let us call this value z^c_{min}, where the superscript indicates *corners*. An improved estimate of the minimum depth of the triangle inside a tile is then:

$$z^{tri}_{min} = \max(z^v_{min}, z^c_{min}). \tag{6}$$

This is a commonly used technique. In the case of rendering a TCT, we again evaluate Equation 6, but the computation of the terms in the max-function becomes a bit more complex. The value z^v_{min} is computed using the six vertices of the TCT, and z^c_{min} is computed using the plane equations of the starting and the ending triangles of the TCT. This is conservatively correct as long as the orientations of the starting and ending triangle are the same. When this is not true, you may not always get a correct conservative value. One reason for this, is that the depth at the corners of a tile can become unbounded when the orientation of a TCT changes from, for example, backfacing to frontfacing. An example is illustrated in Figure 5.

However, there is a straightforward solution to this. If there is *no* change in orientation, we compute z^{tri}_{min} using Equation 6. In the case of a change in orientation, we simply use the minimum of the depths at the vertices of the starting and ending triangles, i.e., $z^{tri}_{min} = z^v_{min}$.

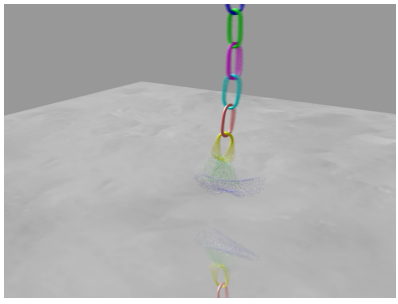
Discussion In the description above, we have assumed that we store one z_{max}-value per tile for all different times of the sample inside a tile. An alternative would be to store, for example, four values per tile: z^i_{max}, $i \in \{0,1,2,3\}$, where z^i_{max} is the max-value of the depths belonging to the time interval, T_i. In a sense, the low-resolution depth buffer that contains z_{max}-values, is extended in the time dimension. While this is clearly possible and would provide more efficient culling, we have decided to leave this for future work, since it does not fit well with contemporary GPUs as they store only one z_{max}-value per tile.

3.5 Time-Dependent Textures

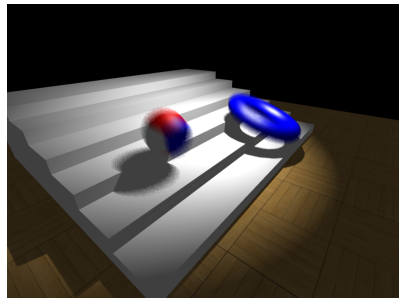
Motion blurred geometry without motion blurred shadows spoils the entire concept, almost. Hence, we would like to support motion blurred shadows in our spatio-temporal framework. Shadow mapping [33] is a commonly used technique for (static) shadow generation. Lokovic and Veach [22] introduce deep shadow maps, where motion blur is handled by associating a random time with every shadow map sample within a texel. The time samples are averaged together, which means that the time dimension is reduced to a single blurred value. As a consequence, the authors concluded that this approach will be correct only for static shadow receivers as seen from the light source.

We alleviate this problem by introducing time-dependent textures, which holds a set of time samples per texel and support time-dependent reads and writes. When generating the shadow map, we use the sampling strategy of Section 3.2 and store n depth values per texel, each associated with a unique time, t_s. When rendering from the camera, the visible sample will be associated with a time t_i. During time-dependent texture lookup, we ensure that the screen space sample, $t_i \in T_i$, access the shadow map sample with time t_s also in T_i. This will reduce self-shadowing artifacts for cases with moving receivers. With n jittered time samples per texel in screen and light space, our approach guarantees that $|t_i - t_s| < 1/n$. If more time samples are added per pixel, the result converges towards the correct image. With uniform time sampling, $t_i = t_s$, the images instead contain apparent strobing artifacts.

In general, time-dependent textures are useful as render targets for dynamically generated effects, where we need to store time-dependent depth or color values. A simple technique for generating reflections for curved geometry is to first render a cube map from the position of the object, and then access this map with the reflection vectors during rendering of reflecting objects. If we use time-dependent textures for cube map generation and lookups, we can handle correct motion blurred reflections, even when both the reflection vector and the cube map changes over time. See Figure 8 for an example.

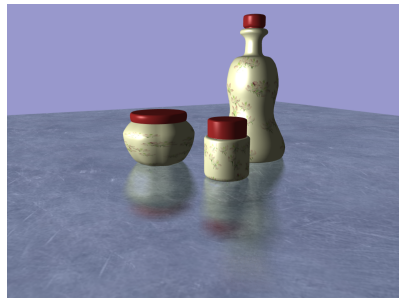


A. Motion Blur + Blurred Reflections



B. Motion Blurred + Shadows

C. DOF



D. Glossy Reflections

Figure 6: Our stochastic rasterization algorithm can render images with a variety of effects. A+B: motion blur with only four samples per pixel. Notice the motion blurred reflections in A, and the motion blurred shadows and highlights in B. C: depth of field rendered with only eight passes with four samples per pixel. D: glossy planar reflections using four passes. Note that the target is real-time graphics, and so to be fair, the quality is best judged from our video.

4 Results

We have implemented a subset of OpenGL 2.0 in a functional simulator in C++. Currently, there are two ways to specify vertex positions. For the first method, you set all your transforms (model + view + projection), and then ask the API to "remember" the composite matrix. This is the transform matrix for $t = 0$. After that you set the all the matrices again (this time for $t = 1$), and then render your objects. The other method simply specifies a double set of vertex positions. We call one such rendering an SR pass.

Note that we use the abbreviation ABT for accumulation buffering of static images. However, we can also accumulate images rendered with SR. We call this *stochastic rasterization accumulation* (SRA).

We emphasize the fact that still images only reveal a small part of the perceived image quality. Since our target is real-time rendering, we refer the reader to the videos of this submission in order to judge the quality of our motion blur, depth of

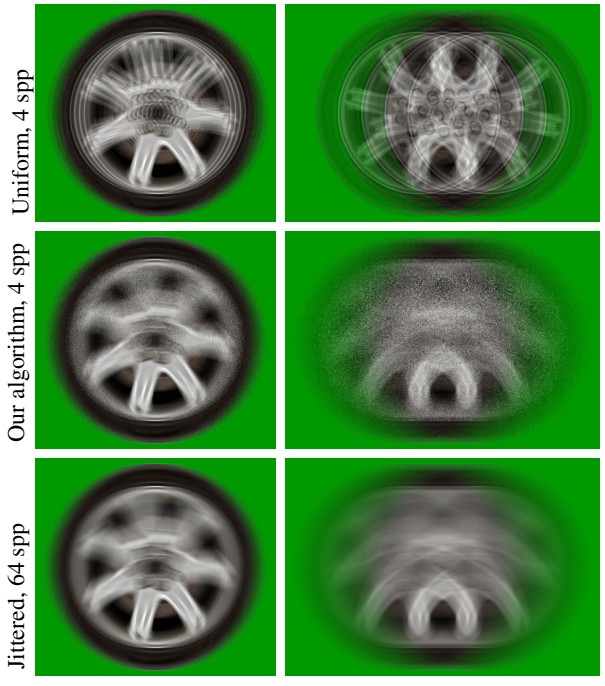


Figure 7: Motion blur caused by both translation and rotation. Note the strobing artifacts obtained using four samples per pixel with uniform sampling, i.e., similar to Wexler et al's method [2005]. The left column shows a slow motion, while the right shows a five times faster motion.

field, and glossy reflections.

For Zmax-culling, we have not gathered statistical results. We note that if the geometry is static, the algorithm works as well as the old Zmax-algorithm. For moving geometry, culling will occur when possible, but there is really no algorithm to compare to, so this has been omitted for now.

In the following, we report our results for motion blur, depth of field, and glossy planar reflections. It should be noted that our framework can only handle one extra dimension at a time, and therefore only one effect at a time. For example, we cannot handle DOF and motion blur in the same image and pass.

4.1 Motion Blur

For our motion blur rendering results, we use only a single SR pass with four samples per pixel, except where otherwise mentioned.

Cook et al. [8] point out a number of hard cases of motion blur: specular highlights, intersecting objects, shadows and reflections. As seen in Figure 6A and B,

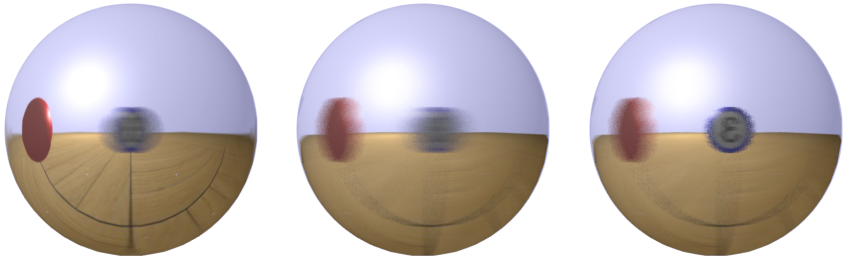


*Figure 8: A moving blue ball and a static red ball are reflected in a chrome sphere using cube mapping. **A**. Static camera. Notice the blurred blue ball and the sharp red ball. **B**. The camera is moving in the same path as the blue ball so that there is no relative motion between them. With a standard cube map, both balls appear blurred. **C**. With a time-dependent cube map, the reflected blue ball approaches the correct result, which is a sharp reflection. Four samples per pixel are used in all these examples.*

our algorithm handles these cases due to its stochastic nature. The chain elements intersect, and have complex motion, and the staircase scene shows specular highlights and blurred shadows using time-dependent shadow maps. Note that these images were rendered using only four samples per pixel. As the algorithm allows sampling at arbitrary times within the frame, strobing artifacts are replaced by (less noticeable) noise without increasing the sampling cost. It should be noted that the algorithm correctly handles scenes where both the camera and geometry are animated as the total motion simply becomes composite transform matrices applied at $t = 0$ and $t = 1$.

In Figure 7, a simple model of a textured wheel is shown. The model is translated and its texture coordinates rotated, which means that motion blur is both obtained due to the translation and rotation. This kind of effect is not handled correctly by methods where a static image is rendered first, and then that image is blurred according to motion vectors [29].

This example clearly shows the flexibility and power of our method, and indicates that the quality converges towards the reference solution (bottom row in Figure 7) in this case, which is a major advantage.

An example of blurred reflections from moving objects using a time-dependent cube map is shown in Figure 8.

Since the TCT uses linear interpolation, the algorithm cannot render higher order movement directly. For example, a rotating sphere gets a blurry edge where the relative motion is largest, and a fast circular arc movement of, say, a sword will get a triangular motion trail. Artifacts from such non-linear motion can be found in our video. These situations can be improved using an SRA technique, and generating TCTs for uniform subintervals of the time inside a frame. Our video shows that stochastic rasterization quickly resembles the ground truth, while accumulation

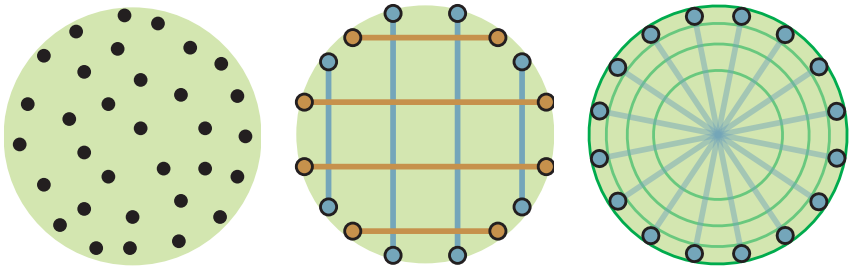


Figure 9: DOF sampling patterns

buffering techniques suffer severely from strobing artifacts in these cases.

In Figure 10, we compare motion blur renderings with 4 samples per pixel against 8 samples per pixel in a single pass. Naturally, the quality is higher the more samples being used.

Blurred shadow maps inherit the shadow bias problem from standard shadow maps, which is somewhat enhanced by the added uncertainty in time. However, already with four jittered time shadow samples per pixel, we can render nice-looking, blurred shadows suitable for real-time content.

4.2 Depth of Field

Computing images with depth of field (DOF) is computationally expensive. Haeberli and Akeley [15] render DOF using an accumulation buffer with point sampling on the aperture of the camera lens, which is illustrated to the left of Figure 9. In this example, we use 32 uniform random points. For DOF with our algorithm, we use an SRA approach, i.e., we accumulate several images from SR passes. With our SR algorithm, we can instead sample an *entire line* on the lens area in a single pass. This is illustrated in the middle illustration of Figure 9 with four horizontal and four vertical lines. Doing this, is a simple matter of setting the camera matrix for the start point of the line, then ask the API to "remember" the composite matrix, and then set the camera matrix for the end point of the line. This gives a DOF-effect in the direction of the line. For example, if we use a horizontal line, the DOF-effect will only be horizontal, but it will be stochastically sampled, i.e., with good quality. Using a multi-pass technique, we can average the results from a number of "line samples."

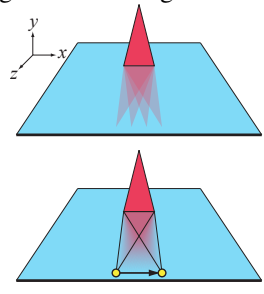
Using the line sample scheme above, it is quite clear that banding artifacts can appear, both horizontally and vertically. For best results, we need to sample using as long lines as possible, while also maximizing the number of angles of the lines. One such sampling pattern is shown to the right in Figure 9. However, this scheme has increased sample density the closer to the center you get. Our solution is to redistribute the sample times, t_i, which is illustrated by the circles. Theoretically,

this should be done with a $\sqrt{t}$-like function being reflected around $(0.5, 0.5)$. In practice, we do it with a `smoothstep`-function, $s_i = t_i^2(3 - 2t_i)$, which is accurate enough. When this transform has been applied, the time-dependent edge functions use s_i (instead of t_i) for inclusion testing as usual.

To the best of our knowledge, we have not seen any DOF algorithm using line samples on the lens aperture. In our experience, this works really well already using only eight lines, i.e., eight passes, with four samples per pixel. This should be compared to grid point sampling the lens, which can require more than 100 samples to get stable results during animation [16]. In our experience, however, similar results to ours can be obtained with uniform random sampling over the lens using 32 image passes. Again, note that such an approach requires the scene geometry to be processed 32 times. See Figure 6C for an example of DOF rendering using our algorithm.

4.3 Glossy Reflections

For rendering planar glossy reflections, Diefenbach and Badler [12] suggested that the reflected object is sheared in the x- and z-directions, with increased shearing effect the smaller y gets. This is illustrated for shearing in x to the right. Rendering the scene many times with different amounts of shear gives glossy reflections in the accumulated image. Our stochastic rasterizer can again be used to advantage even in this case, using an SRA approach, as shown in the bottom right illustration. By using the concept of line samples from the previous subsection, we realize that a shearing pass in x is done as a line sample, where the outer vertex points are sheared the maximum amount in both directions. In practice, the shearing effect can be computed in the vertex shader. Figure 6D shows this effect using stochastic rasterization in the x-direction, and only four passes in z. As shown in the animation, no banding artifacts are noticeable.



4.4 Bandwidth Analysis

One can easily imagine that the random nature of our algorithm can break several of the features in a modern GPU, which exploits coherency in the rendering. This includes buffer compression and texture cache performance, for example.

The two scenes with most motion on textured objects are the Sponza DOF (Figure 6C), and the rotating/translating wheel (Figure 7). For all our tests, we used a single 6 kB texture cache, which is perfectly reasonable (for comparison, a Geforce 8800 has 8 kB per multi-processor). Also, our algorithm used four samples per pixel, while ABT used four passes, which also gives four samples per pixel. In the wheel scene, our algorithm used 225 MB of off-chip texture bandwidth, while

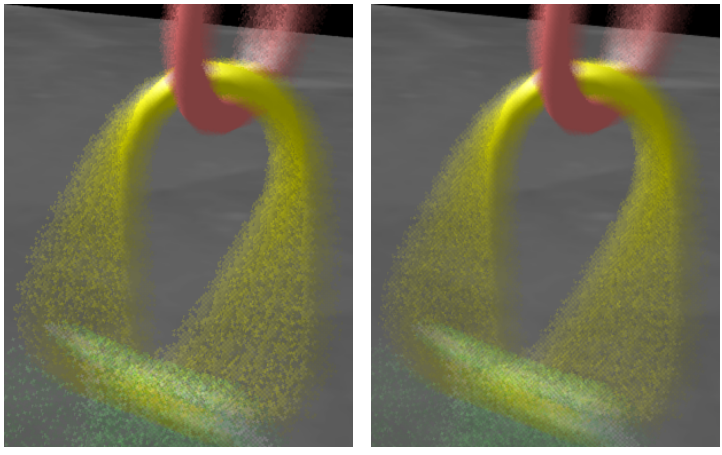


Figure 10: Motion blur rendering in a single pass. Left: 4 samples per pixel. Right: 8 samples per pixel. Note that the motion is about 100 pixels wide in the fastest moving region. Recall that to halve the variance, we need to quadruple (4×) the number of samples.

ABT used 314 MB. For Sponza DOF, the advantage of our algorithm becomes more pronounced: ABT used 2314 MB, while our algorithm used only 1056 MB. In addition to this improvement, we believe that a texture cache coherent rasterization order can improve our numbers further. This is left for future work at this point.

For the depth buffer, we implemented depth offset compression (DOC) [17]. When using this on the chain scene (Figure 6A), the depth is compressed down to 62.5%. Hasselgren and Akenine-Möller report compression rates of about 60% on a set of static scenes [17]. This gives an indication that depth buffer compression can work quite well. However, we believe that clever new algorithms can be implemented to further increase compression. For example, using four layers in DOC could help quite a bit. An interesting avenue for future work would be to compress all samples in each time subinterval separately. For example, we can compress all samples inside the time interval [0.0,0.25) separately. This would increase the coherence, and improve compression. We note that this is important, and we would like to investigate buffer compression for SR in the future.

5 Discussion

Our algorithm for stochastic rasterization (SR) should be seen as a complement to standard rasterization. It is a feature that the programmer can turn on exactly when needed. For parts of a scene with little or no relative motion, standard rasterization

can be used together with multi-sampling.[3] This gives spatial antialiasing at a low cost.

However, for parts with faster relative motion, the more expensive stochastic rasterization can be activated with supersampling to obtain spatio-temporal antialiasing. Thus, the rendering can be seen as a combination of multi-sampling and supersampling. Note that for motion-blurred regions, the spatial positions of the samples do not matter that much. Instead, it is the temporal distribution of the samples that determine quality. For static parts, we get the same quality as using spatial anti-aliasing only, and our algorithm is not directly dependent on any particular scheme. We choose RGSS because it gives good spatial antialiasing, and is accepted in the industry.

We also want to emphasize a few very important features of using stochastic rasterization. First, if stochastic rasterization use n samples per pixel, we can compare this to accumulation buffering techniques (ABT) rendered with n passes, where each pass renders a static image. Our video clearly shows that the strobing artifacts of ABT are more noticeable. However, there is also a significant advantage in terms of sending geometry over the bus and geometry processing. With stochastic rasterization we only send the geometry once, and transform that geometry twice in the geometry shader (using different matrices for $t = 0$ and $t = 1$). In contrast, ABT would send the geometry n times over the bus, and process the geometry n times. This makes for a substantial improvement already at four samples per pixel.

Note that ABT (as defined in the previous paragraph) converges to a correct result the more static images that are accumulated. Our SR algorithm can render n images and accumulate them as well, but in our case convergence will be much quicker due to the stochastic nature of our algorithm. Furthermore, SR degrades more gracefully than ABT, which makes SR useable over a wider range of sampling rates.

A further use of SR is "practically frameless rendering" [35], which is described briefly in Section 2. Assuming that it is possible to disable writing to a specific set of pixels or samples, we can use SR to render motion blurred triangles into, say, only every 4th pixel. This would give better image quality compared to the original approach, since SR can provide stochastic sampling of the geometry in each rendering pass.

Direct hardware support of our stochastic rasterization algorithm would require rather moderate additions since we could leverage on existing supersampling and multisampling hardware in contemporary GPUs. Transforming and setting up a time-continuous triangle can be done in the geometry shader, as well as computing an OBB. For a full implementation, our sampling/filtering, Zmin/Zmax-culling, and time-dependent texture lookups would require some small hardware modifications.

We did a partial implementation of the "inner loop" of our algorithm in a frag-

[3] We use the following terminology: for multi-sampling, the pixel shader is executed only *once per pixel*, while for supersampling, the pixel shader is executed *once per sample*.

ment program. The time of the sample is computed through a texture lookup, and we interpolate the time-dependent edge functions based on that time, evaluate the interpolated edge functions based on spatial coordinates, and finally compute the perspective-correct barycentric coordinates for the sample. We assumed that the edge-function setup was done in a previous step, and used uniform parameters to pass per-triangle data in lack of better alternatives.

By analyzing this program using a shader performance tool, `nvshaderperf`, we found that this shader program took 11 clock cycles to execute on a GeForce7800, with an expected fillrate of 872.73 Mpixels/s. This kind of performance can fill the screen eight times in 100 fps at 1024×1024. With a GeForce 8800, this would be much higher, but `nvshaderperf` did not support this card when we did our tests. Our conclusion is that we need native hardware support for time-dependent edge functions and interpolation using these to reach higher performance. In our current implementation, we practically perform inside-outside test and interpolation twice (first using native hardware, and then in the pixel shader), and it would be nice to avoid that.

For the pseudo-random time pattern, we use a fixed time table of 32×32 random numbers in the interval [0,1), as described in Section 3.2. We have not seen any visual difference between a 128×128 and a 32×32 table. Smaller tables start to alter the image quality. Due to our sampling strategy with the interval [0,1) split into eight subintervals, and random sampling done inside each such subinterval, we already have three bits of the random number implicitly. Empirically, we found that adding another three bits is enough per x and y for the sampling locations. This means that we need a table of $32 \times 32 \times 6$ bits constant pseudo-random numbers. In our experience, such fixed tables can be realized with very few gates.

6 Conclusion and Future Work

One could argue that all we do in this paper is to implement "stochastic rasterization" (SR)—a 20-year old technique [6, 7]. However, we contribute with several techniques well-suited for GPUs. We develop tight-fitting robust OBBs around moving triangles, and introduce time-dependent edge functions for efficient inside-test and interpolation. In addition, we construct a clever scheme using only four samples per pixel, which gives eight samples (perfectly jittered in time) for pixel reconstruction, and still comply to using 2×2 quads. Furthermore, we create a Zmin/Zmax-culling variant, which is crucial to good performance today. We show that SR can be used for depth-of-field, glossy reflections, and motion blur with shadows, highlights, & reflections. In conclusion, we strongly believe our research advances the field of rasterization.

Even though we think that rasterization and ray tracing are somewhat complementary techniques, there is an ongoing debate about which is *the* preferred rendering algorithm. We have showed that SR is a powerful alternative for motion blur, since we can sample the moving triangles at any particular time. For ray tracing, spatial

data structures need to be partly rebuilt for every instant of time where we want to sample the geometry, and this is expensive and impractical.

For future work, we want to investigate how texture-cache coherent rasterization order can be adapted to the case of SR, and work on depth and color buffer compression. Furthermore, we want to combine SR with delay streams [1] for better culling. It would also be important to analyze shader branch efficiency. Also, when motion blur is used, the acceptable frame rate can, in general, be lower compared to not using motion blur. It would be interesting to see whether this can be used to conserve energy in mobile devices.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research, Vetenskapsrådet, and NVIDIA's Fellowship program. Thanks to Timo Aila & anonymous reviewers for their helpful comments.

A Correction

After the publication of the paper, we have realized that the following statement from Section 3.4 is not always correct:

"z^c_{min} *is computed using the plane equations of the starting and the ending triangles of the TCT. This is conservatively correct as long as the orientations of the starting and ending triangle are the same.*"

Below, we present a counterexample, where the minimal depth for a triangle with linear vertex motion occurs in the interior of the temporal interval. Thus, to conservatively derive the minimal depth, it is not sufficient to only test the plane equations of the moving triangle at $t = 0$ and $t = 1$.

Consider the following moving triangle:

$$\mathbf{p}_0(t) = (0,0,1), \quad \mathbf{p}_1(t) = (-1,t,t+1), \quad \mathbf{p}_2(t) = (0,1,t+1) \tag{7}$$

where $t \in [0,1]$ denotes time. The triangle has a normal $\mathbf{n}(t)$ given by:

$$\mathbf{n}(t) = (\mathbf{p}_1(t) - \mathbf{p}_0(t)) \times (\mathbf{p}_2(t) - \mathbf{p}_0(t)) = (t^2 - t, t, -1) \tag{8}$$

The intersection depth at a certain position $\chi = (x,y,1)$ on screen is a function of t, given by:

$$d(t) = \frac{\mathbf{p}_0(t) \cdot \mathbf{n}(t)}{\chi \cdot \mathbf{n}(t)}. \tag{9}$$

This is the plane-ray intersection of the triangle's plane against the ray from the origin through the point χ. Note that the numerator is the backface criterion for a triangle. With our choice of triangle vertices, the numerator is the constant value -1, indicating that the triangle is frontfacing during the entire motion.

Now, we pick $\chi = (\alpha, 0, 1)$, which results in

$$d(t) = \frac{\mathbf{p}_0(t) \cdot \mathbf{n}(t)}{\chi \cdot \mathbf{n}(t)} = \frac{-1}{\alpha(t^2 - t) - 1}. \tag{10}$$

This rational function has a local minima at $t = 0.5$ if $\alpha > 0$.

The minimum of the depths at the triangle's vertices at the start and end of the motion gives a conservative minimal depth, but can be a coarse approximation for the depth per tile of a large triangle. A more accurate approximation can be obtained by bounding Equation 9 over the exents of the tile.

Bibliography

[1] Timo Aila, Ville Miettinen, and Petri Nordlund. Delay Streams for Graphics Hardware. *ACM Transactions on Graphics,*, 22(3):792–800, 2003.

[2] Tomas Akenine-Möller and Jacob Ström. Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics,*, 22(3):801–808, 2003.

[3] ATI. Radeon X800: High Definition Gaming. ATI Technologies White Paper, 2004.

[4] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen J. Scher Zagier. Frameless Rendering: Double Buffering Considered Harmful. In *Proceedings of ACM SIGGRAPH 94*, pages 175–176, 1994.

[5] Edwin Catmull. An Analytic Visible Surface Algorithm for Independent Pixel Processing. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, pages 109–115, 1984.

[6] Robert L. Cook. Stochastic Sampling in Computer Graphics. *ACM Transactions on Graphics,*, 5(1):51–72, 1986.

[7] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, pages 96–102, 1987.

[8] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, pages 137–145, 1984.

[9] Frank Crow. Shadow Algorithms for Computer Graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, pages 242–248, 1977.

[10] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, pages 21–31, 1988.

[11] Joe Demers. *Depth of Field: A Survey of Techniques*, chapter 23, pages 375–390. GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, 2004.

[12] Paul Diefenbach and Norman Badler. Multi-Pass Pipeline Rendering: Realism for Dynamic Environments. In *Symposium on Interactive 3D Graphics*, pages 59–70, 1997.

