Algorithmic Improvements for Stochastic Rasterization & Depth Buffering

Magnus Andersson
Department of Computer Science
Lund University

ISBN (Printed) 978-91-7623-442-6
ISBN (Electronic) 978-91-7623-443-3
ISSN 1404-1219
Dissertation 48, 2015
LU-CS-DISS 2015-04

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

Email: magnus.andersson@cs.lth.se
WWW: http://www.cs.lth.se/home/Magnus_Andersson

Typeset using LaTeX2ε
Printed in Sweden by Tryckeriet i E-huset, Lund, 2015

Abstract

The field of computer graphics refers to the use of computers to generate realistic-looking images from virtual scenes. Graphics processing units use an algorithm known as rasterization to compute images of scenes viewed from a virtual camera. The commonly used pinhole camera model does not account for the imperfections that stem from the physical limitations in real-world cameras. This includes, for example, motion and defocus blur. These two phenomena can be captured using stochastic rasterization, which is an algorithm that extends upon conventional rasterization by being able to handle moving and out-of-focus objects. Using this approach, the virtual scene is sampled at different instances in time and using different paths through the camera lens system. Alas, the extended functionality comes at a higher computational cost and consumes much more memory bandwidth. Much of the increased bandwidth usage is due to the increase in traffic to the depth buffer. The focus of the six papers included in this thesis is three-fold. First, we have explored ways to reduce the high memory bandwidth consumption inherent in depth buffering, targeting both conventional and stochastic rasterization. We have evaluated a number of hardware changes, including novel compression schemes and cache improvements, which efficiently reduce memory bandwidth usage. We also propose a hardware friendly algorithm which reduces the pressure on the depth buffering system by culling unnecessary work early in the pipeline. Second, we propose an algorithm to reduce shading computations for stochastic rasterization. In our approach, we decouple shading and visibility determination into two separate passes. The surface color is sparsely evaluated in the first pass and can be efficiently used in the second pass, when rendering from the camera. The two-pass approach allows us to adaptively adjust the shading rate based on the amount of blur resulting from motion and defocus effects, which greatly reduces rendering times. Third, we propose a real-time algorithm for rendering shadows cast by objects in motion. Due to the complicated interplay between moving objects, moving light sources, and a moving camera, rendering motion blurred shadows is an especially difficult problem. Using our algorithm, high quality, smooth shadows can be achieved on conventional graphics processors. Collectively, I believe that our research is a significant step forward for rendering scenes with motion and/or defocus blur, both in terms of quality and performance.

Acknowledgements

I would like to thank my supervisor Tomas for believing in me and giving me the opportunity to pursue my PhD, and for providing invaluable help through all of my projects. A big thank you to Krisztina for her tremendous support and patience throughout my PhD endeavor, and for bringing snacks to all those late night deadlines. Thanks to my colleagues and co-authors at Intel for letting me steal their precious time with coffee fueled white board discussions; Jon, Jim, Robert, Björn, Petrik, and last, but not least my assistant supervisor Jacob. I would like to thank my colleagues at the university; Michael, Rasmus, Carl-Johan, Per, Jim, and Philip for all the valuable and entertaining discussions during the Grafikas$^{\text{TM}}$ and coffee runs. I am very grateful to the Intel Corporation for funding my research, and to everyone within the company who supported my studies throughout the years. I thank my brother Fredrik and my parents Ann and Göran for their support and encouragement, and my grandfather Ragnar, who unfortunately did not get to see me cross the finish line.

Preface

The following papers are included:

I. Jon Hasselgren, Magnus Andersson, Jim Nilsson, and Tomas Akenine-Möller, "A Compressed Depth Cache", in *Journal of Computer Graphics Techniques*, vol. 1, no. 1, pp. 101–118, 2012.

II. Magnus Andersson, Jon Hasselgren, and Tomas Akenine-Möller, "Depth Buffer Compression for Stochastic Motion Blur Rasterization", in *High Performance Graphics*, pp. 127–134, 2011.

III. Magnus Andersson, Jacob Munkberg, and Tomas Akenine-Möller, "Stochastic Depth Buffer Compression using Generalized Plane Encoding", in *Computer Graphics Forum (Proceedings of Eurographics)*, vol. 32, no. 2, pp. 103–112, 2013.

IV. Magnus Andersson, Jon Hasselgren, and Tomas Akenine-Möller, "Masked Depth Culling for Graphics Hardware", to appear in *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 2015.

V. Magnus Andersson, Jon Hasselgren, Jacob Munkberg, and Tomas Akenine-Möller, "Filtered Stochastic Shadow Mapping Using a Layered Approach", to appear in *Computer Graphics Forum*, 2015.

VI. Magnus Andersson, Jon Hasselgren, Robert Toth, and Tomas Akenine-Möller, "Adaptive Texture Space Shading for Stochastic Rendering", in *Computer Graphics Forum (Proceedings of Eurographics)*, vol. 33, no. 2, pp. 341–350, 2014.

The following paper is also published but is not included in this thesis:

- Magnus Andersson, Björn Johnsson, Jacob Munkberg, Petrik Clarberg, Jon Hasselgren, and Tomas Akenine-Möller "Efficient Multi-view Ray Tracing using Edge Detection and Shader Reuse" in *Visual Computer*, vol. 27, no. 6–8, pp. 665–676, 2011.

Contents

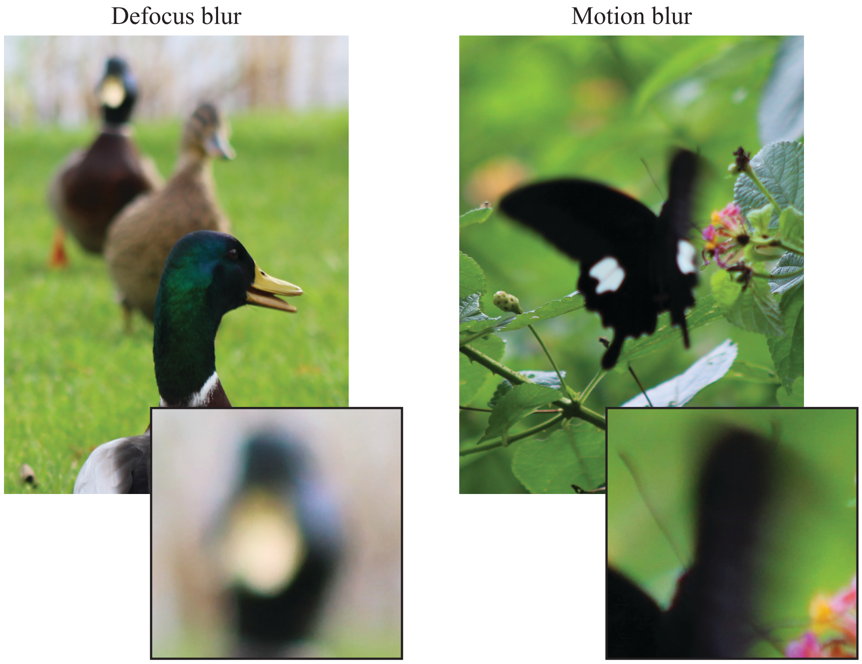


Figure 1: Left: *a duck is smirking because he is in focus while his friends are blurry in the background.* Right: *a butterfly is frivolously flapping its wings, creating trails of motion blur. (Photographs courtesy of Fredrik Andersson)*

1 Introduction

Algorithms in *computer graphics* are used to generate realistic looking images by simulating how light behaves and interacts with objects in the real world, before reaching the eyes of the observer. As a species of avid moviegoers and (mostly) amateur photographers, we have arguably gotten as used to viewing digital images indirectly through cameras as directly through our own eyes. Advanced camera models can be used to model the lens system and the shutter of the camera in order to achieve effects such as *defocus* and *motion blur*, shown in Figure 1.

The human visual system is an intricate interplay of optics and psychology, where different visual cues are subconsciously processed and interpreted by the brain. This must also be taken into account when creating realistic-looking images. For example, the shadows cast by an object can give us information about the size and shape of the object, the distance to the shadowed surface, and the location of the light source. Absence of expected phenomena such as shadows may disturb the viewing experience, which makes the image look artificial, as exemplified in Figure 2.

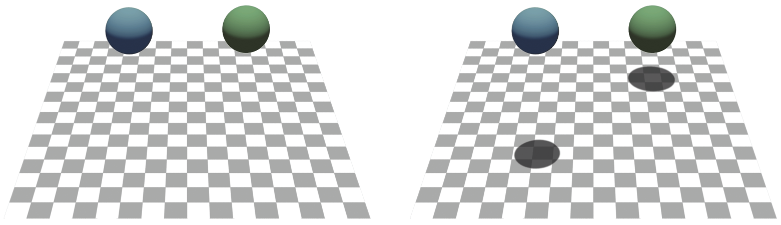


Figure 2: A simple virtual scene with and without shadows. The sizes and locations of the spheres are ambiguous when shadows are absent.

As more and more effects, such as defocus blur, motion blur, and shadows are included in the image generation process, the computational cost unfortunately increases. In this thesis, we propose several hardware improvements to graphics processing units (GPUs) to alleviate this cost, primarily by lowering the memory bandwidth usage. We also present two novel algorithms targeting current generation GPUs. The first is designed to efficiently calculate motion blurred shadows and the second is used to quickly compute surface color in presence of motion and defocus blur.

1.1 Real-Time Computer Graphics

Ideally, the computer generated image or sequence of images, should be indistinguishable from viewing the same scene in real life. This is a notoriously difficult problem, since the light in the real world bounces around and changes properties when it interacts with the elements in the scene before reaching the viewer. There are numerous phenomena in the physical world to account for. For example, when a photon reaches a surface it may be reflected in a seemingly random direction due to the surface's micro structure. There is also a possibility that the photon will be absorbed and thus never reaches the viewer. If the surface is fluorescent, light with a wavelength different from that of the absorbed photon may be emitted. Materials can be semi-transparent and the light path can be altered through refractions and reflections. Even the tiniest speck of dust will interact with some of the light in the scene and therefore ultimately change the appearance of the image.

The process of converting a scene into an image is referred to as *rendering* the scene and the underlying system responsible for performing this task is commonly called the *renderer*. To obtain a realistic-looking image, the renderer needs to utilize accurate physically-based models for how light interacts with everything in the scene. Preferably in as short amount of time as possible.

In many applications and/or environments, some physical phenomena are subtle or non-existent, and can therefore be omitted in order to simplify the rendering process. Since rendering is always time-constrained, sacrificing some physical

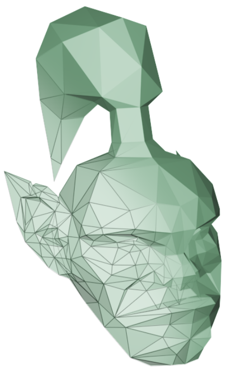


Figure 3: A triangle mesh representing an ogre head.

correctness for simplicity may enable the renderer to spend more compute cycles on the parts of the light simulation that ends up having a greater impact on the final image. This is especially important for *real-time* graphics applications, such as games and other dynamic and interactive environments. To give the appearance of fluid motion, the image displayed on the screen should preferably be updated at rates typically ranging from 30 – 90 Hz, i.e., a new image (or *frame*) should be delivered by the renderer roughly every 11 – 33 ms.

Although the discussion so far has circled around producing realistic images, realism may not always be the desired result, time constraints aside. In computer graphics, we are not bound by the physical laws of reality and as such, it is possible to exaggerate or diminish some effects or to give in to artistic impulses. Perhaps most important, however, is the prospect of "fooling" the user by getting away with substituting a complex effect with a simpler one. In many situations it may be difficult to determine exactly what the correct image should look like, which gives ample opportunities for the renderer to cut corners and use believable approximations in place of the real, ground truth effects. For example, it is common to use the greatly simplified *thin lens model* (see Section 4) in place of the intricate lens system of a real camera when rendering defocus blur. This greatly simplifies computations and the difference is indistinguishable to most users and for most real-time applications.

1.2 Building a Virtual Scene

Before diving into the details of some of the algorithms used in rendering, we must first establish some different constructs used to describe a virtual scene. For the purpose of this thesis, the virtual scene is a collection of three-dimensional geometric objects, light sources, and cameras, all of which may be animated over time.

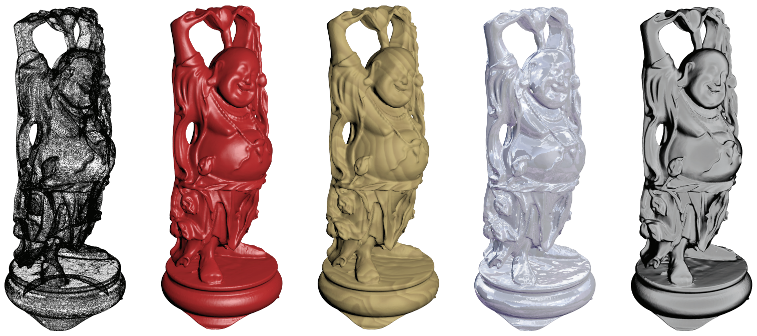


Figure 4: A triangle mesh depicting a laughing Buddha statue with various material properties. The leftmost image shows the finely detailed geometry wire-frame. (Happy Buddha model courtesy of Stanford University Computer Graphics Laboratory)

Geometry The behavior of light interacting with the surface of opaque geometry is arguably the most important phenomenon to capture. The simplest primitive that describes a surface in three-dimensional space is a triangle, which is defined by its three corner points or *vertices*. Each vertex, $\mathbf{q}$, is in turn composed of three coordinates such that $\mathbf{q} = (q_x, q_y, q_z)$. Each pair of vertices forms one *edge* of the triangle. By joining multiple triangles together along the edges and at the vertices, a *triangle mesh* (or *mesh* for short) can be created. An example of a triangle mesh is shown in Figure 3. Given sufficiently many triangles, a mesh can be built to approximate any geometric surface up to any accuracy. Although other geometric primitives could be used to this effect, due to its simplicity, the triangle is the predominant primitive of choice for most rendering systems.

Light-surface interaction The appearance of opaque geometric objects is determined by how light interacts with the surface materials. There is a vast difference in how different materials interact with light, as illustrated in Figure 4.

The *rendering equation* [62] is often used to describe the light/surface interactions within a scene. Following the illustration in Figure 5, given a surface point, $\mathbf{p}_x$, observed from another point in space, $\mathbf{p}_o$, the amount of that light reaching $\mathbf{p}_o$ from $\mathbf{p}_x$ can be calculated as

$$L_o(\mathbf{p}_o, \mathbf{p}_x) = v(\mathbf{p}_o, \mathbf{p}_x) \left(L_e(\mathbf{p}_o, \mathbf{p}_x) + \int_s L_i(\mathbf{p}_x, \mathbf{p}_i) f_r(\mathbf{p}_o, \mathbf{p}_x, \mathbf{p}_i) \, d\mathbf{p}_i \right), \qquad (1)$$

where L_o is the radiance (outgoing light), L_i the irradiance (incoming light), and the term L_e is the emitted radiance. The variable $\mathbf{p}_i$ integrates the set of all surface points, s, in the scene since *all* other points have a potential light contribution.

The *bi-directional reflectance distribution function* (BRDF) [95], f_r, describes the ratio of incoming and outgoing light in the directions of $\mathbf{p}_i$ and $\mathbf{p}_o$ at the surface

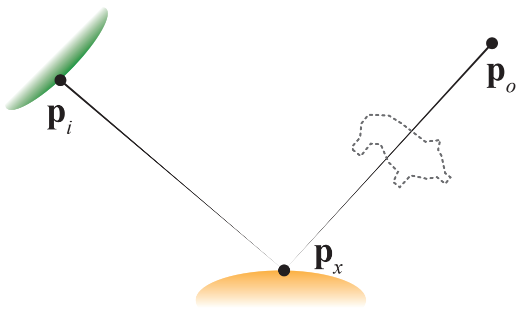


Figure 5: A surface point, $\mathbf{p}_x$, observed from another point, $\mathbf{p}_o$. The amount of light reaching $\mathbf{p}_o$ depends the incoming light from all other points in the scene, $\mathbf{p}_i$, whether there are occluding objects between $\mathbf{p}_x$ and $\mathbf{p}_o$, on the material of the surface, and on the amount of light that is spontaneously emitted from $\mathbf{p}_x$.

point. The BRDF is essentially the material description of the surface, which includes the finer details that are not present in the geometric representation, such as microscopic roughness and color dye.

There may be opaque or semi-transparent objects located on the path between $\mathbf{p}_o$ and $\mathbf{p}_x$ occluding the light, which is accounted for by the visibility factor, v. For scenes containing opaque geometry only, v can only assume the values 0 or 1.

Surfaces with a non-zero emitted radiance, $L_e \neq 0$, act as a light sources. In real-time graphics, it is common to illuminate the scene using light sources with infinitesimal extent, so-called *point light sources*. The point $\mathbf{p}_o$ is lit *directly* by the emissive point situated at $\mathbf{p}_x$ if $v = 1$, i.e., when there are no objects obstructing the direct path between the light source and $\mathbf{p}_o$. Otherwise, the point is in shadow. It could, however, still be illuminated *indirectly* by light bouncing off other surfaces. Other types of light sources commonly used include area lights and environment maps.

The version of the rendering equation outlined above is an approximation and omits some intricate effects, such as subsurface scattering and light polarization. Even so, the equation cannot be solved analytically due to the recursion in the integral. However, it is possible to iteratively refine the solution, for example by using *Monte Carlo integration*, where the integral is *point sampled* using random samples [26]. As more samples are used, the result will approach the limit image or *ground truth*. It is often sufficient to use relatively few recursion steps, depending on the scene and application. Even a single step can produce plausible results, since the most significant contribution often comes from surfaces directly illuminated by the light sources in the scene.

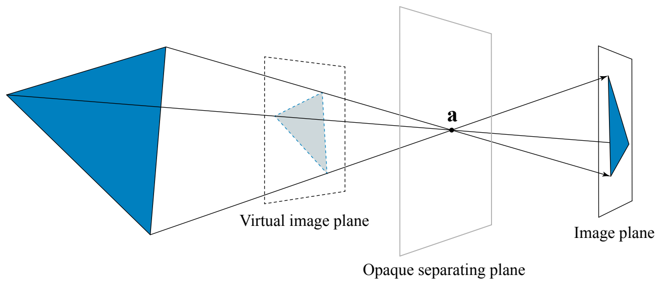


Figure 6: The pinhole camera model. The image plane is only exposed to the light coming through the aperture point **a***.*

Camera The final component needed in our virtual scene description is the camera. It is responsible for capturing the light bouncing around in the scene in order to produce an image. The simplest and most commonly used type of camera is the *pinhole camera*, illustrated in Figure 6. An (infinite) opaque plane separates the motif from the *image plane*. The separating plane only has a small opening, or aperture point **a**, for the light to travel through. The aperture opening ensures that light from a particular direction, going through the point, will intersect the receiver image plane at one specific location. Together, all the incoming light rays form an image, flipped along both axes, on the image plane. In the virtual world, it is convenient to introduce an alternative image plane which is placed in front of point **a**, instead of behind it. The light rays intersect the virtual image plane as they travel towards the point **a**.

Generating an image from a virtual scene With the simple constructs outlined so far it is possible to describe a virtual scene and convert it into an image. There are numerous rendering techniques that could be used to accomplish this. Ray tracing [13, 119] and rasterization [39, 99] are the two most prominent techniques today and there is a great deal of research on these topics. They are both highly parallel processes, but they organize the rendering problem in different ways. Ray tracing is perhaps intuitively easier to understand. Using this method, each pixel in the image can be processed in isolation. For each pixel, a set of rays are first created which originate at the camera center (aperture point) and travel through the pixel. These rays are used to probe the scene, which amounts to searching for the closest intersection points among the scene geometry. Organizing the geometry in a data structure that is both fast to construct and quick to traverse, in order to find an intersection point, is key to good performance for ray tracing. In contrast, with rasterization the scene geometry is processed in a streaming fashion. For each triangle, the pixel overlap is determined and the depth and color are subsequently calculated. Thus, each triangle is processed only once, while each pixel may be visited multiple times by many different triangles. For real-time graphics, it is undoubtedly rasterization which is the most predominant technique and is built in to fixed-function hardware in modern consumer graphics processing units (GPUs).

1.3 Thesis Organization

This introductory chapter should hopefully give the reader sufficient background information to understand the papers included this thesis. How the GPU is used to generate images is explained in Section 2. Concepts such as the rendering pipeline, shaders, and depth buffers are introduced. Since one of the main areas of this thesis is depth buffering, the entirety of Section 3 is dedicated to the depth unit, which includes details on depth computation and representation, hardware implementation, and occlusion culling. Much of the research requires some insight into stochastic rasterization for motion and defocus blur, which is explained in Section 4. This includes a review of the camera models we used, as well as a discussion on how stochastic sampling affects and complicates depth buffering. In addition, a user-space stochastic rasterization implementation with motion and defocus blur is outlined. The topic of motion blurred shadows is addressed in Section 5, where I give a brief overview of the shadow mapping algorithm and how it can be extended to accommodate motion blurred shadows using stochastic rasterization. Section 6 describes how to reduce the computational burden when computing the sample color for scenes with motion and defocus blur. Brief descriptions of the contributions in each of the papers are interspersed where appropriate in the text and the author's contributions to the papers are summarized in Section 7. Finally, some conclusions and possible avenues for future work are found in Section 8. The order of the papers was chosen to give a coherent narrative, but were authored and published in the following chronological order: **II**, **I**, **III**, **VI**, **V**, and **IV**. The layout of the papers have been altered slightly to better fit the format of the thesis. The individual Bibliography sections for each of the papers have been merged to a common section, which can be found at the end of the thesis.

2 Hardware Accelerated Graphics

At the time of writing, there is (at least) one dedicated *graphics processing unit* (GPU) in nearly every desktop computer, gaming console, and handheld device produced. As the name implies, the GPU is responsible for the bulk of the graphics processing in the computer.

2.1 The Graphics Processing Unit

To render an image using a GPU, the scene data is accessed throughout a number of pipeline stages within the so-called *rendering pipeline*. By using a combination of programmable *shader cores* and *fixed-function hardware*, the GPU is able to quickly rasterize triangles and compute their color contribution to each pixel.

On the application side, a graphics API is used to communicate with the GPU. Currently, the two main APIs used in the industry are OpenGL [107] and DirectX [17], with many more emerging, such as Metal, Mantle, and Vulkan. Although there are

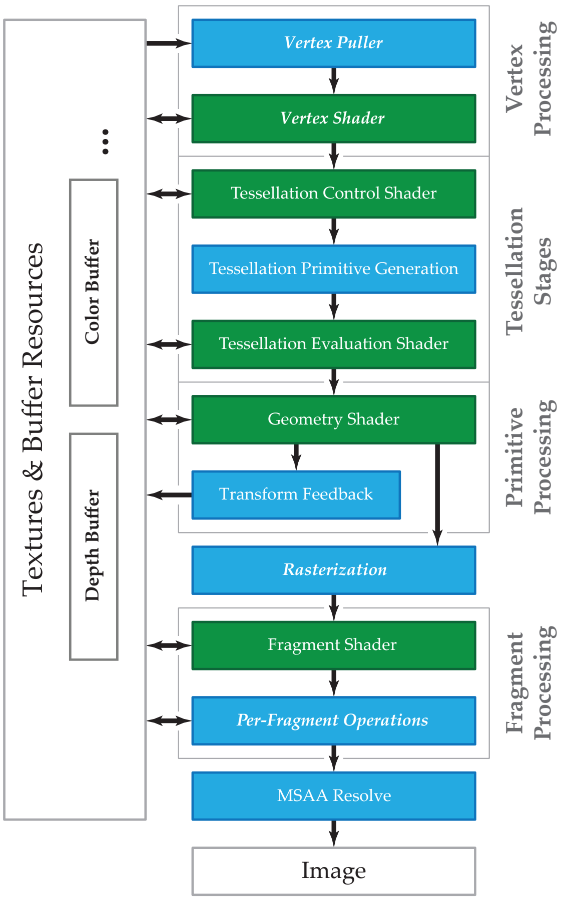


Figure 7: OpenGL 4.5 pipeline overview. The green stages are the programmable shaders and the blue stages are fixed-function. Most of the stages are optional, only the ones with bold, italic typeface are mandatory. It should be noted that compute shaders (not pictured) bypass the entire rendering pipeline by simply running directly on the shader cores, and can read and write to buffers in the GPU memory.

differences in the syntax and the programming model used, they serve the same purpose in controlling the GPU. There are also different versions of each API which adhere to the hardware capabilities of their contemporary GPUs. A simplified flow chart of the pipeline used in OpenGL 4.5 is pictured in Figure 7.

Some of the pipeline stages are programmable and are jointly called *shaders*. As an example, a fragment shader program is used to describe how to compute the output color of a pixel. For most rendering algorithms there are no data dependencies between the pixels and the shader program can therefore be executed in a highly parallel fashion for many pixels at a time. The *shader cores* running the programs are much simpler in their design than full-fledged CPUs and typically operate at a lower frequency, and only very limited inter-core communication is offered. Although the shader cores are less versatile than CPUs, they are smaller and more energy efficient, and many units can therefore be fitted in a small area to run in parallel, which suits most rendering techniques well. The placement of the fixed-function stages and intermittent shader stages can be viewed in Figure 7, and more detailed descriptions of their individual responsibilities are described next, in Section 2.2.

In addition to the regular rendering pipeline, it is also possible to launch isolated *compute shaders*. A compute shader program can read and write to the GPU resources and does not depend on the rest of the fixed-function pipeline. Similar to compute shaders in OpenGL and DirectX, programs that run directly on the shader cores can be written in languages such as OpenCL and CUDA.

2.2 The Rendering Pipeline

In the following section, we will outline the rendering pipeline as it is commonly used in OpenGL to generating an image with a pinhole camera. The first step is to make the scene resources accessible by the GPU, by uploading them to GPU memory. The resources include lists of triangles, texture images, transformation matrices, and so on.

2.2.1 Vertex Processing

The vertex puller reads per-vertex data from the lists of triangles. A vertex shader program is launched for each vertex, transforming its position to the coordinate frame of the camera. Additionally, other per-vertex *attributes* to be interpolated across the triangle surface are set up by the vertex shader. This includes, for example, normals and texture coordinates. Apart from triangles, modern graphics hardware also support point and line primitives, as well as tessellated surfaces.

2.2.2 Tessellation

In the optional tessellation stage, patch and triangle primitives can be tessellated to a new set of triangles. A patch is defined by a set of control vertices. They are interpreted by the tessellation control shader, which is executed once per control vertex and determines the tessellation rate across the patch. Fixed-function hardware subsequently performs the triangulation of the patch, but the actual positions of each output vertex is determined by the tessellation evaluation shader program. Geometric surface displacement effects can also be included in this shader stage. The entire tessellation stage is optional and can be bypassed.

2.2.3 Primitive Processing

The geometry shader is another optional shader stage that follows the tessellation stage. If enabled, the geometry shader program is executed once for every input primitive, which are either triangles, lines, or points. The output is a new set of primitives, which may have a different set of vertex attributes than the input mesh. In addition, primitives may be discarded at this stage. The primitive processing stage can be useful for generating the faces of a cube when rendering voxels or for extruding shadow volumes, for example. In Section 4.5, a geometry shader program is used to bound the blurred image region produced by triangles subjected to motion and defocus blur. Enabling the *transform feedback* feature terminates the pipeline at this stage and outputs the primitive data to a buffer that can be used in a subsequent pass through the pipeline. Otherwise, the primitives continue onward to the rasterizer.

2.2.4 Rasterization

The task of the *rasterizer* is to determine the image region covered by each incoming triangle. The coverage is point sampled in image space using one or more samples per pixel. To determine whether a sample point is overlapping the triangle or not, an *inside test* is used, where each of the triangle's *edge equations* [80, 99] are evaluated. The edge equations are derived from the triangle edges projected onto the image plane. If the equation evaluates to a zero for a sample it lies on the edge. Otherwise, the sign reveals on which side of the edge it is located. The sample is covered by the triangle if all three edge equation evaluations have the same sign.

2.2.5 Fragment Processing

Once triangle coverage is determined, each covered sample is assigned a color and a depth value. The depth is interpolated from the vertex positions and the color is computed through a fragment shader program. The color usually depends on the attributes of the triangle, such as the surface normal for lighting computations, and texture coordinates for surface color texture lookups. The attributes, which are defined in the vertices, are interpolated across the triangle surface by the GPU and the resulting values are accessible through the input arguments of the fragment shader. The fragment shader can then use these values to compute the color, for example using a set of light sources interacting with the surface BRDF. Once the color has been computed, or *shaded*, for a sample location, the sample is *depth tested* before potentially being stored in a *color buffer*. An example of a color buffer can be seen to the left in Figure 8. A detailed discussion on attribute and depth interpolation can be found in Section 3.1.

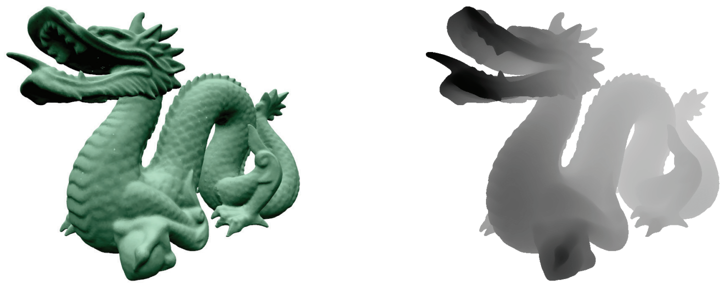


Figure 8: The color and depth buffers for a rendering of a dragon statue. Left: *the color buffer stores the shaded color value for each sample.* Right: *the distance from the camera to the closest surface point is stored in the depth buffer. The distance is visualized using a gray scale, where darker shades are closer to the camera and lighter shades are further away. (Dragon model courtesy of Stanford University Computer Graphics Laboratory).*

Depth testing Assuming opaque geometry, it is only the surface closest to the camera that contributes to the pixel color at each sample location. When rendering a scene, the triangles may be processed in some arbitrary order, but the resulting image should be identical regardless of that order. By utilizing a *depth buffer*, the distance to the closest surface seen *so far* at each sample location can be tracked throughout the rendering. The right image in Figure 8 shows an example of a depth buffer. The sample color in the color buffer and the depth value in the depth buffer are only replaced if the rendered surface point passes the depth test. For most applications a so called *less than* test is used, which passes if the new sample depth is closer than the previous entry.

MSAA Although the depth is computed on a per-sample basis, shading the color is usually done at a lower frequency. Unless otherwise specified by the user, the fragment shader program is only executed *once per triangle within each pixel*. In other words, in each pixel, the color value is replicated among all samples that overlap the same triangle. This method, which is illustrated in Figure 9, is called *multisample anti-aliasing* (MSAA). The additional effort of shading each and every sample does not justify the slight improvement in quality in most cases, making MSAA a useful optimization. In addition to outputting the color for each sample, the fragment shader can also modify its triangle coverage status, as well as overwriting the per-sample depth value used in the depth test and stored to the depth buffer.

Following the fragment shading and depth testing is a series of per-sample operations, which include blending, dithering, and logical operations, which are described in the OpenGL specification [107].

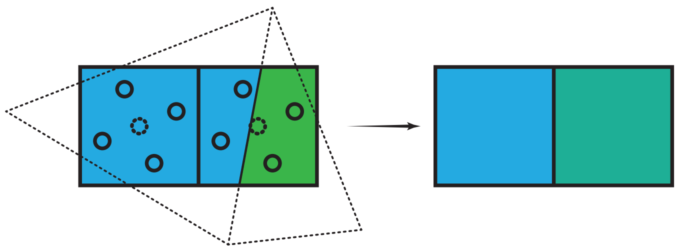


Figure 9: Two adjacent pixels with four visibility samples each (solid circles). The left pixel is completely covered by a blue triangle, and is thus shaded only once *when MSAA is enabled. The right pixel is pierced by an edge and is partially covered by another triangle, and is thus shaded twice, once for each surface. Here, the color is computed at the pixel center (dotted circle) for both triangles. On the right side is the final pixel color, after the MSAA resolve.*

2.2.6 MSAA Resolve

If the sample rate exceeds one sample per pixel, the color of the pixel must be determined from the local sample colors. To achieve this, current GPUs use a simple box filter spanning the extents of the pixel, which is equivalent to averaging the sample colors within the pixel, as shown in Figure 9.

2.2.7 Configuring the Pipeline

There is a large number of settings that can be used to configure the functionality of the pipeline, such as controlling the depth unit, changing how sample colors are blended together, and specifying the output render target texture, to name a few. Collectively, these settings constitute the current *render state*. This configurability combined with the programmability of the shader stages offer a wide variety of ways to utilize the pipeline for different rendering techniques. Some algorithms even require multiple passes through the pipeline, for example to generate data from different viewing directions. In Paper **V**, we create a depth buffer from the light's point of view (known as a shadow map). The shadow map is processed using a series of compute shaders and the result is then used to create filtered shadows when rendering from the camera. In Paper **VI**, we run through the entire pipeline to generate shaded information, which is used in a subsequent pass, lowering the total computational cost of defocus and motion blur rendering.

3 The Depth Unit

The purpose of the depth unit is to determine whether a newly rasterized and shaded sample is visible or occluded from the camera view. This is accomplished by using a depth test, which compares the distance of the new sample to the sample currently in the depth and color buffers. Maintaining one depth value per sample was once deemed too brute force of a method due to being "ridiculously expensive" [112]. Today it is an integral part of high-performance rendering on the GPU. Papers **I**, **II**, **III**, **IV**, and **V** all revolve around the depth unit, and this section is therefore dedicated to describing it in detail.

3.1 Computing and Storing Depth

There are multiple viable choices for how to compute the depth value, d, destined for the depth buffer [69]. As long as d is proportional to the distance from the camera, it may seem unimportant which of the alternatives is used. The world space distance to the camera would be an obvious choice, but as Equation 5 below reveals, it is not linear in image space, (x,y). Linearity is a desirable property because it is simpler to evaluate and easier to compress. In addition, the distance from the camera is unbounded since there is no limit on how close or far away an object might be. In accordance with the OpenGL and DirectX specifications, the depth buffer should contain values in the $[0,1]$-range and hardware fixed-function depth interpolation assumes a linear depth function in image space. In the following, a depth function which fulfills these criteria is derived.

Model space to clip space The first step is to transform the triangle vertices to the camera coordinate frame, which is accomplished in the vertex shader and fixed-function hardware prior to the rasterizer in the rendering pipeline. Recall that each triangle in a mesh consists of three vertices, each with its own position $\mathbf{q}$. These positions are given in the *model space* coordinate frame, which is common to the entire object. The camera is a separate entity with its own *view space* coordinate frame. In this space, the camera is located in the origin, with the z-axis aligned with the viewing direction. In order to determine exactly what the camera sees, the mesh vertices must be transformed from model space to the camera's view space. Transforming a vertex position amounts to a matrix-vector multiplication of the position with a 4×4 affine matrix, $\mathbf{M}$. Note that the matrix may also include additional transformations and deformations, such as rigid body transforms and skinning, for example, which may be unique to each vertex.

For a point to appear in the image, the light must travel unoccluded through the image plane towards the aperture point, as seen in Figure 6. Assume that the image plane is placed at a distance $z = 1$ and is parallel to the xy-plane in view space. A point $\mathbf{r}_{view} = (r_x, r_y, r_z)$ in view space can be projected onto the two-dimensional image plane through perspective division, $\mathbf{r}_{image} = (\frac{r_x}{r_z}, \frac{r_y}{r_z})$. While the image plane extends infinitely, the output image is a delimited region of that plane. The size of the image depends on the field of view of the camera. The projection matrix, $\mathbf{P}$, describes the transform from view space to *clip space*, which represents the image coordinates, *prior* to perspective division. The combined transform, from model space to clip space, can be described as

$$\mathbf{p} = \mathbf{PMq}, \tag{2}$$

where $\mathbf{q} = (q_x, q_y, q_z, 1)$. The perspective projection is achieved using the w-coordinate of the clip space coordinates, which is produced by the last row of $\mathbf{P}_4 = (0,0,1,0)$, such that $p_w = (\mathbf{Mq})_z$. The vertex shader program outputs clip space coordinates, $\mathbf{p}$, and the perspective divide occurs implicitly when the GPU sets up interpolation functions for the vertex attributes.

Barycentric coordinates Next, we explore how vertex attributes are interpolated across the triangle. Any point, $\mathbf{r}$, on the triangle surface plane can be written as a weighted linear combination of its three vertices, $\mathbf{p}_i$, such that

$$\mathbf{r} = \sum_{i=1}^{3} B_i \mathbf{p}_i, \text{ where } \sum_{i=1}^{3} B_i = 1. \tag{3}$$

Here, B_i are the scalar weights, or *barycentric coordinates* of the point $\mathbf{r}$. Furthermore, it must hold that all three $0 \leq B_i < 1$ for the point to lie inside the triangle. Deriving the barycentric weights can be explained in the context of a *signed volume intersection test* [64, 96]. The barycentric weight B_i for the surface point, $\mathbf{r}$, can be obtained by finding the signed volume V_i of the tetrahedron spanned by the origin (i.e., the camera point), $\mathbf{o}$, the triangle edge $\mathbf{p}_j\mathbf{p}_k$, and $\mathbf{r}$, relative to the combined volume, $V = V_0 + V_1 + V_2$. The barycentric coordinate is thus expressed as $B_i = \frac{V_i}{V}$.
Shifting the position from $\mathbf{r}$, away from the triangle surface by some distance t along the direction $\mathbf{d} = \mathbf{r} - \mathbf{o}$ does not change the *ratio* of the signed volumes of the tetrahedra. The image plane coordinate $\mathbf{x} = \mathbf{r} + t\mathbf{d}$ can thus be used to find B_i. The signed volume of a tetrahedron can be obtained by computing the determinant $V_i = \frac{1}{6}\det(\mathbf{p}_j\mathbf{p}_k\mathbf{x})$. For simplicity, we can replace V_i with $e_i = 6V_i$ since the scale factor is canceled out anyway when computing the ratios of the volumes. Here, e_i corresponds to edge functions expressed in clip space, rather than image space as described in Section 2.2. With these observations, the barycentric weights for a particular sample location in the image, $\mathbf{x}$, as can be computed from the three clip space vertices as

$$\begin{aligned} e_i &= (\mathbf{p}_j \times \mathbf{p}_k) \cdot \mathbf{x} = \mathbf{n}_i \cdot \mathbf{x}, \\ B_i &= \frac{e_i}{e_0 + e_1 + e_2}. \end{aligned} \tag{4}$$

Linear depth The barycentric coordinates can be used to interpolate any attribute linearly across the triangle surface in clip space. That is, from Equation 4, we see that the interpolation of an arbitrary attribute, A, can be expressed as a division between two plane equations, that is,

$$A = \sum_{i=1}^{3} B_i A_i = \frac{\left(\sum_{i=1}^{3} \mathbf{n}_i A_i\right) \cdot \mathbf{x}}{\left(\sum_{i=1}^{3} \mathbf{n}_i\right) \cdot \mathbf{x}} = \frac{a_x x + a_y y + a_0}{b_x x + b_y y + b_0}. \tag{5}$$

It is apparent that A does not generally vary linearly with regard to the image coordinate $\mathbf{x}$. In other words, attributes linearly varying over the triangle in clip space do not exhibit the same property in image space. When interpolating p_w we note the numerator of Equation 5 does *not* actually depend on x or y, and thus gives a constant value for the triangle,

$$\left(\sum_{i=1}^{3} \mathbf{n}_i p_{i_w}\right) \cdot \mathbf{x} = \det(\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}) = D. \tag{6}$$

One over the clip space coordinate w is thus linear in image space since

$$\frac{1}{w} = \frac{b_x x + b_y y + b_0}{D} = c_x x + c_y y + c_0, \tag{7}$$

which incidentally is the desired behavior for our depth function.

Bounded depth Using the depth $d = \frac{1}{w}$ does fulfill the linearity criterion but it is unbounded since it approaches ∞ when w moves close to 0. In order to get a bounded depth value, some limits on the depth range, $[z_{near}, z_{far}]$, must be imposed. These bounds correspond to how near and how far away from the camera objects may lie to be considered part of the view frustum. In modern graphics APIs [17, 107], the depth buffer expects values in the $[0,1]$-range and it allows the depth unit to be oblivious to the chosen z_{near} and z_{far} values. The depth, d, should thus assume $d = 0$ when the clip space coordinate $w_{clip} = z_{near}$, and $d = 1$ when $w_{clip} = z_{far}$, and values outside of this region are not part of the view frustum. Some expression which scales and biases $\frac{1}{w}$ is therefore sought, such that $d = \alpha + \frac{\beta}{w}$ fulfills the bounds criteria. Solving for α and β yields $\alpha = \frac{z_{far}}{z_{far} - z_{near}}$ and $\beta = -\frac{z_{near} z_{far}}{z_{far} - z_{near}}$. This scale and bias is included in the projection matrix, $\mathbf{P}$, and constitutes the (previously omitted) z-component of the clip space coordinates. The last two rows of the projection matrix are thus $\mathbf{P}_3 = (0, 0, \alpha, \beta)$ and $\mathbf{P}_4 = (0, 0, 1, 0)$, which multiplied by a camera coordinate yields $z_{clip} = \alpha z_{view} + \beta$ and $w_{clip} = z_{view}$. Finally we have arrived at a linear, bounded depth value,

$$d = \frac{z_{clip}}{w_{clip}} = \frac{\alpha w_{clip} + \beta}{w_{clip}}. \tag{8}$$

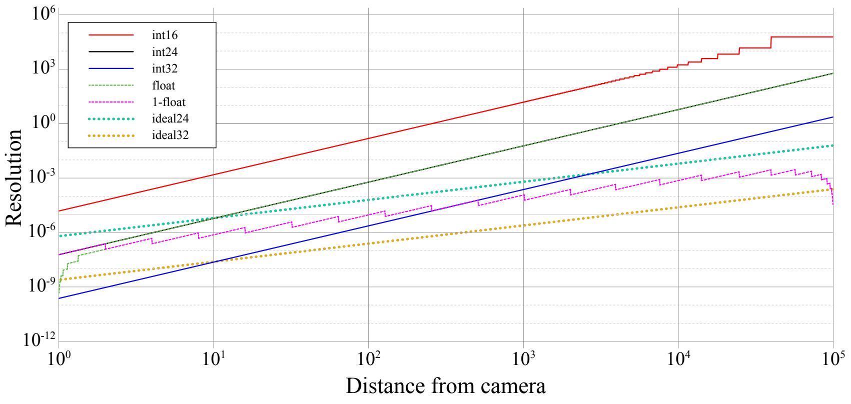


Figure 10: Depth format precision loglog-plot. The near plane is at a distance of 1 *units and the far plane at* 10^5 *units. The jagged appearance of the floating point curves occur when the exponent changes. The* ideal *variants use a logarithmic depth function.*

Depth formats The required resolution of the depth values depends heavily on the scene geometry. If two objects lie too close together for their depths to be resolved, *z-fighting* artifacts will ensue. Quantization errors can cause geometry that is, in fact, occluded to shine through the occluding surface, and even worse, result in flickering when the camera or the geometry moves.

Modern APIs support normalized integer and floating point depth buffer formats. Normalized integers maps the $[0,1]$ range uniformly with 16, 24, or 32 bit precision (though the latter is currently not available in DirectX and is not required by OpenGL). Floating point depth buffers use 32 bits per value, although, since the depth is clipped or clamped to $[0,1]$, positive exponents and the sign bit are unused. Depending on which format is chosen, the resolution in depth varies significantly at different distances from the camera. Figure 10 shows a comparison of the different depth formats and the resolution at a wide range of distances. Floating point buffers "spend" most of the bit combinations very close to the near plane, since the format has more precision closer to 0 than to 1. When the depth exceeds $d > 0.5$, the resolution is no better than 24-bit integers, since the entire range $[0.5, 1)$ is covered by a single exponent value, leaving only the 23 mantissa bits (and the leading 1). To alleviate this, z_{near} and z_{far} can be reversed [69], effectively mapping $d = 0$ at the far plane and $d = 1$ at the near plane. This greatly increases the precision for $d < 0.5$, which is the majority of the $[z_{near}, z_{far}]$ range.

Ideally, the depth resolution at some distance, w, should be proportional to the projected object size at that distance. Choosing a function proportional to the logarithm of w as the depth function will have the desired behavior [121]. Examples using 24 and 32 bits are shown in Figure 10. However, this type of depth function does not vary linearly over the triangle surface in image space, and is thus not natively supported by current graphics APIs.

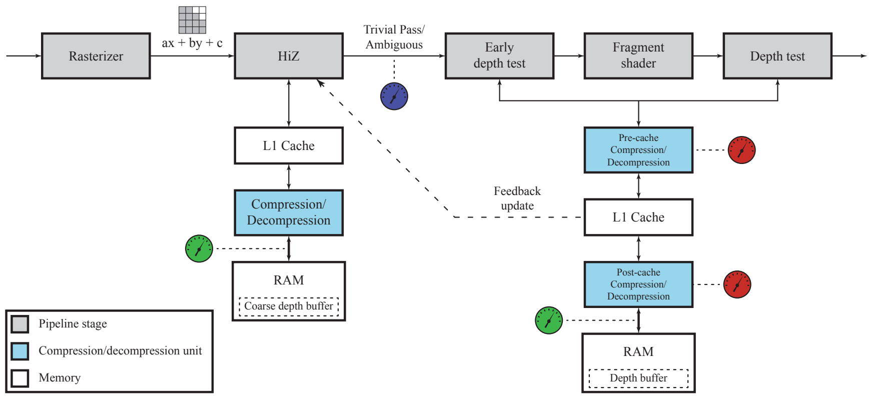


*Figure 11: Depth system hardware architecture overview. The gray boxes show the hardware pipeline stages, white boxes show the caches and RAM, and light blue boxes are the compression/decompression stages that we have introduced or improved upon. In Paper **IV** we improved the HiZ unit and added a compression stage for its coarse depth buffer. Papers **I**, **II**, and **III** all target improvements in the compression unit for the depth buffer. The green gauges show where we measured external bandwidth to RAM in Papers **I**, **II**, **III**, and **IV**. In Paper **I**, we measured how frequently compression and decompression occur for different configurations, as signified by the red gauges. We measured the number of tiles culled by the HiZ unit in Paper **IV** at the location of the blue gauge.*

3.2 Hardware Architecture

The rendering pipeline, illustrated in Figure 7, describes the graphics hardware on a functional level, and the function calls exposed by the graphics API reflect this model. However, the underlying hardware architecture implementing it can be quite different. In the hardware design, various optimization efforts can be made, as long as these are non-intrusive to the specified behavior. Throughout our research we have modeled the depth unit architecture as shown in Figure 11, which roughly follows the model described by Hasselgren and Akenine-Möller [51].

The input to the depth system is provided by the rasterizer. For each triangle, the rasterizer performs inside tests for all samples within a tile at a time. For each tile it outputs a coverage mask and the depth plane of the triangle. Following the rasterizer, and preceding depth testing and depth buffer updates, is a hierarchical z unit, or *HiZ* unit. Its purpose is to relieve the depth testing unit of some of the unnecessary work by swiftly culling tiles, when possible, using a coarse, conservative depth test. To that end, the HiZ unit maintains its own coarse depth buffer, which is conservative with regards to the depth buffer. According to the API specifications, the depth test should be performed after the fragment shader. However, depending on the current render state, it is often possible to perform the test before the fragment shader, using an *early* depth test, which avoids some unnecessary shading work.

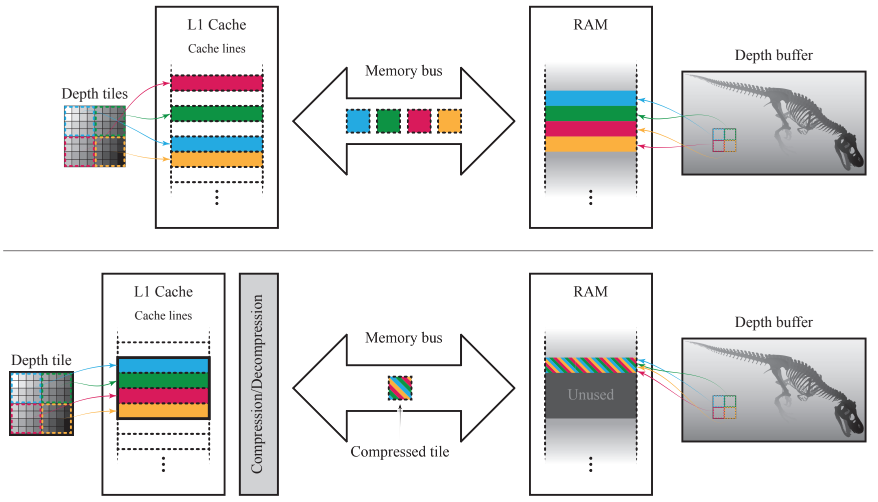


Figure 12: Top: *tiles of depth values reside in the depth cache. Each tile occupies one cache line and are individually transmitted over the memory bus.* Bottom: *a compression/decompression unit is introduced between the memory bus and the cache. A tile now constitutes a set of cache lines, which are compressed together when going in and out of the cache to RAM via the bus. In this example, the compressed representation only occupies $\frac{1}{4}$ of its original size (i.e., 4:1 compression ratio). Since the depth buffer must support uncompressed data, the memory consumption of the buffer remains the same, leaving some of the memory unused when the data is compressed in RAM. (T.rex 2 model courtesy of Joel Anderson)*

3.2.1 Caching and Compression

The depth buffer is randomly accessed and updated throughout rendering, which entails significant memory bandwidth usage and potentially high computational costs. Memory bandwidth consumption and latency is greatly reduced by introducing a system of caches to back the buffers. For simplicity, we have only modeled memory systems with a single L1 cache [98] in our research, since more caches have diminishing returns and are more complicated to implement and tune. Furthermore, lower-level caches are often shared by multiple units, making their effectiveness hard to predict without modeling the entire system. Bandwidth usage can be further reduced if the data can be *compressed* prior to being transmitted to RAM via the memory bus. By introducing a compression unit, tiles can be compressed on the fly when leaving the cache and decompressed when read back in to the cache, as illustrated in Figure 12. It is important to note that depth buffer compression should be lossless. Errors due to lossy compression could be visible since correct ordering of objects would not be guaranteed.

