Energy Analysis for Graphics Processors using Novel Methods
&
Efficient Multi-View Rendering

Björn Johnsson
Department of Computer Science
Lund University

LUND INSTITUTE OF TECHNOLOGY
Lund University

ISBN 978-91-7623-026-8 (Printed)
ISBN 978-91-7623-027-5 (Electronic)
ISSN 1404-1219
Dissertation 44, 2014
LU-CS-DISS:2014-1

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

Email: bjorn.johnsson@cs.lth.se

Typeset using LaTeX2ε
Printed in Sweden by Tryckeriet i E-huset, Lund, 2014

Abstract

Real-time rendering is increasingly being performed on battery powered devices, such as laptops and mobile phones. As a result, performance in rendering time is not the only important factor, but the energy consumption is also becoming more and more important, as a lower energy usage directly translates to longer battery time. The main topic of this thesis is *energy consumption* of graphics processors. To examine this, we perform high-frequency measurements of a number of graphics devices' power consumption. Simultaneously, we render real-time graphics workloads to analyze the power consumption for a number of devices, algorithms, and settings. We draw a number of conclusions for different platforms, e.g., that it is incorrect to assume a direct correlation between rendering time and per-frame energy consumption. We also present a method for evaluating if there is such a correlation on a specific, efficiently utilized, platform. This method uses Pareto frontiers to filter out measurements that are inefficient, with respect to rendering time and energy consumption, and analyses only measured data points with a possible trade-off. Our long-term goal is to use our conclusions for developing energy-efficient algorithms, as well as raising awareness in the developer community to consider rendering time *and* energy consumption while developing real-time graphics algorithms. Furthermore, we develop and improve methods for correlating energy consumption on a per-frame basis with other information collected for specific frames, to enable improved analysis regarding real-time graphics energy consumption.

Our second topic is *efficient multi-view rendering*. We have developed an optimized algorithm for multi-view ray tracing, targeting auto-stereoscopic displays, which performs up to an order of magnitude faster than previous state of the art algorithms. In addition, we have examined the feasibility of enabling a proposed higher-dimensional rasterizer implemented in hardware to render multi-view and stereoscopic image sets in a single pass. We find it straightforward to adapt a higher-dimensional rasterizer to support multi-view rendering, and propose improvements to enhance the rendering performance for such applications.

Acknowledgements

First of all I would like to thank my main supervisor, Tomas Akenine-Möller for guidance, good advice and always interesting discussions. I would also like to thank my assistant supervisors Jacob Munkberg and Michael Doggett for support and enlightening discussions. I also owe gratitude to my manager Jim Nilsson for much needed structure and guidance.

Many thanks to all who spent time reviewing my thesis: Tomas Akenine-Möller, Jim Nilsson, Jacob Munkberg, Michael Doggett, and Magnus Andersson.

My gratitude extends to my colleagues at the Advanced Rendering Team at Intel. I would especially like to thank the people at our Swedish office: Jim Nilsson, Tomas Akenine-Möller, Magnus Andersson, Jacob Munkberg, Jon Hasselgren, Robert Toth, and Petrik Clarberg, always ready to shoot down bad ideas, but at the same time supporting the good ones.

I would also like to thank all the members of the Computer Graphics Group at Lunds Tekniska Högskola: Tomas Akenine-Möller, Michael Doggett, Magnus Andersson, Carl Johan Gribel, Rasmus Barringer och Per Ganestam.

My research was funded by Intel Corporation, and I will always be grateful for this opportunity. I would also like to thank ElektroTekniska Föreningen for all their assistance and cooperation.

But foremost, I owe gratitude to my family, to my wonderfully patient wife Lovisa, who has always supported me and believed in me, and also to my children; Allison, Amadea, and Laura; for showing enthusiasm for my scientific rants.

Preface

This thesis summarizes my research in energy measurements on real-time graphics and efficient multi-view rendering. The following papers are included:

I. Björn Johnsson, Per Ganestam, Michael Doggett, and Tomas Akenine-Möller, "Power Efficiency for Software Algorithms running on Graphics Processors,"
in *High Performance Graphics*, pages 67–75, 2012.

II. Tomas Akenine-Möller and Björn Johnsson,
"Performance per What?,"
in *Journal of Computer Graphics Techniques*, 1(1):37–41, 2012.

III. Björn Johnsson and Tomas Akenine-Möller,
"Measuring Per-Frame Energy Consumption of Real-Time Graphics Applications,"
in *Journal of Computer Graphics Techniques*, 3(1):60–73, 2014.

IV. Björn Johnsson and Tomas Akenine-Möller,
"A Performance and Energy Evaluation of Many-Light Rendering Algorithms,"
submitted to *Visual Computer*, 2014.

V. 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*, 27(6-8):665–676, 2011.

VI. Jim Nilsson, Petrik Clarberg, Björn Johnsson, Jacob Munkberg, Jon Hasselgren, Robert Toth, Marco Salvi, and Tomas Akenine-Möller,
"Design and Novel Uses of Higher-Dimensional Rasterization,"
in *High Performance Graphics*, pages 1–11, 2012.

Contents

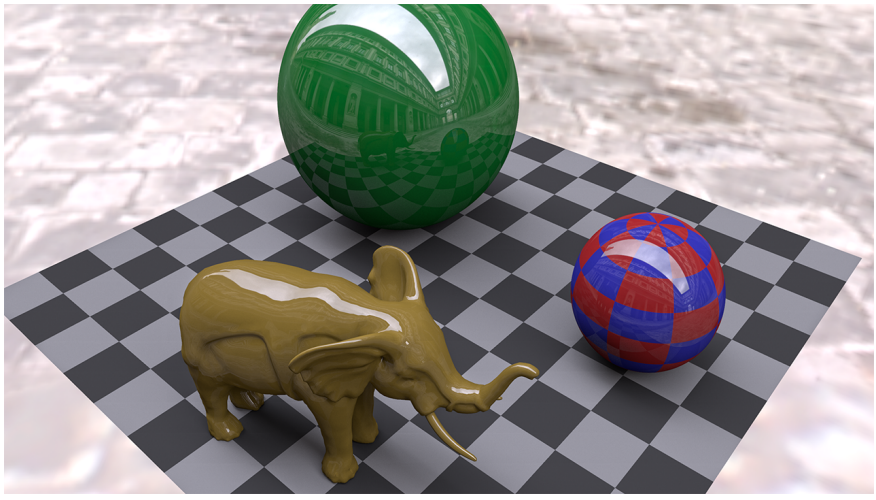


Figure 1: A computer generated image. The photorealistic image is created using path tracing, with 1,048,675 samples per pixel.

1 Introduction

Computer graphics is an academic field that concerns the creation and manipulation of images. A common way to create an image (Figure 1) is to take a geometrical representation of a scene, containing objects, light sources, and a camera (Figure 2), and calculate the light entering each pixel of the camera. The objects of the scene are often described as a set of primitives, usually triangles, which comprises a surface in three-dimensional space. They are also assigned a material, which describes the way the surface interacts with light. For each pixel, the computer then has two tasks to perform. First, it needs to determine the closest primitives that are covering each pixel. Second, for those primitives and the material assigned, and given the light sources in the scene, it needs to calculate the appropriate colors. The act of determining the correct primitive and calculating a color for each pixel is called *rendering* an image.

Rendering images is typically done using one of two main methods, ray tracing [1] and rasterization [2], which differ mainly in which order they traverse pixels and primitives. Ray tracing traverses the pixels first and for each traversed pixel determines which is the foremost of the primitives covering that pixel. Rasterization traverses the primitives first, and determines which pixels each primitive covers.

In this introductory section, I will briefly introduce rasterization and ray tracing, as both methods are used in my research. Then follows Section 2, where I summarize the methodology and contributions of each paper. The two main areas of my research, *a*) *energy usage* in relation to computer graphics, and *b*) *multi-view*

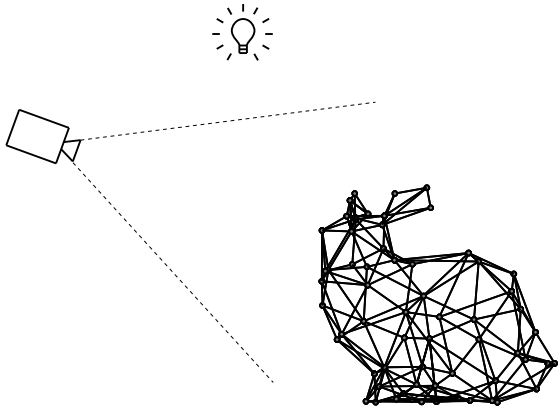


Figure 2: A scene with a camera, a single light source, and an object comprised of triangles.

rendering for multi-stereoscopic displays, are described in Section 3 and Section 4 respectively, including a more detailed overview of my contributions. Finally, Section 5 features my final conclusions and suggestions for future work.

1.1 Ray Tracing

Ray tracing is a rendering method where rays traced through the scene are used to simulate how light travels between and within objects. As the rays can have arbitrary directions and origins, it is a very flexible method, capable of simulating many advanced optical phenomena, e.g., shadows, reflections, refractions, indirect illumination, and caustics [3, 4]. Ray tracing is mainly used as an off-line rendering method, i.e., for films and still images where the rendering is allowed to take minutes to hours for a single image or frame. However, efforts to enable real-time or interactive ray tracing exists [5, 6, 7, 8].

Ray tracing calculates the color of a pixel by generating a number of primary rays, determining which primitives those rays intersects first and calculate how the surfaces of those primitives interacts with the light within the scene. The primary rays have their origin at the position of the camera, and directions determined by the viewing direction of the camera and points on the image plane, e.g., at pixel centers, that are being sampled. To achieve high image quality, it is not sufficient to trace a single primary ray per pixel. The common method to improve image quality in ray tracing, called super sampling, improves the image by tracing several primary rays per pixel, at different positions, and use a weighted sum of the rays' colors.

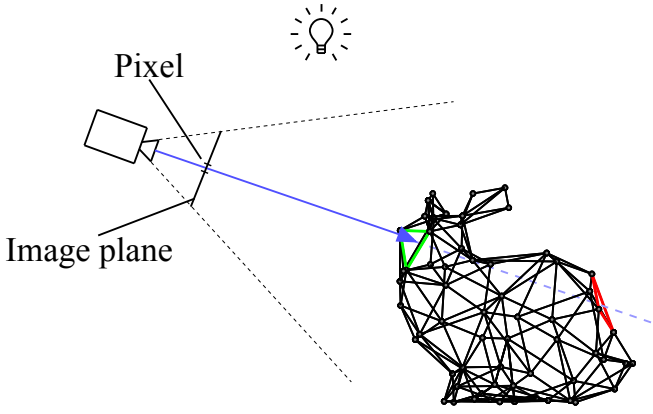


Figure 3: The same scene as in Figure 2, with a ray shot from the camera. The ray (blue) intersects both the green and the red triangle, where the green triangle, being closer to the camera, is returned as the first intersected primitive.

The ray is then "traced", i.e., a program determines which of the primitives the ray intersects, and which of those primitives is closest, shown in Figure 3. Calculations are usually optimized using a spatial data acceleration structure. For that intersected primitive, the position of the intersection on the primitive, which is calculated during the tracing, and the properties of the intersection point are determined. With those properties, it is possible to shade, i.e., calculate the color contribution of that ray to the pixel. Ray tracing supports tracing rays with any origin and in any direction. It is by generating rays while shading at an intersection, that it is possible to achieve the important optical effects mentioned, e.g., shadows, reflections and refractions. Rays created while shading are called secondary rays.

1.1.1 Camera model

The primary rays are generated according to a virtual camera model, and this camera model can be of varied complexity [9]. A simple camera model, i.e., the pinhole camera (Figure 4, left), is defined by a camera position and an image plane behind it, where primary rays are created with an origin at the camera position and a direction defined as the vector from a position on the image plane through the camera position. A slightly more advanced camera model could use the notion of an artificial lens, to create photorealistic effects, e.g, defocus blur, by defining the camera not as a point, but rather as a "lens". This lens has a non-zero extent, as can be seen in Figure 4, right. To create a primary ray using such a camera, instead of only selecting a point on the image plane, it is necessary to select a point also on

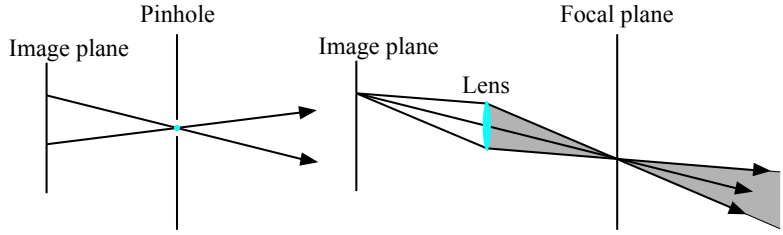


Figure 4: Left: pinhole camera. Right: camera with a lens, supporting depth of field. The gray area is showing the region sampled for a single pixel.

the lens. An example of three primary rays originating from the same point on the focal plane, one through the center of the lens and one at the edge of the lens, are visible in Figure 4, right.

An implication that comes from using the more advanced camera model is that the samples now have four dimensions, instead of two. With a pinhole camera, a two dimensional position on the image plane fully defines a primary ray. To define a primary ray with the more advanced camera model, it is also necessary to have a position on the lens, adding another two dimensions. One may also include time as a dimension, and achieve the effect of motion blur, where objects that move are blurry in the direction of the motion. With motion blur added, the dimensionality is raised to either three or five, depending on the underlying camera model.

1.1.2 Tracing rays

Generated rays are traced against the geometry of the scene by calculating the intersection point of the closest primitive that intersects the ray. This can be done by, for each ray, analytically calculating the intersection points for each primitive and retaining the closest one. However, as both the number of rays and the number of primitives are usually large, this would be very costly. It is therefore necessary to arrange all primitives in a spatial acceleration data structure, where large sets of primitives can be discarded in a shared, conservative intersection test.

There are several types of structures used, usually based on either regular grids or geometrically hierarchical structures, e.g., kD-trees, octrees, or bounding volume hierarchies (BVHs) [1]. Here I will describe bounding volume hierarchies as an example.

A bounding volume hierarchy is constructed as a tree, where the primitives of a node are divided among its children, as illustrated in Figure 5. Each internal node in the tree contains a bounding volume, e.g., an axis-aligned bounding box (AABB), enclosing all its descendant leaf nodes. It furthermore contains links to a number of child nodes, either internal nodes with bounding volumes or leaf

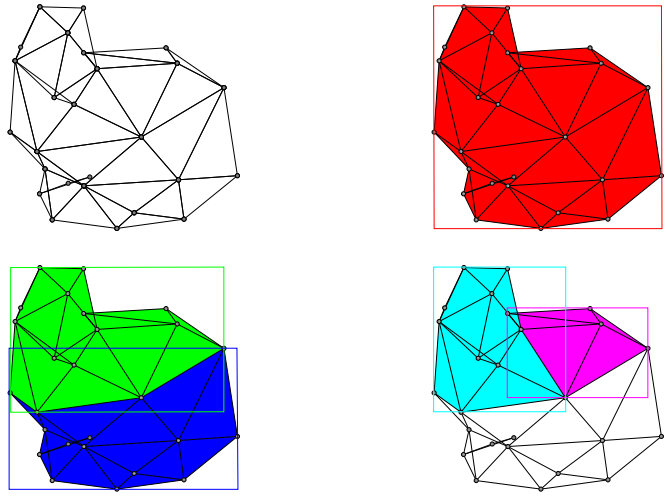


Figure 5: Bounding volume hierarchy. Top left: a simplified version of the model in Figure 2. Top right: the model enclosed in an axis-aligned bounding box, comprising the root node. Lower left: the root node divided in two parts, with the corresponding bounding boxes. Lower right: a further division of the green node (lower left), into two new nodes with bounding boxes.

nodes containing primitives as well as a bounding volume. An example of how the primitives can be divided and enclosed can be seen in Figure 5.

By using a BVH over the geometry while ray tracing, it is possible to trace a ray efficiently and to determine which primitive it hits. This is done by performing a recursive traversal over the tree, performing intersection tests against the bounding volumes of the nodes. If the intersection test between a ray and a node's bounding volume results in a non-hit, the entire sub-tree can be disregarded. Otherwise, the node traversal descend into the subtree of the node. If the traversal reaches a leaf node, the ray is intersection tested against the primitives the node contains. Throughout the traversal, the closest encountered primitive is kept. The efficiency of the traversal can further be improved by, when traversing into the children of a node, first visiting the closest node, according to the ray direction.

To traverse a bounding volume hierarchy as described above, two different intersection tests are required, one for calculating the intersection between the ray and a primitive, and one for calculating the intersection between the ray and the chosen type of bounding volume [1].

The ray-primitive test needs to calculate both a boolean result, a hit or a non-hit, the distance along the ray, and the intersection's position on the primitive. For the most common primitive, a triangle, the position is usually expressed using

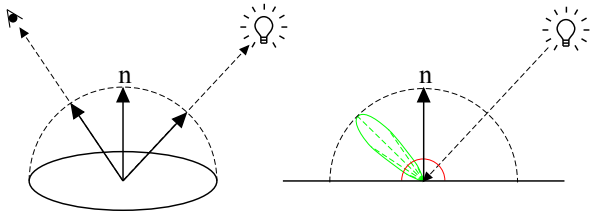


Figure 6: A bidirectional reflectance distribution function (BRDF) describes the amount of reflected light, given the incoming and outgoing direction in relation to the surface normal n. Left: the incident and outgoing direction used to evaluate a BRDF. Right: a simple example of a BRDF, with a specular lobe (green) and a diffuse contribution (red), illustrating the exiting amount of light for a single incident ray.

barycentric coordinates [2]. These barycentric coordinates can then be used for interpolating values assigned to the three vertices.

The primary value sought when testing a ray against a bounding volume is boolean, i.e., if it is a hit or a non-hit. With this value, it is determined if the content of the node is of any interest. However, if the test also returns bounds of the intersection, the point of entry and exit, it is possible to optimize further.

1.1.3 Shading

After calculating the closest intersection point for a ray, the next step is to calculate the radiance, the light emitted from the point towards the origin of the ray. Usually, this is a matter of calculating an approximation of the light leaving the surface in the opposite direction of the ray. For a photorealistic result, this requires evaluating the rendering equation [4], which states:

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega f_r(x,\omega_i,\omega_o) L_i(x,\omega_i)(\omega_i \cdot n)\mathrm{d}\omega_i. \tag{1}$$

In short, the outgoing light from position x in direction ω_o is the sum of two terms. First, the light emitted from position x in direction ω_o, e.g., if x lies on a light source. The second term is an integral over the hemisphere, where incoming light from all directions ω_i is calculated. All incoming light L_i is multiplied with two terms, first f_r, a bidirectional reflectance distribution function (BRDF), that describes the surface's reflective attributes, and second $\omega_i \cdot n$ that diminishes the incoming light due to the incident angle. Note that Equation 1 is recursive, i.e., the term $L_i(x,\omega)$ is calculated using the same equation, at the intersection point x' in

direction ω, and that the incoming light, $L_i(x, \omega)$, is the outgoing $L_o(x', \omega)$ of that evaluation.

The BRDF plays an important role in the visual appearance of different materials. There exists a number of different bidirectional reflectance distribution functions, where many of the functions take a set of attributes, e.g., describing how specular a surface is. The functions may require the incident and exiting directions, in relation to the surface normal, see Figure 6 (left), and often includes both a specular and diffuse term, shown as colored lobes in Figure 6, right.

As integrating over the hemisphere at every single ray-primitive intersection is computationally expensive, a possible simplification is to only take direct lighting into account, i.e, the light emitted directly from light sources, since this is usually the most important part of the lighting. In that case, for each intersection point calculating the light is a matter of evaluating all light sources, i.e., for each light source determine its visibility by tracing a ray from the intersection to the light source and, if visible, evaluate the BRDF from the direction towards the light.

However, for accurate, photorealistic renderings, it is necessary to sample the entire hemisphere for each intersection. Otherwise indirect light, bouncing on one or more surfaces, is not captured.

1.2 Rasterization

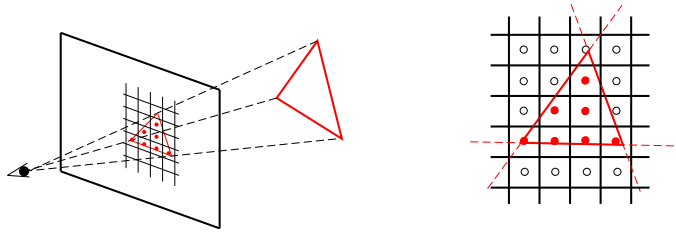


Figure 7: The rasterization of a triangle. Left: the triangle is projected onto the image plane along the dashed lines. Right: the projected triangle (solid lines) and its edge equations (dashed lines). The pixel centers determined to lie within the triangle are shown in red.

Rasterization, in contrast to ray tracing, operates on each primitive sequentially, by determining which samples a primitive covers. For those samples it generates a fragment, i.e., a position on a primitive to shade. Rasterization is generally faster, in large part because it is possible to create efficient hardware implementations, and is at the moment the most commonly used method for real-time graphics. A very high-level outline of a rasterizer is shown in Algorithm 1.

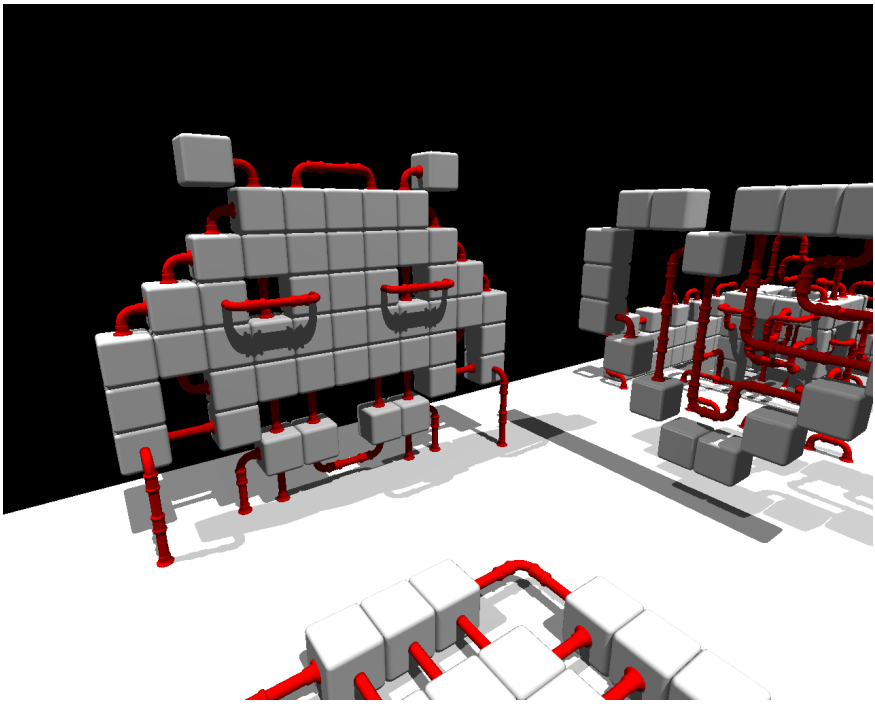


Figure 8: Real-time rendering with shadows rendered using stencil shadow volumes [10].

```
1  FOREACH Primitive
2      Project the primitives vertices
3      In 2D, calculate edge equations from the projected vertices
4      FOREACH sample
5          Evaluate all edge equations
6          IF sample is inside all edge equations
7              Calculate color of sample
8              Send the color to the output merger
```

Algorithm 1: Pseudocode of a rasterizer.

Looking at Algorithm 1, the pseudocode of a rasterizer, the rasterizer first loops over all primitives (Line 1). For each primitive, it projects all its vertices (Figure 7, left). It uses the projected 2D positions of the vertices to calculate a set of edge equations, i.e., implicit representations of the lines defined by two vertices (Figure 7, right). With these edge equations, it evaluates which side of each line the sample lies (Line 5). If the sample is determined to lie within all edges (Line 6), it is within the primitive itself, and the per-fragment calculations need to be executed (Line 7).

For each sample within the primitive, after the per-fragment calculations are performed, the result is sent to the output merger, where it is resolved against the results from other per-fragment calculations from other primitives covering the same sample. This can be done in a number of ways, e.g., the output merger can compare the depth of each fragment, or it can calculate a weighted blend of the colors.

A key advantage of rasterization is that not only are the loops on Line 1 and 4 in Algorithm 1 highly parallelizable, it is also possible to pipeline the necessary calculations to a large extent, as there is no need for any global knowledge of other geometry. Each vertex can be projected knowing only its own position and the applied transformations and the projection, and each fragment can calculate its color with the data from the primitive generating the fragment, and globally set parameters, i.e., without any data from any other primitives or fragments. It is only at the output merger, i.e., the final stage which receives the calculated color and determines how to handle it, that there can be more than one sample to handle. This occurs when multiple primitives overlap the same sample position, and the calculated colors need to be resolved to a single color, in most cases by retaining the color of the closest primitive.