[13] Manfred Ernst, Tomas Akenine-Möller, and Henrik Wann Jensen. Interactive Rendering of Caustics using Interpolated Warped Volumes. In *Graphics Interface*, pages 87–96, 2005.

[14] Charles W. Grant. Integrated Analytic Spatial and Temporal Anti-Aliasing for Polyhedra in 4-Space. In *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*, pages 79–84, 1985.

[15] Paul Haeberli and Kurt Akeley. The Accumulation Buffer: Hardware Support for High-Quality Rendering. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, pages 309–318, 1990.

[16] Jon Hasselgren and Tomas Akenine-Möller. An Efficient Multi-View Rasterization Architecture. In *Eurographics Symposium on Rendering*, pages 61–72, 2006.

[17] Jon Hasselgren and Tomas Akenine-Möller. Efficient Depth Buffer Compression. In *Graphics Hardware*, pages 103–110, 2006.

[18] N. Jones and J. Keyser. Real-Time Geometric Motion Blur for a Deforming Polygonal Mesh. In *Computer Graphics International*, pages 26–31, 2001.

[19] Alexander Keller and Wolfgang Heidrich. Interleaved Sampling. In *Eurographics Workshop on Rendering*, pages 269–276, 2001.

[20] Jonathan Korein and Norman Badler. Temporal Anti-Aliasing in Computer Generated Animation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, pages 377–388, 1983.

[21] Samuli Laine and Timo Aila. A Weighted Error Metric and Optimization Method for Antialiasing Patterns. *Computer Graphics Forum,*, 25(1):83–94, 2006.

[22] Tom Lokovic and Eric Veach. Deep Shadow Maps. In *Proceedings of ACM SIGGRAPH 2000*, pages 385–392, 2000.

[23] J. Loviscach. Motion Blur for Textures by Means of Anisotropic Filtering. In *Eurographics Symposium on Rendering*, pages 7–14, 2005.

[24] Michael D. McCool, Chris Wales, and Kevin Moule. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. In *Graphics Hardware*, pages 65–72, 2002.

[25] Steve Morein. ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings*. ACM Press, August 2000.

[26] H. Niederreiter. Point Sets and Sequences with Small Discrepancy. *Monatshefte für Mathematik*, 104(4):273–337, 1987.

[27] Marc Olano and Trey Greer. Triangle Scan Conversion using 2D Homogeneous Coordinates. In *Workshop on Graphics Hardware*, pages 89–95, 1997.

[28] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, pages 17–20. ACM, August 1988.

[29] Clement Shimizu, Amit Shesh, and Baoquan Chen. Hardware Accelerated Motion Blur Generation. Technical Report 05-03, Computer Science Department, University of Minnesota at Twin Cities, 2003.

[30] Peter Shirley. *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois at Urbana Champaign, December 1990.

[31] K. Sung, A. Pearce, and C. Wang. Spatial-Temporal Antialiasing. *IEEE Transactions on Visualization and Computer Graphics,*, 8(2):144–153, 2002.

[32] Daniel Wexler, Larry Gritz, Eric Enderton, and Jonathan Rice. GPU-Accelerated High-Quality Hidden Surface Removal. In *Graphics Hardware*, pages 7–14, 2005.

[33] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (Proceedings of ACM SIGGRAPH 78)*, pages 270–274. ACM Press, 1978.

[34] M. Wloka and R. Zeleznik. Interactive Real-Time Motion Blur. *The Visual Computer,*, 12(6):283–295, 1996.

[35] Matthias M. Wloka, Robert C. Zeleznik, and Timothy Miller. Practically Frameless Rendering. Technical report, Brown University, 1995.

Paper VI

Non-Uniform Fractional Tessellation using Edge Blending

Jacob Munkberg[†] Jon Hasselgren[†]
Masahiro Takatsuka[‡] Tomas Akenine-Möller[†]

[†] Lund University / Intel Corporation [‡] Sydney University

ABSTRACT

We present a technique that modifies the tessellator in current graphics hardware so that the result is a more uniformly distributed tessellation in screen space. For increased flexibility, vertex tessellation weights are introduced. Our results show that the tessellation quality is improved at a moderate cost. To guarantee consistent quality, care must be taken when a triangle intersects the view frustum. Therefore, we introduce an edge interpolation technique for smooth adjustments of the tessellation scheme when camera frustum planes intersect a base triangle. This interpolation technique is a preferred alternative to clipping against the entire view frustum, and avoids creation of many sliver triangles. Edge interpolation also allows specification of a unique interpolation function per triangle edge, and we illustrate the case of Bézier remappings per triangle edge. These techniques allow fine adjustments of GPU-generated tessellation patterns, and avoid dense tessellation outside the camera's view frustum.

A short version of this paper is published as:
Non-Uniform Fractional Tessellation,
Graphics Hardware 2008, pages 41–45

1 Introduction

Recent graphics processors [7, 16] include a tessellation unit, allowing data amplification by tessellating *base triangles* to many smaller triangles in the graphics hardware. This helps lowering the bus traffic from the host computer to the graphics processor, by sending higher level surface representations instead of finely tessellated geometry.

Surface tessellation is a vast area of research, and we will limit the discussion here to work directly related to our approach. The REYES architecture [6] splits the input primitives in eye-space iteratively until they have a size smaller than a certain threshold. Then, these smaller primitives are diced into pixel-sized bilinear patches called *micro-polygons*. The dicing rate is determined by the projected screen-space size of each primitive. This results in an approximately uniform tessellation in screen space. Notice that dicing is performed prior to displacement shading, so there is no guarantee for fully uniform screen-space tessellation, which is similar to the approach we will present.

On current graphics hardware, an input primitive (line, triangle or quad) is tessellated in parameter space and the vertex positions in the generated mesh are determined by a domain shader. This allows approximations of higher order surfaces, such as Beziér patches and subdivision surfaces [3, 12]. It is hard to adapt the tessellation to the final projection on-screen *before* the domain shader, as the shader may move the vertex position arbitrarily. However, micro-triangles closer to the camera should generally be smaller than micro-triangles far away.

To allow for continuous level-of-detail (LOD) without visual "popping" and T-junctions, a fractional tessellation scheme [13], hereafter denoted *regular fractional tessellation*, can be used, where a floating point tessellation factor per edge is provided. An overview of this approach is provided in Section 2.

In this paper, we present a modification of regular fractional tessellation. By using perspective-correct interpolation [1, 10] and complementary *vertex weights*, we obtain an almost uniform tessellation in screen space. We warp the parametric coordinates of each tessellated mesh vertex *before* the domain shader so that the screen-space projection of the tessellation pattern has triangles with as uniform areas as possible. The only assumption is that the domain shader contains a perspective projection transform. This work is an extended version of our earlier work [14]. The new research include tessellation edge blending (Section 4) as the main contribution, along with a more thorough evaluation of the consequences of triangle/frustum plane intersections. Another example of a non-uniform warping function is also presented is Section 5.

There are many published adaptive tessellation techniques that use information of the tessellated surface *after* surface evaluation [2, 4, 5], which achieve higher quality, but with a substantially higher computational cost. Given a graphics card with regular fractional tessellation, our algorithm can be implemented directly as a first step in a domain shader.

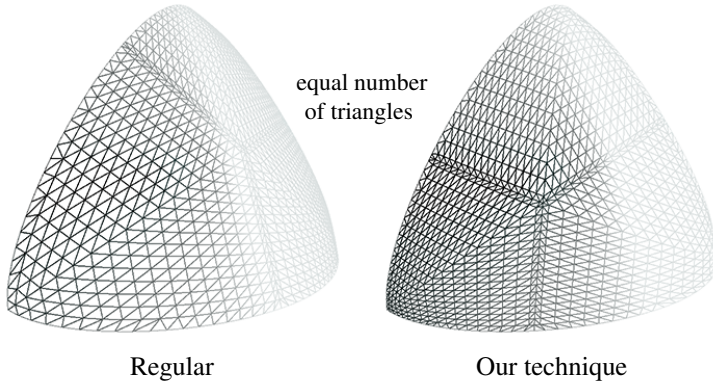


Figure 1: Comparison between tessellation on a PN-displaced triangle [17]. Our algorithm places more vertices (non-uniformly) closer to the camera, which results in more uniform screen-space triangle areas.

For a base triangle with an edge along the view vector, regular fractional tessellation adapts poorly. We adjust the scheme to better distribute the vertices over the triangle, while retaining many of its strong advantages. Figure 1 shows an example of our technique.

2 Regular Fractional Tessellation

Regular fractional tessellation is a continuous tessellation scheme where floating-point weights are assigned to each edge of a triangle. The description in this section is heavily influenced by the original presentation by Moreton [13], but is included here for clarity. To allow for a continuous level of detail, new vertices emerge symmetrically from the center of each edge. Furthermore, vertices must move continuously with respect to the tessellation factors. The scheme consists of one inner, regular part, and a transitional part (the outermost edges). An example of the continuous introduction of new vertices is shown in Figure 2. Each outer edge is divided in two for symmetry, and each half-edge can be treated independently. Given an edge with tessellation factor f, first compute the integer part of f: $n = \lfloor f \rfloor$. Then step n times with a step size $1/f$ (assuming an half-edge length of one), and finally, connect the current vertex with the mid-point of the edge. This allows for efficient surface evaluation schemes, such as forward differencing, which need uniform step sizes. The other half-edge is tessellated symmetrically, resulting in two smaller distances close to the mid-point.

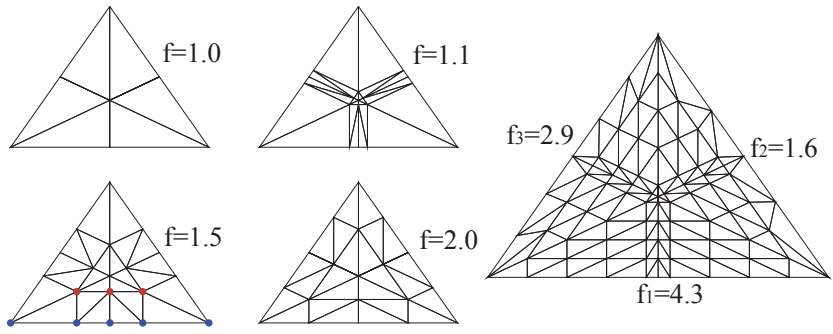


Figure 2: Left: Four regular fractional tessellation examples are shown with a common tessellation factor (f) on all edges, from f=1.0 up to f=2.0. As can be seen in the lower left triangle, for each inner triangle, the number of vertices decreases with two per edge. Right: Each edge of a triangle can also have a unique tessellation factor

.

2.1 Edge tessellation factors

In the uniform case, the edges of an inner triangle have two vertices fewer than the triangle edges one level further out (see Figure 2). In the case of equal tessellation weights, f, on all three edges, the first inner triangle will be regular with a tessellation factor of $f-1$ on all three sides.

In the general setting, each edge has a unique tessellation factor. With different tessellation factors, the symmetric interior and the outermost edges can be connected by a stitching state-machine based on Bresenham's line drawing algorithm (see Moreton's paper [13] for details). Figure 2 illustrates an example triangle with three different edge tessellation factors, f_i. The tessellation factor for the interior part can, for example, be chosen as $\max(f_1, f_2, f_3) - 1$.

The edge tessellation factors can be computed by, for example, projecting each triangle edge on the image plane and computing their screen-space lengths, giving larger weights to edges closer to the camera. This is reasonable, as one strives for having equal area of each generated triangle. For displacement-mapped surfaces, local characteristics of the displacement map, such as heights and normal variations, can also be exploited to determine the tessellation rate [8].

2.2 Fractional Tessellation on Current GPUs

Recent graphics hardware from AMD/ATI supports regular fractional tessellation. In their implementation, the tessellation unit takes vertices and edge tessellation factors of a base triangle as inputs, and generates a set of new vertices. The tessellation unit computes the barycentric coordinates, (u,v), for every created vertex, and executes a *domain shader* [3]. The task of this shader is to compute the position of a vertex as a function of its barycentric coordinates and the three vertices of the base triangle.

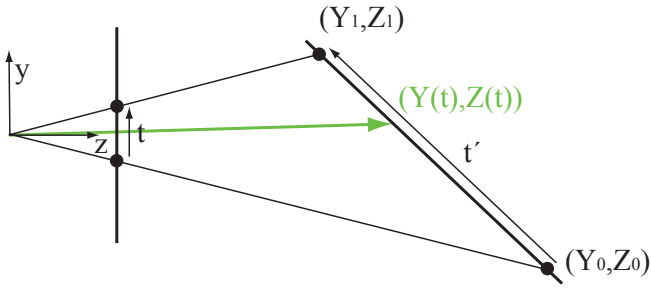


Figure 3: Perspective-correct interpolation.

The edge tessellation factors can be computed either on the CPU, or by adding an additional pass on the GPU and using "render to vertex buffer" capabilities to execute a shader program that computes the factor for each edge.

3 Non-Uniform Fractional Tessellation

A disadvantage of regular fractional tessellation is that vertices along an edge are distributed uniformly (except locally around the center, where new vertices are introduced). If an edge is parallel to the view direction, a uniform tessellation along this edge is far from optimal. Our goal is to create a tessellation pattern that preserves the qualities of regular fractional tessellation, such as continuous level of detail and introduction of new vertices at an existing vertex. In addition, we strive for uniform micro-triangle sizes in screen space before the domain shader is executed, similar to what is done in the REYES architecture [6].

Given a base triangle, we first tessellate using the regular fractional tessellation algorithm as described in Section 2. We then modify the barycentric coordinate of each vertex in the generated tessellation so that its projection in screen space results in uniform micro-triangle sizes. This is achieved by using reverse projection, as described in the following section.

3.1 Reverse Projection

We start with a simple example in two dimensions. Figure 3 shows a line $l = (1-t')(Y_0,Z_0)+t'(Y_1,Z_1)$ in perspective. Let t' denote a parameter along the line in camera space and t a parameter along the projection of the line in screen space. Using similar triangles and linear interpolation in t and t', we can derive a relationship between these parameters as:

$$t' = \frac{t/Z_1}{(1-t)/Z_0 + t/Z_1}. \tag{1}$$

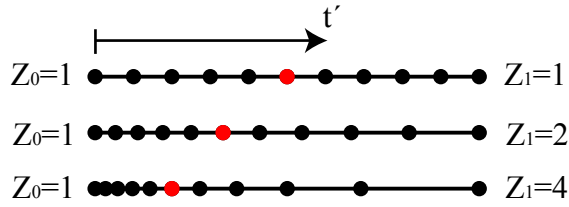


Figure 4: Perspective remapping of a uniform edge in t for three different combinations of vertex depths.

Now, assume we have a uniform distribution of points in t. Figure 4 shows the corresponding distributions in t' for various depth values Z_0 and Z_1. The bigger the depth difference, the more non-uniform distribution in t'. All the distributions t' from Figure 4 will project back to a uniform distribution in screen space, by construction.

Next, this is generalized to two dimensions. Denote the barycentric coordinates of the triangle (u', v'), and the projected barycentric coordinates in screen space as: (u, v). Regular fractional tessellation will create a uniform pattern in the plane of the triangle, but when projected on-screen, this will no longer be uniform. However, assume we have a regular fractional tessellation in screen space, and reverse-project the pattern out on the triangle in camera space. If we know the vertex depths in camera space of our base triangle, we can generalize the derivation above to form the standard perspective-correct barycentric coordinates [1, 10] for triangles:

$$\begin{aligned} u' &= \frac{u/Z_1}{(1-u-v)/Z_0 + u/Z_1 + v/Z_2}, \\ v' &= \frac{v/Z_2}{(1-u-v)/Z_0 + u/Z_1 + v/Z_2}. \end{aligned} \tag{2}$$

These are the barycentric coordinates in camera space that project to a uniform tessellation in screen space. This can also be seen a function that adjust the barycentric coordinates of the triangle (u', v') before projection so that they create a uniform distribution of (u, v) in screen space, using three *vertex weights*, $\{Z_i\}$.

In the GPU pipeline, the domain shader typically receives barycentric coordinates *before* projection as input, and by simply applying Equation 2 to these barycentric coordinates as a first step in the domain shader, the pattern will be roughly uniform in screen space after projection. Note that we need the depth values (in camera space) for each vertex of the base triangle. One approach is to compute these vertex weights in a shader in a preceding pass, similar to how edge tessellation factors are handled in current hardware solutions (see Section 2.2). Another possibility is to compute the depth values in the domain shader (a dot product), just before we perform the reverse projection. This solution avoids sending data between different passes, but performs redundant work.

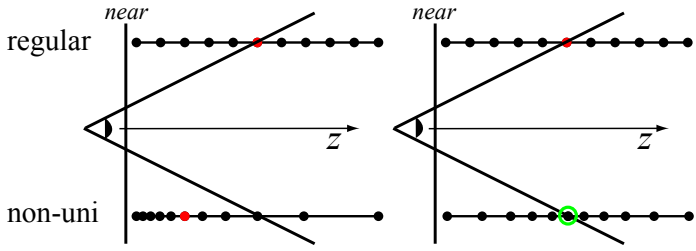


Figure 5: Left: for triangles that are fully in front of the near plane, but straddle another view frustum plane, regular tessellation can be better than our non-uniform algorithm. Right: By clipping the base primitives to the frustum, and update the vertex weights for the clipped triangles, we alleviate this situation.

The same correction technique works for quad primitives by using generalized barycentric coordinates. For example, *mean value coordinates* work as generalized barycentric coordinates for quads. Please refer to Hormann and Tarini's work on quad rendering [11] for details.

Discussion Note that reverse projection gives a (roughly) constant triangle area tessellation in screen space only if the base triangle is not undergoing any transformations other than the projection. In practice, this is not true as subdivision surfaces and displacement mapping are the most common applications of tessellation. However, the resulting tessellation quality is more likely to be better if we start with a uniform tessellation in screen space, even when an arbitrary vertex shader follows.

3.2 Clipping

Our reverse projection is based on perspective-correct interpolation, which means that problems occur when part of a triangle is behind the camera (straddling triangles). The mathematics of the perspective-correct interpolation breaks down as the projected triangle "wraps around" infinity. In most settings, this problem is avoided, as triangles are clipped to the near-plane of the view-frustum. Our algorithm is executed prior to clipping, and must handle this case.

A further complication is that triangles with one or two vertices in front of the near plane, but outside the view frustum will get an unnecessary concentration of vertices outside the view frustum, as shown in the left part of Figure 5. We propose to clip the base triangles against the view frustum (we use Cohen-Sutherland clipping [15]), and split the straddling triangles in smaller triangles entirely on either side of the clip volume. For triangles partly outside the frustum, we compute new weights so that the interpolation distributes triangles closer to the frustum edge. The right-hand-side of Figure 5 shows this. This approach simply updates the ver-

tex weights for each base primitive in the clipping pass, and no detection is needed in the domain shader. Although the clipping is costly, it is only performed on the coarser base geometry in a preceeding shader pass, and usually only a fraction of the base triangles need to execute the inner (expensive) loop of clipping.

We want to stress that the domain shader is not known, and that it may displace the tessellated vertices arbitrarily. For instance, it may move a vertex over the near clipping plane, thereby making it visible. Our mirrored projection is well motivated in that it distributes many vertices around the intersection with the view frustum. Under the assumption that the vertex displacement is reasonably local, it is more likely that a vertex close to a frustum border is moved in front of it, than a vertex further away.

4 Edge Blending

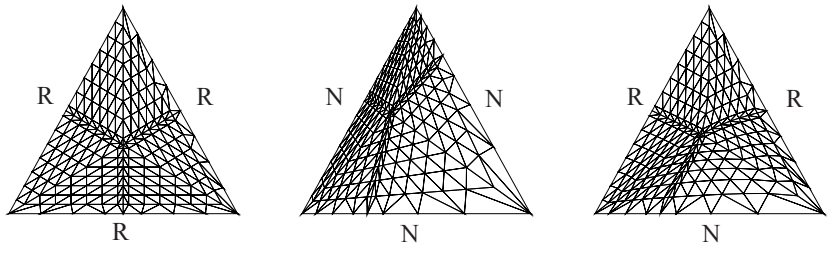


Figure 6: Edge blending on tessellation. From left to right: A triangle with three regular edges resulting in regular fractional tessellation over the whole triangle, a triangle with triangle with three non-uniform edges resulting in our non-uniform tessellation pattern, a triangle with two regular edges and one non-uniform resulting in a smooth blend between the two patterns in the interior of the triangle.

Let us return to the subtle issue presented in Figure 5, where a triangle is entirely in front of the near-plane, but intersects another frustum plane. Although clipping against the entire camera frustum solves this issue, the operation may be costly, and introduces sliver triangles, which may potentially degrade the tessellation quality. In this section, we present an alternative and novel solution, which is based on using regular fractional tessellation for triangle edges straddling a frustum plane, and non-uniform tessellation on all other edges. For this purpose, we introduce a new edge interpolation technique to blend between different edge vertex distribution functions over a triangle. See Figure 6 for an example.

4.1 Edge Interpolation

In order to prevent surface cracks between a base primitive tessellated with the regular fractional tessellation scheme and a base primitive with the non-uniform

fractional tessellation scheme from Section 3, we need a technique that a) allows us to define a vertex distribution per triangle edge, and b) smoothly blends the distributions in the interior of the triangle.

To motivate our approach, we start by looking at how Gouraud shading interpolates three vertex color values $\mathbf{C}_{p_i}$ over the triangle using the barycentric coordinates:

$$\mathbf{C}_{interp} = (1-u-v)\mathbf{C}_{p_0} + u\mathbf{C}_{p_1} + v\mathbf{C}_{p_2}. \tag{3}$$

As can be seen, the color varies linearly between two color values along each edge and is a barycentric combination in the inside of the triangle. This interpolation formula is used heavily in the graphics pipeline to interpolate vertex attributes.

In our case, we will tag each edge as either **R** (using regular fractional tessellation) or **N** (using non-uniform fractional tessellation) depending on whether the edge straddles a frustum plane. For a consistent result, we thus need to be able to interpolate smoothly inside the triangle with different "tessellation tags" per edge. For edge blending, we need an interpolation method that is constant along each edge and varies smoothly inside the triangle. Our strategy is to base the new edge interpolationon on barycentric coordinates, (u,v,w), and we define three new interpolation coordinates, (α,β,γ). We want $\alpha = 1$ on the edge where $u = 0$, so we make α proportional to $1-u$. Also, to ensure that β and γ are zero on the edge $u = 0$, we make them both proportional to u. Taking this into consideration for all three edges, we arrive at the following formulae:

$$\begin{aligned} \alpha &= (1-u)vw \\ \beta &= u(1-v)w \\ \gamma &= uv(1-w). \end{aligned} \tag{4}$$

These variables lead to the following interpolation formula, which is constant along edges (except for at the corners), and can be used to interpolate edge attributes.

$$\mathbf{C}_{interp} = \frac{\alpha\mathbf{C}_{e_1} + \beta\mathbf{C}_{e_2} + \gamma\mathbf{C}_{e_3}}{\alpha+\beta+\gamma} \tag{5}$$

With $\mathbf{C}_{e_1} = (1,0,0)$, $\mathbf{C}_{e_2} = (0,1,0)$, and $\mathbf{C}_{e_3} = (0,0,1)$, we get a color blend as illustrated by Figure 7.

Now, we apply this technique to our tessellation schemes. As shown in Figure 6, each edge of the triangle is tagged as either regular or non-uniform. Given regular barycentric coordinates (u,v) and non-uniform ones (u',v'), we blend between them in the interior of the triangle using a formula similar to Equation 4 above. If the first two edges are regular (use (u,v)), and the third edge non-uniform (uses (u',v')), we modify the barycentric coordinates as follows:

$$\begin{aligned} u_{interp} &= \frac{\alpha u + \beta u + \gamma u'}{\alpha+\beta+\gamma} \\ v_{interp} &= \frac{\alpha v + \beta v + \gamma v'}{\alpha+\beta+\gamma}, \end{aligned} \tag{6}$$

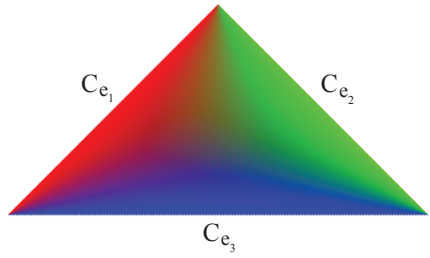


Figure 7: Edge blending. We want a constant color along each edge of the triangle and a smooth blend in the interior.

and the scheme in the interior is warped smoothly to enforce the constraints of the edges. The rightmost part of Figure 6 shows the resulting tessellation pattern.

4.2 Smooth Edge Transitions

Finally, we want to smoothly introduce this transition when a triangle intersects a frustum plane to avoid a discrete change in the tessellation pattern (popping). This is performed by introducing an additional blend when edges start intersecting the frustum and smoothly transform from non-uniform fractional tessellation to regular fractional tessellation. At the edge which intersects the frustum plane, we compute the barycentric coordinate of the intersection point, and use a *smoothstep* function to blend between the regular and non-uniform pattern for that edge, *prior* to the edge interpolation described above.

Given a parameter $x \in [0,1]$ along the edge, and a transition zone w in which we want to blend, the interpolation kernel is simply a smoothstep function:

$$h(x) = \begin{cases} 3(\frac{x}{w})^2 - 2(\frac{x}{w})^3 & x \leq w \\ 1 & x > w \end{cases} \tag{7}$$

Figure 8 illustrates this in two dimensions.

In practice, for an edge fully inside or outside the camera frustum, the choice of tessellation scheme per edge is binary; either (u,v) or (u',v'), as discussed in Section 4.1 (Equation 6). However, for an edge that intersects a frustum plane, the choice of tessellation scheme is a smooth blend:

$$(u_b, v_b) = (1 - h(x))(u',v') + h(x)(u,v), \tag{8}$$

and it is (u_b, v_b) that are fed into Equation 6 for that edge. Figure 9 shows eight frames from an animation where a frustum plane intersects a couple of triangles, and the transition to regular fractional tessellation is introduced smoothly.

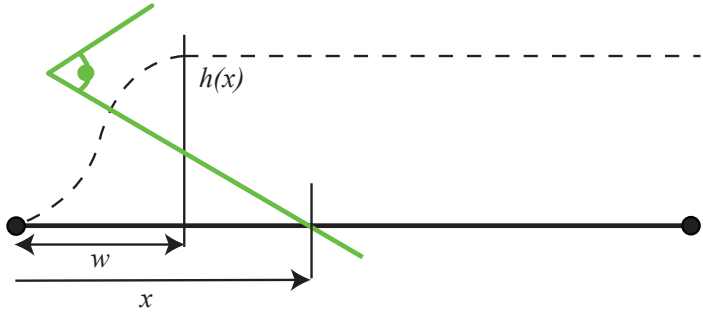


Figure 8: A frustum plane (green) is intersecting a triangle edge (bold black). Based on the parametric coordinate $x \in [0,1]$ along the edge, we apply a smooth step-funtion $h(x) \in [0,1]$, so that in the transition zone, $x \in [0,w]$, $h(x)$ specify a smooth blending weight. Thus, as the triangle edge intersect the frustum plane, the edge will transform smoothly from non-uniform to regular fractional tessellation.