The memory bus width constrains the achievable bandwidth gains when using compression. If the width is 64 bytes, for example, only data packets of that particular size can be transmitted. By operating on larger tiles, say 256 bytes worth of depth data, the compression unit can try to compress it down to $64 \times N$ bytes, where N is an integer, in this case between 1 – 3. How well the data compresses is given by the *compression ratio*, which is the ratio of the uncompressed data size and compressed data size (although it sometimes given as the percentage of the compressed size compared to the uncompressed). The compression ratio can be expressed as a local measure to each tile, or refer to the total compression over all processed tiles. Note that it may be better to use a compression scheme that is successful for *most* of the tiles but has a poor local compression ratio, than a scheme that compresses only a *few* tiles with a high compression ratio.

Compression schemes There are a number of proposed compression schemes targeting the depth buffer. Most of these exploit the linear depth plane, which is domain-specific knowledge, and store the plane equation, possibly in some reduced precision format. There are essentially two ways that the plane equation can be obtained – either by retrieving the depth interpolation used by the rasterizer, or an estimated plane equation can be constructed from the set of depth samples. As described later in Section 3.2.2, depending on the architecture, the former method might not be possible at the time of compression, and the latter must be used. Since compression must be lossless and a reconstructed predictor plane may be inexact, per-sample residuals are needed to correct the prediction error. If multiple triangles with different surface planes overlap the tile, using a single plane to approximate them may give large residuals. In this case, more than one predictor can be used, with each sample being assigned to the best matching one. The total number of bits required to store a tile using this format amounts to the cost of the prediction planes, the per-sample selection mask, and the residuals. This family of compression schemes are commonly called *anchor encoding* [51]. If the rasterizer can provide the plane equation to the compression unit, the depth values will be exact and no residuals are needed at all. This type of scheme is called *plane encoding*.

For highly varying depth data, it may be difficult to fit good prediction planes to the data. In these cases, replacing the linear predictor functions with constants will reduce the predictor storage overhead, and may well produce residuals of the same magnitude as for a poorly fitted plane. *Depth offset compression* stores the minimum and maximum depth values in a tile as the predictors, a selection mask that assigns each sample to the predictors, and the residuals to correct the prediction error. From our experience, using this type of compression leads to somewhat worse compression ratios than using anchor encoding, but is very useful when depth varies greatly.

When considering bandwidth usage, it is important to note that the depth buffer format can have a great impact. The depth resolution offered at a particular viewing distance may be orders of magnitude higher than what is actually required. In this case, if the predictor function is even slightly off, the estimated values will require many residual bits to correct, leading to low compression ratios. If bandwidth is an issue for a particular workload, it might be better to select a lower-resolution format.

Over-fetching As our research demonstrates, compression gives significant bandwidth savings in most situations. Counter-intuitively, however, there are cases where the bandwidth can be negatively impacted by the usage of compression/decompression units. When a compressed tile is fetched from RAM, decompressed and put in the cache, the data will expand to occupy several cache lines, as can be seen in Figure 12. On the contrary, the optimal tile size for fetching uncompressed depths is equal to the amount of data that fits in to one single cache line, i.e., a compressed tile covers a larger screen space region than an uncompressed one. Even though the storage for each individual sample is smaller, more samples are read and put in the cache unnecessarily, which in this context means that they do not partake in any depth test/write before being evicted. This is especially problematic when poor compression ratios are combined with sparse depth buffer accesses. The hand scene from the results section in Paper **III** is an example of such a scenario, where over-fetching is so detrimental that it counteracts and even exceeds the memory bandwidth savings. For the remainder of the scenes, however, the gains from depth compression greatly outweighs the over-fetching effect.

Neither prior art [51] nor Paper **II** identify or address this problem as they both use a fixed tile size which require several cache lines even for incompressible data. Papers **I** and **III**, however, both explicitly handle incompressible tiles using a separate, smaller tile size that fits into a single cache line.

3.2.2 Pre- and Post-Cache Compression

In Paper **I**, we compare how the number of computations and the memory bandwidth are affected by employing different depth system configurations. By keeping data compressed in the cache, we achieve better utilization of the available cache space and utilize less bandwidth as a result. The downside of using this approach, however, is that the data needs to be decompressed each time it is read and recompressed when writing back to the cache. The number of transactions between the depth testing unit and the cache is greater than between the cache and the RAM. Maintaining compressed data in the cache thus requires more computations than the alternative.

We refer to the compression/decompression units for the different cache configurations as *pre-cache* or *post-cache* codecs because of their placement relative to the depth testing unit and the cache. Naturally, the compression algorithms used affect both the memory bandwidth used and the computational burden, which is why we include numbers for a variety of compression schemes in the paper. In addition, we propose a hybrid solution between the pre- and post-cache codecs which uses a simple scheme to introduce new data into an already compressed representation when the tile is in the cache, and defer the full quality compression for the less frequent cache evictions. Our hope is that the different configurations explored can provide some insight and serve as a guide in balancing the number of computations and the memory bandwidth when designing the compression/decompression and memory systems for depth units.

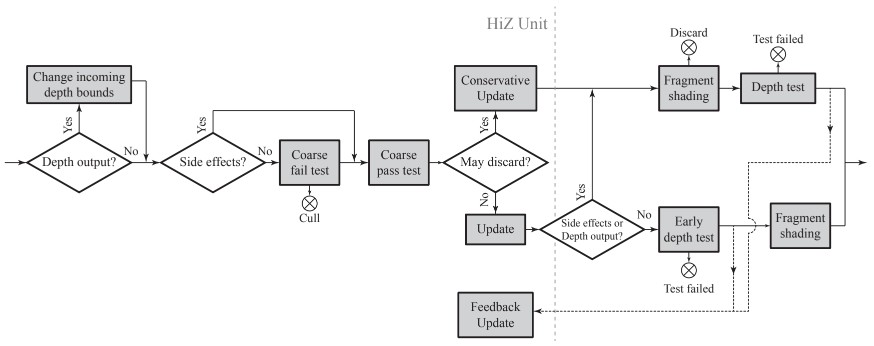


Figure 13: Flow chart over how HiZ, the depth unit and the fragment shader operates under different render states. Note that in this model, depth tests and updates are coupled, which is why the full depth test unit is run after the fragment shader when side effects are active.

3.2.3 HiZ

The HiZ unit is an optimization used to quickly determine whether all samples within a tile will pass or fail the depth test, or if a per-sample test is required. By keeping depth bounds, $[z_{min}, z_{max}]$, for each tile in the depth buffer, a quick interval overlap test can be done with the incoming triangle. Per-sample testing is only necessary if there is an overlap.

For tiles with depth discontinuities, for example around the silhouettes of an object, the depth bounds can become large, which reduces the culling potential. Furthermore, determining the exact z_{max} value requires a pass through all samples in the tile. Since the tile resides in the depth buffer and the information is needed in HiZ, a feedback loop is required. The implementation of such a mechanism in hardware is laden with problems concerning delay and memory design.

In Paper **IV**, we propose a novel HiZ representation which captures high frequency depth discontinuities, without requiring a feedback loop to maintain it. Our format has multiple layers of z_{max} values and a per-sample selection mask. Furthermore, it is simple to test against and to keep up to date. We fuse occluder information by merging incoming surfaces with the layers based on a heuristic.

3.2.4 Render State Dependence

The active render state governs whether an early depth test may be used. In addition, HiZ culling decisions and updates to the coarse depth buffer must be made *conservatively* based on the state. In Papers **I**, **II**, and **III** we only consider the primary render target of the test scenes, and early depth testing is always enabled. In Paper **IV**, however, we run full frames from current games, complete with all render targets, which means that we encounter a variety of different render states.

The order in which culling, depth testing, and fragment shading is executed, and which update strategies are used is illustrated in the flow graph shown in Figure 13. The path taken in the graph is based on the three boolean properties listed below, which are extracted from the render state. These mainly affect the HiZ unit, since if *any* of them is true, early depth testing must be disabled.

- **Depth output** - The fragment shader program can choose to provide the depth values for the samples, rather than using fixed-function interpolation. The HiZ unit has no way of knowing what the depth will be for a given sample and must assume that it could be any value.
- **Side effects** - The fragment shader may write data to auxiliary buffers other than the current color and depth buffers. It can thus have side effects that require it to run, even though the sample may fail the depth test. The coarse fail test must therefore be disabled in these scenarios, and the depth test must be performed after the fragment shader.
- **May discard** - If the fragment shader has a discard operation, the triangle/tile coverage seen by the HiZ unit may be altered. The HiZ unit must consider that each sample can have either the interpolated depth *or* if coverage is altered, that it retains its previous value.

3.2.5 Power and Latency

By using a software implementation of the system described, we are able to measure simulated depth buffer bandwidth and gather various statistics. These measurement are the basis for our results and our conclusions in Papers **I**, **II**, **III**, and **IV**. Power usage and latency are not considered, save for **IV** where some pitfalls relating to delays in the system are discussed. Power and latency are much harder to predict than bandwidth, since they depend heavily on other parts of the graphics chip. In real depth unit hardware, it is likely that much of the design effort is spent on latency hiding and power minimization by fine tuning the various parts of the chip. However, saving memory bandwidth by using compression and efficient caching is likely to lead to both latency reduction and decreased power usage.

4 Advanced Camera Models

Using a real-world camera, some of the light bouncing around in a scene can be captured and focused onto its sensors. Due to the various inherent limitations of the camera, the captured image may contain artifacts such as lens flares, chromatic aberration, and motion and defocus blur, for example. Depending on the application, rendering images which include these imperfections introduced by the camera may be desirable. Examples of motion and defocus blur, which are the effects which we try to recreate in our research, can be seen in Figure 14.

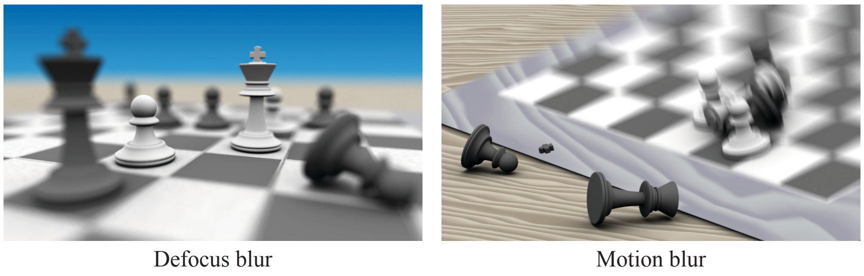


Figure 14: Chess pieces rendered with defocus blur (left) and motion blur (right). (Battlefield model courtesy of Rasmus Barringer)

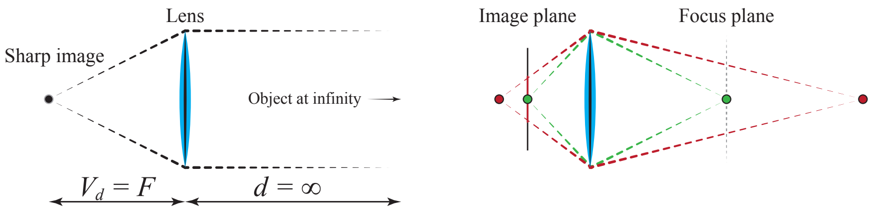


Figure 15: Left: *the light from a point on an object infinitely far away converges to a sharp image at a distance F from the lens.* Right: *the focus plane distance is determined by the location of the image plane. Here, the green point is located on the focus plane and will thus produce a sharp image. The red point if farther away, and is thus not in focused on the image plane, creating a blurred region.*

4.1 Defocus blur

The pinhole camera model offers a seemingly neat and simple way to construct a camera. Alas, its real-world applications are limited. According to the model, the aperture should be infinitely small in order to maintain the one-to-one mapping of light ray direction to image location. Obviously, the amount of light that would find its way through such aperture would approach zero. However, diffraction cannot be disregarded when a small aperture size is used, limiting the resolution that can be achieved with pinhole cameras. With a larger aperture size, more light is allowed through the opening, but as a result the image will also become blurrier.

To improve on the camera, some mechanism capable of gathering more light from the scene, while maintaining a sharp image is needed. To achieve this goal, a series of optical lenses are introduced which makes it possible to focus the incoming light. The most obvious limitation of this approach is that the lens system has limited depth range in which a sharp image can be produced. The lens systems are often intricate and contain many lens elements, which makes it non-trivial

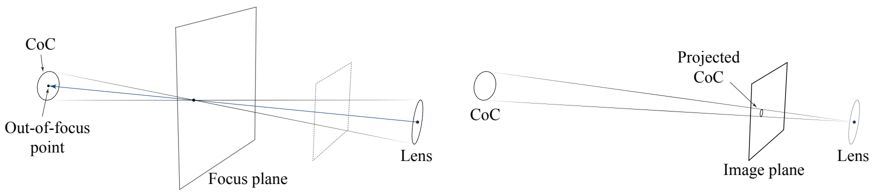


Figure 16: The circle of confusion (CoC) size visualized at a point located some distance behind the focus plane. Left: *the CoC size grows linearly away from the focal plane. (which corresponds to the numerator in Equation 10).* Right: *the CoC projected onto the image plane (the denominator in Equation 10).*

to model their exact behavior [65]. To alleviate this problem, the much simpler *thin lens model* [101] may be a viable approximation, especially in the realm of high performance real-time graphics. As shown in Figure 15, given an object at distance, d, from a lens, the thin lens equation can be used to calculate the related distance, V_d, where the object produces a (sharp) image as

$$\frac{1}{F} = \frac{1}{d} + \frac{1}{V_d}. \tag{9}$$

Here, F is the *focal length* of the lens. For an object infinitely far away, the sharp image will be produced at the distance F behind the lens, i.e., $V_d = F$ when $d = \infty$. The size of the lens is given by the focal length together with the *aperture number*, n, as $\frac{F}{n}$. The image plane in the camera (i.e., where the sensors are) is at a fixed distance, but where the sharp image is produced for a particular object depends on its distance from the lens. Hence, the image plane and the sharp image of the object will only coincide at a particular depth. When these distances differ, the result will be a blurred region proportional to the difference in distance and the lens aperture size. Assuming a circular lens, Potmesil and Chakravarty [101] derive the diameter, C, of the blurred circular region known as the *circle of confusion* (or CoC). This is illustrated in Figure 16. By grouping all of the lens parameters, the equation can be written on the following form

$$\begin{aligned} C &= \left|\frac{c_0 + c_1 d}{d}\right|, \\ c_0 &= -Pc_1, \\ c_1 &= \frac{F^2}{n(F-P)}. \end{aligned} \tag{10}$$

Here, P is the distance from the lens to the *focus plane*. For a point in focus, $d = P = -\frac{c_0}{c_1}$, which yields the expected result $C = 0$. Depending on the application, it may be easier to think of defocus blur in these terms, rather than focal lengths and aperture numbers. The focal plane distance P and the (clip space) lens aperture size c_0 can be directly and intuitively controlled.

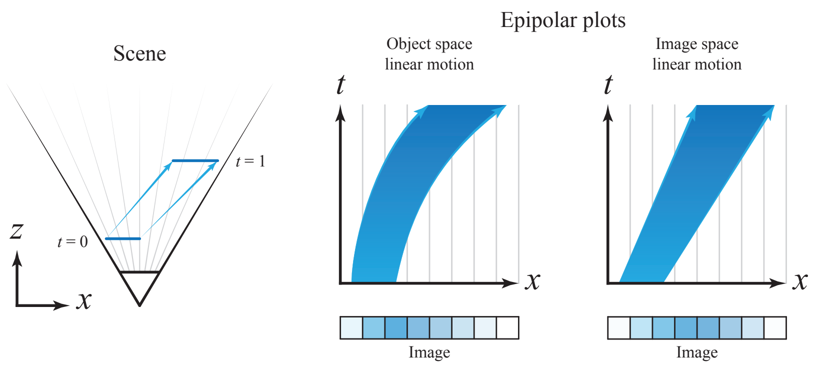


Figure 17: Left: *a scene with a moving line segment with the camera at the bottom.* Middle: *the trajectory of the line segment in the image, x, over time, t, is curved. Most of the color of the object ends up in the left part of the image (when the object is closer to the camera).* Right: *approximating the motion linearly in image space produces an incorrect image.*

4.2 Motion blur

The camera model described so far can produce convincing still images, complete with defocus blur, captured at a single instant in time. However, when watching a movie, playing a video game, or when photographing an object in motion, it becomes evident that *motion blur* also plays an important role. Real cameras do not expose the sensors for an infinitesimal time span, but rather keep the shutter open for a short period. Objects in the motif and/or the camera itself may move around during this time, leaving motion trails in the image. Although there are different shutter designs, for most real-time applications it is sufficient to assume that the shutter opens and closes instantaneously and that the entire image is exposed equally during the open shutter time.

For the open shutter duration, objects can move along arbitrary paths, leaving intricate motion blurred trails. Although some research approximates the motion trails using higher order curves, such as Bézier curves [46], most research on the topic only consider linear vertex motion in clip space or image space. Complicated paths could be approximated using piece-wise linear approximations. With linear vertex motion, each vertex has a position at the beginning and the end of the exposure interval. This first-order approximation often works well in practice, and as discussed in the next section, it greatly simplifies the computational burden.

Linear clip space vertex motion is applied prior to perspective projection. The vertex paths in image space will also be linear, but the velocity along the path will vary non-linearly, due to the projection. Image space linear motion is thus a coarser approximation, since it ignores this shift in velocity. A flat-land example illustrating the difference using two approaches is shown in Figure 17. A comprehensive overview on the topic of motion blur in computer graphics is given by Navarro et al. [93].

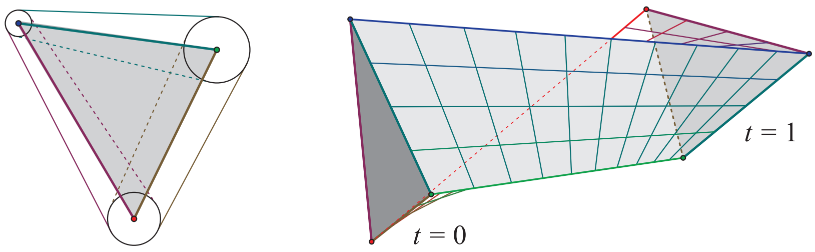


Figure 18: Left: *an out-of-focus triangle. Depending on the lens parameters, the triangle edges will shift within the inner (dotted) and outer (solid) bounds.* Right: *a triangle which is translated and rotated over time. While the vertex motions are linear the edge equations form bilinear patches.*

4.3 Stochastic Rasterization of Motion and Defocus Blur

In conventional rasterization specified by current graphics APIs, only a small sample pattern is repeated throughout the entire image. *Stochastic rasterization*, on the other hand, uses randomly placed samples within each pixel. Furthermore, it is easy to introduce additional dimensions to be sampled, apart from the image coordinates, (x,y). For motion and defocus blur, the samples are also associated with time, t, and camera lens coordinates, (u,v). Each sample can then be used to probe the scene from one particular location on the lens and in the image, at one particular instant in time. This way, the effects of motion and defocus blur can be captured, since prolonged exposure times and the spread of the lens are also taken in to account. In order for the stochastic rasterizer to be able to sample the added dimensions, a new set of primitives describing the triangle behavior in these dimensions is required.

Motion and defocus blurred triangle primitives By including the (u,v,t)-dependencies owing to the addition of motion and defocus blur, expressing the clip space vertex position, $\mathbf{p}$, becomes somewhat more involved [89]:

$$\mathbf{p}(x,y,u,v,t) = \underbrace{\mathbf{p}_0}_{\textit{Static}\ (2\text{D})} + \underbrace{t(\mathbf{p}_1 - \mathbf{p}_0)}_{\textit{Motion}\ (3\text{D})} + \underbrace{\mathbf{u}'C_0}_{\textit{DOF}\ (4\text{D})} + \underbrace{\mathbf{u}'t(C_1 - C_0)}_{\textit{Combination}\ (5\text{D})} . \tag{11}$$

Here, subscript 0 and 1 denote the beginning and end of the time interval. $\mathbf{p}$ is the vertex position, C_i and is the circle of confusion radius, and the lens is parameterized as $\mathbf{u}' = (u, \xi v, 0)$. Here, ξ controls the aspect ratio of the lens. The number of terms to be included from Equation 11 depends on the desired camera effects. Motion and defocus blur each require one additional term to the static case, while all terms must be included for the combination of both effects.

First, we consider only the motion blur effect. The two vertex positions at times $t = 0$ and $t = 1$ can be obtained by having separate transforms $\mathbf{p}_0 = \mathbf{M}_0\mathbf{q}$ and

$\mathbf{p}_1 = \mathbf{M}_1\mathbf{q}$. With the approximation that the vertex moves linearly from $\mathbf{p}_0$ to $\mathbf{p}_1$ it is easy to see that $\mathbf{p}(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0)$.

Next, we consider the lens aperture parameters, (u, v), required for defocus blur. Assuming a standard projection matrix, where $\mathbf{P}_4 = (0, 0, 1, 0)$, the clip space vertex w-component depends on the view space depth such that $p_w = (\mathbf{Mq})_z$. From Equation 10, we see that the circle of confusion radius in clip space (i.e., prior to perspective division), C, depends on the distance $d = p_w$ such that $C = c_0 + c_1 d$. The lens dependency can be incorporated in a shear matrix, $\mathbf{D}$, operating on the clip space coordinates in the following manner [102]

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & c_1 u & c_0 u \\ 0 & 1 & c_1 \xi v & c_0 \xi v \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{12}$$

The clip space vertex position can thus be calculated as $\mathbf{p} = \mathbf{DPMq}$ for defocus blur. Examples of motion and defocus triangles can be seen in Figure 18.

The final term in Equation 11 describes the combination of the two effects. Here, the circle of confusion radius depends linearly on view space depth, p_w, which, in turn, depends linearly on time, t.

Stochastic sampling With the motion and defocus blurred triangle primitives established, the next step is to determine the color contribution to each pixel. Stochastic rasterization employs Monte Carlo integration to evaluate the pixel color integral [6, 26]. Similar to conventional rasterization, each sample is inside tested by evaluating edge functions for each triangle prior to shading. Inside testing is more computationally expensive than for static triangles, because of the more complex vertex positions, $\mathbf{p}$. There are a number of ways to accelerate the inside testing when designing a stochastic rasterizer by limiting the number of per-sample tests that needs to be performed [8, 66, 89, 90].

Another way to save computations is to try to use a lower sample rate, and instead focus on using a well distributed sample set. Instead of using purely random sample points, *quasi-Monte Carlo integration* can be used, which trades reduced noise for aliasing [97]. Yet another way to improve the quality while maintaining a low sample rate is to try to *reconstruct* a better color estimate using the local neighborhood of samples. This approach is explained further in Section 4.5.

Analytical intersections of the higher-order primitives could be used in place of Monte Carlo sampling. However, while such a solution would be impractically complicated, a prospect that has been explored in the literature to some degree is solving at least *some* of the visibility problem analytically. The intersection along the t-dimension can be analytically computed for a fixed (x, y)-position [45]. Using line samples in the image [60], it has still proven useful to find the analytical intersection in the (x, y, t)-domain [44], although keeping track of the intersections is cumbersome. A similar approach has been proposed which uses line samples over the lens [114].

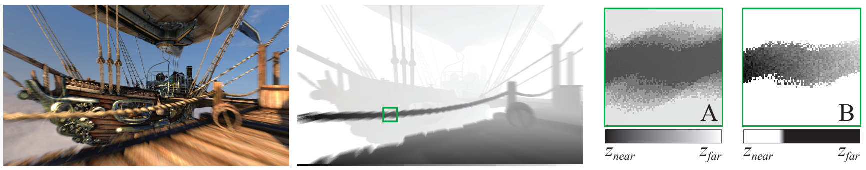


Figure 19: A static scene with a moving camera. The two images to the left show the final rendered image and its corresponding depth buffer, using four samples per pixel. Here, each pixel is displayed using the average depth of its samples. A small crop out of the buffer is shown in inset A. *Note the noisy banding resulting from sample coverage in the transition from the rope to the background. Inset* B *shows the same region, only the gray scale is remapped to only show the pixels where all samples are covered by the rope. Here it becomes apparent that the depth values are irregular due to the samples intersecting the moving rope at different time instances. (Unigine Heaven benchmark courtesy of Unigine Corp.)*

4.4 Stochastic Depth Buffer Compression

As previously mentioned, most of the depth buffer compression algorithms proposed for static scenes exploit the linearity of triangle faces in image space. With the additional terms introduced in Equation 11, it is easy to see that the higher order triangle faces do not abide by this rule. Looking at Figure 19, note that the depth buffer values become highly incoherent and noisy when viewed in the image domain, (x, y). As Paper **III** describes, depth is a function of all parameters, (x, y, u, v, t), for motion and defocus blur. In order to achieve good compression ratios all of these dimensions must be considered.

In Paper **II**, we present an extension to the anchor encoding compression scheme, described in Section 3.2, for handling stochastically sampled motion blur. We use a set of novel predictor functions which take the time dimension into account. Using this scheme, tiles containing fairly uniform motion can be efficiently compressed. Some tiles are overlapped by many triangles, or have highly varying motion vectors, leading to very high frequency depth changes, which is difficult to capture with our predictor functions. Such tiles can be fairly well compressed using depth offset compression, however. When combined, these two compression schemes significantly lower the bandwidth usage for all test scenes compared to previous work.

Like anchor encoding and depth offset compression, the compression unit used in Paper **II** tries to compress the already interpolated depth sample set. In Paper **III**, however, we use an approach which is more akin to plane encoding for static triangles. Here, we assume that the depth function is provided by the rasterizer unit. The depth is a function not only of the image coordinates, but also of the time and lens parameters. Instead of interpolating and storing the raw depth values for a tile, we try to keep a more compact representation by storing the depth functions and a per-sample selection bitmask for as long as possible. This is made possible by using a pre-cache type architecture, as described in Section 3.2 and in Paper **I**. In absence of motion and defocus blur, the higher order depth functions fall back to the regular, static plane equation described in Section 3.1. We are able to significantly lower the memory bandwidth usage over a range of test scenes. We improve further upon the results of Paper **II** for motion blur, as well as handling scenes with defocus blur.

4.5 User-space Stochastic Rasterization

At the time of writing there no time- and/or lens-dependent triangle primitives in the graphics APIs that can be pushed through the conventional rendering pipeline. However, it is still possible to utilize the pipeline of current generation GPUs to implement stochastic rasterization and achieve the same results. The particulars of the motion/defocus primitives must be implemented in the shader stages of the rendering pipeline or using compute shaders. There are arguably many ways to implement such a system, but one common approach is to bound the triangle in screen space in the geometry shader and then to perform a custom inside-test in the fragment shader [82, 113].

While Papers **II**, **III**, and **IV** simulated a modified hardware pipeline with support for motion and defocus blurred triangles, user-space stochastic rasterization was used in Papers **V** and **VI**. The following section gives a high-level description of the implementation used for these two papers.

Vertex shader In the vertex shader, the vertex positions are transformed to clip space, both for the beginning and for the end of the frame interval. Additionally, the CoC radius for each vertex is computed for these two time instances. For time- and lens-independent shading, it is often sufficient to compute the surface color for a single instant in time [27, 102]. The attributes that are required for shading can be computed at the middle of the lens and time interval, for example.

Geometry shader The geometry shader is used to ensure that the fragment shader is executed for all the potentially covered samples, since it there that the sample coverage status will ultimately be determined. In order to do this, the moving and/or defocused triangle is bounded in image space, and the resulting region is outputted as a series of triangles. There are various ways to do this, as illustrated in Figure 20. Different alternatives for motion blur and defocus blur have been explored by Akenine-Möller et al. [6], Toth and Linder [113], and McGuire et al. [82]. The solutions range from creating a bounding box around the primitive, to computing a tight convex hull around it, which must then be tessellated into a number of triangles. Ultimately, the cost of performing additional inside tests in the fragment shader must be weighed against the computational overhead of the geometry shader. The overhead depends, to some degree, on the number of output vertices. The convex hull method proposed by McGuire et al. optimizes the

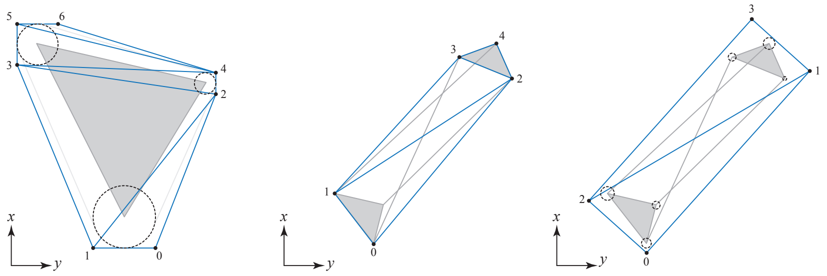


Figure 20: Different bounding alternatives. Left: *an out-of-focus triangle. A triangle strip can be used to tightly bound the triangle, including the CoC.* Middle: *a moving triangle. Constructing the convex hull of the triangle will give good bounds using relatively few vertices (six at the most, compared to nine for McGuire et al. [82]).* Right: *a triangle with simultaneous motion and defocus blur. An oriented bounding box can be used to bound the affected screen space region fairly well [6]. The average motion vector can be used or the largest axis can be found using principal component analysis, for example.*

geometry shader execution time, but the triangulation outputs a fan of triangles. Since the geometry shader can only output triangle strips, one of the vertices must be repeated multiple times to produce a triangle fan. As shown in Figure 20, a triangle strip can be used instead, which requires fewer output vertices, at the cost of some additional instructions.

Fragment shader The fragment shader program will be executed for each of the samples covered by the bounded region set up by the geometry shader. Current generation APIs do not allow for per-sample depth output if MSAA is enabled. In order to handle visibility correctly the shader must run on per-sample granularity. It is, nonetheless, faster to execute a single fragment shader per pixel, and loop through the samples within it, rather than to invoking N separate shader executions, where N is the sample rate per pixel.

In order to evaluate the edge equations, each sample must be associated with its own (u,v,t)-coordinate in addition to its image coordinates. This additional information can be precomputed and stored in a three channel three-dimensional texture, which translates a pixel position and a sample index to a (u,v,t)-tuple, and is read just prior to performing the inside test. The lookup coordinates to the texture are comprised of the pixel coordinates and the sample index. As for the image coordinates, either the sample locations provided by the APIs can be used or, alternatively, two additional channels in the 3D-texture can carry this information. Although the sample locations are programmable in modern GPUs, the size of the allowed patterns is still quite low and thus repeats frequently, which can cause undesirable aliasing.

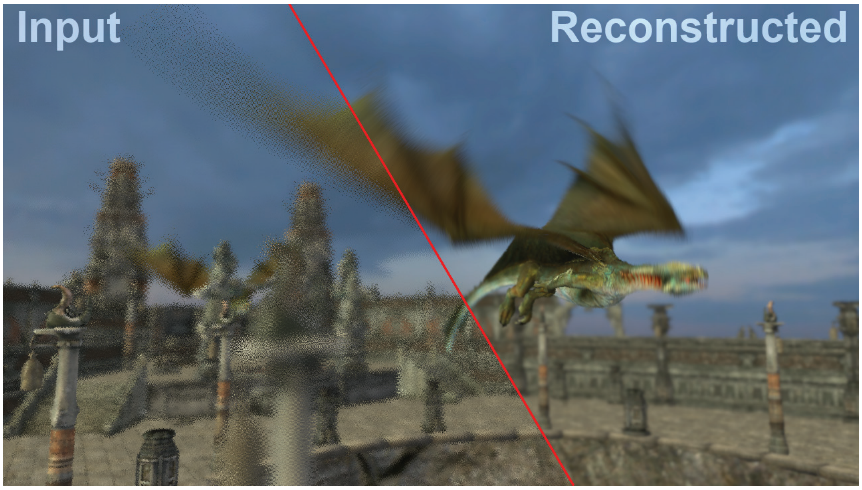
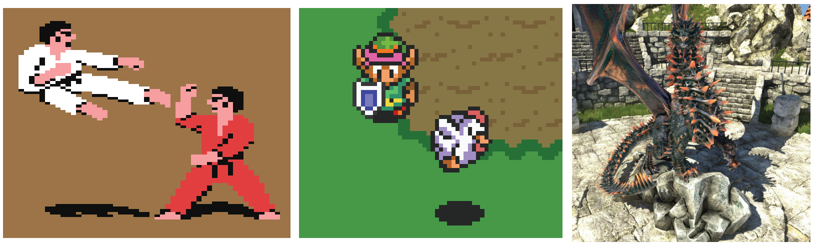


Figure 21: The quality of noisy stochastic renderings can be greatly improved using reconstruction [92].

For best performance, as few operations as possible should be used to perform the inside test. Although not specifically targeted at user-space stochastic rasterization, Laine and Karras [67] present optimized inside tests that work well in practice. Because of the (u,v,t)-dependency, vertex attributes can no longer be interpolated by fixed-function hardware. However, the attributes can be manually interpolated by using the barycentric coordinates obtained as a byproduct from the inside test.

Computing surface shading for stochastic rasterization is very expensive, even if correct visibility is sacrificed by using MSAA, as previously explained. This problem is the topic of Paper **VI** and is addressed in more detail in Section 6.

Reconstruction The image obtained through the MSAA resolve (i.e., by averaging the samples in each pixel) can be quite noisy at low sample rates, especially for long motion trails and/or very out of focus regions. It is possible to use the sampled information to devise a filter in the 3D-5D domain to get a smoother image [33, 73]. The size and shape of the filter centered at some point depends on the motion of the local samples and the amount of defocus blur. Since the circle of confusion radius depends on the depth (distance to the focus plane), the filter size may vary greatly locally in (x,y,u,v) in presence of depth discontinuities. Vaidyanathan et al. [115] propose an algorithm that first splits the samples into depth layers, similar to Lee et al. [72], and subsequently composites the individually filtered layers to get the final image. Munkberg et al. [92] extend their proposed algorithm to include motion blur and using this approach near real-time performance can be achieved using an optimized, GPU friendly version of the algorithm [54]. An example of a reconstructed image can be seen in Figure 21.



International Karate (1986)
Commodore 64

The Legend of Zelda:
A Link to the Past (1991)
Super Nintendo

Unigine Heaven 4.0
Benchmark (2013)
PC

Figure 22: Rudimentary shadows have been used in games as visual cues for a long time. In games today, high resolution, dynamic shadows are the norm. (International Karate was developed and published by System 3. The Legend of Zelda: A Link to the Past was developed and published by Nintendo. Unigine Heaven benchmark courtesy of Unigine Corp.)

5 Shadows

Shadows have been featured since the early days of video games (see Figure 22). The sheer amount of research dedicated to the topic attests to the importance of including shadows in rendering [34, 122].

Revisiting the rendering equation (5), we see that a particular surface point only receives light if the visibility function has a non-zero value. For real-time graphics, it is far more efficient to compute the shading resulting from *direct* illumination from the light source and omitting indirect illumination. Computing the indirect light and shadow effects intertwines more with other, more general techniques for computing scene illumination, such as ambient occlusion and global illumination. While using point light sources will give the best performance, rendering shadows from area light sources can increase realism.

The human brain is aided by the shadows in a scene since they give additional information about the location of objects in relation to each other. Looking at a static scene lit by a point light source, we expect the shadows to have crisp edges. Now, suppose the object is instead moving rapidly, leaving motion blur trails in the image. In order to get a correct and harmonic viewing experience, the shadow of the object should behave accordingly. Figure 23 shows an example of this scenario, highlighting the importance of striving for realistic motion blurred shadows.

Shadow mapping The most popular family of shadow algorithms for real-time graphics is currently *shadow mapping* [120]. We will only cover it in its most rudimentary form for a spotlight source.

The algorithm is comprised of two passes. In the first pass, a *shadow map* is created by rendering the scene from the light source and storing the distances to

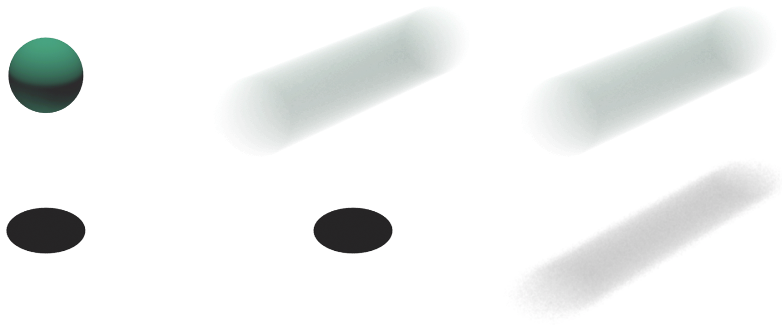


Figure 23: Left: *a static sphere lit by a point light source. The shadow on the floor below is crisp.* Middle: *the sphere is set in motion. The shadow is, incorrectly, computed from the static sphere, causing conflicting visual cues.* Right: *the proper shadow cast from the same moving sphere.*

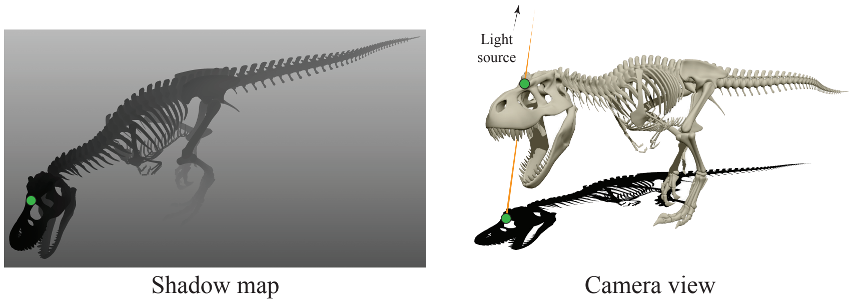


Figure 24: Left: *the shadow map is created by rendering from the light source. The distance is visualized with a gray scale.* Right: *the distance to the light source is compared to the depth value found in the shadow map to determine if a point is in shadow or not. In this example, the same lookup coordinate in the shadow map (green) is encountered twice in the camera rendering. Thus, these two points lie on the same light ray (orange). The light source itself is off-screen, above the skeleton. (T.rex 2 model courtesy of Joel Anderson)*

the closest hit point for each sample. In the second pass, the scene is rendered from the camera. Each point visible to the camera is transformed into the light's coordinate frame, and to determine whether the point is lit or in shadow, the depth obtained through a nearest neighbor texture lookup in the shadow map is compared to the depth of the transformed point. An example can be viewed in Figure 24.

The shadows obtained through shadow mapping are sharp, since the depth comparison gives a boolean result – the point is either fully lit or completely in shadow. In addition, the shadows may appear jagged since they are sampled at a finite resolution. Contrary to texture lookups for color, the depth values of the shadow map

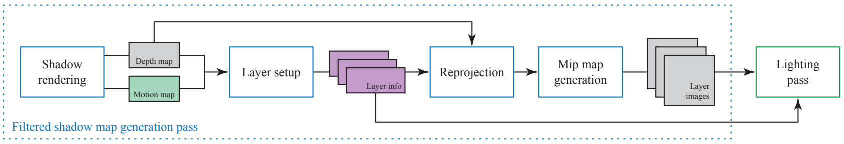


Figure 25: Motion blur algorithm overview. Depth and motion maps are created by stochastic rendering from the light source. Depth layers are then created for tiles of samples. Each tile and depth layer use one motion vector to approximate the local motion. Using the motion vector, the local samples are reprojected to a common time. Mip maps are then created to accommodate filtering. The tile/layer information and the mip mapped, reprojected images are now ready to be used to compute shadows when rendering from a camera.

cannot be filtered before performing the shadow test. Instead, percentage closer filtering (PCF) [105] can be used, which blends the *results* of multiple adjacent depth tests. However, the number of texture lookups depends on the size of the filter, which limits its usability. Instead, alternative shadow map representations can be used which can be filtered prior to the shadow test. This allows for pre-filtering using a simple mip map hierarchy, for example, which can then be sampled using a hardware-accelerated trilinear or anisotropic texture lookup.

In recent years, a few filterable shadow map alternatives have been proposed. Variance shadow maps (VSM) [29] store the distribution of depths, using the first and second moments, $\bar{z}$ and $\overline{z^2}$, within the filter footprint. Exponential shadow maps (ESM) [12] and convolution shadow maps (CSM) [11] approximate the shadow test with an exponential function or a series of functions.

5.1 Motion Blurred Shadows

Efficiently rendering motion blurred shadows is a relatively unexplored research topic. Akenine-Möller et al. [6] proposed time-dependent shadow maps (TSM) to render shadows using a stochastic rasterizer. A TSM is composed of several individual shadow maps, each using samples covering a small non-overlapping time interval. When shading a camera sample, the shadow map corresponding to the time interval of the camera sample is used. Similar to conventional shadow mapping, the nearest point is then used in the shadow test. While this method can be efficiently implemented, as Paper **V** demonstrates, it is non-trivial to get a properly filtered result, rather than a binary outcome per sample.

In Paper **V**, we present an algorithm for rendering filtered motion blurred shadows in real time. An overview of the steps in our approach can be viewed in Figure 25. The algorithm builds on the stochastic color reconstruction work by Munkberg et al. [92], but is modified in several ways to make it suitable for shadows. Our algorithm begins by creating a TSM, where each sample is also augmented with motion vectors. In the next couple of steps, we partition the shadow map into

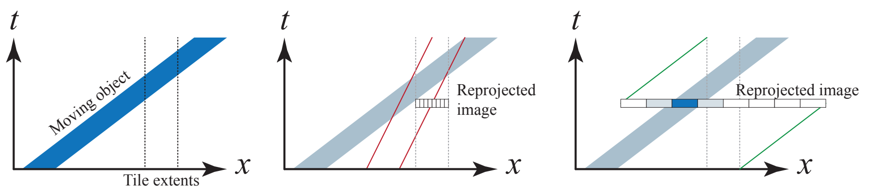


Figure 26: Left: *an epipolar image of an object moving quickly past a tile.* Middle: *using the algorithm by Munkberg et al. [92], the motion length must be clamped, as shown by the red lines. Otherwise, all samples that hit the object will be reprojected outside the image.* Right: *using our proposed algorithm, the reprojected image resolution is lowered, but all samples found within the tile will contribute to the image. No motion vector clamping is required.*

overlapping tiles and then further partition each tile into a set of depth layers. Each layer is approximated with a common motion vector. Next, the samples within a tile are filtered along the motion direction, by reprojecting them to a common time instant, $t = 0.5$. This creates an image, which can be translated along the motion direction in (x, y, z, t)-space to approximate the local geometry. This approximation is key to the efficiency of the algorithm. Since the (approximated) local motion is uniform, the costly sample search required by other methods [33, 73] is avoided.

Contrary to Munkberg et al., we reproject samples onto a stretched and oriented grid along the motion direction, which means that we are able to capture much larger motion and do not have to limit the length of the motion vectors, albeit we sacrifice spatial resolution to do so, as is illustrated in Figure 26. In the reprojected images, we store a variance shadow map representation, since this format is filterable.

In the lighting pass, we render the scene from the camera. For each visible point, we accumulate the contribution from the different layers to obtain the visibility term. We must handle this step differently than the color reconstruction described by Munkberg et al. The filter they use is composed of a spatial component in (x, y) and sheared component in the t-dimension. Both of these filter components are combined and evaluated simultaneously when the pixel color is sought. In our case, however, we can only filter along the motion direction, since the lookup footprint depends on the movement of the camera and the *shadow receiver point*. The same shadow map lookup coordinate may be used in several shadow tests since a light ray can pierce any number of surfaces, each moving in a different direction and velocity, thus preventing pre-filtering in (x, y).

Using our approach, we are able to render filtered motion blurred shadows in real-time performance. We handle many of the cases where other methods struggle, such as having moving shadow receivers and a moving camera.

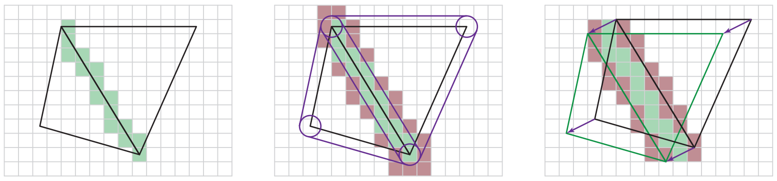


Figure 27: Left: *a shared edge between two triangles. The light green pixels show where the fragment shader potentially runs twice (depending on whether or not both triangles cover samples within the same pixel).* Middle: *adding a small defocus blur to the triangles increases the number of pixels overlapped by both triangles, as indicated by the red pixels.* Right: *similarly, adding motion results in a greater number of pixel overlaps.*

6 Reducing Shading

To improve realism, it may be desirable to use a high quality surface shading with varying material properties across the surface while having many light sources. This, however, requires long and complicated fragment shaders which can be very computationally expensive. MSAA can be used to lower the shading rate to roughly once per pixel, rather than once per sample. However, the shader must run more than once for pixels containing multiple triangles. Unfortunately, in the case of defocus and motion blur, triangles that are in motion and/or are far away from the focal plane will be blurry, which means that they overlap more pixels. As Figure 27 shows, the number of shader executions grows quickly as a consequence, and the benefits of using MSAA diminish.

Instead of relying on MSAA, some other means of reducing the shading rate must be found when rendering with motion and defocus blur. Prior art make a simple observation that the color of a surface point does not vary significantly when viewed from different points on the lens or at different time instances within the frame [27]. Although this is an approximation, the image quality is maintained in most cases, which means that shaded values can be efficiently reused. Since there is no longer any lens- or time-dependence, shading can now be performed in a separate *shading space*, which is parameterized over the triangles of the object, rather than in image space.

In order to reuse shading across multiple samples, a *shading cache* can be used. Shading is performed on demand as rendering progresses and the shaded colors are stored in a data structure. Before performing a potentially expensive shading operation, the data structure can be queried using the parameterized triangle coordinates in order to reuse shading information that is already computed. The data structure can be implemented as a hardware cache [24, 102] or in user-space using conventional GPUs [74]. At the time of writing, shading caches are not realized in actual hardware, and there are no API extensions to support them.

Another important observation that can be used to reduce shading operations is that

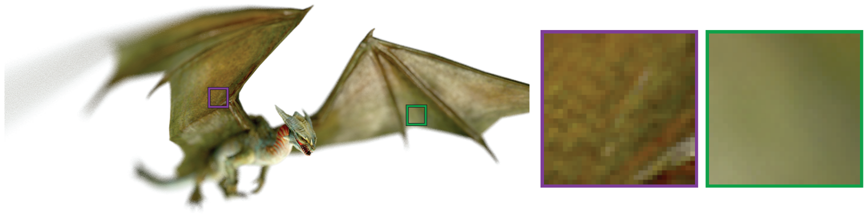


Figure 28: The left, purple inset shows a part of the wing intersecting the focus plane. The detail of the texture used is visible. In contrast, the region shown in the right, green inset is blurry and can thus be shaded less than once per pixel [117]. If MSAA is enabled, the region will be shaded multiple times per pixel, which essentially is the same behavior as super-sampling.

as motion and defocus blur increases, less and less of the high frequency features remain. In these cases, performing shading at the same rate as for still objects in focus is wasteful, as shown in Figure 28. To combat this problem, Vaidyanathan et al. [117] derive bounds on the image space sampling rate required to maintain image quality.

6.1 Texture Space Shading

In Paper **VI**, we present an algorithm for efficiently reusing shaded color when rendering scenes with defocus and motion blur. The algorithm is implemented using APIs targeting current generation GPUs. With our approach, we are able to render scenes with very expensive shading in real-time.

Contrasting prior art, we do *not* perform shading on the fly and nor do we defer it to a later pass. Instead, our approach is more reminiscent of light map rendering. For each object, we first construct a texture parameterized in shading space, which we populate with the shaded information. In the subsequent pass, the object is rendered from the camera and the shaded values are used. The rendering pipeline is used to quickly rasterize triangles directly into the shading space texture. By employing a mip map hierarchy, we can adaptively choose which level to rasterize to, and thereby execute fragment shading more or less frequently for different triangles. We select an appropriate shading rate based on the perspective projection and defocus blur of each triangle.

When rendering from the camera, the fragment shader is minimal, containing only a single texture lookup apart from inside testing and depth interpolation. Our algorithm is increasingly effective as shading complexity increases, compared to prior methods which approach super-sampling for increasingly blurred triangles [82]. For low-to-moderate shader complexity, we substantially outperform contemporary GPU based shading caches [74].

Since the publication of Paper **VI**, Clarberg and Munkberg [23] combined ideas from Clarberg et al.'s previous work [24] and our work to achieve a very efficient user-space shading system for motion and defocus blur. Like our method, they populate the shading space buffers by using fragment shading initiated by conservative rasterization. Contrasting our work, their parameterized shading space is based on the current camera view, rather than pre-computed shading space coordinates.

7 Contributions

I was the first author for all but Paper **I** in this thesis. The research was conducted in collaboration with Tomas Akenine-Möller, Jon Hasselgren, Jacob Munkberg, Jim Nilsson, and Robert Toth. I have been active in every aspect in crafting all the papers, including writing text, algorithmic design, implementation, and evaluation.

In order to understand the problems that arise with depth buffering, I implemented a depth unit simulator (described in Section 3) with extensive visualization features, such as detailed per-tile history information, cache contents, bandwidth tracking, and so on. This tool helped us in identifying unanticipated problems and behaviors, and to quickly experiment with ideas.

For Paper **I**, I was mainly responsible for implementation and evaluation work using the simulator and visualization tool. In addition, I contributed with the variable tile size cache optimization for uncompressed tiles described in Section 3. Paper **II** was the first paper in which we used the simulator and visualizer, and was my first encounter with depth buffer compression. I did most of the algorithmic design and implementation. For the following paper, **III**, my main goal was to demystify exactly how the depth function behaves for triangles with motion and defocus blur. I derived the depth function for 3D (motion blur) and 4D (defocus blur). Jacob tackled the 5D problem (combined motion and defocus blur) based on my $3-4$D derivations. Once again, I did all of the implementation and evaluation work using the simulator and visualizer. The simple, yet effective idea in Paper **IV** sprung from my realization of the potential problems with delayed feedback HiZ updates, and the limitations of the forward HiZ update strategy. The HiZ coarse buffer representation and merge heuristic found in the published paper is the result of extensive discussions with my co-authors who helped me test out and improve on my initial ideas. The algorithm was implemented and evaluated in Intel proprietary software simulating the entire rendering pipeline.