As the methods used to optimize rasterization preclude inter-primitive dependencies, a number of optical effects, e.g., inter-primitive light exchange and shadows, are impeded. However, many solutions to these problems have been developed, usually by employing multiple rendering passes. An example is shown in Figure 8, where shadows are implemented using stencil shadow volumes [10].

1.2.1 Rendering pipeline

A modern rendering pipeline for hardware-accelerated rasterized graphics, e.g., the OpenGL version 4.4 graphics pipeline [11], shown simplified in Figure 9, consists of a number of fixed-function hardware units, with programmable shader stages in between. Several of these stages correspond directly to different lines in Algorithm 1, while others introduce new functionality. I will use OpenGL terminology in this section, however other modern graphics APIs, e.g., Direct3D 11 has a similar pipeline and similar capabilities.

1.2.2 Fixed function stages

The fixed-function stages of the OpenGL pipeline, blue in Figure 9, are usually implemented as specialized hardware, allowing a high level of optimization, both in regard to execution time and energy consumption. However, because of this specialization, they are inflexible.

First, the vertex specification stage collects values from buffers and merges the values into vertices. The values can be collected from several specified buffers, containing either vertex data or indices, that are used to select values from the vertex data buffers.

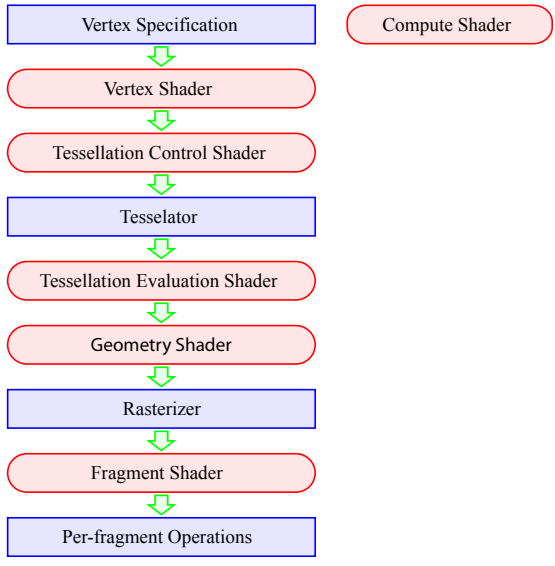


Figure 9: A simplified schematic of the OpenGL version 4.4 pipeline, showing the various fixed function and shader stages. Fixed function stages are shown in blue, while programmable shader stages are red.

The tessellator unit subdivides each input primitive, and is used for creating more detail on models. An example of a tessellated triangle can be seen in Figure 10.

After receiving three vertices, with their projected positions, as input the rasterizer determines the set of samples contained within the triangle and generates the corresponding fragments. It also calculates the barycentric coordinates and uses them to interpolate values from the three vertices to each fragment, e.g., the depth of the fragment.

Finally, the per-fragment operations perform the final operations on each fragment, and write the result to the frame buffer. The final operations can be a selection of actions, e.g., blending with the existing value in the frame-buffer or using a depth comparison to determine what action to perform. The latter is often used to retain the fragment closest to the camera, hiding surfaces farther away. Although this test is conceptually placed at the end of the pipeline, most hardware perform the test earlier when possible, and discard the fragment before invoking the fragment shader to reduce the amount of calculations performed.

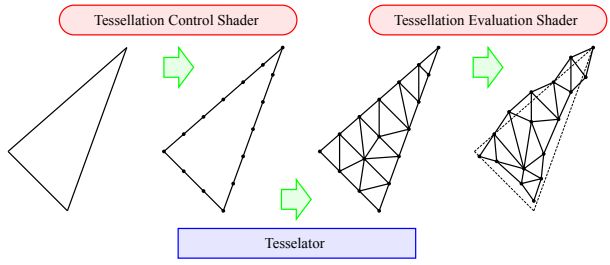


Figure 10: The tessellation control shader calculates a set of tessellation levels. The tessellator divides the triangle into several new triangles, according to these levels. The tessellation evaluation shader calculates the position of each newly created vertex.

1.2.3 Shader stages

The shader stages of the pipeline, red in Figure 9, are programmable using the GL Shading Language (GLSL) [12], and are thus more flexible than the fixed function stages.

The vertex shader stage is responsible for calculating the screen-space positions for the rasterizer. However, it is often used to calculate many other parameters that are interpolated over the triangle and used in the fragment shader. Mostly, the calculation of the screen-space position is as simple as a matrix-vector multiplication, other times it is more advanced, e.g., when ”skinning”, i.e., each vertex is transformed with several transformation matrices and a blended result is used. Usually these transformations are connected to different bones in an animated skeleton, and are used for character animations.

The tessellation control shader calculates the tessellation levels for a primitive that is to be tessellated. These tessellation levels control the number of divides for each edge of the primitive, demonstrated in Figure 10. Furthermore, it has the possibility to calculate and send values directly to the tessellation evaluation shader.

The tessellation evaluation shader evaluates each vertex generated by the tessellator. It uses the primitive, and the interpolation parameters of new vertex on the primitive, to calculate a position, e.g., using bi-cubic interpolation over a quadratic primitive to achieve a smooth surface or by moving it according to a height-map texture. See Figure 10.

The geometry shader is a programmable stage capable of geometry multiplication. As input it receives a single primitive and as output it submits zero or more new primitives. It can be used, among other things, for multiplying a primitive to be rendered to the six sides of an environment cube map in one pass.

The fragment shader is responsible for calculating the per-fragment values sent to the per-fragment operations stage. The value is commonly a color calculated by

evaluating a bidirectional reflectance distribution function for a set of light sources and by sampling one or more texture maps.

Finally, the compute shader is a general computations shader, with user-specified input or output, that can be used to perform a wide range of parallelizable computation on the GPU.

1.2.4 Stochastic rasterization

An interesting further development of rasterization is the introduction of stochastic rasterization. Recall the different camera models introduced in Section 1.1.1, where cameras with two to five dimensions where discussed. Similarly, rasterization can render using camera models with more than two dimensions. This is usually done by introducing an n-dimensional sample pattern, and by testing these samples against special edge equations that take the higher dimensions into account [13].

Unfortunately, when using stochastic rasterization, a larger number of samples may be required to reach an acceptably low noise level, and the rasterization cost for each sample is higher than with regular rasterization. Therefore, it is not commonplace for real-time rendering, and no hardware implementation is currently available. This may change in the future.

1.3 Energy in Computer Graphics

We are entering a time when more and more of our technology is mobile and, as an effect of this, battery powered. At the same time, we expect more advanced computer graphics. This means that the power and energy consumption of our devices is already a large concern, and will probably be more so in the near future. For this reason, energy consumption of computer graphics is the primary topic of my thesis.

Another development to note is that an increasing number of units are integrated on a single, multi-capable chip. In the case of computers, the chip that formerly only contained a CPU can now contain a CPU, a GPU, and the memory interface. For smart phones, the central chip is commonly known as a System on a Chip (SoC), a name that implies its multi-functionality. It often contains a multitude of blocks, e.g., CPU, GPU, modems, radio capable blocks for cellular connectivity or WIFI, and hardware video decoders and encoders. A block diagram of an SoC, the Intel Atom Z2760, is shown in Figure 11 as an example, in which we can see the large number of blocks with different functionality.

1.3.1 Dark silicon and other constraints

For a long time, the development of chip manufacturing has been following two predictions. First, “Moore’s law” stated by Gordon Moore [14], predicting that ev-

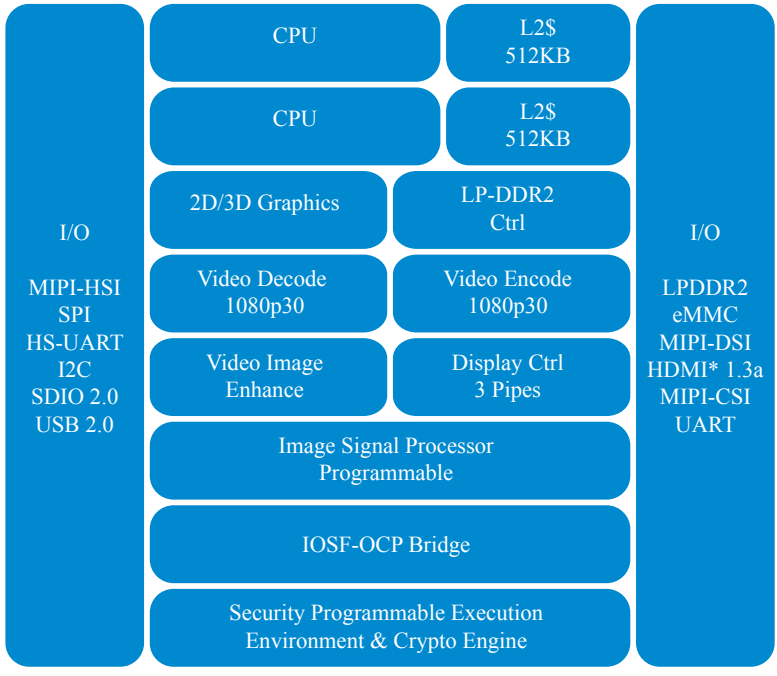


Figure 11: Block diagram of an Intel Atom Z2760 System on a Chip .

ery second year[1], the number of transistors in integrated circuits doubles. Second, Dennard scaling [15] which predicts that, as the number of transistors on a given area increases, the power density, i.e., the power per unit volume is constant.

However, with shrinking process nodes, Dennard's prediction fails. As transistors get smaller, the necessary transistor switching power does not decrease as fast and the current leakage grows exponentially, and thus the power density increases [16]. As the power density reaches levels higher than the chip can dissipate as heat, it is no longer possible to retain a stable temperature, and the power density must be lowered to prevent overheating the chip. As a result, chips are already encountering the utilization wall [17], where the entire chip cannot be run at full speed. The growing gap between the number of transistors it is possible to create on a given surface and the number of transistors that can be active simultaneously, predicted by Esmaeilzadeh et al. [18], is called dark silicon.

[1]Later revised to every 18 months.

1.3.2 Energy reduction methods

With dark silicon and heat dissipation as major constraints on chip design, energy efficiency is increasingly important. Much research has been done in the area from a multitude of angles. Here I will list the most important work concerning energy in relation to computer graphics and graphics processing units.

Most academic work regarding energy efficiency for computer graphics has focused on designing more energy efficient hardware. There have been multiple approaches, e.g., saving energy by performing operations with reduced precision [19, 20, 21], power-gating techniques [20, 22], improvements on the memory hierarchy [23], and by dynamically scaling the voltage and frequency [24].

However, in recent years, there has been several articles emphasizing the ability to evaluate the energy usage of software, and affect the usage with changes to the software. Collange et al. [25] measure the power usage while executing a set of matrix operations on random 1024×1024 matrices using CUDA [26]. It uses the result to extract the approximate energy consumption of individual instructions. They also examine how different memory access patterns affect the energy consumption. Similarly, Ma et al. [27] measured three different mobile platforms running two different games and by disabling various stages of the graphics pipeline, isolated the rendering time and power consumption of the different stages.

1.3.3 Simulating energy usage

Another interesting research area concerning computer graphics and energy consumption is to use simulations. Several simulations have been developed and studies, e.g., Sheaffer et al. [28, 29] developed Qsilver, a GPU simulator and also used it to examine the effectiveness of several different power reduction methods. Pool et al. [30] developed and trained a model on real workloads, and were able to predict the energy consumption of non-existing hardware designs.

1.4 Multi-View Rendering

In a number of use cases, it is necessary to render a scene from several viewpoints simultaneously, e.g., stereo-rendering for 3D displays or virtual reality headsets. This is known as *multi-view rendering*, and although the most common case is stereo, it is not limited to only two views in general. An example of a use case with more than two views is rendering for auto-stereoscopic displays. Auto-stereoscopic displays require multi-view rendering, as they present different images in different directions. Although general multi-view rendering does not have restrictions on the views, most auto-stereoscopic displays require views from a *horizontal parallax multi-view camera*.

Multi-view rendering is the secondary topic of my thesis. Also, as rendering with a horizontal parallax multi-view camera for auto-stereoscopic displays was the target of our multi-view research, this section will be limited to that camera model.

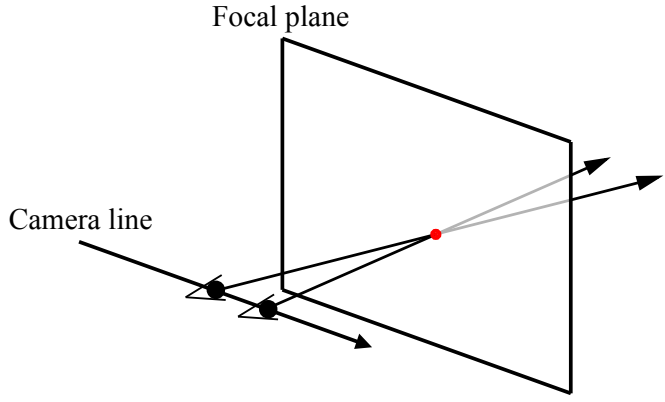


Figure 12: Horizontal parallax multi-view camera.

1.4.1 Horizontal parallax multi-view camera

In this restricted form of multi-view rendering, the conceptual camera model's (in relation to the camera models in Section 1.1.1) focal plane is limited to lie in the plane of the display, and the lens is a one-dimensional line in front of the focal plane, usually called a camera line. Figure 12 shows the camera setup with a focal plane and a camera line, showing two rays intersecting at the same point on the image plane, but originating from different positions on the camera line.

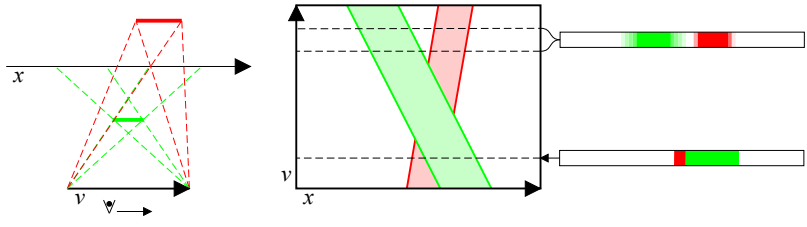


Figure 13: Horizontal parallax multi-view camera rendering two pieces of geometry, a green line in front of the focal plane and a red line behind the focal plane. Left: a simplified camera setup, with a one-dimensional image and a camera line, with the geometry displayed. Middle: the resulting epipolar plane, where it is possible to see how the lines' projected positions move as a function of the position on the camera line. Right: two filtered views from the epipolar plane, the top one filtered over a range of the camera line to avoid inter-perspective aliasing, while the bottom one is a sharp image from a single point on the camera line.

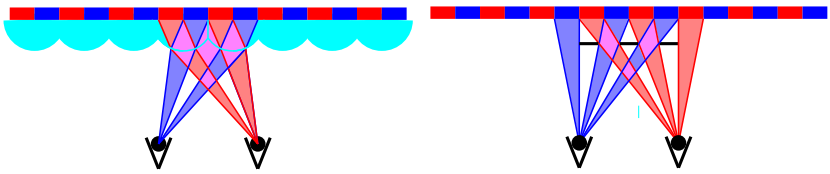


Figure 14: Left: a lenticular sheet display. Note how the different pixels are refracted to different directions. Right: a parallax barrier display, where a different set of pixels are visible from each eye position.

A simplified version of the camera model can be seen in Figure 13, left, with x being a dimension in a one-dimensional image and v being the dimension reflecting the position on the camera's line. Figure 13, middle, shows the rendering of an *epipolar plane* [31], which illustrates how the image (over x) changes as a function of the position on the camera line. For a given point on the camera line, the result is a sharp image, similar to an image rendered with a pinhole camera model. With our simplified setup in Figure 13, a one-dimensional image can be seen in the bottom right. However, to combat inter-perspective aliasing [32], a phenomena experienced when moving sideways in front of auto-stereoscopic displays with a limited number of views, it is necessary to filter across a section of the v-axis, blurring geometry away from the focal plane. An example can be seen in Figure 13 (top right), where both the red and the green geometry is blurred (in contrast to the sharp, unfiltered image below), as neither lies on the focal plane.

1.4.2 Auto-stereoscopic displays

Auto-stereoscopy is achieved when a display is, at each pixel, transmitting different colors in different directions, where each direction correspond to a different view. As a simplification, it is common to only change the color for horizontal directions, as our eyes are generally held at the same height. This results in each eye receiving a different image, and if those images form a stereoscopic image pair, the viewer experiences a stereoscopic image, i.e., a three-dimensional experience.

There are a number of techniques to achieve auto-stereoscopy in displays. Two commercially available methods are lenticular auto-stereoscopy and parallax-barrier auto-stereoscopy [33].

Lenticular auto-stereoscopy, shown to the left in Figure 14, creates the auto-stereoscopic effect using a lenticular sheet, i.e., a sheet of semi-cylindrical lenses on top of an image source, e.g., an image or a display, where each lens is covering a number of pixels. Then each of the different pixels under a single lens is dispersed in a different direction through the lens.

Parallax-barrier auto-stereoscopy, on the other hand (Figure 14, right), creates the auto-stereoscopic effect by placing a number of vertical barriers at some distance in front of the display. Only a subset of pixels are visible from a position in front of the display, and that subset differs according to the direction between the display and the viewer's position.

1.4.3 Multi-view rendering techniques

Both rasterization and ray tracing can be used to create multi-view images. In the rasterization case, there are several of ways to do it. The naïve way is to render each view independently using a sheared pinhole camera, i.e., for each view placing the camera position at a specific point on the camera line and render an image. However, as described by Halle [32], this would result in inter-perspective aliasing. A solution would be to render a large number of images for each view, with different positions on the camera line, and use an average, a technique known as accumulation buffering [34]. As seen in Figure 13, with such a solution, an object at the focal plane would be sharp, as they appear at the same position in the image plane regardless of the position on the camera line. For an object away from the focal plane, on either side of it, the object would appear blurred. With an increasing number of renderings per view, this converges to a correct solution.

Another solution would be to use stochastic rasterization, as described in Section 1.2.4, in which case it can be described as a variant of stochastic rasterization for motion blur, i.e., it has a sample space of three dimensions. For stochastic multi-view rasterization, it is possible to use the time-continuous triangles described by Akenine-Möller et al. [13], where the triangles become view-continuous instead.

Using ray tracing, it is a matter of creating rays using the horizontal multi-view camera model, and accumulate the traced results in a number of views.

2 Contributions and Methodology

This thesis presents my research, which is published in six articles. The research can be divided into two categories, namely *a*) examining the energy consumption of real-time graphics algorithms, and *b*) exploring two methods for rendering multi-view image sets for auto-stereoscopic displays.

My main focus has been on energy consumption, where I have co-authored four papers, Paper I-IV. Two of the papers, Paper I and Paper IV, are case studies, where we executed and measured a set of workloads with a variety of settings and compared the results. For these papers, we also connected a data acquisition device and intercepted the current, and in some cases also the voltage, provided to a platform to obtain a power measurement. The measured values were then compared or related to additional collected data, and we were able to draw conclusions on the characteristics of the platforms' or algorithms' energy usage. In Paper I, we examined the energy consumption of several algorithms running on a range of platforms, from a mobile phone to state of the art discrete graphic cards. In Paper IV, we only explored a single platform, on which we examined four different algorithms over a large range of values on several settings. We also used Pareto frontiers as a means to limit our analysis to only those measurements which are interesting, either from an energy perspective, or from a rendering time perspective.

Paper II is a short report discussing which is the proper unit to report when measuring energy of computer graphics. In Section 3.1, I describe the development of our method between Paper I and Paper IV, based on our experience when measuring energy. The resulting method is described in detail in Paper III, which also documents a number of unexpected behaviors, that can interfere with energy measurements on the platforms we have used.

Each of these papers are summarized in Section 3, listing the most important conclusions drawn and also stating my individual contributions to each paper.

I have also co-authored two papers that concern multi-view rendering. Paper V is a pure optimization paper, where we developed a new and efficient algorithm for ray tracing a set of multi-view images. Our new algorithm is then compared against state of the art solutions with respect to rendering time and image quality. Paper VI however, is not solely a multi-view paper. It is a forward-looking paper, examining how a proposed higher-dimensional hardware rasterizer, aimed at rendering with motion and defocus blur, can be exploited for alternative uses. We propose a set of other, novel, use cases, for which we then examine the feasibility. Single-pass multi-view rasterization is one such use case.

Section 4 summarizes Paper V and Paper VI, and also states my contributions to each paper.

Although there is little connection between the research categories, our research in multi-view rendering should help to lower the energy consumption of rendering a set of images. In Paper V, we render at up to an order of magnitude faster, which likely reduces the energy consumption greatly. Also, one of the primary rea-

sons for implementing rasterization in hardware is to reduce its energy consumption. Using a higher-dimensional rasterizer implemented in hardware for rendering multi-view images would certainly be more energy-efficient than implementing it in software.

3 Energy Efficiency for Computer Graphics

My research in energy consumption for computer graphics focused on two areas, namely *a*) energy consumption of different algorithms and platforms, and *b*) improving measurement and analysis methods. This section will describe our measurements methods and their development in more detail, followed by short descriptions of the energy consumption studies that we have performed.

3.1 Tools and Methods

We have performed two studies, described in Papers I and IV, on energy efficiency of real-time rendering algorithms. For these studies, we used two different measuring methods, namely the recorded timestamp method (Section 3.1.1) and the semi-automatic timestamp generation method (Section 3.1.3). These methods and the development leading to the semi-automatic timestamp generation method are described later in this section. The second method is also detailed and examined in Paper III. For each method, we also developed a set of tools for measuring, processing, and evaluating the energy consumption, with significant improvements between the two different methods.

3.1.1 Recorded timestamps method

In the first method, we use a data acquisition hardware that records current with a sampling rate of 40 kHz, and a set of software tools to interpret the data. It is dependent upon having the measured workload providing timestamps surrounding each frame, and thus requires access to the source code of the workload. Therefore, we call this method "the recorded timestamps method", and it was used in Paper I.

To obtain the energy used by different algorithms on different platforms, the method requires five steps to measure, synchronize, and integrate the recordings:

- Measure the current of the platform.
- Calculate the power from the measured current and a fixed voltage.
- Record timestamps.
  - Before each frame.
  - After each frame is finished rendering, guaranteed by calling `glFinish()`.
- Synchronize the timestamps to the calculated power.
- For each frame, integrate the power between its timestamps.

The measured workloads in Paper I was developed especially for that project, and the workloads could therefore easily record the necessary timestamps. We then

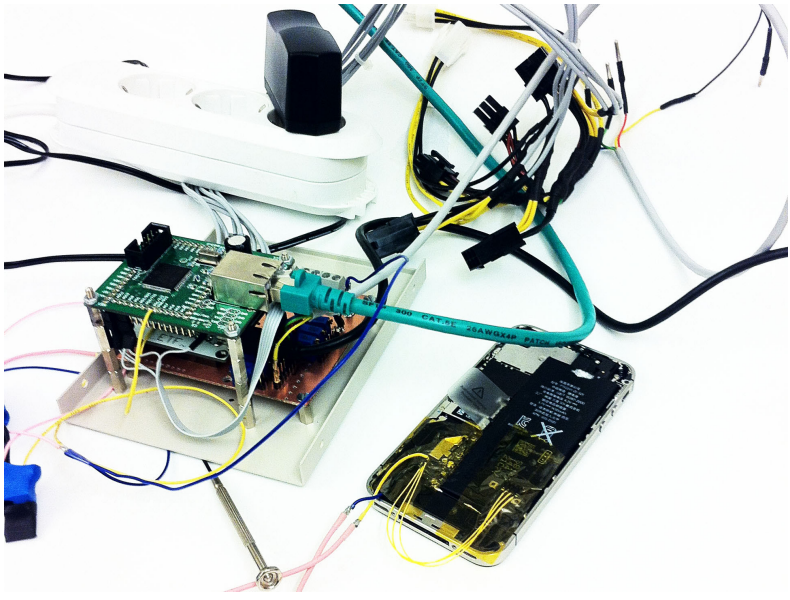


Figure 15: Our first data acquisition hardware (left), connected to an iPhone 4S (right).

used a collection of Matlab scripts to perform the other operations: calculate the power, aid visual synchronization of the power and the timestamps, and integrate the power.