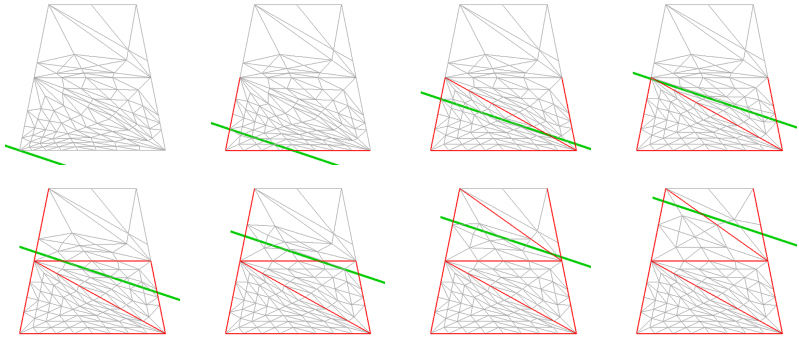


Figure 9: Eight frames that show the gradual blending between uniform and non-uniform tessellation when four base triangles are intersected by a frustum plane. The camera starts in the lower left corner and moves diagonally upwards. The frustum intersection line is marked with green. All triangle edges partly or fully outside the frustum are marked with red. Please note that as the frustum plane moves upwards, all intersected edges smoothly becomes uniform, without introducing any cracks or t-junctions

4.3 Frustum Intersections

In a pre-pass, preferably when the tessellation factors for each edge of the base primitive edge are determined, we also test if a triangle edge intersects any of the camera view frustum planes. In Figure 8, the distance to an intersection along the edge is marked with x, and x is used for smooth edge transitions, as described in the previous section. For clarity of presentation, we have up until now only described the case of one frustum plane intersecting the base triangle. However, as illustrated in Figure 10, a triangle may intersect several frustum planes, and to

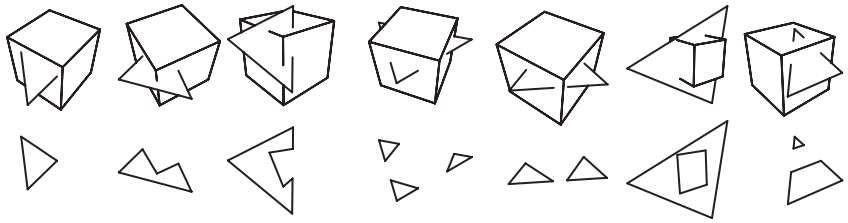


Figure 10: Triangle-frustum intersections in clip space. As can be seen, each triangle edge has at most two intersections with the frustum.

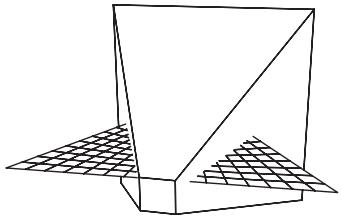


Figure 11: A highly tessellated triangle intersecting the camera frustum. As can be seen in the figure, most of the generated triangles end up outside the camera frustum. In order to guarantee high quality close to the camera, the triangle must be highly tessellated. As a result, many unnecessary triangles will be generated outside the view frustum

handle all cases, we store the *fraction of the edge outside the frustum*, (f) for each edge, instead of the distance to an intersection. A fraction $f = 0$ means that the edge is inside the frustum, $f \in [0,1]$ means that the edge intersects the frustum once or twice, and a $f = 1$ means fully outside the frustum. If the triangle moves continuously, so will the fractions, and we use f (in place of x) as parameter in Equation 7.

5 Bézier Edge Interpolation

In this section, we present an alternative remapping approach that defines a unique vertex distribution function per base triangle edge. There are cases where a perspective remapping, as described in Section 3, would not alleviate the situation. One such case is illustrated in Figure 11, where a distribution function that can gather vertices around a point along a triangle edge, would help. That would push many generated triangles inside the view frustum, where they are more useful. Below, we present a flexible algorithm that uses constrained Bézier curves, defined per edge and blended together, to address these cases.

We use a third order Bézier curve to remap the vertex distributions along each edge, as shown in Figure 12. The Bézier curve goes through $\mathbf{p}_0 = (0,0)$ and $\mathbf{p}_3 =$

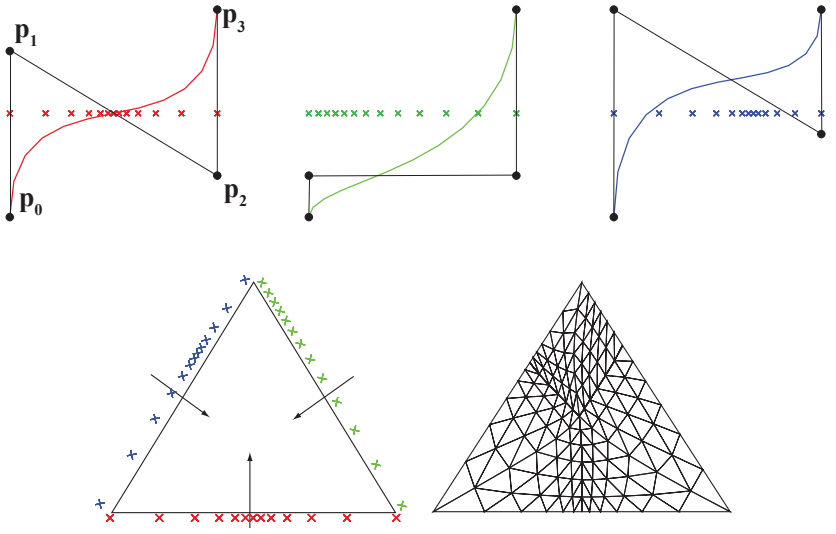


Figure 12: Each edge can specify a unique Bézier curve with two degrees of freedom: the y-coordinates of $\mathbf{p}_1$ *and* $\mathbf{p}_2$*. The image shows three different curves and the remapping when applied to a uniform distribution in* $[0,1]$*. The curves are used to remap the distribution along each edge, as shown in the rightmost figure, and edge interpolation is used to blend between the distributions in the interior of the triangle.*

$(1,1)$ and remaps all the points in between. We allow for two degrees of freedom, $y_1, y_2 \in [0,1]$ and choose the two remaining control points as $\mathbf{p}_1 = (0, y_1)$ and $\mathbf{p}_2 = (1, y_2)$.

A third order Bézier curve is given by:

$$\mathbf{b}(t) = (1-t)^3\mathbf{p}_0 + 3(1-t)^2 t\mathbf{p}_1 + 3(1-t)t^2\mathbf{p}_2 + t^3\mathbf{p}_3. \tag{9}$$

We are interested in the y-component of this curve, and denote it $b(t)$ (to simplify the notation in this chapter). Given our values of $\{\mathbf{p}_i\}, i \in 0\ldots3$, $b(t)$ can be written as:

$$b(t) \equiv b_y(t) = 3(1-t)^2 t\ y_1 + 3(1-t)t^2\ y_2 + t^3. \tag{10}$$

Note that $b(t)$ must be monotonically increasing for $t \in [0,1]$, to avoid reordering of vertices along the edge, so we constrain y_1 and y_2 to $y_1, y_2 \in [0,1]$. Thus, a uniform distribution $t \in [0,1]$ is warped to $t' = b(t) \in [0,1]$. This allows us, with only two parameters per triangle edge, to define a set of useful distributions. Examples are shown in Figure 12.

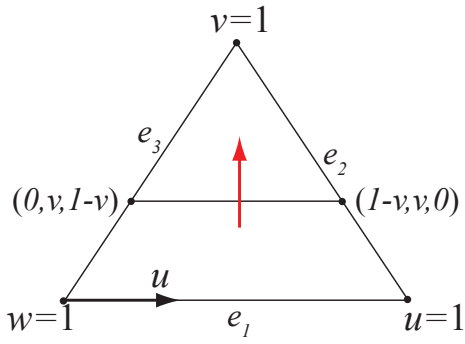


Figure 13: The geometry for a sweep in the positive v-direction

5.1 Blending Edge Functions

Each Bézier curve is specified per edge and should decline as we move away from the edge. Given a triangle with standard barycentric coordinates (u,v) as in Figure 13, let us look at the edge e_1, where the barycentric coordinate $v = 0$. Assume also that we have defined a Bézier curve, as described above, along this edge. We denote this curve $b_{e_1}(t)$. As parameter along edge e_1, we choose u, which goes from zero to one along the edge. The remapped u-coordinate is thus $u' = b_{e_1}(u)$. If we move a line perpendicular from the edge into the triangle in the increasing v-direction, as shown in Figure 13, the interval in u shrinks to $u \in [0, 1-v]$ and we adjust the parameter so that the start and end points of the interval in u still maps to zero and one respectively. We also scale the amplitude so that it fades linearly to zero as we approach $v = 1$. This gives us:

$$u'_{e_1} = (1-v)b_{e_1}(\frac{u}{1-v}), \quad v'_{e_1} = v, \quad \text{and} \quad w'_{e_1} = (1-v)(1-b_{e_1}(\frac{u}{1-v})), \qquad (11)$$

ensuring that the curve has maximum influence on the edge e_1 and smoothly declines as we approach v=1. The same procedure is applied to the edges e_2 and e_3, by permutations of the barycentric coordinates. Finally, the three edge remappings are blended together using Equation 5. This interpolation scheme is similar to the derivation of triangular Gregory patches [9].

Given $b_{e_1}(u), b_{e_2}(v)$ and $b_{e_3}(w)$ defined on the edges e_1, e_2 and e_3 respectively, the remapped barycentric coordinates (u_{interp}, v_{interp}) are:

$$\begin{aligned} u_{interp} &= \frac{\alpha u + \beta(1-v)b_{e_1}(\frac{u}{1-v}) + \gamma(1-w)(1-b_{e_2}(\frac{v}{1-w}))}{\alpha+\beta+\gamma} \\ v_{interp} &= \frac{\alpha(1-u)(1-b_{e_3}(\frac{w}{1-u})) + \beta v + \gamma(1-w)b_{e_2}(\frac{v}{1-w})}{\alpha+\beta+\gamma}. \end{aligned}$$

Discussion The Bézier edge remapping gives more freedom in selecting edge distributions with two parameters per edge. Even more control can be given by allowing $\mathbf{p}_1$ and $\mathbf{p}_2$ to move also in the x-direction, or raise the degree of the Bézier curve, but this means more storage cost per base triangle edge and a higher shader evaluation cost.

It is also possible to replace the Bézier edge curves with other mathematical functions. We have tried a power function as a modified `gain` function, $g(t)$ with two parameters c and n:

$$g(t) = \begin{cases} c(\frac{t}{c})^n & t \leq c \\ 1-(1-c)(\frac{1-t}{1-c})^n & t > c \end{cases} \tag{12}$$

This curve allows us to set a point along the edge of interest (c) and determine the slopes around this point by adjusting the exponent n. However, in our tests, this curve is harder to control, and tend to warp the tessellation pattern more aggressively compared to the Bézier remapping.

6 Implementation

Our algorithm can be implemented in hardware, as well as in shader code. Current fractional tessellation hardware already feeds barycentric coordinates to the domain shader. We can essentially just insert code for our reverse projection algorithm in the beginning of the domain shader to compute new barycentric coordinates. These coordinates are fed to the remainder of the domain shader, which may differ depending on the application.

In our implementation, we perform regular fractional tessellation on the CPU. This could have been done by recent GPUs, but currently there are no public APIs for using the tessellation unit. To estimate the cost of reverse projection, we modified a standard vertex shader, inspired by the evaluation shader approach by AMD [7, 16]. The inputs to the vertex shader are: a) the positions of all three vertices of the base triangle, b) the barycentric coordinates of the current tessellated vertex, and c) vertex weights for all three vertices (typically camera space depth values). Given this setup, our reverse projection code compiles to 11 vertex shader assembly instructions. By comparison, an extremely simple vertex shader that interpolates a single position attribute and transforms it to clip space, compiles to 9 instructions. Edge interpolation and smooth edge transitions use an additional 22 instructions, and needs three additional attributes (blend factors between regular and non-uniform distributions along the three edges) so in total, our complete algorithm compiles to about 33 vertex shader instructions, if frustum intersections should be handled.

Bézier Edge Interpolation comes with a higher cost. Our implementation, using third order Bézier curves with two parameters per edge, as outlined in Section 5, compiles to 55 vertex shader instructions.

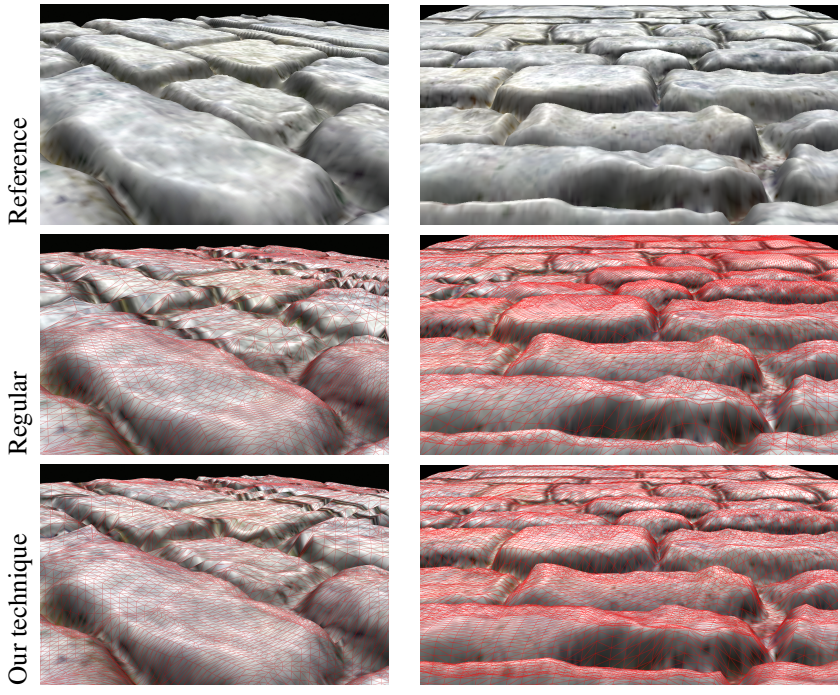


Figure 14: Brick road test scene. We use low tessellation to stress the algorithms. As can be seen, the triangle density is more uniformly spread out in screen space, and the close-up detail is better preserved. This can especially be seen in the front of the images where the bricks have a smoother look with our remapping, and in the far back where more triangles are gathered for regular fractional tessellation.

7 Results

Figure 14 shows regular fractional tessellation and perspective remapping for a displaced brick road. For our technique, the triangle density is more uniformly spread out in screen space, and the close-up detail is better preserved. Similar effects are shown in Figure 1, where a Bézier-triangle has been generated in a domain shader. These images use exactly the same number of micro-triangles. However, as can be seen, the micro-triangles are more uniform in terms of projected micro-triangle area with our technique, which was our goal.

One potential problem is vertex "swimming" during animation, as we warp the parametric space. However, this is true for *any* scheme using fractional tessellation with tessellation weights computed per frame. In practice, we found that our scheme shows about the same or less swimming artifacts compared to regular adaptive fractional tessellation. The accompanying video compares these artifacts

during animation.

Both edge interpolation and clipping handle the frustum intersection problem. The clipping pre-pass is more costly, but the quality can be fine-tuned close to the camera. However, sliver triangles will likely be a big issue in practice. The attached video shows an comparison between clipping and edge interpolation for a simple case.

8 Conclusions and Future Work

The warping technique presented here is not limited to perspective-correction. It should be seen as a more general approach to achieve better control over surface tessellation, where edge and/or vertex weight can be combined to give fine controls over tessellation patterns. Bézier Edge Interpolation in Section 5 is a more flexible example of edge interpolation with a unique distribution function per edge. It further handles near-plane intersections more gracefully (no perspective divide), but comes at a higher cost.

It should also be noted that the warping techniques here are not limited to fractional tessellation patterns, but can be applied to any tessellation schemes that specify vertex positions using barycentric coordinates.

As future work, it would be interesting to investigate more elaborate LOD measures for the edge and vertex weights. We hope that this paper will stimulate further research in the field.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research and Intel Corporation. In addition, Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation.

Bibliography

[1] Jim Blinn. Hyperbolic Interpolation. *IEEE Computer Graphics and Applications,*, 11(1):89–94, 1991.

[2] M. Bóo, M. Amor, M. Doggett, J. Hirche, and W. Strasser. Hardware support for adaptive subdivision surface rendering. In *Graphics Hardware*, pages 33–40, 2001.

[3] Ignacio Castano. Tessellation of Displaced Subdivision Surfaces in DX11. Gamefest, 2008.

[4] Jatin Chhugani and Subodh Kumar. View-dependent adaptive tessellation of spline surfaces. In *Symposium on Interactive 3D graphics*, pages 59–62, 2001.

[5] Kyusik Chung and Lee-Sup Kim. Adaptive Tessellation of PN Triangle with Modified Bresenham Algorithm. In *SOC Design Conference*, pages 102–113, 2003.

[6] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, pages 96–102, 1987.

[7] Michael Doggett. Xenos: XBOX 360 GPU. Eurographics presentation, September 2005.

[8] Michael Doggett and Johannes Hirche. Adaptive View Dependent Tessellation of Displacement Maps. In *Graphics Hardware*, pages 59–66, 2000.

[9] John A. Gregory. Smooth Interpolation without Twist Constraints. In *Computer Aided Geometric Design*, pages 71–87, 1974.

[10] Paul S. Heckbert and Henry Moreton. Interpolation for Polygon Texture Mapping and Shading. In *State of the Art in Computer Graphics: Visualization and Modeling*, pages 101–111, 1991.

[11] Kai Hormann and Marco Tarini. A Quadrilateral Rendering Primitive. In *Graphics Hardware*, pages 7–14, 2004.

[12] Charles Loop and Scott Schaefer. Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. Technical report, MSR-TR-2007-44, Microsoft Research, 2007.

[13] Henry Moreton. Watertight Tessellation using Forward Differencing. In *Graphics Hardware*, pages 25–32, 2001.

[14] Jacob Munkberg, Jon Hasselgren, and Tomas Akenine-Möller. Non-Uniform Fractional Tessellation. In *Graphics Hardware*, pages 41–45, 2008.

[15] W. Newman and R. Sproull. *Principles of Interactive Computer Graphics*. New York: McGraw-Hill, 2nd edition, 1979.

[16] Natalya Tatarchuk, Christopher Oat, Jason L. Mitchell, Chris Green, Johan Andersson, Martin Mittring, Shanon Drone, and Nico Galoppo. Advanced Real-Time Rendering in 3D Graphics and Games. SIGGRAPH course, 2007.

[17] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. Curved PN triangles. In *Symposium on Interactive 3D graphics*, pages 159–166, 2001.

Paper VII

Automatic Pre-Tessellation Culling

Jon Hasselgren Jacob Munkberg Tomas Akenine-Möller

Lund University

ABSTRACT

Graphics processing units supporting tessellation of curved surfaces with displacement mapping exist today. Still, to our knowledge, culling only occurs *after* tessellation, i.e., after the base primitives have been tessellated into triangles. We introduce an algorithm for *automatically* computing tight positional and normal bounds on the fly for a base primitive. These bounds are derived from an arbitrary vertex shader program, which may include a curved surface evaluation and different types of displacements, for example. The obtained bounds are used for backface, view frustum, and occlusion culling *before* tessellation. For highly tessellated scenes, we show that up to 80% of the vertex shader instructions can be avoided, which implies an "instruction speedup" of 5×. Our technique can also be used for offline software rendering.

ACM Transactions on Graphics, 28(2):19, 2009.

1 Introduction

To provide rich surface representations for real-time rendering, it is expected that most graphics hardware in the near future will have support for tessellation of curved surfaces with displacement mapping. The Xbox 360 [5] and the ATI Radeon HD 2000 series [25] already have support for this. A primitive with a triangular or square domain is tessellated, and barycentric coordinates are forwarded to the vertex shader, which may compute an arbitrary position based on these coordinates, and more. To the best of our knowledge, these systems only perform culling *after* tessellation using the conventional graphics pipeline. Clearly, it would be advantageous to be able to cull *before* tessellation occurs, and Figure 1 shows an example of this.

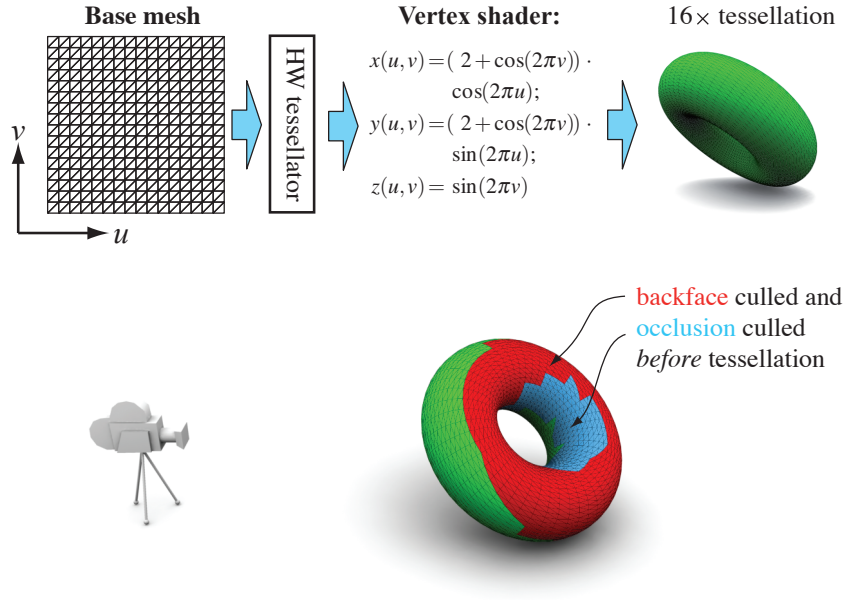


Figure 1: GPUs with tessellation hardware are given a base mesh over a parameter space, (u,v), as input. In this case, the tessellator increases the number of triangles by a factor of sixteen, and a vertex shader evaluates a point on a torus surface. In the lower part, we visualize the base triangles that our culling algorithm automatically *can avoid to tessellate, and where vertex shader evaluations can be avoided. We are able to cull 56% of the triangles prior to tessellation.*

Over the years, culling techniques have seen many uses in both real-time graphics and offline rendering. In general, RenderMan implementations [1, 4] use culling on many different levels. However, the details may vary for different implementations. View frustum and occlusion culling are performed, often prior to tessellation, and splitting of primitives may also occur. Backface culling is usually done after tessellation. Wexler et al. [27] describe a GPU-optimized implementation, where (among other things) occlusion queries are used to accelerate rendering. However, to bound a displaced surface in RenderMan, the user either has to provide the renderer with a conservative upper bound, or the displacement shader is executed on micropolygons, and exact bounds computed from these [1]. In this latter case, no culling occurs before tessellation.

Shirman and Abi-Ezzi [24] use cones to bound a set of normals on a patch, and can thus perform efficient backface culling. Kumar and Manocha [15] derive a different method for backface culling of curved surfaces, and use a conservative technique to bound the normals and then test for culling. However, neither of these techniques can handle arbitrary surface evaluations automatically on the fly. Han et al. [10] describe an alternative GPU implementation, where the part of the vertex shader that computes the position of a vertex is executed first. After that follows backface culling. If the triangle is culled then unnecessary lighting calculations are avoided. Our goal is similar, but we want to perform culling before tessellation even occurs.

There is a wealth of literature on adaptive on-the-fly tessellation, and as our work can be combined with such techniques, we only list some of them. Doggett and Hirche [6] use a summed-area table of the displacement map and a normal test to guide the tessellation. A similar approach is to use interval arithmetic and interval textures to focus the tessellation efforts [22]. To provide a continuous level of detail, Moreton [21] introduces *fractional tessellation* where tessellation factors are specified as floating-point numbers per triangle edge. This allows for adaptive tessellation across a mesh, and similar techniques are used in modern GPUs [25].

In contrast to the previous work described above, we focus on presenting a single automatic solution. Our paper contributes with a novel pre-tessellation backface, view frustum, and occlusion culling method which is:

- fully automatic, based only on arbitrary vertex shader code which can include for example deformations, curved surfaces, and displacement mapping.
- implemented with tightly bounded arithmetic on triangular domains.
- suitable for implementing in both hardware and software rendering systems.

Next, we describe our algorithm in detail.

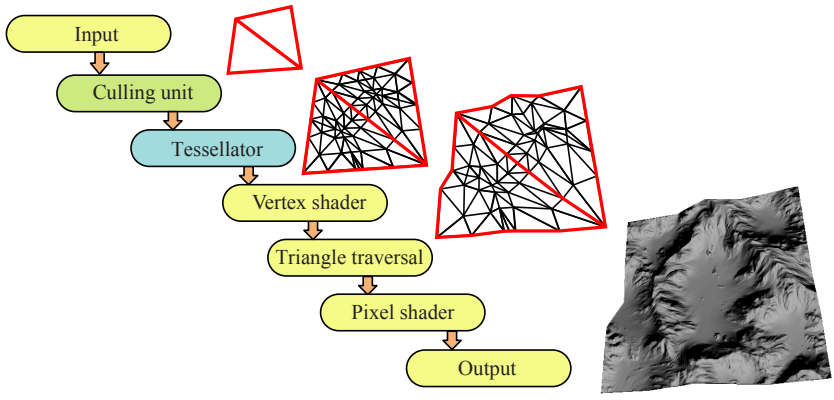


Figure 2: To support tessellation in the GPU pipeline, a tessellation unit has recently been added. We propose to add the culling unit, which automatically determines whether tessellation of a base primitive can be avoided.

2 Tessellation Culling

The goal of our work is to efficiently avoid tessellating the majority of surfaces which do not contribute to the final image. This occurs when a surface is back-facing, outside the view frustum, or occluded by previously rendered surfaces. Furthermore, we believe it is of utmost importance that fully *arbitrary* vertex displacement shaders can be handled in a completely *automatic* way. In this section, we present a novel algorithm for this. Without loss of generality, we restrict ourselves to triangular domains and tessellation.

2.1 Overview

We extend the current GPU tessellation pipeline [25] with our new culling unit as illustrated in Figure 2. Note that this type of pipeline is also rather similar to offline rasterization pipelines. Without our culling unit, *base triangles* are first injected into the pipeline, and these can be tessellated to a desired number of triangles by the hardware. For each created vertex, the tessellator forwards its barycentric coordinates, (u,v), down the pipeline. The vertex shader then computes the position, $\mathbf{p}(u,v)$, of each vertex as a function of its barycentric coordinates. This may include, e.g., the evaluation of a Bézier triangle with texture displacement, procedural noise, and transform matrices. Each term can also depend on a time parameter, in order to animate a water surface, for example.