The basis for Paper **VI** was initiated by me as a small experiment, with the basic idea that a separate shading space could be rendered before primary visibility, in order to reduce shading. Jon joined the project, and was very involved in helping me with the implementation and design of the final algorithm. The extensions to the motion blur texture filter by Loviscach [77] to 4D and 5D was done by Robert. Together we generalized it to *N*D, which is the version presented in the paper. For Paper **V**, I explored a myriad of ideas for rendering real-time motion blurred shadows, before finally landing on the approach presented in the paper. As Jon and Jacob were concurrently working on ways to improve their reconstruction algorithm [54], a lot of fruitful conversations transpired, mainly regarding various compute shader optimizations.

8 Conclusions

It is not unthinkable that stochastic rasterization for motion and defocus blur can find its way into fixed-function hardware at some point in the future, given the attention the field has enjoyed in recent years. If this is ever to be realized, memory bandwidth reduction techniques such as our depth buffer compression, are a must. We do not claim, by any means, that we have found the optimal compression schemes for stochastically sampled depth buffers with motion and defocus blur, but we have taken the first steps into an important research field. In Paper **II**, we included the time dimension as a linear term in the predictor functions, but as Paper **III** later revealed, the depth function is much more complicated. Should the compressed cache approach be too computationally expensive to realize, perhaps a better post-cache compression scheme can be invented based on our depth function derivations. It might also be possible to design a hybrid pre- and post-cache compression scheme for stochastic depth buffers, similar to what we presented for static scenes in Paper **I**. Hopefully, we have sparked some interest in the subject, and should stochastic rasterization find its way into hardware, we hope that our research could provide some valuable insight.

Even without hardware support, it would be possible to include a compression codec targeting stochastically sampled buffers in the compression/decompression units found in hardware today. There are already API support for *alpha-to-coverage*, which stochastically alters the coverage mask based on the alpha value of a shaded fragment, which could benefit from the same compression scheme, stochastically sampled motion and defocus blur in user-space.

As the results in Paper **IV** indicate, our proposed HiZ algorithm works well on stochastic buffers. However, when objects are moving along the viewing axis or if the camera is dollying, there may not be a sufficient separation between layers, which would result in poor culling rates. In the future, we would like to explore how our HiZ algorithm can be extended to include this kind of problematic motion.

Power is often a limiting factor when designing GPUs. In future work, it would also be interesting to try to include latency and power usage in the model used in our depth buffer simulations. The model could be made even more complete by including a full cache hierarchy, possibly with different compression schemes at the different levels.

The natural extension of the layered motion blurred shadow approach, presented in Paper **V**, is to include area light sources. The area light can be sampled much like the lens for defocus blur, so the filters derived by Munkberg et al. [92] could possibly be used for this purpose as well. In turn, it would be interesting to see if their color reconstruction could benefit from the rotated grid, tile smoothing and streak reduction techniques used in Paper **V**.

In Paper **VI**, conservative rasterization is used in one of the passes to fill in missing shaded data. The implementation relies on the geometry shader and requires the vertex attributes to be replicated for multiple vertices [52]. This pass makes up about $6\% - 19\%$ of the execution time of the entire algorithm, which is a significant portion. At the time of writing, a new OpenGL extension has been introduced which enables hardware accelerated conservative rasterization in modern GPUs [18]. Using this extension, the performance of our algorithm could be further improved.

As workloads become more advanced and as display resolution continue to increase, the amount of fragment shading work that needs to be performed increases proportionally. In recent years, there has been a number of research papers that attempt to lower the amount of shading [23, 102, 116]. Our texture space shading technique, presented in Paper **VI**, could be used to lower the shading cost for applications other than motion and defocus blur. For instance, highly tessellated static meshes will also get diminishing returns from MSAA, due to the sheer number of triangles overlapping each pixel. Using our texture space shading approach, for example by parameterizing over the pre-tessellated patches, shaded values would be reused over triangle edges, thus potentially increasing overall performance. Another possible use for our algorithm is to try to reuse shaded values between frames, and perhaps progressively refine the shading over several frames, in order to improve quality over time. For stereoscopic rendering, our technique could be used to reuse shading for both eyes.

Paper I

A Compressed Depth Cache

Jon Hasselgren Magnus Andersson Jim Nilsson Tomas Akenine-Möller

Lund University Intel Corporation

ABSTRACT

We propose a depth cache that keeps the depth data in compressed format when possible. Compared to previous work, this requires a more flexible cache implementation where a tile may occupy a variable amount of cache lines depending on whether it can be compressed or not. The advantage of this is that the effective cache size increases proportionally to the compression ratio. We show that the depth buffer bandwidth can be reduced by on average 17%, compared to a system compressing the data after the cache. Alternatively, and perhaps more interestingly, we show that pre-cache compression in all cases increases the effective cache size with a factor of two or more, compared to a post-cache compressor, at equal or higher performance.

Journal of Computer Graphics Techniques, vol. 1, no. 1, pp 101–118, 2012.

1 Introduction

Reducing memory bandwidth usage in graphics processors is getting increasingly important, both from a performance perspective and from a power efficiency perspective. The data traffic to and from the depth buffer consumes a significant amount of bandwidth, and it is therefore important to reduce this traffic as much as possible. Common approaches include Z_{max}-culling [43], Z_{min}-culling [7], depth caching, and depth compression [50, 87].

We approach this problem by looking at the interplay between the depth cache and depth compression, and propose a system where the content in the depth cache is kept compressed when possible. The implication of this is that tiles (rectangular regions of samples/pixels) that can be compressed in the cache will utilize less storage there, and hence, the effective cache size is increased, with better performance as a result. Alternatively, the cache size can be reduced with unaffected cache performance. By using a compressed level one depth cache, we show that our system can reduce the depth buffer bandwidth by, on average, 17%, and this suggests that it can be potentially important to further study compressed cache architectures for graphics.

We suspect systems similar to ours have already been implemented, or at least considered, by graphics hardware vendors. However, we have not found any previously published work on such a system, and as such, this paper aims to fill that gap by describing the implementation alternatives and evaluating the expected performance.

2 Previous Work

The amount of public work in terms of depth compression is relatively sparse. Morein [87] presented a depth buffer compression system, which included a depth cache, using differential differential pulse code modulation (DDPCM) for compression. It is important to note that depths are required to be lossless by contemporary graphics APIs, and therefore, there is always a fallback that represents uncompressed depth data in a tile.

Hasselgren and Akenine-Möller presented a survey of depth compression algorithms [50]. This information was obtained from patent databases. In addition, they presented a twist of an existing compression algorithm that improved compression a bit. In their survey, a method called *depth offset* compression was presented, and it is likely the most simple depth compression algorithm available. The idea is to find the minimum, Z_{min}, and the maximum, Z_{max}, of the depths in a tile and to cluster the depth values into two groups, namely, one for the depths closest to Z_{min} and another for the depths closest to Z_{max}. The depths are then encoded relative to either Z_{min} or Z_{max}, and often, it is possible to use relatively few bits for these residuals.

Another interesting algorithm is *plane encoding* [50], where the rasterizer provides exact plane equations to the compressor. As a result, only a bitmask is needed per sample/pixel to identify which plane equation a sample/pixel belongs to, and hence, the residuals will always be zero. *Anchor encoding* is a variant, which uses a set of plane equations that are derived from the depths in the tile. The residuals are then encoded relative to one of these planes.

Lloyd et al. [75] develop a logarithmic shadow mapping algorithm, and realized that planar triangles become curved in their space, and hence, most previous depth compression algorithms could not be used. They compute first-order differentials and then use anchor encoding on the differentials. Ström et al. [111] presented the first public algorithm for compressing floating-point depths. The depth values are reinterpreted as integers in order to represent differences without loss. They use a predictor function based on a small set of the depths in the tile and then Golomb-Rice entropy encoding on the residuals. Pool et al. [100] present a general algorithm for floating-point data, which compresses the differences between a run of floating-point numbers, and uses a Fibonacci encoder for entropy encoding. However, any algorithm involving serialized entropy encoding is, in general, too expensive for our purposes. Inada and McCool [58] use a B-tree index to support random access for lossless texture compression with variable bit-rate. However, their tile cache, which is closest to the shading pipeline, is still uncompressed.

Andersson et al. [10] were the first to attack the problem of compressing depth buffers generated using stochastic motion blur rasterization. By incorporating the time dimension, t, into the predictor functions, better predictions were possible. They also noted that most previous algorithms for depth compression break down because they exploit that the depths of triangles are linear. This does not hold when triangles start to move. Interestingly, the depth offset encoding method performed reasonably well even for motion blur.

Compression of cached data in CPUs has received some attention. In general, CPU-based compression targets integer workloads, and in particular zero or near-zero values. Some techniques also try to detect repeating patterns. Lee et al. [71] first described a dynamic approach to compressed cache contents. They introduced a cache architecture capable of simultaneously handling both compressed and uncompressed lines. The core idea was to avoid cache set aliasing by including an extra bit in the index. By doing so, they avoid *data expansion*, resulting from previously compressed lines expanding to cover (in their case) two lines, and relax *fat writes*, which happens when two ways are written to in the same set.

Alameldeen and Wood [9] present a CPU system with uncompressed first level cache, while compressing second level data when possible. They introduce frequent pattern compression, which is a method for detecting and compressing a number of predefined data patterns. A three-bit prefix, stored with cache line tag data, designates one of eight possible compression encodings. Most modes are either covering lower-than-word resolution data types (five of eight), beside runs of zeroes or repeating byte values, and one mode designates uncompressed cache lines. For integer applications the single most useful mode is *zero run*, account-

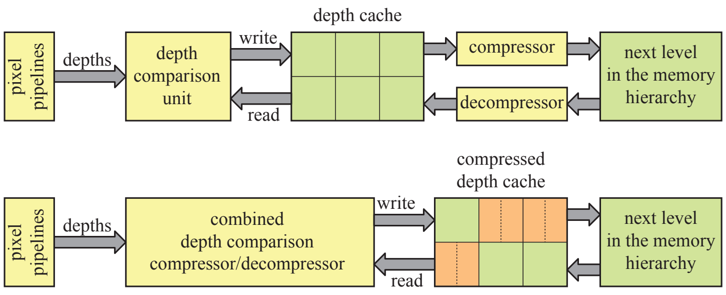


Figure 1: Top: the usual setup in a compressed depth architecture. From left to right: the pixel pipelines compute depths, which are delivered to a depth comparison unit. This unit communicates with a depth cache that can hold six tiles of depth data in this illustration. When depth data is communicated between the depth cache and the next level in the memory hierarchy, it will be compressed/decompressed on the fly when possible. Bottom: our proposal is to keep the content in the depth cache compressed when possible, and to efficiently perform the comparison, and compression/decompression between the pixel pipelines and the compressed depth cache.

ing for about 85% of all compressible patterns. Compression ratios for the integer applications were in the range 1.4–2.4. The compression ratio for the floating point applications was 1.0–1.3. It is clear that previous work on compressed CPU caches is not particularly applicable to depth compression. In particular, depth data rarely resembles the simplified patterns assumed by the compressed CPU cache approaches.

3 Compressed Depth Cache

An illustration of how our system compares to a common depth cache system with compression [50] is shown in Figure 1. In the common system, we use *post-cache codecs* which means that we only keep full and uncompressed tiles in the first-level depth cache, and place the compressor/decompressor (codec) between the cache and the next level in the memory hierarchy. The cache line size in the common system is therefore always equal to the tile size. Whenever a tile is evicted from the cache, we update the per-tile *header data*, which is a memory area separate from the depth buffer that flags the compression mode used for each tile. Typically, the Z_{min} and Z_{max} values are also stored in this area for hierarchical occlusion culling [7, 43]. The advantage of this system is that it has very simple cache logic, as the cache line size will be equal to the size of a tile. However, a drawback is that the tile size must be picked so that the compressed tile is large enough to efficiently burst when writing to or reading from RAM (or the next level in the memory hierarchy). This typically means that an uncompressed tile may be unnecessarily large, leading to wasted memory transactions and increased bandwidth when compression fails.

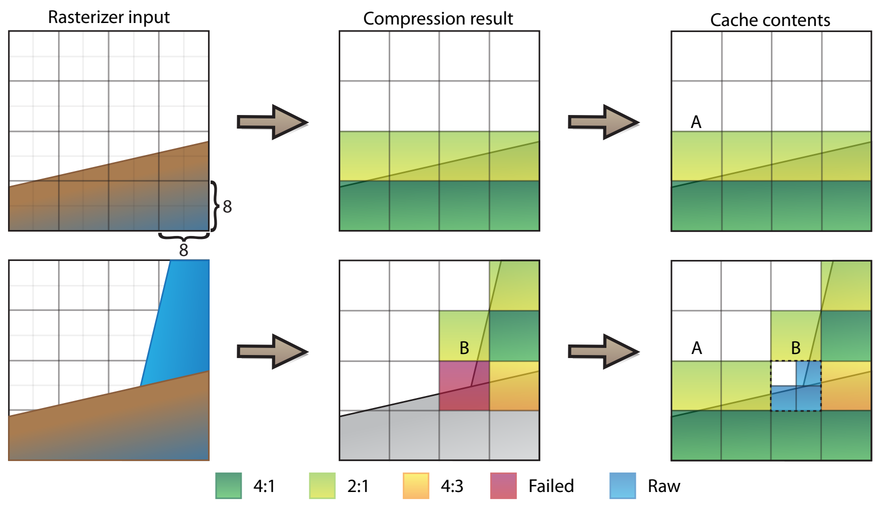


Figure 2: An example scene with two triangles being rasterized. Left: the input from the rasterizer (after z testing). Middle: the results and compression rates generated by the codec. Right: the final data stored in the cache. The tile size in this example is 8×8 *pixels, equal in storage to four cache lines when not compressed. This indicates that this example system may compress to 25%, 50%, and 75%. For the first triangle, it is worth noting that tile A could also be stored as a single RAW cache line, which actually requires less data than the compressed representation (2:1 in this case). However, we keep the compressed representation as our system does not allow recompressing tiles that have been reverted to uncompressed format. The assumption is that the compressed representation will be useful the next time we rasterize a triangle covering that tile. In this example, tile B completely fails compression and is stored in RAW format, but still only 3 cache lines are required.*

We propose using a more flexible cache where the line size is decoupled from the tile size and simply reflects what is efficient for a memory transaction. Furthermore, in contrast to the common setup, we put the compression/decompression logic before the cache, and we call this a *pre-cache codec*. The benefits of this system are twofold. First, we can store compressed tiles in the cache, thereby growing the effective cache size proportionally to the compression ratio. Second, the tiles that cannot be compressed may more easily be split into a number of cache lines, and we can update only the lines touched by a triangle. This pass-through compressor, which stores cache lines that cover smaller screen-space regions than the full tile, is called RAW in this paper. The challenge in this solution is that we complicate the logic involved in depth testing and updating of a tile. Furthermore, since our compression algorithm is now placed before the cache, it needs to have lower latency and higher throughput than if placed after the cache. However, at the same time, the required throughput between the depth comparison unit and the depth cache (see Figure 1) decreases as the data is compressed in the cache. For example, if we get an average compression rate of 50% we could harvest the halved throughput by, e.g., reducing the data path width or reducing the clocking of the cache.

The algorithmic flow of our depth system is illustrated in Figure 2. The rasterizer generates *input tiles* of samples for the current triangle. When a tile is received, we first perform hierarchical depth culling to determine whether the tile can be trivially discarded or accepted. For trivially accepted tiles, we attempt to compress the input tile data and allocate the appropriate number of lines in the cache. If the depth culling result is ambiguous, the tile header data is first accessed to determine whether the frame buffer depth data is compressed or not. For uncompressed tiles, the coverage mask of the input tile is used to read the appropriate lines into the cache, and then depth testing is done. In our implementation, we assume that tiles incrementally become more difficult to compress and therefore, we do not attempt to re-compress tiles that already failed compression unless the current triangle overwrites the entire tile. This also greatly simplifies the implementation as recompression would have to consider cases when a tile only partially exists in the cache. For compressed tiles, we read the full compressed tile from the cache and decompress the data. After that, depth testing is done followed by an attempt to merge the resulting data into the compressed representation. If the merge fails, we first attempt a full recompression, and if that also fails, the data is stored uncompressed (RAW). We experimented with partially reading compressed tiles, only accessing the cache lines required to decompress the samples overlapped by the input tile coverage mask. However, most compression algorithms require global header data, and in practice, the more complex implementation was not motivated by the very modest bandwidth gains.

In practice, the required changes to the depth system and cache logic are quite small. We need to compute cache keys on a per-line granularity, rather than a per-tile granularity, so the codecs should use actual memory addresses rather than a tile index. The biggest challenge occurs when a tile that only partially exists in the cache is evicted. Some operations, such as computing per-tile Z_{min} and Z_{max} values, requires the full tile data. We solve this by performing hierarchical depth culling on a per cache line granularity, thus guaranteeing that the cache line will always exist in the cache. Also, if we want to combine pre-cache and post-cache codecs in the same system, we must verify that the full tile exists in the cache in order to perform post-cache compression. We accomplish this by allowing peeking into the cache and check if the whole tile is present before evicting it. Since evictions are relatively infrequent, we believe this will be reasonably efficient. However, an alternative approach is to allocate one extra bit per cache line in the per-tile header data, and directly flag which parts of the tile are present in the cache. This operation is very efficient, but at the cost of a slight bandwidth increase for the tile headers.

In this paper, we focus only on the plane encoding and depth offset compression algorithms. The reason is that they have simple implementations, and therefore, we have been able to design efficient and incremental compression methods, which makes them good candidates for pre-cache codecs. Although we leave it for future work, we would like to mention that other traditional compression algorithms, such as anchor encoding [50], could also potentially be adapted for pre-cache compression. In our pipeline, we use a clear mask per tile that indicates which samples are cleared, so the minimum, $Z_{\min}$, and maximum, $Z_{\max}$, depth values for a tile are computed using only valid samples.

3.1 Plane Encoding

In plane encoding, the representation for a tile is a list of plane equations, which can reconstruct triangle depth exactly, and a per-sample bit mask that indicates which plane a sample belongs to. On-the-fly decompression from such a representation residing in the cache is straightforward. Assume we would like to decompress the depth of a certain sample/pixel location, (x_s, y_s). The bit mask value is used as an index, i, into the set of plane equations, and the plane equation is simply evaluated as $z = c_0^i + c_x^i \cdot x_s + c_y^i \cdot y_s$, where the constants c_0^i, c_x^i, and c_y^i together define plane equation i.

When a triangle is being rasterized, the rasterizer forwards the plane equation to the pre-cache codec. Depth comparisons are done by decompressing depth values as described above. If at least one depth value passes the depth test, the incoming plane equation is added to the compressed representation in the cache, and the bit masks are updated for each affected sample/pixel. Note that the size of the compressed tile will dictate how many plane equations can be stored in a compressed tile, and when there are no more available indices for new plane equations, the tile has to be decompressed and put into the cache again in uncompressed format.

There are different strategies for adding a new plane. In the simplest implementation, the planes are just added to the list of planes and compression fails when too many planes overlap a tile. However, better compression is possible by deleting unused planes from the header, either by scanning the index bitmask for unused bit combinations, or by keeping counters of how many samples belong to each plane. In such an implementation, the compressor must be able to work with one more plane than is representable by the compressed format.

3.2 Depth Offset Compression Algorithm

Depth offset is a very simple compression algorithm, but it works surprisingly well. It does not enable high compression ratios, but it successfully manages to compress many tiles with moderate compression ratios. This makes it rather efficient overall. In addition, it is a simple algorithm from an implementation perspective. Recall that the compressed representation consists of two reference values, $Z_{\min}$ and $Z_{\max}$, a bit, m_{xy}, per sample that indicates whether a sample's residual is relative to $Z_{\min}$ or $Z_{\max}$, and then an n-bit per-sample residual, r_{xy}. The depth values are reconstructed as $z(x,y) = Z_{\min} + r_{xy}$ if $m_{xy} = 0$, and otherwise as $z(x,y) = Z_{\max} - r_{xy}$.

It should be noted that the best bit distribution depends on the cache line size and

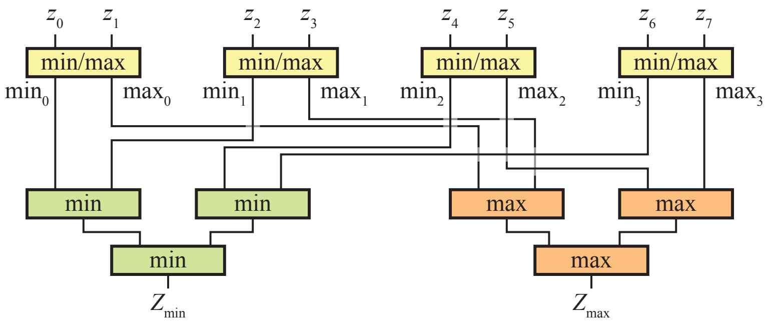


Figure 3: Computation of Z_{min} and Z_{max} using tree of comparisions for eight incoming depth values, z_i, $i \in \{0,\ldots,7\}$.

the tile size. However, we find that it is often sufficient to quantize Z_{min} and Z_{max} to 16 bits precision, and use the remaining bits for the residuals. For compression, there are more options, and below, we present two different ways to compress the depth in a tile when a new triangle is being rasterized.

3.2.1 Brute-force Approach

In this approach, we first decompress all depth values in the tile, as described above, perform depth tests, and update the depths that pass. Then the Z_{min} and Z_{max} of these depths are found using, for example, a tree-like evaluation as shown in Figure 3. In general, for s depths, such a tree will use $s/2+2(s/2-1) = 3s/2-2$ comparisons to compute both Z_{min} and Z_{max}.

The residuals, r_{xy}, and the selector bit, m_{xy}, are straightforward to compute. We just compute residuals from Z_{min} and Z_{max}, respectively. If either residual is small enough to encode in the given budget, we set m_{xy} to flag the appropriate reference value. Otherwise, the tile fails compression and needs to be stored in uncompressed form.

In the next subsection, we present a conservative and less expensive approach to updating Z_{min} and Z_{max}. The rest of the algorithm is intact though.

3.2.2 Opportunistic Approach

We base this compressor on the assumption that the depth pipeline supports hierarchical Z_{min} and Z_{max}-culling [7, 43]. These algorithms require conservative estimates of the minimum $Z^{\text{tri}}_{\text{min}}$, and the maximum depth, $Z^{\text{tri}}_{\text{max}}$, of a triangle inside a tile. Regardless of exactly how they are computed, we can assume they are readily available since the hierarchical culling unit is placed before the depth compression unit in the pipeline.

We can exploit these estimates during compression by assuming that $Z_{\text{min}} = \min(Z_{\text{min}}, Z^{\text{tri}}_{\text{min}})$, and $Z_{\text{max}} = \max(Z_{\text{max}}, Z^{\text{tri}}_{\text{max}})$ are good estimates for the true minimum and maximum values of the tile. We then compute all residuals as in Section 3.2.1. As a small optimization, we use only the triangle values if the current triangle overwrites the entire tile.

In practice, this will, in the majority of cases, cause our depth range to grow until a tile can no longer be compressed. However, the implementation is more efficient as we can avoid the rather costly Z_{min} and Z_{max} computations. We suggest that this implementation is combined with a post-cache brute force compressor. The simpler pre-cache codec will handle the high throughput data and keep it compressed in the cache for as long as possible. If the compression fails, the more expensive post-cache codec will refine the Z_{min} and Z_{max} values and re-compress the tile if possible. When the data is read back into the cache, the pre-cache codec can use the refined values as a starting point.

As a further optimization, we note that the residual computations can be done in two passes. First, the residuals are computed from Z_{min}, and in the following pass from Z_{max}. The second pass is conditional and can be skipped if all samples can be encoded relative to Z_{min}. Our tests indicate that it is sufficient with one reference value for 55% of the tiles, which may save substantial power in a hardware implementation.

4 Results

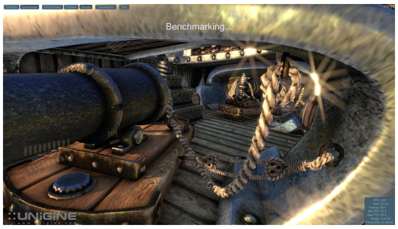
We evaluated our system using a functional simulator, written in C++, where it is possible to change cache settings, tile sizes, and configure the compression algorithms. Our simulator implements common depth buffer optimizations, such as Z_{min}- and Z_{max}-culling, and fast clears [50]. These optimizations are used for all our measurements, even for the uncompressed reference bandwidth, so the bandwidth gains presented here come strictly from the compression algorithm and the cache system described in this paper. Also, we only present figures for the depth buffer bandwidth, since the bandwidth for tile header data (Z_{min}, Z_{max}, and clear mask) is the same regardless of which type of cache (pre/post) is used.

DirectX 11 supports 32-bit floating point and 24/16-bit integer data. Of these formats, the 24-bit integer is still by far the dominating use case, and is used by all our workloads. Looking at codecs for 32-bit floating point depth data is interesting future work, but outside the scope of this paper. DirectX 11 only supports 24-bit integer depth when coupled with a stencil buffer, we therefore assume that most hardware vendors rely on their depth compression to reduce the bandwidth, and store the full 32 bits (D24S8) when a tile fails compression, even if the stencil is unused. Alternatively, stencil can be compressed along with the depth data, but we have also left this for future work. Therefore, the bandwidth figures presented for the RAW algorithm include 32-bit reads and writes per sample.

For the evaluation, we used the scenes shown in Figure 4 rendered at 1920×1080

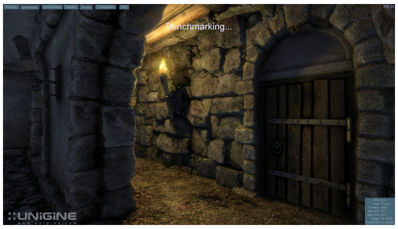
Heaven A - 158K tris, 23 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	28.9M	47%	40%	35%	26%
32 kB	23.6M	47%	41%	34%	27%



Heaven B - 346K tris, 45 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	59.0M	51%	46%	34%	29%
32 kB	52.4M	51%	47%	34%	29%

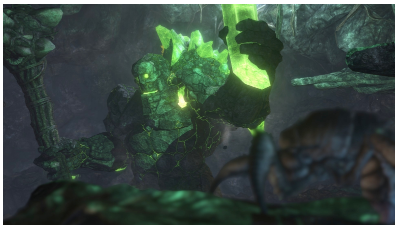


Heaven C - 283K tris, 25 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	38.9M	49%	42%	36%	29%
32 kB	32.9M	49%	45%	36%	32%

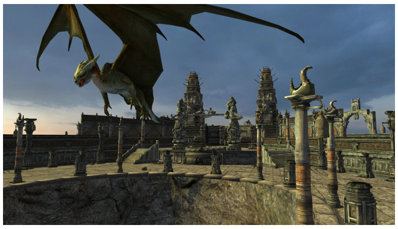
Stone Giant A - 447K tris, 23 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	36.9M	47%	41%	38%	32%
32 kB	31.9M	47%	42%	38%	34%



Stone Giant B - 218K tris, 34 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	44.5M	44%	40%	37%	34%
32 kB	40.9M	44%	39%	37%	33%



Dragon - 168K tris, 25 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	27.3M	47%	40%	35%	27%
32 kB	22.1M	48%	44%	34%	29%

Figure 4: The test scenes used in this paper were taken from the Heaven 2.0 benchmark by Unigine, the Stone giant demo by Bitsquid, and a Dragon scene created in house. We show the number of triangles and average triangle area in pixels (ppt) for each test scene. The tables show bandwidth figures, as a fraction of the RAW (no compression) bandwidth, for post/pre-cache depth offset (Post/Pre DO), and post/pre-cache plane encoding combined with depth offset (Post/Pre C). We used 8×8 sample tiles with 512-bit cache lines (4 lines per tile).

pixels resolution. We experimented with varying the tile size and bus parameters, but since the results were very similar, we only present numbers for a system using 8×8 sample tiles with 512-bit cache lines, which means that an uncompressed tile occupies four cache lines. We show the performance of two different configurations. The first (Post/Pre DO) uses only a depth offset codec, which compresses the data to either one or two cache lines (25% or 50%), where we use 6 and 14 bits, re-

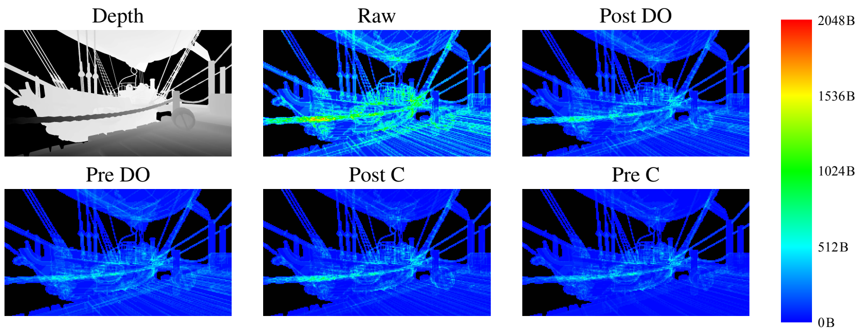


Figure 5: False color coding of the bandwidth for each 8×8 *tile using the different compression schemes described in this paper. The images show Heaven A with a 32 kB cache. The black areas are cleared.*

spectively, for the residuals. The second configuration (Post/Pre C) combines both depth offset and plane encoding. Here, we found that using plane encoding, with up to 4 planes per tile, for the 25% mode and depth offset for the 50% mode gave the best blend. This combination was just 1% from the bandwidth of using all possible combinations of plane encoding and depth offset compression. It should be noted that plane encoding is not well-suited as a post-cache codec since it communicates directly with the rasterizer. In order to generate post-cache results for the plane encoder, we still performed the compression pre-cache, but reserved enough cache lines to keep the tile data uncompressed in the cache. An alternative would be to compare with a post-cache anchor codec, but we feel that the results are more representative when comparing the exact same codec post- and pre-cache.

As can be seen from the results (Figure 4), post-cache depth offset rarely manages to compress below 50%, but we still get a significant 11% relative bandwidth gain from using a compressed cache, which amounts to 5.5% of the total RAW bandwidth. For the second configuration, with plane encoding (25%) combined with depth offset (50%), the pre-cache approach is even more successful, and here we see a 17% relative bandwidth gain or 6.0% of the total RAW bandwidth. In Figure 5 the per tile bandwidth of the different compression schemes for the Heaven A test scene can be viewed.

Cache size As can be seen in Figure 6, increasing the size of the depth cache gives diminishing returns. Typically, the “knee” of the curve indicates the most efficient cache size in terms of performance versus implementation cost. An encouraging result is that the pre-cache codecs do not only outperform the post-cache codecs significantly for a given cache size, but also seem to push the knee of the bandwidth curve to a lower cache size. For example, the knee for the pre-cache combined codec seems to lie around 10 kB, whereas the knee for the post-cache lies at around 16 kB and with higher bandwidth usage. An alternative way of read-

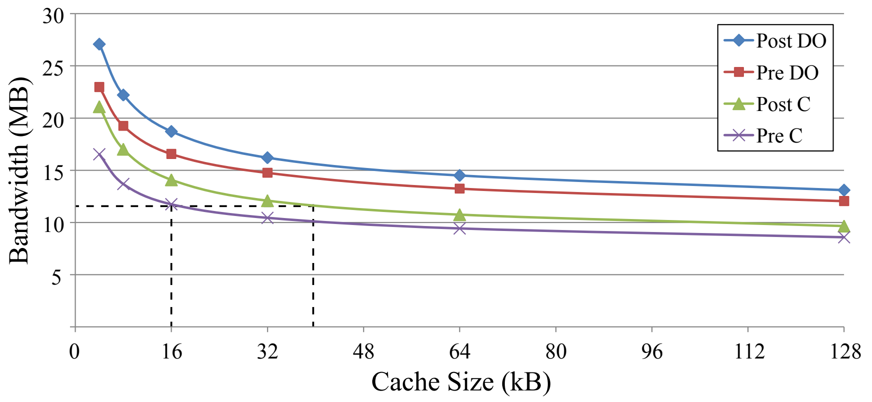


Figure 6: The average depth buffer bandwidth for the six test scenes with varying cache size. The graph compares post- and pre-encoding with depth offset, and post- and pre-encoding with combined plane encoding and depth offset. The dashed lines show identical bandwidths for the post-cache and pre-cache codec for the combined compressor (Post/Pre C).

ing the diagram is to decide on a target bandwidth and design the cache around that. The dashed lines in Figure 6 show an example of this, where we can reduce the cache size to roughly 40% for the pre-cache codec, which is directly proportional to the compression ratio of about 40%.

Opportunistic depth offset The impact of using the opportunistic depth offset compression algorithm (Section 3.2.2) was also measured. We found that it results in a bandwidth increase of 4.2% compared to the brute-force approach, or 1.8% of the RAW bandwidth. However, we still see a worthwhile improvement over the post-cache codecs by 8.3%, or 3.7% of the RAW bandwidth. Depending on the pipeline architecture, it may be beneficial to consider the opportunistic approach if the cost of passing around $Z_{\text{min/max}}$ values are considerably lower than recomputing them.

Recompression frequency With pre-cache codecs, the number of times a tile is compressed and decompressed will increase as a function of the cache size. In Figure 7, we show how pre-cache compression affects the number of tiles the compressor and decompressor must be able to process per frame. Larger cache sizes means that a tile needs to be compressed and decompressed more times with the pre-cache codecs. This is due to the fact that the tiles stay in the cache longer, and therefore they will be accessed more times before being evicted.

We note that compression scales better than decompression, which is good since compression is usually the more costly operation. For our design points of a cache around 16–32 kB, the opportunistic depth offset approach only requires

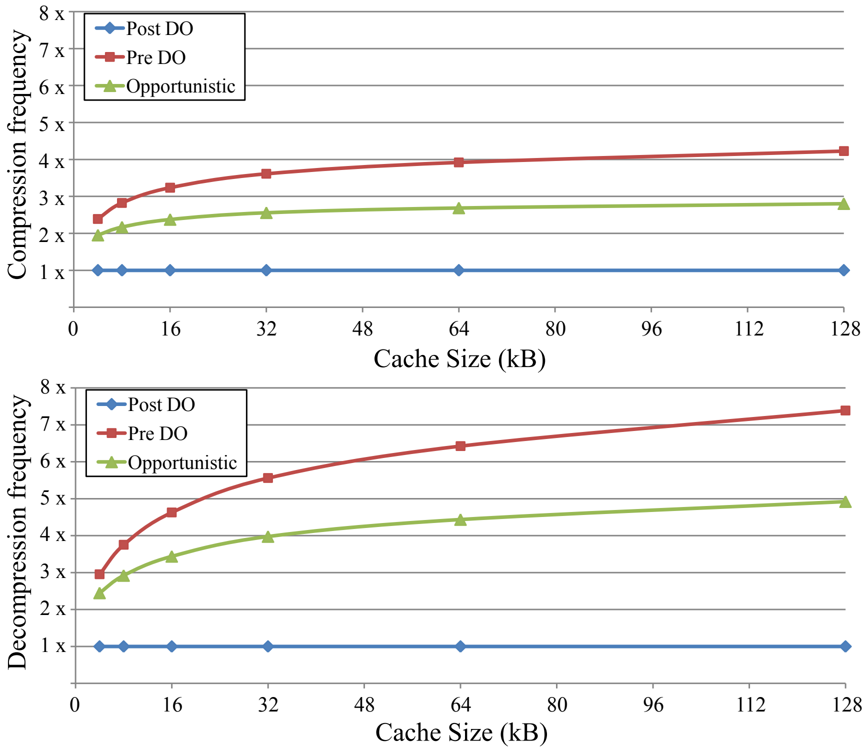


Figure 7: The average compression and decompression frequencies (i.e. the number of times a tile is compressed and decompressed) for the six test scenes with pre-cache, post-cache, and the opportunistic depth offset approach. The figures are normalized so that the post cache depth offset frequency is always $1\times$.

around $2.5\times$ higher compression throughput and about $3.75\times$ higher decompression throughput. This is very low considering that we use tessellated benchmark scenes which tend to use more and smaller triangles than most games.

With an increased focus on energy efficiency for graphics processors (see, for example, the work by Johnsson et al. [59]), we argue that the trade-off in our proposed system is very attractive. The reasons for this are as follows. First, our compressors and decompressors are very simple, using only a number of integer math operations that is largely proportional to the number of samples in a tile. Second, a memory access to DRAM uses more than three orders of magnitude the power of an integer operation [28]. Third, there are signs [63] that memory bandwidth development slows down even more than what we are used to. Hence, the motivation for a system with pre-cache codecs is clear, and could be even more relevant in the future.

5 Conclusions and Future Work

We have shown that using a flexible depth cache may enable pre-cache data compression and that such compression will roughly increase the cache size by the effective compression ratio. This can either be used to reduce bandwidth to RAM (or to the next level in the memory hierarchy), or to reduce cache size and free up silicon area without affecting bandwidth. In our implementation, we have shown a significant 17% average relative bandwidth reduction for reasonable pipelines, when compared to a post-cache codec. Similarly, we have shown that the cache size can be reduced by the effective compression ratio with no impact on performance. In fact, for all our measurements, the effective cache size was more than doubled when going from a post-cache codec to a pre-cache codec. This is true for the depth offset only configuration, and to an even larger extent for the combined depth offset and plane encoding configuration.

For future work, we would like to explore other existing depth compression algorithms and see how they perform in our system. Furthermore, it could be interesting to attempt to make the hardware implementations of complex codecs simpler (perhaps at the cost of reduced compression ratios). Also, since depth offset compression works rather well for stochastic motion blur rasterization [10], it will be interesting to see what happens to its performance in our system.

Acknowledgements

The authors thank Aaron Lefohn, Charles Lingle, Tom Piazza, and Aaron Coday at Intel for supporting this research. The benchmark scenes used in this paper are courtesy of Unigine and Bitsquid. Tomas Akenine-Möller is a Royal Academy of Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation.

Paper II

Depth Buffer Compression for Stochastic Motion Blur Rasterization

Magnus Andersson Jon Hasselgren Tomas Akenine-Möller

Lund University Intel Corporation

Abstract

Previous depth buffer compression schemes are tuned for compressing depths values generated when rasterizing static triangles. They provide generous bandwidth usage savings, and are of great importance to graphics processors. However, stochastic rasterization for motion blur and depth of field is becoming a reality even for real-time graphics, and previous depth buffer compression algorithms fail to compress such buffers due to the irregularity of the positions and depths of the rendered samples. Therefore, we present a new algorithm that targets compression of scenes rendered with stochastic motion blur rasterization. If possible, our algorithm fits a single time-dependent predictor function for all the samples in a tile. However, sometimes the depths are localized in more than one layer, and we therefore apply a clustering algorithm to split the tile of samples into two layers. One time-dependent predictor function is then created per layer. The residuals between the predictor and the actual depths are then stored as delta corrections. For scenes with moderate motion, our algorithm can compress down to 65% compared to 75% for the previously best algorithm for stochastic buffers.

High Performance Graphics, pp 127–134, 2011.

1 Introduction

In general, graphics processors are dependent on a number of techniques to reduce memory bandwidth usage. A memory access may cost several orders of magnitudes more in terms of power consumption than an arithmetic operation [28], and the gap between compute power and the available bandwidth continues to grow. Hence, it is well worth the silicon to add fixed-function units for those techniques.

Over the past few years, a lot of effort [6, 21, 37, 82, 113] has been put into getting stochastic rasterization [27] of motion blur and depth of field closer to interactive or even real-time rendering. The characteristics of stochastic rasterization is likely to influence some of the techniques for reducing usage of memory bandwidth, especially the ones based on compression.

Depth buffer compression is one very important technique [87] to reduce memory bandwidth usage. We review the known algorithms [50, 75, 87, 111] in the last part of Section 2. Note that lossy compression of other depth buffer representations can also be done [45, 106], but these do not solve the problem of compression of stochastically generated buffers.

Most existing algorithms depend of the fact that depth, z, is linear over a triangle, and this is exploited to construct inexpensive compression and decompression algorithms. One of the state-of-the-art algorithms, called *plane encoding* (described by Hasselgren and Akenine-Möller [50]), is particularly good at exploiting this. Briefly, the rasterizer feeds exact plane equations to the compressor, and hence do not need any residual terms. However, for motion blur, where each vertex may move according to a linear function, the depth function at a sample is a rational cubic function [45]. This fact makes it substantially harder to predict depth over an entire tile using a simple predictor function. As a consequence, the standard depth buffer compression techniques, especially the ones exploiting exact plane equations, will in many cases fail to compress such “noisy” buffers.

We present a novel technique for compression of stochastic depth buffers generated with motion blur. Our technique is able to compress a substantial amount of blocks of pixels, where previous techniques break down. This can be seen in Figure 1. One of the best algorithms, *plane encoding*, for static scenes breaks down completely. The best existing algorithm for compressing noisy depth buffers is called *depth offset* compression, and as can be seen in the same figure, that algorithm has decent performance for motion blur renderings. We believe our technique could become important in order to bring stochastic rasterization into fixed-function units in graphics processors.

2 Compression Framework

Here, we present a very simple general framework (Section 2.1) that can be used to describe all existing depth buffer compression schemes that we know of, which is done in Section 2.2.

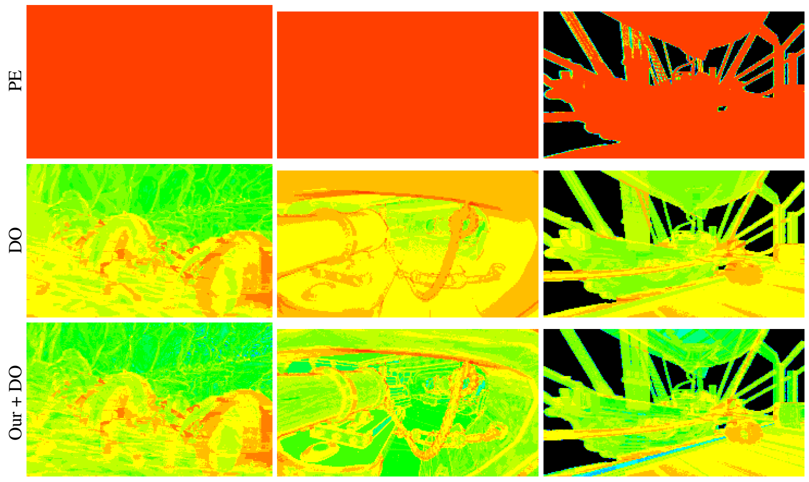


Figure 1: False color visualizations of depth buffer compression ratios of some scenes, where our algorithm outperforms the competition for compressing stochastic motion blur depth buffers. Plane encoding (PE) is one of the best algorithms for compressing static scenes, and depth offset (DO) compression is the best existing scheme for handling "noisy" buffers. As can be seen, plane encoding breaks down completely due to that plane equations turn into higher order rational polynomials when motion is introduced. Depth offset compression is substantially better, but the combination of our novel time-dependent compression algorithm with DO is even better. The color scale is, from low compression ratio (good) to high compression ratio (bad), blue, cyan, green, yellow, and red (which represents uncompressed).

2.1 Framework

Let us start with some assumptions. A block of $w \times h$ pixels, sometimes called a *tile*, is processed independently, and we assume that each pixel has n samples. The i:th sample is denoted by $\mathbf{s}^i = (s^i_x, s^i_y, s^i_t, s^i_z)$, where the first two components are the x- and y-coordinates of the sample inside the tile, and the third component, $s^i_t \in [0,1]$, is the time of the sample. It is also possible to add more components, for example, (s^i_u, s^i_v), for the lens position for depth of field rendering. Current depth compression schemes do not handle motion blur and depth of field explicitly, and hence do not have the time component nor the lens parameters. Note that all of (s^i_x, s^i_y, s^i_t) are fixed for a particular sample. It is only the depths that results from the rasterization process, and as a consequence, it is only the depth values, s^i_z, that needs to be compressed. Our notation for depth here is $s^i_z = z_c/w_c$, where z_c and w_c are the z- and w-components of a sample in clip-space, as usual. In general, a compression algorithm may attempt to exploit the fixed components for better compression.

From studying the sparse set of previous work on depth buffer compression [50, 75, 87, 111], we realized that all known schemes share three common steps, namely:

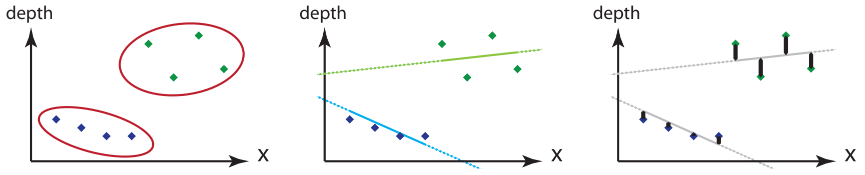


Figure 2: Illustration of the three steps of a depth buffer compression algorithm. These are, from left to right, 1) *clustering,* 2) *predictor function generation, and* 3) *residual encoding.*

1. clustering,
2. predictor function generation, and
3. residual encoding.

These three steps are illustrated in Figure 2. It should be noted, though, that an algorithm may choose not to have one or two of the steps above. A high level description of each of the steps follows.

Clustering is needed when there are, for example, a set of samples in a tile that belongs to a background layer, and the rest of the samples in the tile belongs to a foreground layer. In these cases, it is very hard to compress all depths in the tile using the same predictor function. The clustering step therefore attempts to separate the samples of a tile into two or several *layers*, where the samples in each layer typically should share some characteristics (e.g., lie in a common plane). The goal of splitting the samples into two or more layers is that each layer should ideally become simpler to compress compared to compressing all samples as a single layer. For a tile with only foreground samples though or when only one triangle covers an entire tile, clustering may not be needed. In general, a bitmask or several bitmasks are needed to indicate which layer a sample belongs to.

As the next step, each layer generates its own *predictor function*. The goal here is to use the depth samples and possibly their fixed (x,y,t)-coordinates to create a predictor function, $z(x,y,t)$, whose task is to attempt to predict the depth at each sample using an inexpensive (in terms of storage, generation, and evalation) function. For example, assume that a rectangle with small per-pixel displacements has been rendered to a tile. As a predictor function, one may use the plane of the rectangle, since it probably is a good guess on where the displaced depths will be. This guess will not be 100% correct, and so it is up to the next step to correct this.

Residual encoding must make sure that the exact depths, s_z^i, can be reconstructed during decompression of the tile, since a common requirement by graphics APIs is that the depth buffer is non-lossy. The residual, which is the difference between the predictor function, $z(x,y,t)$, and a sample's depths, is computed as:

$$\delta_i = z(x,y,t) - s_z^i. \tag{1}$$

Given a good predictor function, the residuals, δ_i, between the depth of the samples and the predictor function should be small. As a consequence, the deltas can be encoded using few bits. Good compression ratios can be achieved if there are a small number of layers, storage (in bits) for predictor function is small, and if the deltas can be encoded using few bits as well. Another success factor of a compression scheme is that the algorithm should succeed in compressing many tiles during rendering.

2.2 Depth Compression Algorithms

In this subsection, we will briefly describe the existing depth buffer compression algorithms in terms of our introduced framework. We will use the same names of the algorithms as introduced in the survey of depth buffer compression algorithms [50], and we will also describe Lloyd et al's algorithm [75].

The *depth offset* algorithm uses the z_{min} and z_{max} of the tile to cluster the samples into two layers; one being closer to z_{min} and the other with samples being closer to z_{max}. The predictor functions are as simple as they could be — for the closer layer, the differences between the sample's depth and z_{min} is encoded, and vice versa. In the *differential differential pulse code modulation* (DDPCM) algorithm, the idea is to compute first-order and second-order differentials, which essentially generate a plane equation as the predictor function. The differences between this plane equation and the actual depth values are encoded using two bits per sample. An extension is discussed, where a search for two different layers is performed, and each layer is encoded using a plane equation. Hence, this is a clustering step. *Anchor encoding* is very similar to DDPCM. As a predictor function, a plane equation is created from an anchor position in the tile, and two delta depth values stored as part of the plane equation. The residuals are encoded using five bits, and so could potentially be more useful for scenes rendered with lots of small triangles. Clustering is not used in this algorithm.

Plane encoding uses information from the rasterizer to create the clustering and exact plane equations. The rasterizer can assist the compression algorithm with bitmasks indicating which samples are covered by the triangle being rasterized, which means that clustering is done implicitly. In addition, the rasterizer can also provide full accuracy plane equations, meaning that there will be no residuals to encode, and so this is an example where the last step is missing. If the rasterizer is disconnected from the compression unit, a search algorithm for clustering into two layers can be used [50]. Each layer is then encoded using a plane equation with only a single bit as a correction factor, which gives a bit better performance.

Ström et al. [111] presented the first public algorithm for compressing floating-point depth buffers. The floating-point numbers were interpreted as integers in order to be able to encode differences. They used small set of previously decompressed depth values to feed a predictor function, and Golomb-Rice for entropy encoding of the residuals.

Lloyd et al. [75] develop a compression algorithm specifically for logarithmic shadow maps. In this case, the planar triangles become curved, and linearity cannot be exploited. Instead, first-order differentials are first computed, as done in DDPCM, and then anchor encoding is used on the differentials. No clustering is done for this method.

As can be seen, most of these algorithms use some kind of plane equation of the geometry as predictor functions. With motion blur, the depth at each sample for a single triangle is a cubic rational polynomial [45], and so it should be clear that they stand little chance at being successful at compressing scenes with stochastic motion blur (or depth of field). Some of the efficiency of these algorithms also stem from the assumption that the samples are positioned in a regular grid in xy, which is not true for stochastic samples. Depth offset does not use any such assumptions, but on the other hand, that algorithm uses the simplest possible clustering and also the simplest predictor functions. Akenine-Möller et al. [6] ran some initial tests on using depth offset compression for stochastic buffers, and found that it worked reasonably well, but conjectured that better algorithms should be possible. In the following sections, we will present new ways to approach stochastic depth compression using our framework.