Our first custom data acquisition hardware, shown in Figure 15, is based on an ARM Cortex-M3 processor. It has two different types of current sensors, which handle different types of platforms. It uses a set of four ACS710 Hall effect current sensors, capable of measuring up to 12 A. We use these to measure discrete graphics cards, and CPUs with integrated GPUs. It also contains two shunt current sensors, measuring up to 1 A, suitable for the current of cellular phones. All sensors are fed to an A/D converter, which further feeds into the processor. The processor then samples the current at 40 kHz. The sampled signal is sent over Ethernet to a host computer, which runs a receiving Python script, that saves the signal to a file for later examination.

In our first method, we obtained two sources of raw data for each measurement. The first source is the current, from which we computed the power, given the known, fixed voltage of the platform. For measurements where the platform was supplied with power from several sources, e.g., most discrete graphics cards receive power both from the PCI Express bus and from PCI Express auxiliary power connectors, all channels are transformed into power and then added together for a total power consumption. Second, the measured workload recorded a set of times-

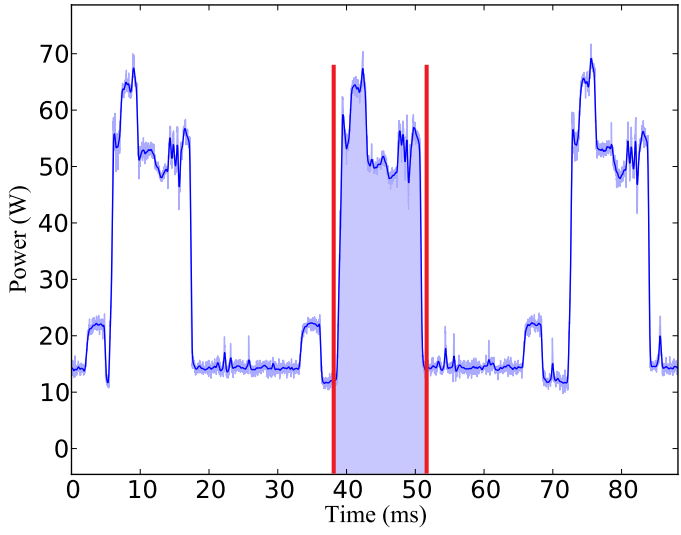


Figure 16: The power curve of three consecutive frames, showing the timestamps enclosing the middle frame as in the recorded timestamps method.

tamps, one before and one after each frame. However, the two sources of data do not share a common, known timeframe, which makes it impossible to extract energy values per frame without aligning the timeframes.

A major task performed on the collected data, was aligning it to a shared timeframe, necessary to enable the extraction of per-frame data. As the extent of a rendered frame is visible on a power curve, as can be seen in Figure 16, it is a matter of selecting appropriate samples that correspond to at least two timestamps, and with these as anchors, align the timeframes. With these sources of data synchronized, extracting per-frame energy is trivial, i.e., a summation of all samples scaled by the inverse of the frequency.

For our first project that involved measuring the power and energy for different rendering algorithms, we chose to perform the synchronizing, i.e., relating the timeframes, utilizing Matlab's plotting functionality. This was done by plotting the calculated power curve in Matlab and by selecting specific samples thought to correspond to the beginning of the first and the last frame. Simultaneously, the scaled and translated set of timestamps were displayed. By visually inspecting how the timestamps aligned with the featured frames in the power curve, it was possible to select the correct samples corresponding to the two timestamps and thus synchronize the timeframes.

Matlab was also used to integrate over the power curve to obtain an energy value per frame, and to visualize the energy consumption over the frames of the workload.

3.1.2 Method development

The recorded timestamps method was used for several projects, e.g., in Papers I and II, and as a result, several problems and weaknesses became apparent. This lead to the development of a new, improved method. I will here describe the problems and weaknesses that were encountered, and our proposed improvements.

While measuring and processing data as described in the recorded timestamps method, we encountered three major problems:

- The data acquisition hardware sampled and transmitted the signal without storing it locally or validating that the transmission to the host computer was successful. With the relatively high bandwidth required, this led to dropped packages over the connection. However, as the receiving software warned when it encountered dropped packages, each occurrence was detected. Unfortunately, when a dropped package was detected, it was not possible to synchronize the measurement, as parts of the curve where missing and, as a result, later parts of the curve were shifted in time. The only solution was to redo the entire measurement, which was highly time-consuming.

- The data acquisition hardware transmitted a fixed number of measured channels, which the receiving Python script recorded to disk, even if only a subset of the channels contained useful data. As a result, our longer measurements were recorded in unnecessarily large files. Also, for cases where fewer channels were actually required, the higher bandwidth contributed to a higher risk of dropped packages.

- Using the plotting functionality of Matlab proved to be frustratingly slow for large data sets. Our method of finding the correct samples for synchronizing the timeframes relied on frequent inspection of the curve in both detail and with overview. However, when the curve consisted of hundreds of millions of samples, each zooming maneuver could take up to 40 seconds to complete, which increased the time required for synchronizing immensely.

In addition to encountering severe problems, we also identified a number of weaknesses in the method, which we wanted to remove.

- The first generation data acquisition hardware is only capable of recording current. In our measurements using the recorded timestamps method, we calculated the power and energy using either the voltage as written in the appropriate standard specification documents [35, 36] or, in cases with a custom power supply unit, with a stable, fixed voltage. However, in the

cases of using a standard specification, they also state that the power supplies have an allowed error marginal, in some case up to 8%. This is a large and unnecessary source of error.

- The data acquisition hardware had a fixed measurement frequency of 40 kHz. In some cases, this is not necessary, and produces an unnecessary amount of data. In other cases, it may not be enough to capture high-frequency features of the energy consumption.
- The method for extracting per-frame energy is dependent on having timestamps at the beginning and at the end of each frame supplied by the measured workload. This requires access to the source code of the workload, and changes, albeit small, to the workload to record timestamps. Furthermore, to ensure that all rendering is performed for a specific frame, a call to `glFinish()` was performed before recording the ending timestamp, blocking the rendering thread until all rendering commands had finished. As shown in Paper III, calling `glFinish()` can substantially alter the power usage of a workload, and skew the results.
- The timestamps for enclosing a frame do not include the idle time to the next frame. This has two undesirable effects. First, any benefit of a lower idle power is disregarded. Second, the summed energy result of a sequence of frames does not give the energy consumption of the platform for that period of time.

From these encountered problems and weaknesses, we concluded that for our improved method we wanted:

- A new data acquisition hardware that:
  - Records both current and voltage.
  - Can record at a configurable frequency.
  - Records only the necessary data.
  - Has a more stable recording, i.e., it does not drop data.
- New software that:
  - Is faster and more interactive.
  - Does not rely on recorded timestamps.
  - Visualizes more aspects of the data.

3.1.3 Semi-automatic timestamp generation method

Our second generation method is based upon a new data acquisition hardware and the insights we made while measuring with our first generation hardware.

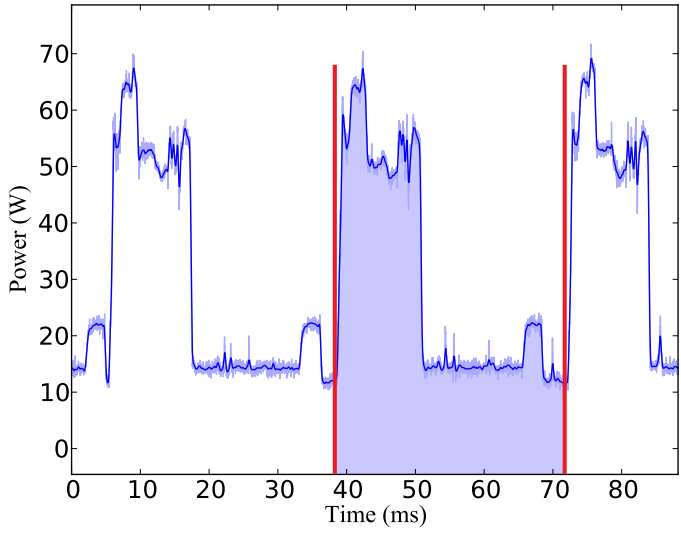


Figure 17: The power curve of three consecutive frames, showing the timestamps enclosing the middle frame for the semi-automatic timestamp generation method.

Similar to the first method, the improved method included three main tasks: measuring, identifying individual frames, and integrating. However, it differs on several points. First, we measure both current and voltage for CPUs with integrated GPUs.[2] Another improvement of the actual measuring is that by having a new and improved DAQ, the frequency is configurable. Second, while the first method had the workload record timestamps and synchronize them with the power curve, the new method has the possibility to find the frame beginnings from the power curve using a semi-automatic feature search and generate a set of timestamps. Finally, instead of having two timestamps per frame, at the beginning and the end, it only has a single timestamp per frame, at the beginning, and integrates from this until the timestamp of the subsequent frame, as illustrated in Figure 17.

Our second data acquisition hardware is a more refined solution, shown in Figure 18. It is a modular design based around a Raspberry PI computer, where different sensors can be attached for different measurements. It is not constrained to measuring current and as it supports up to eight channels, can record both current and voltage simultaneously, which allows for a measurement with fewer sources of error. With its modular design, it also has the potential of measuring other types of data, by developing other measurement modules. The station also has substantial

[2]For cellular phones, we use a custom power supply with a fixed, stable voltage, and thus measuring the voltage is not necessary.

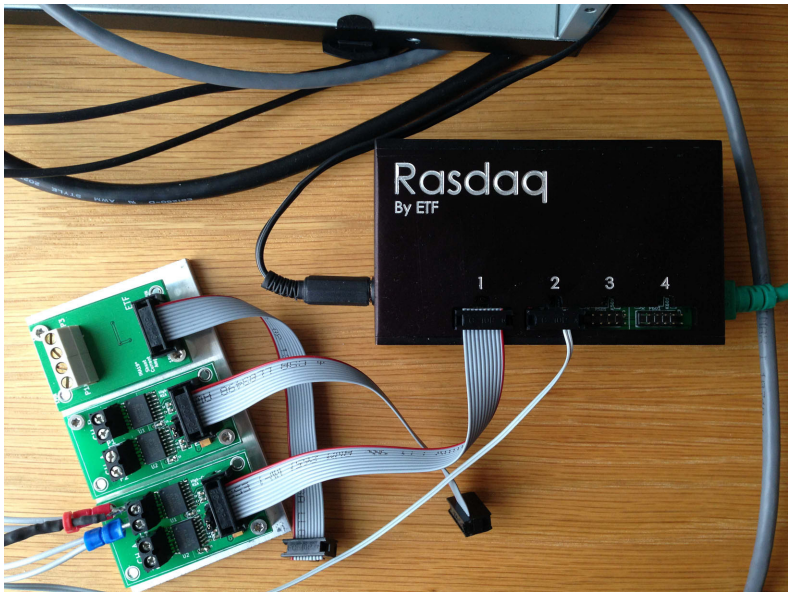


Figure 18: Our second, improved, data acquisition hardware named Rasdaq. Each slot numbered 1 – 4 supports 2 channels, for a total of 8 channels. Note also the connected module on the left for measuring current and voltage on desktop PCs.

internal storage, for recording the measurements locally, and transferring the data to the host computer after the measurements are completed.

Recording a measurement is done by running a command-line program called rdq-tool on the DAQ's CPU, accessed by using the common secure shell (SSH) protocol from a host computer. The program has several configurable arguments, e.g., frequency, number of channels to record, and number of samples to record before ending the measurement. An improvement over our first DAQ is the ability to sample at different frequencies. The new DAQ supports frequencies in the range 20 kHz – 200 kHz.

Similar to our first method, described in Section 3.1.1, we required software capable of plotting the power curves, as a means of inspecting our measurements. However, we would prefer faster and more specialized software. Furthermore, as a method to enable measuring per-frame energy of closed source application, we wanted to be able to find timestamps from the curve, without requiring the measured application to record the timestamps.

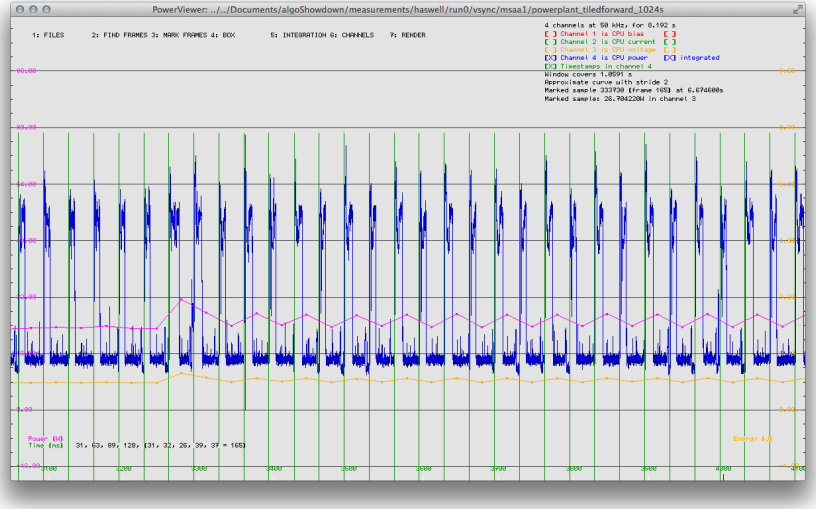


Figure 19: A screenshot from our software, showing a power curve and a set of frames from Paper IV, rendering 1,024 light sources with tiled forward rendering.

The main capabilities that we needed from the new software was:

- Interactivity.
- Functionality for finding frames.
- Extensive visualization.

Furthermore, we would like to have:

- Faster start-up time.
- Powerful graph creation.

Our power curve inspector was developed to match these specifications. It was developed in C, using OpenGL for interactive visualizations, and representative screenshots are visible in Figure 19 and Figure 20. The method for finding frames, matching feature using least square error, is described in Paper III. Besides interactive visualization, it is also able to create a number of different graphs, e.g., with energy per frame, power curve over a short segment, mean power per frame, and rendering time per frame. These are output as PDF files, suitable for direct use in documents.

The resulting measurement method for obtaining per-frame energy is described in Paper III, which was written by myself and Tomas Akenine-Möller. Paper II is a

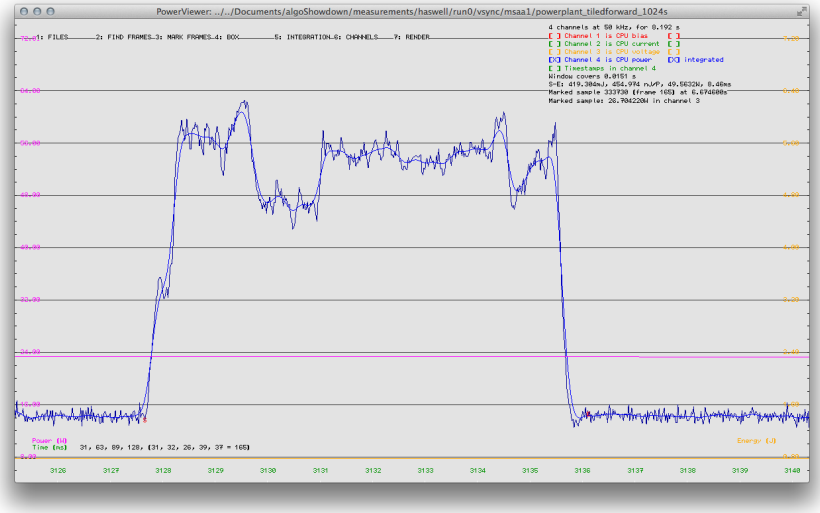


Figure 20: A screenshot from our software, showing the power curve and a single frame from Paper IV, rendering 1,024 light sources with tiled forward rendering. A curve segment is selected between the S and E markers, enclosing the plateau of the frame, with statistics for that segment displayed to the upper right, saying "S-E: 419.304mJ, 454.974nJ/P, 49.5632W, 8.46ms", where "nJ/p" denotes nano-Joules per pixel.

discussion on what metric to use when measuring energy in computer graphics, and the implications of different metrics. It is an extension and emphasis on a similar reasoning in Paper I.

3.2 Research Studies

We have performed two research studies of computer graphics from an energy perspective. The first study (Paper I) is an examination of energy consumption over a range of *algorithms*, which solve the same problem, running on a collection of *platforms*. The second study (Paper IV) is an in-depth examination of a single platform, with detailed measurements of different *algorithms*, with a range of *settings*, again solving the same problem.

3.2.1 Platform overview

In our first study of power and energy efficiency of graphics algorithms, we examined two basic problems of computer graphics, namely primary visibility shading

and shadow rendering. We implemented three different *algorithms*, for each problem:

- Primary visibility shading:
  - Forward rendering.
  - Forward rendering with Z-prepass.
  - Deferred shading.
- Shadow rendering:
  - Stencil shadow volumes.
  - Shadow maps.
  - Variance shadow maps.

We then ran these algorithms on four different *platforms*:

- High-end discrete graphics cards:
  - NVIDIA GeForce GTX 580.
  - AMD Radeon HD7970.
- CPU with integrated GPU:
  - Intel HD 3000, integrated in an Intel Sandy Bridge Core i7 2700K.
- Mobile phone GPU:
  - PowerVR SGX543MP2, in an iPhone 4S.

We ran these algorithms on two scenes, one for the primary visibility shading, and one for shadow rendering, both scenes with a camera-path over a fixed number of frames. Simultaneously, we measured the current of the platforms and had the application record timestamps, according to the recorded timestamps method in Section 3.1.1.

As this enabled us to obtain per-frame values of both power and energy, we were able to draw conclusions regarding how each algorithm behaved on the different platforms. For example, we found a clear case where rendering time and per-pixel energy where disconnected, as can be seen in Figure 3 of Paper I, where forward rendering (FR) and deferred rendering (DR) have similar rendering times, but significantly different rendering power.[3] Furthermore, we saw that, although the different platforms can have more than an order of magnitude difference in

[3] Disconnection between rendering time and energy consumption on graphics hardware was shown by Pool et al. [30], for a smaller example algorithm on simulated hardware, however, we were unaware of this at the time of our research. Furthermore, we showed the disconnection for full rendering algorithms on mainstream hardware.

rendering power, the energy consumption per pixel was in many cases within only a $2\times$ difference.

The research in Paper I was performed as a joint project between myself, Per Ganestam, Michael Doggett, and Tomas Akenine-Möller.

3.2.2 In-depth study of a single platform

Our second energy study (Paper IV) is, in contrast to Paper I, focused on a single platform. Instead we compared different *algorithms* over a range of different *settings*. The type of algorithms is focusing on real-time many light rendering. We measured four different *algorithms*:

- Forward rendering with Z-prepass.
- Tiled forward rendering.
- Tiled deferred rendering.
- Visibility buffer rendering with tiled light sources.

We ran these while altering a number of settings, i.e., with and without V-sync, different MSAA rates, different number of light sources, and rendering scenes with different amounts of geometry. Similar to Paper I, we measured per-frame energy and power, but, this time using the semi-automatic timestamp generation method, described in Section 3.1.3. In addition, we collected further data, e.g., read and write bandwidth, shader execution times, and shader stalls using the Intel Graphics Performance Analyzer (GPA).

We observed that our measurements had a significant variation in the mean power for the rendered frames, suggesting that there might be a possible trade-off between rendering time and energy. To be able to draw any conclusions, we sorted our measurements based on settings, i.e., we examined groups by varying only the algorithm, with all other parameters identical. In each of those groups, we created Pareto frontiers based on their rendering time and per-frame energy, to be able to distinguish in which cases it was possible to trade energy for rendering time. However, we discovered that there where very few cases (only 4%), where a possible trade-off existed. Consistent with that discovery, when we limited our analysis to Pareto efficient values, the variation in power significantly diminished. This suggests that although the mean power varies with changing parameters, there is a only small power range with optimal efficiency.

The research in Paper IV was performed by me, supported by my supervisor, Tomas Akenine-Möller. The implementation of the measured algorithms is based upon Andrew Lauritzen's framework [37], already containing forward with Z-prepass and tiled deferred, to which I added implementations of tiled forward and visibility buffer.

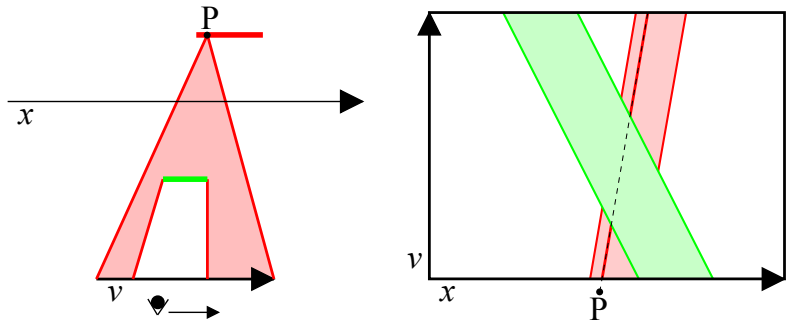


Figure 21: Shading coherence with a horizontal parallax multi-view camera. A point in the scene becomes a line in an epipolar plane as it is projected toward different positions on the camera line.

4 Multi-View Rendering

I have co-authored two research papers on the topic of multi-view rendering. In Paper V, we developed an optimized algorithm for ray tracing multi-view rendering and Paper VI is a study of alternative uses for a higher-dimensional rasterizer, where rasterized multi-view rendering is one such example.

4.1 Multi-View Ray Tracing

In our first multi-view research project, we optimized multi-view ray tracing with adaptive sampling, based on Hachisuka et al.'s [38] multi-dimensional adaptive sampling. We do this by exploiting sample space coherency, i.e., for parts of the sample space, the shading is either constant or varies slowly and can be approximated. The coherence we exploited is mainly a result of the constrained horizontal parallax multi-view camera model. In that model, a position in the scene becomes a straight line in an epipolar plane of the sample space, as illustrated in Figure 21.

Unfortunately, the shading of a point is usually dependent upon the direction towards the camera, i.e., it is view-dependent, and the shading differs for different points on the camera line. However, the shading can be separated into a view-dependent section and a view-independent section, where the view-independent section can be reused for several samples. Furthermore, there can be sections of the shading with weak view-dependence, where the shading for the entire camera line can be approximated using either a single sample on the camera line, or by using a few different samples and linearly interpolating the shading.

Paper V also introduces multi-view silhouette edges that are used to improve the filtering of the sample space into images. Multi-view silhouette edges are the edges in the scene that are silhouettes from some point on the camera line. These edges becomes bi-linear patches in sample space, which describe discontinuities over which no filtering should occur.

The research in Paper V was performed as a joint effort by myself and Magnus Andersson, supported by Tomas Akenine-Möller, Jacob Munkberg, Petrik Clarberg, and Jon Hasselgren. The initial idea for this research came from Tomas Akenine-Möller, Jacob Munkberg, Petrik Clarberg, and Jon Hasselgren, however Magnus and I extended it with many of the central ideas, e.g., multi-view silhouette edges.

4.2 Multi-View Rasterization

Paper VI is based upon having a supposedly, efficient multi-dimensional rasterizer implemented in hardware. It describes the pipeline of this proposed rasterizer and examines a number of forward-looking use cases.

The examined use cases include collision detection, caustic rendering, optimized sampling, glossy reflections and refractions, motion blurred soft shadows, and multi-view rendering. My contributions are mainly focused on glossy reflections, glossy refractions, and multi-view rendering.

Section 6 in Paper VI on glossy reflection and refraction approximates a reflective and refractive effect in a planar surface by rendering from a reflected or refracted point of view to a render target, and then using that render target as texture for the selected surface [2]. By using a stochastic rasterizer for defocus blur, it is possible to adapt the focal plane and the lens to approximate a glossy effect with contact hardening in a single pass.

Section 8 in Paper VI on multi-view rendering with a higher dimensional rasterizer describes how such a rasterizer's ability to render with defocus blur can be extended to rendering horizontal parallax multi-view images. This can be accomplished in a single pass by extruding the lens to encompass the entire camera line, and use a selective resolve to extract several images from a single multi-sampled render target. It also describes rendering stereoscopic image pairs in a single pass similarly, which furthermore can be optimized by expanding the tile-triangle test to support two disjoint square region on the lens, one for each view.

Paper VI was a joint research project, where my contributions are the section on glossy reflections and refractions and the section on multi-view rendering. However, I do not consider the section on glossy refractions and reflections an integral part of this thesis, as it is unrelated to my two major research areas.

5 Conclusions and Future Work

My research has had two goals, namely, to design efficient algorithms and to understand how different algorithmic solutions affect the energy consumption of performing a rendering task on graphics processors. My second goal led me further into devising and evaluating methods for measuring and analyzing energy consumption for real-time computer graphics.

From my research in the energy consumption of computer graphics, I have drawn two sets of conclusions. First, from our case studies, it is clear that there are cases when there is no reliable correlation between rendering time and energy consumption. It is therefore ill-advised to solely use rendering time as an indicator for the energy consumption, and in extension, the battery time of a platform. However, I have also shown that for some platforms, it is possible to show that all efficient solutions lie within a narrow power range, which suggests that the platform is designed for efficient calculations at that power. Furthermore, I have identified cases where there is a clear disconnect between bandwidth and energy consumption, suggesting the great benefit of the large and deep cache hierarchies of current hardware.