Our culling algorithm works as outlined in Fig. 3. First, we analyze the vertex shader program and isolate all instructions that are used to compute the vertex position. We then compute geometric bounds for this position over an entire base

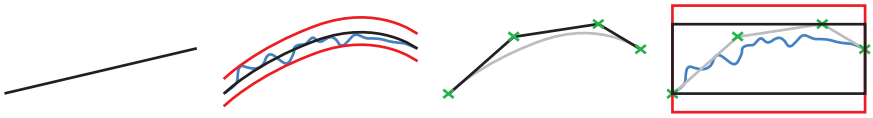


Figure 3: Algorithm overview: ***a)*** *A base triangle (seen from the side) with pre-computed tessellation factors is sent to the tessellation unit.* ***b)*** *By expressing the vertex program in Taylor form (polynomial + interval remainder), a conservative estimate of the surface is obtained.* ***c)*** *The Taylor polynomial is expanded in Bernstein form for efficient range bounding (using the convex hull property),* ***d)*** *Finally, by adding the interval remainder term from the Taylor model to the Bernstein bounds, conservative surface bounds (red) are obtained.*

triangle, and use these bounds to perform the culling.

Recently, it has been shown [12] that pixel shaders can be executed, bounded, and culled over a block of pixels using interval arithmetic [19]. In this case, the programs used for culling are often short (terminated by a `KIL` instruction). However, in our context, the shader programs are significantly more complex, and therefore we use Taylor models [2] to approximate the shader function over the triangle domain. We then use Bernstein expansion [14] to compute tight bounding boxes for the Taylor models, and use these bounding boxes for culling.

In the following, we first present some background on Taylor models in Section 2.2. Then follows an algorithm for computing tight polynomial bounds in Section 2.3, and our program analysis and generation in Section 2.4. In Section 2.5, we describe how selective execution of our culling can be done, and finally, the culling algorithms are described in Section 2.6.

2.2 Taylor Arithmetic

Taylor arithmetic has seen little use in computer graphics research, but there is a recent exception in collision detection [28]. Interval arithmetic [19], on the other hand, has been used extensively in graphics. Intervals are used in Taylor models, and the following notation is used for an interval $\hat{a}$:

$$\hat{a} = [\underline{a}, \overline{a}] = \{x \,|\, \underline{a} \leq x \leq \overline{a}\}. \tag{1}$$

Given an $n+1$ times differentiable function, $f(u)$, where $u \in [u_0, u_1]$, the Taylor model of f is composed of a Taylor polynomial, T_f, and an interval remainder term, $\hat{r}_f$ [2]. An nth order Taylor model, here denoted $\tilde{f}$, over the domain $u \in [u_0, u_1]$ is then:

$$\tilde{f}(u) = \underbrace{\sum_{k=0}^{n} \frac{f^{(k)}(u_0)}{k!} \cdot (u-u_0)^k}_{T_f} + \underbrace{[\underline{r_f}, \overline{r_f}]}_{\hat{r}_f} = \sum_{k=0}^{n} c_k u^k + \hat{r}_f. \tag{2}$$

This representation is called a Taylor model, and is a conservative enclosure of the function, f over the domain $u \in [u_0, u_1]$.

Similarly to interval arithmetic, it is also possible to define arithmetic operators on Taylor models, where the result is a conservative enclosure (another Taylor model) as well [2]. Addition is defined as follows: Assume that $f+g$ shall be computed and these functions are represented as Taylor models, $\tilde{f} = T_f + \hat{r}_f$ and $\tilde{g} = T_g + \hat{r}_g$. The Taylor model of the sum is then

$$\widetilde{f+g} = (T_f + T_g) + (\hat{r}_f + \hat{r}_g). \tag{3}$$

Note here that $T_f + T_g$ is an addition of two polynomials.

Similarly, for multiplication of a Taylor model, $\tilde{f}$, by a scalar value, λ, we get that:

$$\widetilde{\lambda \cdot f} = (\lambda \cdot T_f) + (\lambda \cdot \hat{r}_f). \tag{4}$$

Multiplication between two Taylor models is more complicated. Assume again that we want to compute $f \cdot g$ where f and g are represented by Taylor models. The Taylor model of the product is then

$$\widetilde{f \cdot g} = \underbrace{\underline{T_f \cdot T_g}}_{T_{f \cdot g}} + \underbrace{B\left(\overline{T_f \cdot T_g}\right) + B\left(T_f\right) \cdot \hat{r}_g + B\left(T_g\right) \cdot \hat{r}_f + \hat{r}_f \cdot \hat{r}_g}_{\hat{r}_{f \cdot g}}. \tag{5}$$

The polynomial part of this equation, $T_{f \cdot g}$ is simply the multiplication of the polynomials T_f and T_g, but clamped (denoted $\underline{T_f \cdot T_g}$) so that all terms of higher order than the Taylor model has been removed.

The remainder has several contributing terms. First, we have the part of the polynomial multiplication that overflows and has terms only of higher order than the Taylor model ($\overline{T_f \cdot T_g} = T_f \cdot T_g - \underline{T_f \cdot T_g}$). Note that we want the remainder term on interval form, and therefore we must bound the overflow of the polynomial multiplication over the domain (this is indicated by the bounding operator, $B()$). To compute the bounds, we directly evaluate overflowing terms using interval arithmetic and accumulate them to the remainder. More complex bounding computations, such as the one presented in Section 2.3, are possible, but since multiplication is such a frequent operation, we must ensure that it is fast to compute its bounds. The other terms found in the remainder involve computing the bounds of T_f and T_g and are treated similarly to the overflow from the polynomial multiplication. It should be noted that one or more of the terms in the remainder often are zero. For instance, if $\hat{r}_f$ or $\hat{r}_g$ is zero, then the corresponding terms will be zero as well. As an optimization, we detect these cases and avoid the computations.

By using Taylor expansion, and the addition and multiplication operations presented above we can derive more complex arithmetic operators, like sine, log, exp, reciprocal, and so on. We refer to the work of Berz and Hoffstätter [2] and Makino and Berz [18] for more details.

Motivation The motivation for us to use Taylor models is that curved surfaces and subdivision schemes are often based on polynomials. Polynomial computations can be represented exactly by Taylor models (provided they are of high enough order) which leads to very tight bounds. Previous work on shader analysis [7, 13, 12] have successfully used interval and affine arithmetic, which are computationally less expensive than Taylor models. However, note that they subdivide the domain into small tiles before evaluating the bounded shader. In contrast, we must bound the shader over the entire domain (the base triangle) in a *single* evaluation, and consequently we need much tighter bounds. A side by side comparison between the tightness of interval arithmetic, affine arithmetic and Taylor models can be found in the example in Section 2.3.

Taylor models also provide a flexible framework since it is essentially a superset of interval and affine arithmetics. It allows us to tweak interval sharpness versus computational overhead by changing the order of the Taylor model. Orders zero and one correspond to interval arithmetics, and generalized interval arithmetics [11], which is similar to affine arithmetics.

2.3 Tight Polynomial Bounds

Our approach to tessellation culling is to evaluate the vertex shader using Taylor arithmetic as described above. We execute the part of the shader that affects the position attribute using Taylor arithmetic. This results in a Taylor model for each of the components in the position attribute: (x, y, z, w). To find a geometrical bounding box, one could then find local minima and maxima for each of these. However, this requires numerical, iterative methods for polynomials of degree $n > 4$, and also quickly becomes impractical due to the dependence on the two parametric coordinates, (u, v).

Instead, we use a faster, conservative approach which still produces tight bounds. The resulting Taylor polynomials are in power form, and the core idea is to convert these to Bernstein form. The convex hull property of the Bernstein basis guarantees that the actual surface or curve of the polynomial lies inside the convex hull of the control points. Thus, we compute a bounding box by finding the minimum and maximum control point value in each dimension.

In practice, we obtain bivariate polynomials from the vertex shader evaluation using Taylor arithmetic, and for a single component (e.g., x), this can be expressed in the power basis as follows (where we have omitted the remainder term, $\hat{r}_f$, for clarity):

$$p(u,v) = \sum_{i+j \le n} c_{ij} u^i v^j. \tag{6}$$

We want to transform Equation 6 into the Bernstein basis:

$$p(u,v) = \sum_{i+j \le n} p_{ij} B_{ij}^n(u,v), \tag{7}$$

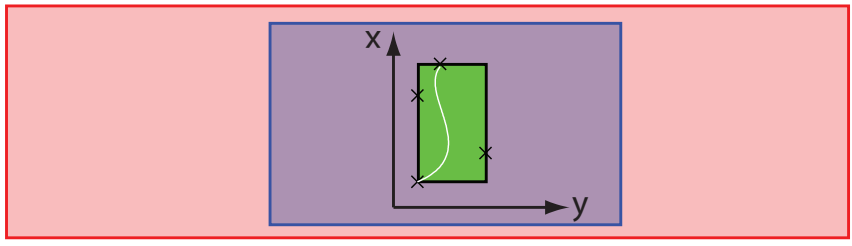


Figure 4: A comparison of the bounds for a parametric curve $(p_x(t), p_y(t))$ of degree 3 in t for interval aritmethic (red), affine arithmetic (blue) and Taylor models with Bernstein bounds (green).

where $B_{ij}^n(u,v) = \binom{n}{i}\binom{n-i}{j}u^i v^j (1-u-v)^{n-i-j}$ are the Bernstein polynomials in the bivariate case over a triangular domain. We can convert a polynomial in the power basis form, into the Bernstein form using the following formula [14]:

$$p_{ij} = \sum_{l=0}^{i}\sum_{m=0}^{j} \frac{\binom{i}{l}\binom{j}{m}}{\binom{n}{l}\binom{n-l}{m}} c_{lm}. \tag{8}$$

To compute a bounding box, we simply compute the minimum and the maximum value over all p_{ij} for each dimension, x, y, z, and w. This gives us a bounding box, $\hat{\mathbf{b}} = (\hat{b}_x, \hat{b}_y, \hat{b}_z, \hat{b}_w)$, in clip space. Next, we will give an example of the effectiveness of this technique when compared to interval and affine arithmetic.

Example Assume we have the following parametric curve, $\mathbf{p}(t) = (p_x(t), p_y(t))$, where $t \in [0,1]$, $p_x(t) = 1 + 3t + 3t^2 - 2t^3$, and $p_y(t) = 1 + 9t - 18t^2 + 10t^3$. We will illustrate how interval and affine arithmetic compare to our tight polynomial bounds when computing a two-dimensional axis-aligned bounding box of this curve over the domain, $t \in [0,1]$. The resulting bounds are visualized in Fig. 4. Using standard interval arithmetic, we obtain $\hat{p}_x = [1,1] + [0,3] + [0,3] + [-2,0] = [-1,7]$ and $\hat{p}_y = [1,1] + [0,9] + [-18,0] + [0,10] = [-17,20]$, and these two intervals represent a box with an area of 296. Similarly, applying affine arithmetic [3] on the same example gives us $p_x = 3 + 9/4\varepsilon_1 + 1/2\varepsilon_2 - 3/4\varepsilon_3$ and $p_y = 9/4 - 3/4\varepsilon_1 - 13/4\varepsilon_2 + 15/4\varepsilon_3$, where $\varepsilon_i \in [-1,1]$ are noise symbols. The bounding box becomes $\hat{p}_x = [-0.5, 6.5]$, $\hat{p}_y = [-5.5, 10]$, which represent a box with an area of 108.5. To apply our tight polynomial bounds, we first observe that the polynomials for p_x and p_y are essentially in Taylor form already. Our strategy is therefore to rewrite these on Bernstein form: $p_x(t) = \mathbf{1}\cdot(1-t)^3 + \mathbf{2}\cdot 3(1-t)^2 t + \mathbf{4}\cdot 3(1-t)t^2 + \mathbf{5}\cdot t^3$, and $p_y(t) = \mathbf{1}\cdot(1-t)^3 + \mathbf{4}\cdot 3(1-t)^2 t + \mathbf{1}\cdot 3(1-t)t^2 + \mathbf{2}\cdot t^3$, where the control points have been typeset in boldface. The bounding box is then found as the minimum and maximum of the control points in x and y. This gives us $\hat{p}_x = [1,5]$ and $\hat{p}_y = [1,4]$, which has an area of 12. The tightest fit axis-aligned box has $\hat{p}_x = [1,5]$ and $\hat{p}_y = [1, 2.37]$, with an area of 5.48.

2.4 Program Analysis and Generation

In a graphics pipeline with a tessellation unit, the vertex shader receives barycentric coordinates and the associated base triangle information, and then outputs a vertex position in clip space. In the simplest vertex shader, the vertex position is computed by interpolating the base triangle vertices, using the barycentric coordinates, and transforming this position into clip space by a matrix multiplication. In the general case, the vertex position is displaced using an arbitrary function (of the barycentric coordinates) before the clip space transform.

We want to bound this position over the entire barycentric domain, and must therefore evaluate the vertex shader output for every possible barycentric coordinate, since this is the only input that varies over the base triangle. To accomplish this, we reformulate the vertex shader using Taylor models.

We represent each Taylor model as a coefficient list. Each coefficient has a scalar value and an id i, indicating that it is the coefficient of the x^i term. For example, the polynomial $4+3x+0.5x^2$ over the domain $x \in [0,1]$ would be represented, as a Taylor model of order 2, by the list $[\{4,0\},\{3,1\},\{0.5,2\},\hat{r}=0]$. It could also be represented as a Taylor model of order one as $[\{4,0\},\{3,1\},\hat{r}=[0,0.5]]$.

Our only varying input, the barycentric coordinates, are here expressed as two-dimensional Taylor models. Generalizing the list representation from above to two dimensions so that a polynomial term $\alpha x^i y^j$ is represented by a coefficient $\{\alpha,i,j\}$, we can write the barycentric coordinates as two-dimensional Taylor models:

$$\begin{aligned} u &= 0+1\cdot u+0\cdot v = [\{1,1,0\}] \\ v &= 0+0\cdot u+1\cdot v = [\{1,0,1\}] \\ w &= 1-1\cdot u-1\cdot v = [\{1,0,0\},\{-1,1,0\},\{-1,0,1\}] \end{aligned}$$

These are Taylor models of order 1 ($i,j \leq 1$) over the domain $u \in [0,1], v \in [0,1]$. Note that no remainder is needed.

We then proceed by evaluating all instructions using Taylor models. We will briefly exemplify the implementation of addition and multiplication of Taylor models, as more complex operations will be expressed in these in the end.

Addition: Addition is done by adding the polynomial part of each Taylor model. Our internal representation of the polynomial part is a list of non-zero coefficients. Thus, the polynomial addition essentially becomes a sparse vector addition at runtime. Here is an example:

$$u+w = [\{1,0,0\},\{\mathbf{1}-\mathbf{1},1,0\},\{-1,0,1\}] = [\{1,0,0\},\{-1,0,1\}] = 1-v. \quad (9)$$

Note that we only need to perform additions for non-zero terms existing in both u and w, as the other terms can be handled using variable renaming. The remainder term, if non-zero, is handled using normal interval arithmetic. A more realistic shader would include linear interpolation between two, at compile time unknown, positions. This requires us to work with variables rather than constants. Thus the

example becomes:

$$p_1 u + p_0 w = [\{p_0, 0, 0\}, \{\mathbf{p_1} - \mathbf{p_0}, 1, 0\}, \{-p_0, 0, 1\}] = p_0 + (p_1 - p_0)u - p_0 v. \tag{10}$$

Multiplication: Here, we loop over the non-zero components in one Taylor model and multiply it by all non-zero components in the other. Thus, the runtime complexity is roughly $O(a \cdot b)$ multiplications, where a and b represent the number of non-zero coefficients in each of the two polynomials being multiplied. We bound the remainder terms using interval arithmetics. This can be optimized by exploiting that our domain is $(u, v) \in [0, 1]$, as all multiplications by zero can be omitted. For multiplication, the order of the Taylor model will increase, so we have the choice to bound the higher-order terms and add to the remainder, or increase the order of the model. A higher order Taylor model has more precision (polynomials up to the order of the model can be represented exactly), but is also more costly computationally. With the sparse list representation above, we can use a fixed order and models of lower orders will not have any computational overhead, as only non-zero terms are stored and used in the arithmetic operations.

Polynomial displacement shaders (Bézier surfaces) are simply a sequence of Taylor multiplications and additions, and elementary functions can also be bounded by Taylor models. Like standard Taylor expansions, a higher-order representation leads to tighter bounds. Once all arithmetic operations have been converted to Taylor form, we express them using regular vertex shader code. Therefore, we do not need to introduce any new specialized instruction set for our bounding shader. However, the bounding shaders will be significantly longer than the corresponding vertex shader.

Finally, our program analysis gives us a polynomial approximation of the vertex position attribute. We then compute its bounds using the algorithm in Section 2.3. Once again, we generate the necessary vertex shader code for this operation.

Discussion Program analysis is done in the exact same way as a standard implementation [2] of Taylor models, with the exception that we need to treat symbolic constants (variables) rather than values, and we need to emit code rather than executing the operations.

It should be noted that the Taylor models for the barycentric coordinates are the same for all base triangles, and thus we can treat them as constants rather than varying input. This means that the order for all Taylor instructions can be computed statically at compile time. Furthermore, we can do most standard optimizations (for example, exploiting $c \cdot 0 = 0$, and $c + 0 = c$), as well as all control flow that is internally needed in the Taylor model computations, at compile time. This greatly increases the runtime shader performance.

In conclusion, the complexity of each Taylor operation is highly dependent on the "order" of the vertex shader. For instance, for a program with only interpolation and a matrix multiplication, the Taylor models will have no non-zero coefficients over order one. In contrast, cubic Bézier triangle evaluation uses polynomials of

degree 3, and consequently the Taylor models will have more higher-order coefficients. The instruction ratio between the culling program and vertex shader grows for more complex shaders (see Section 4). Note that we can determine the number of instructions during compile time. Thus we can compile the program, see how expensive it gets, and only trigger culling if there is potential for performance gain.

Texture Mapping

Shaders using texture map lookups are problematic as the texture map may contain an arbitrarily complex function. However, texture mapping is an important feature as displacement mapping is a prime use-case of a tessellation unit.

We implement bounded texture mapping using interval-based texture lookups [12, 22], which computes a bounding interval for the texture in a given region. If, for example, we want to displace a surface in the direction of an interpolated normal, then the texture interval will be used in subsequent arithmetic computations. Therefore, we must convert the interval to Taylor form.

A naïve way of doing this is to treat the texture lookup in the interval remainder term, $\hat{r}_f$, of the Taylor model. However, we found this approach to be unsatisfactory as the remainder term in Taylor models is treated using standard interval arithmetic, which cause the bounds to grow rapidly. Instead, we treat every texture lookup as a functional parameter. That is, instead of treating the shader as a two-dimensional Taylor model:

$$\tilde{f}(u,v) = \sum_{i+j\leq n} c_{ij} u^i v^j + \hat{r}_f, \tag{11}$$

we treat it as a three-dimensional Taylor model:

$$\tilde{f}(u,v,a(u,v)) = \sum_{i+j+k\leq n} c_{ijk} u^i v^j a(u,v)^k + \hat{r}_f, \tag{12}$$

where $a(u,v)$ is an unknown (texture map) function defined over the interval domain, which we computed in the interval texture lookup. By increasing the dimensionality of the Taylor models, we can track correlations for arithmetic operations which depend on texture lookups. In effect we defer the interval evaluations to the last part of the shader, which is the bounds computations. To support an arbitrary number of texture lookups, all Taylor arithmetic, as well as the tight bounding computations of Section 2.3, can be generalized to n-dimensional domains. For details, we refer to the work by Berz and Hoffstätter [2], and Lin and Rokne [16].

Branching and Looping

We can easily support branching and looping when the conditional expression is a value (or equivalently, a zero:th order Taylor model with no remainder). In this

case, it is uniquely determined which branch we should take, or how many iterations of a loop we should perform. A typical example would be looping over an, at compile time unknown, number of fractal noise octaves.

We can also handle branches with Taylor models for conditional expressions. In such cases we compute quick bounds for the Taylor model based on interval arithmetic (see multiplication in Section 2.2). If the bounds of the condition is ambiguous, we must execute both branches. Furthermore, if a variable is assigned a value in both branches, we must assign it the union of those values. A union of two Taylor models could be derived by computing the average of their polynomial parts, and growing the remainder term accordingly to enclose both polynomials.

A construct that we cannot handle is loops with a Taylor model as the conditional expression. For example, some iterative computation on the barycentric coordinates, that loops until the result has converged. As previously explained by Hasselgren and Akenine-Möller [12], such computations are not guaranteed to converge when bounded arithmetics are used, and we may get an infinite loop. Fortunately, we can easily detect those cases and simply disable our culling.

2.5 Selective Execution

We have observed that the bounding shaders are roughly $3-15\times$ more expensive than the corresponding vertex shader in terms of instructions. Since this cost is rather significant, it makes sense to execute the bounding shader only in regions where we are likely to improve overall performance. A statistical analysis shows that it is beneficial to execute the bounding shader if the following holds:

$$\frac{c(cull)}{c(vertex)} \leq p(cull) \cdot n. \tag{13}$$

Where $\frac{c(cull)}{c(vertex)}$ is the cost ratio between the cull and the vertex program, $p(cull)$ is the probability that a base triangle is culled, and n is the number of vertices that will be generated during tessellation.

2.6 Culling

In this section, we will describe how the actual culling is performed. We want to emphasize that the culling algorithm per se is not a novel contribution. However, some details are given here for the sake of completeness. Recall that the output from the bounding shader program are geometrical bounds:

$$\tilde{\mathbf{p}}(u,v) = (\tilde{p}_x, \tilde{p}_y, \tilde{p}_z, \tilde{p}_w),$$

i.e., four Taylor models. As described above, we use the convex hull property of the Bernstein form to obtain a bounding box from these Taylor models. This box is denoted $\hat{\mathbf{b}} = (\hat{b}_x, \hat{b}_y, \hat{b}_z, \hat{b}_w)$, where each element is an interval, e.g., $\hat{b}_x = [\underline{b_x}, \overline{b_x}]$.

View Frustum Culling

For view frustum culling, we simply need to test the geometrical bounds against the planes of the frustum. Since we have the bounding box, $\hat{\mathbf{b}}$, in homogeneous clip space, we can perform the test in this space as well. We use the standard optimization for plane-box tests [9], where only a single corner of the box is used to evaluate the plane equation. Each plane test then amounts to an addition and a comparison. For example, testing if the box is outside the left plane is done with: $\overline{b_x} + \overline{b_w} < 0$. Since these tests are inexpensive, our culling always starts with the view frustum test.

Backface Culling

After the vertex shader has been executed, the vertex $\mathbf{p}$ is in homogeneous clip space (before division by w). This means that the model-view transform has been applied, so the camera position is at the origin. Now, given a point, $\mathbf{p}(u,v)$, on a surface, backface culling is in general computed as:

$$c = \mathbf{p}(u,v) \cdot \mathbf{n}(u,v), \tag{14}$$

where $\mathbf{n}(u,v)$ is the normal vector at (u,v). If $c > 0$, then $\mathbf{p}(u,v)$ is backfacing for that particular value of (u,v). For a parameterized surface, the unnormalized normal, $\mathbf{n}$, can be computed as:

$$\mathbf{n}(u,v) = \frac{\partial \mathbf{p}(u,v)}{\partial u} \times \frac{\partial \mathbf{p}(u,v)}{\partial v}. \tag{15}$$

After our bounding shader has been executed, we have Taylor models, $\tilde{\mathbf{p}}(u,v)$, for the position. As part of the bounding shader program, these are differentiated as well, resulting in $\partial\tilde{\mathbf{p}}(u,v)/\partial u$ and $\partial\tilde{\mathbf{p}}(u,v)/\partial v$. Finally, the Taylor model of the normal, $\tilde{\mathbf{n}}(u,v)$, is computed using these.

There are two issues with this technique, which we need to solve. The first problem arises if $\tilde{\mathbf{p}}(u,v)$ contains a non-zero remainder term, $\hat{r}_p$, since this must be accounted for when computing the partial derivatives. We solve this by using knowledge about the tessellation frequency of the base primitive. Assume that a worst-case sawtooth tessellation pattern is generated by the remainder term, as shown in Figure 5a. The maximum slope for such a configuration is $(f(x+\Delta x) - f(x) + w)/\Delta x$, where Δx is the shortest edge generated during tessellation and w is the width of the interval remainder term. This expression is bounded by $f'(x) + w/\Delta x$ according to the mean value theorem. Similar reasoning holds for the minimum slope. Thus $\partial\tilde{\mathbf{p}}(u,v)/\partial u$ is bounded by $\partial T_p/\partial u \pm (\overline{r_p} - \underline{r_p})/\Delta x$.

It should be noted that fractional tessellation may introduce edges that are arbitrarily short, since new vertices may be inserted at the position of old ones. This makes it very hard to bound the derivatives of Taylor models with remainder terms, as we must assume that $\Delta x = 0$. We propose to modify the fractional tessellation

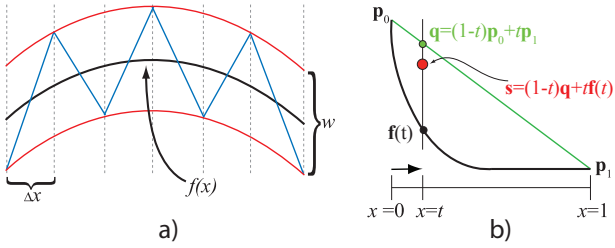


Figure 5: We must take special care of the interval remainder term when performing backface culling. Figure a) shows a worst case derivative of a Taylor model with a polynomial $f(x)$ an interval remainder term with width w. The worst case derivative that can be introduced by the remainder term is given by the blue sawtooth pattern, which has a period of $2\Delta x$ where Δx is the length of the shortest edge created during tessellation. Figure b) shows how we alter the original fractional tessellation algorithm to avoid problems that would arise in Figure a) if Δx is very small.

algorithm so that new vertices are inserted in a bi-linearly interpolated fashion. As shown in Figure 5b, we find the point $\mathbf{q}(t)$ by linearly interpolating between the two neighbors $\mathbf{p_0}$ and $\mathbf{p_1}$. Then we interpolate again between the actual position, $\mathbf{f}(t)$, and $\mathbf{q}(t)$. Given this modification, one can show that the derivative from the previous section will behave as if the minimum edge length is half of the edge length in a corresponding uniform tessellator. This means that we can now bound the slope.

The second issue concerns treatment of texture maps. As can be seen in Equation 12, a Taylor model with texture lookups will contain terms which depend on some unknown texture function $a(u,v)$. When such a term is differentiated, we will obtain partial derivatives $\partial a(u,v)/\partial u$ and $\partial a(u,v)/\partial v$. Our solution is to evaluate these terms using textures of pre-computed differentials. These differential textures are treated just like the regular textures described in Section 2.4, and increase the dimension of the Taylor models. It should be noted that this increase in dimension is not computationally costly as we rarely get more than linear dependencies of a texture.

Occlusion Culling

Our occlusion culling technique is similar to hierarchical depth buffering [8], except that we use only a single extra level (8×8 pixel tiles) in the depth buffer. The maximum depth value, z_{max}^{tile}, is stored in each tile. This is a standard technique in GPUs [20] used when rasterizing triangles. We project our clip-space bounding box, $\hat{\mathbf{b}}$, and visit all tiles overlapping this axis-aligned box. At each tile, we perform the classic occlusion culling test: $z_{min}^{box} \geq z_{max}^{tile}$, which indicates that the box is occluded at the current tile if the comparison is fulfilled. The minimum depth of the box, z_{min}^{box} is obtained from our clip-space bounding box, and the maximum depth of the tile, z_{max}^{tile}, from the hierarchical depth buffer (which already exists in a contemporary GPU). Note that we can terminate the testing as soon as a tile is

found to be non-occluded, and that it is straightforward to add more levels to the hierarchical depth buffer. Our occlusion culling test can be seen as a very inexpensive pre-rasterizer of the bounding box of the triangle to be tessellated. Since it operates on a tile basis, it is less expensive than an occlusion query.