3 New Algorithms

In this section, we describe our new contributions to the field of depth compression. First, we describe a simple clustering technique that can be applied to both stochastic depth buffers and to depth buffers rendered without blur. In Section 3.2, we introduce *time-dependent* predictor functions, and finally, we also describe how residual encoding is done in Section 3.3.

3.1 Clustering

In this subsection, we present a novel clustering technique, which is extremely simple and rather inexpensive to implement in hardware. Usually, two depth layers within a tile are separated by a large gap in depth. To find this separation, a simple approach would be to sort the samples according to their depths, and then find the largest depth difference between two successive samples. However, this is much too expensive for a hardware implementation, which needs to avoid sorting in order to reduce complexity. Instead, we propose a more pragmatic approach, which is not optimal, but tends to give good results. First, we split the depth interval between z_{min} and $\hat{z}_{max}$ for the tile into n bins, where $\hat{z}_{max}$ is the maximum z-value of non-cleared samples. For each bin, we store one bit, which records whether there is at least one sample in the bin. The bits are initiated to zero. Each sample is then classified to a bin based on the sample's depth value, and the corresponding bit set to one. Samples that are cleared may be ignored in this step. When all samples have been processed, each 0 signals a gap in depth of at

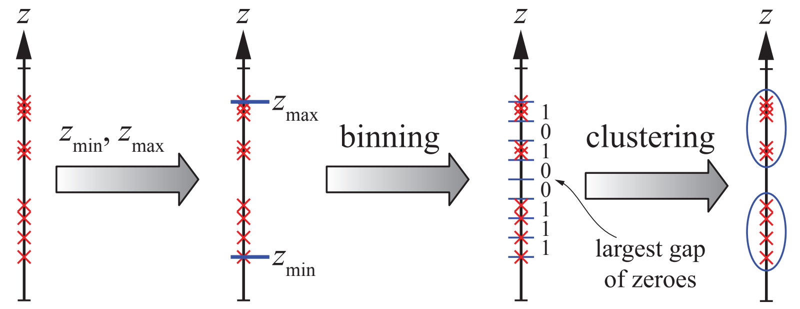


Figure 3: Illustration of our new clustering technique. From left to right: the depth values are marked as red crosses on the depth axis, and these depth values are then bounded by z_{min} and z_{max}. Then follows binning where, in this case, eight small bins between z_{min} and z_{max} are created, and bins with at least one depth sample are marked with 1, and otherwise marked with 0. Finally, the largest gaps of zeroes is found, and this separates the depths into two layers.

least $(\hat{z}_{max} - z_{min})/n$. By finding the largest range of consecutive zeroes, a good approximation of the separation of the two depth layers is obtained. This entire process is illustrated in Figure 3. Each of the sample clusters produced by this step is then processed independently as a layer by the predictor function generation step. In addition, the clustering process implicitly generates one or more bitmasks that indicate which layer each sample belongs to. The bitmask(s) will be part of the compressed representation of a tile. If needed, the samples can be clustered into more layers simply by finding the second and third (and so on) longest ranges of consecutive zeroes.

3.2 Predictor Functions

At this point, we have a bitmask, generated from the previous step, indicating which of the tile's $w \times h \times n$ samples that should be compressed for the current layer. Note that we may have only one layer, in which case all samples are included.

As mentioned earlier, most depth buffer compression schemes rely on that depth, $z = z_c/w_c$, is linear in screen space across a triangle. This means that a planar predictor function, such as the one shown below, is frequently used.

$$z(x,y) = a + bx + cy. \tag{2}$$

However, as soon as the time dimension is included so that motion blur is rendered, this is no longer ideal. We approach the problem of compressing stochastic buffers generated with motion blur by adding the time, t, to the predictor, but also combine x, y and t in different permutations. In general, we can use a predictor function,

$z(x,y,t)$ as follows:

$$z(x,y,t) = \sum_{mno} a_{mno} x^m y^n t^o. \tag{3}$$

For a hardware compressor, it is not feasible to try all possible combinations when performing compression, and having too many terms makes it very expensive to compute the predictor function. Based on the equation above, we propose to use three predictor functions, which have not been used in depth compression before. They were chosen due to their simple nature (few coefficients and low degree of the terms). For future work, it may be interesting to attempt to use other predictor functions, with higher degree polynomial terms, as well. Our new modes are listed below:

Mode	Equation
0: $Patch(x,y)$	$z_0(x,y) = a + bx + cy + dxy$
1: $Plane(x,y,t)$	$z_1(x,y,t) = a + bx + cy + dt$
2: $Patch(x,y,t)$	$z_2(x,y,t) = (1-t)(a_0 + b_0x + c_0y + d_0xy)$ $+t(a_1 + b_1x + c_1y + d_1xy)$

Our three modes have predictor functions with 4 or 8 unknown coefficients (a, b, etc). These can be obtained in a number of ways. However, instead of using conventional solutions with a high computational cost, we will instead use an inexpensive, approximate method to do this.

Since each tile contains many samples, it is possible to set up an over-constrained linear system when determining the coefficients of the predictor functions. For the predictor function, $z_1(x,y,t) = a + bx + cy + dt$ (mode 1 above), this would be done as shown below (where a has been removed, because it is computed in the final stage of the algorithm):

$$\underbrace{\begin{pmatrix} s_x^0 & s_y^0 & s_t^0 \\ s_x^1 & s_y^1 & s_t^1 \\ & \vdots & \\ s_x^{m-1} & s_y^{m-1} & s_t^{m-1} \end{pmatrix}}_{\mathbf{M}} \underbrace{\begin{pmatrix} b \\ c \\ d \end{pmatrix}}_{\mathbf{k}} = \underbrace{\begin{pmatrix} s_z^0 \\ s_z^1 \\ \vdots \\ s_z^{m-1} \end{pmatrix}}_{\mathbf{z}} \Leftrightarrow \mathbf{Mk} = \mathbf{z}, \tag{4}$$

where m is the number of samples in the current layer.

At a first glance, it may seem like a good idea to use least-squares fitting to solve this problem. The unknown is $\mathbf{k}$, and such an over-constrained system is often solved by performing a (costly) multiplication with the transpose of $\mathbf{M}$, which gives a square (and hence, possibly invertible) matrix: $\mathbf{M}^T\mathbf{M}\mathbf{k} = \mathbf{M}^T\mathbf{z}$. The solution is then: $\mathbf{k} = (\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T\mathbf{z}$, which is often called a pseudo-inverse [40], and gives us a solution in the least-squares sense (i.e., minimizing the errors in the 2-norm). Note that for the example in Equation 4, we need to invert a 3×3 matrix to solve for b, c, and d. This can be done with Cramer's rule,

While this would make for a decent estimation, our real goal is to minimize the maximum difference between the samples' depths and the predictor function, since this minimizes number of bits needed for the residual encoding (see Section 3.3). This can be done using minimax fitting [57], which is an even more expensive algorithm than least squares fitting. Since both these approaches are too expensive, we propose instead to use a heuristic data reduction technique. We reduce the samples in a layer into a more manageable number of *representative points*, which can then be used to solve a small 3×3 linear system. These points should be selected in a manner such that the resulting prediction function lies as close to the minimax solution as possible.[1]

Data reduction: The following algorithm is used to compute the representative sample points. When time, t, is *not* included (e.g., for mode 0), the first step is to find the bounding box in x and y for all the samples in the layer. The bounding box is then split into 2×2 uniform grid cells. For each cell, we find the two samples with the minimum and maximum depth values. The mid-point of these two samples (in xyz) is then computed. This gives us four representative points, $\mathbf{r}^{ij} = (r_x^{ij}, r_y^{ij}, r_t^{ij}, r_z^{ij})$, with $i, j \in \{0,1\}$, where i and j are grid cell coordinates in our 2×2 grid. There will be at most four representative points, and these will be used to compute the predictor function inexpensively. Analogously, for modes that takes t into account (mode 1 & 2), we can compute the bounds in t as well, and instead split the bounding box into $2 \times 2 \times 2$ grid cells. This results in at most 8 representative points, $\mathbf{r}^{ijk}$, with $i, j, k \in \{0,1\}$. An illustration of the data reduction algorithm for the 2×2 case can be found in Figure 4.

Next, we describe how we compute each mode's predictor function from these reduced representative data points.

Common step: All our three modes share a common step when computing their predictor functions. They all need to solve one or two 4×4 systems of linear equations to get the coefficients for the predictor function. To simplify this, we first move the origin to one of the representative points, and instead compute a later in the residual encoding step (see Section 3.3). This leaves three coefficients left to solve, and three remaining representative points. Any method suitable for solving 3×3 linear systems, such as Cramer's rule, can be used to compute these.

Mode 0: This mode was added mainly for static geometry, i.e., for parts of the rendered frame without motion blur. Therefore, it does not contain the time parameter. However, it will also be used in mode 2, as will be seen later. We propose to use a bilinear patch, which is described by $z_0(x,y) = a + bx + cy + dxy$. The motivation for this mode, compared to using just plane equations (see Section 2), is that the bilinear patch is somewhat more flexible, since it is a second-degree surface, and hence has a higher chance of adapting to smoother changes of the surface.

[1] Some experimental results were obtained by comparing our solution for a 4D plane (mode 1) to Matlab's least squares (backslash operator) and minimax (`fminimax`) solutions of the over-constrained system. A set of random tiles were retrieved from our test scenes, and the mean error span was calculated using all three methods. We were well within 10% of the minimax error, and on par with the least squares solution.

Mode 1: This mode describes a plane in four dimensions, i.e., $z_1(x,y,t) = a + bx + cy + dt$. This representation is useful for moving geometry, since it contains the dt term. In this case, we can use any four representative points, $\mathbf{r}^{ijk}$, where at least one has $k = 0$ and at least one has $k = 1$ in order to capture time dependence. Although a plane in four dimensions is not enough to capture all possible triangle movements, this approximation works surprisingly well, as we will see in Section 5.

Mode 2: This mode linearly interpolates two bilinear patches (P_0 and P_1) positioned at $t = 0$ and at $t = 1$, to capture a surface moving over time. The resulting equation becomes: $z_2(x,y,t) = P_0 + t(P_1 - P_0)$. To compute this representation, we first perform data reduction to produce $2 \times 2 \times 2$ representative points, $\mathbf{r}^{ijk}$. For $k = 0$, the four representative points, $\mathbf{r}^{ij0}$, $i, j \in \{0,1\}$, are used to compute the P_0 patch in the same way as done for mode 0. A similar procedure is used to compute P_1. Each patch, P_k, now approximately represents the tile data at the time $t_k = \frac{\max(r_t^{ijk}) + \min(r_t^{ijk})}{2}$. We now have all eight coefficients needed for this mode. In a final step, the two patches are positioned at times $t_0 = 0$ and $t_1 = 1$ through extrapolation, which gives us $z_2(x,y,t)$.

Missing data: Due to clustering and cleared samples, some grid cells in the data reduction step may end up without any samples, which means that representative points, $\mathbf{r}^{ij(k)}$, for such grid cells cannot be computed. To be able to compute the predictor functions, we choose to make reasonable estimations of the missing representative points using the existing representative points. For simplicity, we only fill in missing data over the xy neighbors, and not in t. Hence, for time-dependent modes, we use the same technique twice. If only one representative point is missing, i.e., one grid cell is missing samples, we create a plane from other three points, and evaluate it at the center of the empty grid cell.

If there are only two representative points, e.g., $\mathbf{r}^{00}$ and $\mathbf{r}^{01}$, and two empty grid cells, we create a new point, $\mathbf{r}^{10}$, as shown below:

$$\begin{aligned} \mathbf{e} &= \mathbf{r}^{01} - \mathbf{r}^{00}, \\ \mathbf{r}^{10} &= (r_x^{00} - e_y, r_y^{00} + e_x, r_t^{00}, r_z^{00}), \end{aligned} \tag{5}$$

where the first two components of $\mathbf{r}^{10}$ are created by rotating the difference vector, $\mathbf{e}$, 90 degrees in the xy-plane and adding it to the x and y of $\mathbf{r}^{00}$. The other components are simply copied from $\mathbf{r}^{00}$. This extrudes a plane from the vector from $\mathbf{r}^{00}$ to $\mathbf{r}^{01}$. When this third representative point has been created, we proceed as if only one representative point is missing. Finally, if only one representative point exists, this implies that the layer only contains a single sample. In such a case, only the a-coefficient is needed since it encodes a constant function, which is all we need to reconstruct a single depth value. As a consequence, the representative points are not needed in this case.

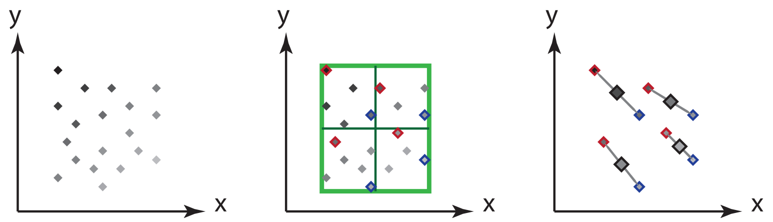


Figure 4: All our predictor functions uses a similar data reduction step. This figure illustrates how a set of samples are reduced to 2×2 representative points. ***(left)*** *We start with a set of irregular samples. The grayscale of the samples indicate their depths.* ***(middle)*** *The bounding box of the samples is found. The box is split in half in x and y. In each resulting sub-region, the samples with the maximum (blue) and minimum (red) depths are found.* ***(right)*** *The mean position and depth for each pair of min- and max-samples is used as a new representative point for the sub-region.*

3.3 Residual Encoding

In a final step, we compute correction terms that encode how a specific sample can be recreated from the predictor functions. We need to store two values for every sample. The first is a *layer index*, which associates that sample with a certain layer, as described in Section 3.1. Typically, we use only between one and two layers, so we need at most one bit per sample for this index. If a tile can be compressed using a single layer, we do not have to store these indices.

The second per-sample values to store are the correction terms, δ_i. These are found by looping over all of samples in the layer and computing the difference between the predicted value, $z(x,y,t)$, and the actual depth of the sample, s_z^i. During this phase, we track the required number of bits to store the correction terms, and also compute the a-constant for our predictor functions. The a-constant is set so that we only get unsigned correction terms (i.e., all samples lie above the predictor function).

Our correction terms are used in a slightly new way. For k correction bits per correction term, we reserve the value $2^k - 1$ as a *clear* value, and can hence only use correction terms of up to (and including) $2^k - 2$. However, we get the benefit of being able to signal whether a particular sample is cleared in a very inexpensive way. Otherwise, this is usually done using a particular value in the layer index, which is more costly.

4 Implementation

We use a depth compression architecture with a tiled depth buffer cache and a tile table [50]. This has been implemented in a software rasterizer in order to gather statistics about different algorithms and configurations. In order to reduce the dimensionality of the evaluation space, we decided to use 512 bits, that is 64 bytes, as the cache line width in our simulations. This implies that when we compress a tile, the compressed representation is always padded to the next 512-bit alignment regardless of the size of the desired memory transaction. In all of our tests, we use 32-bit fixed-point depth values in the depth buffer.

Furthermore, we have experimented with different tile sizes ($w \times h \times t$), and have arrived at two resonable sizes, namely, $4 \times 4 \times 4$ and $8 \times 8 \times 4$ samples per tile. We use these tile sizes both for $m = 4$ and $m = 16$ samples per pixel (SPP) rates. For $m = 16$, this means that 4×4 pixels would need $4 \times 4 \times 16$ samples, i.e., four $4 \times 4 \times 4$ tiles. Hence, the samples in a 4×4 pixel region are split along the time dimension. For $4 \times 4 \times 4$ tiles, the number of samples sums to $n_s = 64$ samples per tile, which fits in 4 cache lines in uncompressed form. Similarily, a tile of size of $8 \times 8 \times 4$ has $n_s = 256$, which occupies 16 cache lines in uncompressed form.

For each tile, we store 64 bits of tile table header information, which includes z_{min} and z_{max} and various mode bits. The involved memory bandwidth usage for this data is typically around a few percent of the total depth memory bandwidth usage, and so this layout has been left mostly unoptimized. In our results section, we will compare to the depth offset compression algorithm (see Section 2.2), and also to the combination of depth offset and our algorithm, since their strengths are somewhat complementary. The tile table header bit layouts for depth offset, our algorithm, and their combination are shown below.

	n_s	N	Mode bits 1	Mode bits 2	z_{min}/z_{max}
Depth offset	64	2	0	0	31 + 31
	256	4	0	0	30 + 30
Our algorithm	64	2	2	2	29 + 29
	256	4	2	2	28 + 28
Combination	64	2	2	2	29 + 29
	256	4	2	2	28 + 28

Here, N is a two or four-bit number that indicates how many cache lines that the current tile has been compressed to, where $N = 0$ indicates that the tile is uncompressed. Hence, if the value of N is 8 (and $n_s = 256$), the tile has been compressed down to 50% of its original size, for example. As can be seen in the table above, the depth offset algorithm uses its remaining tile table header bits to encode z_{min} and z_{max} at 31 or 30 bits each. For our algorithm, we store N as done for depth offset, and then we store two bits for the first layer, where 00 indicates that all the samples in the tile are cleared, 01 indicates that the layer is compressed using mode 0 ($Patch(x,y)$), 10 indicates mode 1 ($Plane(x,y,t)$), and 11 indicates mode 2 ($Patch(x,y,t)$). These modes are described in Section 3.2. The second two mode bits are used to describe layer 2, where 00 indicates that there is no second layer, and the rest is the same as for mode bits 1. This leaves 29 or 28 bits for each of z_{min} and z_{max}, which we have found to be sufficient precision for all our test scenes.

For the combination of the two algorithms, we can use the header layout from our algorithm with a small modification. If the *Mode bits 1* are 00, either we have a cleared tile *or* a tile compressed with depth offset, and this is determined by *Mode* *bits 2* being set to 00 for a cleared tile and 01 for depth offset. In our algorithm, the *Mode bits 2* are unused if *Mode bits 1* are 00, and therefore, we simply exploit unused bit combinations to fit depth offset compression into our tile header. This leaves us the same number of z_{min} and z_{max} bits as for our algorithm.

In our rasterizer, the z_{min} value for a tile is updated each time a sample depth, which is smaller than the current z_{min}, passes the depth test. The z_{max} value, on the other hand, requires that all samples are compared, and is thus updated only when the tile is evicted from the tile cache, prior to compression. We employ a tile cache of 64 kB and a tile table cache of 4 kB.

In the following, we describe the format of a compressed tile. The predictor function coefficients (a, b, c, etc) are stored in 32 bits each. For mode 0 and 1, this sums to 128 bits, while for mode 2, it amounts to 256 bits. When a single layer is used to compress a tile, we therefore need either $p = 128$ or $p = 256$ bits to encode the predictor. This leaves $k = \lfloor \frac{N \cdot 512 - p}{n_s} \rfloor$ bits for the residual, δ, for each sample.

If the tile contains two layers, two predictor functions must be encoded. The cost, denoted by p_1 and p_2, of each of these is either 128 or 256 bits. In addition, an extra bit per sample for the layer index must be stored, which costs n_s bits. Thus, for the two-layer mode, we have $k = \lfloor \frac{N \cdot 512 - n_s - \sum p_i}{n_s} \rfloor$ bits for the per-sample residual, δ.

We want to minimize the number of cache lines, N, needed to compress each tile in order to reduce memory bandwidth usage as much as possible. The value of N is calculated as follows:

$$N = \left\lceil \left(\underbrace{n_s \max k_i}_{\mathbf{M}} + \underbrace{n_s(n_l - 1)}_{\mathbf{I}} + \underbrace{\sum p_i}_{\mathbf{P}} \right) /512 \right\rceil, \qquad (6)$$

where $n_l \in \{1, 2\}$ is the number of layers. The **M** term is the maximum correction bits needed for any layer, the **P** term is the sum of the predictor coefficients, and **I** is the predictor index (only used for two layers). The division by 512 is to convert the number of bits into number of cache lines. In our implementation, we simply evaluate N for all compression combinations and pick the best one.

5 Results

Using the test scenes in Figure 5, we evaluated *depth offset* compression, which is described in Section 2.2, our algorithm, and the combination of our algorithm and depth offset. This combination is described in Section 4. We used tiles of either $4 \times 4 \times 4$ ($w \times h \times t$) or $8 \times 8 \times 4$ samples when measuring compression rates. This means that the time dimension is split into four groups for 16 samples per pixel. We do not claim to have found the optimal tile configurations, but they worked well for our scenes and memory system, and gave consistent results.

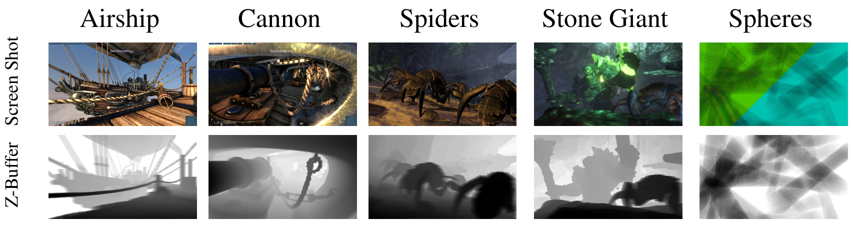


Figure 5: Images from the test scenes used to generate our results. The images are taken from the Heaven benchmark and Stone Giant demo and feature highly tessellated geometry. The Spheres scene is a synthetic benchmark scene intended to stress test highly varying motion. The top row contains reference screenshots without motion (except for the Spheres scene which includes motion), and the bottom row shows depth buffers, rendered using our rasterization framework, including motion blur. All images were rendered at 1920×1080 *pixels resolution. The images Airship and Cannon images are courtesy of Unigine Corp., and the Spiders and Stone Giant images are courtesy of BitSquid.*

Compression ratios for the algorithms are presented in Figure 6, where compression ratio is defined as the total compressed z-bandwidth for a frame divided by the total uncompressed z-bandwidth. The best improvement occurs for the Cannon scene, with 4 samples per pixel and $8 \times 8 \times 4$ samples per tile. In this case, depth offset compresses down to about 75%, while our algorithm manages to compress down to 55%, which is a substantial difference. The combination of depth offset and our algorithm is consistently better than our algorithm, which implies that they are somewhat complementary. In general, the combination of the two algorithms improves upon depth offset compression by between 3–20%.

When compressing tiles with our algorithm, more than one mode could often be used to achieve the best compression. For mode 0, this happens around 75% of the time, for mode 1 around 85%, and mode 2 lands on about 60%. 20–50% of the time (depending on scene and if one or two layers are used), a tile can be best compressed with one mode exclusively. Again, mode 1 is by far the most important, and can be used roughly 60–70% of the time to achieve the best compression, where as mode 0 is used for 10–30% of the tiles. Mode 2 is only used up to 5% of the time for $4 \times 4 \times 4$ tile sizes. For $8 \times 8 \times 4$ tiles, however, this figure raises slightly to 5–15%.

We present results for both $4 \times 4 \times 4$ and $8 \times 8 \times 4$ tile configurations. The reason for this is that, while our algorithms performs better on configurations with more samples in a tile, the larger tile sizes gave a larger overall bandwidth for the highly tessellated scenes (roughly 50% higher in our framework). The tile size of a hardware architecture will most likely be based on balancing a number of factors, such as the rasterizer, shader unit, memory controller, and so on. Therefore, the two configurations are presented in order to give a sense of how the compression efficiency scales with tile size.

In Figure 7, we study the performance of the algorithms as a function of increasing motion. As can be seen in the diagrams, our algorithm is substantially better than

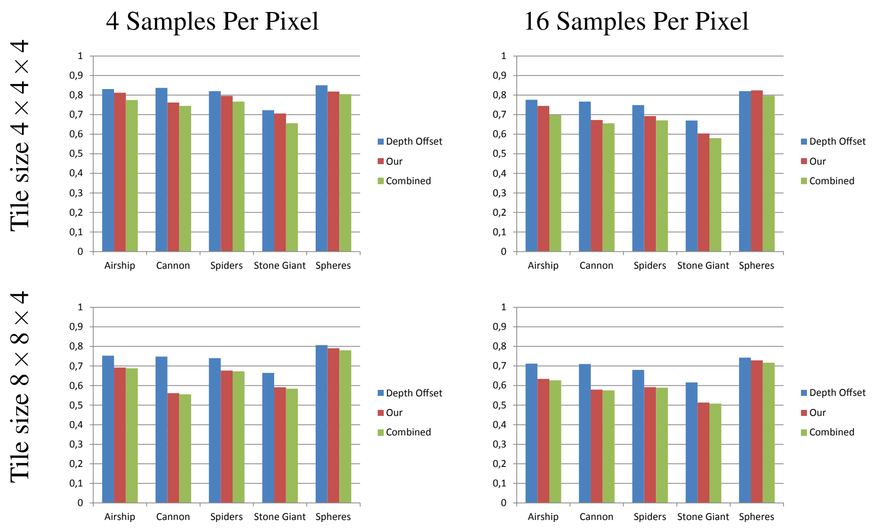


Figure 6: Effective compression ratios for depth offset compression, our algorithm, and the combination of depth compression and our algorithm. We present compression results for 4 and 16 samples per pixel, as well as for two different tile configurations, namely, $4 \times 4 \times 4$ *and* $8 \times 8 \times 4$.

depth offset compression for small amounts of motion, and then for larger amounts of motion, the gap is somewhat reduced.

6 Discussion

We have presented an algorithm that preserves the benefits of traditional depth buffer plane encoding algorithms while also being robust for stochastically sampled depth data, which previous algorithms have not even attempted to target.

To generate our results, we opted to split the *time* dimension into different tiles when working with 16 samples per pixel. When motion grows large relative to the triangle size, the rasterizer access pattern will become increasingly random which, in turn, makes the frame buffer cache less efficient. It is therefore important to design the whole frame buffer and rasterizer with this in mind, and to find a good balance which gives low bandwidth for static scenes, while scaling gracefully with increasing motion. We leave a detailed study of this for future work.

In our research, we have taken a first step towards efficient depth buffer compression of motion blur renderings. However, we believe that our work can be improved further, and in a near future, we will investigate various optimizations of our basic algorithms. For future work, we also want to look into how our algorithms can be extended to handling depth of field, as well as the combination of motion blur and depth of field at the same time.

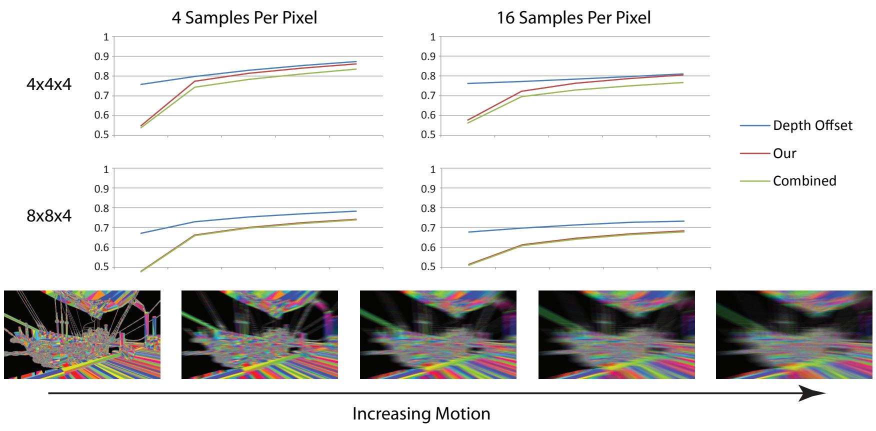


Figure 7: Effective compression ratios for depth offset compression, our algorithm, and the combination of depth offset with our algorithm, with varying amounts of motion blur. Note that our algorithm performs better than depth offset across all amounts of motion for each configuration, and that the combination of the two algorithms is even a bit better for the $4 \times 4 \times 4$ *tile configuration. In general, all three algorithms have stable performance with varying amounts of motion, which is in contrast to traditional planar encoders that typically break down when motion is introduced.*

Acknowledgements

Thanks to Tobias Persson from BitSquid for letting us use the StoneGiant demo, and to Denis Shergin from Unigine for letting us use images from Heaven 2.0. 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.

Errata

Omitting a and using the least-squares method on the system described in Equation 4 will not yield the b, c and d coefficients as intended. Instead, the equation should read

$$\underbrace{\begin{pmatrix} 1 & s_x^0 & s_y^0 & s_t^0 \\ 1 & s_x^1 & s_y^1 & s_t^1 \\ & & \vdots & \\ 1 & s_x^{m-1} & s_y^{m-1} & s_t^{m-1} \end{pmatrix}}_{\mathbf{M}} \underbrace{\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}}_{\mathbf{k}} = \underbrace{\begin{pmatrix} s_z^0 \\ s_z^1 \\ \vdots \\ s_z^{m-1} \end{pmatrix}}_{\mathbf{z}} \Leftrightarrow \mathbf{Mk} = \mathbf{z}.$$

Paper III

Stochastic Depth Buffer Compression using Generalized Plane Encoding

Magnus Andersson Jacob Munkberg Tomas Akenine-Möller

Lund University Intel Corporation

ABSTRACT

In this paper, we derive compact representations of the depth function for a triangle undergoing motion or defocus blur. Unlike a static primitive, where the depth function is planar, the depth function is a rational function in time and the lens parameters. Furthermore, we show how these compact depth functions can be used to design an efficient depth buffer compressor/decompressor, which significantly lowers total depth buffer bandwidth usage for a range of test scenes. In addition, our compressor/decompressor is simpler in the number of operations needed to execute, which makes our algorithm more amenable for hardware implementation than previous methods.

Computer Graphics Forum (Proceedings of Eurographics), vol. 32, no. 2, pp 103–112, 2013.

1 Introduction

Depth buffering is the standard technique to resolve visibility between objects in a rasterization pipeline. A *depth buffer* holds a depth value for each sample, representing the current closest depth of all previously rendered triangles overlapping the sample. In a stochastic rasterizer with many samples per pixel, the depth buffer bandwidth requirements are much higher than usual, and the depth data should be compressed if possible. The depth value, d, can be defined in a number of ways. In current graphics hardware APIs, the normalized depth, $d = \frac{z_{clip}}{w_{clip}}$, is used since it is bounded to $[0,1]$, and distributes much of the resolution closer to the viewer. Alternatively, the raw floating-point value, w_{clip}, can be stored. The former representation has the important property that the depth of a triangle can be linearly interpolated in screen space, which is exploited by many depth buffer compression formats. Unfortunately, for moving and defocused triangles, this is no longer true. Therefore, we analyze the mathematical expression for the depth functions in the case of motion blur and depth of field. We show that although the expressions may appear somewhat complicated, they can be effectively simplified, and compact forms for the depth functions can be used to design algorithms with substantially better average compression ratios for stochastic rasterization. Our method only targets motion *or* defocus blur, but for completeness, we also include the derivation for simultaneous motion blur and depth of field in Appendix A.

In general, we assume that the compressors and decompressors exist in a depth system, as described by Hasselgren and Akenine-Möller [50]. Compression/decompression is applied to a *tile*, which typically is the set of depth samples inside a rectangular screen-space region. Due to bus widths, compression algorithms, and tile sizes, only a few different compression ratios, e.g., 25% & 50%, are usually available. Typically, a few bits (e.g., two) are stored on-chip, or in a cache, and these are used to indicate the compression level of a tile, or whether the tile is uncompressed or in a fast clear mode.

2 Previous Work

Previous depth compression research targeting static geometry typically exploits that the depth function, $d(x,y) = \frac{z(x,y)}{w(x,y)}$, is linear in screen space, (x,y), and this can be used to achieve high compression ratios. Morein was the first to describe a depth compression system, and he used a differential-differential pulse code modulation (DDPCM) method for compression [87]. By examining patent data bases on depth compression, Hasselgren and Akenine-Möller presented a survey on a handful of compression algorithms [50].

One of the most successful depth compression algorithms is *plane encoding* [50], where the rasterizer feeds the exact plane equations to the compressor together with a coverage mask indicating which samples/pixels inside a tile that are covered by the triangle. The general idea is simple. When a triangle is rendered to a tile, first check whether there is space in the compressed representation of the tile for another triangle plane equation. If so, we store the plane equation, and update the plane-selection bit mask of the tile to indicate which samples point to the new plane equation. When there is not enough space to fit any more plane equations, we need to decompress the current depths, update with the new incoming depth data, and then store the depth in an uncompressed format. To decompress a tile, just loop over the samples, and look at the plane selection bit mask to obtain the plane equation for the sample, and evaluate that plane equation for the particular, (x,y), of the sample. A compressed tile will be able to store n plane equations together with a plane selection bit mask with $\lceil \log n \rceil$ bits per sample, where n depends on the parameters of the depth compression system, and the desired compression ratio. In this paper, we generalize this method so that it works for motion blur and defocus blur, and we optimize the depth function representations.

Anchor encoding is a method similar to plane encoding. It uses approximate plane equations (derived from the depth data itself, instead of being fed from the rasterizer), and encodes the differences, also called residuals, between each depth value and the predicted depth from the approximate plane equation. *Depth offset* encoding is probably one of the simplest methods. First, the minimum, Z_{min}, and the maximum, Z_{max}, depths of a tile are found. Each sample then uses a bit to signal whether it is encoded relative to the min or the max, and the difference is encoded using as many bits that are left to reach the desired compression ratio.

The first public algorithm for compressing floating-point depth [111] reinterpreted the floats as integers, and used a predictor based on a small set of depths in the tile. The residuals, i.e., the difference between the predicted values and the real depths, were then entropy encoded using Golomb-Rice encoding. A general method for compressing floating-point data was presented by Pool et al. [100]. The differences between a sequence of floating-point numbers is encoded using an entropy encode based on Fibonacci codes.

A compressed depth cache was recently documented [53], and some improvements to depth buffering were described. In particular, when data is sent uncompressed, smaller tile sizes are used compared to when the tiles are compressed. We will also use this feature in our research presented here.

Color buffer compression algorithms [103, 104, 111] are working on different data (color instead of depth), but otherwise, those algorithms operate in a similar system inside the graphics processor as do depth compression.

Gribel et al. [45] perform lossy compression of a time-dependent depth function per pixel. However, this approach requires a unique depth function per pixel, and does not solve the problem of compressing stochastically generated buffers over a tile of pixels. They derive the depth function for a motion blurred triangle and note that when the triangle is moving, the linearity of the depth in screen space is broken. From their work, we know that the depth is a rational cubic function of time, t, for a given sample position (x,y).

Recently, higher-order rasterization, i.e., for motion blur and depth of field, has become a popular research topic. All existing methods for static geometry, except depth offset encoding, break down in a stochastic rasterizer [6, 10]. In the depth compression work by Andersson et al. [10], the time dimension was incorporated in the compression schemes, which led to improved depth buffer compression for stochastic motion blur. By focusing on both motion blur and defocus blur, we solve a much larger problem. In addition, we analyze the depth functions and simplify their representations into very compact forms.

3 Background

In this section, we give some background on barycentric interpolation and show how the depth function, $d = \frac{z}{w}$, is computed for static triangles. Towards the end of this section, we show a generalized version of the depth function without deriving the details. All this information is highly useful for the understanding of the rest of the paper.

Suppose we have a triangle with clip space vertex positions $\mathbf{p}_k = [p_{k_x}, p_{k_y}, p_{k_w}]$, $k \in \{0,1,2\}$. In homogeneous rasterization, the 2D homogeneous (2DH) edge equation, $e_k = \mathbf{n}_k \cdot \mathbf{x}$, corresponds to a distance calculation of an image plane position, $\mathbf{x} = [x, y, 1]^{\mathrm{T}}$, and the edge plane, which passes through the origin, with, for example, $\mathbf{n}_2 = \mathbf{p}_0 \times \mathbf{p}_1$.

Let us introduce an arbitrary per-vertex attribute, A_k, that we wish to interpolate over the triangle. McCool et al. [80] showed that each of the barycentric coordinates, B_0, B_1, B_2, of the triangle can be found by evaluating and normalizing the corresponding 2DH edge equation, such that $B_k = \frac{e_k}{e_0+e_1+e_2}$. The interpolated attribute, A, for a given sample point, $\mathbf{x}$, can then be found by standard barycentric interpolation:

$$A(x,y) = \sum_{k=0}^{2} A_k B_k = \frac{A_0 e_0 + A_1 e_1 + A_2 e_2}{e_0 + e_1 + e_2}. \tag{1}$$

The depth value, d, is formed by interpolating z and w individually, and then performing a division:

$$d(x,y) = \frac{z(x,y)}{w(x,y)} = \frac{\sum z_k B_k}{\sum w_k B_k} = \frac{\sum z_k e_k}{\sum w_k e_k}. \tag{2}$$

If we look at the denominator, we see that:[1]

$$\begin{aligned} \sum_{k=0}^{2} w_k e_k &= \Big(\sum_{k=0}^{2} w_k \, \mathbf{p}_i \times \mathbf{p}_j\Big) \cdot \mathbf{x} \qquad (3) \\ &= [0, 0, \det(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)] \cdot \mathbf{x} = \det(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2), \end{aligned}$$

which is independent of (x,y). This is six times the signed volume of the tetrahedron spanned by the origin and the triangle, which can be used to detect if a triangle is backfacing.

[1]Throughout the paper, we will sum over k, $k \in \{0,1,2\}$ and use the notation $i = (k+1) \bmod 3$ and $j = (k+2) \bmod 3$.

If we use a standard projection matrix, such that the transformation of $(z_{cam}, 1)$ to clip space (z, w) can be expressed as (c.f., the standard Direct3D projection matrix):

$$z = a\, z_{\text{cam}} + b, \quad w = z_{\text{cam}}, \tag{4}$$

then the depth function can be simplified. The coefficients a and b depend solely on z_{near} and z_{far}. Combining Equations 2 and 4 and simplifying gives us:

$$d(x,y) = \frac{z(x,y)}{w(x,y)} = a + \frac{b \sum e_k}{\sum w_k e_k}. \tag{5}$$

We have now derived the 2D depth function, which is widely used in rendering systems today. However, Equation 5 can be augmented so that it holds for depth sampled in higher dimensions. For example, adding motion blur and depth of field means that z, w, and the edge equations are functions of shutter time, t, and lens position, (u, v). Thus, we can write the depth function on a more general form:

$$d(x,y,\dots) = a + \frac{b \sum e_k(x,y,\dots)}{\sum w_k(x,y,\dots) e_k(x,y,\dots)}, \tag{6}$$

where $\dots$ should be replaced with the new, augmented dimensions.

4 Generalized Plane Encoding

In Section 2, we described how the plane encoding method works for static triangle rendering. For higher-order rasterization, including motion blur and defocus blur, static plane equations cannot be used to represent the depth functions, because the depth functions are much more complex in those cases. For motion blur, the depth function is a cubic rational polynomial [45], for example. Therefore, the goal of our work in this paper is to generalize the plane encoding method in order to also handle motion blur and defocus blur.

Our new *generalized plane encoding* (GPE) algorithm is nearly identical to static plane encoding, except that the plane equations for motion blurred and/or defocused plane equations use more storage, and that the depth functions are more expensive to evaluate. This can be seen in Equation 6, which is based on more complicated edge equations, e_k, and w_k-components. However, in Section 5, we will show how the required number of coefficients for specific cases can be substantially reduced, which makes it possible to fit more planes in the compressed representation. This in turn makes for higher compression ratios and faster depth evaluation.

Similar to static plane encoding, the compression representation for generalized depth (motion and defocus blur, for example) includes a variable number of generalized plane equations (Section 5), and a plane selector bitmask per sample. If there are at most n plane equations in the compressed representation, then each sample needs $\lceil \log n \rceil$ bits for the plane selector bitmask. Next, we simplify the depth functions for higher-order rasterization.

5 Generalized Depth Function Derivations

In the following subsections, we will derive compact depth functions for motion blurred and defocused triangles. Some readers may want to skip to the results in Section 7. Since we ultimately could not simplify the combined defocus and motion blur case, we skip that derivation in this section and refer interested readers to Appendix A.

5.1 Motion Blur

We begin the depth function derivation for motion blur by setting up time-dependent attribute interpolation in matrix form. Then, we move on to reducing the number of coefficients needed to exactly represent the interpolated depth of a triangle.

The naïve approach to store the depth functions for a motion blurred triangle is to retain all vertex positions at $t = 0$ and $t = 1$, which are comprised of a total of $4 \times 3 \times 2 = 24$ coordinate values (e.g., floating-point). If the projection matrix is known, and can be stored globally, then only $3 \times 3 \times 2 = 18$ coordinate values are needed, as z then can be derived from w. In the following section, we show how the depth function can be rewritten and simplified to contain only 13 values, which enables more efficient storage.

Time-dependent Barycentric Interpolation In the derivation below, we assume that vertices move linearly in clip space within each frame. Thus, the vertex position, $\mathbf{p}_k$, becomes a function of time:

$$\mathbf{p}_k(t) = \mathbf{q}_k + t\mathbf{d}_k, \tag{7}$$

where $\mathbf{d}_k$ is the corresponding motion vector for vertex k. Akenine-Möller et al. [6] showed that since the vertices depend on time, the 2DH edge equations form 2nd degree polynomials in t:

$$e_k(x,y,t) = (\mathbf{p}_i(t) \times \mathbf{p}_j(t)) \cdot \mathbf{x} = (\mathbf{h}_k + \mathbf{g}_k t + \mathbf{f}_k t^2) \cdot \mathbf{x}, \tag{8}$$

where

$$\mathbf{h}_k = \mathbf{q}_i \times \mathbf{q}_j, \;\; \mathbf{g}_k = \mathbf{q}_i \times \mathbf{d}_j + \mathbf{d}_i \times \mathbf{q}_j, \;\; \mathbf{f}_k = \mathbf{d}_i \times \mathbf{d}_j. \tag{9}$$

For convenience, we rewrite the edge equation in matrix form:

$$e_k(x,y,t) = \mathbf{t}_2 \mathbf{C}_k \mathbf{x}, \;\; \text{where} \;\; \mathbf{C}_k = \begin{bmatrix} h_{k_x} & h_{k_y} & h_{k_w} \\ g_{k_x} & g_{k_y} & g_{k_w} \\ f_{k_x} & f_{k_y} & f_{k_w} \end{bmatrix}, \tag{10}$$

where $\mathbf{t}_2 = (1, t, t^2)$ is a row vector and $\mathbf{x} = (x, y, 1)^{\mathrm{T}}$ is a column vector. By combining the matrix notation and Equation 1, we have a general expression of how to interpolate a vertex attribute, A_k, over a motion blurred triangle:

$$A(x,y,t) = \frac{\mathbf{t}_2(\sum A_k \mathbf{C}_k)\mathbf{x}}{\mathbf{t}_2 \sum \mathbf{C}_k \mathbf{x}}. \tag{11}$$

However, if the attribute itself varies with t, e.g., $A_k(t) = A_k^o + tA_k^d$, we obtain a general expression for interpolating a time-dependent attribute over the triangle, with a numerator of cubic degree:

$$A(x,y,t) = \frac{\mathbf{t}_2 \sum((A_k^o + tA_k^d)\mathbf{C}_k)\mathbf{x}}{\mathbf{t}_2 \sum \mathbf{C}_k \mathbf{x}} = \frac{\mathbf{t}_3 \mathbf{C}_A \mathbf{x}}{\mathbf{t}_2 \sum \mathbf{C}_k \mathbf{x}}, \tag{12}$$

where $\mathbf{t}_3 = (1, t, t^2, t^3)$, and the vertex attributes, A_k, are multiplied with each $\mathbf{C}_k$ and summed to form the 4×3 coefficient matrix $\mathbf{C}_A$.

To compute the depth function $d = \frac{z}{w}$, we perform barycentric interpolation of the z- and w-components of the clip space vertex positions, which are now linear functions of t, e.g., $z(t) = q_z + td_z$ and $w(t) = q_w + td_w$.

Let us consider the depth function, $d(x,y,t)$:

$$d(x,y,t) = \frac{z(x,y,t)}{w(x,y,t)} = \frac{\mathbf{t}_2 \sum((q_{k_z} + td_{k_z})\mathbf{C}_k)\mathbf{x}}{\mathbf{t}_2 \sum((q_{k_w} + td_{k_w})\mathbf{C}_k)\mathbf{x}} = \frac{\mathbf{t}_3 \mathbf{C}_z \mathbf{x}}{\mathbf{t}_3 \mathbf{C}_w \mathbf{x}}, \tag{13}$$

where:

$$\mathbf{C}_z = \sum (q_{k_z} \underbrace{\begin{bmatrix} & & \\ & \mathbf{C}_k & \\ & & \\ 0 & 0 & 0 \end{bmatrix}}_{\overline{\mathbf{C}}_k} + d_{k_z} \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ & & \\ & \mathbf{C}_k & \\ & & \end{bmatrix}}_{\underline{\mathbf{C}}_k}), \tag{14}$$

and $\mathbf{C}_w$ is defined correspondingly. We now have the depth function in a convenient form, but the number of coefficients needed are no less than directly storing the vertex positions. We will now examine the contents of the coefficient matrices, $\mathbf{C}_z$ and $\mathbf{C}_w$, in order to simplify their expressions.

Using Equation 14 and the definition of $\mathbf{C}_k$, we can express the first and last row of $\mathbf{C}_w$ as:

$$\begin{aligned} \mathbf{C}_{w_0} &= \sum q_{k_w} \mathbf{h}_k = \sum q_{k_w} \mathbf{q}_i \times \mathbf{q}_j = [0, 0, \det(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2)], \\ \mathbf{C}_{w_3} &= \sum d_{k_w} \mathbf{f}_k = [0, 0, \det(\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2)], \end{aligned} \tag{15}$$

where, in the last step, the terms cancel out to zero for the x- and y-components. The two remaining rows can be simplified in a similar fashion:

$$\begin{aligned} \mathbf{C}_{w_1} &= \sum (q_{k_w} \mathbf{g}_k + d_{k_w} \mathbf{h}_k) \\ &= \sum (q_{k_w} (\mathbf{d}_i \times \mathbf{q}_j + \mathbf{q}_i \times \mathbf{d}_j) + d_{k_w} (\mathbf{q}_i \times \mathbf{q}_j)) \\ &= (0, 0, \sum \det(\mathbf{d}_k, \mathbf{q}_i, \mathbf{q}_j)), \\ \mathbf{C}_{w_2} &= \sum (q_{k_w} \mathbf{f}_k + d_{k_w} \mathbf{g}_k) = (0, 0, \sum \det(\mathbf{q}_k, \mathbf{d}_i, \mathbf{d}_j)). \end{aligned} \tag{16}$$

Using these expressions, we can formulate $\mathbf{t}_3 \mathbf{C}_w \mathbf{x}$ as a cubic function in t independent of (x,y):

$$\mathbf{t}_3 \mathbf{C}_w \mathbf{x} = \Delta_0 + \Delta_1 t + \Delta_2 t^2 + \Delta_3 t^3, \tag{17}$$

where:

$$\begin{aligned}\Delta_0 &= \det(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2),\\ \Delta_1 &= \sum \det(\mathbf{d}_k, \mathbf{q}_i, \mathbf{q}_j),\\ \Delta_2 &= \sum \det(\mathbf{q}_k, \mathbf{d}_i, \mathbf{d}_j),\\ \Delta_3 &= \det(\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2).\end{aligned}$$

Expressed differently, the denominator $\mathbf{t}_3\mathbf{C}_w\mathbf{x}$ is the backface status for the moving triangle, e.g., $\det(\mathbf{p}_0(t), \mathbf{p}_1(t), \mathbf{p}_2(t))$ [90].

As a result of these simplifications, we reveal that $\mathbf{t}_3\mathbf{C}_w\mathbf{x}$ has no dependency on x and y and is reduced to a cubic polynomial in t, needing only 4 coefficients. Thus, with this analysis, we have shown that the depth function can be represented by 12 (for $\mathbf{C}_z$) $+4$ (for $\mathbf{C}_w$) $= 16$ coefficients, which should be compared to the 24 coefficients needed to store all vertex positions. Our new formulation is substantially more compact.

Further optimization If we use a standard projection matrix, according to Equation 4, we can simplify the depth function further. If we return to Equation 14, and insert the constraint from the projection matrix, i.e., $q_z = aq_w + b$ and $d_z = z_{t_1} - z_{t_0} = a(w_{t_1} - w_{t_0}) = ad_w$, we obtain:

$$\begin{aligned}\mathbf{C}_z &= \sum \left(q_{k_z}\overline{\mathbf{C}}_k + d_{k_z}\underline{\mathbf{C}}_k\right) = \sum \left((aq_{k_w} + b)\overline{\mathbf{C}}_k + ad_{k_w}\underline{\mathbf{C}}_k\right)\\ &= a\mathbf{C}_w + b\sum \overline{\mathbf{C}}_k. \end{aligned} \tag{18}$$

We combine this result with Equation 13 to finally arrive at:

$$\begin{aligned}d(x,y,t) &= \frac{\mathbf{t}_3\mathbf{C}_z\mathbf{x}}{\mathbf{t}_3\mathbf{C}_w\mathbf{x}} = \frac{\mathbf{t}_3(a\mathbf{C}_w + b\sum\overline{\mathbf{C}}_k)\mathbf{x}}{\mathbf{t}_3\mathbf{C}_w\mathbf{x}} = a + b\frac{\mathbf{t}_3(\sum\overline{\mathbf{C}}_k)\mathbf{x}}{\mathbf{t}_3\mathbf{C}_w\mathbf{x}}\\ &= a + b\frac{\mathbf{t}_2(\sum\mathbf{C}_k)\mathbf{x}}{\Delta_0 + \Delta_1 t + \Delta_2 t^2 + \Delta_3 t^3}.\end{aligned} \tag{19}$$

As can be seen above, we have reduced the representation of the depth function from 24 scalar values down to 13 (with the assumption that a and b are given by the graphics API). Later, we will show that this significantly improves the compression ratio for depth functions with motion blur.

Equal Motion Vectors Next, we consider an extra optimization for the special case of linear translation along a vector, since this is a common use case in some applications. In the examples below, we assume that a standard projection matrix is used (i.e., Equation 4). The transformed clip space positions, $\mathbf{p}' = (p'_x, p'_y, p'_w)$, of each triangle vertex are: $\mathbf{p}'_k = \mathbf{q}_k + \mathbf{d}$, where $\mathbf{d} = (d_x, d_y, d_w)$ is a vector in clip space (xyw).