Second, from all our measurements and analysis regarding power and energy, I have come to a number of conclusions regarding measurement methods, workload considerations, and reporting metrics. The most important ones are:

- It is beneficial to integrate between frame beginnings, as it requires fewer timestamps and allows summing frames together for a total energy consumption. The integrated energy also includes possible benefits of a lower idle power.
- There are several benefits of using *nano-Joules per pixel* instead of the commonly used *performance per watt*.
- It is important to carefully examine any changes to the workload, and preferably avoid them, as it is difficult to predict the effect on the energy characteristics of the workload.
- As a continuation on the previous conclusions, it is important to perform a number of measurements on the targeted platform, to learn its characteristics. By doing so, it is possible to identify and avoid unintended traits.
- Using Pareto frontiers to evaluate the possible rendering time and energy trade-offs is an appropriate method when evaluating energy measurements.

Regarding multi-view rendering, I conclude that there is a large amount of sample space coherence that it is possible to exploit to optimize rendering times. It is also beneficial to extract and use multi-view silhouette edges during filtering to decrease noise. Also, when designing future, higher-dimensional rasterizers, it would probably be a small task to extend them to support rendering stereo and

multi-view image sets in a single pass. Since this research was performed, there has been substantial research in light-field reconstruction techniques. We see great potential for using these techniques for improving the reconstruction of multi-view light-fields, and thus require a lower number of samples in the light-field.

As future work, it would be very interesting to extend the analysis in Paper IV with measurements on a platform with identical calculation capabilities, but with varying cache hierarchies, and focus on whether it changes how bandwidth affects energy consumption. Also, for future work, it would be interesting to use the performed analysis and the developed methods to design more energy efficient rendering algorithms.

However, the main goal would be to see the research concerning energy consumption of computer graphics continued. In that area, it would be interesting to see more resources for investigating energy consumption in computer graphics, be it better analysis software, convenient measurement hardware, or more hardware counters for power and energy in GPUs. Furthermore, I would like to see more reporting on energy consumption in computer graphics research, as the importance of energy will only increase in the future.

Bibliography

[1] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory To Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[2] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*. A. K. Peters Ltd., 3^{rd} ed., 2008.

[3] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, vol. 23, pp. 343–349, June 1980.

[4] J. T. Kajiya, "The Rendering Equation," *Proceedings of ACM SIGGRAPH*, vol. 20, pp. 143–150, August 1986.

[5] M. J. Muuss, "Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models," in *Proceedings of BRL-CAD Symposium '95*, June 1995.

[6] S. Woop, J. Schmittler, and P. Slusallek, "RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing," *ACM Transactions on Graphics*, vol. 24, pp. 434–444, July 2005.

[7] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing," *Computer Graphics Forum*, vol. 26, pp. 415–424, September 2007. (Proceedings of Eurographics).

[8] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark, "Toward a Multicore Architecture for Real-time Ray-tracing," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 176–187, 2008.

[9] C. Kolb, D. Mitchell, and P. Hanrahan, "A Realistic Camera Model for Computer Graphics," in *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pp. 317–324, 1995.

[10] E. Eisemann, M. Schwarz, U. Assarsson, and M. Wimmer, *Real-Time Shadows*. A.K. Peters, 2011.

[11] M. Segal and K. Akeley, "The OpenGL® Graphics System: A Specification," October 2013.

[12] D. B. John Kessenich and R. Rost, "The OpenGL® Shading Language," January 2014.

[13] T. Akenine-Möller, J. Munkberg, and J. Hasselgren, "Stochastic Rasterization Using Time-continuous Triangles," in *Graphics Hardware*, pp. 7–16, 2007.

[14] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics Magazine*, p. 4, 1965.

[15] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, Oct. 1974.

[16] S. Thompson, P. Packan, and M. Bohr, "MOS scaling: Transistor challenges for the 21st century," *Intel Technology Journal*, 1998.

[17] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pp. 205–218, 2010.

[18] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *38th International Symposium on Computer Architecture*, pp. 365–376, 2011.

[19] J. Pool, A. Lastra, and M. Singh, "Energy-Precision Tradeoffs in Mobile Graphics Processing Units," in *International Conference on Computer Design*, pp. 60–67, 2008.

[20] J. Pool, A. Lastra, and M. Singh, "Precision Selection for Energy-Efficient Pixel Shaders," in *High Performance Graphics*, pp. 159–168, 2011.

[21] J. Pool, A. Lastra, and M. Singh, "Power-Gated Arithmetic Circuits for Energy-Precision Tradeoffs in Mobile Graphics Processing Units," *Journal of Low Power Electronics,*, vol. 7, no. 2, pp. 148–162, 2011.

[22] P.-H. Wang, C.-L. Yang, Y.-M. Chen, and Y.-J. Cheng, "Power Gating Strategies on GPUs," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 3, pp. 13:1–13:25, 2011.

[23] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McCaughy, D. Patterson, T. Anderson, and K. Yelick, "The Energy Efficiency of IRAM Architectures," in *International Symposium on Computer Architecture*, pp. 327–337, June 1997.

[24] B. Mochocki, K. Lahiri, and S. Cadambi, "Power Analysis of Mobile 3D Graphics," in *Proceedings of the conference on Design, automation and test in Europe*, pp. 502–507, 2006.

[25] S. Collange, D. Defour, and A. Tisserand, "Power Consumption of GPUs from a Software Perspective," in *Proceedings of the 9th International Conference on Computational Science: Part I*, pp. 914–923, 2009.

[26] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, pp. 40–53, Mar. 2008.

[27] X. Ma, Z. Deng, M. Dong, and L. Zhong, "Characterizing the Performance and Power Consumption of 3D Mobile Games," *Computer*, vol. 46, no. 4, pp. 76–82, 2013.

[28] J. W. Sheaffer, D. Luebke, and K. Skadron, "A Flexible Simulation Framework for Graphics Architectures," in *Graphics Hardware*, pp. 85–94, 2004.

[29] J. W. Sheaffer, K. Skadron, and D. P. Luebke, "Studying Thermal Management for Graphics-Processor Architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 54–65, 2005.

[30] J. Pool, A. Lastra, and M. Singh, "An Energy Model for Graphics Processing Units," in *IEEE International Conference on Computer Design*, pp. 409–416, 2010.

[31] M. W. Halle, *Multiple Viewpoint Rendering for Three-Dimensional Displays*. PhD thesis, MIT, 1997.

[32] M. W. Halle, "Holographic Stereograms as Discrete Imaging Systems," in *Practical Holography VIII (Proceedings of SPIE)*, vol. 2176, pp. 73–84, 1994.

[33] B. Javidi and F. Okano, *Three-Dimensional Television, Video, and Display Technologies*. Springer-Verlag, 2002.

[34] P. Haeberli and K. Akeley, "The Accumulation Buffer: Hardware Support for High-quality Rendering," in *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pp. 309–318, 1990.

[35] Intel Corporation, *ATX Specification*, 2.2 ed.

[36] PCI-SIG, *PCI Express^{TM}x16 Graphics 150W-ATX Specification*, 1.0 ed., October 2004.

[37] A. Lauritzen, "Deferred Rendering for Current and Future Rendering Pipelines," in *Beyond Programmable Shading (SIGGRAPH course)*, 2010.

[38] T. Hachisuka, W. Jarosz, R. Weistroffer, G. H. K. Dale, M. Zwicker, and H. Jensen, "Multidimensional Adaptive Sampling and Reconstruction for Ray Tracing," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 33.1–33.10, 2008.

Paper I

Power Efficiency for Software Algorithms running on Graphics Processors

Björn Johnsson Per Ganestam Michael Doggett Tomas Akenine-Möller

Lund University

ABSTRACT

Power efficiency has become the most important consideration for many modern computing devices. In this paper, we examine power efficiency of a range of graphics algorithms on different GPUs. To measure power consumption, we have built a power measuring device that samples currents at a high frequency. Comparing power efficiency of different graphics algorithms is done by measuring power and performance of three different primary rendering algorithms and three different shadow algorithms. We measure these algorithms' power signatures on a mobile phone, on an integrated CPU and graphics processor, and on high-end discrete GPUs, and then compare power efficiency across both algorithms and GPUs. Our results show that power efficiency is not always proportional to rendering performance and that, for some algorithms, power efficiency varies across different platforms. We also show that for some algorithms, energy efficiency is similar on all platforms.

High Performance Graphics pages 67–75, 2012.

1 Introduction

All kinds of computing devices, be it CPUs, GPUs, or integrated CPUs with graphics processors, face great challenges in terms of power efficiency. Transistor technology scaling will no longer provide the performance improvements that we are used to [1]. One of the reasons for this is that when the supply voltage, and consequently the threshold voltage, of the transistor is reduced, the current leakage of the transistor increases exponentially [2]. This, in turn, means that the supply voltage cannot be reduced, and that future architectures will be limited by power instead of area.

It is well known that the power consumption of an external memory access, e.g., to DRAM, is substantially higher than both floating-point (more than an order of magnitude) and integer (more than three orders of magnitude) operations [3], and for CPUs, logic tends to use more power than caches [2]. In addition, moving data inside a chip is also becoming increasingly expensive, and starts to be a major part of power dissipation. Historically, we are used to the growth of memory bandwidth being slower than compute growth, but lately, the memory bandwidth growth has slowed down more [1]. Recently, Esmaeilzadeh et al. [4] have modeled multi-core speedup as a combination of single-core scaling, multi-core scaling, and device scaling, and predict that with a 22 nm technology process, 21% of the chip has to be powered off. When the technology scales down to 8 nm, more than half the chip has to be powered off. This under-utilization is called *dark silicon*. On top of this, current leakage also increases exponentially with the temperature of the chip [5]. All this indicates that the main optimization axis for the foreseeable future for any computing architecture is *power*.

It should be clear that predicting power consumption of a particular architecture is not an easy undertaking, and in fact, it may not always be meaningful, since power consumption also is a function of the program that runs on the architecture. However, optimizing for lower power consumption is very important, and there are opportunities for improving the power efficiency of future architectures by developing new hardware mechanisms to reduce power. This has been the focus of mobile graphics, which often has translated to algorithms for bandwidth savings [6]. For GPUs, simulators can be used to model power and leakage [7], and different low-level hardware optimization techniques can be developed and studied [8]. Power gating techniques can also be used [9]. It is also possible to save energy by reducing the precision in the computations in the vertex shader unit [10, 11] and in the pixel shader cores [12]. If you are not in a position to make power-efficient hardware changes, another approach to reduce power consumption remains, namely, to develop power-efficient *software*. Koduri [13] suggests that software developers should optimize for power as well, and in particular so for mobile devices. Some of the advice includes minimizing the frame rate and continuing to optimize the code of an application even if the frame rate goal has been reached.

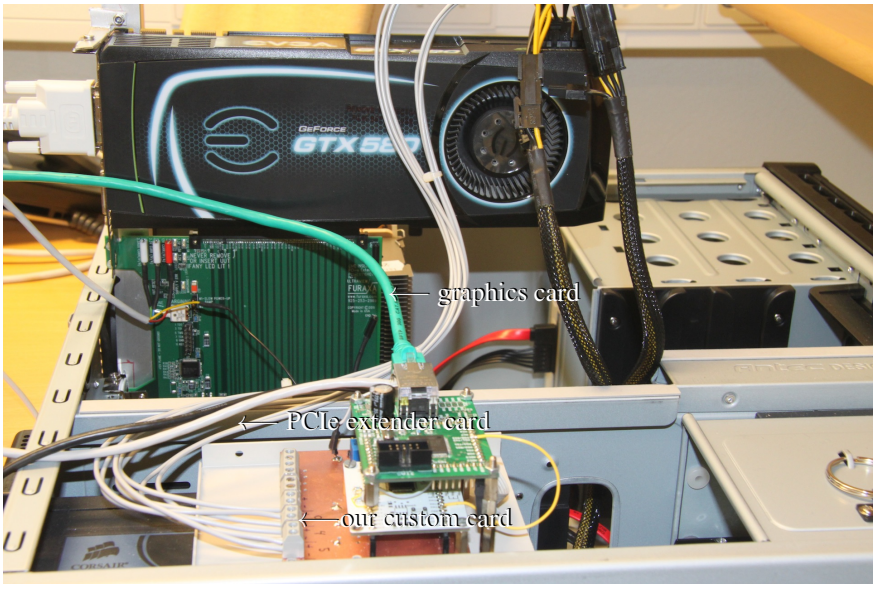


Figure 1: We have built a power measurement station, which measures power on the PCI express bus (which can deliver up to 75 W) and on the graphics card's two power connectors, which in this case can deliver up to 75+150 W. This sums to a max of 300 W.

In this paper, we take a different approach to study power efficiency of software running on graphics processors. We have built a power measurement station, as shown in Figure 1, which measures power consumption directly on the PCIe bus and on the power connectors of discrete graphics cards. For integrated CPU and graphics processors, we measure directly at the battery connection (for mobile phones) or directly on the power connectors of the motherboard. Using our power measurement station, we have studied the power efficiency of algorithms that generate exactly or approximately the same result, and we compare the power consumption per frame with the time needed to render the frame. For example, one of our case studies uses different shadow algorithms. In our study, power consumption is measured on two different discrete graphics cards, on a CPU with integrated graphics processor, and on a mobile phone, and these have widely different power efficiency characteristics. We hope that our research will spark an increased interest in the power efficiency of graphics software, and as such, that it opens a new research area for the graphics community.

2 Methodology

Our goal in this research is to measure power consumption on a number of different rendering algorithms running on several different types of graphics architectures.

This includes discrete graphics cards from different vendors, integrated graphics on the CPU die, and inside a mobile phone. Some graphics architectures have built-in counters to estimate some kind of power draw, but not all architectures expose those, and they tend to estimate different things anyway.

Instead, we take another approach, which allows for a fairer comparison (at least between different discrete graphics cards, or between different mobile phones, etc). The general idea is to measure power draw on all incoming power sources. It suffices to measure the current of the power sources, since the voltages are constant, and due to the following relationship between power, P, voltage, U, and current, I:

$$P = UI. \tag{1}$$

The units are watts (W or joules/second) for power, volts (V) for voltage, and ampere (A) for current. Note that energy is the integral of P over time. In addition, the dissipation power due to switching in CMOS is $P = CU^2 f$, where C is the capacitance, and f is the clock frequency. For discrete graphics cards, there are several sources of power, namely, the PCI express bus, which can deliver up to 75 W, and between 0 and 2 power connectors, where connectors with 6 pins can deliver up to 75 W, and 8-pin connectors can deliver up to 150 W.

To measure all currents on these power sources, we have built a custom power measurement station, as shown in Figure 1. The currents from the PCI express bus is measured using an Ultraview PCIeEXT-16HOT expander card, which has test points for measuring the currents. On our custom card, we have four ACS710 Hall effect current sensors, which can measure currents up to 12 A. There are also two shunt current sensors that can accurately measure smaller currents of up to 1 A, which are useful for mobile phone measurements. In Figure 2, we show the setup when measuring power on a mobile phone.

All these currents are going to an A/D converter, and these are fed to an ARM Cortex-M3 processor that samples the currents at 40 kHz. The resulting sampled signals are sent via Ethernet to a PC, which can show the currents in real time directly in a window, or save them to a file for later analysis (e.g., conversion to power and filtering).

The power measurement station as described above can be used directly to measure the power consumption of discrete graphics cards. For mobile phones and for CPUs with an integrated graphics processor, this approach cannot be used directly since the graphics processor is not an isolated unit. For mobile phones, we decided to measure *all* (including, for example, the display) power consumption by measuring power draw at the battery connector inside the phone. This is not perfect, but it is hard to measure more accurately than that, and at least, no power consumption is missed with this methodology. Similarly for CPUs with integrated graphics processors, we measure the power consumption on the 4-pin power connector, which supplies +12 V to the mother board. For both these architectures, we subtract the power consumption in some form of "idle" state in order to isolate the power consumption of the graphics processor. We define the idle power as mea-

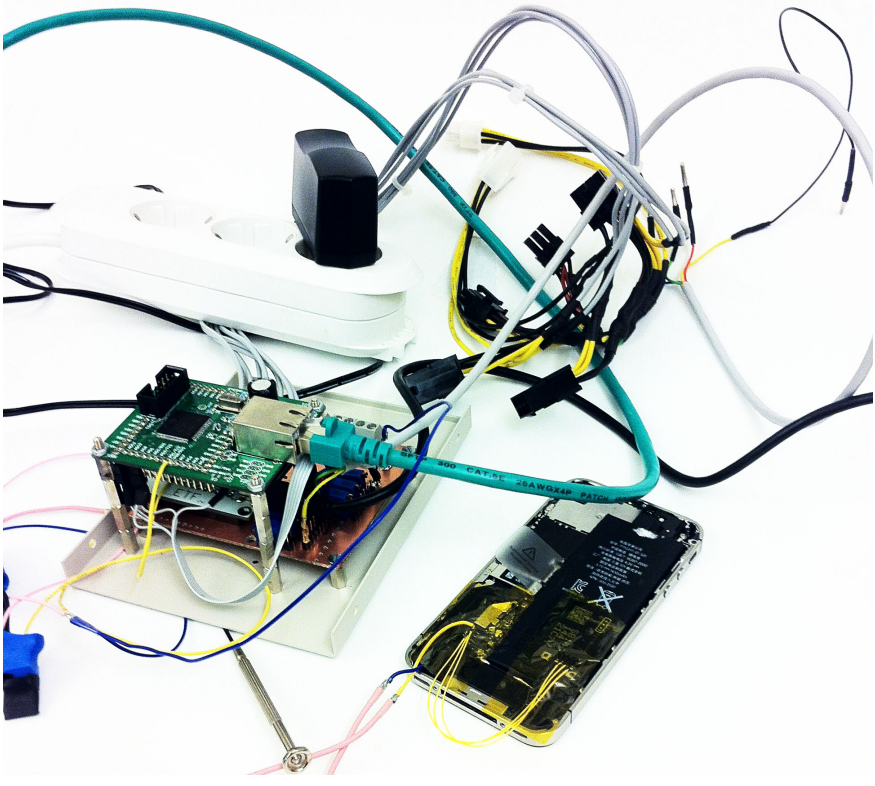


Figure 2: Our power measurement station, also seen in Figure 1, connected directly on the battery power connects on an iPhone 4S.

sured power draw of our application without submitting any OpenGL API calls at all[1]. These differences in measuring methodology affects the comparability of the results across platforms. On the integrated platform and mobile phone, the idle memory power usage is removed, but the power for graphics usage of memory is included. For this reason, we compare the relative consumption of different algorithms on different platforms, but in general, we attempt to not draw too detailed conclusions from comparing power consumption of discrete graphics cards and mobile phones or CPUs with integrated graphics.

The questions that we were interested in answering when starting this project include:

1. What are the power characteristics of different graphics algorithms solving the same problem on different graphics architectures?

2. Is energy directly proportional to frame time?

[1]The CPU power cost of the OpenGL calls *is* included in the CPU with integrated graphics and mobile phone measurements.

3. What does the power consumption look like during an animation?

4. What does the power consumption look like inside a frame (for different algorithms)?

5. Can power optimization of software algorithms become a new subtopic in graphics?

In the following, we will attempt to answer these questions.

3 Case Studies

We have chosen two case studies to measure power consumption on. Both of these are commonly used in real-time rendering today. The first is simply rendering the scene from the eye, and the second is shadow rendering. Those are described in the subsequent subsections. All our algorithms have been written in OpenGL and some have been ported to OpenGL ES, since we want to test them on mobile phones as well.

3.1 Case 1: Primary Rendering

It is likely that graphics hardware's most common use case is to render a scene from the eye, i.e., to evaluate both primary visibility and shading. We have three different flavors of this case study, where lighting with 32 spotlights (without shadows) is included. The first is basic forward rendering (FR), where the triangles simply are submitted as vertex arrays, and lighting computed for non-culled fragments. Our second technique starts with rendering the scene only to the depth buffer, which is followed by a pass with the depth test set to `GL_LEQUAL` and lighting computed for each fragment with a loop over the light sources. This way of priming the depth buffer avoids expensive pixel shading for fragments that will not be visible in the final image. We call this method Z-prepass rendering (ZR). Finally, we use deferred rendering (DR), which starts by creating various G-buffers [14], e.g., one buffer for depth, one for the normal in world space, one for specular exponent, and one for the diffuse texture. Then, for each light source, we render a volume covering the region of influence of the spotlight, and accumulate the lighting to each affected pixel.

For all primary rendering algorithms, we use the same camera path through the Sponza atrium with five Stanford dragons added. The Sponza atrium contains $224,337$ triangles and the dragons contain $100,000$ triangles each. Some frames from this animation can be seen in the middle left part of Figure 3.

3.2 Case 2: Shadow Algorithms

As our second case study, we have chosen three different shadow algorithms, namely, shadow volumes (SV) [15], shadow mapping (SM) [16], and variance shadow mapping (VSM) [17]. While the algorithms for case 1 (Section 3.1) all generate exactly the same result, our chosen shadow algorithms only generate approximately the same result. For example, the shadow volume algorithm generates pixel-exact shadows without anti-aliasing, while the quality of both shadow mapping techniques depends on the shadow map resolution. In addition, variance shadow mapping provides filtered shadow lookups, and so has smoother edges. We chose those three shadow algorithms because they are rather different, and our hypothesis was that they may have different power consumption characteristics.

The shadow volume algorithm extracts a shadow volume from each shadow caster by determining which of its edges are silhouette edges as seen from the light source. These edges are then extruded away from the light source, creating the sides of the shadow volume as quads. This shadow volume is then capped at each end. These shadow primitives are rasterized from the eye, and front facing quads increment the stencil buffer, while back facing quads decrement the stencil buffer. We have used Carmack's reverse (also called z-fail) [18], where the increment/decrement is done for occluded shadow quads. Since a shadow quad often can cover many pixels, the shadow volume algorithm is known to burn fill rate.

The shadow map and variance shadow map algorithms need to render a shadow map — containing depths to the closest surfaces as seen from the light source — of the shadow casting geometry in a first pass. The variance shadow map algorithm then creates two mipmap hierarchies containing filtered depths, and filtered squared depths. Using Chebyshev's inequality and those two mipmap hierarchies, the shadow test provides a floating-point value, instead of a binary outcome. Normal shadow maps do not use a mipmap hierarchy, and so gain some speed there. However, when the filter is large, variance shadow mapping will go up towards the tip of the mipmap hierarchy. For mipmap-based algorithms [19], this is known to increase cache hit ratio as compared to not using a mipmap. As a result, the memory bandwidth usage to main memory is reduced. So even though variance shadow maps create a mipmap hierarchy, it is not clear that it will be more expensive in terms of power consumption.

For the shadow algorithms, we use a camera path over the scene with tessellated geometry without textures. The scene contains $396,344$ polygons. Some frames from this animation can be seen in the middle right part of Figure 3.

3.3 OpenGL ES

For the mobile phone, we use OpenGL ES, and there we have chosen to omit deferred rendering (DR), shadow volumes (SV), and variance shadow mapping (VSM). Deferred rendering was omitted since our target mobile platform does not support multiple render targets. A multi-pass solution for creating the G-buffers

would be possible, but would not result in a fair comparison. Likewise, variance shadow maps were omitted, since it was not possible to implement without adding an extra pass. Shadow volumes were omitted because the mobile phone could not support the amount of geometry of our test scene without drastically splitting up geometry into more draw calls, which would incur a significant overhead.

4 Results

Using our two case studies and our power measurement station, we have measured power on a series of GPUs. Starting with high-end discrete GPUs, we have taken measurements on an AMD Radeon HD7970 and on an NVIDIA GeForce GTX 580. For integrated GPUs, we measure power on an Intel Sandy Bridge Core i7 2700K with Intel HD 3000 graphics. The graphics processor part of Sandy Bridge is running at between 850 MHz and 1350 MHz because we have turbo mode enabled. In addition, we have set idle turbo mode to "high performance." For mobile GPUs, we measure power on an iPhone 4S, which contains an Apple A5 chip with a dual core PowerVR SGX543MP2 GPU running at 250 MHz. In all main diagrams, we show the raw data, which often is a bit noisy, in a lighter color, while we show a low-pass filtered version with a fatter curve using a stronger color.

In Figure 3, we show power consumption diagrams for both the GeForce 580 and the Radeon 7970 for our primary rendering application and for our shadow algorithms. These animations were rendered at 2560×1440 pixels.