3 Implementation

We have implemented our automatic culling unit in a C++ software framework simulating the GPU pipeline. We execute the bounding shader program before tessellating each base primitive. We noted that both view frustum and backface culling may be realized in the bounding shader, and our implementation generates code for this. The output of our bounding shader is therefore a single boolean indicating if the base triangle should be culled or not, and a positional bounding box. The bounding box is required for the occlusion culling, which cannot be implemented in vertex shader code as it includes (coarse level) rasterization operations. Occlusion culling is implemented further down the pipeline as a quick rasterization algorithm.

We use fourth order Taylor models in our program analysis. This gives us an exact representation of the position and normal for cubic polynomial surfaces, which are frequently used. Some examples are curved PN-triangles [26] and bicubic patches, such as Loop and Shaefer's Catmull-Clark approximation [17]. Higher-order terms will be handled by the remainder term in the Taylor model.

We believe that our automatic tessellation culling could be implemented in a graphics hardware system at a moderate cost. For a full implementation, we need additional hardware that enables us to do the following:

- Execute a bounding shader once per base primitive. The instruction set and program inputs are identical to the vertex shader. With unified shader architectures, this should be fairly straightforward to add.
- Perform the occlusion culling described in Section 2.6.
- Remove a base triangle before tessellation based on a boolean culling flag.

The remaining tasks can be done either in the bounding shader code or in a preprocessing step in a driver.

A partial implementation of our automatic culling algorithm could be realized on current hardware in two passes. First, we would execute the bounding shader program and use it to compute tessellation factors for the subsequent rendering pass. The tessellation factor can then be set to zero for all culled triangles.

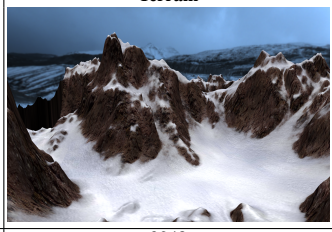
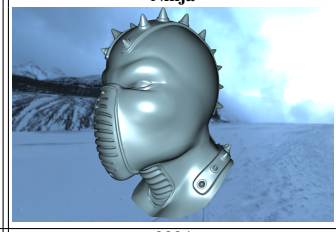
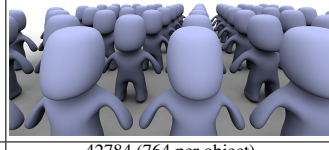
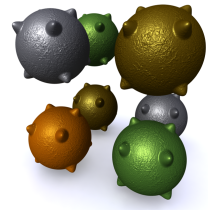
Scene	**Terrain**				**Ninja**			
								
# Base tris	2048				8884			
# Instructions (BS / VS)	140 / 50				825 / 69			
Cull rate (VF/BF/OC)	**31.8%** (26.8 / 0 / 5.0)				**39.6%** (0 / 13.5 / 26.1)			
Opt. cull rate	38.7% (27.8 / 5.3 / 5.6)				53.0% (0 / 40.7 / 12.3)			
Avg. tri area	8.0	4.0	2.0	0.5	8.0	4.0	2.0	0.5
Instruction speedup	**3.39**×	**3.53**×	**3.46**×	**3.23**×	**0.96**×	**0.99**×	**1.08**×	**1.26**×
Scene	**Figurines**				**Spike Balls**			
								
# Base tris	42784 (764 per object)				4480 (560 per object)			
# Instructions (BS / VS)	1612 / 126				2400 / 149			
Cull rate (VF/BF/OC)	**71.0%** (18.0 / 29.6 / 23.4)				**34.9%** (0 / 23.6 / 13.3)			
Opt. cull rate	74.1% (18.2 / 33.3 / 22.6)				59.5% (0 / 48.8 / 10.7)			
Avg. tri area	8.0	4.0	2.0	0.5	8.0	4.0	2.0	0.5
Instruction speedup	**2.03**×	**2.95**×	**3.81**×	**5.55**×	**0.92**×	**1.12**×	**1.24**×	**1.37**×

Table 1: Performance evaluation for our four test scenes. The instructions row shows the number of scalar instructions for the vertex shader (VS), and the bounding shader (BS). The cull rate row shows how many base primitives our algorithm can automatically cull. The bold figure is the total culling rate, and the numbers in the parenthesis are for view frustum (VF), back face (BF), and occlusion (OC) culling. The Optimal cull rate row shows the best possible culling. For each scene, we then show Instruction speedup for four different tessellation rates, so that the average tessellated triangle area is 8, 4, 2, and 0.5. These figures were computed by dividing the number of instructions to compute the vertex position of every tessellated triangle by the sum of the instructions used by our bounding shader program and the instructions used for the non-culled vertices.

4 Results

Our test setup and results will be presented in this section. We use the software GPU simulator described in the previous section, and render all images in 1920×1280 resolution. Since, to the best of our knowledge, no system exists that can *automatically* perform culling based on vertex shader analysis, cull shader generation, and on-the-fly execution, we decided to compare our system against an "optimal" culling unit. This unit can, for example, backface cull a base triangle

only if all tessellated triangles are backfacing. In practice, such optimal culling uses too much resources and so will not provide much (if any) speedup. However, from a scientific point of view, it is interesting to find out how close to an optimal culling unit our algorithm performs.

To investigate the performance of our algorithm, we use four test scenes, two of which have recently been used in GPU tessellation contexts. These are *Ninja*, *Terrain*, *Figurines*, and *Spike balls*, as can be seen in Table 1. In addition, we decided to use four tessellation rates, giving approximate triangle areas of 8, 4, 2, and 0.5 pixels. We motivate these rates by the fact that GPUs were balanced for eight-pixel triangles already two years ago [23], and the introduction of tessellation units indicates an intention to further decrease the size. Our highest tessellation rate (0.5 pixels) is inspired by production rasterization pipelines [1], which is another possible application of our culling unit.

The **Terrain** scene is a common usage area for tessellation. A coarse mesh is finely tessellated and displaced. The camera moves over the landscape, and so a fair amount of view frustum culling should be possible. This scene has the most inexpensive bounding shader, which is only $2.8\times$ as expensive as the vertex shader. The **Ninja** scene uses displacement mapping along an interpolated normal. The model is always inside the view frustum, and so only backface and occlusion culling can occur. Furthermore, the base mesh is highly tesselated, which makes it a rather hard case for our algorithm. A highly tesselated base mesh will not generate as many tesselated vertices, and hence, there is not as much to be gained by the culling. The **Figurines** scene consists of a set of models using PN-triangles [26], i.e., cubic Bézier triangles. We included this scene to demonstrate that render-time mesh smoothing can be handled efficiently by our culling algorithm. The scene shows a grid of meshes seen from the front and tests all three types of culling. It has the highest number of base primitives, but also has many more separate objects and the most complex geometry. The final scene, **Spike Balls**, shows PN-triangulated spheres with displacement mapping. This scene has the most expensive bounding shader program, approximately 2400 instructions long. Since everything is inside the view frustum, this scene only uses backface and occlusion culling.

We present our performance figures in Table 1. The culling rates show how our culling unit compares to the optimal culling unit described above. Note that our culling unit in some cases performs better than the optimal unit at occlusion culling. This only occurs because the occluded triangles were removed by the backface culling test in the optimal culling unit. For the culling rate figures, we execute our bounding shader for every base triangle in order to make a fair comparison to the optimal culling unit. For the performance figures (Instruction speedup), we instead execute the bounding shader based on Equation 13, where we chose $p(cull) = 0.5$.

It should be noted that the instruction counts for bounding and vertex shaders presented in Table 1 is the number of *scalar* instructions used, and not vector instructions. The motivation for this is that modern graphics hardware architectures use scalar instructions internally, and achieve parallelism by operating on multiple

vertices or pixels instead. Note also that we counted multiplications and additions separately for simplicity. It is, however, likely that the bounding shader programs can be significantly shortened using multiply-add.

It should also be noted that our performance numbers do not include the actual tessellation (i.e., generation of connected vertices) nor execution of instructions in the vertex shader not dealing with computing vertex position (e.g. vertex lighting, tangent space transforms etc.). In addition, our simple occlusion culling is not included either since it has to be implemented in custom hardware, but given its simple nature it should be very efficient. In summary, we believe that our performance would be even better if these factors were taken into account.

Discussion Given that our culling is automatically derived from a vertex shader program, we consider our culling rates very high, compared to the optimal culling rates. Note that we have intentionally avoided very simple test scenes where, for example, a detailed, tessellated character is behind a wall. In such cases, our occlusion culling would cull the entire character given that the wall was rendered first. One thing we noted in particular is that backface culling of displacement mapped surfaces is a very hard task (although our algorithm handles the Ninja and Spike Balls scenes fairly well).

We also compared our culling results with generalized interval arithmetic (first order Taylor models), and noted that the results directly dropped to 0% culling for the scenes with Bézier surfaces, namely, Figurines and Spike Balls. This clearly motivates our choice of higher-order Taylor models as a suitable arithmetic for bounding shaders. For the remaining test scenes, Terrain and Ninja, we get the exact same behavior for generalized interval arithmetic and higher order Taylor models. This is to be expected, since our Taylor model implementation never use higher order than necessary. Thus, the culling performance, and the instruction ratio between the bounding and vertex shader, are identical for these scenes.

Our PN-triangle scenes (Figurines and Spike Balls) use third order surfaces, similar to the popular Catmull-Clark subdivision schemes. We also performed initial experiments with Loop and Schaefer's [17] implementation of Catmull-Clark, for the Figurines scene. As the surfaces are bicubic, they contain more high-order terms than corresponding Bézier triangles, and consequently the bounding shader becomes more expensive (6536 instructions bounding shader, and 159 instructions vertex shader, as compared to 1612/126 instructions with PN-triangles). However, we only need to execute the bounding shader once for every quad, in this case. The culling rate was within 2% of that of the PN-triangle version. It should be noted that shaders as long as 6536 instructions may not fit in current instruction caches, which may harm performance. However, we believe that future hardware will be able to handle longer shaders.

As can be seen in Table 1, the performance is very high for scenes with view frustum culling (the Terrain and Figurines scenes). In all scenes, we use fractional tessellation and projected edge lengths to determine the tessellation factors for each edge of the base triangles. A fundamental problem with this approach is that we

cannot conservatively determine if the edge will be visible or not without tessellating it. Therefore, we chose tessellation factors based on projected edge lengths, without clipping the edges by the view frustum. This leads to highly tessellated base triangles close to the (infinite) near clipping plane, and consequently we get a substantial speedup if we can cull these. This is a general problem in tessellation, and not bound to our application. In fact, using our culling unit makes it much simpler to design a tessellation heuristic, since culling is handled automatically.

Our tessellation heuristic also includes a maximum tessellation factor to avoid generating base triangles being too highly tessellated. This limit is reached when the Terrain scene is rendered at high tessellation rates. Consequently, the vertex rate of the base triangles close to the camera (many of which we can cull) goes down, and this explains why the performance gain (instruction speedup) for this scene decreases when we increase the tessellation rate.

It should be noted that our culling technique is not limited to polynomial surfaces. Fig. 1 shows an example of a vertex shader with sines and cosines, wrapping a planar surface to a torus. Still, we can cull 56% (60% optimal) of the triangles before tessellation.

5 Conclusion and Future Work

The trend in GPU rendering is steadily continuing to close in on the quality of rasterization-based production pipelines. Using hardware to obtain highly tessellated objects is another step in this direction. We are therefore excited about the recent developments in hardware tessellation, and hopefully, our work can be used in future implementations of GPUs to accelerate rendering further. As we have shown, this would give significantly better performance, and since our technique is fully automatic, we believe the application developers would find more motivation to use hardware tessellation if the culling is done for them by the system. For future work, we would like to investigate hierarchical tessellation, so that parts of a base primitive can be culled, or even several base primitives in a single cull operation. In addition, we have realized that backface culling is the most difficult type of culling when it comes to handling arbitrary vertex shaders. Therefore, we would like to do research on novel techniques to further increase the backface cull rate at a low cost. Furthermore, our work can be used in a software rendering pipeline as well, and it would be interesting to evaluate exactly what kind of performance can be obtained in such contexts.

Acknowledgements

Tomas Akenine-Möller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation. In addition, we acknowledge support from the Swedish Foundation for strategic research. Thanks to Natalya Tatarchuck for giving us access to the ninja model.

Bibliography

[1] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000.

[2] Martin Berz and Georg Hoffstätter. Computation and Application of Taylor Polynomials with Interval Remainder Bounds. *Reliable Computing,*, 4(1):83–97, 1998.

[3] J. L. D. Comba and J. Stolfi. Affine Arithmetic and its Applications to Computer Graphics. In *Proc. VII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'93)*, pages 9–18, 1993.

[4] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, pages 96–102, 1987.

[5] Michael Doggett. Xenos: XBOX 360 GPU. Eurographics presentation, September 2005.

[6] Michael Doggett and Johannes Hirche. Adaptive View Dependent Tessellation of Displacement Maps. In *Graphics Hardware*, pages 59–66, 2000.

[7] Ned Greene and Michael Kass. Error-bounded antialiased rendering of complex environments. In *Proceedings of ACM SIGGRAPH 1994*, pages 59–66, 1994.

[8] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-Buffer Visibility. In *Proceedings of ACM SIGGRAPH 93*, pages 231–238, August 1993.

[9] Eric Haines and John Wallace. Shaft Culling for Efficient Ray-Traced Radiosity. In *Proceedings of the Second Eurographics Workshop on Rendering*, pages 122–138, 1994.

[10] Chang-Young Han, Yeon-Ho Im, and Lee-Sup Kim. Geometry Engine Architecture with Early Backface Culling Hardware. *Computers & Graphics,*, 29(5):415–425, June 2005.

[11] E. R. Hansen. A Generalized Interval Arithmetic. In *Proceedings of the International Symposium on Interval Mathemantics*, pages 7–18, 1975.

[12] Jon Hasselgren and Thomas Akenine-Möller. PCU: The Programmable Culling Unit. *ACM Transactions on Graphics,*, 26(3):92.1–92.10, 2007.

[13] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Sampling procedural shaders using affine arithmetic. *ACE Transactions on Graphics,*, 17(3):158–176, 1998.

[14] Ralf Hungerbuhler and Jurgen Garloff. Bounds for the Range of a Bivariate Polynomial over a Triangle. *Reliable Computing,*, 4(1):3–13, 1998.

[15] Subodh Kumar and Dinesh Manocha. Hierarchical Visibility Culling for Spline Models. In *Graphics Interface*, pages 142–150, 1996.

[16] Qun Lin and J.G. Rokne. Interval Approximation of Higher Order to the Ranges of Functions. *Computers & Mathematics with Applications*, 31(7):101–109, 1996.

[17] Charles Loop and Scott Schaefer. Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. Technical report, MSR-TR-2007-44, Microsoft Research, 2007.

[18] Kyoko Makino and Martin Berz. Taylor Models and Other Validated Functional Inclusion Methods. *International Journal of Pure and Applied Mathematics,*, 4(4):379–456, 2003.

[19] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[20] Steve Morein. ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings*. ACM Press, August 2000.

[21] Henry Moreton. Watertight Tessellation using Forward Differencing. In *Graphics Hardware*, pages 25–32, 2001.

[22] Kevin Moule and Michael D. McCool. Efficient Bounded Adaptive Tessellation of Displacement Maps. In *Graphics Interface*, pages 171–180, 2002.

[23] Matt Pharr. Interactive Rendering In The Post-GPU Era. Keynote in Graphics Hardware, 2006.

[24] Leon A. Shirman and Salim S. Abi-Ezzi. The Cone of Normals Technique for Fast Processing of Curved Patches. *Computer Graphics Forum,*, 12(3):261–272, 1993.

[25] Natalya Tatarchuk, Christopher Oat, Jason L. Mitchell, Chris Green, Johan Andersson, Martin Mittring, Shanon Drone, and Nico Galoppo. Advanced Real-Time Rendering in 3D Graphics and Games. SIGGRAPH course, 2007.

[26] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. Curved PN triangles. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 159–166, 2001.

[27] Daniel Wexler, Larry Gritz, Eric Enderton, and Jonathan Rice. GPU-Accelerated High-Quality Hidden Surface Removal. In *Graphics Hardware*, pages 7–14, 2005.

[28] X. Zhang, S. Redon, M. Lee, and Y.J. Kim. Continuous Collision Detection for Articulated Models using Taylor Models and Temporal Culling. *ACM Transactions on Graphics,*, 26(3):15.1–15.10, 2007.

Paper VIII

Efficient Bounding of Displaced Bézier Patches

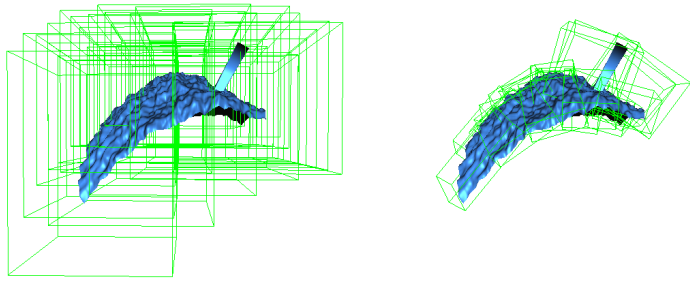
Jacob Munkberg[†‡] Jon Hasselgren[‡] Robert Toth[‡] Tomas Akenine-Möller[†‡]

[†]Lund University [‡]Intel Corporation

ABSTRACT

In this paper, we present a new approach to conservative bounding of displaced Bézier patches. These surfaces are expected to be a common use case for tessellation in interactive and real-time rendering. Our algorithm combines efficient normal bounding techniques, min-max mipmap hierarchies and oriented bounding boxes. This results in substantially faster convergence for the bounding volumes of displaced surfaces, prior to tessellation and displacement shading. Our work can be used for different types of culling, ray tracing, and to sort higher order primitives in tiling architectures. For our hull shader implementation, we report performance benefits even for moderate tessellation rates.

High Performance Graphics 2010, pages 152–162



CBOX TPATCH

Figure 1: CBOX, which represent previous work, bounds displaced Bézier surfaces by its control points and a user-provided displacement bound. Our approach, TPATCH, uses oriented bounding boxes, a min/max hierarchy of the displacement map and an efficient normal bounding algorithm, that combined bound the patches significantly tighter.

1 Introduction

Modern graphics processors contain dedicated hardware for tessellating parametric patches into many small triangles. The Direct3D 11 API adds three new stages to the graphics pipeline to support tessellation: the *hull shader*, which is executed once per patch and once per control point, typically to compute tessellation factors and change control point bases. The fixed-function *tessellator*, which generates a large set of vertex positions in the domain of the input primitive. The *domain shader*, which is executed once per generated vertex position and outputs a displaced point in clip space. We expect high pressure on these shader stages, due to significant geometry amplification. It is therefore of utmost importance to reduce the number of domain shader evaluations. This can be done by culling patches that do not contribute to the final image. To make this efficient, an algorithm for computing tight bounds of displaced surfaces is needed.

In tile-based rendering architectures [4, 9], bounds for input primitives are needed for efficient sorting into tiles. Since the domain shader is programmable, it is hard to give conservative and tight bounds of the output positions. Thus, the generated small triangles have to be sorted into tiles individually. This increases the memory requirements for the tile queues and prevents efficient occlusion culling on a patch level.

Related Work In some REYES/RenderMan [1, 2] implementations, the user can provide an explicit `displacementbound` parameter, so that the primitive can be bounded and possibly culled during the split-dice step of the pipeline. However, this places the burden on the user, who has to estimate the maximum displacement

radius. In addition, this value does not decrease during the split-dice loop, so the convergence is rather poor, as can be seen in the left side of Figure 1. Our approach is to compute these bounds based on the domain shader only (i.e., no need for any user specified parameter), and to adaptively refine the bounds as the primitive is split into smaller sub-patches.

Previous work on pre-tessellation culling [7] has shown that bounding displaced surfaces can give performance benefits for sub-pixel sized polygons. In contrast to that work, we focus on a particular use case (displaced Bézier patches). In addition, we approach the problem *hierarchically* in order to improve the total performance.

Several algorithms for normal vector bounding of Bézier surfaces exist [10, 19, 20, 21]. We extend these approaches so that they fit in our framework of bounding *displaced* patches. This is a harder problem than bounding the Bézier normal vector in isolation.

Displacement map lookups can be bounded by min-max mipmap hierarchies [6, 14], storing the minimum and maximum displacement values for each texture footprint and miplevel. We use this technique for conservative texture bounds.

The main contribution of this paper is a complete algorithm for conservative and tight bounding of *displaced* Bézier patches, using efficient normal bounding, oriented bounding boxes and min-max mipmap hierarchies of the displacement texture. The algorithm is applicable in DX11 GPUs and for hierarchical bounding in offline rendering.

2 Bounding Displaced Bézier Patches

Collections of bi-cubic Bézier patches are popular rendering primitives in production pipelines and CAGD [16]. Commonly, displacements from high resolution textures are added in the patch's normal direction to increase the surface detail. Furthermore, recent work [11, 12, 15, 17] has shown that Catmull-Clark subdivision surfaces can be approximated by collections of Bézier patches. This implies that the Bézier patch with displacement could be a prime use case for domain shaders in DX11. The Bézier patch is compactly represented by its control points, and this parametric surface representation can be efficiently evaluated in parallel (unlike recursive subdivision surfaces).

A Bézier patch, $\mathbf{p}(u,v)$, is a surface defined over two parametric coordinates, u and v. A *displaced* Bézier patch,

$$\mathbf{d}(u,v) = \mathbf{p}(u,v) + \hat{\mathbf{n}}(u,v)t(u,v), \tag{1}$$

contains the base patch position, $\mathbf{p}(u,v)$, and a displacement value, $t(u,v)$, acting along the normalized surface normal $\hat{\mathbf{n}}(u,v)$. Typically $t(u,v)$ is taken from a texture. The clip space position, $\mathbf{q}$, in homogeneous coordinates, is obtained by multiplying the displaced point with the model view projection matrix, $\mathbf{M}$:

$$\mathbf{q}(u,v) = \mathbf{M}\,\mathbf{d}(u,v) = \mathbf{M}(\mathbf{p}(u,v) + \hat{\mathbf{n}}(u,v)t(u,v)). \tag{2}$$

This equation constitutes the domain shader we want to bound. The task at hand is finding conservative bounds of $\mathbf{q}(u,v)$ over a parametric domain, $(u,v) \in [a,b] \times [c,d]$.

3 Algorithm

This section describes how we bound each term in Equation 2.

3.1 Bounding Bézier Patches

Following standard notation for tensor product Bézier surfaces [3], a Bézier patch $\mathbf{p}(u,v) : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ is defined by:

$$\mathbf{p}^{m,n}(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} \mathbf{c}_{i,j} B_i^m(u) B_j^n(v), \tag{3}$$

where $\mathbf{c}_{i,j}$ are the control points, m and n are the degrees of the patch in the parametric coordinates, u and v, respectively, and the $B(\cdot)$'s are Bernstein polynomials. In the following, we will use the term *base patch* to denote the Bézier patch which has not (yet) been displaced. This is to distinguish it from the final displaced surface. Bézier patches have the convex hull property [3], and they can easily be bounded by their control points. Finding an axis-aligned bounding box (AABB) for a Bézier patch accounts for 3 `min` and 3 `max` operations per control point.

Coordinate Frame from Control Points

We have devised a simple method for finding a coordinate frame which more tightly encloses the base patch. For a Bézier curve, the vector between the first and last control point often forms a good, first axis for a two-dimensional OBB. For a Bézier patch, we simply average the vectors from the corner control points (Figure 2), to get two axes. Given a patch with $m \times n$ control points, we denote the four corner control points $\mathbf{c}_{0,0}$, $\mathbf{c}_{m,0}$, $\mathbf{c}_{0,n}$ and $\mathbf{c}_{m,n}$, and form the two vectors:

$$\mathbf{t} = \mathbf{c}_{m,0} - \mathbf{c}_{0,0} + \mathbf{c}_{m,n} - \mathbf{c}_{0,n}, \tag{4}$$

$$\mathbf{b} = \mathbf{c}_{0,n} - \mathbf{c}_{0,0} + \mathbf{c}_{m,n} - \mathbf{c}_{m,0}. \tag{5}$$

$\mathbf{t}$ and $\mathbf{b}$ can be seen as approximate average gradients in the u and v parametric directions respectively. Their cross product gives a third axis $\mathbf{n} = \mathbf{t} \times \mathbf{b}$, and to form an orthonormal coordinate system, we set $\mathbf{x} = \mathbf{t}$, $\mathbf{y} = \mathbf{n} \times \mathbf{t}$, and $\mathbf{z} = \mathbf{n}$ and normalize each vector. The final coordinate system is: $(\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}})$. More elaborate OBB fitting schemes based on the control point cage could be derived, but in practice, the simple approach above produces axes for OBBs that bound the surface tightly. The difference in quality between bounding with AABBs and OBBs is highlighted in Figure 3 for curves and in Figure 1 for a displaced Bézier patch. As we will show below, the derived OBB axes are reused in the normal bounding algorithms.

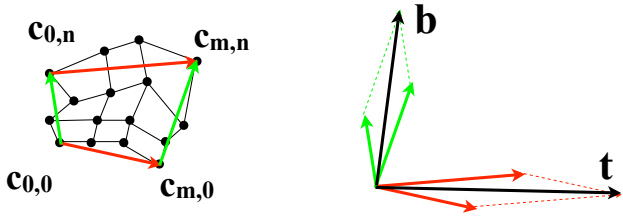


Figure 2: By forming vectors between the corners of the patch, the OBB axes can be derived.

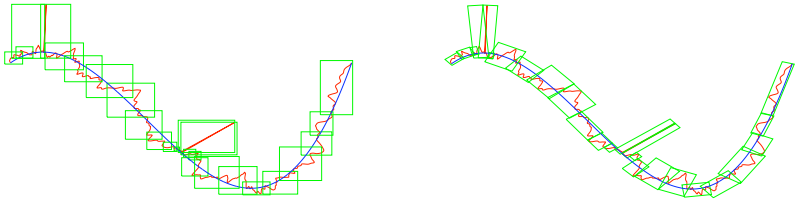


Figure 3: A cubic Bézier curve with high frequency displacement is bounded. The left image use AABBs, and the right image use OBBs, whose axes are determined by the control points of the Bézier curve.