With all motion vectors equal for the three vertices of a triangle, we can derive a simplified depth function. Note that the coefficients $\mathbf{f}_k = 0$, and

$$\begin{aligned}\det(\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2) &= \det(\mathbf{d}, \mathbf{d}, \mathbf{d}) = 0,\\ \det(\mathbf{q}_i, \mathbf{d}_j, \mathbf{d}_k) &= \det(\mathbf{q}_i, \mathbf{d}, \mathbf{d}) = 0.\end{aligned} \tag{20}$$

Furthermore, it holds that:

$$\sum \mathbf{g}_k = \sum \mathbf{d} \times (\mathbf{q}_j - \mathbf{q}_i) = \mathbf{d} \times \sum (\mathbf{q}_j - \mathbf{q}_i) = 0. \tag{21}$$

The depth function can then be simplified as:

$$d(x,y,t) = a + b\frac{\mathbf{x} \cdot \sum \mathbf{h}_k}{\Delta_0 + \Delta_1 t}. \tag{22}$$

We have reduced the representation of the depth function from 18 scalar values down to 5 (again with the assumption that a and b are given by the graphics API).

5.2 Depth of Field

There are not as many opportunities to simplify the depth function for defocus blur as there are for motion blur. If we simply store all vertex positions, then $4 \times 3 = 12$ coordinate values are needed. If, however, the projection matrix is known, the number is reduced to $3 \times 3 = 9$. We assume that the camera focal distance and lens aspect are known globally. In the following section, we will show how to reduce the storage requirement of the depth function to 8 scalar coefficients for a defocused triangle.

When depth of field is enabled, a clip-space vertex position is sheared in xy as a function of the lens coordinates (u,v). The vertex position is expressed as:

$$\mathbf{p}_k = \mathbf{q}_k + c_k \mathbf{u}', \tag{23}$$

where c_k is the signed clip space circle of confusion radius, $\mathbf{u}' = (u, \xi v, 0)$, and ξ is a scalar coefficient that adjusts the lens aspect ratio. Note that c_k is unique for each vertex and is typically a function of the depth. We use these vertices to set up the edge equations:

$$\begin{aligned} e_k(x,y,u,v) &= (\mathbf{p}_i(u,v) \times \mathbf{p}_j(u,v)) \cdot \mathbf{x} \\ &= (\mathbf{q}_i \times \mathbf{q}_j + \mathbf{u}' \times (c_i \mathbf{q}_j - c_j \mathbf{q}_i)) \cdot \mathbf{x} \\ &= (\mathbf{h}_k + \mathbf{u}' \times \mathbf{m}_k) \cdot \mathbf{x}, \end{aligned}$$

where we have introduced $\mathbf{m}_k = (c_i \mathbf{q}_j - c_j \mathbf{q}_i)$ and $\mathbf{h}_k = \mathbf{q}_i \times \mathbf{q}_j$ to simplify notation. With $\mathbf{u} = (u, \xi v, 1)$, we can write the edge equation in matrix form as:

$$e_k(x,y,u,v) = \mathbf{u}\mathbf{C}_k\mathbf{x}, \tag{24}$$

where:

$$\mathbf{C}_k = \begin{bmatrix} 0 & -m_{k_w} & m_{k_y} \\ m_{k_w} & 0 & -m_{k_x} \\ h_{k_x} & h_{k_y} & h_{k_w} \end{bmatrix}. \tag{25}$$

Analogous to the motion blur case, we can express the depth function as a rational function in (x,y,u,v) as follows:

$$d(x,y,u,v) = \frac{z(x,y,u,v)}{w(x,y,u,v)} = \frac{\mathbf{u}\mathbf{C}_z\mathbf{x}}{\mathbf{u}\mathbf{C}_w\mathbf{x}}, \tag{26}$$

where $\mathbf{C}_z = \sum q_{k_z}\mathbf{C}_k$ and $\mathbf{C}_w = \sum q_{k_w}\mathbf{C}_k$. By combining the observation that:

$$\sum q_{k_w} m_{k_w} = \sum q_{k_w}(c_i q_{j_w} - c_j q_{i_w}) = 0, \tag{27}$$

and the top row in Equation 15, $\mathbf{C}_w$ is reduced to a single column, similar to the motion blur case. Thus, the denominator can be written as:

$$\mathbf{u}\mathbf{C}_w\mathbf{x} = \mathbf{u}\begin{bmatrix} 0 & 0 & \sum q_{k_w} m_{k_y} \\ 0 & 0 & -\sum q_{k_w} m_{k_x} \\ 0 & 0 & \det(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2) \end{bmatrix}\mathbf{x} = \Delta_u u + \Delta_v v + \Delta_0. \tag{28}$$

This is equal to $\det(\mathbf{p}_0(u,v), \mathbf{p}_1(u,v), \mathbf{p}_2(u,v))$, which is also the backface status for a defocused triangle [90].

If we introduce the restrictions on the projection matrix from Equation 4, then $\mathbf{C}_z$ can be expressed in the following manner:

$$\mathbf{C}_z = \sum q_{k_z}\mathbf{C}_k = \sum((a q_{k_w} + b)\mathbf{C}_k) = a\mathbf{C}_w + b\sum\mathbf{C}_k. \tag{29}$$

If we further assume that the clip-space circle of confusion radius follows the thin lens model, it can be written as $c_k = \alpha p_{k_w} + \beta$. With this, we see that:

$$\begin{aligned} \sum m_{k_w} &= \sum(c_i q_{j_w} - c_j q_{i_w}) \\ &= \sum((\alpha q_{i_w} + \beta) q_{j_w} - (\alpha q_{j_w} + \beta) q_{i_w}) \\ &= \alpha\sum(q_{i_w} p_{j_w} - q_{j_w} p_{i_w}) + \beta\sum(q_{j_w} - q_{i_w}) = 0, \end{aligned} \tag{30}$$

and $\sum\mathbf{C}_k$ takes the form:

$$\sum\mathbf{C}_k = \begin{bmatrix} 0 & 0 & \sum m_{k_y} \\ 0 & 0 & -\sum m_{k_x} \\ \sum h_{k_x} & \sum h_{k_y} & \sum h_{k_w} \end{bmatrix}. \tag{31}$$

With this, we have shown that:

$$d(x,y,u,v) = \frac{\mathbf{u}\mathbf{C}_z\mathbf{x}}{\mathbf{u}\mathbf{C}_w\mathbf{x}} = a + b\frac{\sum\mathbf{h}_k\cdot\mathbf{x} + \sum m_{k_y} u - \sum m_{k_x}\xi v}{\Delta_u u + \Delta_v v + \Delta_0}, \tag{32}$$

which can be represented with 8 scalar coefficients (given that a and b are known). Note that the denominator is linear in each variable.

6 Implementation

We have implemented all our algorithms in a software rasterizer augmented with a depth system [50] containing depth codecs (compressors and decompressors), a depth cache, culling data, and a tile table, which will be described in detail below. To reduce the design space, we chose a cache line size of 512 bits, i.e., 64 bytes,

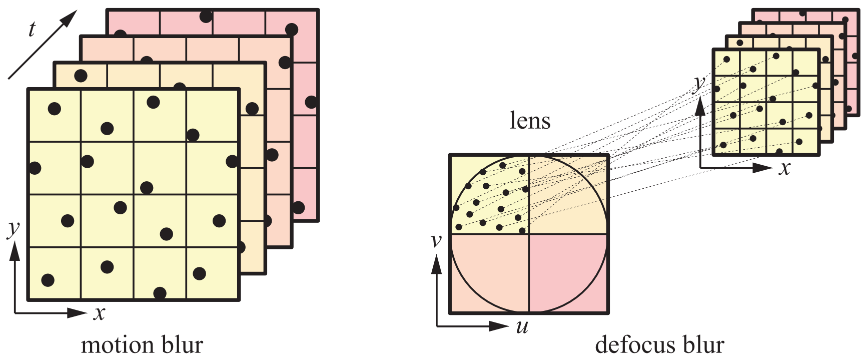


Figure 1: Left: motion blur for 4×4 *pixels where there are four samples per pixel (indicated by the four different layers). In total, there are* $4 \times 4 \times 4$ *samples here. If n layers are used we denote such a tile* $4 \times 4 \times n$*. As an example, if each layer is compressed as a separate tile, then we denote these tiles by* $4 \times 4 \times 1$*. Right: we use the same notation for defocus blur, but with a different meaning. Here, the lens has been divided into* 2×2 *smaller lens regions, and as before, there are four samples per pixel (again indicated by the four layers). However, for defocus blur,* $4 \times 4 \times n$ *means that n lens regions are compressed together as a tile.*

which is a reasonable and realistic size for our purposes. The implication of this choice is that a tile, which is stored using $512 \cdot n$ bits, can be compressed down to $512 \cdot m$ bits, where $1 \leq m < n$ in order to reduce bandwidth usage.

For our results, we present numbers for both 24b integer depth as well as for the complementary depth $(1 - z)$ [69] representation for 32b floating-point buffers. The reason is that this technique has been widely adopted as a superior method on the Xbox 360 (though with 24 bits floating point), since it provides a better distribution of the depths. For all our tests, we use a sample depth cache of 64 kB with least-recently used (LRU) replacement strategy.

Even though motion blur is three-dimensional, and defocus blur uses four dimensions, we are using the same tile notation for both these cases in order to simplify the discussion. An explanation of our notation can be found in Figure 1. We perform Z-max culling [43] per $4 \times 4 \times 1$ tiles for 4 spp and $2 \times 2 \times 4$ for 16 spp, where we store z_{max} of the tile using 15 bits. If all of the samples within the tile are cleared, we flag this with one additional bit. If an incoming triangle passed the Z-max test, the per-sample z-test is executed, and the tile's z_{max} is recomputed if any sample pass the per-sample test. For complementary depth z_{min} is used instead.

The tile table, which is accessed through a small cache or stored in an on-chip memory, stores a tile header for each tile. For simplicity, we let the header store eight bits, where one combination indicates that the tile is stored uncompressed, while the remaining combinations are used to indicate different compression modes. In Section 7, we describe which tile sizes have been used for the different algo-

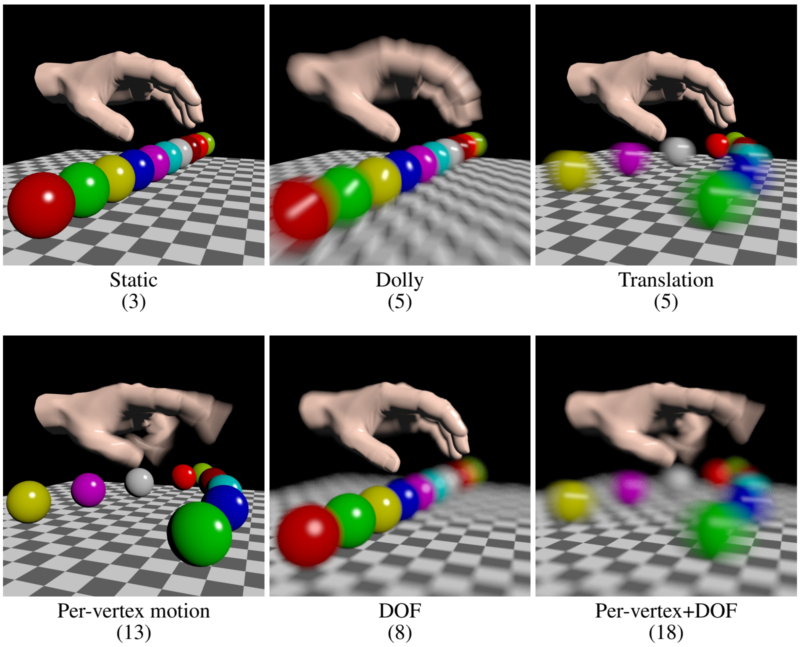


Figure 2: Different configurations of motion blur and depth of field on a simple scene. Below each image, the number of coefficients needed to store the generalized depth function is shown.

rithms. Using a 32kB cache, we have seen that the total memory bandwidth usage for culling and tile headers is about 10% of the total raw depth buffer bandwidth in general, and approximately the same for all algorithms. Note that culling is essential to performance and is orthogonal to depth buffer compression. Therefore, we chose to exclude those numbers from our measurements and instead just focus on the depth data bandwidth.

Our implementation of the generalized plane encoder, denoted GPE, is straightforward. For motion blur, the rasterizer forwards information about the type of motion applied to each triangle. The three different types of motion that we support are static (no motion), only translation, and arbitrary linear per-vertex motion. In each case, the encoder receives a set of coefficients, representing the depth function for the current triangle. In addition, the rasterizer forwards a coverage mask, which indicates which samples are inside the triangle. The depth is evaluated for these samples, and depth testing is performed. A depth function of a previously drawn triangle is removed if all of its sample indices are covered by the incoming triangle's coverage mask. The depth of field encoder works in exactly the same way, except that there are no special types for defocus blur that are forwarded.

As shown in the four leftmost images in Figure 2, the number of coefficients needed per triangle is a function of the motion type and varies per triangle from

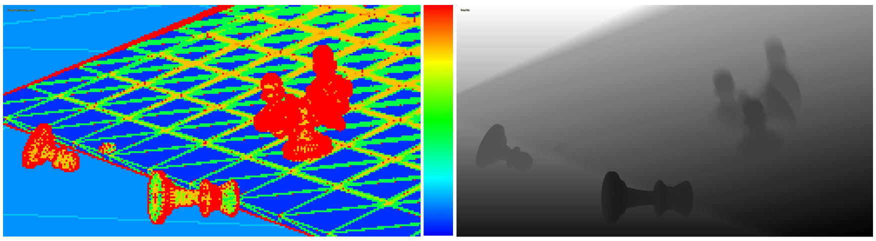


Figure 3: Left: a false-color visualization of the different GPE *modes for the Chess scene. Pure red represents uncompressed regions, and blue the modes with the highest compression ratio (16:1). Right: the stochastic depth buffer for the chess scene.*

3 (static), 5 (translation) to 13 (per-vertex motion). Recall that these reductions in the number of coefficients were derived in Section 5. A compressed tile may include x static plane equations, y depth functions for translated triangles, and z depth functions with arbitrary per-vertex motion. The total storage for the depth function coefficients is then $3x+5y+13z$ floats. Additionally, for each sample, we need to indicate which depth function to use (or if the sample is cleared), which is stored with $\lceil \log_2(x+y+z+1) \rceil$ bits per sample. We work on $16 \times 16 \times 1$ tiles for 4 spp and $8 \times 8 \times 4$ tiles for 16 spp, or 16 cache lines, which can be compressed down to one cache line in the best case. Total storage for motion blur is then $(3x+5y+13z) \times 32 + \lceil \log_2(x+y+z+1) \rceil \times 256$. If this sum is less than 16 cache lines, we can reduce the depth bandwidth usage. To simplify the exposition in this paper, we allow only compression to 1, 2, 3, 4, and 8 cache lines, and only use one set of functions for each tile. It is trivial to extend the compressor to the intermediate compression rates as well. Figure 3 shows an example on the usage of the modes in one of our test scenes.

For defocus blur, the expression is simplified, as the depth function is always stored with 8 coefficients per triangle (see Figure 2). If the number of depth functions in a mode is denoted n, the number of bits per mode is given by $8 \times n \times 32 + 256\lceil \log_2(n+1) \rceil$. The set of modes we have used in this paper is listed in Appendix B.

Note that all coefficients are stored using 32b float, regardless if the depth buffer is 24 or 32 bits. While this precision will not produce the same result as interpolating the depth directly from the vertices, we also would like to note that there is currently no strict rules for how depth should be interpolated in a stochastic rasterizer. If we use the same (compressed) representation in the codec and for depth interpolation, we are guaranteed that the compressed data is lossless. Using the same representation for compression and interpolation makes the algorithms consistent, which is perhaps the most important property. However there is still a question if the interpolation is stable and accurate enough to represent the depth range of the scene. Unfortunately we have not done any formal analysis, and have to defer that to future work.

In absence of any compression/decompression units, it makes more sense to use a tile size that fits into a single cache line [53]. Therefore, we allow keeping raw data in cache line sized tiles if the compression unit was unable to compress data beyond the raw storage requirement. However, for compressed tiles, we only allow memory transactions from and to the cache of the entire tile. Our baseline, denoted RAW, simply uses uncompressed data on cache line size tiles, which is a much more efficient baseline than previously used [10, 53] (where the RAW represents uncompressed on the same tile size as the compressed tiles). Since the baseline is more efficient, it means our results are even more significant compared to previous work.

7 Results

In this section, we first describe the set of codecs (compressor and decompressor) we use in the comparison study and then report results on a set of representative test scenes.

Codecs We denote the uncompressed method as RAW below. Note that the RAW mode include Z-max culling and a clear bit per tile, as described in Section 6. An uncompressed $4 \times 4 \times 1$ (4 spp) or $2 \times 2 \times 4$ (16 spp) tile occupies one cache line, i.e., $16 \times 32 = 512$ bits. Our method is denoted GPE, and a detailed implementation description can be found in Section 6.

We compare against a depth offset (DO) codec, working on $8 \times 8 \times 1$ tiles for 4 spp, and $4 \times 4 \times 4$ tiles for 16 spp, where the min and max values are stored in full precision and a per-sample bit indicates if the sample should be delta-encoded w.r.t. the min or the max value. We use three different allocations of the delta bits per sample: 6, 14, and 22. With these layouts, we can compress the four cache lines of depth samples down to one (4:1), two (4:2), and three (4:3) cache lines, respectively. The two bits needed to select one of the three modes or if the sample is cleared are stored in the tile header.

By including time, t, in the predictor functions for a plane encoder, better compression ratios could be achieved for motion blur rasterization [10]. This technique analyzes all samples in the tile and fits a low-order polynomial surface to the depth samples and encode each sample with an offset from this surface. We include this encoder (denoted AHAM11) in our results and refer to the cited paper for a detailed description of this compression format. We use the same tile sizes as for the DO compressor. Note that unlike GPE, the AHAM11 encoder does not rely on coefficients from the rasterizer, but works directly on sample data. The drawback, however, is that the derivation of the approximate surface and subsequent delta computations are significantly more expensive than directly storing the exact generalized depth function. AHAM11 cannot handle defocus blur.

In a *post-cache* codec the compressor/decompressor is located between the cache and the higher memory levels. In a *pre-cache* codec the compressor/decompressor is located between the Z-unit and the cache, which means that data can be stored in compressed format in the cache, at the cost that the compressor/decompressor is invoked more often. Note that AHAM11 is a post-cache codec, while DO is a pre-cache codec[2]. GPE is a pre-cache codec as well, similar to plane encoding for static triangles. For a more detailed overview of pre- vs post-cache codecs, we refer to the paper by Hasselgren et al. [53].

Test Scenes The **Chess** scene contains a mix of highly tessellated geometry (chess pieces) and a coarse base plane. All objects are animated with rigid body motion. The DOF chess scene (32k triangles) has more pieces than the motion blur chess scene (26k triangles). The **Airship** scene (157k triangles) is taken from the Unigine Heaven 2 DX11 demo (on normal tessellation setting), and has been enhanced with a moving camera. This is the only scene tested which uses the depth functions optimized for camera translation. **Dragon** (162k triangles) shows an animated dragon with skinning and a rotating camera. **Sponza** is the CryTek Sponza scene with a camera rotating around the view vector. The scene has 103k triangles for motion blur and 99k triangles for DOF. Finally, the **Hand** scene (15k triangles) is a key-frame animation of a hand with complex motion. All triangle counts are reported after view frustum culling. Furthermore, in **Airship** and **Dragon**, the triangle counts are after backface culling. All scenes are rendered at 1080p.

The results for motion blur can be seen in Table 1, where the resulting bandwidth for each algorithm is given relative to the RAW baseline. While the numbers reveal substantial savings with our algorithm, it is also interesting to treat the previously best algorithm (which is AHAM11 for most scenes) as the baseline, and see what the improvement is compared to that algorithm. For example, for a complementary depth floating point buffer (F32) at four samples per pixel (spp), we see that the relative bandwidth on the Chess scene is $37/64 \approx 58\%$, which is a large improvement. For Airship, this number is 64%, while it is about 77-80% for Dragon and Sponza. The Hand scene is extremely difficult since it only has per-vertex motion (most expensive) on a densely tessellated mesh, and GPE is unable to compress it further. For 16 spp F32, the corresponding numbers are (from Chess to Hand): 64%, 60%, 80%, 68%, and 89%.

For defocus blur, the results can be found in Table 2. The results are even more encouraging. The relative bandwidth, computed as described above for motion blur, compared to the best algorithm (DO) is (from Chess to Hand): 44%, 74%, 70%, 49%, and 88% for 4 spp. For 16 spp, the corresponding numbers are: 35%, 67%, 66%, 34%, and 67%.

Encoder Complexity Analysis Here, we attempt to do a rough comparison of the complexity of the encoder of GPE vs AHAM11, where we assume that there are n samples per tile. AHAM11 starts by finding min and max of the depths, which

[2]DO can be either pre- or post-cache codec, but we use pre-cache since it gives better results [53].

MB		Chess		Airship		Dragon		Sponza		Hand	
		4 spp	16 spp	4 spp	16 spp	4 spp	16 spp	4 spp	16 spp	4 spp	16 spp
RAW (MB)		59.7	256	84.9	355	80.4	366	155	721	36.1	152
DO	F32	84%	75%	100%	95%	92%	89%	93%	88%	100%	98%
	U24	54%	56%	68%	64%	59%	56%	62%	58%	99%	95%
AHAM11	F32	64%	58%	96%	84%	87%	83%	87%	81%	102%	99%
	U24	52%	44%	73%	62%	62%	60%	65%	59%	97%	95%
GPE		37%	37%	62%	50%	70%	66%	67%	55%	100%	88%

Table 1: Motion blur depth buffer memory bandwidth for both 4 samples per pixel (spp) and 16 spp compared to the baseline, RAW. *For the comparison methods, we show results for both 32b float, stored as* $1 - \text{depth}$ *(*F32*) and 24b unsigned int (*U24*) depth buffers. By design, the* GPE *scores are identical for 32b float and 24b unsigned int buffers. We disabled the 4:3 compression mode for* DO *float in the hand scene, because otherwise the bandwidth usage rose to* 107% *and* 103%*.* DO *did, however, benefit from the 4:3 mode in all of our other test scenes. As can be seen, our method (*GPE*) provides substantial bandwidth savings compared to the previous methods in most cases.*

DOF		Chess		Airship		Dragon		Sponza		Hand	
		4 spp	16 spp	4 spp	16 spp	4 spp	16 spp	4 spp	16 spp	4 spp	16 spp
RAW (MB)		102	460	118	413	85.1	413	116	486	40.1	171
DO	F32	88%	86%	100%	99%	93%	92%	86%	82%	99%	95%
	U24	64%	62%	73%	72%	62%	58%	59%	55%	91%	81%
GPE		39%	30%	74%	66%	65%	61%	42%	28%	87%	64%

Table 2: Depth of field results. Notice that AHAM11 *is not included here, since it cannot handle defocus blur. Similar to motion blur, we disabled the 4:3 mode for* DO *for 4 spp* F32 *for the hand scene, because otherwise it used* 101% *of the* RAW *bandwidth. Note again that our method (*GPE*) provides significant savings in most test cases. For the comparison method, we show results for both 32b float, stored as* $1 - \text{depth}$ *(*F32*) and 24b unsigned int (*U24*) depth buffers. By design, the* GPE *scores are identical for 32b float and 24b unsigned int buffers.*

results in $\approx 2n$ operations. Each depth value is then binned (n ops), and the largest gap in the bins is found, where the last step is excluded in our complexity estimate, since it is hard to estimate. For the whole set of samples, and for each of the bins, a bounding box in x and y is found ($4n$ ops), and the box is split into 2×2 regions, or $2 \times 2 \times 2$ regions (depending on which mode is used). In each region, the min and the max depth is found ($\approx n$ ops). For each of the two bins and the whole set of samples, the three modes in AHAM11 uses Cramer's rule to compute the predictor function. We estimate this to about 25 FMA (fused multiply-add) operations. The residuals are found by evaluating the predictor and computing the difference. For the three modes, the predictor evaluation costs $4n$, $4n$, and $9n$ ops respectively (including residual computation). Since each sample belongs to the whole set, as well as to one of the two bins, the steps after binning are performed twice per sample. An under-conservative estimation of AHAM11 is then $n(2+1)+2n(4+1+4+4+9) = 47n$ ops plus $9 \cdot 25 = 225$ ops for Cramer's rule, i..e, a total of $47n+225$ ops. GPE computes the coefficients, which for the most expensive case (per-vertex motion) costs about 130 FMA ops, and then updates the selection masks, which we estimate to be $5n$ to $13n$ operations, depending on which depth function is used. Since $47n+225 \gg 13n+130$ (for $n = 8 \times 8$ samples), we conclude that our encoder is more efficient. In fact, for reasonable tile sizes, the constant factors are insignificant, which means that AHAM11 approximately uses between 4 and 9 times more operations. Furthermore, if the stochastic rasterizer performs a backface test, most of the computations needed for the depth function coefficients can be shared. In that scenario, we estimate the constant factor for GPE to be only 20 ops.

8 Conclusions and Future Work

We have presented a generalized plane encoding (GPE) method, where we optimized the depth function representation significantly for motion blur and depth of field separately. GPE provides substantial depth buffer bandwidth savings compared to all previous methods. Our research can have high impact, since we believe that it gets us a bit closer to having a fixed-function stochastic rasterizer in a graphics processor with depth buffer compression.

At this point, we have not been able to provide any positive results for the combination of motion blur and DOF. In future work, we would like to use the theory developed in Appendix A to design more efficient predictors. Although we concluded that the generalized depth function for the case of simultaneous motion blur and depth of field is too expensive in practice, we could analyze the size of each coefficient for a large collection of scenes, and obtain a lower order approximation. As a final step, the residuals would be encoded. Yet another avenue for future research is to explore the depth function for motion blur with non-linear vertex paths.

Acknowledgements

The authors thank Aaron Coday, Tom Piazza, Charles Lingle, and Aaron Lefohn for supporting this research. The Advanced Rendering Technology team at Intel provided valuable feedback. Thanks to Denis Shergin from Unigine for letting us use images from Heaven 2.0. Tomas Akenine-MÃűller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation.

Appendix A - Motion Blur + Depth of Field

Following the derivation for motion blur and depth of field, we want to create general depth functions for the case of triangles undergoing simultaneous motion blur and depth of field. We first consider the 5D edge equation [91]:

$$e_k(x,y,u,v,t) \quad = \quad (\mathbf{n}_k(t) + \mathbf{u}' \times \mathbf{m}_k(t)) \cdot \mathbf{x}, \tag{33}$$

where $\mathbf{u}' = (u, \xi v, 0)$, $\mathbf{n}_k(t) = \mathbf{p}_i(t) \times \mathbf{p}_j(t)$, and $\mathbf{m}_k(t) = c_i(t)\mathbf{p}_j(t) - c_j(t)\mathbf{p}_i(t)$. The interpolation formula becomes:

$$A(x,y,u,v,t) = \frac{\sum A_k e_k(x,y,u,v,t)}{\sum e_k(x,y,u,v,t)}. \tag{34}$$

Our first goal is to derive a compact formulation of the denominator in Equation 6:

$$\begin{aligned} & \sum p_{k_w}(t) e_k(x,y,u,v,t) \\ = \quad & \sum p_{k_w}(t)\mathbf{n}_k(t) \cdot \mathbf{x} + \sum (p_{k_w}(t)\mathbf{u}' \times \mathbf{m}_k(t)) \cdot \mathbf{x}. \end{aligned} \tag{35}$$

From Equation 17, we have: $\sum p_{k_w}(t)\mathbf{n}_k(t) \cdot \mathbf{x} = \sum_0^3 \Delta_i t^i$. Similarly, by generalizing Equation 27, we obtain:

$$\sum p_{k_w}(t) m_{k_w}(t) = 0, \tag{36}$$

which can be used to simplify the term below:

$$\begin{aligned} & \sum (p_{k_w}(t)\mathbf{u}' \times \mathbf{m}_k(t)) \cdot \mathbf{x} \\ = \quad & \left(0, 0, u \sum p_{k_w}(t) m_{k_y}(t) - \xi v \sum p_{k_w}(t) m_{k_x}(t)\right) \cdot \mathbf{x} \\ = \quad & u \sum_0^3 \gamma_i t^i + \xi v \sum_0^3 \delta_i t^i. \end{aligned} \tag{37}$$

Simplification for the thin lens model If we assume that the clip space circle of confusion radius follows the thin lens model, it can be written as $c_k(t) = \alpha p_{w_k}(t) + \beta$. We use the equality:

$$\sum (p_{k_w}(t)\mathbf{u}' \times \mathbf{m}_k(t)) \cdot \mathbf{x} = \mathbf{u}' \cdot \sum c_k(t)\mathbf{p}_i(t) \times \mathbf{p}_j(t), \tag{38}$$

and see that:

$$\begin{aligned} & \mathbf{u}' \cdot \sum c_k(t)\mathbf{p}_i(t) \times \mathbf{p}_j(t) \\ = \quad & \mathbf{u}' \cdot \sum ((\alpha p_{k_w}(t) + \beta)\mathbf{p}_i(t) \times \mathbf{p}_j(t)) \\ = \quad & \beta \mathbf{u}' \cdot \sum (\mathbf{p}_i(t) \times \mathbf{p}_j(t)) = u \sum_0^2 \gamma_i t^i + \xi v \sum_0^2 \delta_i t^i. \end{aligned} \tag{39}$$

We have shown that the denominator can be expressed as:

$$\sum p_{k_w}(t) e_k(x,y,u,v,t) = u \sum_0^2 \gamma_i t^i + \xi \; v \sum_0^2 \delta_i t^i + \sum_0^3 \Delta_i t^i, \tag{40}$$

Mode	x	y	z	c	cache lines
0	29	0	0	1	8
1	0	17	0	1	8
2	0	0	8	0	8
3	0	0	7	1	8
4	10	0	0	1	4
5	0	8	0	0	4
6	0	7	0	1	4
7	0	0	3	1	4
8	8	0	0	0	3
9	7	0	0	1	3
10	0	4	0	1	3
11	0	0	2	1	3
12	4	0	0	0	2
13	0	3	0	1	2
14	0	0	1	1	2
15	2	0	0	0	1
16	0	1	0	1	1
17	0	0	1	0	1

Table 3: GPE compression modes for motion blur.

which can be represented by 10 coefficients. The numerator:

$$\sum e(x,y,u,v,t) = \left(\sum \mathbf{n}_k(t) + \mathbf{u}' \times \sum \mathbf{m}_k(t)\right) \cdot \mathbf{x}, \tag{41}$$

can be represented by 18 coefficients, but again, if we assume that the clip space circle of confusion radius follows the thin lens model, we can generalize Equation 31 and see that $\sum m_{k_w}(t) = 0$. Then we obtain $(\mathbf{u}' \times \sum \mathbf{m}_k(t)) \cdot \mathbf{x} = u \sum m_{k_y}(t) - \xi v \sum m_{k_x}(t) = u \sum_0^2 \lambda_i t^i + \xi v \sum_0^2 \kappa_i t^i$.

Thus, we can represent the depth function, $d = z/w$, with 25 coefficients. Note that simply storing the vertices, (x,y,w), would require $3 \times 3 \times 2 = 18$ values, which is a more compact representation. We conclude that for the combination of motion and defocus blur, the raw vertex representation is a better alternative in term of storage. Our derivation was still included in order to help others avoid going down this trail of simplifying the equations.

Appendix B - Compression modes for GPE

The total storage cost of a compressed block is: $(3x+5y+13z) \times 32$ bits for the depth function coefficients plus $\lceil \log_2(x+y+z+1) \rceil \times 256$ bits to indicate which depth function to use for each of the 256 samples (or if the sample is cleared). As an additional optimization, if no samples are cleared, we skip the clear bit in some modes. If a clear bit is present in the mode, this is indicated as $c = 1$ in Table 3. Similarly, we show the modes for defocus blur in Table 4. We empirically found a reasonable subset of the large search space of possible predictor combinations that worked well in our test scenes.

Mode	number of planes	c	cache lines
0	12	1	8
1	5	1	4
2	4	0	3
3	3	1	3
4	2	1	2
5	1	1	1

Table 4: GPE compression modes for defocus blur.

Addendum

After the publication of this paper we were able to further simplify the depth function for depth of field (Equation 32).

With the thin lens model, the CoC radius is defined as $c_k = \alpha q_{w_k} + \beta$. First, we note that the difference of two such radii can be written as

$$c_k - c_i = \alpha q_{w_k} + \beta - \alpha q_{w_i} + \beta = \alpha(q_{w_k} - q_{w_i}).$$

In Equation 25 note that $m_{k_y} = c_i q_{j_y} - c_j q_{i_y}$. Expanding $\sum m_{y_k}$ yields

$$\begin{aligned}\sum m_{y_k} &= q_{y_0}(c_1 - c_2) + q_{y_1}(c_2 - c_0) + q_{y_2}(c_0 - c_1)\\ &= \alpha\left(q_{y_0}(q_{w_1} - q_{w_2}) + q_{y_1}(q_{w_2} - q_{w_0}) + q_{y_2}(q_{w_0} - q_{w_1})\right). \end{aligned} \tag{42}$$

Next, we expand $\sum h_{x_k}$, such that

$$\sum h_{x_k} = q_{y_0}(q_{w_1} - q_{w_2}) + q_{y_1}(q_{w_2} - q_{w_0}) + q_{y_2}(q_{w_0} - q_{w_1}). \tag{43}$$

Comparing Equations 42 and 43 it is easy to see that

$$\alpha \sum h_{x_k} = \sum m_{y_k}$$

Similarly, $\alpha \sum h_{y_k} = \sum m_{x_k}$. With α globally known, storing $\sum m_{x_k}$ and $\sum m_{y_k}$ is superfluous, and the final expression can instead be written as

$$d(x,y,u,v) = \frac{\mathbf{u}\mathbf{C}_z\mathbf{x}}{\mathbf{u}\mathbf{C}_w\mathbf{x}} = a + b\frac{\sum \mathbf{h}_k \cdot \mathbf{x} + \alpha\left(\sum h_{k_y} u - \sum h_{k_x} \xi v\right)}{\Delta_u u + \Delta_v v + \Delta_0},$$

which amounts to 6 coefficients.

Paper IV

Masked Depth Culling for Graphics Hardware

Magnus Andersson   Jon Hasselgren   Tomas Akenine-Möller

Lund University   Intel Corporation

ABSTRACT

Hierarchical depth culling is an important optimization, which is present in all modern high performance graphics processors. We present a novel culling algorithm based on a layered depth representation, with a per-sample mask indicating which layer each sample belongs to. Our algorithm is feed forward in nature in contrast to previous work, which rely on a delayed feedback loop. It is simple to implement and has fewer constraints than competing algorithms, which makes it easier to load-balance a hardware architecture. Compared to previous work our algorithm performs very well, and it will often reach over 90% of the efficiency of an optimal culling oracle. Furthermore, we can reduce bandwidth by up to 16% by compressing the hierarchical depth buffer.

to appear in *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 2015.

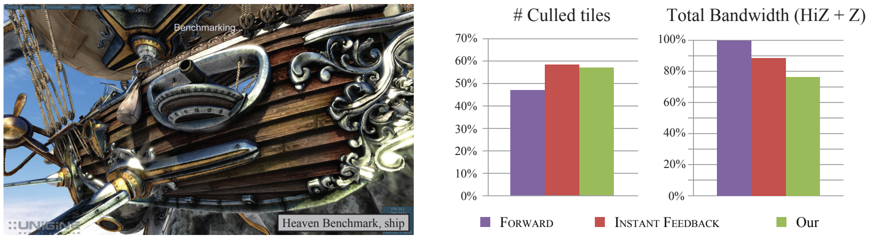


Figure 1: Best case culling performance and bandwidth of our algorithm as compared to previous work. Our algorithm performs similarly to an idealized version of the more complicated feedback algorithm, while keeping the simplicity of the much less efficient forward culling approach. By compressing the depth representation, we show that we achieve significantly less bandwidth than competing algorithms.

1 Introduction

Over 400 million graphics processors were sold in the notebook and desktop segments in 2014. In each of these GPUs, there is a highly optimized fixed-function hierarchical depth culling unit that uses occlusion culling techniques on a per-tile basis [43, 87]. Substantial engineering efforts have been spent fine-tuning such units, in order to minimize memory traffic to the depth buffer, which, in turn, improves performance and/or reduces power. Occlusion culling is integrated transparently in GPUs, i.e., most users enjoy its benefits without ever knowing it is there. The large number of units shipped each year and the performance/power benefits that comes with hierarchical depth culling makes it very important not only to increase efficiency, but also to make the implementation simpler and more robust to different use cases.

We present a novel culling algorithm that uses a layered depth representation with a selection mask that associates each sample to a layer. In our algorithm, culling and updating the representation is very inexpensive and simple, and unlike previous methods we compute accurate depth bounds without requiring an expensive feedback loop [51]. Furthermore, we have greater freedom in choosing tile size since we do not require scanning the depth values of all samples in the backend. This, in turn, makes it easier to load-balance the graphics pipeline. See Figure 1 for an example of the culling potential of our algorithm.

2 Previous Work

Greene et al. presented a culling system based on a complete depth pyramid, with conservative z_{max}-values at each level [43]. However, while highly influential, it is not practical to keep the entire pyramid of depths updated at all times. Morein [87] had a more practical approach, where the maximum depth, z_{max}, was stored and computed per tile. If the conservatively estimated minimum depth of a triangle inside a tile is greater than the tile's z_{max}, then the portion of the triangle overlapping the tile can be culled. In addition, it is also possible to store the minimum tile depth, z_{min}, which is used to avoid depth reads. If the triangle's conservatively estimated maximum depth is smaller than z_{min} [7], the triangle can trivially overwrite the tile (assuming no alpha/stencil test etc), and the read operation can be skipped. From the literature [51, 87], we deduce that z_{max} is typically computed from the per-sample depths in a tile, and must be passed to the hierarchical depth test using a feedback loop. Ideally, the z_{max}-value of a tile should be recomputed and updated every time the sample with the maximum depth value is overwritten, but updates are typically less frequent in order to reduce computations. For example, z_{max} may be recomputed when a tile is evicted from the depth cache.

Occlusion queries count the number of fragments passing the depth test and can be used to cull entire objects [16, 48, 79, 110] using simple proxy geometry, such as a bounding box. A system for dynamic occlusion culling has been presented by Aila and Miettinen [1], and in the gaming industry, it has proven useful to base occlusion queries on software rasterization to better load balance the CPU and GPU [25]. Zhang et al. [124] proposed using hierarchical occlusion maps for occlusion queries. Rather than basing the queries on the depth buffer, they use a full resolution, hierarchical coverage map, and store depth separately in a low resolution depth estimation buffer.

Our algorithm uses occluder merging inspired by the work of Jouppi and Chang [61]. They propose an algorithm for low-cost anti-aliasing and transparency by storing low-precision depth plane equations. A fixed number of planes are stored per pixel and overflow is handled using a merge heuristic. Similarly, Beaudoin and Poulin [15] extend MSAA [3] to use a hierarchical indexing structure to reference a small set of color and depth values per tile. To handle layer overflow they opted to reduce the sampling rate, rather than using a lossy merge heuristic.

Greene and Kass extended their earlier work [43] to include anti-aliasing with error bounds using interval arithmetic for the shaders and quad tree subdivision for visibility handling [42]. Furthermore, Greene et al. [41] used a BSP tree to hierarchically traverse the scene and the screen space using a pyramid of coverage masks for efficient anti-aliasing. In order to save pixel shading work for small triangles, Fatahalian et al. [36] gather and merge quad-fragments from adjacent triangles, using an aggregate coverage mask. The purpose of their mask differs from ours in that they use it to avoid merging shading over geometric discontinuities, while our masked representation is a lossy, but conservative, approximation of the depth buffer.

3 Overview of Current Architectures

In order to contextualize our algorithm, we first describe a typical implementation of a GPU depth pipeline as presented by previous work [51]. The top row of

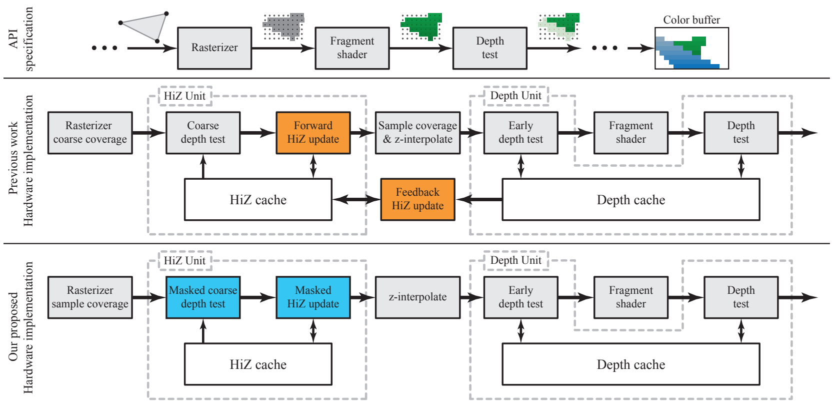


Figure 2: Top: *a simplified overview of the rasterizer, fragment shader, and depth units according the OpenGL and DirectX API specifications.* Middle: *a hardware architecture, according to previous work, featuring a HiZ culling unit and an early depth test. The purpose of these additional units is to improve performance through early occlusion culling. They are transparent to the programmer and can be implemented without changing the API.* Bottom: *our proposed architecture. Our novel HiZ culling algorithm completely removes the need for feedback HiZ updates by using a layered depth representation with a per-sample selection mask, which is efficient for culling and is easy to update.*

Figure 2 depicts the pipeline from a functional standpoint, as specified in most modern graphics APIs. Depth and color buffers are progressively updated as a sequence of triangles goes through the rendering pipeline. For each sample, the depth of the closest triangle is stored along with its color. The actual hardware pipeline typically differs from the API specifications for performance reasons, and a common implementation can be seen in the middle row of Figure 2. In the following, we will limit the discussion to a *less than* depth function to simplify the description, but the techniques generalize to all types of depth functions used in popular API's, such as DirectX, OpenGL, and presumably also in the coming Vulkan API.

Rasterizer We skip the geometry processing part of the graphics processor, and begin our discussion at the rasterizer unit. The rasterizer is responsible for determining which samples overlap a particular triangle. As an optimization, modern rasterizers typically work on *tiles*, which are groups of $w \times h \times d$ samples, where d is the number of samples per pixel. A conservative test is performed for each tile to determine if it is fully covered, is entirely outside the triangle, or partially overlap it. Per-sample coverage testing is only required for tiles partially overlapping the triangle. Once the sample coverage is computed, each fragment is shaded using the fragment shader, followed by the depth test which determines visibility. As can be seen in the middle row of Figure 2, a common optimization is to place the per-sample coverage test after the hierarchical z or *HiZ* unit. The rationale is that the HiZ may remove or cull tiles before the per-sample coverage test occurs, which improves the performance of that unit.

HiZ Unit Depth testing may consume significant memory bandwidth and compute power [2]. For this reason, the hardware pipeline typically has a HiZ unit with the purpose of quickly discarding (culling) or accepting tiles using a *coarse depth test*, whenever the outcome of the depth test can be unambiguously determined for the entire group of samples. For this purpose, the HiZ unit maintains a conservative version of the depth buffer, referred to as the *coarse depth buffer*, which contains per-tile depth bounds, $[z^{tile}_{min}, z^{tile}_{max}]$.

Coarse depth test When a tile reaches the HiZ unit, the first step is to compute conservative bounds, $[z^{tri}_{min}, z^{tri}_{max}]$, of the depth of the incoming triangle within the tile [5]. These bounds are then tested against the coarse depth buffer using interval overlap tests. For example, for a *less than* depth function, we can conclude that the per-sample depth test will fail for all samples if $z^{tri}_{min} \geq z^{tile}_{max}$ and pass if $z^{tri}_{max} < z^{tile}_{min}$. This leaves an ambiguous depth range, where the outcome of the per-sample depth test cannot be determined. Thus, the coarse depth test has one of three outcomes, namely, *fail*, *pass*, or *ambiguous*. Failed, i.e., *culled*, tiles are immediately thrown away and require no further processing. Passing and ambiguous tiles are sent down the pipeline for further processing, with the main difference being that ambiguous tiles must be fully depth tested in the depth unit, while trivially passing tiles can simply overwrite the contents of the depth buffer. This is a small difference, but depending on the architecture, write-only operations may result in lower bandwidth than the read-modify-write operation required for performing the full depth test [7].

Coarse depth buffer update As rendering progresses, the coarse depth buffer is continually updated. From previous work [87, 7], we note that z^{tile}_{min} and z^{tile}_{max} are updated separately in the pipeline using two different mechanisms, namely, a *forward* update located immediately after the coarse depth test and a *feedback* update located between the depth unit and the HiZ unit. This is illustrated in the middle row in Figure 2.

In the forward update stage, z^{tile}_{min} can be efficiently computed as $z^{tile}_{min} = \min(z^{tri}_{min}, z^{tile}_{min})$. The z^{tile}_{max}-value, however, can only be updated if *all* the samples in the tile are overwritten. If the coverage mask is fully set, then $z^{tile}_{max} = \min(z^{tri}_{max}, z^{tile}_{max})$. Unfortunately, this update scales very poorly when using smaller triangles since they are less likely to completely overlap a tile. An example of the irregular coverage resulting from solely using the forward stage can be viewed in Figure 9.

A better z^{tile}_{max} value is obtained using a feedback update. Here, a max-reduction on an entire tile of depth samples is performed in the depth unit and the result, $z^{feedback}_{max}$, is sent to the HiZ unit through the feedback mechanism, as depicted in

Figure 2. To reduce the number of max-reductions performed, the feedback update typically occurs each time a tile is evicted from the depth cache. The feedback mechanism introduces a large delay, as the HiZ and depth units may be separated by hundreds of cycles in the hardware pipeline. Depending on the render state, we may also need to wait for the fragment shader to be executed and for the tile to be evicted from the cache before the update can occur. As a consequence, z_{max}^{tile} updates may lag behind, leading to decreased culling rates. Furthermore, the delay has non-obvious side effects, such as how to conservatively handle cases where the feedback message originated from a different GPU-state than is currently active.

Depth Unit Similar to the HiZ unit, the depth unit typically works on tiles of samples, but instead of performing a single test, each sample is individually tested against the value stored in the depth buffer. The size of a tile in the depth unit is typically correlated to the size of a cache line, which in turn is determined by how much data can be efficiently streamed to and from memory. It should be noted that the feedback mechanism creates a constraint between the tile sizes of the HiZ and depth unit. The max-reduction operation needs to be performed on the granularity of the tiles in the HiZ buffer. Therefore, it is important that all data required for the operation resides in the depth unit cache, and the easiest way to guarantee this is to couple the tile sizes of the coarse depth buffer and the regular depth buffer.

The most straightforward optimization in the depth unit is the *early depth test*, which takes advantage of that depth testing can be performed before fragment shading in many cases. This typically improves performance significantly as fragment shading is expensive and often a bottleneck. For the most part, it is safe to use the early depth test, but it must be disabled when, for example, the fragment shader alters the coverage through a `discard` operation, outputs a depth value, or writes to an unordered access view (UAV) resource.

4 Algorithm

It is challenging to accurately update the maximum depth of the tile without relying on the feedback mechanism. The key innovation in our algorithm is an efficient and accurate way to update the coarse depth buffer using *only* forward updates, which completely removes the need for the feedback mechanism, as illustrated in the bottom row of Figure 2. The updates are performed progressively in a streaming fashion and only use information about the current triangle. No buffering of triangles or rendering history is required.

Without loss of generality, we limit the description of our algorithm to only two depth layers[1] per tile. Each tile has one z_{min} value and two z_{max}^{i} values. In addition, we store a *selection mask* of one bit per sample, which associates each sample with

[1]All depth layers are disjoint and jointly cover the entire tile. This is not to be confused with layered depth images.

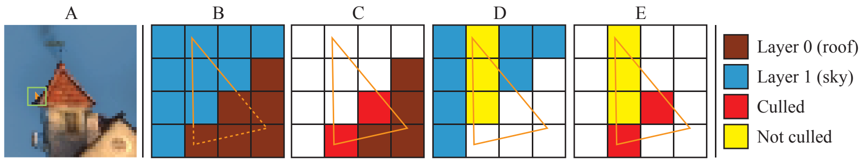


Figure 3: A step by step illustration of how the fail-mask is calculated in our coarse depth test. A: The orange triangle is rasterized and forwarded to the HiZ unit. B: The coarse depth buffer already contains a tile with two layers, namely, the roof in the foreground (brown) and the sky background (blue). As indicated by the dashed lines, the triangle is occluded by the roof, while visible in front of the sky. C: The triangle is overlap tested against the depth of the roof layer and since $z^{tri}_{min} > z^{0}_{max}$, the result is the fail-mask indicated by the two red pixels. D: The triangle is overlap tested against the sky, but cannot be culled. Thus, the fail-mask for this layer is actually empty, but we indicate the three ambiguous pixels for clarity. E: The aggregate fail-mask. We cannot cull the triangle since the fail-mask does not contain all pixels covered by the triangle. However, it would be possible to skip the depth test for the two red pixels.

one of the two layers, i. We keep the z_{min} update strategy described Section 3, as it is simple and efficient. Each z^{i}_{max} must be greater or equal to all samples associated with that layer. We achieve this using a conservative merge of the incoming triangle and the layered representation. In the following, we describe the coarse depth test and update in detail.

Coarse depth test As described in Section 3, the triangle and its coverage mask is provided by the rasterizer, which enables us to compute z^{tri}_{min} and z^{tri}_{max} as before. Similar to how the coarse depth tests are performed for $z_{\text{min}}/z_{\text{max}}$-culling, we do interval overlap tests between $[z^{tri}_{min}, z^{tri}_{max}]$ and $[z_{min}, z^{i}_{max}]$ for each layer, as outlined in Figure 3. Aggregate per-sample pass- and fail-masks can be constructed from the triangle's coverage mask and the selection mask using simple bitwise operations. The exact depth test is only required for the samples that are not present in either of the pass- or fail-masks. Pseudo-code for how the coarse depth test is performed is given in Listing 1 in the appendix.