It should be noted that the GeForce was manufactured in 40 nm, while the Radeon was manufactured in 28 nm, which gives a power advantage for the Radeon.[2] This advantage is one of the possible reasons that Radeon power varies between 170–245 W, while GeForce power varies between 240–310 W. For the shadow algorithm runs, the number of lights is varied from 2 to 8 lights in increments of 2, and the resolution of the shadow maps was set to 2560^2 pixels. As can be seen, the Radeon 7970 has spikes where the frame time increases and, as a result, the average frame power decreases at the same time, as a longer period of idle power is taken into account. Looking at the frames in which this occurs, we see that the power usage does not really change compared to neighboring frames, but the time stamps for start and end of a frame are further apart as shown in Figure 4. At this point, we do not know the root cause of this, but since the frame time increases, and those measurements are independent of our power measurement station, we are certain that it is not a shortcoming of our custom card. For primary rendering, FR is more expensive in terms of power on both platforms. On the GeForce 580, DR generally uses the least power, and has the best performance. On the Radeon 7970, ZR has higher frame times than FR and DR at times, while FR and DR have

[2] In all fairness, it should be noted that NVIDIA recently released the Kepler architecture [20], which also is manufactured in 28 nm, and has been optimized for power (e.g., tripling the number of shader cores while lowering the shader core clock frequency). However, at the time of writing no such cards were available to us.

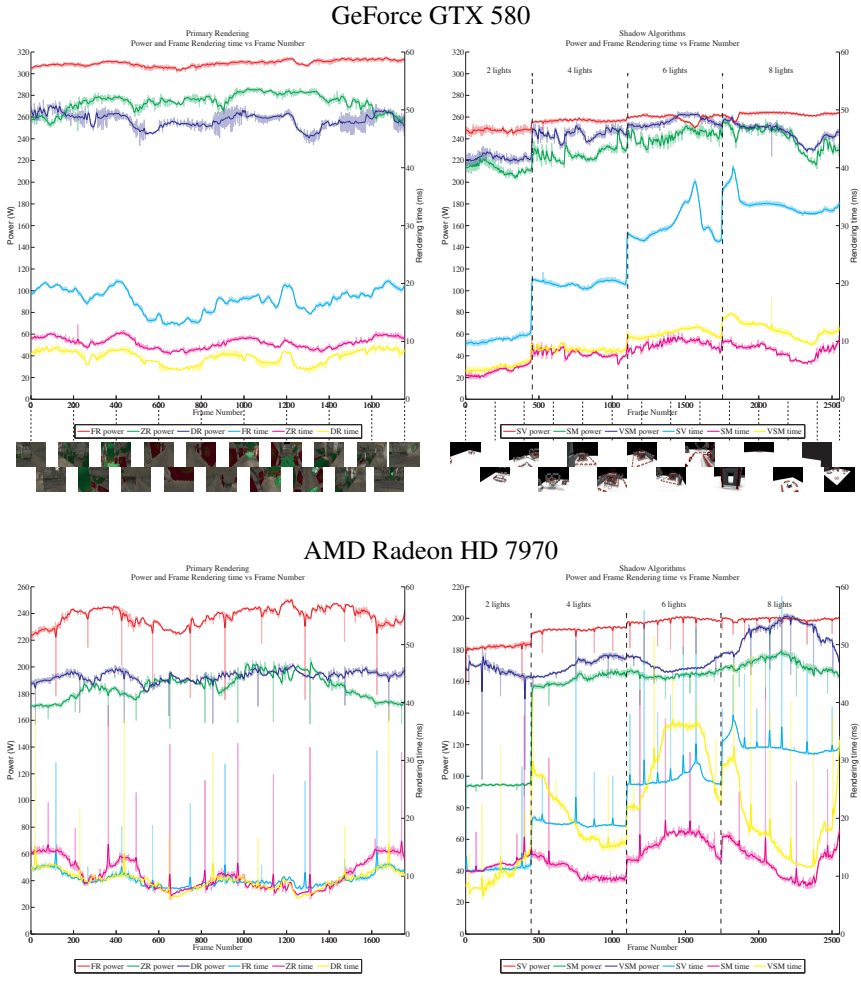


Figure 3: Frame times and power consumption for primary rendering (left) and the different shadow algorithms (right) on an NVIDIA GTX580 (top) and on an AMD Radeon HD 7970 (bottom). Abbreviations: FR (forward rendering), ZR (Z-prepass), DR (deferred), SV (shadow volumes), SM (shadow mapping), and VSM (variance shadow mapping).

about the same. For power, FR clearly uses the most power, while ZR and DR are more similar, but ZR often uses a little less power than DR. So even though FR and DR are generally faster, ZR uses less power. From these measurements, it is clear that one cannot just measure frame times in order to find the most energy-efficient algorithm (since FR and DR have about the same frame times, but widely different power usage, for example).

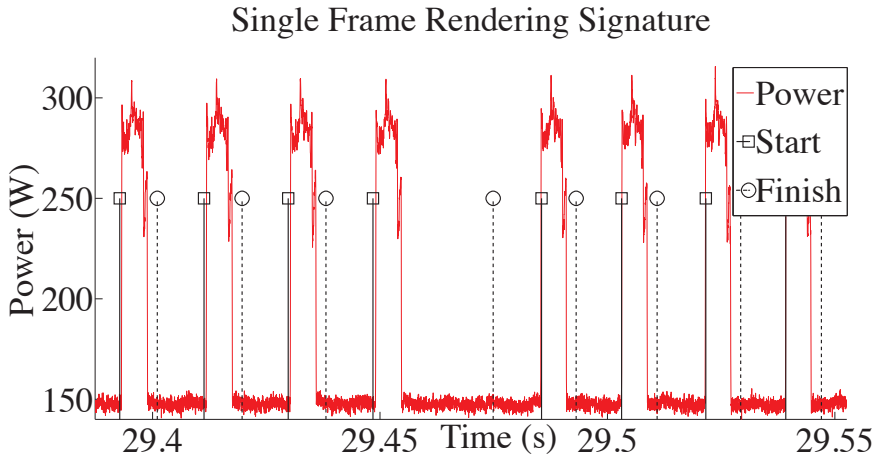


Figure 4: Power signature of eight frames of forward rendering on the AMD Radeon 7970, where the fourth frame is followed by a delay which also affects our time stamp measurements.

For the shadow rendering results, the order of power usage is SV (highest), VSM, and then SM (lowest) for both discrete GPUs. Increasing the number of lights has little impact on power, except when going from 2 lights to 4 lights. In particular, on the Radeon 7970 when going from 2 to 4 lights, there is a large increase in SM power, but little change in SM frame times, and in fact, frame time goes down. The Radeon 7970 has two power states, where the first runs at 300 MHz and the second at 925 MHz. It would appear that SM with 2 lights runs at the 300 MHz state and then switches to the 925 MHz state for 4 lights. We cannot determine this exactly because we do not have accurate frequency measurements. Shadow performance on the GeForce 580 is clearly separated with SM being fastest, followed by VSM and then SV. Power consumption is clearly lowest for SM. On the Radeon 7970, SM is again the fastest, but SV and VSM vary over the animation with VSM varying a lot and SV staying fairly steady.

The power measurements for the Sandy Bridge, which is manufactured in 32 nm, are shown in Figure 5, where the animation was rendered at 1600×1200 to reach real-time frame rates. The shadow maps for the Sandy Bridge measurements were scaled with a factor taking into account the difference in screen resolution, i.e., scaled by $k = 1600 \times 1200/(2560 \times 1440)$ compared to the discrete GPUs. This means that we used 1792^2 as resolution for the shadow maps, and this is to make the energy/pixel measurements in Table 1 fairer. As can be seen, the power consumption for graphics varies between 8–22 W, and it contains a regular oscillation, which we presume is the effect of the turbo mode dynamically scaling voltage and frequency. This oscillation originates in the idle power measurement, where the pattern is inversed. In general, we observed an idle power draw of about 40 W.

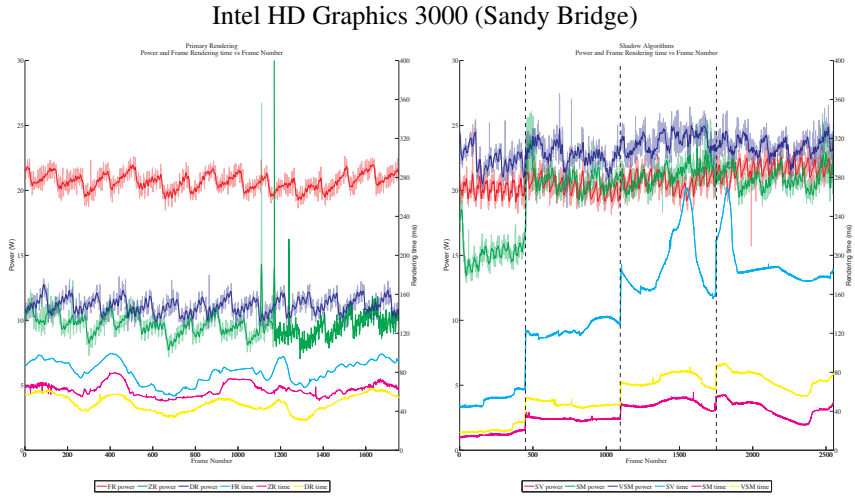


Figure 5: Frame times and power consumption for primary rendering (left) and the different shadow algorithms (right) on an Intel Sandy Bridge integrated graphics HD3000.

The power results show that FR draws the most power followed by DR and ZR. The order of DR and ZR is the opposite compared to the GeForce 580, and yet the performance results show that DR is the fastest, then ZR and FR, which is the same order as the GeForce 580. This shows that performance is *not* always a good indicator of power and that it varies across platforms. We also note that both ZR and DR uses about 50% of the power of FR, and since ZR and DR also are faster, this turns into a significant difference in energy efficiency as we will see later (Table 1). For shadows, the power draw is highest for VSM, followed by SV. SM uses less power for only 2 lights, but then generally matches SV. This is quite different compared to the discrete cards in that SV and VSM have changed places, and we note that SV on Sandy Bridge uses consistently less power than VSM. Shadow performance on Sandy Bridge has similar curves for each algorithm as the GeForce 580, with SM being the fastest, followed by VSM, and then SV.

In Figure 6, we show the power measurements for the iPhone 4S, where the animation was rendered at 960×640. The shadow map resolutions were scaled in a similar manner as done for Sandy Bridge, resulting in a resolution of 1024^2 for this architecture. In general, the power consumption for graphics varies between 0.7–1.1 W. The power results show that ZR uses more power than FR to achieve lower frame times for ZR compared to FR. Again, we observe that frame time does not correlate to power usage, and it is only through power measurement analysis that the lower power algorithm can be determined. It is interesting to note that the iPhone uses a sort-middle architecture with deferred rendering [21], which essentially performs a pre-Z pass in hardware before shading, and yet our ZR, which

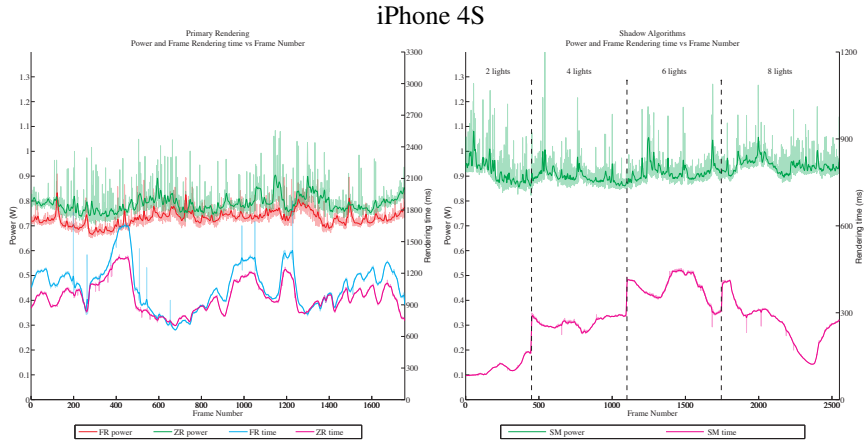


Figure 6: Frame times and power consumption for primary rendering (left) and the different shadow algorithms (right) on an iPhone 4S. Note that the frame time axis is different in the two graphs.

adds another pre-Z pass, still improves performance. SM runs at a much lower frame time than primary rendering due to the shadow scene having less geometry, but SM still uses more power for each frame.

	Average energy/pixel (nJ) [std. dev]					
	Primary rendering			Shadow algorithms		
	FR	**ZR**	**DR**	**SV**	**SM**	**VSM**
GeForce 580	1443 [180]	722.1 [62.1]	510.6 [87.1]	1325 [114]	446.6 [80.3]	532.0 [67.5]
Radeon 7970	607.4 [73.0]	512.0 [103]	489.2 [79.9]	953.9 [44.9]	469.0 [88.3]	804.0 [250]
Sandy Bridge	871.5 [134]	314.2 [46.8]	280.0 [53.4]	1317 [212]	311.3 [76.2]	511.3 [87.9]
iPhone 4S	2234 [423]	2015 [290]	– – –	– – –	460.5 [135]	– – –

Table 1: Average energy per pixel measurements for all our architectures and algorithms. To measure standard deviation in a meaningful way, we have kept the number of light sources constant at four for the shadow algorithms. Note that Sandy Bridge and iPhone energy measurements have excluded idle memory power usage, but included the driver overhead.

It is also interesting to compute how much energy is used over an entire animation divided by the number of frames in the animation and the screen resolution. This is computed as shown below:

$$E = \frac{\int_0^{t_{\text{tot}}} P(t)\mathrm{d}t}{F \cdot R}, \tag{2}$$

where t_{tot} is the total time it took to render the entire animation, F is the number

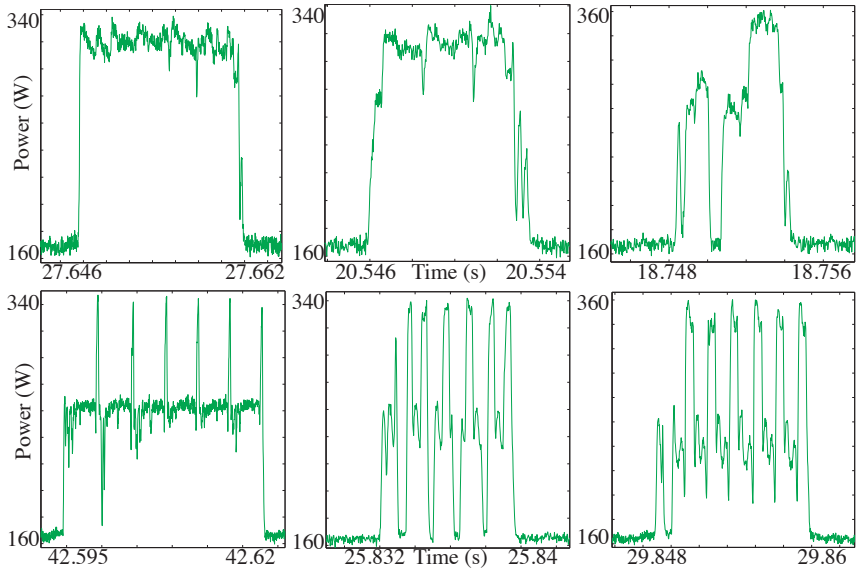


Figure 7: Single frame rendering power signature for the NVIDIA GeForce GTX580. The top row contains measurements for (from left to right) FR, ZR, and DR, while the bottom row shows SV, SM, and VSM. All shadow algorithms use 6 light sources.

of frames in the animation, and R is the screen resolution in pixels. By dividing with screen resolution, we weigh in that different resolutions are used for different platforms. Two advantages of this measure are that GPUs that are faster to render the animation will integrate over a shorter time domain (t_{tot}), which should be taken into account, and that it is a screen resolution independent measure. We believe this makes the comparison fairer. In Table 1, we have gathered statistics for the average energy per pixel and its standard deviation. We note that in some situations, performance/Watt is reported. This could be interpreted as frames per second per Watt, which is frames per joule, and due to the resolution differences, we would report pixels/joule. While all the information is available in the table, we have chosen joules per pixel for the same reasons that frames per second often is avoided, and pure time per frame is preferred. One of these reasons is that one cannot split the running time of an algorithm into different parts and measure them using frames per second, while on the other hand, it makes sense to measure the time of a certain part of an algorithm. For power efficiency, we foresee a future where it may be possible to measure the power consumption for a certain part of an algorithm, and therefore, joules/pixel is chosen in our presentation.

Table 1 shows that energy/pixel follows similar trends across all GPUs. For primary rendering, it is noticeable that the differences between algorithms are much greater for the GeForce 580 and the Sandy Bridge than the Radeon 7970. For the

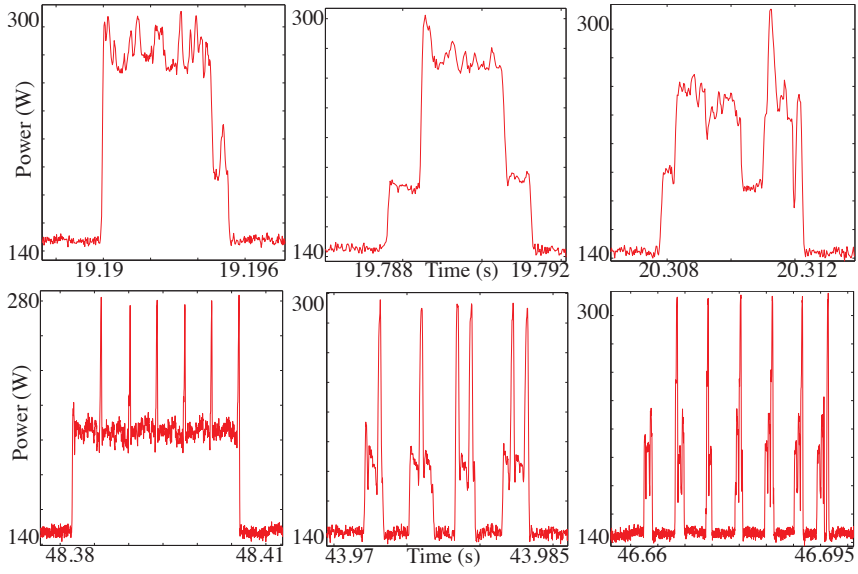


Figure 8: Single frame rendering power signature for the AMD Radeon HD7970. The top row contains measurements for (from left to right) FR, ZR, and DR, while the bottom row shows SV, SM, and VSM. All shadow algorithms use 6 light sources.

shadow algorithms, we observe that SM, which uses a lighter rendering load, has similar energy/pixel over all four GPUs. However, it is also interesting to note that there is about an order of magnitude in difference in frame times (discrete GPUs are fastest), while at the same time there is more than an order of magnitude in difference in the number of transistors used for graphics (discrete GPUs use the most transistors). Also VSM, compared to SM, requires a small energy/pixel increase on the GeForce 580, but requires a more significant increase on the Radeon 7970 and on the Sandy Bridge. While SV, compared to VSM, has a large energy/pixel increase on the GeForce 580 and on the Sandy Bridge, but requires a small increase on the Radeon 7970.

Our power measuring station samples at a high frequency, so we can look at the characteristics of individual frame power usage. Figure 7 shows frames for the GeForce 580 and Figure 8 shows frames for the Radeon 7970. For the primary rendering algorithms, FR shows full power usage throughout the frame, while ZR has some drops in power usage. DR shows a drop to idle power in the middle of the frame before finishing with full power usage. The shadow rendering algorithm frames have 6 lights and the processing for the 6 lights can be clearly seen in each graph. It is interesting to note that in SM, both discrete cards drop to idle power

between some lights, but for VSM only Radeon 7970 drops fully to idle, and does that between each light. Also, the amount of idle-time is larger for Radeon 7970.

5 Conclusions and Future Work

Power is a major concern for all graphics processors today, and will be even more important in the future when technology continues to scale down. In this work, we have built a power measurement station, and measured power and frame times for a set of different GPUs and graphics algorithms. As we have shown, the fastest algorithm is not always the least power hungry algorithm, and we have also shown that this varies greatly between different architectures. More importantly, we believe that power is so incredibly important that it will become an integral part of most graphics research papers in the near future. We speculate that it will become as common to report joules per pixel as it is to report milliseconds per frame today.

At this point, we have not provided any new and more energy-efficient algorithms. So, for future work, we want to focus on studying more algorithms, and to explore optimizations for existing algorithms that reduce power consumption, or even invent new algorithms with better power behavior. It would also be useful to put together a graphics benchmark for measuring power consumption and frame times. We hope that our work has opened up a new small subfield for graphics performance optimization.

Bibliography

[1] 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.

[2] S. Borkar and A. A. Chien, "The Future of Microprocessors," *Communications of the ACM,*, vol. 54, no. 5, pp. 67–77, 2011.

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

[4] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *38th International Symposium on Computer Architecture*, pp. 365–376, 2011.

[5] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects," Tech. Rep. CS-2003-05, University of Virginia, March 2003.

[6] 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.

[7] J. W. Sheaffer, D. Luebke, and K. Skadron, "A Flexible Simulation Framework for Graphics Architectures," in *Graphics Hardware*, pp. 85–94, 2004.

[8] J. W. Sheaffer, K. Skadron, and D. P. Luebke, "Studying Thermal Management for Graphics-Processor Architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 54–65, 2005.

[9] P.-H. Wang, C.-L. Yang, Y.-M. Chen, and Y.-J. Cheng, "Power Gating Strategies on GPUs," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 3, pp. 13:1–13:25, 2011.

[10] J. Pool, A. Lastra, and M. Singh, "Energy-Precision Tradeoffs in Mobile Graphics Processing Units," in *International Conference onf Computer Design*, pp. 60–67, 2008.

[11] J. Pool, A. Lastra, and M. Singh, "Power-Gated Arithmetic Circuits for Energy-Precision Tradeoffs in Mobile Graphics Processing Units," *Journal of Low Power Electronics,*, vol. 7, no. 2, pp. 148–162, 2011.

[12] J. Pool, A. Lastra, and M. Singh, "Precision Selection for Energy-Efficient Pixel Shaders," in *High Performance Graphics*, pp. 159–168, 2011.

[13] R. Koduri, ""Power" of Realtime 3D Rendering," in *Beyond Programmable Shading (SIGGRAPH course)*, 2011.

[14] T. Saito and T. Takahashi, "Comprehensible Rendering of 3-D Shapes," in *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, pp. 197–206, 1990.

[15] F. Crow, "Shadow Algorithms for Computer Graphics," in *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, pp. 242–248, July 1977.

[16] L. Williams, "Casting Curved Shadows on Curved Surfaces," in *Computer Graphics (Proceedings of ACM SIGGRAPH 78)*, pp. 270–274, ACM Press, 1978.

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

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

[19] L. Williams, "Pyramidal Parametrics," in *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, pp. 1–11, ACM, July 1983.

[20] NVIDIA, "GeForce GTX 680," tech. rep., 2012.

[21] Imagination Technologies Ltd, *POWERVR Series5 Graphics SGX architecture guide for developers*, 2011.

Paper II

Performance per What? Energy-Efficiency Units for Graphics

Tomas Akenine-Möller Björn Johnsson

Lund University

ABSTRACT

In this short note, we argue that *performance per watt*, which is often used in the graphics industry and on web forums, is *not* a particularly useful unit for power efficiency for similar reasons that frames per second is not a good metric when presenting research about the performance of rendering algorithms. For rendering performance, milliseconds is preferred, and for power and energy efficiency, we argue that watts and joules are the more reasonable performance metrics. We also show an example where the inverse measure, i.e., performance per watt, is non-intuitive, while reporting in nanojoules per pixel is much more intuitive and easier to reason about.

Journal of Computer Graphics Techniques 1(1):37–41, 2012.

1 Introduction

Power and energy efficiency is one of the most important optimization axes for graphics [1], and in particular so for mobile phones [2], tablets, and laptops, but also so for desktop graphics [3]. Predicting or estimating power for a certain graphics algorithm running on a particular graphics processor is extremely difficult. The reasons for this are many, including, for example, the fact that current leakage increases exponentially with the temperature of the chip [4], and that moving data inside the chip makes up for a significant part of power consumption. Therefore, it makes sense to *measure* the current(s) of the graphics processor, which can be translated to power consumption, since voltage to a discrete GPU or to the motherboard of an integrated CPU/GPU is constant [1].

We believe that reporting energy and/or power consumption in future graphics research will become as common as it is to report performance (milliseconds per frame) today. Most of the graphics industry (at least the marketing departments) is reporting power efficiency in "performance per watt", which we argue is a less useful unit, and in some contexts, it is quite useless. Public energy efficiency research in graphics is in its infancy today, and therefore, it is important to settle on which units to report results in. This is the topic of this note.

2 Unit Motivation

As mentioned above, the quantity "performance per watt" is often used, with the objective to reach an as high number as possible. Hence, we can conclude that performance can be given in frames per second, for example, denoted F/t, where F is the number of frames rendered in t seconds. It is well-known that $P = E/t$, where P is power measured in watts (W), E is energy measured in joules (J), and t is time measured in seconds (s). This means that performance per watt becomes:

$$\frac{F/t}{P} = \frac{F \cdot t}{t \cdot E} = \frac{F}{E}, \tag{1}$$

which is *frames per joule*, which actually is an energy measure, and not a power measure, as "performance per watt" implies.