3.2 Bounding the Normal

Bounding the patch normal, $\hat{\mathbf{n}}(u,v)$, over a domain is considerably more difficult than bounding the base position. The normal direction is computed as the cross product of two parametric derivatives of the base patch, $\mathbf{p}(u,v)$. The partial derivatives of a Bézier patch (Equation 3) can be written as:

$$\frac{\partial \mathbf{p}}{\partial u}(u,v) = \sum_{i=0}^{m-1}\sum_{j=0}^{n} \mathbf{a}_{i,j} B_i^{m-1}(u) B_j^n(v), \tag{6}$$

$$\frac{\partial \mathbf{p}}{\partial v}(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n-1} \mathbf{b}_{i,j} B_i^{m}(u) B_j^{n-1}(v), \tag{7}$$

where:

$$\mathbf{a}_{i,j} = m(\mathbf{c}_{i+1,j} - \mathbf{c}_{i,j}),\ \mathbf{b}_{i,j} = n(\mathbf{c}_{i,j+1} - \mathbf{c}_{i,j}). \tag{8}$$

Note that $\mathbf{a}_{i,j}$ and $\mathbf{b}_{i,j}$ are (scaled) differences of the control points of the base patch, and therefore vectors. If the *bidegree* of $\mathbf{p}(u,v)$ is (m,n) in the parametric coordinates (u,v), the first order parametric derivatives have degrees $(m-1,n)$ and $(m,n-1)$, which can be seen in Equations 6 and 7. As shown below, the bidegree of the patch after taking the cross product of the patches is $(m+n-1,m+n-1)$. A

patch representing the normal vector of a bi-cubic Bézier patch thus needs bidegree (5,5) to be represented exactly.

Normal Bounds from the Normal Vector Patch

Here, we describe a normal bounding algorithm, inspired by *Bézier cone* techniques [19, 20]. In summary, Bézier patches for the parametric derivatives are computed, and used to calculate a normal vector Bézier patch [21]. Its control vectors are normalized, and the solid angle of this patch on the unit sphere is bounded in an OBB coordinate frame, resulting in conservative bounds of the normalized normal.

The Bézier patch's normal direction is defined by:

$$\mathbf{n}(u,v) = \frac{\partial \mathbf{p}}{\partial u}(u,v) \times \frac{\partial \mathbf{p}}{\partial v}(u,v) = \sum_{i=0}^{m-1}\sum_{j=0}^{n} \mathbf{a}_{i,j} B_i^{m-1}(u) B_j^n(v) \times \sum_{k=0}^{m}\sum_{l=0}^{n-1} \mathbf{b}_{k,l} B_k^m(u) B_l^{n-1}(v). \tag{9}$$

Using the formula for products of Bernstein polynomials [3],

$$B_i^m(u) B_j^n(u) = \frac{\binom{m}{i}\binom{n}{j}}{\binom{m+n}{i+j}} B_{i+j}^{m+n}(u), \tag{10}$$

Equation 9 is written as:

$$\sum_{i,j,k,l} \mathbf{a}_{i,j} \times \mathbf{b}_{k,l} \frac{\binom{m-1}{i}\binom{m}{k}\binom{n}{j}\binom{n-1}{l}}{\binom{m+n-1}{i+k}\binom{m+n-1}{j+l}} B_{i+k}^{m+n-1}(u) B_{j+l}^{m+n-1}(v). \tag{11}$$

This is a Bézier patch of bi-degree $(m+n-1, m+n-1)$ with control vectors, $\mathbf{v}_{p,q}$, given by:

$$\mathbf{v}_{p,q} = \sum_{\substack{i+k=p \\ j+l=q}} \mathbf{a}_{i,j} \times \mathbf{b}_{k,l} \frac{\binom{m-1}{i}\binom{m}{k}\binom{n}{j}\binom{n-1}{l}}{\binom{m+n-1}{i+k}\binom{m+n-1}{j+l}}. \tag{12}$$

To conservatively bound the normal over the patch, we follow the approach by Sederberg and Myers [19]. The control vectors, $\mathbf{v}_{p,q}$, are normalized and bounded by a cone on the unit sphere, as shown in Figure 4. For efficiency, we reuse the $\hat{\mathbf{z}}$-axis from the OBB coordinate frame derived for the base patch (Section 3.1) as cone axis, which is an approximation of the patch's average normal. The minimal scalar product between this axis and any normalized control vector gives the cosine of the half-angle, θ, of a cone $N : \{\hat{\mathbf{n}}, \theta\}$, where $\hat{\mathbf{n}} = \hat{\mathbf{z}}$. The cone N will enclose all the normals. As shown in Figure 5A, the bounds for the normal expressed in the OBB coordinate frame are:

$$([-\sin\theta, \sin\theta], [-\sin\theta, \sin\theta], [\cos\theta, 1]). \tag{13}$$

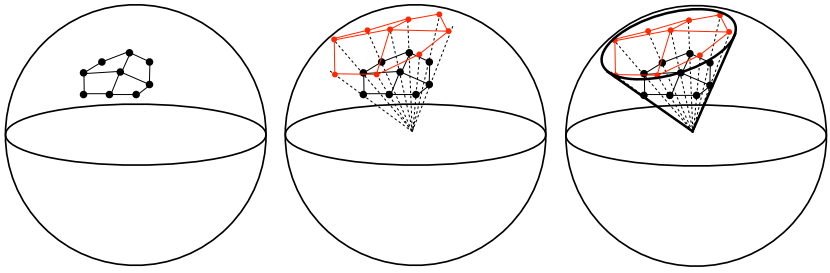


Figure 4: Bounding control vector patches (e.g. normal or tangents). The leftmost image shows a control vector patch. In the middle image, each control vector is normalized, so that they map to points on the unit sphere (marked in red). Finally, in the rightmost image, points on the unit sphere are bounded by a cone.

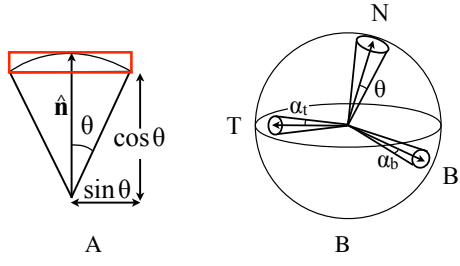


Figure 5: A. In an OBB coordinate frame with one axis aligned with the cone's axis, the bounds of the cone on the unit sphere are easily derived using the cone half angle θ. B. Given bounding cones for the two parametric derivatives (denoted T and B), a cone that bounds the cross product of any vector inside T and any vector inside B can be derived, here denoted N.

In our experience, this approach gives very tight bounds, and as the patch is subdivided, the normal bounds converge quickly. The normal vectors could be bounded using a more elaborate algorithm for finding a bounding volume on the spherical surface. However, the cone approach combined with our OBB coordinate frame is efficient and facilitates the enclosure of the bounds from the base patch and the displacement along the normal vector. The main disadvantage is the cost of deriving the normal vector patch. For a bi-cubic Bézier patch, the computation of $\mathbf{v}_{p,q}$ includes 144 cross products and 36 normalization operations. The binomial coefficients, though, can be pre-computed in a small lookup table of 36 entries.

Normal Bounds From Tangent Cones

As shown by Sederberg and Myers [19], coarser bounds can be obtained more quickly by forming two *tangent cones* from the control vectors of the first order

parametric derivative patches, $\partial\mathbf{p}/\partial u$ and $\partial\mathbf{p}/\partial v$ (see Equations 6 and 7). The control vectors of the two derivative patches are normalized and bounded on the unit sphere (as shown in Figure 4), forming two cones $T:\{\hat{\mathbf{t}},\alpha_t\}$ and $B:\{\hat{\mathbf{b}},\alpha_b\}$. We use the $\hat{\mathbf{t}}$ and $\hat{\mathbf{b}}$ axes derived in Section 3.1 as axes for the cones T and B. Note that these are not necessarily orthogonal. As discussed in Section 3.2, the cosine of the cone angle α_t is the minimum scalar product of any normalized control vector from the tangent patch $\partial\mathbf{p}/\partial u$ with the $\hat{\mathbf{t}}$ axis. The half angle α_b is derived analogously. If the cones T and B do not overlap, a cone N that bounds all possible cross products of two vectors, one from each of T and B, can be constructed (Figure 5B). Its axis is in the direction $\mathbf{t}\times\mathbf{b}$ and its half-angle is given by [19]:

$$\sin\theta = \frac{\sqrt{\sin^2\alpha_t + 2\sin\alpha_t\sin\alpha_b\cos\beta + \sin^2\alpha_b}}{\sin\beta}, \tag{14}$$

where β is the smallest of the two angles between the axes in the $\hat{\mathbf{t}}$ and $\hat{\mathbf{b}}$ directions. The cone, $N:\{\widehat{\mathbf{t}\times\mathbf{b}},\theta\}$, conservatively bounds the patch's normalized normal. Given θ and our choice of tangent cone axes, the normal cone axis is aligned with the OBB $\hat{\mathbf{z}}$-axis, and we can again use Equation 13 to obtain normal vector bounds in the base patch's OBB coordinate frame.

If the tangent cones overlap ($\alpha_t+\alpha_b>\beta$), we bound the normal using the unit box in the OBB coordinate frame. The tangent cone approach results in coarser bounds than the full normal vector patch approach, but is considerably less expensive. Furthermore, if the input patch is subdivided, the bounds converge quickly.

3.3 Bounded Texture Lookups

Techniques for bounding texture lookups are covered in previous work [6, 14]. The idea is to keep two extra mipmap hierarchies. The first stores maximum displacement values for each texture footprint and level and the second stores the corresponding minimum displacement values. In general, when the parametric domain decreases (e.g. the patch is subdivided), so do the texture bounds, which is a desirable characteristic.

The final bounds of the displacement vector, $\mathbf{o}=\hat{\mathbf{n}}t$, is the product (on interval arithmetic form) of the interval from the texture lookup $[t_{min},t_{max}]$ times the intervals of the normalized normal vector along each axis. Using the notation $[\underline{a},\bar{a}]$ to define an interval, where $\underline{a}$ is the lower limit and $\bar{a}$ is the upper limit, multiplication of two intervals is defined by [13]:

$$[\underline{a},\bar{a}]\otimes[\underline{b},\bar{b}] = [\min(\underline{ab},\underline{a}\bar{b},\bar{a}\underline{b},\bar{a}\bar{b}),\max(\underline{ab},\underline{a}\bar{b},\bar{a}\underline{b},\bar{a}\bar{b})]. \tag{15}$$

Hence, the interval version of the x-component of $\mathbf{o}$, is simply: $[\underline{o_x},\overline{o_x}] = [\underline{t},\bar{t}]\otimes[\underline{\hat{n}_x},\overline{\hat{n}_x}]$, and the other components are derived similarly.

3.4 Matrix Transformation

The last step in Equation 2 is the matrix transformation to clip space, so the remaining part in obtaining bounds for **q** is the model view projection matrix, which does not depend on the parametric domain, and can be seen as a constant. This constant matrix is multiplied with the eight corners of the OBB obtained for the displaced patch **d**, resulting in clip space bounds for **q**.

3.5 Hierarchical Refinement

To obtain tighter bounds, the patch can be subdivided in its parametric domain. In each subdivision step, a patch is split in two pieces, p_A and p_B. The normal bounds are recomputed for each subpatch and the min/max displacement maps are queried on the smaller footprints. The de Casteljau steps needed to generate the control points for p_A will generate the control points for p_B as a side product. The control point cage for the base patch converges quickly. The normal bounds and texture lookups generally become more accurate in each subdivision steps, resulting in a convergent hierarchical bounding algorithm. Re-evaluating the normal bounds for each subdivision step is costly, so in some scenarios, we can keep the normal bounds from a coarse level, and rely on inexpensive base patch subdivision and bounded texture lookups in the remaining steps. Also, for position bounding in surface regions without displacement (regions where $t(u,v)$ is zero), no normal bounding is needed and can be bypassed.

For adaptive refinement, such as in a REYES-like *bound & split* loop, we can maintain a priority queue of the bounding boxes of the subdomains and in each subdivision step, take the top element of the queue, split it, and insert the child boxes back into the queue. The exact sorting criteria is application dependent, and may include the screen-space extents of the bounding box, the depth values, or to prioritize boxes intersecting a frustum plane for view frustum culling.

4 Applications

As mentioned in Section 1, the obtained bounds can be used in a wide array of rendering techniques and optimizations. In this section, we present a few applications areas and suitable subdivision metrics for each.

Culling View frustum culling is performed by testing the OBB corners against the frustum planes. We can prioritize sub-patches straddling the camera frustum planes, so that geometry outside the frustum planes is culled. The culling results of the patch can also be used to avoid clip-testing the generated triangles when the patch is completely inside the view frustum.

Given a coarse depth buffer, a subpatch can be occlusion culled if its bounding box is entirely occluded by already drawn primitives [5]. We can adapt the subdivision

criterion so that sub-patches closer to the camera are processed and rasterized first, therefore increasing the likelihood of z-culling.

Backface culling is the hardest type of culling, due to the difficulty of efficiently bounding the geometric normal after displacement. However, given the tessellation rate, the normal bounds and a tight interval of the displacement, bounds for the displaced surface normal can be derived [7]. Furthermore, the subdivision criterion can be adapted to split patches with high normal variation [10].

Tile-Sorting from Bounds A bounded representation of the displaced Bézier patch can be used to sort patches into tiles before tessellation. Tile-overlap can be reduced by hierarchical subdivision of the largest screen-space bounding box.

Ray Tracing & Collision Detection In a ray tracing environment, we want to reduce the total surface area of each bounding box. Using the algorithms from Section 3, we can build a tight bounding hierarchy for the displaced patches offline, where each split is carefully chosen to minimize the surface area of the child boxes. This bounding hierarchy can then be used at runtime for efficient hierarchical intersection testing. Alternatively, the hierarchy can be built on the fly and be cached for coherent ray paths [8, 18]. In collision detection, the splits should be chosen to minimize the OBB volumes in world space.

5 Results

In this section, we denote the bounding algorithms as follows: CBOX refers to bounding the patch by its control points by finding the minimum and maximum value along the Cartesian axes. A constant displacement bound (the min-max value of the displacement texture) is added in all directions. In OBBTEX, the control points are projected on OBB axes, and the displacement value is bounded by min-max mipmap textures. No normal bounding is applied. NPATCH extends OBBTEX with the normal patch bounding algorithm from Section 3.2. TPATCH extends OBBTEX with the tangent cone normal bounding approach from Section 3.2. Finally, TAYLOR is Taylor model domain shader bounding [7] of bi-degree 5 (so that the normal direction of a cubic patch can be represented exactly), using an OBB for the bounds computations.

5.1 Cost Analysis

We first look at the case of a displaced bi-cubic patch and compare the execution cost of the bounding shader with the cost of the domain shader (evaluating Equation 2). We measure the relative performance running the shaders on an Intel Core i7 3.2 GHz CPU (on one thread) and an ATI Radeon HD5870 graphics card. We also count the number of scalar shader assembly instructions for reference. As

	#instructions	ATI HD5870	CPU
Domain Shader	1	1	1
CBOX	1.5	1.6	1.5
OBBTEX	2.7	2.7	2.4
TPATCH	4.5	3.8	4.5
NPATCH	11	83	11

Table 1: Cost comparison of bounding algorithms. The presented cost is relative to the cost of executing a single domain shader. The domain shader evaluates a cubic Bézier patch, including texture based displacement in the normal direction and model view projection. For reference, we report CPU scores with texture lookups removed (as texture sampling is considerably more costly on CPUs).

seen in Table 1, the algorithms scale as expected from the instruction count, with the exception of the NPATCH algorithm which exhausts the hardware resources (temporary registers) of the ATI card, making it perform very poorly. TAYLOR is considerably more expensive than the other bounding approaches, due to the normalization operation, which is very costly when implemented using Taylor models. When measured on the CPU *without* normalization, TAYLOR has approximately the same cost as NPATCH, but with the normalization operation included, the cost increases to about 25× the cost of NPATCH, which makes it non-competitive from a cost perspective.

It should be noted that although the bounding shaders are more expensive than the corresponding domain shader, we only need to execute the bounding shader *once* per patch, while the domain shader may be executed thousands of times per patch due to tessellation. Therefore, the total cost of executing the bounding shaders is typically considerably lower than the total cost of executing the domain shaders. For example, if we assume that we tessellate only down to the control point level (16 vertices / patch), the cost of the TPATCH bounding algorithm will only be approximately 25% of the total domain shader cost. However, it is reasonable that the tessellation level is higher than the number of control points, since it would otherwise be better to simply send the vertices and avoid tessellation and Bézier evaluations altogether. The tessellation factors are often known at the time the culling shader is applied, which implies that the bounding shader can be dynamically enabled only in areas of high tessellation.

5.2 Quality Analysis

Our test scenes consist of the three subdivision meshes shown in Figure 11, as well as the *Spikelog* mesh shown in Figure 9, which is a difficult stress case for the OBBTEX algorithm. The SubD11 mesh comes from a February 2010 DX11 SDK sample, and the *Killeroo* and *Monsterfrog* meshes are popular test cases for subdivision surfaces.

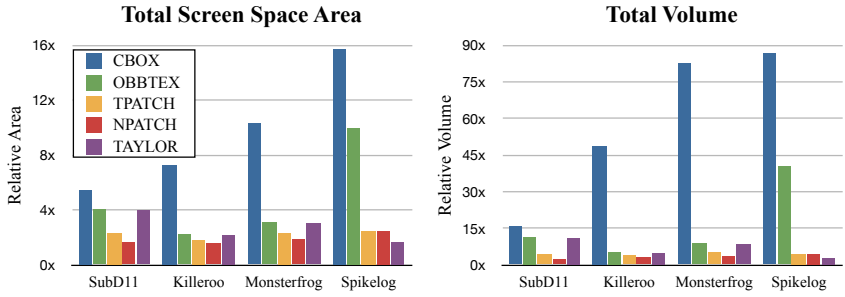


Figure 6: Quality comparison of the bounding methods. The left chart shows the total screen space bounding box area obtained by the different methods, relative to reference screen space bounding boxes. Similarly, the right chart shows the total volume of the generated bounding boxes, relative to reference bounds.

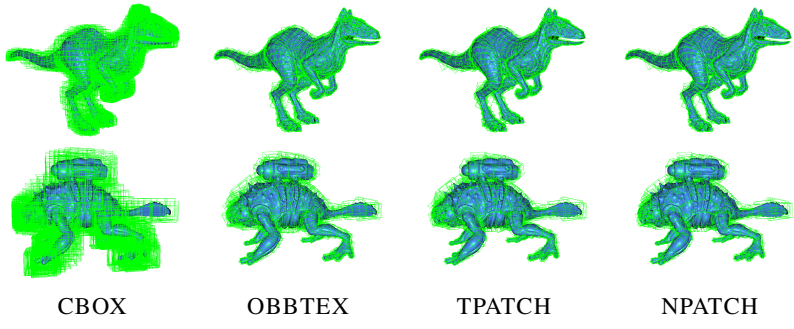


Figure 7: Object space volumes for the Killeroo and Monsterfrog models. OBBTEX *bounds are smaller than* CBOX *thanks to the use of OBBs and the min-max texture hierarchy. The low displacement amplitudes make the benefit of accurate normal bounds small for these models.*

For all our test scenes, the Catmull-Clark subdivision mesh is converted to bicubic Bézier patches with corresponding tangent patches, using Loop & Schaefer's ACC algorithm [11]. The conversion gives us 3753 Bézier patches for the SubD11 mesh, 2728 patches for Killeroo, 1292 patches for Monsterfrog, and 96 patches for Spikelog. It should be noted that all meshes except SubD11 use displacement maps to add surface detail. For the SubD11 mesh, a constant displacement is added in the normal direction, replicating the SDK sample. We use a displacement magnitude of 1.0 for the SubD11 mesh unless explicitly specified.

Figure 6 presents volume and projected screen space area relative to to a near-optimal reference. The reference is computed by evaluating the domain shader at 32×32 domain points per patch and bounding the generated vertices in the OBB coordinate frame described in Section 3.1. Thereafter we apply our bounding

algorithms and compare the resulting bounds with the reference bounds. We use the relative total volume (the total volume for an algorithm divided by the total reference volume) and relative projected screen space area as accuracy metrics. The volume metric is intended to represent quality for volume based algorithms, such as collision detection, and the projected screen space area is an efficiency metric for tile-based rendering. Both metrics are also indicators for view frustum and occlusion culling potential.

We observe that OBBTEX is significantly tighter than CBOX for all four scenes. This indicates that the OBB coordinate frame and min-map displacement lookups do make the bounds tighter. Also note that for Killeroo and Monsterfrog, OBBTEX is close in quality to TPATCH and NPATCH despite the lack of normal bounding. This is due to the low displacement magnitudes relative to the patch sizes in these scenes. Figure 7 shows the patch bounding boxes visually.

The Spikelog scene contains large displacement amplitudes. This is a difficult case for the OBBTEX algorithm, where the bounding boxes are expanded in all directions rather than just around the surface normal. As can be seen in Figure 6 and Figure 9, the TPATCH algorithm gives tighter bounds. Also note that the TPATCH bounds converge quickly as the patches are subdivided.

In a tile-based architecture, higher order primitives may be sorted into tile-specific queues based on their screen space extents before they are tessellated into small triangles. Depending on the rendering architecture, each tile may tessellate and domain shade its overlapping primitives independently, instead of caching and reusing processed geometry. This is especially true in highly parallel tiling architectures where the communication between processing units often should be kept at a minimum. It is therefore important to reduce the *tile overlap* so that primitives are not added to more tile-queues than necessary. However, this requires tight screen space bounds. With accurate bounds, the tile overlap can be significantly reduced for displaced patches. This is shown in Figure 11, where the screen-space overlap has been encoded as a heat map.

Figure 8 shows the bounding quality as function of displacement amplitude and subdivision level for the SubD11 mesh. When the displacement amplitude increases, TPATCH and NPATCH provide significantly tighter bounds, since they bound the normal more accurately. When the patch is subdivided, the convergence rate compared to CBOX and OBBTEX is even more significant. As the displacement is a constant offset in this test, the min-max textures do not help, and the only quality difference between CBOX and OBBTEX is due to the use of the OBB coordinate frame.

TAYLOR bounds the base patch very tightly, but as soon as displacement is added, the bounds are similar in quality to OBBTEX, as the Taylor model algorithm struggles to bound the normal efficiently. This is largely due to the high polynomial degrees involved in the Taylor model normalization operation. As seen in the rightmost chart in Figure 8, as the patches are subdivided and their normal vectors become more coherent, TAYLOR converges, but it is far from the quality of TPATCH

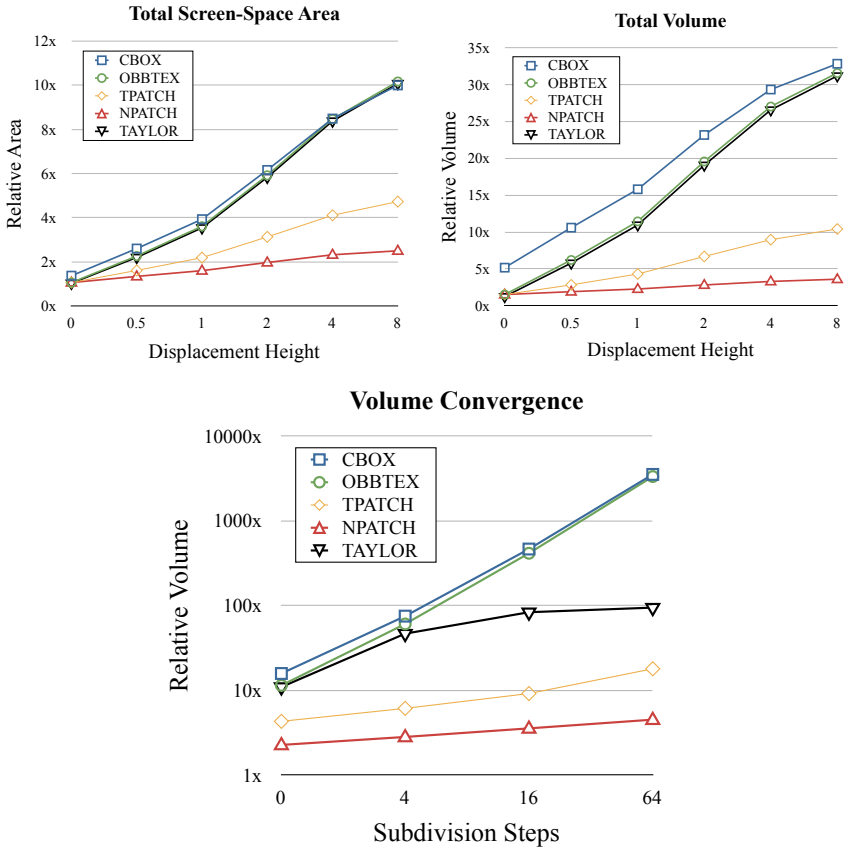


Figure 8: Measurements of bounding quality of all patches from the SubD11 sample. The total volume/area for each algorithm is divided by a reference total volume/area, and we report this ratio for each algorithm. The upper left chart shows the screen space area as a function of the displacement height. The upper right chart shows the total volume (before transformation into clip space). Finally, the bottom chart shows the total volume as a function of the number of subdivisions applied to each patch. In this chart, the displacement value is set to 1.0. As can be seen, normal bounding is critical for convergence. Note that the bottom chart uses a logarithmic scale on the y-axis.

and NPATCH. For very high subdivision levels (> 64), TAYLOR, TPATCH and NPATCH are very similar in quality, but TAYLOR is considerably more expensive. The Spikelog scene is an exception, where TAYLOR performs very well. The reason for this is that the curvatures of the base patches are relatively low, which means that the normalization operation can be accurately represented. When this happens, the higher polynomial degree of the Taylor model gives an additional improvement.

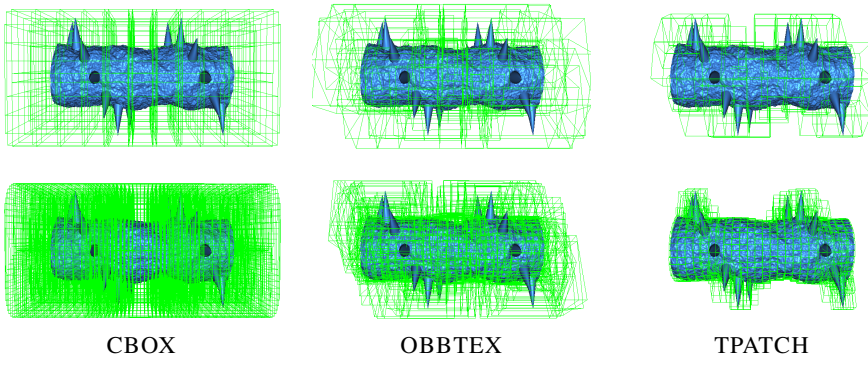


Figure 9: The Spikelog scene contains high amplitude displacement compared to the size of of the base patches. The upper row shows the bounding volumes around the base patches, and in the lower row, each patch has been divided into 16 subpatches. This is a difficult case for the CBOX *algorithm as it can never refine the texture bounds. Similarly, the* OBBTEX *algorithm gives poor bounds, as the displacement is added in all directions. In contrast, the* TPATCH *algorithm only applies the displacement around the normal direction. This gives tighter bounds, that converge towards the underlying surface when the base patches are subdivided.*

5.3 GPU Based Culling

As a stress test case for our bounding algorithms, we implemented culling in the shaders of the SubD11 sample. Due to the poor GPU scaling of the NPATCH algorithm that we observed in the cost analysis, we chose not to use that algorithm for this application.