Coarse depth buffer update Unless all samples were culled by the coarse depth test, we must update the coarse depth buffer in a way that makes it conservatively bound the contents of the depth buffer. Updating the z_{min}-value is done as previously described in Section 3. The challenge lies in updating the z^{i}_{max} values and the selection mask. The incoming triangle forms a third depth layer, in addition to the, up to, two layers already populating the tile. We handle layer overflow by merging two of the layers using a heuristic, as shown in Figure 4 and described in detail below.

First, consider a single sample, S, which belongs to layer i and is also found to be overlapping the incoming triangle. With a *less than* depth test, we know that

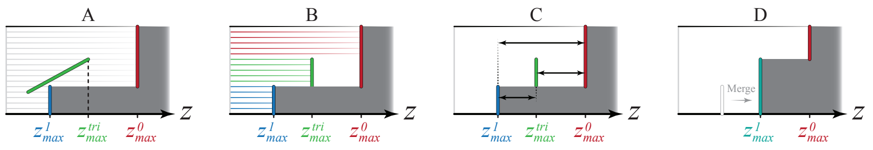


Figure 4: An example of our occluder merge heuristic for a single tile shown in normalized device coordinates (NDC). The occluded region, as encoded by our algorithm, is shown in dark gray. A: *a green, slanted primitive is rendered in front of two existing layers, illustrated with red and blue lines respectively.* B: *each sample is classified as belonging to one of the layers to create sample masks.* C: *for our merge heuristic, we find the closest pair of layers along the z-axis, which in this case is between z^1_{max} and z^{tri}_{max}.* D: *we select the maximum of these two depth values as the new z^1_{max} and fuse their sample masks to form a new selection mask.*

the depth of S after the depth test will be *at most* the minimum (closer) value of z^i_{max} and z^{tri}_{max}. Based on this observation, by comparing both z^i_{max}-values to z^{tri}_{max}, we can categorize which layer each sample belongs to – either its previous layer, i, or the incoming triangle layer. From this, we construct three non-overlapping *sample masks* signaling which of the three layers, z^0_{max}, z^1_{max}, and z^{tri}_{max}, each sample belongs to, as step B in Figure 4 exemplifies.

After categorizing the samples, if there are any layers that do not have samples associated with them (i.e., the sample mask is empty for a layer), the coarse buffer update is simple. Since the resulting number of layers is ≤ 2, the data will fit in our representation and we can simply write the populated layers to the coarse depth buffer. If there are samples in all three layers, we use a simple distance-based heuristic to select which layers should be merged. The underlying assumption is that triangles that have similar depth values are likely to be part of the same surface. As illustrated in step C in Figure 4, we first compute the distances between all of the layers as

$$\begin{aligned} d_{T0} &= |z^{tri}_{max} - z^0_{max}|, \\ d_{T1} &= |z^{tri}_{max} - z^1_{max}|, \\ d_{01} &= |z^0_{max} - z^1_{max}|. \end{aligned}$$

The shortest distance is then used to determine which merge operation is performed, as depicted in step D in Figure 4.

1. If d_{T0} is smallest then $z^0_{max} = \max(z^{tri}_{max}, z^0_{max})$.
2. If d_{T1} is smallest then $z^1_{max} = \max(z^{tri}_{max}, z^1_{max})$.
3. Otherwise $z^0_{max} = \max(z^0_{max}, z^1_{max})$ and $z^1_{max} = z^{tri}_{max}$.

The sample masks of the two closest layers are also merged (using simple bitwise operations) to produce the new selection mask. Pseudo-code for the update and merge functions can be found in Listing 3 and Listing 4 in the appendix.

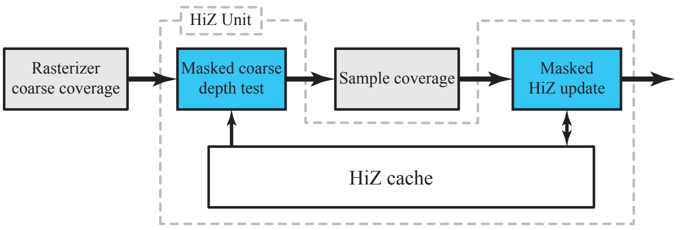


Figure 5: It is possible to modify the coarse depth test to rely only on $z_{\text{min}}/z_{\text{max}}$*, and not per-sample coverage results. Unlike our solution illustrated in Figure 2 (bottom), this alternative implementation does not incur the expense of per-sample coverage testing prior to HiZ, but can decrease efficiency as culling is performed on more conservative information.*

Switching Depth Functions Our algorithm can easily handle depth function switches while rendering. For the *greater than* depth functions, tiles are instead represented by two z^i_{min} values and one z_{max} value. We store a single bit for each coarse depth buffer tile indicating which representation is currently used. If the tile does not match the current depth function we convert it before updating the coarse depth buffer. Conversion is performed by conservatively swapping the min and max values. For example, if the tile stored in the coarse depth buffer has two max layers, but the depth function is changed to *greater than*, we convert the tile by setting $z_{max} = \max(z^0_{max}, z^1_{max})$ and $z^0_{min} = z^1_{min} = z_{min}$ and clearing the selection mask. The conversion is quite crude and may lose a lot of culling information, but we have not found any workload where this has been an issue, as depth function changes are infrequent in most scenes. All standard OpenGL/DirectX depth functions can be handled using the *less than* or *greater than* representation.

Coarse depth test pipeline placement As can be seen in the bottom row of Figure 2, our coarse depth test is based on the coverage mask, which means that per-sample coverage testing must be moved to the rasterizer block. As previously mentioned, placing the per-sample coverage test behind the HiZ unit is beneficial, as coverage testing may be skipped for culled tiles, reducing the load of this unit.

We can achieve a similar effect by using an alternate coarse depth test, where we perform an overlap test between the $[z^{tri}_{min}, z^{tri}_{max}]$ and $[z_{min}, \max(z^0_{max}, z^1_{max})]$ intervals, similar to the classic HiZ test. As shown in Figure 5, we may then place the per-sample coverage test between the coarse depth test and update, which results in similar load balancing to previous work. This version of the coarse depth test is less accurate, but most of the benefits of our algorithm comes from the accurate update. Pseudo-code for this alternative approach can be found in Listing 2 in the appendix. Compared to the results presented in Section 5, the culling rate of our algorithm decreases by $0.2 - 2.1$ percentage points and total bandwidth increases by $0.7 - 1.4$ percentage points. It is also possible to perform both versions of the coarse test, efficiently filtering most per-sample coverage tests while retaining the benefits of the accurate version.

4.1 Compression

In order to reduce coarse depth buffer bandwidth, we use a simple compression scheme when entries are evicted from the coarse depth buffer, similar to how regular depth buffer compression works [51, 53].

Our compression scheme is inspired by zerotree encoding of wavelet coefficients [108]. Each tile is first split into a set of *blocks*, each containing b samples. For each block, we store a single bit signaling whether its samples contain a mix of indices to both layers or if all samples belong to the same layer. If all indices are the same, only one additional bit is required to assign the entire block to the layer. A block containing indices to both layers require an explicit mask with b bits. With this scheme, the compressed selection mask cost, c, for the tile containing s samples is $c = 2\frac{s}{b} + (b-1)m$, where m is the number of blocks that need to be explicitly stored. The selection mask can be compressed without loss if $c \leq s$. Interestingly, we can limit m by performing lossy compression, without introducing artifacts in the rendered image. The coarse depth buffer representation is still valid (i.e., conservative w.r.t. the depth buffer) if we alter an index in the selection mask to use the farther of the two z^i_{max}-values. It is thereby possible to enforce a maximum value of m by forcing some blocks to use a single layer, instead of a mix of both.

Furthermore, we decrease the precision of z_{min} and z^i_{max}. There is a variety of possible options depending on the bit budget available, the depth buffer target format, and the expected distribution of depth samples. We have opted to use a simple reduced precision float with fewer exponent and mantissa bits. In addition, we only use negative exponents and no sign bit, limiting the representable range to $[0, 1]$.

5 Results

When comparing different culling algorithms, there are two main quantities that are of interest – memory bandwidth usage and throughput. Bandwidth is primarily consumed by the depth unit when reading and updating the depth buffer, and to a lesser extent by the HiZ unit for maintaining the coarse depth buffer. The number of per-sample tests the depth unit has to perform depends on the amount of tiles culled (failed) by the coarse depth test, and consequently a higher culling rate leads to better throughput. Depending on the system and the expected workloads, these quantities must be balanced against each other for maximum performance. We evaluated five different pipeline configurations listed below with regards to bandwidth and culling rates (i.e., the percentage of tiles culled by the coarse depth test):

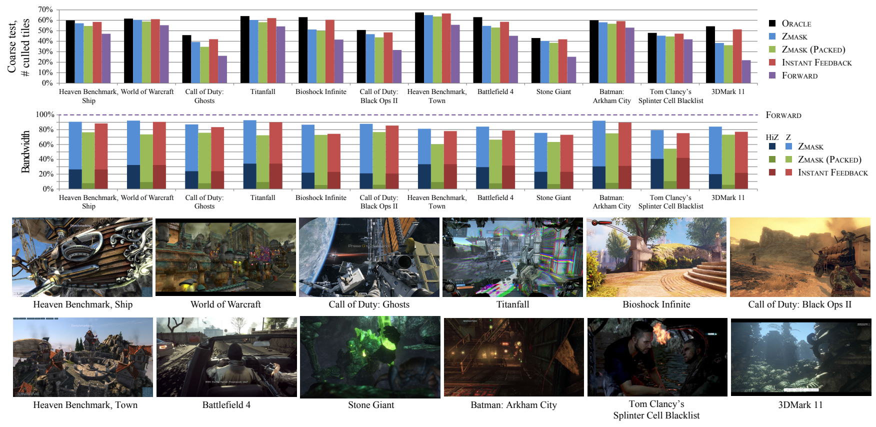


Figure 6: Top: *the percentage of incoming 16 sample tiles (i.e.* 2×2 *pixels for* $4\times$ *MSAA targets and* 4×4 *pixels for single sample targets) that were culled by the coarse depth test for the different algorithms. A higher number means that less work is pushed through the pipeline.* Bottom: *the simulated depth buffer (Z) and HiZ bandwidth, normalized to the* FORWARD *algorithm. The darker bars indicate the HiZ bandwidth and the lighter bars is the depth buffer bandwidth. Note that even though the* PACKED ZMASK *culls fewer tiles and has a higher depth buffer bandwidth, the total bandwidth is still very low due to the HiZ bandwidth savings. Note that the screen shots are generated using our hardware simulator. While we strive to make it feature complete according to the latest OpenGL/DirectX specifications, some visual differences may occur compared to commercial GPUs and production drivers.*[3]

- ORACLE - The HiZ unit replicates the exact depth buffer and performs per-sample depth tests. For each sample, there is no ambiguous outcome, only pass or fail. A tile is only classified as ambiguous if it contains both samples passing and failing the depth test. This pipeline is only used to get an upper bound on possible cull rates.
- FORWARD - Only the forward update unit for $z_{\min}/z_{\max}$-culling is enabled. A feedback unit with infinite delay will act as a forward-only pipeline, which makes this configuration a lower bound on the culling rate such a design can achieve.
- INSTANT FEEDBACK - Both the forward and the feedback HiZ update mechanisms are used. z_{max}-updates are triggered directly on depth buffer updates (i.e., as early and as often as possible), and there is no feedback delay, which gives us an upper (albeit unrealistic) bound on how well a forward/feedback-design can perform.
- ZMASK - Our proposed feed forward algorithm.
- PACKED ZMASK - Our algorithm tailored to minimize bandwidth. This variant requires an additional post-HiZ cache compression stage, as described in Section 4.1.

Our results are based on a C++ hardware simulator, which models the system on a functional level. We use a 32 kB depth cache and a 16 kB HiZ cache, both using a cache line size of 64 B (unless otherwise stated) and both with a least recently used (LRU) replacement policy. For our main results, found in Figure 6, we allocate a coarse depth buffer which amounts to an overhead of 4 bits per depth sample for the FORWARD, INSTANT FEEDBACK, and ZMASK configurations. With this storage, we can keep z_{min}^{tile} and z_{max}^{tile} in a 32 bit format each at a 16 sample granularity for FORWARD and INSTANT FEEDBACK (this corresponds to 4×4 pixel tiles for single sample targets and 2×2 pixels for $4\times$ multi-sample targets). For ZMASK we store one z_{min} and two z_{max}^{i} at a 32 sample granularity, using 32 bits for each entry, as well as a 32 bit per-sample selection mask (corresponding to 8×4 pixel tiles for single sample targets and 4×2 pixel tiles for $4\times$ MSAA targets). All configurations use the common *fast clears* [87] optimization to ensure that the results are not biased by how the different algorithms handle clear values.

The PACKED ZMASK is similar to the ZMASK algorithm, but uses larger tiles (16×8 pixel and 8×4 pixel tiles for single and $4\times$ MSAA targets respectively), and compresses them as they are evicted from the HiZ cache. In the cache, each tile occupies $128 + 3 \cdot 32 = 224$ b, or 28 B of memory. Since we do not want to alter the memory transaction size of 64 B, we group 4 tiles to a common cache line of 112 B, which is compressed down to 64 B on eviction. To achieve this level of compression, we use a reduced precision float format with 4 exponent bits and 11 mantissa bits for the z_{min} and z_{max}^{i} values, one bit to encode the test direction (see Section 4), and the remaining 82 bits are spent on storing the selection mask using the compression format described in Section 4.1. Thus, when using the PACKED ZMASK pipeline, each tile occupies 1.75 bits per sample while in the cache, and reading or writing the tile from/to memory uses 1 bit of bandwidth per sample. The tile size was selected empirically by finding the best balance between depth buffer bandwidth and HiZ bandwidth, as described in Figure 8.

Our main results are shown in Figure 6, where we present the coarse culling rates and the bandwidth consumed by each pipeline. The culling rates have been normalized to 16 sample tiles for the ZMASK and PACKED ZMASK algorithm to simplify comparison. Since the coarse depth test pass rates are very similar between the algorithms, we focus solely on the number of culled tiles (i.e., tiles where the depth test unambiguously fails) as a measure of throughput. The test suite contains a number of traces from modern games, with a variety of different render state combinations, and includes $1-4\times$ MSAA buffers as well as auxiliary targets such as shadow maps. As can be seen from the results, our algorithm is often close to the ORACLE in terms of culling efficiency. The FORWARD pipeline is al-

[3] *Heaven Benchmark* screenshots courtesy of UNIGINE Corp. *World of Warcraft*, *Call of Duty®: Ghosts* and *Call of Duty®: Black Ops II* screenshots courtesy of Activision Blizzard, Inc. *Titanfall* screenshot courtesy of Respawn Entertainment. *Bioshock Infinite* Screenshot Courtesy of Irrational Games and Take-Two Interactive Software, Inc. *Battlefield 4*, © 2013 Electronic Arts Inc. Battlefield and Battlefield 4 are trademarks of EA Digital Illusions CE AB. Image from *Stone Giant* demo, courtesy of BitSquid. *Batman: Arkham City* screenshot courtesy of Rocksteady Studios. *Tom Clancy's Splinter Cell Blacklist* screenshot courtesy of Ubisoft. *3DMark 11* screenshot courtesy of Futuremark.

ways considerably less efficient than all of the alternatives. On average, we retain 90% of the rejection rate of the ORACLE pipeline, with the more difficult cases typically being scenes with a greater amount of alpha tested geometry, or scenes that output depth in the fragment shader. Note that ZMASK uses about 14% less total bandwidth compared to FORWARD, while INSTANT FEEDBACK uses about 18% less than FORWARD. It should be noted, however, that INSTANT FEEDBACK is idealized and impractical to implement. Increasing the pipeline delay reveals the weakness of the feedback algorithm, as we will show later in this section. By increasing the tile size and enabling compression using the PACKED ZMASK configuration, we lower the number of culled tiles and depth buffer bandwidth increases as a result. However, since the HiZ bandwidth is reduced, total bandwidth consumption is approximately 30% less than FORWARD on average, which is a substantial reduction.

The culling numbers presented for the ORACLE algorithm include tiles where the coverage is modified by the fragment shader. Alpha tested billboards, for examples, can have large, fully transparent portions that are discarded in the fragment shader.

Feedback Delay As previously mentioned, the performance of the feedback algorithm depends on how long delay can be expected in the pipeline. We opted to implement the feedback mechanism on depth buffer updates, rather than cache evicts, and simulate delay by introducing a FIFO-queue when feeding the messages back. This allows us to delay the messages by an arbitrary number of processed tiles and study how increased delay affects system performance.

Figure 7 shows how the number of culled tiles and total simulated depth buffer bandwidth (for both the coarse and exact depth buffers) are affected by an increased delay compared to the ZMASK approach. As expected, the number of culled tiles decreases with increasing delay. Similarly, total bandwidth usage increases significantly for larger delays. When the delay becomes sufficiently large, the coarse depth buffer entries may already have been evicted from the HiZ cache before a feedback message is received. Consequently, the data must be read back into the cache in order to perform the feedback update, and the HiZ and feedback mechanisms will compete for which data should be resident in the cache. Therefore, it is important to balance the size of the HiZ cache and the depth unit cache based on the expected delay of the system. Alternatively, feedback updates of tiles not resident in the HiZ cache could be discarded, which avoids thrashing at the cost of reducing culling rates even further.

It should be noted that the delays of pipelining, shading, and caching in a real system causes culling efficiency and bandwidth to scale in a much more intricate way, and Figure 7 should only be seen as indicative of the trend. In our system, we observe a significant bandwidth impact when using an evict-based feedback strategy. If we halve the HiZ cache size to 8 kB, while keeping depth cache constant, HiZ bandwidth increase by about 10% to 45%, depending on the scene.

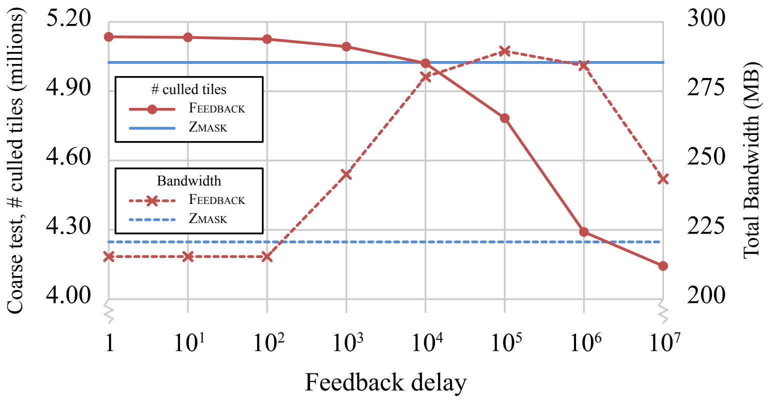


Figure 7: Artificial delay of the z^{tile}_{max} feedback updates for the Heaven Benchmark, Ship *scene. Here, each unit corresponds to delaying the feedback update by one processed tile. At a delay of 10^3 tiles, we see that the coarse depth test rejection rate starts to drop, while the bandwidth rapidly increases. While part of the increasing bandwidth is explained by reduced culling efficiency, the lion's share is due to cache behavior. As the delay increases, the tiles referenced by the feedback messages may already have been evicted from the HiZ cache, which may lead to cache thrashing and increased bandwidth. This creates an intricate relation between the system delay, which may depend on depth buffer cache size and shader execution, and the size of the HiZ cache. At 10^7 steps, the delay is so large that the rendering finishes before any of the feedback updates occur, leaving only the forward updates (i.e., equivalent of running the* FORWARD *algorithm). Since there are no delayed updates at this point, there are also no cache conflicts, which explains the bandwidth reduction.*

Tile size As our algorithm is of feed forward nature, it allows us to easily decouple tile sizes of the coarse and exact depth buffers, and this gives flexibility to chose whatever tile size gives the best trade-off between culling efficiency and coarse depth buffer bandwidth. In Figure 8, we show how our algorithm scales when varying tile size. As a reference, we also include baseline results for the INSTANT FEEDBACK pipeline.

Multiple depth layers Our algorithm can be extended to use more depth layers and we have observed some improvement in culling rates when using three or four layers, as shown in Figure 9. However, we feel that the improvement is not significant enough to motivate the added complexity. Increasing the number of layers requires more coarse depth buffer storage, as we must store additional z^i_{max} values and use more bits per sample to store the selection mask. It also complicates layer merging as the number of ways that we can merge layers scales $\mathcal{O}(n^2)$.

Stochastic motion blur As a proof of concept, we plugged the ZMASK algorithm in to our existing framework which simulates a stochastic rasterization

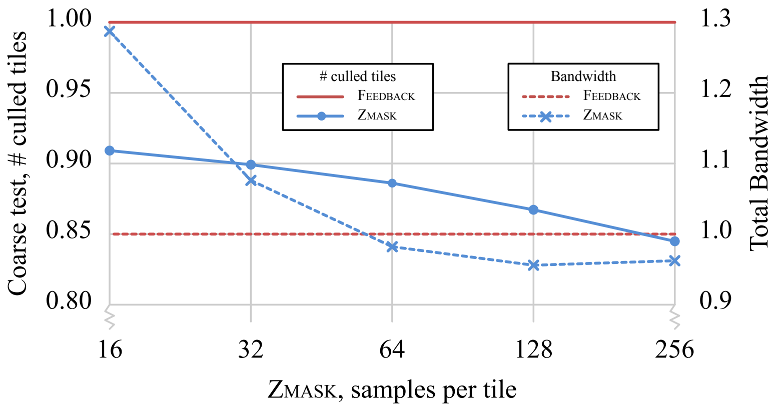


Figure 8: The graph illustrates how bandwidth and culling rates are affected by varying the tile size for ZMASK*. The numbers are combined over all of our test scenes, normalized to the* INSTANT FEEDBACK *algorithm with fixed* 16 *sample tiles. To simplify implementation, the cache line and memory transaction size match the footprint of a single tile. Although depth buffer bandwidth increases due to the lowered cull rate, the decrease in HiZ memory traffic counter this effect. In our setup, the lowest combined bandwidth occurs around* 128 *samples, which is the tile size selected for the* PACKED ZMASK *algorithm.*

pipeline in hardware to render images with motion blur. The framework uses the TZSLICE algorithm [6] to perform $z_{\max}$-culling, which is the most bandwidth efficient variant of the more general tz-pyramid [20], according to Munkberg et al. [90]. Updates to the coarse depth buffer is done in the same way as the INSTANT FEEDBACK algorithm. For TZSLICE, we used one 32-bit float z_{max}-value for each slice of $4 \times 4 \times 1$ $(w \times h \times time)$ samples. For ZMASK, we used $4 \times 4 \times 4$ tiles (64 samples) with two layers. Thus, both of these configurations use a 2 bit overhead per sample of HiZ data. As can be seen in Figure 10, even without any algorithmic modifications, ZMASK compared very well to TZSLICE. Note that these are early experiments and we believe that there is a lot more potential for improvement in this area.

Limitations While our algorithm handles even complex geometry very well, the main drawback relative to the feedback approach is that we cannot handle alpha testing, pixel shader discards, or pixel shader depth writes as accurately. While this could be an issue in extreme cases, none of our test applications seem to rely on alpha tested geometry for the main occluders (although we present an abundance of examples using alpha testing). Note that the feedback approach is also affected by alpha testing, as it implies deferring the feedback update until after the shader has been executed. It should also be noted that our algorithm could be used in conjunction with feedback updates.

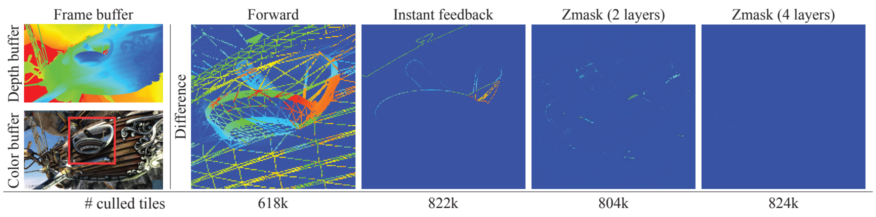


Figure 9: The impact of multiple depth layers on the quality of the coarse depth buffer. The cropped images show the difference between the z_{max}-value of the coarse depth buffer and the exact depth buffer. The forward pipeline has massive leakage along triangle edges, and silhouette edge shows up in INSTANT FEEDBACK *as only one z_{max}-value is stored per tile. The layer merging inaccuracies visible in the two layer version of our algorithm almost entirely disappears with four layers. However, as can be seen in the table, two layers perform well enough not to motivate the extra complexity of adding additional layers.*

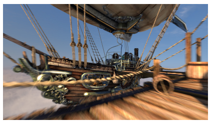
			
# culled tiles	94%	91%	84%

Figure 10: Three scenes with stochastic motion blur. We count the number of $4 \times 4 \times 1$ *tiles culled with* ZMASK *compared to a* TZSLICE *baseline.*

6 Conclusions

We have proposed a novel $z_{\text{min}}/z_{\text{max}}$-culling algorithm, which we believe is an interesting and competitive alternative to the traditional feedback update mechanism. Our algorithm has similar performance to that of an *ideal* feedback architecture (without delay), but retains the benefits of a strict feed forward pipeline. This means that implementation and validation is simplified as we do not need to consider or handle hazards that may occur from the feedback delay. Furthermore, as we decouple the tile sizes of the coarse and regular depth buffers, we have great freedom in choosing tile sizes and bit layout for the coarse depth buffer entries. This makes it easy to load-balance a hardware system and simplifies the re-design cycle if memory bus width or cache line size changes. Thus, we believe our approach is a cost-efficient and flexible solution that is suitable for current and future GPUs.

Acknowledgements

We would like thank the anonymous reviewers for their feedback. We also thank David Blythe and Mike Dwyer for supporting this research. Tomas Akenine-Möller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg foundation. Thanks to the following people for helping us with permissions for the screenshots in this paper: Beth Thomas at UNIGINE Corp., Christer Ericson at Activision, Abbie at Respawn Entertainment, Naty Hoffman at 2K Games, Martin Lindell and Johan Andersson at EA Digital Illusions CE, Tobias Persson at Bitsquid, Kelly Ekins at WB Games Montréal Inc., Guy Perkins at Rocksteady Studios, Heather Steele at Ubisoft and James Gallagher at Futuremark.

Appendix

Coarse depth test The coarse depth test produces two per-sample masks – a pass mask and a fail mask. The remainder of the samples must to be tested using the regular per-sample depth test.

```
function coarseZTest(tile, tri)
    failMask0 = tri.zMin >= tile.zMax[0]
        ? tri.rastMask & ~tile.mask : 0
    failMask1 = tri.zMin >= tile.zMax[1]
        ? tri.rastMask & tile.mask : 0
    failMask = failMask0 | failMask1
    passMask = tri.zMax < tile.zMin ? tri.rastMask : 0
    return [passMask, failMask]
```

Listing 1: Perform coarse depth test.

An alternative version of the coarse test may be performed before the per-sample coverage test. Contrasting the version above, this test does not account for coverage. We still observe good cull rates, as the accurate update is the key to the performance of our algorithm.

```
function coarseZTest_noMask(tile, tri)
    if tile.mask == 0:
        maxOfMax = tile.zMax[0]
    else if tile.mask == ~0:
        maxOfMax = tile.zMax[1]
    else:
        maxOfMax = max(tile.zMax[0], tile.zMax[1])
    fail = tri.zMin >= maxOfMax
    pass = tri.zMax < tile.zMin
    return [pass, fail]
```

Listing 2: Perform coarse depth test without coverage mask.

Updating the coarse buffer The coarse buffer is trivially updated if any of the layers are overwritten, while the heuristic-based merge function is called to resolve complicated multi layered situations.

```
function coarseZUpdate(tile, tri)
    triMask0 = tri.zMax < tile.zMax[0]
        ? tri.rastMask & ~tile.mask : 0
    triMask1 = tri.zMax < tile.zMax[1]
```

```
        ? tri.rastMask & tile.mask : 0

    triMask = triMask0 | triMask1
    layer0Mask = ~tile.mask & ~triMask
    layer1Mask = tile.mask & ~triMask

    if triMask != 0:
        if layer0Mask == 0:
            // Layer 0 is empty and is replaced
            tile.zMax[0] = tri.zMax
            tile.mask = ~triMask
        else if layer1Mask == 0:
            // Layer 1 is empty and is replaced
            tile.zMax[1] = tri.zMax:
            tile.mask = triMask
        else:
            // All layers contain samples, merge
            merge(tile, tri, triMask)
```

Listing 3: Update coarse depth buffer.

Merging depth layers The merge function reduces three layers to two and updates the selection mask.

```
function mergeClosest(tile, tri, triMask)
    dist0 = abs(tri.zMax - tile.zMax[0])
    dist1 = abs(tri.zMax - tile.zMax[1])
    dist2 = abs(tile.zMax[0] - tile.zMax[1])
    if dist0 < dist1 && dist1 < dist2:
        // Merge triangle layer with layer 0
        tile.zMax[0] = max(tile.zMax[0], tri.zMax)
        tile.mask = tile.mask & ~triMask
    else if dist1 < dist2:
        // Merge triangle layer with layer 1
        tile.zMax[1] = max(tile.zMax[1], tri.zMax)
        tile.mask = tile.mask | triMask
    else:
        // Merge layer 0 and 1
        tile.zMax[0] = max(tile.zMax[0], tile.zMax[1])
        tile.zMax[1] = tri.zMax
        tile.mask = triMask
```

Listing 4: Merging heuristic

Paper V

Filtered Stochastic Shadow Mapping using a Layered Approach

Magnus Andersson Jon Hasselgren Jacob Munkberg
Tomas Akenine-Möller

Lund University Intel Corporation

ABSTRACT

Given a stochastic shadow map rendered with motion blur, our goal is to render an image from the eye with motion blurred shadows with as little noise as possible. We use a layered approach in the shadow map, and reproject samples along the average motion vector, and then perform lookups in this representation. Our results include substantially improved shadow quality compared to previous work and a fast GPU implementation. In addition, we devise a set of scenes that are designed to bring out and show problematic cases for motion blurred shadows. These scenes have difficult occlusion characteristics, and may be used in future research on this topic.

to appear in *Computer Graphics Forum*, 2015.

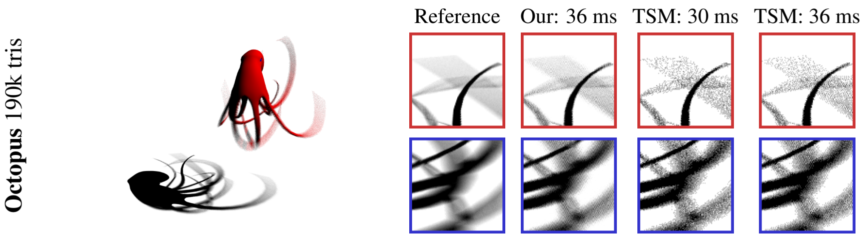


Figure 1: An octopus in motion casting a complex motion blurred shadow rendered by our algorithm. With the same input samples, our algorithm has significantly less noise compared to time-dependent shadow maps (TSM). At equal time, the noise level is still largely reduced. The animated octopus mesh is taken from the Alembic source distribution.

1 Introduction

Motion blur in photographed images, offline rendering, and in real-time graphics provides the viewer with a sense of motion direction and also reduces temporal aliasing [93]. When motion blur is present, the shadows of moving objects should be motion blurred as well. However, while shadow rendering has received a lot of attention in the research community for static scenes [34], rendering of motion blurred shadows has remained relatively unexplored.

Accumulation buffering can be used to generate motion blurred shadows [49], but the algorithm only supports using identical sample times for all pixels in the shadow map [120]. This often shows up as banding artifacts unless many full-screen passes are used, and that is often not possible within the time budget for real-time rendering. Instead, one can use stochastic sampling when generating the shadow map [6]. This removes banding artifacts by allowing samples to have unique times, but the resulting image is often too noisy at affordable sample rates. Deep shadow mapping [76] is a technique originally intended for rendering shadows for hair, smoke, fur, etc. Motion blurred shadows can also be generated using deep shadow maps, however, they are only correct when the receiving object is static. A thorough review of previous work can be found in Section 2.

Since the receiver, the shadow caster, and the light source can all move at the same time, rendering motion blurred shadows is a notoriously difficult problem, and largely unsolved in the domain of real-time rendering. Our approach to motion blurred shadows is to generate a time-dependent stochastic shadow map (TSM) [6] from the light source, and then develop a novel filtering algorithm in order to reduce shadow noise. In our algorithm description, we make extensive use of epipolar images (see Figure 2). Examples of our results can be seen in Figure 1. We also implement our algorithms on a graphics processor and provide interactive or near real-time rendering performance with substantially reduced noise levels compared to TSM, and in addition, our algorithm handles more cases correctly than previous methods.

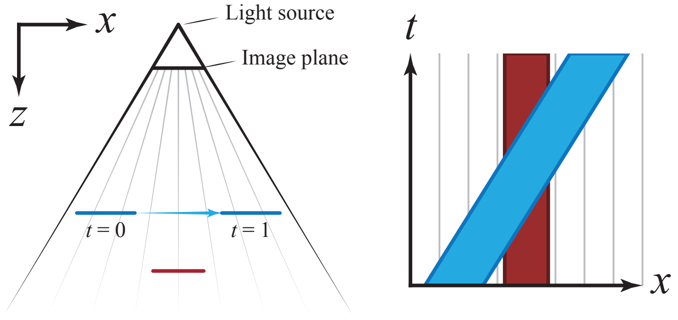


Figure 2: Left: a blue line segment is moving parallel to the image plane of the light, in front of a static red line segment. Right: the light space epipolar image*, i.e., a spatio-temporal plot in x and t. The blue segment partially occludes the red line for a certain time interval.*

2 Previous Work

The number of publications for shadow rendering is huge, and for a comprehensive overview, we refer to the book by Eisemann et al. [34]. For an overview of motion blur in graphics, we refer to the state-of-the-art report by Navarro et al. [93]. Most of the shadow rendering algorithms have been developed for static scenes, while the number of methods that incorporate motion blur is sparse.

Akenine-Möller et al. introduce time-dependent shadow mapping (TSM) [6], where multiple shadow maps are created using stochastic rasterization. Each shadow map represents a time slice of the full exposure and stratified sampling is used to ensure that exactly one sample falls into each time slice. In a second pass, the scene is rendered using stochastic rasterization from the camera's point of view. Again, stratified sampling is used to draw one sample from each time slice. The shadow test is performed using the shadow map sample at the time slice corresponding to the camera sample. This means that they are close in time and, given a static or slowly moving light source, in space, though they do not match exactly.

The idea behind deep shadow mapping (DSM) [76] is to store multiple semi-transparent occluders in each pixel of a shadow map. The visibility of a point, **p**, can then be computed as $\prod_{p_z<z_i}(1-\alpha_i)$, where z_i and α_i are the depth and transparency of the i:th occluder. The authors show that DSM can be used for motion blurred shadows by treating each moving occluder as a semi-transparent layer. The scene is rendered from the light's point of view using stochastic rasterization. For each pixel in the shadow map, the samples are clustered into distinct depth layers and each layer is given an opacity proportional to the number of samples drawn from that layer. The layers are then encoded into the DSM and are treated like transparent occluders. Since the time dimension is collapsed, i.e., motion blur is

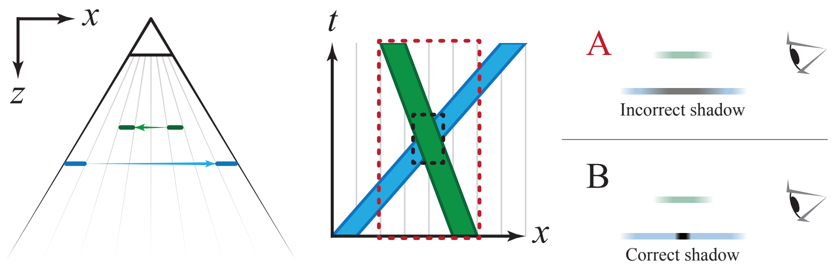


Figure 3: Left: two objects are moving in opposite directions at different distances from the light source. Middle: the light space epipolar image reveals that the green segment only covers the blue segment for a brief moment, as indicated by the black, dotted outline. By omitting time information, the green occluder appears to cast a shadow on the blue receiver throughout the entire exposure time, as indicated by the red dotted outline. Right: collapsing the time dimension produces a smeared shadow (A)*. The correct shadow is generated by taking temporal occlusion into account* (B)*.*

treated as transparency, certain problems can occur, e.g., for moving receivers. An example of this problem is illustrated in Figure 3.

McGuire and Enderton [81] present an alternative to DSM called colored stochastic shadow mapping (CSSM), where a transparent layer is encoded stochastically by allowing a proportional amount of shadow map samples to pass through the transparent layer, and instead get the depth value of the layer behind, similar to stochastic transparency. The shadow map lookup is filtered by drawing a number of spatially coherent samples and using the averaged shadow term. Stochastic motion blur rasterization and stochastic transparency is analogous, and therefore the samples generated by a stochastic rasterizer could be used directly as input to the CSSM algorithm to extend it to handle motion blurred shadows. However, similar to DSM, the time dimension is collapsed in CSSM, and hence inherits the same problems.

While the results of multiple depth tests can be filtered together [105], filtering the depth values prior to the depth test, however, will not yield the correct result. The problem of creating a filterable shadow map representation has been given a lot of attention, most notably variance shadow mapping (VSM) [29], convolution shadow mapping (CSM) [11], and exponential shadow mapping (ESM) [12]. VSMs have also been extended to render plausible soft shadows [123], where the filter size is computed based on the average occluder distance to the receiver and the light source [38]. We rely on variance shadow maps to create a representation where the time dimension can be efficiently filtered. We also experimented with using ESM, but saw no convincing benefit in neither quality nor performance. As our shadow map representation is split into a set of depth layers, we avoid most of the shine-through artifacts from VSMs, similar to layered variance shadow maps [70].

Lehtinen et al. [73] present a motion and defocus blur reconstruction algorithm, which also supports the case of motion blurred and soft shadows by taking the entire 7D light-field into account during reconstruction. Their algorithm uses a dual acceleration structure, one for the samples visible from the camera, and a second for the depth as seen from the light source. When reconstructing a camera space point $\mathbf{p} = (x, y, t)$, the corresponding light space sample at (l_x, l_y, t) is reconstructed and a binary shadow test is performed. While their algorithm produces very high quality images, results rely on reconstructing and filtering many ($\sim$128) locations per pixel. The reconstruction time is over 10 seconds even for high-end discrete GPUs on 5D examples (excluding motion blur shadows).

Egan et al. [33] use frequency analysis [30] of the motion blurred light field to derive sheared filters for reconstruction and to drive adaptive sampling. Frequency analysis can also be used to derive filters and adaptive sampling techniques for shadows from complex occluders [32], directional occlusion [31], soft shadows filtering [84], diffuse indirect lighting [85], and multiple distribution effects [86].

Our shadowing approach has similarities with recent reconstruction algorithms, most notably the work by Munkberg et al. [92] and Hasselgren et al. [54], who use a similar layered representation and sheared $5D$-filter to reconstruct *primary visibility* for motion blur and depth of field. Their work does not handle reconstruction of motion blurred shadows and as such, our work can be seen as an important complement. In addition, we extend on their work by taking into account the correlation between time of samples in the shadow map and samples relating to primary visibility. We also propose a novel filtering approach.

3 Theory

In this section, we briefly present the theory behind our setup for motion blurred shadow rendering. The outgoing radiance from a point x in a direction ω_o is given by:

$$l(x, \omega_o) = \int_{\Omega_i} v(x, \omega_i) f(x, \omega_i, \omega_o) l(\omega_i) d\omega_i, \tag{1}$$

where v is the visibility function, f is the BRDF (including the cosine term), and $l(\omega_i)$ is the incoming light. We only consider direct lighting from a set of discrete point or directional light sources, $\{l_i\}$, and the expression for the outgoing radiance can therefore be simplified to a sum over the light sources:

$$l(x, \omega_o) = \sum_i v(x, \omega_i) f(x, \omega_i, \omega_o) l_i. \tag{2}$$

Now, we assume that the scene is dynamic, where the objects, lights, and the camera can move. The corresponding expression for the outgoing radiance at a certain time, t, can be expressed in a coordinate system following x as:

$$l(x, \omega_o(t), t) = \sum_i v(x, \omega_i(t), t) f(x, \omega_i(t), \omega_o(t)) l_i. \tag{3}$$

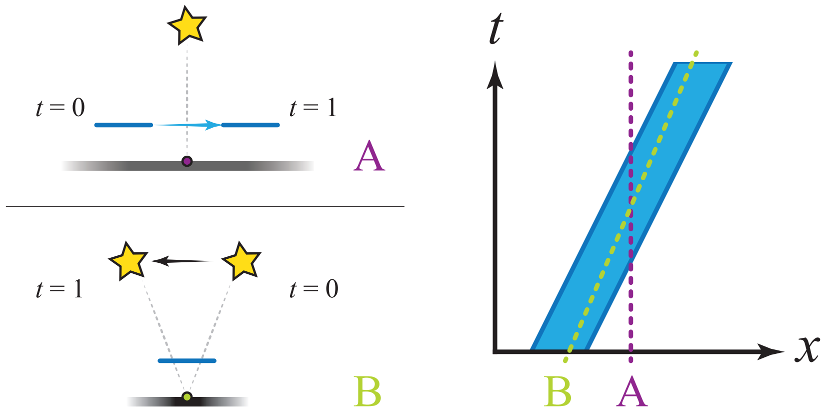


Figure 4: Left: scene A *shows a static light source and a moving occluder, while scene* B *instead shows a moving light and a static occluder. Right: the two different scenes produce the same light space epipolar image, yet the observed shadows are very different. This is a consequence of the receiver being static in light space in scene* A*, while the receiver point move in scene* B *(due to the moving light source). The footprint for the receiver point is shown by the dashed lines in the epipolar image.*

This expression shows the outgoing radiance at x, but does not take occlusions into account between x and the camera.

To render motion blurred shadows, we want to evaluate the occlusion term $v(x, \omega_i(t), t)$. In a ray tracer, one can simply answer this query with a shadow ray through the dynamic scene. In a stochastic rasterizer, one can instead query a time-dependent shadow map [6], which stores a light space depth value (z) for each spatio-temporal coordinate (x_l, t). To do this, the query coordinate x and direction $\omega_i(t)$ is remapped into the moving coordinate system of the light (denoted with subscript l): $(x, \omega_i(t), t) \mapsto (x_l(t), t)$. If the shadow map depth is smaller than the light space depth of x, the light source, l_i, is occluded from x at time t.

To determine the color for each pixel, we want to integrate over t to compute a blurred value over the open interval of the camera shutter. Hence, due to motion and the spatial pixel filter, many points will contribute to the blurred radiance value of each pixel. The shading evaluation may include multiple shadow map lookups within a spatio-temporal footprint, as illustrated in Figure 4.

Furthermore, there may be discrete changes in primary visibility, as different primitives move over the pixel's view frustum over the temporal interval. In the general case, the camera, all objects, *and* all lights may move in time. To approximate this result, one often take a large number of spatio-temporal Monte Carlo (MC) samples. However, with an estimate of the footprint in the spatio-temporal shadow map of hit points on visible primitives, one can apply filtered lookups in order to reduce shadow noise. This is the main goal in this paper.

Due to perspective motion, $x_l(t)$ may be a rational polynomial in t. However, similar to recent motion blur filters [33, 83, 92], we make a linear motion assumption in our shadow map representation.

4 Algorithm

We propose a layered, filtered shadow mapping algorithm for motion blurred shadows. The algorithm is divided into two passes, namely a *shadow pass* and a *lighting pass*. The shadow pass renders the scene using stochastic rasterization [6] and generates a time-dependent shadow map augmented with per-sample motion vectors. The subsequent lighting pass renders the scene from the camera's point of view, and performs a shadow query for each sample seen from the camera.

In contrast to time-dependent shadow mapping (TSM), wherein the shadow query gives a binary result, i.e., if the sample is in shadow or not, our algorithm estimates the temporal integral of the visibility term (discussed in Section 3), which results in smoother motion blurred shadows.

Our algorithm is based on the assumption that the motion in a small spatial region of a depth layer is slowly varying, which has been a successful approximation in previous work [92]. During the shadow pass, we divide the stochastic shadow map into texture space tiles, and split samples of each tile into depth layers. We then process each such tile and depth layer individually.

First, we compute an average motion vector $\mathbf{d}$ for each depth layer in each tile. If all spatio-temporal samples, $\{(\mathbf{x}_i, t_i)\}$, in the depth layer move with the same motion vector $\mathbf{d}$, then each sample's movement is described by the following equation:

$$\mathbf{x}_i(t) = \mathbf{x}_i + \mathbf{d}(t - t_i). \tag{4}$$

At $t = 0.5$, a sample has a spatial coordinate:

$$\mathbf{x}'_\mathbf{i} = \mathbf{x}_i + \mathbf{d}(0.5 - t_i). \tag{5}$$

With this observation, we create a compact time-dependent shadow map by *reprojecting* all samples along the depth layer's average motion vector, $\mathbf{d}$, to $t = 0.5$ using Equation 5, and storing this layered, reprojected, shadow map in memory for use in the subsequent lighting pass. To perform a shadow lookup in the lighting pass, we offset this representation along the layer's motion vector to get the depth layer represented at a particular time.

4.1 Shadow Pass

Creating the shadow map representation involves a number of steps, which are covered in detail in this section.

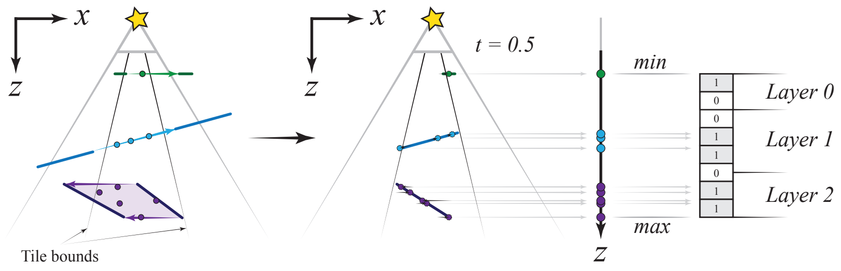


Figure 5: A simple scene illustrating the clustering approach that we use. The minimum and maximum depths, z, as seen from the middle of the shutter interval, $t = 0.5$, are found, and this interval is split uniformly. Each bin that contains at least one sample is marked with a 1. Finally, the resulting bit mask for the bins is used to find a small set of depth layers.

Visibility sampling First, the scene is stochastically rasterized with N samples per pixel in (x, y, t) light space. For each sample, we store depth and motion vectors. The motion vectors are comprised of the shadow map texture space motion in xy and depth motion in z.

Depth clustering Next, the samples in a tile in shadow map texture space are clustered in depth to obtain a set of depth layers of samples. To find suitable depth layers, we perform a simple depth clustering [10] step over all samples within a search window centered around each tile. We perform the layer split at the middle of the exposure interval, by offsetting the sample depths to $t = 0.5$ with their respective motion vector. Next, the depth range $[z_{min}, z_{max}]$ of the relocated samples is computed, which is then subdivided into uniform intervals. Intervals containing samples are flagged as *occupied*. Layer delimiters are then introduced where the largest stretches of *unoccupied* intervals are found. This process is illustrated in Figure 5. In our current GPU implementation we use 64 uniform intervals, which are clustered into (up to) four depth layers.

Per-layer motion We assume that the motion is slowly varying within each depth layer of the tile. We find a common representable motion vector, **d**, for the layer by averaging the motion vectors of the samples in the layer.

Grid setup The reprojection step builds upon previous work for motion blur filtering [92], with the difference that we work with depth values instead of color. However, unlike previous work, we reproject onto a stretched grid, which is aligned with the average motion direction, **d**. For clarity of presentation, we let the (x, y) coordinates have origin at the center of the tile. We parameterize the stretched grid with coordinates, (u, v), where the u-axis is aligned with the layer's motion vector, **d**, and the v-axis is perpendicular. As illustrated in the epipolar images in Figure 6,

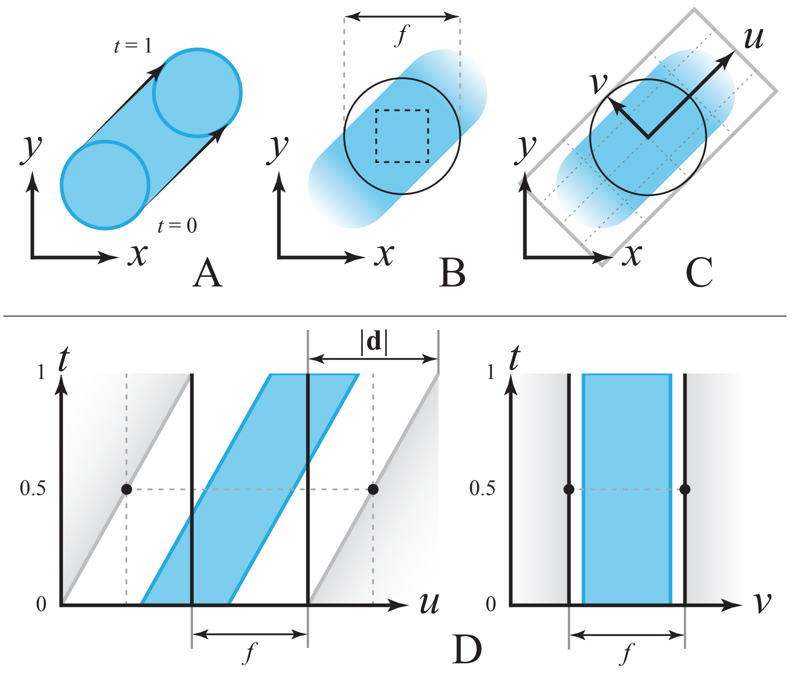


Figure 6: A: *a moving disc, viewed from the light source.* B: *we consider a single tile in the middle of the image. Samples from within the guard band will be used for this tile.* C: *the average motion direction is found, and a new coordinate system,* (u,v), *is constructed. The u-axis is scaled and aligned with the motion vector.* D: *the epipolar images for* $v = 0$ *(left) and* $u = 0$ *(right). The final, reprojected layer representation contains samples within the gray borders, which are derived from the guard band size,* f, *and the motion vector,* $\mathbf{d}$.

any sample originating from within the *guard band*, *f*, of a tile may reproject anywhere within a region that is $f + ||\mathbf{d}||$ wide. Therefore, we use a grid scaling factor of $\frac{f}{f+||\mathbf{d}||}$ along the u-direction, which ensures that no samples will be reprojected outside our scaled grid. We now define a rotation and scaling transform, M, for each layer, such that $M\mathbf{d} = \left(\frac{f||\mathbf{d}||}{f+||\mathbf{d}||}, 0\right)$. If we apply this transform to a moving sample: $\mathbf{x}_i(t) = \mathbf{x}_i + \mathbf{d}(t - t_i)$, cf. Equation 4, we obtain the corresponding sample in the stretched grid as:

$$(u_i(t), v_i) = M\mathbf{x}_i + \left(\frac{f||\mathbf{d}||}{f+||\mathbf{d}||}, 0\right)(t - t_i). \tag{6}$$

In presence of motion, the grid stretches outside the tile bounds, as shown in Figure 6D. The scaling factors of M used in our implementation are discussed further in Section 5.