Frames per joule (fpJ) is similar to *frames per second* (fps) in that it is an inverse measure, and very few researchers are reporting performance using only frames per second today. In fact, the "JCGT Formatting Guidelines and Template" recommends that performance should primarily be reported in time units, i.e., in seconds. Also, it makes much more sense if you develop a graphics algorithm responsible for part of a frame, e.g., a screen-space ambient occlusion algorithm (SSAO) and report in milliseconds rather than in fps. Imagine that the SSAO algorithm takes 5 ms to execute. This is useful information to, e.g, game developers, because they can immediately know how that will affect their game engine. However, saying that the SSAO runs at 200 fps is less intuitive when it comes to what effect it

would have on one's game engine. It should be acknowledged that it sounds better to report in these inverse measures. For example, getting a $3\times$ performance improvement in fps rather than reducing the frame time by 67% may sound better. This explains the popularity from the marketing departments.

It should also be noted that these inverse measures (fps and fpJ) cannot be averaged. So if fps/fpJ is measured over an animation, it is incorrect to average all the frames' fps/fpJ in order to compute an average frame per second or frames per joule. Instead, the numbers should be inverted (1/fps and 1/fpJ), and then these numbers are averaged. Finally, one may convert back to fps/fpJ.

For similar reasons that frame time is more useful when reporting performance, frame energy (joules per frame) is more useful when it comes to reporting energy efficiency. So, one could measure the energy it takes to run an entire animation, and divide by the number of frames. If the same animation is rendered on another platform or device at a different resolution, then the resulting measures are not comparable due to the difference in resolutions. To that end, we have recently proposed to normalize by screen resolution [1], which results in the following equation:

$$E = \frac{\int_0^{t_{\text{tot}}} P(t)\mathrm{d}t}{F \cdot R}, \tag{2}$$

where t_{tot} is the total time it takes to render the entire animation, F is the number of frames in the animation, and R is the screen resolution in pixels. The unit is joules per pixel (Jpp), or nanojoules per pixel (nJpp), to make the decimals more manageable.

Hypothetically, one can creative an "efficient" architecture that uses very little power, but takes a very long time to render the frame. Such an architecture would be energy-efficient, but not very usable. Therefore, we argue that both frame time and energy per pixel (or energy per frame) is reported at the same time. As a complement, one can also report energy per frame divided by time per frame, which gives use a strict power measure ($P = E/t$).

In the following, we will give a practical example that clearly shows that performance per watt (frames per joule) is to avoid.

3 Example

In this section, we will show an example, where we have measured power on an Intel Ivy Bridge with an Intel HD Graphics 4000 integrated GPU. We use the same methodology as proposed by Johnsson et al. [1]. The algorithm we measure on is a deferred shading renderer consisting mainly of two passes. The first pass creates the G-buffers (depth, position, diffuse texture, etc), while the second loops over light sources in order to accumulate lighting. Similar to Johnsson et al. [1], we remove the idle power of the CPU by measuring an execution of the program

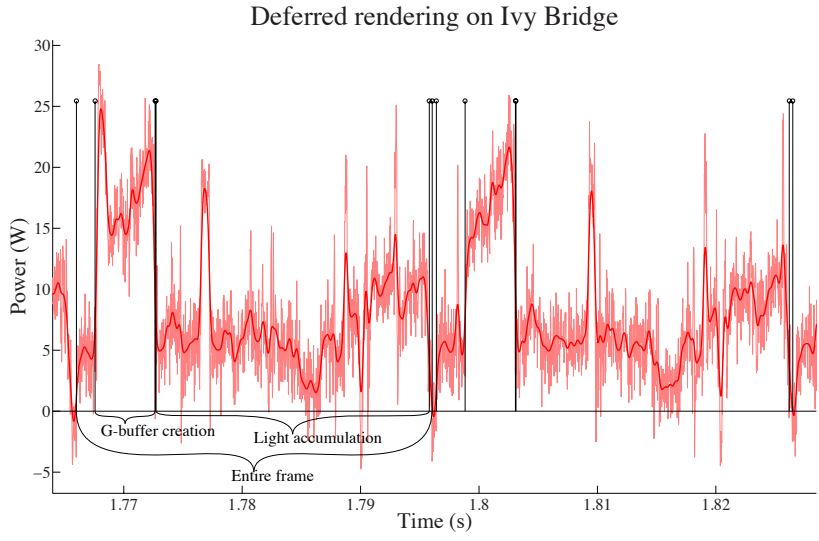


Figure 1: Measured power for deferred shading on an Ivy Bridge (using a 22 nm process technology). The entire frame on the left uses 135 nJ/pixel. The G-buffer creation pass uses 47 nJ/pixel and the light accumulation pass uses 84 nJ/pixel. Note that the removal of the CPU power pushes the curves below zero at times. As described earlier, this is due to that the idle power that is removed is an average over several entire frames, which reduces noise in the graphs and still gives correct average energy per pixel.

with all calls to the OpenGL API removed. However, instead of doing this instantaneously 40,000[1] per second, we compute an average power over one or more frames and remove that. This removes noise in the idle computation, but, as we will see, it may push the power curve below zero in some cases. Note that with this method of removing idle power, we are still including the graphics driver in the final numbers.

The power curves over time for two frames are shown in Figure 1. We have estimated that the energy per pixel is 135 nJ/pixel including both rendering passes, and any overhead in between. Moreover, we inserted time stamps into the rendering, which made it possible to isolate the G-buffer pass and the lighting accumulation passes. The G-buffer pass uses 47 nJ/pixel, while the lighting accumulation pass uses 84 nJ/pixel. The rest of the rendering (overhead) uses $135 - 47 - 84 = 4$ nJ/pixel.

Now, let us convert these numbers to their inverses (performance per watt), i.e., the entire frame uses $1/(135 \cdot 10^{-9}) = 7.4$ Mpixels/joule. Similarly, the G-buffer pass converts to 21.3 Mpixels/joule, and the lighting accumulation pass to 11.9 Mpixels/joule. It becomes very non-intuitive, when it comes to reasoning about the

[1] Our power measurement station operates at 40 kHz

overhead energy. Just looking at the numbers, 7.4, 21.3, and 11.9 Mpixels/joule, does not easily convey the energy overhead of the rendering. Another approach would be to convert the overhead, i.e., 4 nJ/pixel, to its inverse, which becomes 250 Mpixels/joule, but this clearly does not help either in understanding what the overhead is.

We conclude that it is clear that performance per watt (fps per watt, frames per joule, pixels per joule, etc) should be avoided in both academic and industry research about energy/power efficiency.

Bibliography

[1] B. Johnsson, P. Ganestam, M. Doggett, and T. Akenine-Möller, "Power Efficiency for Software Algorithms running on Graphics Processors," in *High Performance Graphics*, pp. 67–75, 2012.

[2] 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.

[3] NVIDIA, "GeForce GTX 680," tech. rep., 2012.

[4] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects," Tech. Rep. CS-2003-05, University of Virginia, March 2003.

Paper III

Per-frame Energy Measurement Methodology for Computer Graphics

Björn Johnsson Tomas Akenine-Möller

Lund University

ABSTRACT

Energy and power efficiency are becoming important topics within the graphics community. In this paper, we present a simple, straightforward method for measuring per-frame energy consumption of real-time graphics workloads. The method is non-invasive, meaning that source code is not needed, which makes it possible to measure on a much wider range of applications. We also discuss certain behaviors of the measured platforms that can affect energy measurements, e.g., what happens when calling `glFinish()`, which ensures that all issued graphics commands are finished executing. Measurements are done both on a smartphone and on CPUs with integrated graphics processors.

Journal of Computer Graphics Techniques 3(1):60–73, 2014.

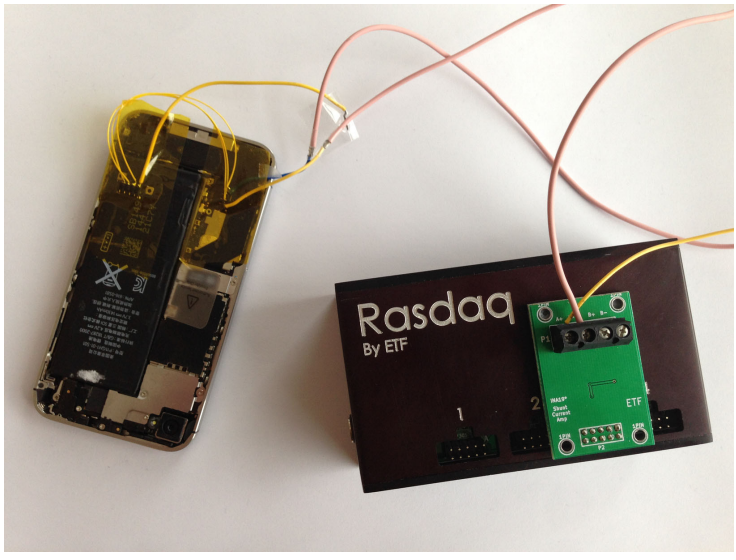


Figure 1: Our power measurement station, connected to an iPhone 4S.

1 Introduction

The topic of both power and energy efficiency is increasingly important. It is recognized as a crucial design point for several reasons. Not only is the number of computing devices powered by batteries increasing, but heat dissipation, and thus power, is becoming one of the major design constraints for computer chips [1, 2].

Much work has been devoted to designing energy-efficient hardware, and optimizing on a hardware level by, e.g., performing operations with lower precision [3, 4, 5], power-gating i.e., by turning off parts of the chip [6, 5], dynamic voltage and frequency scaling (DVFS) [7], or evolving the memory hierarchy [8]. Even if some work advocates programmer input [4], the main focus is still on hardware implementation.

Other work is focused on writing rendering software optimized for energy usage [9, 10, 11, 12, 13], suggesting that programmers should optimize for power and energy in combination with rendering times. Pool et al. [14] and Johnsson et al. [10] showed that it is not sufficient to measure at rendering times alone as an indicator for energy efficiency, since different algorithms can have similar rendering times, but substantially different energy consumption. Thus, it is necessary to also measure the power usage to accurately evaluate the energy used by an algorithm. Another path of research is to statistically model the power usage, as done by Nagasaka et al. [15] and Pool et al. [14].

However, there has been no real discussion or comparison of the measurement methods. After measuring power and energy efficiency in several projects, we have gathered a set of advice for accurately measuring energy consumption using a simple method. Compared to our own previous method [10], we present and evaluate a simpler method, which avoids blocking the rendering thread, and is non-invasive. We also study the effects of blocking, to ensure that all submitted commands are executed until completion, and rendering a number of initial frames to achieve a steady state for the application before measuring commences. Detailed power measurements on both a mobile phone and on PCs with integrated GPUs are presented.

Similar to our previous method [10], the end result is per-frame energy measurements. Measuring over a number of frames to achieve a mean energy is easier and more straightforward, however, separating the energy on a per-frame basis is often beneficial, since it enables the possibility to correlate energy to other data collected per-frame, and also to find bottlenecks and anomalies on a per-frame basis.

2 Hardware setup

We have performed our measurements on either an iPhone 4S with a PowerVR SGX 543MP2 GPU, a PC with an Intel i7 3770k with an Intel HD4000 integrated GPU, or on a PC with an Intel i7 4850HQ with an Intel Iris Pro integrated GPU. The device is connected to a measurement station (Figure 1), similar to the one used by Johnsson et al. [10], which is using two shunt current sensors, which can accurately measure current up to 1 A for the iPhone 4S and through a Hall effect current sensor for the PC. Measurements on the iPhone are fed from a custom power supply with a known voltage of 3.9 volts. However, on the PC, as the ATX specification have a $\pm 5\%$ error tolerance, we simultaneously measure the voltage as well.

For the iPhone 4S, we intersect the current where the power supply connects to the battery. For the Intel i7 3770k, we intersect at the ATX $+12$ V power connector, which supplies the CPU with power and for the Intel i7 4850HQ we intercept all current and voltage supplied to the motherboard. As a result of intersecting the power at different points, the measurements on each platform include different parts of that platform. This may need to be considered when drawing conclusions. During all measurements, we had the iPhone in airplane mode, i.e., all transmitting hardware was shut off. We also shut off the adaptive screen brightness and set the brightness to its lowest setting.

3 Measuring Method

Our previous method for measuring per-frame energy inserted timestamps at the beginning of each frame, and called `glFinish()`) at the end, which blocked the rendering thread until all rendering commands were finished, so that only work

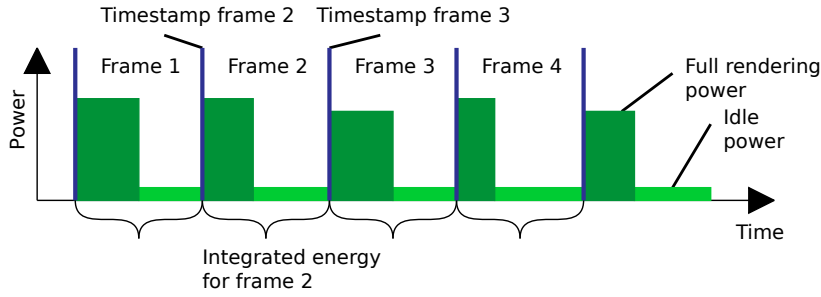


Figure 2: Our new method is illustrated here, with timestamps at the beginning of each frame. The integrated energy is from one frame's timestamp to the subsequent frame's timestamp. As can be seen, this also includes the idle energy when no rendering is done.

done with respect to that particular frame was measured [10]. This may have some benefits, e.g., if you are interested in only the energy for all the work done for a particular frame. In fact, something like this must be done if an architecture interleaves work over several frames, and if no energy from previous or future frames are to be included in the current frame's energy. For example, the Larrabee software rasterizer [16] interleaved work over frames, and early mobile phone graphics architectures [17] interleaved vertex shading (current frame) and pixel shading (previous frame) in sort-middle architectures.

However, in many cases, measuring from the start of one frame to the start of the next frame generates much more realistic energy consumption values, since that is how rendering is done most of the time, i.e., without blocking until all commands have finished executing and including interleaving effects (if any).

In the next subsection, we present a very simple method for measuring per-frame energy consumption by a real-time graphics application. The input is regularly spaced power measurements and a set of timestamps, either generated from the measured application or detected in the power measurements.

3.1 Our Method

Our new method records a timestamp at the beginning of each frame, and integrates the power until the beginning of the next frame's timestamp. It therefore guarantees that no energy is unaccounted for during a sequence of frames, and it also includes possible effects from algorithms exploiting inter-frame coherency. A schematic of the method can be seen in Figure 2.

For workloads where the source code is not available, it is possible to detect the beginnings of the frames in a semi-automatic way, by searching for features on the curve. In most rendering workloads, it is easy for humans to visually detect the be-

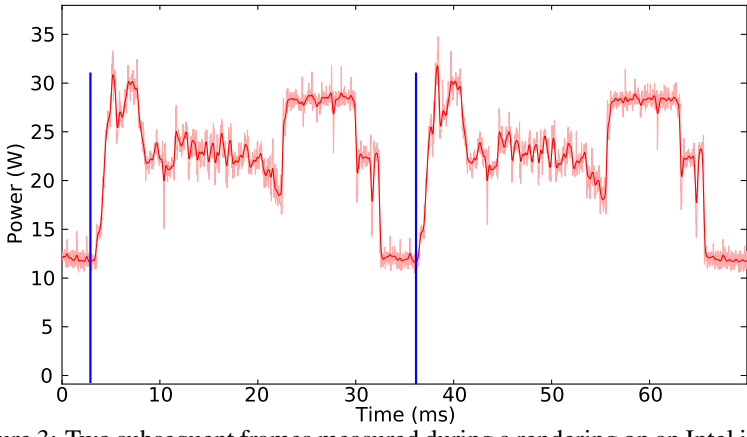


Figure 3: Two subsequent frames measured during a rendering on an Intel i7 3770k with the framerate capped to 30 frames per second. Note that the signature of the beginning of both frames are quite similar. By letting the user mark the beginning of one frame, an automatic search for the rest of the frame starts can be done.

ginning of each frame. An example is shown in Figure 3. We have developed software that, given a manually selected sample, records a window of samples around it. It then searches the entire curve and marks samples with similar surroundings as the recorded window by comparing the summed square distance between the curves. As there most often are several subsequent samples where the surrounding is sufficiently similar, it is necessary to find a limited segment of the curve where the samples similarity is below a threshold, and find the local minimum. As the characteristics of the frames can alter over the course of the rendering, it is often required to mark frame starts at different parts of the rendering, and perform multiple searches.

We have been able to mark frames on a large collection of measurements with different configurations and from different applications. For PC platforms, it greatly simplifies the search if V-sync is enabled, however, for most cases, it is still possible to find frame starts with V-sync disabled. Figure 4 shows a power curve, covering three frames, which are from a rendering of the Heaven benchmark by Unigine with V-sync disabled. For the most difficult cases, semi-automatic search is not possible. Our advice then is to enable V-sync, which is usually possible without changing the source code. For some difficult cases, it is still possible to find distinct features repeated in each frame. However, integrating between those features will provide an approximate result, since the distinct features may be offset from the frame start. In this case, our advice is to either accept the approximate result, and avoid basing conclusion on frames close to abrupt energy changes, or enable V-sync.

The described technique is non-invasive since no source code is needed (so that timestamps can be generated and saved), and it makes it possible to measure energy

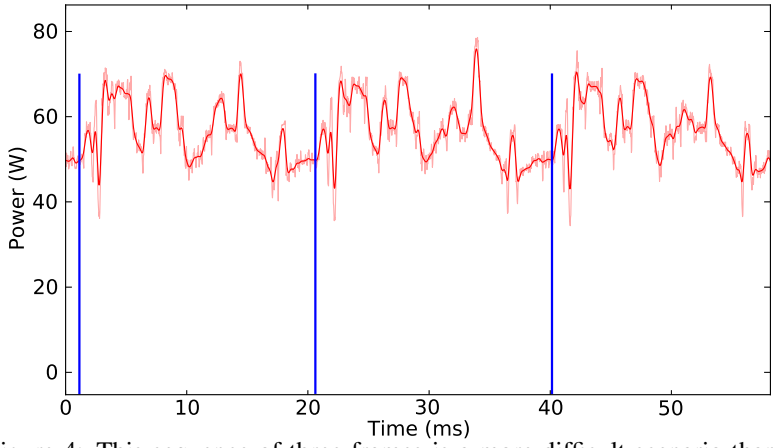


Figure 4: This sequence of three frames is a more difficult scenario than shown in Figure 3. This is measured from Unigine's Heaven benchmark on an Intel i7 4850HQ with V-sync disabled.

consumption on many more applications. If source code is available, however, we can still insert our own timestamps.

3.2 Discussion

We also tried two other methods, which were less successful. One was a generalization of our previous method [10], where the same frame was rendered r times after each other. The flush at the end was still used, but with higher values on r, the energy converged to the same as our new method. In another attempt, we tried to use b warm up frames, followed by a timestamp, and then another r frames, followed by an ending timestamp at the beginning of a trailing frame. To get one measurement for one frame, $b+r+1$ frames were rendered. Some architectures may also detect inter-frame coherency, and either reuse calculations between frames when possible or avoid costly memory transactions [18]. Note that for both these methods, these optimizations will likely increase their level of success, which can skew the results. Also, both these methods needed many more frames of rendering compared to our new method, which made them more time consuming.

4 Test Workloads

The rendering workload we have used for our test on the iPhone 4S is a synthetic scene, consisting of between 23 and 62 geometric objects per frame. Each object is a pre-tessellated quadrilateral with 2048 triangles. We render a sequence of 24 frames, and the framerate is capped to 30 frames per second. The vertex shader

performs three 4×4 matrix-vector multiplications, and the fragment shader performs two independent texture lookups, a dot product, and a normalization. We use two 256×256 RGB textures, sampled with linear minification and magnification filtering without mipmaps.

The main rendering workload used on PC is a camera path running through the Sponza scene, with 32 spotlights accumulated in a single rendering pass using forward rendering. It is capped at 30 frames per second and have been measured both with each frame ending with a blocking call to `glFinish()` to ensure that all rendering is finished, and without blocking. To show our ability to measure on real-world applications, we have also measured on the Heaven benchmark by Unigine. For our synthetic iPhone 4S application, we record timestamps in the application, and synchronize them with the power curve. For our PC applications, both the Sponza scene and the Heaven benchmark, we detect the beginnings of the frames as described in Section 3.1.

5 Observations

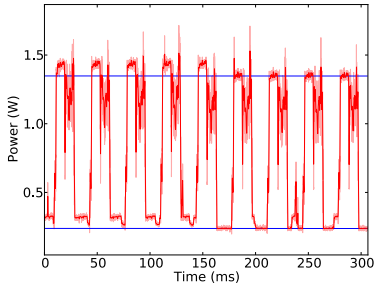
A set of important observations that we have made during several projects involving power and energy measuring for graphics workloads are presented in this subsection.

5.1 Application Launch

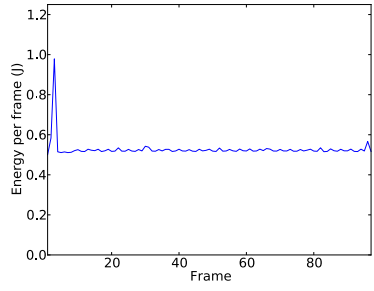
To get accurate results, we first note that it is important to avoid basing conclusions on the first few seconds after an application is launched. This is necessary as some platforms appear to have an increased energy consumption during that period. As seen in Figure 5c, the difference between measuring immediately after launch, or waiting for the platform to settle can be significant. This can also be seen in Figure 7, where our method, close to application launch (center) has a rendering power and an idle power that is approximately 0.05 W above the power obtained using our method with a frame measured a few seconds after the application is launched (left). According to our measurements, the number of frames with an increased power is around 48 on an iPhone 4S (see Figure 5c). As the decrease in power is sudden (see Figure 5a), it is likely a result of a state change in the System on a Chip, e.g., in its voltage and frequency scaling [7]. On an Intel i7 3770k, it is not necessary to perform more than a few frames for the application to settle. This can be seen in Figure 5b, where a sequence of frames, directly after the application launch and with identical graphics workload, have been rendered and measured.

5.2 Effects of Pipeline Flushing

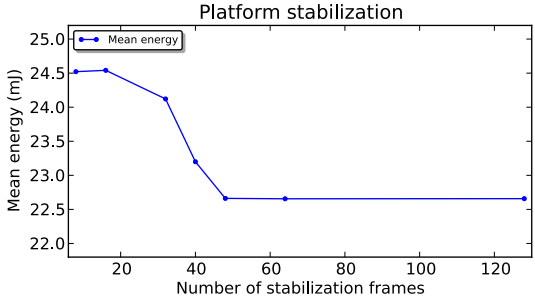
In Figure 6, we have compared our new method (Section 3.1) and our previous method [10]. These methods differ in two ways. First, the previous method measures from the beginning of a frame until the frame has finished rendering. Our



(a) A sequence of 9 frames, showing the power drop after a number of initial frames on an iPhone 4S.



(b) Rendering a number of identical frames on an Intel i7 3770k, showing that the energy settles after a few frames of rendering.



(c) Energy consumption on an iPhone 4S with different number of initial frames.

Figure 5: Upper left: a sequence of frames showing a sudden drop in power, that occur a number of frames after the application is launched. Blue lines are added to ease comparison, at both the idle level and top plateau level after the drop. Upper right: a sequence of identical frames on an Intel i7 3770k, measured directly after application launch, shows that the application settles after a few frames. Bottom: mean per-frame energy for renderings on an iPhone 4S as a function of the number of disregarded initial frames. As can be seen, it is necessary to disregard about 48 frames at after the application launch.

new method measures from the beginning of the frame until the beginning of the next frame. As a result, when using the previous method, the idle energy in between frames is not accounted for, and hence, the total energy of an entire rendering cannot be obtained by using a sum over per-frame energies. This is, however, possible with our new method, and this is usually what is desired.