We implement our bounding algorithms and culling tests in the *patch-constant* hull shader. This part of the hull shader may read the Bézier control cage generated in the *control point* hull shader, and we use this control cage in our bounding algorithms. We then perform simple view frustum and backface culling tests and output a zero tessellation factor if the patch can be culled. Passing zero as tessellation factor will cause the tessellation hardware to discard the patch early in the pipeline. Due to graphics API limitations, we do not subdivide the patches hierarchically. The application supports displacement, but in the current version, all displacement maps contain a constant value that the user can scale by a slider. Therefore, we can implement backface culling using the normal bounds computed in the TPATCH algorithm, by creating a cone that bounding both the geometric normal of the patch and the normal given by the ACC tangent patches. It should be noted that backface culling can be done even for general displacement maps [7], but in this case the culling rate is expected to be significantly lower.

For regular patches, there is an exact Bézier surface representation of the Catmull-Clark surface. However, for irregular patches, the Catmull-Clark surface and its

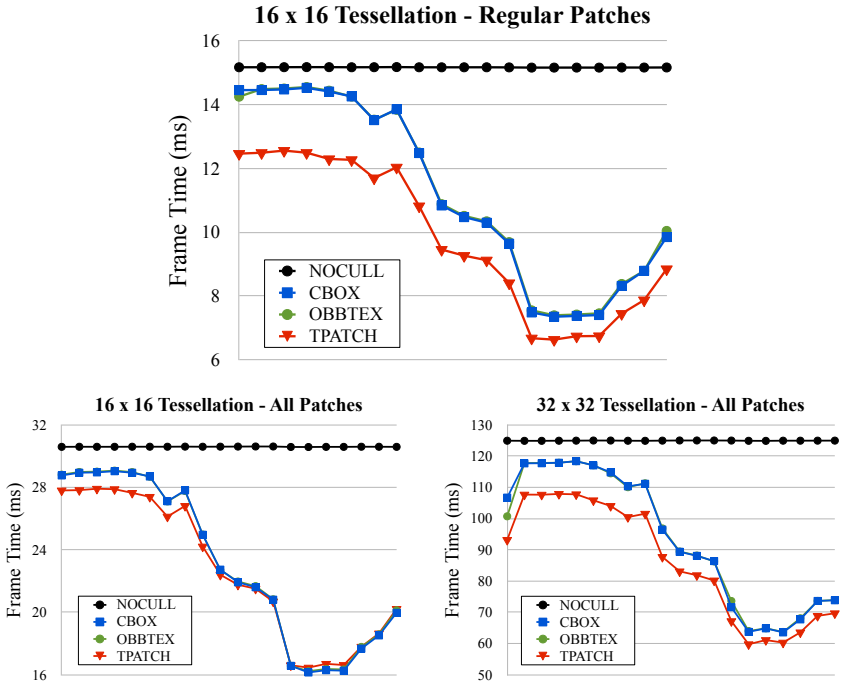


Figure 10: Each chart shows the frame time during the SubD11 animation measured on an ATI HD5870 GPU. In the second part of the animation, the camera zooms in on the character, and there is more view frustum culling potential. NOCULL *represent the original demo without culling. As can be seen, for high tessellation levels, and for the regular patches,* TPATCH *has a performance edge, but for lower tessellation levels, the naïve bounding approaches are faster. Note that* TPATCH *reduces the longest frame time in all three charts, which is the most important to accelerate for real-time rendering.*

normal needs to be approximated by separate Bézier patches for the position and tangent vectors [11]. Unfortunately, this approximation is relatively complex and needs to be done in the hull shader. When we add our bounding algorithms, it is very easy to reach the hardware resource limits mentioned in Section 5.1, which causes hull shader performance to scale very poorly.

Since this is a limitation of the particular hardware architecture, we ran two benchmarks, which gave the results shown in Table 2. In the first benchmark, we modified the SubD11 sample to render only regular patches, which we believe represents approximately how the culling will scale on future hardware with sufficient registers or efficient support for register spilling. In the second benchmark, we perform culling on all patches. As can be seen in Table 2, this approach can still be beneficial for high-quality GPU accelerated rendering applications where the tessellation factors are expected to be very high. Figure 10 shows the frame time

	Tessellation:	4×4	8×8	16×16	32×32
Regular	No Culling	2.39	3.59	15.2	61.3
	CBOX	2.42	2.93	11.2	45.0
	OBBTEX	2.50	2.93	11.2	45.0
	TPATCH	2.48	2.69	9.82	39.1
All Patches	No Culling	2.75	7.01	30.6	125
	CBOX	3.14	5.76	23.1	93.5
	OBBTEX	3.27	5.83	23.1	93.6
	TPATCH	3.89	6.92	22.7	86.0

Table 2: Average frame time (ms) for the SubD11 animation at different tessellation levels. In the upper four rows, the sample is modified to render only the regular patches. The lower four rows is the original sample, including both regular and irregular patches.

variation over the animation for 16×16 and 32×32 tessellation.

For the irregular patches, the pressure on the hull shader is significant, and high tessellation rates are needed to maintain a consistent performance benefit from the TPATCH algorithm. For the (cheaper) regular patches, there is a clear performance benefit even for lower tessellation rates.

6 Conclusions and Future Work

We have presented algorithms for efficient bounding of displaced Bézier patches, which accelerates early culling of geometry, binning of higher order primitives and construction of high quality bounding volume hierarchies. In many cases, the OBBTEX algorithm performs very well, and we expect that this algorithm will be the best short time alternative for GPU-based culling. However, for high quality tile-based renderers, larger displacements need to be handled robustly and subdivision convergence rate is important. For these cases, we believe that the TPATCH algorithm provides a better tradeoff between performance and bounding box tightness. With hardware/pipeline modifications such as support for coarse occlusion culling based on hull shader bounds, min-max texture filtering and better register management, we believe this technique can be even faster. As future work, we want to apply a variant of the TPATCH algorithm for efficient culling of displaced Gregory patches [12].

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. Tomas Akenine-Möller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation. In addition, we acknowledge support from the Swedish Foundation for strategic research. The original SubD11 code sample and mesh is a part of Microsoft's DirectX11 SDK. The

Killeroo subdivision model is courtesy of Headus (metamorphosis) Pty Ltd (available at `www.headus.com.au`). The Monsterfrog model is courtesy of Bay Raitt, Valve Software.

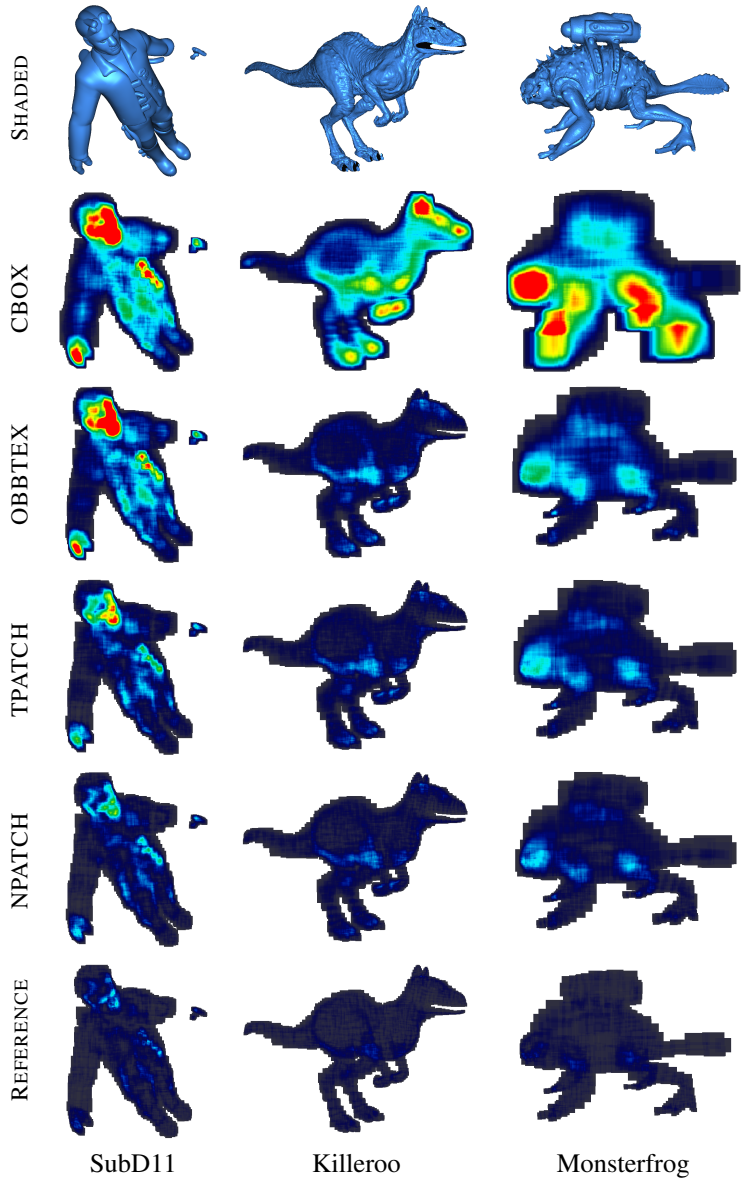


Figure 11: False color images that show the bounding box overlap in screen space. Red means 128 or more overlapping bounding boxes. For the SubD11 mesh, a constant displacement is added to the base mesh in the base patch's normal direction. For the Killeroo and Monsterfrog meshes, the original displacement maps are used.

Bibliography

[1] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000.

[2] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, pages 96–102, 1987.

[3] Gerald Farin. *Curves and Surfaces for GAGD - A Practical Guide*. Academic Press, 1996.

[4] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System using Processor-Enhanced Memories. *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, 23(3):79–88, 1989.

[5] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-Buffer Visibility. In *Proceedings of ACM SIGGRAPH 93*, pages 231–238, August 1993.

[6] Jon Hasselgren and Tomas Akenine-Möller. PCU: The Programmable Culling Unit. *ACM Transactions on Graphics,*, 26(3):92.1–92.10, 2007.

[7] Jon Hasselgren, Jacob Munkberg, and Tomas Akenine-Möller. Automatic Pre-Tessellation Culling. *ACM Transactions on Graphics,*, 28(2):1–10, 2009.

[8] Wolfgang Heidrich and Hans-Peter Seidel. Raytracing procedural displacement shaders. In *Proceedings of Graphics Interface 1998*, pages 8–16, 1998.

[9] Seiler Larry, Carmean Doug, Sprangle Eric, Forsyth Tom, Abrash Michael, Dubey Pradeep, Junkins Stephen, Lake Adam, Sugerman Jeremy, Cavin Robert, Espasa Roger, Grochowski Ed, Juan Toni, and Hanrahan Pat. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[10] Charles Loop and Christian Eisenacher. Real-Time Patch-Based Sort-Middle Rendering on Massively Parallel Hardware. Technical Report MSR-TR-2009-83, Microsoft Research, 2009.

[11] Charles Loop and Schott Schaefer. Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. *ACM Transactions on Graphics,*, 27(1):1–11, 2008.

[12] Charles Loop, Scott Schaefer, Tianyun Ni, and Ignacio Castaño. Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation. *ACM Transactions on Graphics,*, 28(5):1–9, 2009.

[13] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[14] Kevin Moule and Michael D. McCool. Efficient Bounded Adaptive Tessellation of Displacement Maps. In *Proceedings of Graphics Interface*, pages 171–180, 2002.

[15] Ashish Myles, Tianyun Ni, and Jörg Peters. Fast Parallel Construction of Smooth Surfaces from Meshes with Tri/Quad/Pent Facets. *Computer Graphics Forum,*, 27(5):1365–1372, 2008.

[16] Tianyun Ni, Ignacio Castaño, Jörg Peters, Jason Mitchell, Philip Schneider, and Vivek Verma. Efficient Substitutes for Subdivision Surfaces. In *ACM SIGGRAPH 2009 Courses*, pages 1–107, 2009.

[17] Tianyun Ni, Young In Yeo, Ashish Myles, Vineet Goel, and Jörg Peters. GPU Smoothing of Quad Meshes. *IEEE International Conference on Shape Modeling and Applications*, 27(1):1–11, 2008.

[18] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of ACM SIGGRAPH 1997*, pages 101–108, 1997.

[19] Thomas W. Sederberg and Ray J. Meyers. Loop Detection in Surface Patch Intersections. *Computer Aided Geometric Design,*, 5(2):161–171, 1988.

[20] Leon A. Shirman and Salim S. Abi-Ezzi. The Cone of Normals Technique for Fast Processing of Curved Patches. *Computer Graphics Forum,*, 12(3):261–272, 1993.

[21] Yasushi Yamaguchi. Bézier Normal Vector Surface and Its Applications. In *Proceedings of the 1997 International Conference on Shape Modeling and Applications*, page 26. IEEE Computer Society, 1997.

Paper IX

Backface Culling for Motion Blur and Depth of Field

Jacob Munkberg Tomas Akenine-Möller

Lund University / Intel Corporation

ABSTRACT

For triangles with linear vertex motion, common practice is to backface cull a triangle if it is backfacing at both the start and end of the motion. However, this is not conservative. We derive conservative tests that guarantee that a moving triangle is backfacing over an entire time interval and over the area of a lens. In addition, we present tests for the special cases of only motion blur and only depth of field. Our techniques apply to real-time and offline rendering, and to both stochastic point sampling and analytical visibility methods. The rendering errors introduced by the non-conservative test can easily be detected for large defocus blur, but in the majority of cases, the errors are hard to detect. We conclude that our tests are needed if one needs guaranteed artifact-free images. Finally, as a side result, we derive time-continuous Bézier edge equations.

journal of graphics, gpu, and game tools, to appear.

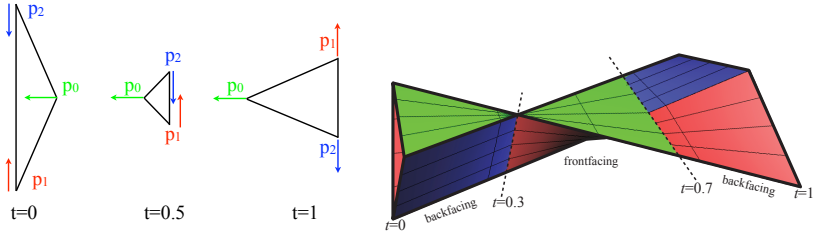


Figure 1: ***Left****: The triangle is backfacing at $t = 0$ and $t = 1$, and frontfacing at $t = 0.5$.* ***Right****: The surfaces that the moving triangle sweeps out are visualized, and one can see that the "edge surfaces" intersect when the facing changes. This happens at $t = \alpha = 0.3$ and $t = \beta = 0.7$.*

1 Introduction

Backface culling is one of the most important culling techniques in real-time graphics. When rendering motion blur and depth of field, an excessive amount of visibility tests may be executed, and therefore, accurate backface culling tests are needed also for these contexts. For moving triangles, where each vertex moves along a line in three dimensions, a commonly used technique assumes that a moving triangle is backfacing over the entire time interval if the triangle is backfacing at the start and end of the motion. However, this is not true in the general case, and certainly, this has been known in some groups, but we are not aware of any documentation of this, nor of any previous solution. We derive tests for the special cases of using either only motion blur or only depth of field. The combination of these effects is harder to handle, and therefore, we derive a coarser conservative test for motion blur with depth of field.

Let us start with a motivating example that clearly shows that a triangle can be backfacing at the start of the motion (at $t = 0$), then turns frontfacing, and then again is backfacing at the end of the motion (at $t = 1$). Consider a triangle with vertices in *xyw* space as follows:

$$\mathbf{p}_0 = (\alpha - t, 0, 1), \qquad \mathbf{p}_1 = (0, t - \beta, 1), \qquad \mathbf{p}_2 = (0, \beta - t, 1), \tag{1}$$

where $\alpha, \beta \in [0,1]$, $\alpha < \beta$ are constants, and t is the time parameter. Each triangle vertex moves linearly in time, and the triangle is shown at three different times in Figure 1. In the same figure, the swept triangle is also illustrated. The sign of the determinant of the three points determines whether the triangle can be backface culled [5]:

$$\mathbf{p}_0 \cdot (\mathbf{p}_1 \times \mathbf{p}_2) = -2(\alpha - t)(\beta - t), \tag{2}$$

where a negative sign indicates a backfacing triangle. Given $\alpha < \beta$, we note that the triangle is frontfacing when $t \in [\alpha, \beta]$, even though the triangle is backfacing at $t = 0$ and $t = 1$. At $t = \alpha$ and $t = \beta$, the triangle is degenerate, i.e., it is here that

the switch from backfacing to frontfacing or vice versa occurs. A stochastically rasterized example of this configuration is shown to the left in Figure 2.

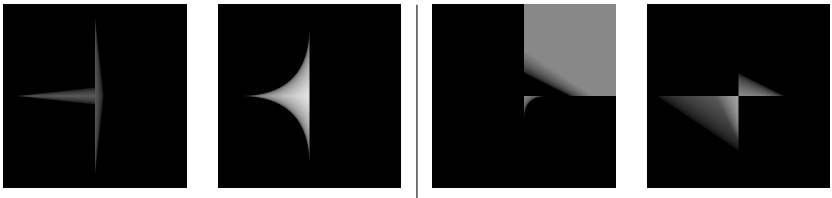


Figure 2: Motion blur rasterized triangles. The shutter is open in $t \in [0,1]$. A: The triangle moving according to Equation 1 is rasterized on screen with $\alpha = 0.1$ and $\beta = 0.9$, where the triangle is backfacing at $t = 0$ and $t = 1$. The left image shows the covered samples for the backfacing region, while the right image shows the covered samples for the frontfacing region. Note that with naïve backface culling, all the covered samples to the right will be missed, which is incorrect. B: The triangle moving according to Equation 3 flips from front to backfacing (or vice versa) three times in the time interval. The left image shows a rendering of the backfacing region, and the right image shows the frontfacing region.

In general, the determinant is a cubic polynomial, which can be seen when considering the following time-continuous triangle:

$$\mathbf{p}_0 = (\alpha - t, 0, 1), \qquad \mathbf{p}_1 = (0, \beta - t, 1), \qquad \mathbf{p}_2 = (0, 0, \gamma - t), \tag{3}$$

where $\alpha, \beta, \gamma \in [0,1]$ are constants, and t is the time parameter. The determinant then becomes:

$$\mathbf{p}_0 \cdot (\mathbf{p}_1 \times \mathbf{p}_2) = (\alpha - t)(\beta - t)(\gamma - t), \tag{4}$$

which has three real roots in the interval $t \in [0,1]$. This means that the facing can change three times. An illustration of such a triangle is shown to the right in Figure 2. At this point, we have motivated the need for a more correct test, and derivations for such tests follow in the subsequent sections.

2 Backface Culling for Motion Blur

In general, we follow the notation from previous work in the field of stochastic [1] and analytical rasterization [3]. Assume that we have a moving triangle, where the vertices move linearly within a frame, from time $t = 0$ to $t = 1$. At $t = 0$, we denote the vertices $\mathbf{q}_i$ and at $t = 1$ we call them $\mathbf{r}_i$. We work in clip space, using 2D homogeneous coordinates (2DH), so a vertex is defined as $\mathbf{p} = (p_x, p_y, p_w)$ [7]. A linearly interpolated vertex is then expressed as:

$$\mathbf{p}_i(t) = (1 - t)\mathbf{q}_i + t\mathbf{r}_i. \tag{5}$$

Given a moving triangle with vertices $(\mathbf{p}_0(t), \mathbf{p}_1(t), \mathbf{p}_2(t))$, we form the matrix:

$$\mathbf{M}(t) = \begin{bmatrix} p_{0_x} & p_{1_x} & p_{2_x} \\ p_{0_y} & p_{1_y} & p_{2_y} \\ p_{0_w} & p_{1_w} & p_{2_w} \end{bmatrix}, \tag{6}$$

where we have omitted the temporal dependence for readability. The triangle can be backface culled if $\det(\mathbf{M}) < 0$, where the determinant is expressed as [5]:

$$\det(\mathbf{M}) = \mathbf{p}_0 \cdot (\mathbf{p}_1 \times \mathbf{p}_2). \tag{7}$$

Geometrically, this can be interpreted as a (scaled) signed volume computation of the tetrahedron spanned by the origin and the triangle. Thus, we want to determine if $\mathbf{p}_0(t) \cdot (\mathbf{p}_1(t) \times \mathbf{p}_2(t)) < 0$ for $t \in [0,1]$. The cross product of two linearly moving vertices can be expanded as [1]:

$$\mathbf{p}_1 \times \mathbf{p}_2 = ((1-t)\mathbf{q}_1 + t\mathbf{r}_1) \times ((1-t)\mathbf{q}_2 + t\mathbf{r}_2) = t^2\mathbf{f} + t\mathbf{g} + \mathbf{h}, \tag{8}$$

where:

$$\begin{aligned} \mathbf{f} &= (\mathbf{r}_1 - \mathbf{q}_1) \times (\mathbf{r}_2 - \mathbf{q}_2), \\ \mathbf{g} &= (\mathbf{r}_1 - \mathbf{q}_1) \times \mathbf{q}_2 - (\mathbf{r}_2 - \mathbf{q}_2) \times \mathbf{q}_1, \\ \mathbf{h} &= \mathbf{q}_1 \times \mathbf{q}_2. \end{aligned} \tag{9}$$

Using this expression, we can derive the time-dependent determinant:

$$\begin{aligned} \det(\mathbf{M}) &= \mathbf{p}_0(t) \cdot (\mathbf{p}_1(t) \times \mathbf{p}_2(t)) \\ &= ((1-t)\mathbf{q}_0 + t\mathbf{r}_0) \cdot (t^2\mathbf{f} + t\mathbf{g} + \mathbf{h}) \\ &= at^3 + bt^2 + ct + d, \end{aligned} \tag{10}$$

where:

$$\begin{aligned} a &= (\mathbf{r}_0 - \mathbf{q}_0) \cdot \mathbf{f}, \\ b &= (\mathbf{r}_0 - \mathbf{q}_0) \cdot \mathbf{g} + \mathbf{q}_0 \cdot \mathbf{f}, \\ c &= (\mathbf{r}_0 - \mathbf{q}_0) \cdot \mathbf{h} + \mathbf{q}_0 \cdot \mathbf{g}, \\ d &= \mathbf{q}_0 \cdot \mathbf{h}. \end{aligned} \tag{11}$$

Note that the coefficient $d = \mathbf{q}_0 \cdot \mathbf{h} = \mathbf{q}_0 \cdot (\mathbf{q}_1 \times \mathbf{q}_2)$ is the backface test for the triangle at $t = 0$. Also, the value of the polynomial at $t = 1$ is $a + b + c + d = \mathbf{r}_0 \cdot (\mathbf{r}_1 \times \mathbf{r}_2)$, which, analogously, is the backface test at $t = 1$. The expression for the coefficient:

$$a = (\mathbf{r}_0 - \mathbf{q}_0) \cdot [(\mathbf{r}_1 - \mathbf{q}_1) \times (\mathbf{r}_2 - \mathbf{q}_2)], \tag{12}$$

is the determinant test for the motion vectors of the three vertices, and if they all lie in the same plane, the cubic term is zero, i.e., $a = 0$. Thus, it is only when the motion vectors span a volume in 2DH that the determinant will be a cubic function.

We note that if the polynomial does not have any roots in $t \in [0,1]$ and $d < 0$, then the triangle can be safely backface culled.

Optimization for Motion Along a Vector If the triangle's motion vectors are parallel, such that:

$$\mathbf{p}_i(t) = (1-t)\mathbf{q}_i + t\mathbf{r}_i = \mathbf{q}_i + t\gamma_i\mathbf{d}, \tag{13}$$

where $\gamma_i \in \mathbb{R}$, the computations can be simplified. It follows that $\mathbf{f} = (\mathbf{q}_1 - \mathbf{r}_1) \times (\mathbf{q}_2 - \mathbf{r}_2) = \gamma_1\gamma_2\mathbf{d} \times \mathbf{d} = 0$, and the cubic coefficient $a = 0$. Using $\mathbf{f} = 0$ and Equation 13, we can show that the coefficient b is zero when the motion vectors are parallel:

$$\begin{aligned} b &= (\mathbf{r}_0 - \mathbf{q}_0)\cdot\mathbf{g} + \mathbf{q}_0\cdot\mathbf{f} \\ &= (\mathbf{r}_0 - \mathbf{q}_0)\cdot((\mathbf{r}_1 - \mathbf{q}_1)\times\mathbf{q}_2 - (\mathbf{r}_2 - \mathbf{q}_2)\times\mathbf{q}_1) \\ &= \gamma_0\gamma_1\mathbf{d}\cdot(\mathbf{d}\times(\mathbf{q}_2 - \mathbf{q}_1)) = 0. \end{aligned}$$

Thus, for motion along a common direction, including pure translation, the determinant is a linear function in t, and given that the value of the determinant at both $t = 0$ and $t = 1$ is negative, the triangle can be backface culled.

2.1 Practical Backface Culling Test

The cubic coefficient a (Equation 12) of the backface function is the determinant of the triangle's three motion vectors, which are often small or near parallel. Therefore, directly computing the backfacing function on power form (Equation 10) can be numerically unstable. To alleviate the problem, we note that a vertex moving linearly in time as in Equation 5 is a Bézier curve of degree 1. Furthermore, the time-dependent edge equations are cross products of linearly moving vertices, and can also be expressed on Bernstein form as:

$$\mathbf{p}_i(t)\times\mathbf{p}_j(t) = (1-t)^2\mathbf{c}_0 + 2(1-t)t\mathbf{c}_1 + t^2\mathbf{c}_2, \tag{14}$$

where:

$$\mathbf{c}_0 = \mathbf{q}_i\times\mathbf{q}_j, \quad \mathbf{c}_1 = \frac{1}{2}(\mathbf{q}_i\times\mathbf{r}_j + \mathbf{r}_i\times\mathbf{q}_j) \quad \text{and} \quad \mathbf{c}_2 = \mathbf{r}_i\times\mathbf{r}_j. \tag{15}$$

Hence, the full edge equation is expressed on Bernstein form as shown below:

$$e(t,x,y) = \left((1-t)^2\mathbf{c}_0 + 2(1-t)t\mathbf{c}_1 + t^2\mathbf{c}_2\right)\cdot(x,y,1). \tag{16}$$

The backfacing function, given by the determinant (Equation 10) expressed on cubic Bernstein form then becomes:

$$\det(\mathbf{M}(t)) = \sum_{i=0}^{3} b_i \binom{3}{i}(1-t)^{3-i}t^i, \tag{17}$$

with coefficients, b_i, given by:

$$\begin{aligned} b_0 &= \mathbf{q}_0\cdot(\mathbf{q}_1\times\mathbf{q}_2), \\ b_1 &= 1/3[\mathbf{q}_0\cdot(\mathbf{q}_1\times\mathbf{r}_2 + \mathbf{r}_1\times\mathbf{q}_2) + \mathbf{r}_0\cdot(\mathbf{q}_1\times\mathbf{q}_2)], \\ b_2 &= 1/3[\mathbf{r}_0\cdot(\mathbf{q}_1\times\mathbf{r}_2 + \mathbf{r}_1\times\mathbf{q}_2) + \mathbf{q}_0\cdot(\mathbf{r}_1\times\mathbf{r}_2)], \\ b_3 &= \mathbf{r}_0\cdot(\mathbf{r}_1\times\mathbf{r}_2). \end{aligned} \tag{18}$$

We exploit the convex hull property of the Bernstein basis for our test. By simply checking if all of the coefficients, b_i, $i \in \{0,1,2,3\}$, are negative, we know that the triangle is conservatively backfacing. This is a coarser test than testing against the true maximum of the cubic polynomial, but reduces the risk of numerical precision issues. Note that b_0 and b_3 are the backface tests at t=0 and t=1 respectively. The test can be refined by applying de Casteljau steps to the coefficients and testing if any generated coefficient is positive.