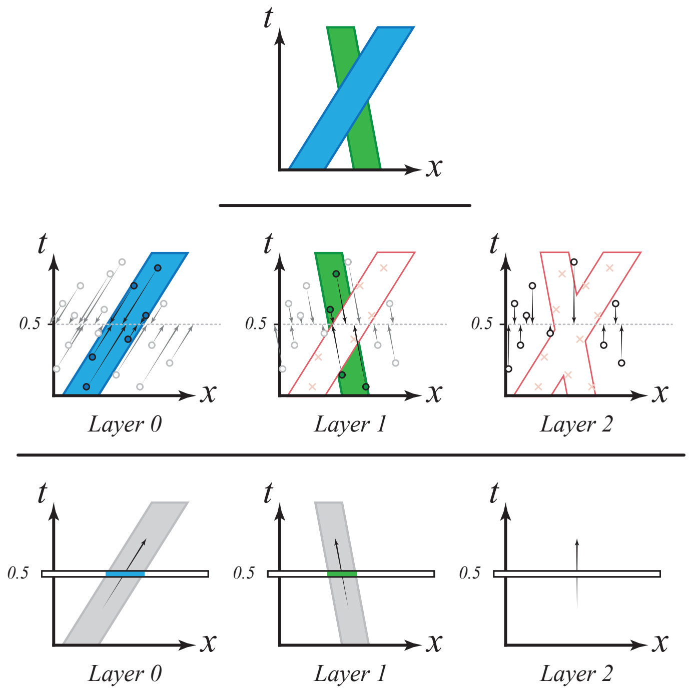


Figure 7: Top: an epipolar image of a simple scene with two objects and a static background layer. Middle: the samples are reprojected to $t = 0.5$ for each layer, using the layer's motion vector. Samples within a layer (black outlines) increase the opacity. Samples behind the layer (gray outlines) (farther away from the light source) decrease the opacity. Samples in front of the layer (red crosses) are discarded. Bottom: the shadow map can be queried at different times using the reprojected samples at $t = 0.5$ for each layer along with the layer's motion vector

Sample reprojection For each layer, each sample is transformed to the local coordinate system and is moved along the layer's motion vector to the middle of the shutter interval, by using Equation 6 with $t = 0.5$. Additionally, the depth of the sample at $t = 0.5$, $z_i^{\text{reproj}} = z_i + (0.5 - t_i)d_z$ is computed. The sample's position in *uv*-space maps to a texel location.

The next step is to compute the coverage and depth contribution of the sample to the texel. We track four quantities which are used for filtering later in the lighting pass described in Section 4.2. First, we need the depth value for the shadow test. We use the filterable variance shadow map (VSM) representation with the first and second depth moments (z and z^2) [29]. The next quantity is the opacity, α, which tells us how much each texel in each layer should contribute to the final shadow result. The remaining quantity is the weight, w, of the filter kernel used in the reprojection. Each sample will contribute with different α_i and w_i values for each layer. For each texel, (u, v), in the shadow map, these quantities are accumulated into a tuple with four elements on the form:

$$T(u,v) = \left(\sum w_i \alpha_i z_i,\ \sum w_i \alpha_i z_i^2,\ \sum w_i \alpha_i,\ \sum w_i\right), \tag{7}$$

i.e., a weighted sum of the first and second depth moments, a weighted opacity, and the total weight.

The values of w_i and α_i for the current layer are calculated as follows. If a sample lies in or behind the current layer (i.e., farther away from the light source), then w_i has a non-zero value based on the filter used. Otherwise, $w_i = 0$ (i.e., the sample does not contribute to this layer). In our implementation we use a box filter, and thus w_i corresponds to the number of samples falling in a pixel. The opacity value is one ($\alpha_i = 1$) if the sample lies within the layer, and is zero otherwise. The idea behind this is that if a sample that belongs to a background layer is visible through the foreground layer, then the foreground layer must be transparent for that sample. Since fewer samples affect layers farther back, this implies that the opacity estimate is better for foreground layers. This accumulation strategy is discussed in further detail in Vaidyanathan et al.'s [115] reconstruction work. Figure 7 illustrates a simple example of the reprojection process.

4.2 Lighting pass

In the lighting pass, the scene is rendered from the camera using stochastic rasterization [6], and we search for the amount of light that reaches a *receiver* sample, $(\mathbf{x}_r, z_r, t_r)$. For every receiver sample, the corresponding tile in the shadow map is found, and the visibility contribution of its layers are combined to a final shadow term.

Our shadow map is compactly represented as a set of tiles with a set of layers at $t = 0.5$ with the accompanying coordinate transforms, M. To retrieve a shadow map value for a particular layer at receiver time t_r, the receiver sample is reprojected using Equation 6 with $t = 0.5$. The reprojected coordinate maps to a location in the shadow map.

Furthermore, we account for the camera filter footprint when performing a shadow map lookup. Since the camera, light, and receiver point may move, this is an anisotropic footprint in xyt. We make the assumption that the receiver point is static in camera space for a short duration around the receiver sample time t_r. The duration is inversely proportional to the number of samples per pixel, N, used in the lighting pass. The camera filter footprint is approximated by transforming the receiver point to light space at times $t_r - \frac{1}{2N}$ and $t_r + \frac{1}{2N}$. Figure 8 shows two examples of how footprints are computed. Should the footprint stretch outside the layer's allotted region in the shadow map, it is clamped to avoid fetching invalid data from a neighboring tile. The size of the guard band, f, used in the coordinate transform M, determines how far outside the original tile the footprint may stretch, as illustrated in Figure 6.

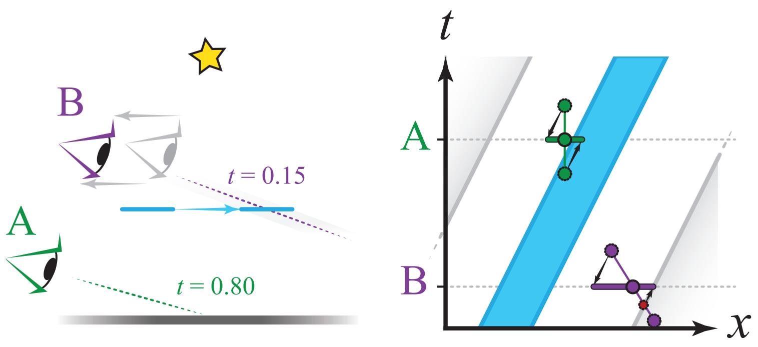


Figure 8: Two examples of how we calculate the filter footprint for a moving layer. In case A*, the camera and the light are static, and thus the filter in the light space epipolar image is a vertical line. Reprojecting the filter end points along the motion vector gives the final filter footprint. In case* B*, the camera is moving in the opposite direction to the layer motion and thus produces a slanted trail in the light space epipolar image. The footprint stretches outside the region that can be reconstructed for this layer, and is clamped. In these examples, we use* 4 *samples per pixel, and thus have filter footprints stretching over $\frac{1}{4}$ of the time interval.*

The shadow map location, along with the footprint axes as gradient vectors, are used in the hardware anisotropic filtering to retrieve a tuple on the form given in Equation 7. From this, we derive:

$$\bar{z} = \frac{\sum w_i \alpha_i z_i}{\sum w_i \alpha_i}, \quad \overline{z^2} = \frac{\sum w_i \alpha_i z_i^2}{\sum w_i \alpha_i}, \quad \bar{\alpha} = \frac{\sum w_i \alpha_i}{\sum w_i}. \tag{8}$$

With $\bar{z}$ and $\overline{z^2}$, we compute a *visibility term*, V, using a standard variance shadow map (VSM) test [29] with two moments. We base the test on the receiver point depth moved to the reprojected shadow map time $z_r^{\text{reproj}}(t_r) = z_r + (0.5 - t_r)d_z - b$, where t_r is the receiver sample time and b is a VSM shadow bias term. It should be noted that moving the receiver sample to $t = 0.5$ is equivalent to moving the depth of the shadow casting sample to the time of the receiver sample, t_r. Given $\bar{z}$ and $\sigma^2 = \overline{z^2} - \bar{z}^2$, the variance shadow map visibility is computed as follows [29]:

$$V = \frac{\sigma^2}{\sigma^2 + (z_r^{\text{reproj}} - \bar{z})^2}. \tag{9}$$

Combined with the opacity of the layer at the point of lookup, $\bar{\alpha}$, we can approximate the visibility of the receiver point through this particular layer as:

$$V_l = 1 - \bar{\alpha}(1 - V). \tag{10}$$

The visibility through all layers is accumulated using $V_{total} = \prod_l V_l$ to get a final visibility approximation.

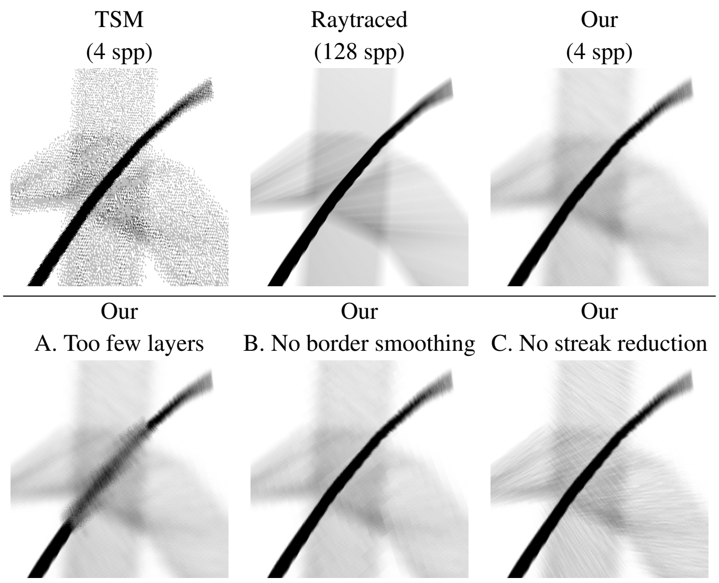


Figure 9: This figure shows the need for the various parameters and additional smoothing used in our algorithm. The camera is positioned so that the shadowed region in the red (top) inset from Figure 1 is viewed closely from above (instead of from an angle). Three tentacles are moving in different directions and create complex shadows on the receiver floor. From the raytraced reference, the detailed geometry from the suction cups are visible as they produce a striped pattern in one of the shadow layers. We use the same sample set for TSM *and our algorithm to create the shadow. In inset* (A)*, we have reduced the number of maximum layers from* 4 *to* 3*, and subsequently a single depth layer is created from two of the tentacles, with an incorrect shadow as the result. Similar artifacts are visible when the clustering step fails to produce proper splits between layers with vastly different motion and/or depth characteristics. Inset* (B) *shows our algorithm without the stochastic neighbor selection enabled, leading to visible tile boundaries. Finally, in inset* (C)*, the v-direction filter for large motion is disabled, with some streaking artifacts as a consequence. In this particular case, note that there are some streaks in the reference image, which we slightly over-blur with the v-filter enabled. Also note that, in general, one might want to increase the shadow map resolution and the number of samples per pixel to increase the quality further.*

5 Implementation

We implemented our algorithm and TSM in a GPU software stochastic rasterizer, similar to McGuire et al. [82], which we have extended with time-dependent shadow maps (TSMs) [6] and faster coverage tests [67].

Apart from the tile size and guard band used to derive the scaling factors in the coordinate transform, M, in Equation 6, an additional parameter, o, is used, such that:

$$(s_u, s_v) = \frac{o}{f}\left(\frac{f}{f+||\mathbf{d}||}, 1\right), \tag{11}$$

where s_u and s_v are the scaling factors in the matrix, M. Here, o controls the resolution of the output grid, and thereby also the resolution of the shadow map. In our implementation, for each 8×8 pixel tile in the input depth and motion maps, we use an output shadow map tile size $o = 16$, and a search region $f = 16$. Or more intuitively, in absence of motion, for each input pixel in an $f \times f$ neighborhood around each tile, $\frac{o}{f}$ output texels are produced. As motion increases, the output grid grows proportionally, while its resolution in the shadow map is retained, essentially trading spatial detail for covering a larger volume in xyt, as illustrated in Figure 6.

During the lighting pass, described in Section 4.2, we may sometimes end up with a total weight of zero, $\sum w_i = 0$. This happens when the shadow map density is too sparse compared to the filter (or size of motion), and is more likely to happen for layers further back as most of the samples will be caught by the layers in front. We alleviate this problem by expanding the filter size until a valid sample ($w_i \neq 0$) is included. This is similar to how the problem is solved in previous work [92]. They use an exponential filter and therefore samples further away can be used if there are no local samples, and giving them an insignificant weight if local samples exist.

Furthermore, we note that Munkberg et al. [92] exhibit some streaking artifacts in regions with large motion and low circle of confusion. In order to reduce such streaking artifacts, we apply a small filter aligned with the v-axis in such tiles. In our current implementation and selection of scenes, we found that a small tent filter with pixel width $w_v = \max(0, 0.05\,(||\mathbf{d}|| - 2))$, where $\mathbf{d}$ is the average motion vector, works well in practice.

In areas with varying motion vectors, we may get tile artifacts due to the rotated grids not aligning at tile borders. We remedy this using an approach similar to the one proposed by Guertin et al. [47]. When performing a shadow map lookup, we compute a probability based on the distance between the lookup position and the tile border. We then use that probability to stochastically perform the lookup in either the current tile or the neighboring tile. This means that the region in which a valid shadow test can be performed for a tile in somewhat increased, i.e., it depends on that a sufficiently large f-parameter is selected. In practice, we found that using a linear ramp starting at $1/3$ from the tile center and going up to 50% probability for selecting a neighbor at the tile border produces visually pleasing results, and works well with the selected $f = 16$.

In Figure 9, the benefits of border smoothing and streak reduction are shown. In addition, we show the quality impact of using too few layers.

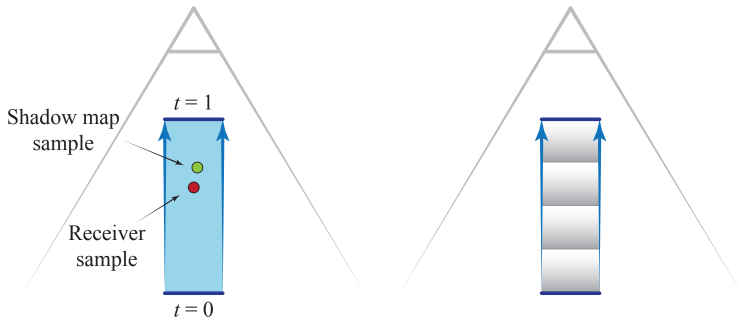


Figure 10: An illustration of a self shadowing issue with TSM, where a surface is moving towards the light source. The surface should be fully lit since it is visible from the light source throughout the animation. Left: the shadow map lookup for the receiver sample (red) returns the closest shadow sample (green) in (x, y). Since the shadow sample is closer to the light source than the receiver sample, it will falsely *report that the sample is in shadow. Right: within each time slice, the probability of encountering an occluder sample increases with the distance to the light source. In the resulting image, the time slices are visible as striped self-shadows. Our dynamic bias decreases the likelihood of such events.*

Memory Consumption The memory requirement for the shadow map depends on the number of layers, the output tile sizes, and the input depth and motion map resolution. In addition, for each tile and layer, we store the transform M using two values (scale and direction of major axis), the motion in z and the layer split position. We use 32 bit floats for the four tuples in Equation 7, and 16 bit floats for the tile data. For a 1024^2 pixel shadow map at 4 spp, the total cost amounts to 90 MB per depth layer. TSM stores one depth value for each sample, and the same input consumes 16 MB. It should be noted that we have optimized our algorithm for speed rather than size. There are, however, several avenues for reducing the memory usage. For example, using ESM instead of VSM, reducing the precision of the opacity and weight in Equation 7, or dynamically allocating output tiles. We leave this for future work since it would require further investigation on how these changes would affect the shadow quality.

5.1 Dynamic bias for TSM

In TSM, the exposure time is partitioned into N slices, each containing a subset of the samples. When a shadow test is performed for a sample at a particular time, the corresponding enclosing pixel and time slice in the shadow map is found. However, due to the discretization, there is a discrepancy between the receiver sample time and the shadow map sample time. This difference may be as large as $\frac{1}{N}$ of the exposure time, and can lead to self shadowing artifacts, as illustrated in Figure 10. In some cases, a shadow map bias alleviates the problem, where the bias

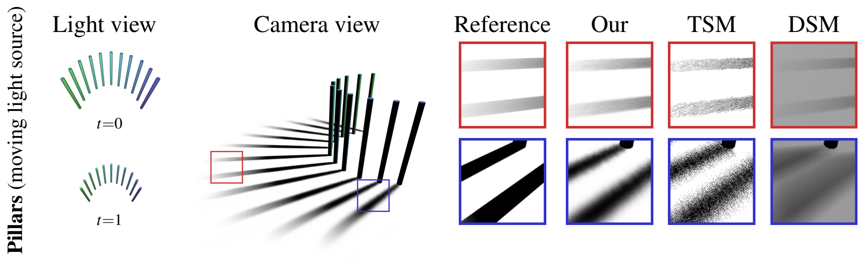


Figure 11: Shadows are produced by a light source, moving upwards, away from the pillars. This scene is difficult since points on the receiver plane that remain static viewed from the camera travel long distances in light space. Due to perspective, the sample paths through light space are non-linear. Some over-blurring is apparent, but our linear motion approximation works reasonably well. TSM *also exhibits some additional blurring. When a shadow map lookup is performed, the camera sample is transformed to light space at the camera time t_c to obtain the lookup coordinates. The time t_s for the sample at this location is (almost) always different from t_c. During this time discrepancy, the light source may have moved arbitrarily, potentially resulting in a poor match between the camera and shadow map samples in xyt-space.* DSM *does not account for moving light sources either, and get severe self-shadowing artifacts.*

incorporates the distance in depth that any object travels. This quickly becomes problematic, since contact shadows will disappear when the bias is increased.

As discussed in Section 4, in our algorithm, we compensate for the discrepancy in time between the receiver sample and the shadow map sample. Depth motion is accounted for by moving the samples along the average motion vector's depth component based on the sample time difference. We improve TSM in a similar way, but use the samples' own motion vectors. The receiver point is translated to the shadow map sample's time to get a new receiver depth:

$$\dot{z}_i = z_i + d_{z_i}(t_r - t_i), \tag{12}$$

In Section 6, we will show how this adjustment improves image quality of TSM and makes motion blurred shadow map rendering more robust.

6 Results

We have constructed a set of scenes, shown in Figures 11, 12 and 13, to test difficult cases, where both the receiving and shadow casting geometry move relative to the light source in various ways. The pillar scene illustrates the difficult case with a moving light source. Although the pillars are static as seen from the camera, they move a large distance in light space. The other two scenes show variations of moving receivers and shadow casters.

To evaluate the quality and robustness of our algorithm, we compare the quality against time-dependent shadow mapping (TSM) and deep shadow mapping

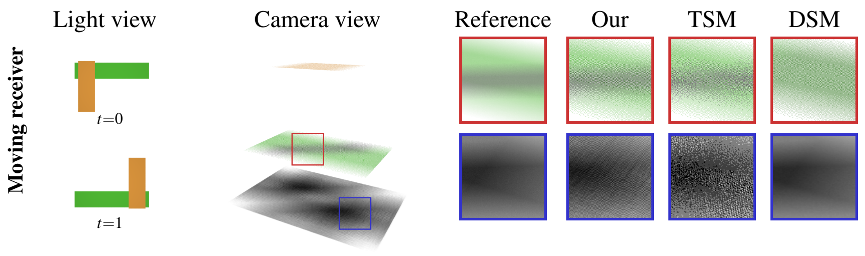


Figure 12: Two rapidly moving quads at different depths with motion directions orthogonal to each other. Both our algorithm and TSM *are able to capture the diagonal shadow stripe that appears on the green, moving receiver, whereas* DSM *does not.*

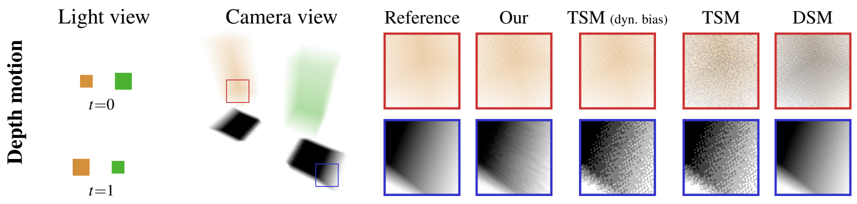


Figure 13: Two quads with perspective motion in light space, one moving towards and the other moving away from the light source. Our algorithm performs well, even though there is a high degree of motion along the z-axis, which violates our assumption of linear sample paths. With our dynamic bias from Section 5.1, TSM *does not exhibit self-shadowing artifacts. The bias in* DSM *cannot be fixed with an increased bias value, since the movement of the quads is larger than the gap between them and the ground plane.*

(DSM), as well as against the reference images rendered using the Embree [118] ray tracing kernels with 128 stratified rays per pixel. For the quality comparison, we use four samples per pixel for our algorithm and TSM. To accentuate the quality differences, we use fairly low resolution shadow maps of 512^2 pixels. An input tile size of 4×4 pixels was used for our algorithm, with the parameter settings $f = 12$ and $o = 8$. We implemented DSM by sampling the scene with 128 samples per pixel (spp) from the light source. This way, we can build a high quality visibility function for each shadow map pixel. Lokovic and Veach [76] suggested using a lower sampling rate and computing the visibility function by filtering over a larger footprint in the shadow map. However, the size of the filter is not well described, and therefore we chose to use a high quality (but inefficient) DSM implementation, which illustrates the algorithm's asymptotic behavior at high sampling rates.

The results of our quality evaluation can be seen in Figures 11, 12 and 13. DSM has problems with self-shadowing due to the assumption of a static receiver. Furthermore, the shadows from the pillars become smeared in Figure 11. The same behavior can be seen in the depth motion example in Figure 12, where a mov-

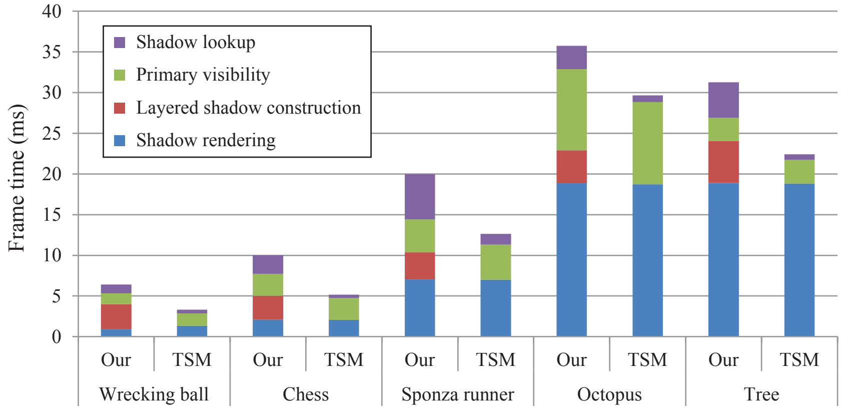


Figure 14: The number of milliseconds spent in each of the algorithms steps for our approach and TSM. All results are measured on NVIDIA GTX 980.

ing object incorrectly causes self shadowing. The scene has two moving objects, where the orange object cast a shadow on the moving green object, and both objects shadow a static ground plane. As can be seen, the shadow on the moving receiver is washed out for DSM. The correlation between the receiver and shadow caster must be captured to faithfully recreate the shadow for these cases. In all scenes, our algorithm and TSM produce results similar the reference. However, at equal samples per pixel and shadow resolution, our algorithm has considerably less noise.

In addition to the stress test scenes, we use another set of test scenes, shown in Figure 15, which have substantially richer geometrical detail and much higher occlusion complexity. We focus on evaluating performance of our shadowing algorithm compared to TSM, since the other competing algorithms collapsed samples over time, which generated severe artifacts as shown in Figures 11, 12 and 13. All results were measured on an NVIDIA GTX 980. We include both an *equal input* and an *equal time* comparison. For *equal input*, we use 4 spp both for the primary visibility and the shadow map. The resolution of the shadow maps for both algorithms is 1024^2 for all test scenes. For the *equal time* comparison, when constructing the shadow map for TSM we used 8 spp and increased the resolution to fit within the same frame budget. For low complexity scenes, such as Wrecking ball (8k tris), TSM achieves slightly better quality at equal time. However, at higher polygon counts, the cost of stochastic rasterization is non-negligible, and our algorithm produces better quality shadows at equal time. This is visible in the Sponza runner scene and in Figure 1. The Tree scene has a moving light source, which makes it difficult to get an accurate result with TSM, although having more time slices in the shadow map alleviates the problem. The image resolution is 1280×720 for all scenes.

Figure 14 shows timing for the different steps in our algorithm and TSM. Although we store motion vectors when rendering from the light (visibility sampling), the cost for that step is very similar for both algorithms. Our algorithm then converts the incoming depth and motion buffers to its layered shadow map representation, which takes 3 – 5 ms. The cost of stochastic rasterization from the camera is roughly the same for both algorithms. However, the shadow lookup and shading pass are more expensive with our algorithm as we perform filtering in this step.

Finally, in the accompanying video, we show the temporal behavior of our algorithm.

7 Conclusions

We have presented a novel time-dependent shadow mapping algorithm, which supports high-quality filtering and accurately handles time dependencies between shadow casters and receivers. Our algorithm has real-time performance for reasonably complex scenes and scales with the number of samples, rather than geometrical complexity. This can be seen in that our best results were obtained with the most complex scenes.

For future work, we note that the weakness of our algorithm is when the depth complexity is too high or when there is large local motion variation within a small depth range. The clustering algorithm is crucial for both performance and quality, and we would like to explore clustering using additional attributes, apart from depth, such as the direction and magnitude of the motion vector. However, this is a non-trivial extension, which requires a solution to intra-layer visibility if layers have overlapping depth ranges.

Acknowledgements

We thank the anonymous reviewers and the Advanced Rendering Technology (ART) team at Intel. We also thank David Blythe and Chuck Lingle for supporting this research. Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg foundation.

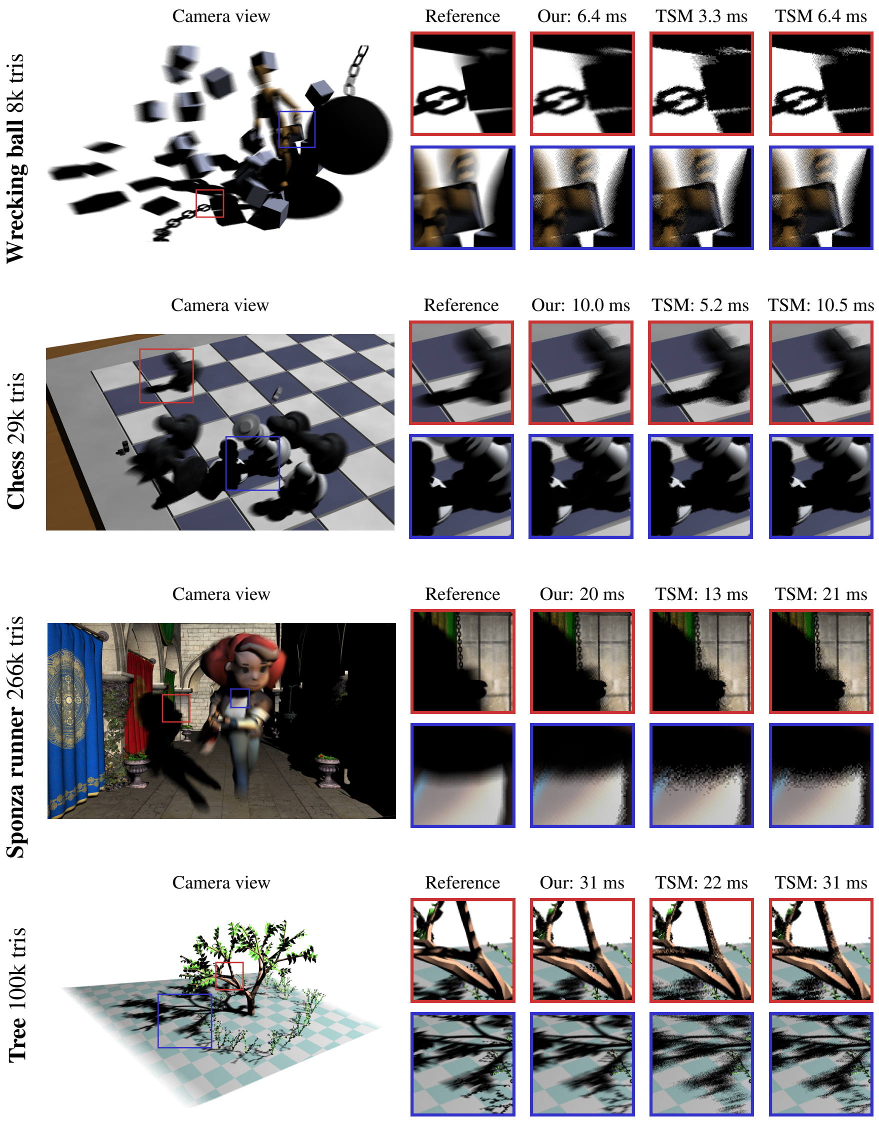


Figure 15: Our algorithm has significantly less noise compared to time-dependent shadow maps (TSM) and handles complex occlusion. The animated wooden doll model in the wrecking ball scene is from the Utah 3D Animation Repository. The chess scene was created by Rasmus Barringer. The Sponza model was made by Frank Meinl at Crytek, and the skinned runner model by Björn Sörensen. The tree scene with a moving light source was created using Autodesk Maya.

Paper VI

Adaptive Texture Space Shading for Stochastic Rendering

Magnus Andersson Jon Hasselgren Robert Toth Tomas Akenine-Möller

Lund University Intel Corporation

Abstract

When rendering effects such as motion blur and defocus blur, shading can become very expensive if done in a naïve way, i.e. shading each visibility sample. To improve performance, previous work often decouple shading from visibility sampling using shader caching algorithms. We present a novel technique for reusing shading in a stochastic rasterizer. Shading is computed hierarchically and sparsely in an object-space texture, and by selecting an appropriate mipmap level for each triangle, we ensure that the shading rate is sufficiently high so that no noticeable blurring is introduced in the rendered image. Furthermore, with a two-pass algorithm, we separate shading from reuse and thus avoid GPU thread synchronization. Our method runs at real-time frame rates and is up to 3× faster than previous methods. This is an important step forward for stochastic rasterization in real time.

Computer Graphics Forum (Proceedings of Eurographics), vol. 33, no. 2, pp. 341–350, 2014.

1 Introduction

Stochastic rasterization [6, 27] continues to be an interesting path forward for realistic camera models for real-time rendering. Camera effects, such as motion blur and depth of field (also called defocus blur), are favorably expressed with this technique. For good quality, the number of samples per pixel often needs to be high, and due to motion or lens effects, standard multi-sampled anti-aliasing (MSAA) will often deteriorate to super-sampled anti-aliasing [88]. This, in turn, means that shading becomes prohibitively expensive. Hence, better methods are needed for shading in a stochastic rasterizer.

To this end, new hardware mechanisms for reusing shaded values for many different rasterized fragments in a stochastic rasterization setting have been proposed [22, 24, 102]. Common to all of these approaches is that they assume that shading is constant for a certain point on a surface, regardless of motion and the lens. Note that this assumption is heavily used even in high-quality production renderers [27].

In contrast to new hardware mechanisms, Liktor and Dachsbacher [74] present a method that works on current graphics hardware. They use a compact geometry buffer, which stores shading information independent of visibility. When computing and storing new entries in this buffer they rely on per-fragment synchronization and atomic counters. While our method has the same goals, i.e., reusing shaded values as efficiently as possible, it has been designed without high frequency synchronization. The contributions of our work are summarized below:

- A lock-free (no atomics) method for reusing shading values in a stochastic rasterizer.
- Sparse and adaptive evaluation of shading (triangles are shaded directly into a hierarchical data structure).
- An algorithm well suited for current graphics processors.
- Results show that our method is substantially faster than previous methods.

We believe that our method brings us one step closer to high-quality real-time rendering of stochastic effects on current graphics hardware.

2 Previous Work

The idea of decoupling shading rate from visibility sampling [27] is old and has frequently been used in offline or photo-realistic rendering systems to improve performance. In the following, we will focus mainly on the more recent approaches targeting real-time performance and GPU implementations. The idea of a shading cache is illustrated in Figure 1.

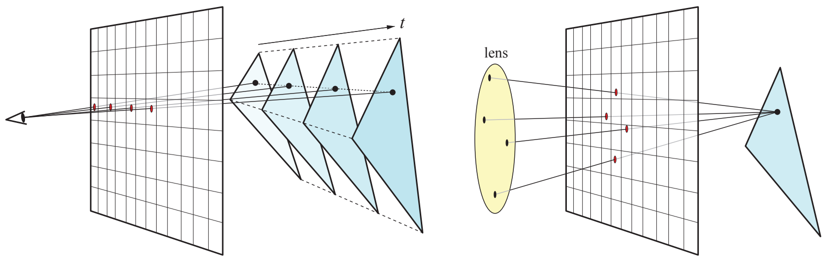


Figure 1: Caching shaded values works for both motion blur (left) and defocus blur (right). The assumption, used by Cook et al. and many others after them, is that the shading at a certain point on a triangle is independent of time and lens position. Hence, shaded values can be cached and later reused.

Many of the shading caches adopted for GPU implementation target temporal reprojection of shaded colors [94, 109]. While there is some overlap with our work, we only wish to reuse shaded values over different stochastic samples, and may therefore work with parametric texture space shading. In contrast, temporal reprojection algorithms typically work with screen space reprojection between frames and those techniques are thus not well suited for exploiting coherency between samples in a single frame.

As our algorithm works in object- or texture-space, it bears some resemblance to light mapping [5], which has been a popular technique (primarily) for baking static global illumination for computer games. Illumination maps were used by Arvo [14] for storing the result of tracing rays from the light sources, and Heckbert [56] stored radiosity in textures, which adapted their size to the shaded content. Some similarities can be seen between our approach and GPU accelerated light map generation [78], although those algorithms are typically more complex as they are targeted at solving light transport through the scene. However, a big difference between light map generation and our shading approach is that light maps are typically view-independent and should be suitable for viewing from any camera position. In contrast, since we will recompute the shading every frame, we can use the camera parameters to choose an appropriate sampling rate and texture filtering footprint. Texture space shading has also previously been used by Borshukov and Lewis [19] for realistic skin rendering.

Recently, several decoupled shading approaches targeting real-time rendering and graphics hardware [22, 24, 102, 117], have been developed. However, these papers focus on new hardware implementations, and are not easy to implement in the context of current generations of GPUs and graphics APIs. Of particular interest to us is the work by Vaidyanathan et al. [117] in which they derive minimum shading rates for defocus and motion blur. Liktor and Dachsbacher [74] built on the work by Ragan-Kelley et al. [102] and presented a deferred shading cache approach. They implemented their algorithm using shader programs and showed

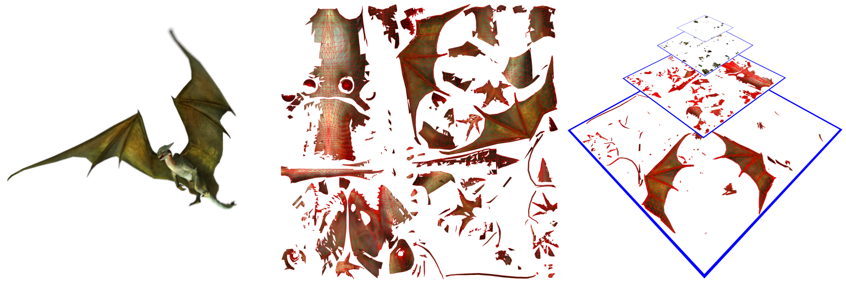


Figure 2: An illustration of our object-space texture shading approach. Left: the Dragon object rendered with defocus blur using our object-space shading approach. Middle: the mesh rendered with shading to the object-space texture after view-frustum and back-face culling (for the camera used in the left image). Right: the same mesh rendered to object space, with an appropriate mipmap level selected per-triangle in the geometry shader. Note that the image in the middle is there for illustration purposes only—our method renders only to the structure, called a shading texture, shown to the right.

performance gains on existing GPUs with stochastic rasterization. However, the algorithm requires per-sample synchronization when the shading cache is updated. This has a significant performance impact, and consequently the algorithm starts being beneficial only when expensive shaders are used.

Gribel et al. [44] use an object-space shading cache for offline rendering with semi-analytical methods. Similar to our approach, they also use a hierarchy and populate it as needed. They allocate a large chunk of memory for their cache, where subsets of the hierarchy are allocated lazily. Atomics must be used to avoid race conditions. Adapting this algorithm to run on the GPU would probably result in an algorithm similar to that of Liktor and Dachsbacher [74].

Although this paper focuses on the shading aspect, we have also implemented a stochastic rasterizer for evaluation purposes. Our implementation is very similar to that of McGuire et al. [82], but modified to perform shading texture lookups in the pixel shader rather than computing actual shading. We also use the five-dimensional sample-in-triangle test proposed by Laine and Karras [68], as well as the back-face culling test for motion and defocus blur by Munkberg and Akenine-Möller [90].

3 Algorithm

Our shading approach is split into two passes, called the *shading pass* (Section 3.1) and the *stochastic rasterization pass* (Section 3.2). In the first pass, illustrated in Figure 2, we set up a hierarchical, i.e., mipmap-like, *shading texture* for each object. We do this by rasterizing the current object directly into the shading texture using the object's parametric shading space (or texture space) coordinates. During this pass, we also compute how each triangle will project on screen during the succeeding stochastic rasterization pass, and use that information to dynamically select an appropriate mipmap level to render into. In the second pass, where we typically wish to use stochastic rasterization effects, such as defocus and/or motion blur, we simply fetch the color value from the shading texture for each fragment. The division of our algorithm into two passes is crucial to performance because it enables lock-free lookups of shaded values. This should be compared to previous methods that require fine-grained thread synchronization [74].

By switching from the "shade on miss" model of the original cache systems, we lose the ability to shade only fragments that are visible from the camera and risk overshading if we choose a texture resolution that do not match the screen resolution. Therefore, it is important not only to perform back-face and frustum culling when setting up the shading texture, but also to select a mipmap level in the shading texture that closely matches the screen space signal frequency (including blur from depth of field).

We typically use one shading texture per object. If the scene requires more textures than is possible to allocate, we keep a pool of a few shading textures and alternate between pass one and pass two. It is beneficial to merge smaller objects and use the same shading texture whenever possible, since that minimizes the involved state changes when alternating between the passes.

Next, the two passes of our algorithm are described in more detail.

3.1 Shading Pass

The shading pass consists of two sub-passes, described in Section 3.1.3, in which we sparsely populate a hierarchical shading texture with shaded fragments. The stochastic rasterization pass can thereafter reuse these shaded values several times. In contrast to other research dedicated to lowering shading rate for stochastic rasterization, we propose to pay the shading cost up front, prior to visibility determination, which is similar to how Reyes handles shading [27].

First, the mesh vertices need to be augmented with shading space coordinates, which must ensure that all surface points on the mesh can be uniquely mapped to a location in shading space. Texture coordinates are often already available and can double as the shading space so long as there are no overlaps. We would also like to point out that our algorithm can be used selectively, and alternative algorithms could be used for objects where non-overlapping atlases are an issue. Furthermore, like previous work [22, 24, 102, 117] all shading is calculated at the center of the lens and at a fixed time. The lack of view-dependent shading is widely regarded as an acceptable trade-off, given the complexity of the problem.

In Section 3.1.1 and 3.1.2, we describe the theory on how to conservatively determine the highest shading frequency required for any point on the primitive, in order to shade as sparsely as possible. Section 3.1.3 then describes how this is used in practice in order to generate the shading texture.

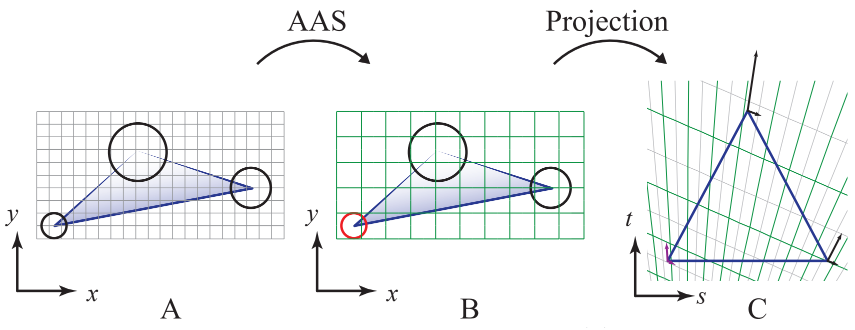


Figure 3: Shading rate algorithm outline. A: triangle to be rendered. B: we use the AAS algorithm to derive a screen space shading grid, or inverse shading rate, which is dictated by the minimum circle of confusion shown in red. C: the shading grid is projected into shading space (note that it is not aligned with the Cartesian grid).

3.1.1 Shading Rate

Our algorithm for selecting shading rate is illustrated in Figure 3. Starting from a triangle in screen space, we use a method, called *anisotropic adaptive sampling* (AAS), proposed by Vaidyanathan et al. [117]. Given a lens with particular characteristics, AAS can be used to determine the screen-space shading rate needed in order to capture a certain desired percentage of the frequency content due to defocus blur.

We use the thin lens model, where the (non-signed) screen-space circle of confusion (CoC) radius can be modelled as:

$$r_c(w) = \left| \frac{c_0 + wc_1}{w} \right|, \tag{1}$$

where c_0 and c_1 are parameters derived from the camera's aperture and focal distance, and w is the vertex depth. Note that the focus plane is located at $w = -c_0/c_1$, since $r_c(-c_0/c_1) = 0$. Similar to the derivation of AAS, we find the smallest circle of confusion, $\min_w r_c(w)$, as shown in Figure 3B, and use it to bound the shading frequency. If the circle of confusion is less than a pixel, or if the triangle straddles the focus plane, the frequency is bound by the pixel grid instead. This gives us a shading grid spacing, or inverse shading rate, expressed in pixels:

$$d = \max(a(\min_w r_c(w)), 1), \tag{2}$$

where the function a depends on the aperture of the lens. We use a circular lens in our models and use $a(x) = x/2$ [117]. For mipmap level selection, we only consider defocus blur, while the full AAS algorithm handles the combination of both motion and defocus blur.

Once we have established a suitable shading grid spacing in screen space (Equation 2), we wish to transform it into our shading space. This is a projective trans-

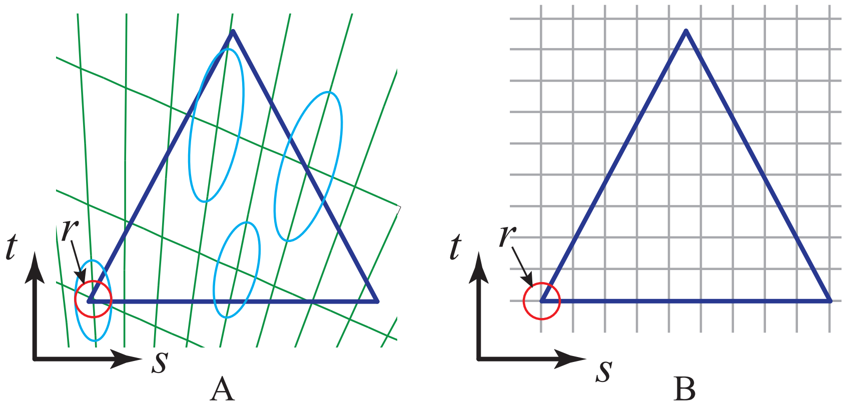


Figure 4: Finding the highest frequency in texture space. A: transforming the AAS grid into texture space gives us a projective grid that does not align with the Cartesian grid we use for our shading space. B: the resolution of our shading space corresponds to the smallest radius (red circle to the left) of the filter kernels used.

form, and we will therefore end up with a grid similar to that shown in Figure 3C. It should be noted that this grid is not aligned with the standard Cartesian grid, which we use for our shading space. However, if we assume that shading is filtered using EWA [55] (or using hardware-accelerated anisotropic filtering), we obtain the Gaussian filter kernels illustrated in Figure 4A.

The filter kernels can be modeled as ellipses [55], and are usually computed from the screen-space derivatives, $\frac{\partial \mathbf{T}}{\partial x}$ and $\frac{\partial \mathbf{T}}{\partial y}$, for a shading space texture coordinate, $\mathbf{T} = (s,t)$. We assume that the mapping $(x,y) \rightarrow \mathbf{T}$ is locally linear, and therefore the shading grid spacing computed using AAS (d from Equation 2) can be directly used to scale the screen-space derivatives: $d \cdot \frac{\partial \mathbf{T}}{\partial x}$ and $d \cdot \frac{\partial \mathbf{T}}{\partial y}$.

Given these derivatives, one may compute the minor axis of the ellipse as described in Appendix B. As shown in Figure 4A, the smallest minor axis:

$$r = \min_{x,y \in tri} R_{minor}(x,y), \tag{3}$$

of any ellipse over the whole triangle indicates the highest visible content frequency. It should be noted that this is a non-trivial function that is difficult to bound conservatively. We therefore conjecture that the minimum minor axis radius across a triangle occurs in one of the triangle vertices. Currently, we have no firm proof for that, but it matches previous knowledge that the mipmap level can be perspectively interpolated across a triangle in high quality [35]. Given the minimum radius, r, shown in Figure 4, we select the mipmap level as $\lfloor \log_2 r \rfloor$, and we end up with the grid shown in Figure 4B. This choice ensures that we shade densely enough to resolve details, while the actual filter footprint takes the non-discretized r into account.

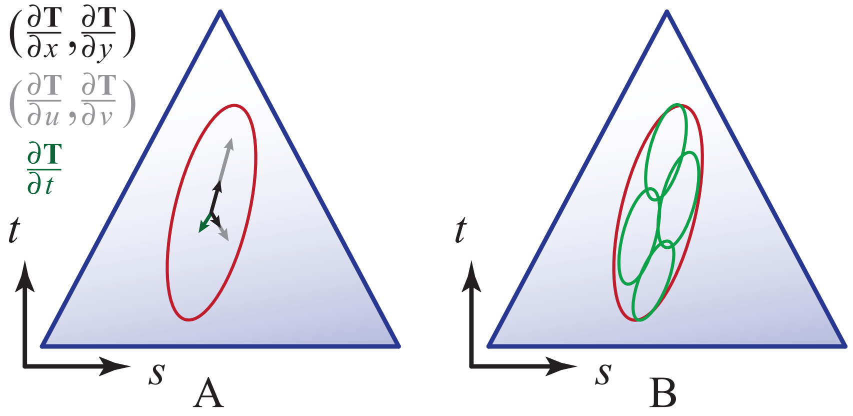

$\left(\frac{\partial \mathbf{T}}{\partial x}, \frac{\partial \mathbf{T}}{\partial y}\right)$
$\left(\frac{\partial \mathbf{T}}{\partial u}, \frac{\partial \mathbf{T}}{\partial v}\right)$
$\frac{\partial \mathbf{T}}{\partial t}$
t
s
A
t
s
B

Figure 5: A: a triangle and its corresponding shading space along with texture derivatives with respect to x, y, u, v, and t. Given these five derivatives, we generalize the method by Loviscach [77] to compute a single elliptical footprint, which takes all partial derivatives into account. B: depending on whether we access the shading texture once per rasterized pixel (MSAA) or once per sample (SSAA), we should pre-filter the shading texture with an appropriate footprint. The figure shows a pixel footprint used for MSAA as the red ellipse. If SSAA is used instead, we will get multiple lookups in the shading texture, illustrated as the green ellipses, and the footprints need to be scaled to reflect this.

In practice, the maximum shading frequency is limited by the finest mipmap level resolution. Having a low resolution shading texture and moving sufficiently close to an object may cause some blockiness, and therefore it is important to select a suitable maximum shading texture resolution for each asset. Selecting a resolution that is too high only affects memory consumption, as our shading system will pick an appropriate mipmap level for actual shading.

3.1.2 Filtering

After selecting an appropriate mipmap level for the triangle in the geometry shader, the triangle will be rasterized to the shading texture. The pixel shader is executed for every pixel in the shading texture that overlaps the triangle, in the selected mipmap level. In the pixel shader, we compute pre-filtered shading values, filtering over the footprint of each pixel with respect to (x, y, u, v, t), where (x, y) are the screenspace coordinates, (u, v) are the lens coordinates and t is the shutter time. In the pixel shader, we must maintain enough triangle data so that we can compute $\frac{\partial \mathbf{T}}{\partial x}, \frac{\partial \mathbf{T}}{\partial y}, \frac{\partial \mathbf{T}}{\partial u}, \frac{\partial \mathbf{T}}{\partial v}, \frac{\partial \mathbf{T}}{\partial t}$, in order to determine the filter footprint. We use a generalized form of the filter formulated by Loviscach [77], who only covered the case of screen space and motion blur filtering. The full derivation of how the generalization is done is found in Appendix A. The resulting combined filter has an elliptical kernel, which is used for computing the shading of the pixel.