The other difference is that the previous method calls `glFinish()` before recording the second, ending timestamp. `glFinish()` blocks until all commands submitted to the GPU has finished, which ensures that all rendering is performed

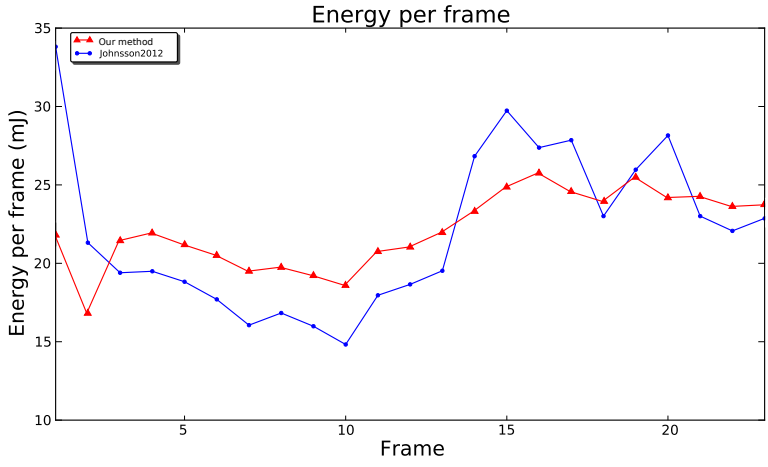
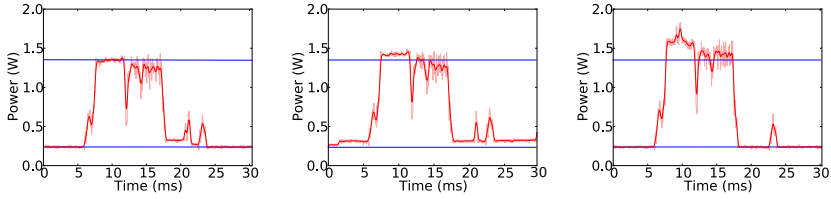


Figure 6: Comparison between the method used by Johnsson et al. [10] and our new, improved method, showing measurements performed on an iPhone 4S, rendering our synthetic scene. It is interesting to note is that while the introduction of `glFinish()` increases the energy usage and not measuring the idle time between frames decreases it, the method used in Johnsson et al. [10] can result in both higher and lower amounts of energy measured.

before returning. Surprisingly, the power usage on an iPhone 4S increases when `glFinish()` is used, which can be seen in Figure 7c (right). This should be compared to our method, which is shown in Figure 7a (left). We also note that the idle power is comparable. The only difference between these configurations is the introduction of `glFinish()` in the previous method. As `glFinish()` blocks the rendering thread, a consequence may be that the CPU is not able to sleep. In that case, that could be a possible explanation for the raised power consumption.

However, our measurements on an HD4000 graphics processor, both with and without `glFinish()`, revealed that there is very little difference. In our renderings with a capped frame rate, the mean energy per frame is 629.73 mJ with a pooled standard deviation of 21.07 mJ using `glFinish()`, and without flush, we obtain 630.18 mJ as mean energy per frame with a pooled standard deviation of 19.02 mJ. The difference in mean energy is 0.45 mJ, which is substantially lower than the pooled standard deviation, and hence a flush does not affect the measurements significantly.

Note also the small spikes at the end of the frames in Figure 7. These spikes are a semi-regular feature occurring on an iPhone 4S, where one spike occurs approximately 33ms apart. However, the frequency differs slightly from the capped frequency of the rendering, and thus the spikes are not occurring at the same position within the frames. There are also spikes of this size that have no observable frequency or pattern. This is, with a high probability, impact from the operating system. However, as they have a relatively small impact on the measured energy



(a) Our method (b) At application launch (c) With `glFinish()`

Figure 7: Power measurements of frame 9 in the synthetic scene on an iPhone 4S. A frame after the application has settled (left), a frame rendered soon after application launch (middle), a frame ended with `glFinish()`, as in the method used by Johnsson et al. [10] (right). Note that directly after the application launch, both the idle power between frames and the rendering power is approximately 0.05 W above the power of an identical frame after the application has settled. Horizontal blue lines, at the level of idle power and the plateau power level of our new method, are added to all three graphs, to aid the comparison. When calling `glFinish()` between frames, the idle power is the same as the idle power using our new method after the application has settled. However, the rendering power is significantly higher. The observable rendering times for all three frames are nearly identical.

(approximately 0.2 mJ per spike), and they usually occur $1-2$ per frame, the impact on the measurements is of no major concern. It is important not to base any conclusions on the placement or quantity of spikes in the presented frames, as they are both equally common within all methods and are irregularly positioned within the frames.

5.3 Operating System Interference

We also studied the standard deviation on the iPhone 4S as a function of the number of repeated measurements. Sometimes, we saw that the standard deviation increased abruptly even though the number of repeated measurements increased. Investigating this more closely, we saw that there were outliers that stretched over several frames in each measurement it occurred. Examples of such frames can be seen in the top of Figure 8, where the upper left is a normal frame and the upper right has an outlier.

Examining the actual power curves of these measurements, we discovered groups of frames with a significantly higher power signature, seen in the bottom right of Figure 8. In the bottom left part of the same figure, we show a power measurement on the iPhone 4S without any applications running. As can be seen, similar behavior (distinct spikes) were measured. These spikes occur with exactly five seconds apart.

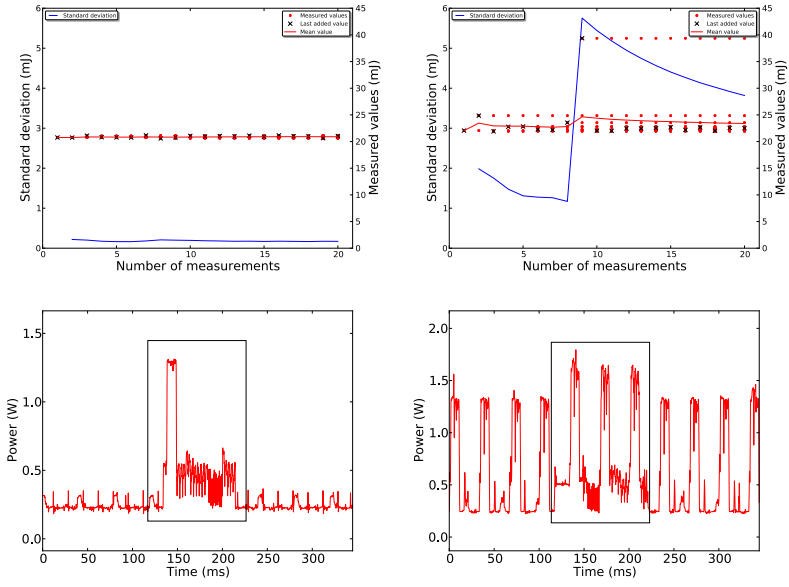


Figure 8: Top: standard deviation (blue curve) as a function of the number of performed measurements. Note that both figures use the same measurements, but represent different frames in the sequence. Top left: standard deviation slightly decreasing with more measurements. Top right: a decreasing standard deviation followed by a rapid increase as a result of an outlier. Bottom left: power curve from the iPhone 4S while it is not running any application. These spikes, with an approximate duration of 90 ms, occur every five seconds. Bottom right: each frame is supposed to be identical, however, a spike (bottom left) occurs in the middle and is overlaid on top of the graphics power. As the duration of the spike is approximately 90 ms, it affects 3-4 frames at 30 fps. Those spikes are the root cause of standard deviation increases (top right).

Similarly, there are frames and series of frames when running on Intel i7 3770k, where the power or rendering time, and sometimes both, are substantially higher. However, there is not an easy pattern to distinguish, as on the iPhone 4S.

There are several methods to handle these outliers, depending on the purpose of the measurement. If the sought energy is the actual energy used when rendering, the outliers represent real energy usage that should be included. In that case, more measurements might be needed to get a stable result. If the purpose of the measurement is to isolate the actual graphics algorithm, it is possible to detect frames with unacceptably high standard deviation, identify the outlier and exclude it from the measurement.

6 Conclusion

We have described a method which we have used extensively in our research, for measuring per-frame energy consumption for graphics workloads. The method consists of three straightforward steps:

1. Record power at a high frequency.
2. Find or sync a set of frame beginnings in the measured power recordings.
3. Integrate between frame beginnings to obtain per-frame energy.

Also, based on our experience with measuring energy, we have formed a set of best practices to avoid the pitfalls we ran into. These best practices include:

- Know your platform.
- Avoid changes to the measured workload.
- Disregard the first few seconds of the workload.
- Be aware of that the operating system might interfere.

Depending on the platform, changes to the software can have substantial effect on the energy consumption, e.g., we have discovered that using `glFinish()`, as a mean for isolating the consumption of a single frame, raises the power consumption. A general advice is to avoid changing the software at all, if possible, as it is hard to ensure that a change does not affect the power consumption. Our method does not depend on changes to the software. We have also experienced that on some platforms, applications have a higher power for a brief period after launching it. If the expected result is the general rendering power for an algorithm or applications, it is advisable not to measure, or not to base conclusions on, the first seconds after launch. In addition, we have also observed that the operating system can start some process that also substantially increase the measured energy.

However, these are only the pitfalls we have encountered. The most important advice is to get to know the measured platform, as that is the only way to avoid pitfalls that have not yet been encountered.

Bibliography

[1] S. Borkar and A. A. Chien, "The Future of Microprocessors," *Communications of the ACM,*, vol. 54, no. 5, pp. 67–77, 2011.

[2] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *38th International Symposium on Computer Architecture*, pp. 365–376, 2011.

[3] J. Pool, A. Lastra, and M. Singh, "Energy-Precision Tradeoffs in Mobile Graphics Processing Units," in *International Conference on Computer Design*, pp. 60–67, 2008.

[4] J. Pool, A. Lastra, and M. Singh, "Precision Selection for Energy-Efficient Pixel Shaders," in *High Performance Graphics*, pp. 159–168, 2011.

[5] J. Pool, A. Lastra, and M. Singh, "Power-Gated Arithmetic Circuits for Energy-Precision Tradeoffs in Mobile Graphics Processing Units," *Journal of Low Power Electronics,*, vol. 7, no. 2, pp. 148–162, 2011.

[6] P.-H. Wang, C.-L. Yang, Y.-M. Chen, and Y.-J. Cheng, "Power Gating Strategies on GPUs," *ACM Transactions on Architecture and Code Optimization,*, vol. 8, no. 3, pp. 13:1–13:25, 2011.

[7] B. Mochocki, K. Lahiri, and S. Cadambi, "Power Analysis of Mobile 3D Graphics," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 502–507, 2006.

[8] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McCaughy, D. Patterson, T. Anderson, and K. Yelick, "The Energy Efficiency of IRAM Architectures," in *International Symposium on Computer Arhchitecture*, pp. 327–337, June 1997.

[9] S. Collange, D. Defour, and A. Tisserand, "Power Consumption of GPUs from a Software Perspective," in *9th International Conference on Computational Science*, (Berlin, Heidelberg), pp. 914–923, Springer-Verlag, 2009.

[10] B. Johnsson, P. Ganestam, M. Doggett, and T. Akenine-Möller, "Power Efficiency for Software Algorithms running on Graphics Processors," in *High Performance Graphics*, pp. 67–75, 2012.

[11] R. Koduri, ""Power" of Realtime 3D Rendering," in *Beyond Programmable Shading (SIGGRAPH course)*, 2011.

[12] X. Ma, Z. Deng, M. Dong, and L. Zhong, "Characterizing the Performance and Power Consumption of 3D Mobile Games," *Computer,*, vol. 46, no. 4, pp. 76–82, 2013.

[13] M. Ribble, "Power Friendly GPU Programming," in *Beyond Programmable Shading (SIGGRAPH course)*, 2012.

[14] J. Pool, A. Lastra, and M. Singh, "An energy model for graphics processing units," in *International Conference on Computer Design*, pp. 409–416, IEEE, 2010.

[15] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical Power Modeling of GPU Kernels using Performance Counters," in *International Conference on Green Computing*, pp. 115–122, 2010.

[16] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics,*, vol. 27, no. 3, pp. 18.1–18.15, 2008.

[17] K. Beets, "Developing 3D Applications for PowerVR MBX Accelerated ARM Platforms," *Information Quartely,*, vol. 4, no. 3, pp. 26–34, 2005.

[18] K. Han, Z. Fang, P. Diefenbaugh, R. For, R. R. Iyer, and D. Newell, "Using Checksum to Reduce Power Consumption of Display Systems for Low-Motion Content," in *IEEE International Conference on Computer Design*, pp. 47–53, 2009.

Paper IV

Evaluation of Many-light Rendering Algorithms from an Energy Perspective

Björn Johnsson Tomas Akenine-Möller

Lund University

ABSTRACT

Recently, the performance of many-light algorithms, where thousands of light sources are used to compute the lighting in a scene, has improved so much that they have reached the realm of real-time rendering. In general, the algorithm that is considered "best" is the one that is the fastest in terms of time per frame. Given that power efficiency may become or already is one of the most important optimization factors for both hardware and software vendors for graphics, we take a different route and instead measure both energy usage per frame *and* frame time for a number of popular many-light rendering algorithms on an Intel Iris Pro. We use Pareto frontiers for each configuration to examine the possibilities for trade-offs between rendering time and energy consumption. Furthermore, we examine the optimal algorithms at each configuration, and are able to draw generalized conclusions on when each algorithm is most efficient. We also record several other statistics on the algorithms, e.g., bandwidth, and are able to draw further conclusions with regard to energy consumption.

Submitted to Visual Computer, 2014.

1 Introduction

The age of dark silicon is upon us [1]. This means that already today, it may not be possible to have an entire chip fully powered up at the same time, and instead, some part of the chip may be powered down or run at a lower frequency, while another part is fully powered and/or run at a higher frequency. In the future, with smaller and smaller process technologies for silicon, this will become even worse. In addition, memory bandwidth improvements are known to fall behind computing improvements [2]. As a consequence, when designing a CPU with an integrated graphics processor, optimizing for minimal power usage is probably the most important design aspect. We also note that the discrete GPU market is flattening out, while the market for graphics processors for tablets and smart phones is increasing rapidly [3]. This means that battery-powered devices become the main design point for many. Since up-time for mobile devices is critical, power optimization becomes yet more important.

At the same time, realistic lighting in real-time rendering is a highly desired feature. It is well known that dramatically more complex shading can be approximated by inserting virtual point lights into a scene, and then render the geometry with these lights using a simple shading model [4]. During the last five years, such *many-light* algorithms have been thoroughly optimized for real-time rendering [5, 6, 7, 8, 9], and several thousands of light sources can be rendered in real time even using integrated graphics in laptops.

In this paper, we evaluate the energy efficiency of several such many-light algorithms on a laptop chip with integrated graphics processor. Our study is unique in that the energy perspective is mostly left out of current software research in graphics, with some exceptions [10, 11, 12, 13], and we also believe that our results will be valuable in the near future, where energy will be at least as important as pure performance measured in time per frame. We have used four many-light rendering algorithms and recorded rendering time and frame energy, as well as the memory bandwidth (reads+writes) and the accumulated time spent on vertex, pixel, and compute shading in the graphics processor. For these algorithms, we have performed measurements at a variety of settings, with changes in multisample anti-aliasing (MSAA) rate, number of light sources, and testing both with and without V-sync. Our results can help a developer choose the best many-light algorithm for an energy-constrained device, and our study can possibly also lead to optimized many-light techniques in the future.

Next, we describe the different many-light algorithms that we have used, followed by an extensive evaluation and analysis in Section 3. Finally, we offer some concluding remarks and advice in Section 4.

2 Algorithms

We have performed measurements on four different many-light real-time rendering algorithms. Three of these, *tiled forward rendering*, *tiled deferred rendering*, and *visibility buffer rendering*, all aim at supporting thousands of lights. In addition, traditional forward rendering with a Z-prepass (Section 2.1) is measured as a comparison. In the following subsections, we will briefly describe these four algorithms followed by a discussion on how the tiled algorithms each handle MSAA.

2.1 Forward Rendering with a Z-prepass

This variant simply loops over all lights in each pixel. The computation per light per pixel involves computing distance attenuation, and evaluating lighting and texturing and accumulating everything. Without a Z-prepass, all this work would be unnecessarily done even for occluded surfaces. However, instead it is common to commence by rendering the scene to the depth buffer, and this pass is called the Z-prepass. Then follows the light accumulation pass, where the depth test is set to `equal`, and the scene is rendered again, which means that lighting is computed only for visible surfaces. The code for this is included in Lauritzen's framework [8], and our framework is based on his code. Again, note that this algorithm is simply included for comparison. For production many-light rendering, one of the three following algorithms would be used.

2.2 Tiled Forward Rendering

In order to handle thousands of lights in real time, it is crucial to divide the viewport into tiles, i.e., rectangular regions of pixels, and cull the individual light sources against the tiles' frusta before accumulating the light sources' contribution. This type of tiling is used both in tiled forward rendering (this section), in deferred rendering (Section 2.3), and also for the visibility buffer (Section 2.4).

Tiled forward rendering [7, 9] is similar to forward rendering with a Z-prepass. However, between the Z-prepass and the light accumulation pass, the algorithm launches a compute shader that calculates a minimum and maximum depth for each tile and culls the light sources against the frustum of each tile. Culling occurs when the distance attenuation is decreased below a certain threshold. The light sources that potentially affect the tile are saved per tile in a buffer. Note that this pass substantially reduces the number of light computations per tile since each light only has a limited region of influence due to light attenuation. As a result, a list of light numbers is created for each tile. In the light accumulation pass, all geometry is rendered once again, and each pixel shader determines which tile it resides in, and only reads and accumulates the relevant light sources. Tiled forward rendering is also known as Forward+ [7], and this algorithm is visualized in the top row in Figure 1. For brevity, this algorithm will be referred to as *TF*. As can be seen, we split the algorithms into three passes, similar to Lauritzen [14].

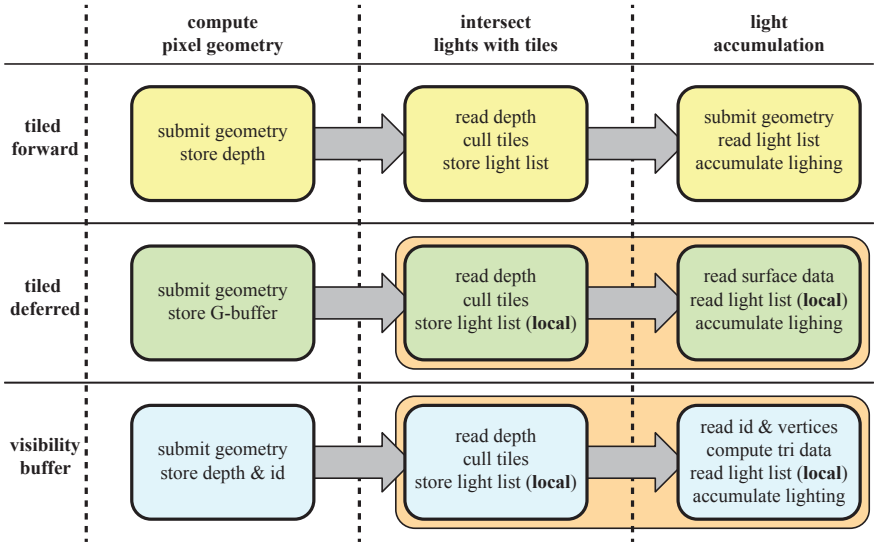


Figure 1: The three tiled many-light algorithms are *tiled forward* (top), *tiled deferred* (middle), and *visibility buffer* (bottom). All are divided into three passes, namely, compute pixel geometry, intersect lights with pixels, and light accumulation. The differences can be found in the rounded boxes. Note that the two algorithms at the bottom use local memory for the last two passes.

2.3 Tiled Deferred Rendering

Deferred rendering algorithms often use a G-buffer [15], which may contain depth, normals, diffuse textured shading, etc. Similar to tiled forward rendering, the screen is divided into tiles, and an extra pass is added before light accumulation, which determines which lights overlap with the tiles [6, 5]. In contrast to tiled forward rendering, the light accumulation pass is simply a full screen pass. Conceptually, each tile is processed in turn, and the light accumulation loop is only over the light list of the tile. In practice, the light culling pass and the light accumulation pass are performed within a single compute shader, and the light list can therefore reside in shared memory. This algorithm is also included in Lauritzen's code [8]. Hereafter, this algorithm is referred to as *TD*, and it is visualized in the middle row in Figure 1.

2.4 Visibility-Buffer Rendering

As an optimization for reducing memory bandwidth, Burns and Hunt [16] further developed deferred shading by using a *visibility buffer* instead of a G-buffer. While a G-buffer contains all necessary information for a fragment, a visibility buffer only contains a single integer, which identifies the drawcall and the primitive id. With this id, it is possible to re-transform the geometry and calculate all

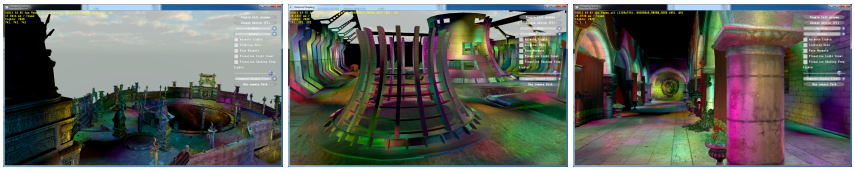


Figure 2: Screenshots from the scenes that we have used in our evaluation. From top to bottom: Arena (710k tris), Power Plant (115k tris), and Sponza (263k tris).

necessary information per fragment in the light accumulation pass. The core idea here is to reduce the memory bandwidth for creating the G-buffer. This algorithm is illustrated in the bottom row in Figure 1, and in the subsequent sections, we will refer to this algorithm as *VB*.

2.5 Multisample Anti-Aliasing

An important area for rendering algorithm evaluation is their ability to incorporate methods for reducing geometric aliasing in screen space. The most common method for doing this is multisample anti-aliasing (MSAA) [17]. MSAA works by sampling the visibility at a sub-pixel level, usually with 2–16 samples per pixel, while still evaluating the shading once per pixel. Most graphics hardware have native support for MSAA.

However, some algorithms, such as TD (Section 2.3) and VB (Section 2.4) cannot utilize that support natively. Instead, they use the hardware-supported MSAA sampling while creating their respective multi-sampled buffers. In the light accumulation pass, however, they then detect and handle different cases in a compute shader. The key technique when handling MSAA is to detect when all samples within a pixel belongs to the same polygon or the same surface and shade only once there (in the light accumulation pass, shown to right in Figure 1). Since these two algorithms have different data available in their buffers, they use different methods for this detection.

The TD rendering algorithm (Section 2.3) has depth, partial depth derivatives, and the normals available in the shader for the light accumulation pass. A single shading sample suffices if two tests are passed, and otherwise, multisampling is used [8]. The first test, done in the shader, is to check if each sample is within a small epsilon compared to the plane equation formed from the depth of the first sample in the pixel and its two depth derivatives. The second test is to compute a normal cone (using the normals of the samples in the pixel) and check whether the angle of the cone is smaller than a threshold value. With these tests, the algorithm avoids multisampling of near-coplanar connected polygons, however, it does not detect if the polygons have different textures or texture coordinates, and can thus fail to remove aliasing from sharp texture switches. For VB (Section 2.4), the visibility buffer contains polygon and drawcall ID, and can thus mimic the behavior

of hardware MSAA by multisampling all pixel where the samples originate from different polygons.

The two algorithms also handle cases where more than one shading sample is needed per pixel differently. TD calculates the shading of all samples, which in some cases is unnecessary, since several samples within the pixel can originate from the same polygon and have identical surface parameters. In contrast, VB sorts the samples into buckets, determined by the polygon and drawcall ID, and shades once for each bucket. The bucketing incurs a small overhead for all pixels, but for pixels with more than one shading sample, fewer samples are shaded.

3 Results

We have evaluated the different algorithms (Section 2) on an Intel Core i7 4850HQ with an integrated Intel Iris Pro 5200 GPU with 128 MB eDRAM and a TDP of 47 W. Each algorithm was executed with and without V-sync at $1\times$, $4\times$, and $8\times$ MSAA. When using V-sync, the frame rate was set to 30 frames per second. The resolution for all our rendered images is 1280×720 pixels. For each of these settings, the algorithms were measured with different numbers of light sources. The three tiled algorithms (Sections 2.2-2.4) were measured with 256, 512, 1024, 2048, 4096, and 8192 light sources. Forward rendering with Z-prepass (Section 2.1) was only rendered with 256, 512, and 1024 light sources since it is significantly slower than the other algorithms, and for more than 1024 light sources, the performance could not be considered real time any more. We render three different scenes, namely, the *Arena* scene, *Power plant* from the Direct X SDK, and *Sponza*. Arena and Power Plant were rendered from five different view points, and Sponza was rendered from six view points, where the view points were chosen to represent varying amounts of visible light sources and objects, and varying depth complexity and depth variation. Our scenes are depicted in Figure 2.

3.1 Data Collection

We have collected data using two different tools. First, we performed a high-frequency current and voltage measurement on the platform, collecting all power entering the motherboard at a frequency of 50kHz. This was done using a custom measurement station similar to that of Johnsson et al. [10], however it measures voltage as well as current and at a higher frequency (50 kHz). We connect our measurement station at the power connector supplying the motherboard, and thus our measurements include idle power. As a consequence, we also measure the power supplied to its solid state drive. However, since no data is read or written during the measurements, it should not disturb significantly. We use this data to extract per-frame timestamps. This gives us with a rendering time per frame, and by identifying well-known power signatures in the frame sequence, we are able to identify the first relevant frame in the entire trace [18].