2.2 Higher-Order Vertex Motion

If the triangle vertex motion can be expressed as a polynomial, we can generalize the previous test. We express the motion of each triangle vertex as a Bézier curve of degree n in 2DH:

$$\mathbf{p}_i(t) = \sum_{j=0}^{n} \mathbf{b}_j^i B_j^n(t). \tag{19}$$

The backface test then becomes:

$$\begin{aligned} \det(\mathbf{M}(t)) &= \mathbf{p}_0(t) \cdot (\mathbf{p}_1(t) \times \mathbf{p}_2(t)) \\ &= \sum_{i=0}^{n} \mathbf{b}_i^0 B_i^n(t) \cdot \Big(\sum_{j=0}^{n} \mathbf{b}_j^1 B_j^n(t) \times \sum_{k=0}^{n} \mathbf{b}_k^2 B_k^n(t)\Big) \\ &= \sum_{i,j,k=0}^{n} B_{i+j+k}^{3n} \frac{\binom{n}{i}\binom{n}{j}\binom{n}{k}}{\binom{3n}{i+j+k}} \mathbf{b}_i^0 \cdot (\mathbf{b}_j^1 \times \mathbf{b}_k^2). \end{aligned}$$

This is a Bézier curve of degree $3n$, where the control points are sums of scaled determinants of three control points, one from each of the three curves describing the vertex motion. A conservative backface test can again be derived by using the convex hull property and testing the sign of the (scalar) control points. As expected, we obtain Equation 18 for the linear motion case, i.e., when $n = 1$.

2.3 Backface Culling for Motion Blur in Screen Space

For screen-space rasterization, a common backface test is given by the sign of the screen-space area of the triangle. Let us define two edges of the projected triangle as:

$$\mathbf{e}_1(t) = \frac{\mathbf{p}_1(t)}{p_{1_w}(t)} - \frac{\mathbf{p}_0(t)}{p_{0_w}(t)}, \quad \mathbf{e}_2(t) = \frac{\mathbf{p}_2(t)}{p_{2_w}(t)} - \frac{\mathbf{p}_0(t)}{p_{0_w}(t)}. \tag{20}$$

Twice the signed area can now be expressed as:

$$2A(t) = \mathbf{e}_1(t) \times \mathbf{e}_2(t) = \frac{p_{2_w}\mathbf{p}_0 \times \mathbf{p}_1 + p_{0_w}\mathbf{p}_1 \times \mathbf{p}_2 + p_{1_w}\mathbf{p}_2 \times \mathbf{p}_0}{p_{0_w} p_{1_w} p_{2_w}}. \tag{21}$$

Recall that that each vertex is a function of t (Equation 5), which results in that the backface test in screen space is a cubic rational function in t. The triangle moves

in a plane, but the vertex positions are no longer linearly interpolated in t and the triangle can change facing at most three times.

The magnitude of the denominator $p_{0_w} p_{1_w} p_{2_w}$ is irrelevant for the area test, so if we know the signs of the vertices w components, the denominator can be omitted. Similar to the homogeneous case, the signed area is a cubic polynomial.

Linear motion in screen space For this case, the area function becomes: $2A(t) = (\bar{\mathbf{p}}_1(t) - \bar{\mathbf{p}}_0(t)) \times (\bar{\mathbf{p}}_2(t) - \bar{\mathbf{p}}_0(t))$, where $\bar{\mathbf{p}}_i$ are moving vertices derived from linearly interpolating the *projected* start ($\bar{\mathbf{q}}_i$) and end ($\bar{\mathbf{r}}_i$) vertices. This is a quadratic polynomial, so even in this case, the moving triangle can be backfacing at $t = 0$ and $t = 1$, and still be frontfacing somewhere in between. This is precisely the scenario illustrated in Figure 1.

3 Backface Culling for Depth of Field

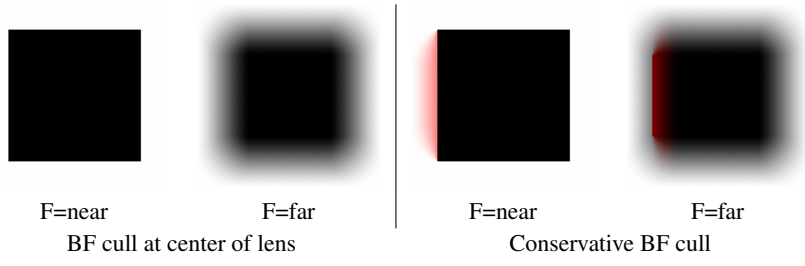


Figure 3: A red cube with a black front face is rendered with depth of field. Focus is set at the front and back of the cube. The cube is positioned so that the left face has a normal perpendicular to the view vector at the center of the lens. The left face can thus be backface culled for that camera position. For rays starting slightly off-center, the left (red) side is visible, and contributes to the final image. The two leftmost images show the result when the backface test is done only at the center of the lens, and the two rightmost images the reference result.

A naïve backface test for depth of field (DOF) is to only check the backface status at the center of the lens. Figure 3 shows that this is not correct. Depth of field is a shear in clip space [2], which can be represented by applying the matrix [8]:

$$\mathbf{S}(u,v) = \begin{pmatrix} 1 & 0 & -Hu/J & Hu \\ 0 & 1 & -Iv/J & Iv \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{22}$$

to the clip space coordinates of the triangle. H, I and J are constants given by the location of the focal plane, the camera aperture size and the near & far plane. The location on the lens is given by (u,v). Applying this matrix to a three-dimensional

homogeneous vertex, $\hat{\mathbf{p}}_i = (p_{i_x}, p_{i_y}, p_{i_z}, p_{i_w})$, in clip space results in a sheared position

$$\hat{\mathbf{s}}_i(u,v) = (p_{i_x} + \alpha_i u, p_{i_y} + \beta_i v, p_{i_z}, p_{i_w}) \tag{23}$$

where $\alpha_i = -H/J p_{i_z} + H p_{i_w}$ and $\beta_i = -I/J p_{i_z} + I p_{i_w}$. To simplify notation below, we introduce $\mathbf{l}_i = (\alpha_i u, \beta_i v, 0)$ and let $\mathbf{s}_i(u,v)$ and $\mathbf{p}_i(u,v)$ denote the two-dimensional homogeneous vertices, consisting of the *xyw* components of $\hat{\mathbf{s}}_i(u,v)$ and $\hat{\mathbf{p}}_i$ respectively, that is:

$$\mathbf{s}_i(u,v) = \mathbf{p}_i + \mathbf{l}_i(u,v). \tag{24}$$

Note that $\mathbf{l}_i \times \mathbf{l}_j = uv(0,0,\alpha_i\beta_j - \alpha_j\beta_i) = 0$ since $\alpha_i = \frac{H}{I}\beta_i$. The backfacing criterion then becomes:

$$\begin{aligned} \det(\mathbf{M}(u,v)) &= \mathbf{s}_0 \cdot (\mathbf{s}_1 \times \mathbf{s}_2) \\ &= \mathbf{p}_0 \cdot (\mathbf{p}_1 \times \mathbf{p}_2) + \mathbf{l}_0 \cdot (\mathbf{p}_1 \times \mathbf{p}_2) \\ &\quad + \mathbf{l}_1 \cdot (\mathbf{p}_2 \times \mathbf{p}_0) + \mathbf{l}_2 \cdot (\mathbf{p}_0 \times \mathbf{p}_1) \\ &= au + bv + c. \end{aligned} \tag{25}$$

The coefficients are given by:

$$\begin{aligned} a &= \alpha_0(\mathbf{p}_1 \times \mathbf{p}_2)_x + \alpha_1(\mathbf{p}_2 \times \mathbf{p}_0)_x + \alpha_2(\mathbf{p}_0 \times \mathbf{p}_1)_x \\ b &= \beta_0(\mathbf{p}_1 \times \mathbf{p}_2)_y + \beta_1(\mathbf{p}_2 \times \mathbf{p}_0)_y + \beta_2(\mathbf{p}_0 \times \mathbf{p}_1)_y \\ c &= \mathbf{p}_0 \cdot (\mathbf{p}_1 \times \mathbf{p}_2). \end{aligned} \tag{26}$$

The triangle will change its facing somewhere on a circular lens with radius R only if there is a solution to the following system of equations:

$$\begin{aligned} au + bv + c &= 0, \\ u^2 + v^2 &< R^2. \end{aligned} \tag{27}$$

Geometrically, this is an intersection between a circle and a line, which has solutions only if $c^2 \leq R^2(a^2 + b^2)$. Consequently, if the triangle is backfacing at the center of the lens ($c < 0$) and there are no face changes when moving over the lens, the triangle can be backface culled. More formally, if:

$$\begin{aligned} c &< 0, \quad \text{and} \\ c^2 &> R^2(a^2 + b^2), \end{aligned} \tag{28}$$

the triangle can be conservatively backface culled. Intuitively, the triangle changes facing over the lens only if the triangle's plane equation (in three dimensions) intersects the shape of the lens.

4 Conservative Backface Culling in 5D

By multiplying the moving vertex in Equation 5 with the shear matrix, $\mathbf{S}$, in Equation 22, the resulting vertex displacement from motion *and* DOF, $\mathbf{o}(u,v,t)$, is ob-

tained:

$$\begin{aligned}\hat{\mathbf{o}}_i(u,v,t) &= \mathbf{S}(u,v)\hat{\mathbf{p}}_i(t) = \mathbf{S}(u,v)((1-t)\hat{\mathbf{q}}_i + t\hat{\mathbf{r}}_i),\\ \mathbf{o}_i(u,v,t) &= (p_{i_x}(t) + \alpha_i(t)u, p_{i_y}(t) + \beta_i(t)v, p_{i_w}(t)),\end{aligned} \tag{29}$$

where $\alpha_i(t) = H/Jp_{i_z}(t) + Hp_{i_w}(t)$ and $\beta_i(t) = I/Jp_{i_z}(t) + Ip_{i_w}(t)$ are linear functions in t.

The corresponding backface test from Equation 25 is now expressed as:

$$\det(\mathbf{M}(u,v,t)) = \mathbf{o}_0 \cdot (\mathbf{o}_1 \times \mathbf{o}_2) = a(t)u + b(t)v + c(t). \tag{30}$$

The coefficients $a(t)$, $b(t)$, and $c(t)$ are cubic functions in t, and following Equation 28, the triangle can be conservatively backface culled when:

$$\begin{aligned}c(t) &< 0, \quad \text{and}\\ c^2(t) &> R^2(a^2(t) + b^2(t)), \quad t \in [0,1].\end{aligned} \tag{31}$$

4.1 Practical 5D Backface Culling Test

In this section, we sketch a practical implementation of the backface culling test for motion blurred and defocused triangles.

Solving Equation 31 using a numerical root solver is unlikely to be worth the effort. Instead, interval arithmetic [6] can be used, where we denote an interval $\hat{c} = [\min_t(c), \max_t(c)] = [\underline{c}, \bar{c}]$. A conservative test is then:

$$\begin{aligned}\bar{c} &< 0, \quad \text{and}\\ \bar{c}^2 &> R^2(\max(a^2 + b^2)), \quad t \in [0,1],\end{aligned} \tag{32}$$

where we used $\bar{c} < 0 \Rightarrow \min(\underline{c}^2, \bar{c}^2) = \bar{c}^2$ to simplify the second condition. We first note that a coarse, but fast approximation of the upper bound of $a^2(t) + b^2(t)$ is given by:

$$\max_{t\in[0,1]} \left(a^2 + b^2\right) \leq \max_{t\in[0,1]} (\underline{a}^2, \bar{a}^2) + \max_{t\in[0,1]} (\underline{b}^2, \bar{b}^2), \tag{33}$$

which essentially is a Manhattan distance approximation.

In the scenes tested, the t^3 and t^2 terms of the polynomials $a(t), b(t)$ and $c(t)$ are close to zero for most triangles. This implies that the $a(t), b(t)$ and $c(t)$ terms in Equation 32 are approximately linear, but also that care must be taken to avoid precision issues. By using first-order Taylor models [4], we ensure stability when the t^3 and t^2 terms are small, but preserves the linear dependence in t. An arbitrary cubic polynomial is bounded over $t \in [0,1]$ by:

$$k_3t^3 + k_2t^2 + k_1t + k_0 \approx k_1t + k_0 + \hat{r}_k, \tag{34}$$

where $\hat{r}$ is a remainder interval which bounds the quadratic and cubic terms:

$$\hat{r}_k = [-|k_3| - |k_2|, |k_3| + |k_2|]. \tag{35}$$

We use this to conservatively express $a^2(t)+b^2(t)$ as:

$$a^2(t)+b^2(t) \approx (a_1t+a_0+\hat{r}_a)^2+(b_1t+b_0+\hat{r}_b)^2. \quad (36)$$

A conservative upper bound is given by:

$$\max_{t\in[0,1]}\left(a^2(t)+b^2(t)\right) \leq a_1^2+b_1^2+2\max(0,a_0a_1+b_0b_1)+a_0^2+b_0^2+\bar{r}_{a^2+b^2}, \quad (37)$$

where $\bar{r}_{a^2+b^2}$ is a linear function in the remainder intervals $\hat{r}_a$ and $\hat{r}_b$. If $a(t)$ and $b(t)$ are linear in t, $\hat{r}_a=\hat{r}_b=0$ and $\bar{r}_{a^2+b^2}=0$. We use the backface test for motion blur to determine the backface status at the center of the lens as described above. The final test is given by the conditions in Equation 32, where $a^2(t)+b^2(t)$ is bounded using Equation 37.

Square Lens Approximation If we approximate the lens with a square with side length $2R$, we get a coarser backfacing conditions given by the equations:

$$\begin{aligned} \overline{c} &< 0, \quad \text{and} \\ \max(\pm R(a\pm b)+c) &< 0, \quad t\in[0,1]. \end{aligned} \quad (38)$$

However, our experiments show that Equation 32 using the Taylor model bounds (Equation 37) gives slightly higher cull rates, as the vast majority of false backfacing decisions are due to DOF. This indicates that the test should be optimized for DOF.

5 Results

In this section, statistics are gathered from a number of key-framed animations taken from the Utah Animation Repository[1] and the UNC Dynamic Scene Benchmark.[2] We denote the test of only checking backface status at the start and end time as STARTEND in the evaluation below.

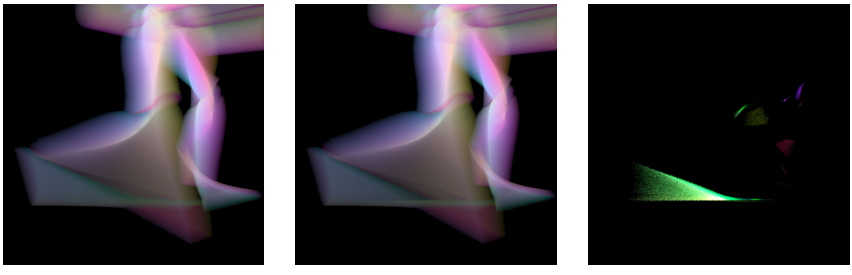
5.1 Cost Estimation

The cost in scalar instructions of the different tests are given in Table 1, with the assumption that the edge equation coefficients (Equation 15) can be reused. For DOF, the coefficients α_i and β_i are triangle constants needed for stochastic sampling and are therefore not included in the cost. The correct backface test costs less than a single inside tests (using time-dependent edge functions) for the motion blur *only* or DOF *only* case. For the full 5D test, we do not use 5D edge equations, but instead position the triangle for each *uvt* tuple. Therefore, no triangle setup

[1] `http://www.sci.utah.edu/~wald/animrep/`
[2] `http://gamma.cs.unc.edu/DYNAMICB/`

Instr.	INSIDE	STARTEND	BFMB	BFDOF	INSIDE5D	BF5D
MAD/MSUB	24	4	12	7	39	100
MUL	-	2	6	6	27	54
ADD/SUB	-	-	3	-	1	60
CMP	3	2	4	1	3	2
MIN/ABS	-	-	-	-	-	14
SUM:	**27**	**8**	**25**	**14**	**70**	**$>$ 230**

Table 1: Cost of correct backface tests for motion blur and depth of field. INSIDE *reports the cost for one per-sample inside test (without tiebreaker rules).* INSIDE5D *is the positioning and inside test for full 5D rasterization.* BF5D *denotes the 5D backface test with depth of field and motion blur, and is a lower estimate with coarse bounds of the involved cubic functions.*



Correct BF | STARTEND | Diff image ($\times$20)

Figure 4: A frame from the Wood Doll animation ($\Delta = 10$) with a camera rotating around the character ($\pi/2$ rad/frame), using 256 spp. In this example, the STARTEND *test falsely rejects about 6% of the triangles. In total, 11k (4%) pixels differ and 5.7k (2%) pixels fail a perceptual test and the PSNR is 47dB.*

computations can be reused, and the backface test becomes considerably more expensive. As a lower limit, a 5D backface test with coarse bounds of the cubic polynomials involved in the expressions, costs $> 3\times$ of a single 5D inside test.

5.2 Motion Blur

All examples were rendered in 512×512 pixels using 64 samples per pixel. The motion blur test scenes are shown in Figure 5 with three different motion blur settings. For each animation, we report the percentage of triangles potentially visible, but rejected by the STARTEND test. The cubic backfacing function is bounded using a subdivided Bézier control cage (using 10 control points), so the percentage reported is slightly higher than the actual miss rate. The number of affected fragments are reported by the per-sample inside test and is an exact measure of the impact on the final result. The results are presented in Table 2.

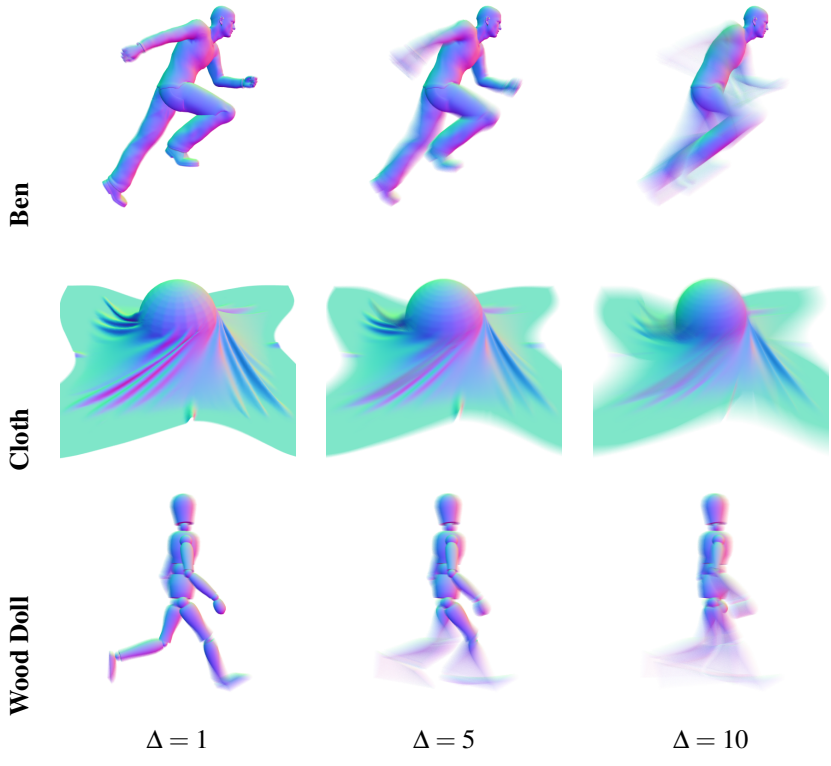


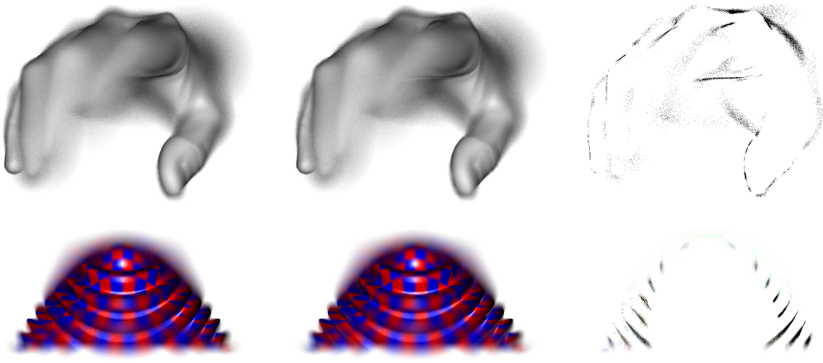
Figure 5: The test scenes with different animation speeds. Δ denotes the steps in keyframes in the animations.

The **Ben** animation shows a running man with 78029 triangles. **Cloth** contains complex motion and a large triangle count (92230). **Wood Doll** is a simple rigid body animation with modest triangle count (5378 triangles). A camera motion has been added to highlight the common case when both the object and the camera are animated. Figure 4 shows a zoomed in version of the animation with exaggerated camera rotation. Using a perceptual metric [9], 2% of the pixels differ for this frame.

For motion blur, the number of triangles falsely rejected by the STARTEND test is modest, and the visual impact is negligible. It is indeed possible to construct cases where this matters (see Section 1), but for many practical examples with small triangles, closed meshes and limited motion per frame, we conclude that the STARTEND backface test is sufficient.

	$\Delta = 1$		$\Delta = 5$		$\Delta = 10$	
Name	rej.tris (%)	$\bar{s}$	rej.tris (%)	$\bar{s}$	rej.tris (%)	$\bar{s}$
Ben	0.1	13	0.2	726	0.7	4450
Cloth	0.004	78	0.02	425	0.1	4162
Wood Doll	0	0	0.1	532	0.3	1539

Table 2: Statistics gathered from animations with motion blur. Δ indicates the step in key frames between the start and end position of the triangle. For each Δ, we report the percentage of triangles potentially visible but culled by the STARTEND *backface test, and $\bar{s}$ reports the maximum number of samples in any frame that comes from triangles falsely culled by the* STARTEND *test.*



Exact backface test | Center of lens BF test | Difference $\times 10$

Figure 6: Two examples with large defocus blur with a correct backface test (left) and the center of the lens test (middle). Note the white streak in the blur on top of the hand in the middle image, and the missing occlusion on the right side of the displaced patch.

5.3 Depth of Field

For large apertures, the number of frontfacing triangles increases. The naïve test of only checking backface status in the center of the lens gives visible artifacts around silhouettes. This is shown in Figure 3 and Figure 6. The correct test is inexpensive and we recommend that it is always enabled.

5.4 Depth of Field and Motion Blur

For the combined case of both motion blur and DOF, the vast majority of artifacts stems from DOF. In our experience the tests should be tuned for a tight DOF test and a coarser motion blur test, as presented in Section 4.1. An example of this is shown in Figure 7.

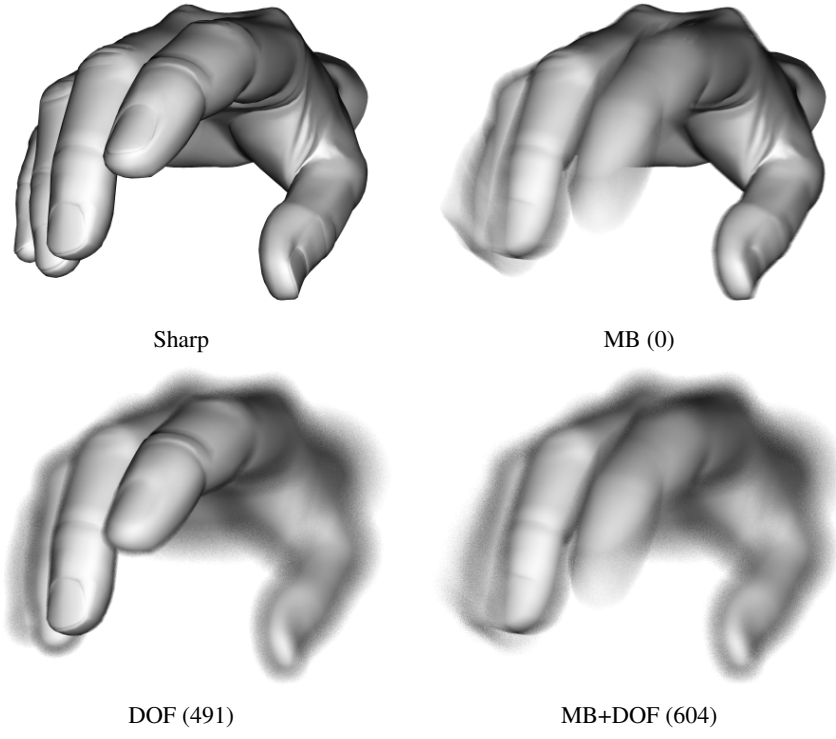


Figure 7: An example with both motion blur and depth of field. The numbers indicate how many triangles that are falsely culled by the incorrect backface test for each configuration. The model contains 15855 triangles.

Acknowledgements

Thanks to Jon Hasselgren and Robert Toth for many fruitful discussions and help with generating illustrations. Tomas Akenine-Möller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation. In addition, we acknowledge support from the Swedish Foundation for strategic research.

Bibliography

[1] Tomas Akenine-Möller, Jacob Munkberg, and Jon Hasselgren. Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware*, pages 7–16, 2007.

[2] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 18(3):137–145, 1984.

[3] Carl Johan Gribel, Michael Doggett, and Tomas Akenine-Möller. Analytical Motion Blur Rasterization with Compression. In *High Performance Graphics*, pages 163–172, 2010.

[4] Kyoko Makino and Martin Berz. Taylor Models and Other Validated Functional Inclusion Methods. *International Journal of Pure and Applied Mathematics,*, 4(4):379–456, 2003.

[5] Michael D. McCool, Chris Wales, and Kevin Moule. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. In *Graphics Hardware*, pages 65–72, 2002.

[6] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[7] Marc Olano and Trey Greer. Triangle Scan Conversion using 2D Homogeneous Coordinates. In *Graphics Hardware*, pages 89–95, 1997.

[8] Jonathan Ragan-Kelley, Jaakko Lethinen, Jiawen Chen, Michael Doggett, and Fredo Durand. Decoupled Sampling For Real-Time Graphics Pipelines. *To appear in ACM Transactions on Graphics*.

[9] Hector Yee. A Perceptual Metric For Production Testing. *Journal of Graphics Tools*, 9(4):33–40, 2004.