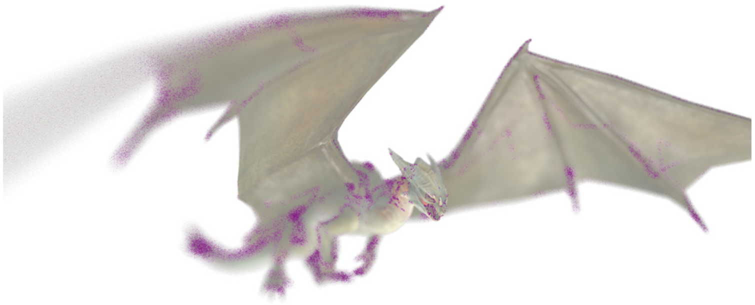


Figure 6: Without conservative rasterization, the regions around triangle edges in the shading texture may have incomplete information, which propagates to the final image as shown here (emphasized in purple color).

In practice, we let the hardware for anisotropic texture filtering effectively perform the filtering for us, which is in accordance with Loviscach's approach. An example of the combined filter for a sample position on a triangle with defocus and motion blur can be seen in Figure 5A.

We must make sure that the correct derivatives are used when pre-filtering, depending on how the shading texture is sampled in the subsequent stochastic rasterization pass, as illustrated in Figure 5B. If supersampling is used, several samples will be drawn from within the pixel, lens, and time footprints in the shading texture. The pixel, lens, and time derivatives must therefore be scaled prior to shading to ensure that the color value is correctly reconstructed. We assume that we have N evenly distributed sample points, and we therefore divide the (x,y) and (u,v) derivatives by $\sqrt{N}$ as this will distribute the pixel and lens area evenly among the samples. Similarly, the time derivative is divided by N since it is a one-dimensional attribute. For multisampling, we only wish to retrieve one shaded value per pixel, and thus $N = 1$. Regardless of what strategy we use for sampling the shading texture, we use the grid resolution of the selected shading texture mipmap level as a lower bound of the extent of the filter.

3.1.3 Populating the Shading Texture

As described in the previous subsections, we now have methods to compute an appropriate shading rate, i.e., mipmap level, per triangle. The actual population of the shading texture is done in two sub-passes. In the geometry shader of the first sub-pass, the triangles are conservatively view-frustum culled and are back-face culled with respect to the lens and time, using the method described by Munkberg and Akenine-Möller [90]. The surviving triangles are then rasterized to the shading texture using the shading space coordinates transformed to the selected mipmap level as the position attribute. In the pixel shading stage, we recompute the texture footprint and perform the surface shading.

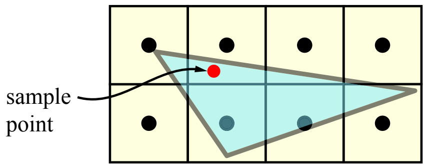


Figure 7: With standard rasterization rules, only the two middle pixels in the bottom row would be considered to be inside this triangle since those pixels' center locations are the only ones that are inside the triangle. However, our sample locations when looking up shading in the shading texture can be arbitrary inside the triangle. The red sample point, for example, is inside the triangle, but the enclosing pixel would not be rendered to with standard rasterization. We overcome this with conservative rasterization, which processes all pixels that are touched by the triangle (which in this case includes all 4×2 *pixels).*

Current graphics APIs do not allow dynamically selecting the output mipmap level, as mentioned in Section 3.1, and we therefore manually maintain our own mipmap hierarchy. Using this layout, we select a mipmap level for each triangle by scaling and offsetting the texture coordinates. This ensures that we can use a standard hardware accelerated bilinear texture lookup, instead of doing custom texture filtering in the pixel shader, when performing the stochastic rasterization pass (described in Section 3.2).

Using standard rasterization results in artifacts in the final image, which can be seen in Figure 6. The reason for this is that our sample locations are arbitrary within a triangle, while the rasterization hardware samples visibility at the pixel center. This is explained further in Figure 7. As described next, we use conservative rasterization [4, 52] in a second sub-pass to solve this problem.

In the second sub-pass, the same set of triangles are rendered with conservative rasterization. First, the culling and shading rate computations from the first pass are repeated. Next, we build the conservative outer hull [52] of the input triangle for the selected mipmap level. It is important that the shaded values that are already computed in the first sub-pass are not overwritten. To accomplish this, we use a depth buffer in both the first and the second sub-pass with the depth test set to *less than*. The first sub-pass writes a smaller depth value the second sub-pass. This guarantees that the formerly shaded values have priority over the latter. An example of this process can be viewed in Figure 8.

3.2 Stochastic Rasterization Pass

In the second pass, we use a stochastic rasterizer, similar to that of McGuire et. al [82], to render the final image. However, we want to point out that our shading algorithm is orthogonal to the rasterization algorithm. For example, we use optimized inside tests [68].

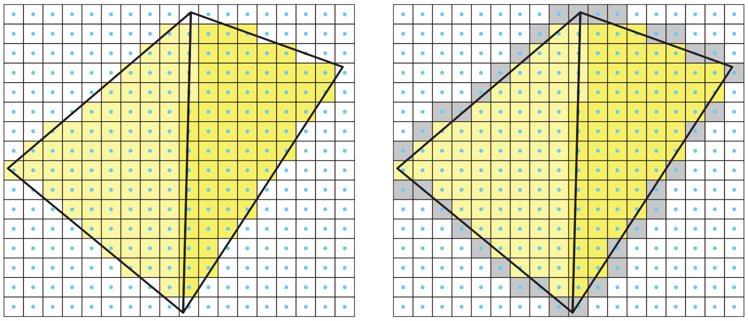


Figure 8: The inner regions are yellow and the visible parts of the outer regions are gray. Left: the inner regions of two triangles are rendered in the first sub-pass. The depth is set to z_i. Right: the outer regions of the triangles are rendered using conservative rasterization in the second sub-pass, with depth z_o, where $z_o > z_i$. Since the depth of the inner region is closer than the depth of the outer region, the gray pixels are only located around the triangles.

Since the shading texture was completely set up during the first pass, we do not need to handle cache misses and therefore the second pass becomes straightforward. Instead of traditional stochastic shading approaches [82, 88], we simply perform a texture lookup into the shading texture for each sample being rendered. We must use the exact same model as used in the first pass when computing the mipmap level, since only one level per triangle is populated with shaded data. The texture coordinates for the shading texture can be interpolated using the perspective-correct barycentric coordinates for a sample. The barycentric coordinates are computed as a byproduct of our sample-in-triangle inside test, which we also need to execute for each sample as part of the stochastic rasterization [6, 82].

By default, we use a per-sample frequency shader for the stochastic rasterization pass. However, as an optimization it is possible to run a per-pixel shader while still inside-testing each sample. In this case, we only need to perform a single texture lookup per pixel in an algorithm that mimics multisampling. However, due to API limitations, it is only possible to output a single depth value if the shader is executed on a per-pixel basis and this may lead to artifacts similar to the shading approach proposed by McGuire et al. [82]. The artifacts may be quite severe in some cases, as illustrated in Figure 9. However, the multisampling approach performs significantly better than supersampling approaches on current GPUs, and it may be valuable if performance is crucial.

After the stochastic rasterization pass has finished, the image is complete. Since the hardware is used to composite the framebuffer from all triangles in the second pass, blending works as in any forward renderer. Note that this is not possible with deferred shading methods, such as the one proposed by Liktor and Dachsbacher [74].

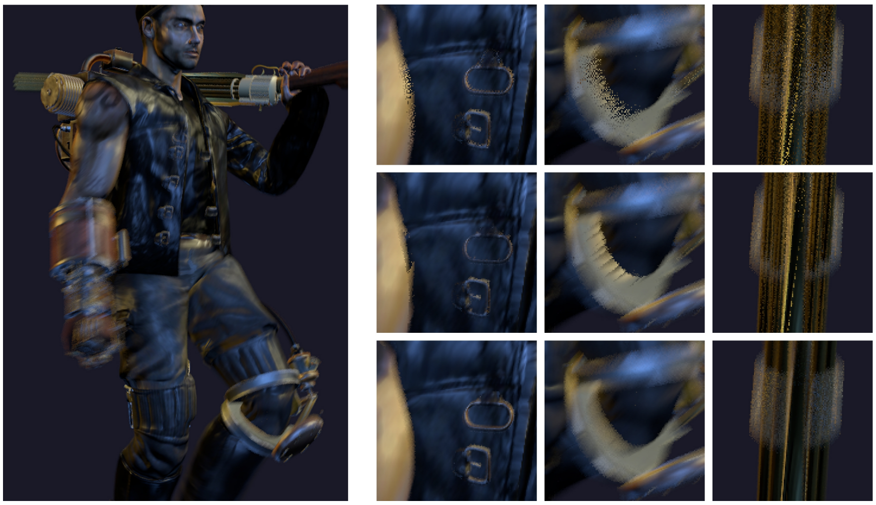


Figure 9: Examples of artifacts caused by writing a single depth compared to correct per-sample depth. The top row uses the depth of the last covered sample, the middle row uses the depth of the nearest sample, and the bottom row uses the correct per-sample depth.

4 Results

We compare our implementation to the deferred shading approach by Liktor and Dachsbacher [74] and stochastic rasterization with supersampled shading. The original stochastic rasterizer proposed by McGuire et al. [82] relied on multisampling, but it is easy to extend to supersampling and we used this as reference. When possible, we implemented both supersampled and multisampled versions of the algorithms, since multisampling is desired if performance is a primary concern. We use supersampling unless otherwise is indicated.

Our implementation of Liktor and Dachsbacher's algorithm differs somewhat from their description. In their implementation, a cache maintains *ssID*s, which uniquely identify a shading point in shading space, coupled with a particular primitive. In the geometry shader, each primitive is assigned a range of ssIDs, which is an operation that requires an atomic counter to avoid collisions. However, they implemented their algorithm using OpenGL 4.2, while we based all our algorithms on Direct3D 11, which lacks unordered access binding for the geometry shader stage. We remedied this limitation by adding a 32-bit `primitiveID` field to each cache entry to uniquely identify each shading point and primitive coupling. The hash function was also modified to offset the cache address based on the `primitiveID`. The cache size was set to 64k entries, and increasing this number did not result in significantly fewer shader executions. In our implementation, the shading rate of defocused triangles in Liktor and Dachsbacher's algorithm is also reduced using the AAS approach [117] and filtered with the generalized form of Loviscach's

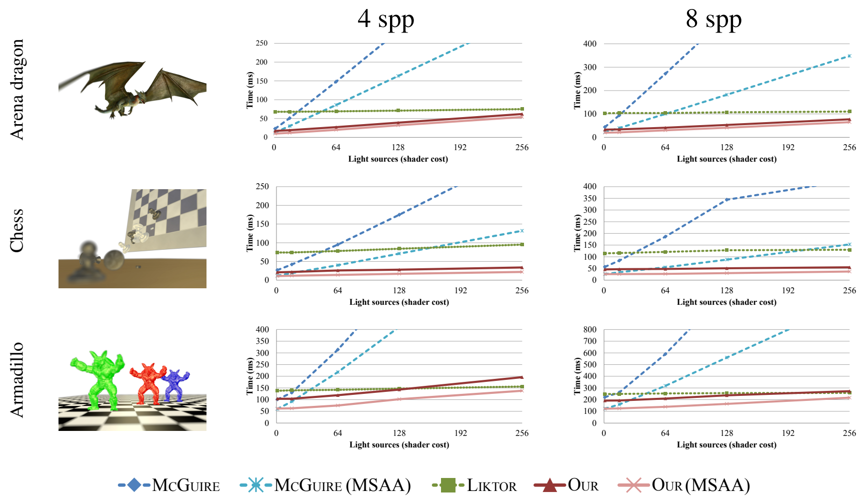


Figure 10: Results for our test scenes rendered with a variety of algorithms. In the diagrams, we present render times in milliseconds (lower is better) for 4 and 8 samples per pixel (spp), respectively. The algorithms called McGuire, Liktor, *and* Our *are directly comparable in terms of image quality. The algorithms with (MSAA) in their names rely on multisampling, and only do a single shader execution per pixel, with a single shading texture lookup. While this is beneficial for performance, we can only output a single depth value per pixel, and there is a risk of artifacts due to inaccurate depth test, as illustrated in Figure 9.*

method [77]. Our goal has been to put together an implementation that is as close as possible to Liktor and Dachsbacher's to make for a fair comparison.

To evaluate the performance, we measured the average rendering times for three different scenes with defocus blur. The *Arena dragon* scene contains one object with 74k triangles. The *Chess* scene is comprised of 12 objects with a total of 29k triangles. The *Armadillo* scene uses a high triangle count of 640k triangles. We can control shading complexity by computing shading using between one and 256 directional light sources. This allows us to study how the different algorithms scale with increasing shader cost. All images were rendered at 1920×1280 resolution. Throughout our experiments, the finest mipmap level resolution of the shading texture was set to 2048^2, which worked well for all of our test scenes.

All benchmarks were run on a PC with an Intel Core i7 965 CPU, 6 GB RAM and an NVIDIA 780 GTX GPU with 3 GB RAM (the benchmark application is completely GPU bound). Figure 10 shows the rendering time in milliseconds for the different scenes using $4\times$ and $8\times$ MSAA. We note that the algorithm by Liktor and Dachsbacher [74] is not affected as much by shader complexity, and at some point, their algorithm may be faster than ours. In the rightmost diagrams in Figure 10, Liktor and Dachsbacher's algorithm is faster at 256 light sources, and in

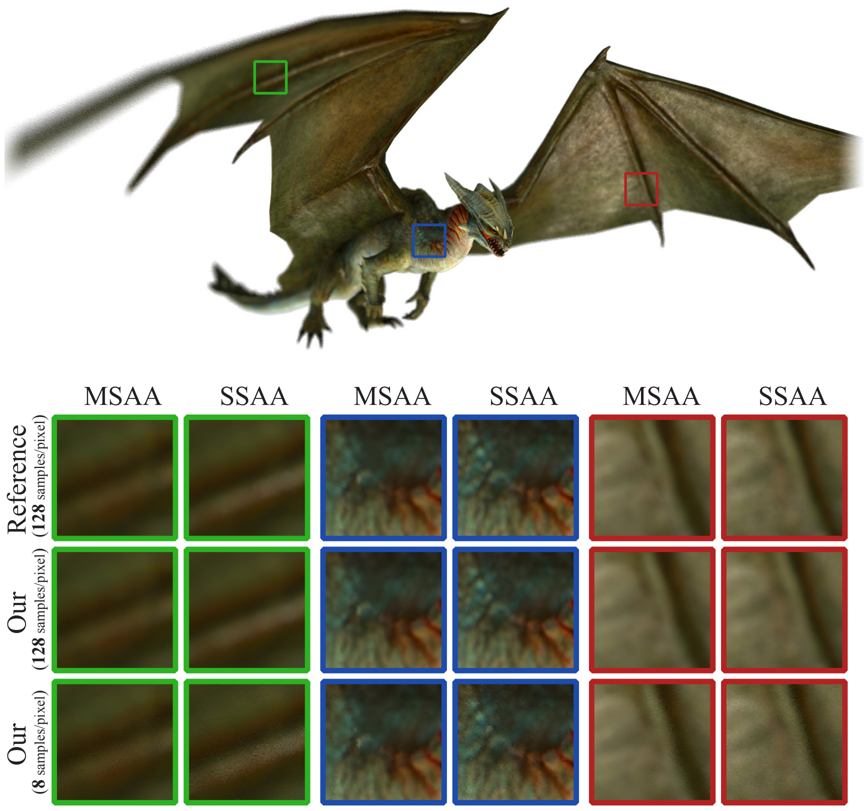


Figure 11: Image quality comparison between our algorithm with 8 and 128 samples/pixel and McGuire's algorithm with 128 samples/pixel. We ran the algorithms both with MSAA and SSAA enabled. Notice that the highlight in the green cutouts is less prominent for MSAA compared to SSAA. The blue cutout shows that we are able to maintain high quality, even when the defocus effect is close to zero. In the red cutout, we note that some texture detail is lost for MSAA, but this occurs both for our algorithm and for the reference. Overall, we find that the major difference between the images stem from noise due to the lower number of samples drawn.

the leftmost diagrams we estimate that it will be faster when using about 512 light sources. For reasonably complex shaders, our algorithm has performance characteristics that makes it very desirable for real-time rendering. The results are very encouraging since our algorithm outperforms the best competing algorithm by up to 3× in some cases.

The rendered image quality of our algorithm closely matches that of the reference solution. In Figure 11, we compare the image quality of McGuire's algorithm using 128 samples per pixel against our algorithm with 8 and 128 samples per pixel.

4.1 Timings by Render Pass

We have timed the different passes in our algorithm. Not surprisingly, the stochastic rasterization pass (Pass 2 Rast.), which includes the lookups in the shading texture, makes up for the bulk of the time. Even though MSAA speeds up the stochastic rasterization pass (Pass 2 MSAA) considerably, it is still a major part of the total rendering time. Roughly 80 – 90% of the time is spent on the inside tests within Pass 2, which means that the shading texture lookups are relatively inexpensive. When creating the shading texture, we note that the sub-pass performing conservative rasterization (Pass 1 Cons.) requires more time than the sub-pass doing normal rasterization (Pass 1 Shade) even though it writes to considerably fewer pixels. The reason for this is mainly that the geometry shader of the conservative rasterizer is quite expensive, and offsets whatever shading cost we gain from the many pixels failing the depth test. This is most apparent in the Armadillo scene as it has a very high triangle count, and we note that our algorithm would benefit greatly from faster algorithms for conservative rasterization [4]. Finally, we note that 1–2 ms of the frame time is spent in miscellaneous setup tasks, such as animation, frame buffer clearing, and multi-sampling resolve, which are not related to our texture space shading algorithm. The table below shows a breakdown of rendering times (in milliseconds), of the Dragon and Armadillo scenes from Figure 10 with 16 light sources for shading.

[ms]	**Dragon**		**Armadillo**	
	4×	8×	4×	8×
Misc	2.3	2.3	0.8	0.8
Pass 1 Shade	2.1	2.1	3.8	3.8
Pass 1 Cons.	2.3	2.3	11.9	11.9
Pass 2 Rast.	13.8	29.3	87.5	178.8
Pass 2 (MSAA)	(7.1)	(16.6)	(46.1)	(108.3)
Total	20.5	36.0	104.0	195.3
Total (MSAA)	(13.6)	(21.0)	(62.1)	(124.8)

5 Conclusions and Future Work

Shading cost needs to be substantially reduced in order to make stochastic rasterization a viable rendering method. To this end, we have presented a novel technique for reusing shading for stochastic depth of field rasterization on current GPUs, and shown significant performance improvements of up to 3×. In contrast to deferred shading methods, we also support blending. There are several interesting aspects that we would like to research further in the future. Liktor and Dachsbacher's [74] algorithm shades very sparsely due to their approach doing deferred shading, which means that they get full benefit of occlusion and never have to shade any occluded samples. However, it has to explicitly sync threads, which proved expensive, and therefore the method performs better only when very expensive shaders are used. It would be useful to investigate how our algorithm would perform with occlusion queries and/or software occlusion culling on the CPU [25], which is often done in mature game engines.

To minimize state changes for very large scenes, we would like to develop a system for dynamically allocating properly-sized shading textures (or using a part of a shading texture) on a per-object basis. This would not require any artist interaction and at the same time it would reduce state changes significantly, which would also increase the performance of our algorithm. Finally, we would like to extend the frequency analysis to motion blur and to the combination of motion and defocus blur. Currently, our system does not compute optimized shading rates for motion blur—however we would like to point out that our system can already handle these effects too, even though we may over-shade.

Acknowledgements

We thank the reviewers for their insightful comments, and thank David Blythe, Marco Salvi, Charles Lingle, Tom Piazza, and the Advanced Rendering Technology team at Intel for their support. We would also like to thank Jacob Munkberg, Petrik Clarberg, Jim Nilsson and Björn Johnsson for all the fruitful discussions, and Gabor Liktor for answering our questions about his research. Tomas is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation.

Appendix A - Filter Derivation

Loviscach [77] showed how to efficiently filter textures in time and space by extending elliptical weighted average (EWA) filtering to handle motion. We generalize this approach to handle Gaussian filter kernels in n dimensions. Like Loviscach, we make some local approximations about the various dimensions that we wish to integrate over. First, we assume that the texture coordinates are locally linear with regard to the augmented variables, which means that we locally ignore effects such as perspective distortion. EWA already uses this approximation in (x,y), using concentric ellipses in (s,t). Second, Loviscach also approximates (s,t) linearity in time, and we will do the same for all additional variables. Finally, we locally assume that all variables are independent of each other, again in line with Loviscach's work.

Sums of independent normal distributions are normal distributions. The Gaussian distribution in 2D is:

$$X = \begin{pmatrix} s_0 \\ t_0 \end{pmatrix} + \alpha \begin{pmatrix} \partial_x s \\ \partial_x t \end{pmatrix} + \beta \begin{pmatrix} \partial_y s \\ \partial_y t \end{pmatrix}, \tag{4}$$

where α and β are normal distributions with zero mean. Without loss of generality, we assume α and β are $N(0,1)$ distributions, as the variance can be chosen arbitrarily by pre-scaling the partial derivatives. If we augment with a number of additional independent distributions, we get:

$$Y = \begin{pmatrix} s_0 \\ t_0 \end{pmatrix} + \sum_i^n \gamma_i \begin{pmatrix} \partial_{X_i} s \\ \partial_{X_i} t \end{pmatrix}, \tag{5}$$

where $\gamma_1 = \alpha$, $\gamma_2 = \beta$, $X_1 = x$, $X_2 = y$, and all γ_i are again $N(0,1)$ distributions.

Next, we focus on the distribution around the point (s_0, t_0). Summing distributions is most easily accomplished using characteristic functions. The characteristic function of a sum of distributions is the product of the characteristic functions of the distributions. The characteristic function for the sum in Equation 5 is:

$$\begin{aligned} \Phi_Y(p,q) = & \quad \mathbf{E}\left(e^{i\begin{pmatrix} p \\ q \end{pmatrix} \cdot \left[\sum_i^n \gamma_i \begin{pmatrix} \partial_{X_i} s \\ \partial_{X_i} t \end{pmatrix}\right]}\right) \\ = & \quad e^{-\frac{p^2}{2}\sum_i^n (\partial_{X_i} s)^2} \cdot e^{-pq\sum_i^n \partial_{X_i} s \partial_{X_i} t} \cdot e^{-\frac{q^2}{2}\sum_i^n (\partial_{X_i} t)^2}. \end{aligned} \tag{6}$$

The distribution described by Equation 5 can be expressed as a sum of two independent distributions:

$$Z = \begin{pmatrix} s_0 \\ t_0 \end{pmatrix} + \zeta \begin{pmatrix} e \\ f \end{pmatrix} + \eta \begin{pmatrix} g \\ h \end{pmatrix}, \tag{7}$$

where ζ and η are $N(0,1)$ distributions. The characteristic function for Z is:

$$\begin{aligned} \Phi_Z(p,q) = & \quad \mathbf{E}\left(e^{i\begin{pmatrix} p \\ q \end{pmatrix} \cdot \left[\zeta\begin{pmatrix} e \\ f \end{pmatrix} + \eta\begin{pmatrix} g \\ h \end{pmatrix}\right]}\right) \\ = & \quad e^{-\frac{p^2}{2}(e^2+g^2)} \cdot e^{-pq(ef+gh)} \cdot e^{-\frac{q^2}{2}(f^2+h^2)}. \end{aligned} \tag{8}$$

Comparing Equation 8 with Equation 6 reveals that these are the same distribution as long as the following equalities are true:

$$\begin{aligned} e^2 + g^2 = A := & \quad \sum_i^n (\partial_{X_i} s)^2, \\ ef + gh = B := & \quad \sum_i^n \partial_{X_i} s \partial_{X_i} t, \\ f^2 + h^2 = C := & \quad \sum_i^n (\partial_{X_i} t)^2. \end{aligned} \tag{9}$$

The system of equations (9) is underdetermined with three equations for four variables. Loviscach solved this equation in three dimensions by aligning one distribution to either the s- or t-axis, depending on which is more numerically robust [77]. If $A > C$, the following expressions are used:

$$e = \sqrt{A}, \quad f = B/e, \quad g = 0, \quad h = \sqrt{C - f^2},$$

and otherwise:

$$h = \sqrt{C}, \quad g = B/h, \quad f = 0, \quad e = \sqrt{A - g^2}.$$

Appendix B - Elliptical Texture Filter

The task at hand is to find the minimal radius of the elliptical footprint on the triangle. The projection of a pixel with circular footprint in the plane of the triangle is an ellipse with arbitrary orientation. We let a screen space coordinate be defined as (x,y) and texture space coordinates (s,t).

The elliptical footprint in texture space, centered around $(0,0)$, is:

$$E(s,t) = As^2 + Bst + Ct^2, \tag{10}$$

where (s,t) is inside the ellipse if $E(s,t) < F$, and from Heckbert [55], we have the following:

$$\begin{aligned} A(x,y) &= (\partial t/\partial x)^2 + (\partial t/\partial y)^2, \\ B(x,y) &= -2\left(\frac{\partial s}{\partial x}\frac{\partial t}{\partial x} + \frac{\partial s}{\partial y}\frac{\partial t}{\partial y}\right), \\ C(x,y) &= (\partial s/\partial x)^2 + (\partial s/\partial y)^2, \\ F(x,y) &= \left(\frac{\partial s}{\partial x}\frac{\partial t}{\partial y} - \frac{\partial s}{\partial y}\frac{\partial t}{\partial x}\right)^2. \end{aligned} \tag{11}$$

Furthermore, if we introduce $r = \sqrt{(A-C)^2 + B^2}$, the minimum radius of the ellipse is given by:

$$R_{\text{minor}}(x,y) = \sqrt{2F/(A+C+r)}. \tag{12}$$

To determine the highest resolution mipmap level needed somewhere over the triangle, our task is to find the minimal value of R_{minor} over the surface of the triangle. More formally we search for:

$$\min_{(x,y)\in\text{Tri}} \left[R^2_{\text{minor}}(x,y)\right] = \min_{(x,y)\in\text{Tri}} \left[\frac{2F}{A+C+r}\right]. \tag{13}$$

The derivative of a perspective correctly interpolated attribute can be expressed as [35]:

$$\begin{aligned} \frac{\partial s}{\partial x}(x,y) &= \frac{c_3 x + c_4}{Q^2}, \\ \frac{\partial s}{\partial y}(x,y) &= \frac{c_5 y + c_6}{Q^2}, \\ &\cdots \end{aligned} \tag{14}$$

where:

$$Q = c_0 x + c_1 y + c_2. \tag{15}$$

The expression we try to minimize becomes a high order irrational function. While it is possible to simplify this function somewhat, we have yet to find any elegant solution to finding the minimum, or proving that the minimum lies in any of the triangle's vertices.

Bibliography

[1] T. Aila and V. Miettinen, “dPVS: An Occlusion Culling System for Massive Dynamic Environments,” *IEEE Computer Graphics and Applications*, vol. 24, no. 2, pp. 86–97, 2004.

[2] T. Aila, V. Miettinen, and P. Nordlund, “Delay Streams for Graphics Hardware,” *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 792–800, 2003.

[3] K. Akeley, “RealityEngine Graphics,” in *Proceedings of SIGGRAPH*, 1993, pp. 109–116.

[4] T. Akenine-Möller and T. Aila, “Conservative and Tiled Rasterization Using a Modified Triangle Setup,” *Journal of Graphics Tools*, vol. 10, no. 3, pp. 1–8, 2005.

[5] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*, 3rd ed. AK Peters Ltd., 2008.

[6] T. Akenine-Möller, J. Munkberg, and J. Hasselgren, “Stochastic Rasterization using Time-Continuous Triangles,” in *Graphics Hardware*, 2007, pp. 7–16.

[7] T. Akenine-Möller and J. Ström, “Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones,” *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 801–808, 2003.

[8] T. Akenine-Möller, R. Toth, J. Munkberg, and J. Hasselgren, “Efficient Depth of Field Rasterization using a Tile Test based on Half-Space Culling,” *Computer Graphics Forum*, vol. 31, no. 1, pp. 3–18, 2012.

[9] A. R. Alameldeen and D. A. Wood, “Adaptive Cache Compression for High-Performance Processors,” in *International Symposium on Computer Architecture*, 2004, pp. 212–223.

[10] M. Andersson, J. Hasselgren, and T. Akenine-Möller, “Depth Buffer Compression for Stochastic Motion Blur Rasterization,” in *High Performance Graphics*, 2011, pp. 127–134.

[11] T. Annen, T. Mertens, P. Bekaert, H.-P. Seidel, and J. Kautz, “Convolution Shadow Maps,” in *Eurographics Symposium on Rendering*, 2007, pp. 51–60.

[12] T. Annen, T. Mertens, H.-P. Seidel, E. Flerackers, and J. Kautz, “Exponential Shadow Maps,” in *Graphics Interface*, 2008, pp. 155–161.

[13] A. Appel, “Some Techniques for Shading Machine Renderings of Solids,” in *Spring Joint Computer Conference*, 1968, pp. 37–45.

[14] J. Arvo, “Backward Ray Tracing,” SIGGRAPH ’86 Course Notes - Developments in Ray Tracing, 1986.

[15] P. Beaudoin and P. Poulin, “Compressed Multisampling for Efficient Hardware Edge Antialiasing,” in *Graphics Interface*, 2004, pp. 169–176.

[16] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer, “Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful,” *Computer Graphics Forum*, vol. 23, no. 3, pp. 615–624, 2004.

[17] D. Blythe, “The Direct3D 10 System,” in *SIGGRAPH*, 2006, pp. 724–734.

[18] J. Bolz, “NV_conservative_raster (Revision 3),” NVIDIA OpenGL Extension, 2015. [Online]. Available: https://www.opengl.org/registry/specs/NV/conservative_raster.txt

[19] G. Borshukov and J. P. Lewis, “Realistic Human Face Rendering for "The Matrix Reloaded",” in *ACM SIGGRAPH Sketches & Applications*, 2003, pp. 1–1.

[20] S. Boulos, E. Luong, K. Fatahalian, H. Moreton, and P. Hanrahan, “Space-Time Hierarchical Occlusion Culling for Micropolygon Rendering with Motion Blur,” in *High Performance Graphics*, 2010, pp. 11–18.

[21] J. Brunhaver, K. Fatahalian, and P. Hanrahan, “Hardware Implementation of Micropolygon Rasterization with Motion and Defocus Blur,” in *High Performance Graphics*, 2010, pp. 1–9.

[22] C. A. Burns, K. Fatahalian, and W. R. Mark, “A Lazy Object-Space Shading Architecture with Decoupled Sampling,” in *High Performance Graphics*, 2010, pp. 19–28.

[23] P. Clarberg and J. Munkberg, “Deep Shading Buffers on Commodity GPUs,” *ACM Transactions on Graphics*, vol. 33, no. 6, pp. 227:1–227:12, 2014.

[24] P. Clarberg, R. Toth, and J. Munkberg, “A Sort-Based Deferred Shading Architecture for Decoupled Sampling,” *ACM Transactions on Graphics*, vol. 32, no. 4, pp. 141:1–141:10, 2013.

[25] D. Collin, "Culling the Battle Field," Game Developer's Conference, 2011.

[26] R. L. Cook, "Stochastic Sampling in Computer Graphics," *ACM Transactions on Graphics*, vol. 5, no. 1, pp. 51–72, 1986.

[27] R. L. Cook, L. Carpenter, and E. Catmull, "The Reyes Image Rendering Architecture," in *Computer Graphics (Proceedings of SIGGRAPH)*, vol. 21, 1987, pp. 95–102.

[28] W. Dally, "Power Efficient Supercomputing," 2009, Accelerator-based Computing and Manycore Workshop (presentation).

[29] W. Donnelly and A. Lauritzen, "Variance Shadow Maps," in *Symposium on Interactive 3D Graphics and Games*, 2006, pp. 161–165.

[30] F. Durand, N. Holzschuch, C. Soler, E. Chan, and F. X. Sillion, "A Frequency Analysis of Light Transport," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 1115–1126, 2005.

[31] K. Egan, F. Durand, and R. Ramamoorthi, "Practical Filtering for Efficient Ray-Traced Directional Occlusion," *ACM Transactions on Graphics*, vol. 30, no. 6, pp. 180:1–180:10, 2011.

[32] K. Egan, F. Hecht, F. Durand, and R. Ramamoorthi, "Frequency Analysis and Sheared Filtering for Shadow Light Fields of Complex Occluders," *ACM Transactions on Graphics*, vol. 30, no. 2, pp. 9:1–9:13, 2011.

[33] K. Egan, Y.-T. Tseng, N. Holzschuch, F. Durand, and R. Ramamoorthi, "Frequency Analysis and Sheared Reconstruction for Rendering Motion Blur," *ACM Transactions on Graphics*, vol. 28, no. 3, pp. 93:1–93:13, 2009.

[34] E. Eisemann, M. Schwarz, U. Assarsson, and M. Wimmer, *Real-Time Shadows*. AK Peters Ltd./CRC Press, 2011.

[35] J. Ewins, M. Waller, M. White, and P. Lister, "MIP-map Level Selection for Texture Mapping," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 4, pp. 317–329, 1998.

[36] K. Fatahalian, S. Boulos, J. Hegarty, K. Akeley, W. R. Mark, H. Moreton, and P. Hanrahan, "Reducing Shading on GPUs using Quad-Fragment Merging," *ACM Transactions on Graphics*, vol. 29, no. 4, pp. 67:1–67:8, 2010.

[37] K. Fatahalian, E. Luong, S. Boulos, K. Akeley, W. R. Mark, and P. Hanrahan, "Data-Parallel Rasterization of Micropolygons with Defocus and Motion Blur," in *High Performance Graphics*, 2009, pp. 59–68.

[38] R. Fernando, "Percentage Closer Soft Shadows," in *ACM SIGGRAPH Sketches*, 2005, p. 35.

[39] H. Fuchs and J. Poulton, "PIXEL-PLANES: A VLSI-Oriented Design for a Raster Graphics Engine," *VLSI DESIGN (Third Quarter 1981)*, pp. 20–28, 1981.

[40] G. Golub and C. V. Loan, *Matrix Computations*, 3rd ed. John Hopkins University Press, 1996.

[41] N. Greene, "Hierarchical Polygon Tiling with Coverage Masks," in *Proceedings of SIGGRAPH*, 1996, pp. 65–74.

[42] N. Greene and M. Kass, "Error-bounded Antialiased Rendering of Complex Environments," in *Proceedings of SIGGRAPH*, 1994, pp. 59–66.

[43] N. Greene, M. Kass, and G. Miller, "Hierarchical Z-Buffer Visibility," in *Proceedings of SIGGRAPH*, 1993, pp. 231–238.

[44] C. J. Gribel, R. Barringer, and T. Akenine-Möller, "High-Quality Spatio-Temporal Rendering using Semi-Analytical Visibility," *ACM Transactions on Graphics*, vol. 30, no. 4, pp. 54:1–54:12, 2011.

[45] C. J. Gribel, M. Doggett, and T. Akenine-Möller, "Analytical Motion Blur Rasterization with Compression," in *High Performance Graphics*, 2010, pp. 163–172.

[46] C. J. Gribel, J. Munkberg, J. Hasselgren, and T. Akenine-Möller, "Theory and Analysis of Higher-Order Motion Blur Rasterization," in *High Performance Graphics*, 2013, pp. 7–16.

[47] J.-P. Guertin, M. McGuire, and D. Nowrouzezahrai, "A Fast and Stable Feature-Aware Motion Blur Filter," in *High Performance Graphics*, 2014, pp. 51–60.

[48] M. Guthe, Á. Balázs, and R. Klein, "Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries," in *Eurographics Symposium on Rendering*, 2006, pp. 207–214.

[49] P. Haeberli and K. Akeley, "The Accumulation Buffer: Hardware Support for High-Quality Rendering," in *Computer Graphics (Proceedings of SIGGRAPH)*, vol. 24, 1990, pp. 309–318.

[50] J. Hasselgren and T. Akenine-Möller, "An Efficient Multi-View Rasterization Architecture," in *Eurographics Symposium on Rendering*, 2006, pp. 61–72.

[51] J. Hasselgren and T. Akenine-Möller, "Efficient Depth Buffer Compression," in *Graphics Hardware*, 2006, pp. 103–110.

[52] J. Hasselgren, T. Akenine-Möller, and L. Ohlsson, "Conservative Rasterization," in *GPU Gems 2*, M. Pharr and R. Fernando, Eds. Addison-Wesley Professional, 2005, ch. 42, pp. 677–690.

[53] J. Hasselgren, M. Andersson, J. Nilsson, and T. Akenine-Möller, "A Compressed Depth Cache," *Journal of Computer Graphics Techniques*, vol. 1, no. 1, pp. 101–118, 2012.

[54] J. Hasselgren, J. Munkberg, and K. Vaidyanathan, "Practical Layered Reconstruction for Defocus and Motion Blur," *Journal of Computer Graphics Techniques*, vol. 4, no. 2, pp. 45–58, 2015.

[55] P. S. Heckbert, "Survey of Texture Mapping," *IEEE Computer Graphics & Applications*, vol. 6, no. 11, pp. 56–67, Nov. 1986.

[56] ——, "Adaptive Radiosity Textures for Bidirectional Ray Tracing," in *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 1990, pp. 145–154.

[57] M. Houle and G. Toussaint, "Computing the Width of a Set," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 5, pp. 761 –765, 1988.

[58] T. Inada and M. D. McCool, "Compressed Lossless Texture Representation and Caching," in *Graphics Hardware*, 2006, pp. 111–120.

[59] B. Johnsson, P. Ganestam, M. Doggett, and T. Akenine-Möller, "Power Efficiency for Software Algorithms running on Graphics Processors," in *High Performance Graphics*, 2012, pp. 67–75.

[60] T. R. Jones and R. N. Perry, "Antialiasing with Line Samples," in *Eurographics Workshop on Rendering*, 2000, pp. 197–205.

[61] N. P. Jouppi and C.-F. Chang, "Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency," in *Graphics Hardware*, 1999, pp. 85–93.

[62] J. T. Kajiya, "The Rendering Equation," in *Computer Graphics (Proceedings of SIGGRAPH)*, 1986, pp. 143–150.

[63] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[64] A. Kensler and P. Shirley, "Optimizing Ray-Triangle Intersection via Automated Search," in *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, Sep 2006, pp. 33–38.

[65] C. Kolb, D. Mitchell, and P. Hanrahan, "A Realistic Camera Model for Computer Graphics," in *Proceedings of SIGGRAPH*, 1995, pp. 317–324.

[66] S. Laine, T. Aila, T. Karras, and J. Lehtinen, "Clipless Dual-Space Bounds for Faster Stochastic Rasterization," *ACM Transactions on Graphics*, vol. 30, no. 4, pp. 106:1–106:6, 2011.

[67] S. Laine and T. Karras, “Efficient Triangle Coverage Tests for Stochastic Rasterization,” NVIDIA Corporation, Tech. Rep. NVR-2011-003, Sep 2011.

[68] ——, “Stratified Sampling for Stochastic Transparency,” in *Proceedings of EGSR*, 2011, pp. 1197–1204.

[69] E. Lapidous and G. Jiao, “Optimal Depth Buffer for Low-cost Graphics Hardware,” in *Graphics Hardware*, 1999, pp. 67–73.

[70] A. Lauritzen and M. McCool, “Layered Variance Shadow Maps,” in *Graphics Interface*, 2008, pp. 139–146.

[71] J.-S. Lee, W.-K. Hong, and S.-D. Kim, “Design and Evaluation of a Selective Compressed Memory System,” in *International Conference on Computer Design*, 1999, pp. 184 –191.

[72] S. Lee, E. Eisemann, and H.-P. Seidel, “Depth-of-field Rendering with Multiview Synthesis,” *ACM Transactions on Graphics*, vol. 28, no. 5, pp. 134:1–134:6, Dec. 2009.

[73] J. Lehtinen, T. Aila, J. Chen, S. Laine, and F. Durand, “Temporal Light Field Reconstruction for Rendering Distribution Effects,” *ACM Transactions on Graphics*, vol. 30, no. 4, pp. 55:1–55:12, 2011.

[74] G. Liktor and C. Dachsbacher, “Decoupled Deferred Shading for Hardware Rasterization,” in *Symposium on Interactive 3D Graphics and Games*, 2012, pp. 143–150.

[75] D. B. Lloyd, N. K. Govindaraju, S. E. Molnar, and D. Manocha, “Practical Logarithmic Rasterization for Low-error Shadow Maps,” in *Graphics Hardware*, 2007, pp. 17–24.

[76] T. Lokovic and E. Veach, “Deep Shadow Maps,” in *Proceedings of SIGGRAPH*, 2000, pp. 385–392.

[77] J. Loviscach, “Motion Blur for Textures by Means of Anisotropic Filtering,” in *Eurographics Symposium on Rendering*, 2005, pp. 105–110.

[78] C. Luksch, R. F. Tobler, R. Habel, M. Schwärzler, and M. Wimmer, “Fast Light-Map Computation with Virtual Polygon Lights,” in *Symposium on Interactive 3D Graphics and Games*, 2013, pp. 87–94.

[79] O. Mattausch, J. Bittner, and M. Wimmer, “CHC++: Coherent Hierarchical Culling Revisited,” *Computer Graphics Forum*, vol. 27, no. 2, pp. 221–230, 2008.

[80] M. D. McCool, C. Wales, and K. Moule, “Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization,” in *Graphics Hardware*, 2001, pp. 65–72.

[81] M. McGuire and E. Enderton, "Colored Stochastic Shadow Maps," in *Symposium on Interactive 3D Graphics and Games*, 2011, pp. 89–96.

[82] M. McGuire, E. Enderton, P. Shirley, and D. Luebke, "Real-Time Stochastic Rasterization on Conventional GPU Architectures," in *High Performance Graphics*, 2010, pp. 173–182.

[83] M. McGuire, P. Hennessy, M. Bukowski, and B. Osman, "A Reconstruction Filter for Plausible Motion Blur," in *Symposium on Interactive 3D Graphics and Games*, 2012, pp. 135–142.

[84] S. Mehta, B. Wang, and R. Ramamoorthi, "Axis-Aligned Filtering for Interactive Sampled Soft Shadows," *ACM Transactions on Graphics*, vol. 31, no. 6, pp. 163:1–163:10, 2012.

[85] S. U. Mehta, B. Wang, R. Ramamoorthi, and F. Durand, "Axis-Aligned Filtering for Interactive Physically-based Diffuse Indirect Lighting," *ACM Transactions on Graphics*, vol. 32, no. 4, pp. 96:1–96:12, 2013.

[86] S. U. Mehta, J. Yao, R. Ramamoorthi, and F. Durand, "Factored Axis-aligned Filtering for Rendering Multiple Distribution Effects," *ACM Transactions on Graphics*, vol. 33, no. 4, pp. 57:1–57:12, 2014.

[87] S. Morein, "ATI Radeon HyperZ Technology," in *Graphics Hardware, Hot3D Proceedings*, 2000.

[88] J. Munkberg and T. Akenine-Möller, "Backface Culling for Motion Blur and Depth of Field," *Journal of Graphics, GPU, and Game Tools*, vol. 15, no. 2, pp. 123–139, 2011.

[89] J. Munkberg and T. Akenine-Möller, "Hyperplane Culling for Stochastic Rasterization," in *Eurographics 2012 – Short Papers*, 2012, pp. 105–108.

[90] J. Munkberg, P. Clarberg, J. Hasselgren, R. Toth, M. Sugihara, and T. Akenine-Möller, "Hierarchical Stochastic Motion Blur Rasterization," in *High Performance Graphics*, 2011, pp. 107–118.

[91] J. Munkberg, R. Toth, and T. Akenine-Möller, "Per-Vertex Defocus Blur for Stochastic Rasterization," *Computer Graphics Forum*, vol. 31, no. 4, pp. 1385–1389, 2012.

[92] J. Munkberg, K. Vaidyanathan, J. Hasselgren, P. Clarberg, and T. Akenine-Möller, "Layered Light Field Reconstruction for Defocus and Motion Blur," *Computer Graphics Forum*, vol. 33, no. 4, pp. 81–92, 2014.

[93] F. Navarro, F. J. Serón, and D. Gutierrez, "Motion Blur Rendering: State of the Art," *Computer Graphics Forum*, vol. 30, no. 1, pp. 3–26, 2011.

[94] D. Nehab, P. V. Sander, J. Lawrence, N. Tatarchuk, and J. R. Isidoro, "Accelerating Real-Time Shading with Reverse Reprojection Caching," in *Graphics Hardware*, 2007, pp. 25–35.

[95] F. E. Nicodemus, "Directional Reflectance and Emissivity of an Opaque Surface," *Applied Optics*, vol. 4, no. 7, pp. 767–775, Jul 1965.

[96] M. Olano and T. Greer, "Triangle Scan Conversion using 2D Homogeneous Coordinates," in *Graphics Hardware*, 1997, pp. 89–95.

[97] A. B. Owen, "Quasi-Monte Carlo Sampling," SIGGRAPH Monte Carlo Ray Tracing Course, 2003.

[98] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed. Morgan Kaufmann Publishers Inc., 2007.

[99] J. Pineda, "A Parallel Algorithm for Polygon Rasterization," in *Computer Graphics (Proceedings of SIGGRAPH)*, vol. 22, 1988, pp. 17–20.

[100] J. Pool, A. Lastra, and M. Singh, "Lossless Compression of Variable-Precision Floating-Point Buffers on GPUs," in *Symposium on Interactive 3D Graphics and Games*, 2012, pp. 47–54.

[101] M. Potmesil and I. Chakravarty, "A Lens and Aperture Camera Model for Synthetic Image Generation," in *Computer Graphics (Proceedings of SIGGRAPH)*, vol. 15, 1981, pp. 297–305.

[102] J. Ragan-Kelley, J. Lehtinen, J. Chen, M. Doggett, and F. Durand, "Decoupled Sampling for Graphics Pipelines," *ACM Transactions on Graphics*, vol. 30, no. 3, pp. 17:1–17:17, 2011.

[103] J. Rasmusson, J. Hasselgren, and T. Akenine-Möller, "Exact and Error-Bounded Approximate Color Buffer Compression and Decompression," in *Graphics Hardware*, 2007, pp. 41–48.

[104] J. Rasmusson, J. Ström, and T. Akenine-Möller, "Error-Bounded Lossy Compression of Floating-Point Color Buffers using Quadtree Decomposition," *The Visual Computer*, vol. 26, no. 1, pp. 17–30, January 2008.

[105] W. T. Reeves, D. H. Salesin, and R. L. Cook, "Rendering Antialiased Shadows with Depth Maps," in *Computer Graphics (Proceedings of SIGGRAPH)*, vol. 21, 1987, pp. 283–291.

[106] M. Salvi, K. Vidimče, A. Lauritzen, and A. Lefohn, "Adaptive Volumetric Shadow Maps," in *Eurographics Symposium on Rendering*, Jun. 2010, pp. 1289–1296.

[107] M. Segal and K. Akeley, "The OpenGL Graphics System: A Specification, v 4.5," August 2015.

[108] J. Shapiro, "Embedded Image Coding using Zerotrees of Wavelet Coefficients," *IEEE Transactions on Signal Processing*, vol. 41, no. 12, pp. 3445–3462, 1993.

[109] P. Sitthi-Amorn, J. Lawrence, L. Yang, P. V. Sander, D. Nehab, and J. Xi, "Automated Reprojection-based Pixel Shader Optimization," *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 127:1–127:11, 2008.

[110] D. Staneker, D. Bartz, and M. Meissner, "Improving Occlusion Query Efficiency with Occupancy Maps," in *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2003, pp. 111–118.

[111] J. Ström, P. Wennersten, J. Rasmusson, J. Hasselgren, J. Munkberg, P. Clarberg, and T. Akenine-Möller, "Floating-Point Buffer Compression in a Unified Codec Architecture," in *Graphics Hardware*, 2008, pp. 75–84.

[112] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *ACM Computing Surveys*, vol. 6, no. 1, pp. 1–55, 1974.

[113] R. Toth and E. Linder, "Stochastic Depth of Field using Hardware Accellerated Rasterization," Master's thesis, Lund University, 2008.

[114] S. Tzeng, A. Patney, A. Davidson, M. S. Ebeida, S. A. Mitchell, and J. D. Owens, "High-Quality Parallel Depth-of-Field Using Line Samples," in *High Performance Graphics*, 2012, pp. 23–31.

[115] K. Vaidyanathan, J. Munkberg, P. Clarberg, and M. Salvi, "Layered Light Field Reconstruction for Defocus Blur," *ACM Transactions on Graphics*, vol. 34, no. 2, pp. 23:1–23:12, 2015.

[116] K. Vaidyanathan, M. Salvi, R. Toth, T. Foley, T. Akenine-Möller, J. Nilsson, J. Munkberg, J. Hasselgren, M. Sugihara, P. Clarberg, T. Janczak, and A. Lefohn, "Coarse Pixel Shading," *High Performance Graphics*, pp. 9–18, 2014.

[117] K. Vaidyanathan, R. Toth, M. Salvi, S. Boulos, and A. Lefohn, "Adaptive Image Space Shading for Motion and Defocus Blur," *High Performance Graphics*, pp. 13–21, 2012.

[118] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: A Kernel Framework for Efficient CPU Ray Tracing," *ACM Transactions on Graphics*, vol. 33, no. 4, pp. 143:1–143:8, 2014.

[119] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[120] L. Williams, "Casting Curved Shadows on Curved Surfaces," *Computer Graphics (Proceedings of SIGGRAPH)*, vol. 12, pp. 270–274, 1978.

[121] M. Wimmer, D. Scherzer, and W. Purgathofer, “Light Space Perspective Shadow Maps,” in *Eurographics Symposium on Rendering*. Eurographics, jun 2004, pp. 143–151.

[122] A. Woo and P. Poulin, *Shadow Algorithms Data Miner*. AK Peters/CRC Press, 2012.

[123] B. Yang, Z. Dong, J. Feng, H.-P. Seidel, and J. Kautz, “Variance Soft Shadow Mapping,” *Computer Graphics Forum*, vol. 29, no. 7, pp. 2127–2134, 2010.

[124] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff, III, “Visibility Culling Using Hierarchical Occlusion Maps,” *Proceedings of SIGGRAPH*, pp. 77–88, 1997.