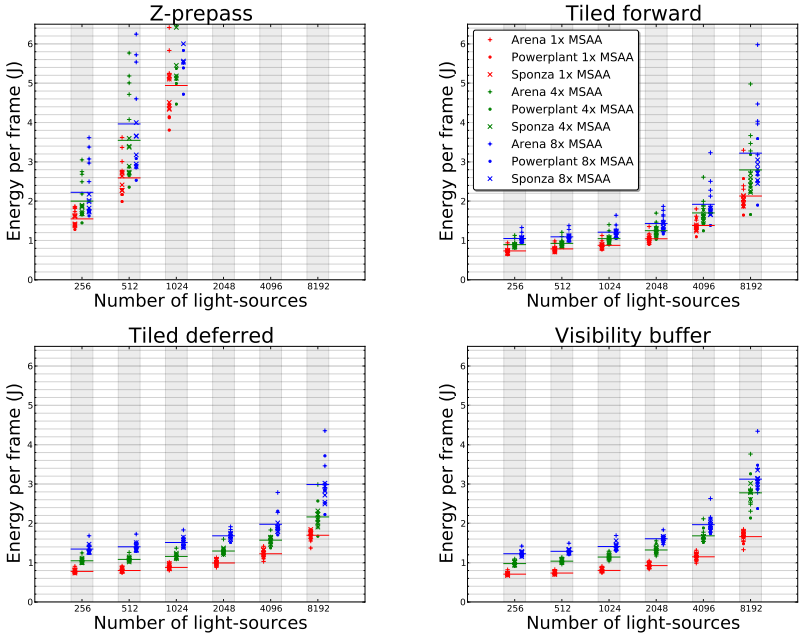


Figure 3: Energy measurements as a function of the number of light sources for all four algorithms with V-sync enabled. The mean for each light source configuration and MSAA rate is presented as a red, green, or blue horizontal line. First, note that Z-prepass (upper left) has a rapid increase in energy usage compared to the three algorithms with tile-based light culling. For these tiled algorithms, it is clear that $4\times$ MSAA uses more energy than $1\times$, and $8\times$ uses more than $4\times$, as expected. However, the differences between the horizontal lines also reveal that TF scales best when enabling MSAA and increasing the rate. The difference between the green and red lines for VB is also larger than the difference between blue and green, which indicates that the starting cost for MSAA is expensive for VB. In contrast, for TD, the penalty for $4\times$ is small with an almost linear effect on the number of samples. Also note that when rendering without MSAA, both VB and TD have a lower energy usage for larger numbers of light sources compared to TF. However, at $4\times$ MSAA, VB and TF have a similar energy usage, while TD's usage is visibly lower.

Second, we ran our application through Intel Graphics Performance Analyzer (GPA 2013 R3), and collected the read and write bandwidth from the GPU, and also the approximate execution time (including idle time) for the vertex (VS), pixel (PS), and compute shader (CS) stages. As a complement to the time spent in the shaders, we also extracted the percentage of the time spent executing instructions in order to get a more accurate measurement of the actual amount of computations performed by each algorithm. The bandwidth statistics were collected after the L3

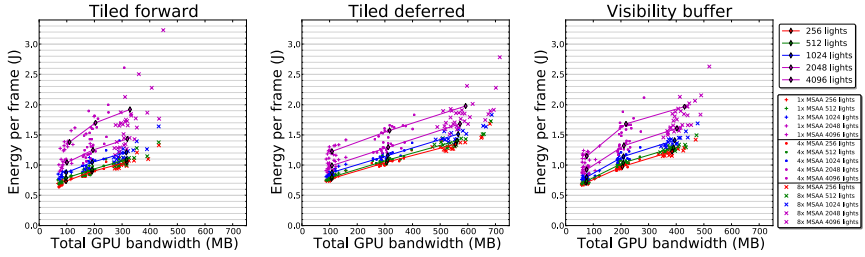


Figure 4: Energy consumption per frame as a function of GPU bandwidth usage for three tiled-based many-light algorithms. As expected, the energy increases with the number of MSAA samples ($+$ is $1\times$, $\bullet$ is $4\times$, and $\times$ is $8\times$) and increases with the number of light sources. It is also interesting to note that increasing the MSAA rate has a profound effect on the bandwidth usage. Another observation is that TD has, by far, the highest bandwidth usage, but this is not completely reflected in its energy usage. Note that that these measurements were performed with V-sync enabled.

cache, and thus before the 6 MB Last Level Cache (LLC) and the 128 MB eDRAM which the Intel i7 4850HQ is equipped with.

3.2 Analysis

Our analysis is divided in four parts, where the first reflects on how the tiling affect energy and rendering time, followed by reflections on bandwidth usage, decomposition of the algorithms' resource usage, and finally on the relationship between energy and rendering time, and how it affects a performance analysis of the algorithms.

3.2.1 Tiling

The top left diagram of Figure 3, which shows forward rendering with a Z-prepass, compared to the other three diagrams (which all use tiled-based many-light algorithms), clearly shows that tiling is a key enabler for many-light algorithms. For example, forward rendering with a Z-prepass uses about twice the energy for 256 light sources, and has a rendering time that is 50–65% longer. Therefore, we will only briefly mention forward rendering with Z-prepass and focus our evaluation on the tiled algorithms.

Figure 3 also show how the different tiled algorithms react to increasing MSAA rates. Looking at TD, the energy is linear to the MSAA rate, except at 8192 light sources, where it is slightly super-linear. For both TF and VB the energy is sub-linear, and the relationship is more apparent for VB. This is as expected according to how the algorithms handle MSAA samples as described in Section 2.5. For each of the tiled algorithms, the relationship between MSAA rate and rendering time is similar to the relationship between MSAA rate and energy.

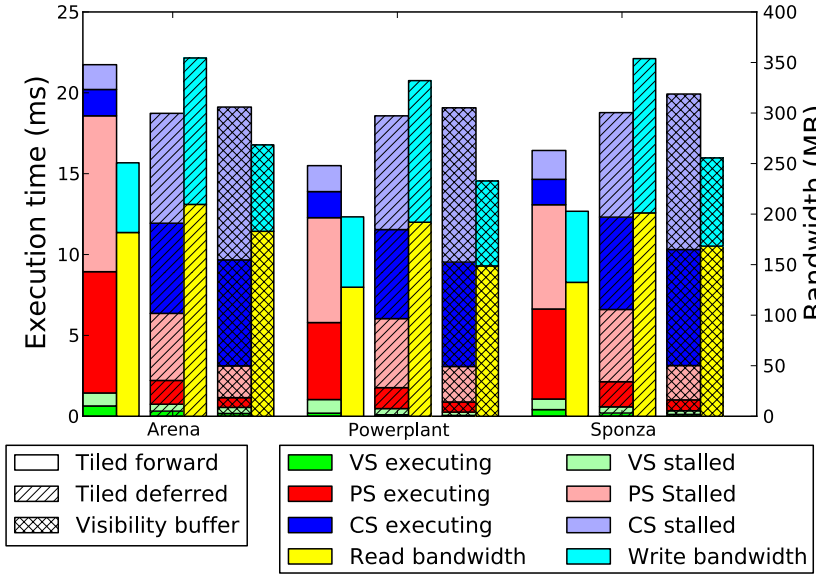


Figure 5: The execution time and bandwidth usage of each algorithm in the different scenes.

3.2.2 Bandwidth

Figure 4 shows the relationship between energy and bandwidth for the tiled algorithms. Increasing the MSAA rate or the number of light sources increases both the energy and the bandwidth. Another thing to note is that increasing the MSAA rate increases the bandwidth for TD linearly, and TF and VB sub-linearly, similar to energy and rendering time, probably due to the implementation differences previously discussed. However, it is even more interesting to note that while the bandwidth increase between $1\times$ MSAA and $8\times$ MSAA, at 4096 light sources, is more than 121% higher for TD than for TF, and the energy increase is only 38% higher. Even though bandwidth usage increases, the energy effect is less pronounced. This is likely due to current hardware's deep and large cache hierarchies.

3.2.3 Resource Decomposition

Figure 5 shows a decomposition of each algorithm's resource usage. We note that the compute shader time is almost scene-independent for TD and VB. This is expected, since those passes are essentially proportional to the number of light sources, and because we have averaged over the number of light sources in this diagram. The interesting data here is that the stall time for vertex, pixel, and compute shader stages are shown here as well. Those stall times can be seen as opportuni-

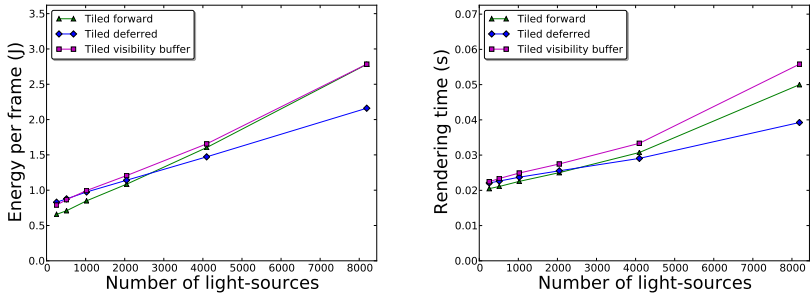


Figure 6: Per-frame energy and rendering time as a function of the number of light sources at $4 \times MSAA$, averaged over all viewpoints in all scenes, measured with V-sync turned off. Note the difference in behavior, especially between TF and VB at 8192 light sources.

ties for the different algorithms. For TF, the biggest opportunity lies in optimizing the pixel shader stalls, while it is biggest for the compute shader for TD and VB. This corresponds to where each algorithm performs its light accumulation, which should represent the largest part of their computations. Hence, the light accumulation pass is in need of most optimization for all algorithms. Finally, we note that for Arena, the scene with the most triangles, TF is slower, while for the scenes with a lower amount of triangles, it is faster. However, as neither algorithm spend a significant time on vertex shading, and as it is the time spent on pixel shading that increases for TF in Arena, the increased rendering time is a result of shading more samples. As TF performs a Z-prepass, we draw the conclusion that this is due to shading more quad fragments [17, 19] that affect only a part of its shaded fragments.

3.2.4 Energy and Rendering Time

In this Section, we analyze the algorithm performance regarding both energy and rendering time, and it is within this analysis our most interesting conclusions are drawn.

In Figure 6 we note that there is a significant difference in energy consumption and rendering time as a function of the number of light sources. It is especially noticeable at 8192 light sources, where TF and VB have nearly identical energy consumption per frame, but the rendering time has a 6 ms difference. Recalculating this difference to power, the result is that TF consumes 6 W more than VB, which is approximately 10% of the total platform power.

This difference in power is also visible in Figure 7 to the left. As can be seen TF has a small spread around 55 W, with a few outliers above 60 W. VB is concentrated at approximately 56 W at shorter rendering times but for rendering times longer than 10 ms, it is spread out between 49 W and 57 W. These outliers will be examined further on. This hints at a possibility to optimize for either time or energy, which

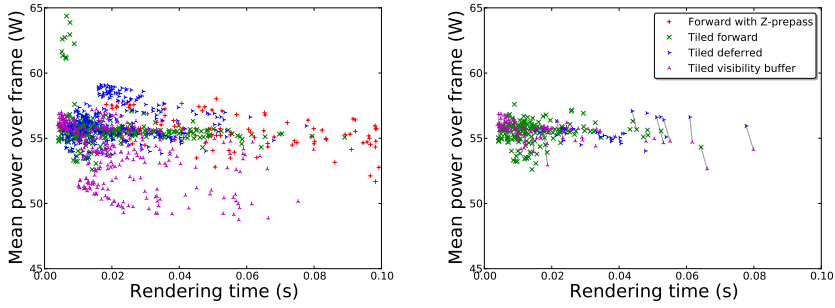


Figure 7: Mean power per frame as function of rendering time. Left: all data points with V-sync turned off, i.e., each point is a measured rendering covering all combinations of MSAA rate, number of light sources, scenes, viewpoints and rendering algorithms. Right: measured data points that lies on the Pareto frontiers for energy and rendering time created over the algorithms. Data point on the same Pareto frontier is connected with gray lines. Note that we measure all power supplied to the platform, and can thus get measurements above 47 W, the TDP of the CPU.

perhaps results in choosing different algorithms according to target efficiency.

However, if comparing all algorithms for a particular setting, i.e., for a specific number of lights, using a specific MSAA rate, or at a specific viewpoint in a scene, we create a Pareto frontier over energy and time[1]. Performing this for all settings with V-sync off, and only displaying the measured data points that lie on such a Pareto frontier, we have significantly fewer data points left, as can be seen in Figure 7 to the right. Note that there are very few (less than 6%), of the settings with more than one point on its Pareto frontier, and the spread in power is significantly smaller for all algorithms.

We conclude that, even as the platform is capable of rendering at both a significantly lower and significantly higher power than the majority of our measured points, those corresponding settings are neither faster nor more energy efficient. It is only in 6% of the cases where it is possible to trade energy for speed, i.e., lowering the energy consumption by using a slower algorithm or using less power.

Now we concentrate on only the values on the Pareto frontiers, i.e., the values that are shown to be most efficient in either time or energy (in 94% of the cases, both). If we divide them by MSAA rate and number of light sources, Figure 8, we can clearly distinguish a few cases where one of the algorithms is to prefer. For $1 \times MSAA$, it is clearly VB that is the most efficient, however, there are a few cases with few light sources where TF is more efficient. For the higher MSAA rates, it

[1]For a more conservative Pareto front, when comparing the energy of two algorithms, we compensate with an idle power of 14 W for the difference in rendering time, i.e., for the faster algorithm we add energy corresponding to that algorithm sleeping to achieve the same rendering time as the slower algorithm.

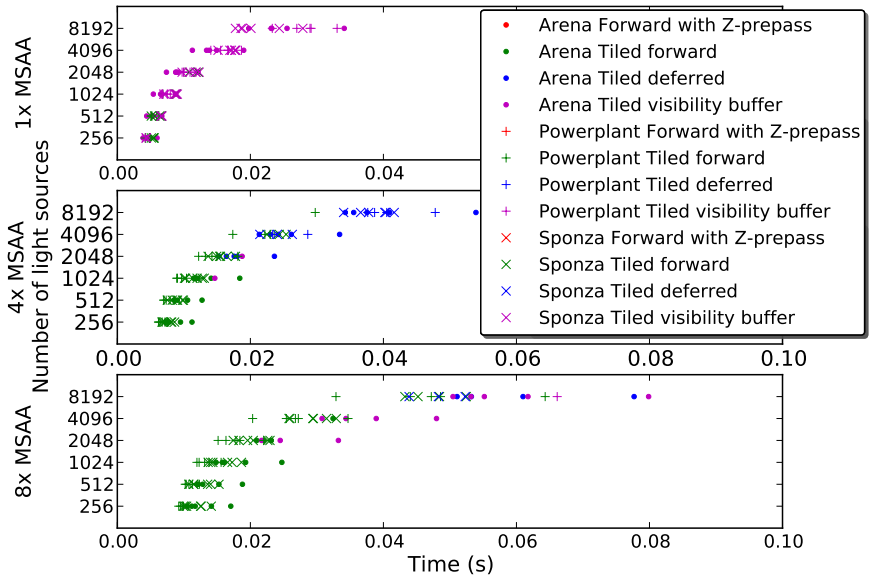


Figure 8: The measured data points included in the Pareto frontiers, separated by MSAA rate. Top: at 1× MSAA, VB is the most efficient. Middle: at 4× MSAA, TF is most efficient up to 2048 light sources. Above that, TD is more efficient. Bottom: at 8× MSAA TF is the most efficient up to 2048 light sources. At more light sources, it is not as clear whether TF, TD or VB is the most efficient, although TF still has a slight advantage at 4096 light sources.

is clear that for 2048 light sources or fewer, TF is to prefer. For higher amounts of lights, TD is more efficient at $4 \times MSAA$. However, at $8 \times MSAA$, TF is still at an advantage at 4096 light sources, however at 8192 light sources it is a mix of all three tiled algorithms.

Furthermore, Figure 8 reveals that, in cases where raising the number of light sources results in a transition between algorithms, it is always the renderings of the Arena scene, with the most triangles, that transitions first. This implies that the choice of algorithm is dependent on both the scene complexity and on the number of light sources.

As mentioned in Figure 7, there are noticeable groups of values outside of the power range (52.6–57.6 W) of the Pareto-efficient values. These groups are shown in Figure 9, compared to all measured values. The data points below 52.6 W are almost exclusively renderings with VB, and most render at $4 \times MSAA$. This is the setting where VB suffers the penalty of sorting samples into buckets for each pixel containing more than one triangle, but before the sorting is amortized over a larger number of MSAA samples. The configurations above 57.6 W mainly belong to TD at $8 \times MSAA$. There is also a small group of values with TF at $1 \times MSAA$, which

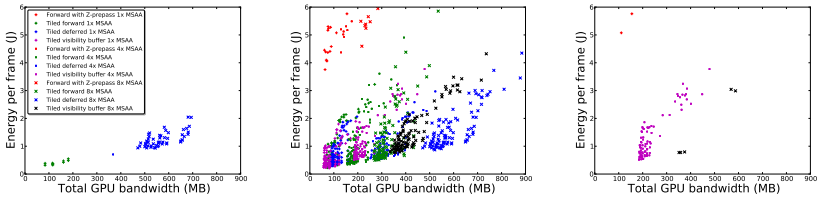


Figure 9: The configurations outside of the "efficient power range",52.6–57.6 W, configurations above to the right and configurations below to the left. Center image contains all configurations for comparison.

corresponds to all renderings of the Arena scene with 256 and 512 light sources, i.e., few light sources, but the heaviest geometry.

4 Conclusions

In this paper, we have shown that tiling light sources is an efficient way of reducing lighting calculations, for both forward and deferred rendering. Furthermore, we have shown that on Intel Iris Pro, increased bandwidth does not necessarily result in an as massive energy consumption increase as we expected. This is probably due to the efficiency of the large and deep cache hierarchy.

We also found that the fastest algorithm is also the one using least energy. However, there is a small set of cases where a time-energy trade-off exists. In addition, we recommend constructing Pareto frontiers over energy consumption and rendering time as a method for analyzing the possible trade-offs. By limiting our analysis to only these values, shown to be most efficient either for energy consumption or rendering time (in 94% of the cases, both), we can give concrete advice on which algorithm to use in each setting:

- $1 \times MSAA$: Visibility buffer (VB).
- $4 \times MSAA$: Tiled forward (TF) up to 2048 light sources, and then tiled deferred (TD).
- $8 \times MSAA$: Tiled forward (TF) up to 2048-4096 light sources, and then either tiled algorithm.

Bibliography

[1] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *38th International Symposium on Computer Architecture*, pp. 365–376, 2011.

[2] 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.

[3] M. Shebanow, "An Evolution of Mobile Graphics," in *Keynote talk at High Performance Graphics*, 2013.

[4] B. Walter, G. Alppay, E. Lafortune, S. Fernandez, and D. P. Greenberg, "Fitting Virtual Lights for Non-Diffuse Walkthroughs," in *Proceedings of ACM SIGGRAPH 1997*, pp. 45–48, 1997.

[5] J. Andersson, "Parallel Graphics in Frostbite – Current & Future," in *Beyond Programmable Shading (SIGGRAPH course)*, 2009.

[6] C. Balestra and P.-K. Engstad, "The Technology of Uncharted: Drake's Fortune," in *Game Developer's Conference*, 2008.

[7] T. Harada, J. McKee, and J. C.Yang, "Forward+: Bringing Deferred Lighting to the Next Level," in *Eurographics short papers*, pp. 5–8, 2012.

[8] A. Lauritzen, "Deferred Rendering for Current and Future Rendering Pipelines," in *Beyond Programmable Shading (SIGGRAPH course)*, 2010.

[9] O. Olsson and U. Assarsson, "Tiled Shading," *Journal of Graphics, GPU, and Game Tools,*, vol. 15, no. 4, pp. 235–251, 2011.

[10] B. Johnsson, P. Ganestam, M. Doggett, and T. Akenine-Möller, "Power Efficiency for Software Algorithms running on Graphics Processors," in *High Performance Graphics*, pp. 67–75, 2012.

[11] R. Koduri, ""Power" of Realtime 3D Rendering," in *Beyond Programmable Shading (SIGGRAPH course)*, 2011.

[12] B. Mochocki, K. Lahiri, and S. Cadambi, "Power Analysis of Mobile 3D graphics," in *Design, Automation and Test in Europe*, DATE '06, pp. 502–507, 2006.

[13] J. M. Vatjus-Anttila, T. Koskela, and S. Hickey, "Effect of 3D Content Simplification on Mobile Device Energy Consumption," in *International Conference on Making Sense of Converging Media*, pp. 263:263–263:268, ACM, 2013.

[14] A. Lauritzen, "Intersecting Lights with Pixels: Reasoning about Forward and Deferred Rendering," in *Beyond Programmable Shading (SIGGRAPH course)*, 2012.

[15] T. Saito and T. Takahashi, "Comprehensible Rendering of 3-D Shapes," in *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, pp. 197–206, 1990.

[16] C. A. Burns and W. A. Hunt, "The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading," *Journal of Computer Graphics Techniques,*, vol. 2, pp. 55–69, August 2013.

[17] K. Akeley, "Reality Engine Graphics," in *Proceedings of ACM SIGGRAPH 1994*, pp. 109–116, 1993.

[18] B. Johnsson and T. Akenine-Möller, "Measuring Per-Frame Energy Consumption of Real-Time Graphics Applications," *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, pp. 60–73, March 2014.

[19] 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.

Paper V

Efficient Multi-View Ray Tracing using Edge Detection and Shader Reuse

Magnus Andersson Björn Johnsson Jacob Munkberg Petrik Clarberg Jon Hasselgren Tomas Akenine-Möller

Lund University

ABSTRACT

Stereoscopic rendering and 3D stereo displays are quickly becoming mainstream. The natural extension is auto-stereoscopic multi-view displays, which by the use of parallax barriers or lenticular lenses, can accommodate many simultaneous viewers without the need for active or passive glasses. As these displays, for the foreseeable future, will support only a rather limited number of views, there is a need for high-quality interperspective antialiasing. We present a specialized algorithm for efficient multi-view image generation from a camera line using ray tracing, which builds on previous methods for multi-dimensional adaptive sampling and reconstruction of light fields. We introduce multi-view silhouette edges to detect sharp geometrical discontinuities in the radiance function. These are used to significantly improve the quality of the reconstruction. In addition, we exploit shader coherence by computing analytical visibility between shading points and the camera line, and by sharing shading computations over the camera line.

Visual Computer 27(6-8):665–676, 2011.

1 Introduction

Already around the beginning of the 20th century, inventions such as parallax barriers, lenticular lenses, and integral photography saw the light of day [1]. These form the basic foundations for stereoscopic and multi-view displays available today about a hundred years later. However, it is only during the past couple of years that this technology has seen more widespread use. For example, many new feature films are being produced with 3D in mind, and if this development continues, 3D cinema viewing may be more common than old-fashioned "normal" (2D) viewing. Recently, stereo rendering has also been re-introduced to real-time graphics with products targeting game players, and in South Korea, broadcasts of public television for stereo started in January 2010. In addition, automultiscopic displays, with many more than two views (e.g., 8 or 64) have become publicly available, and there is ongoing standardization work to augment MPEG to handle multiple views (see the previous work section and proposed algorithm by Ramachandra et al. [2]). All this implies that stereo and multi-view display technology is going mainstream, and may soon be ubiquitous.

Hence, it is expected that rendering for such displays will be increasingly important in the near future. For ray tracing-based algorithms, techniques such as multi-dimensional adaptive sampling (MDAS) [3] can easily render images for multi-view displays by sampling in the image plane, (x, y), and along the camera axis, v. While MDAS can be used for rendering any multi-dimensional space, we focus on a specialized algorithm for high-quality rendering using ray tracing with complex shaders for multi-view images.

We present two core contributions. First, we detect multi-view silhouette edges, which encode a subset of the geometric edges in the scene. We use these multi-view silhouette edges during reconstruction of the final image to improve quality. Second, since shading is expensive, we exploit the special case of multi-view image rendering by reusing the shading computations between views when possible. This is done efficiently using analytical visibility. Apart from these techniques, our algorithm uses adaptive sampling as described by Hachisuka et al. [3]. We render images with substantially higher image quality in equal amounts of time as state-of-the-art adaptive sampling techniques.

2 Previous Work

The most relevant work is reviewed here. Techniques and algorithms for multi-view rasterization are not discussed as they have little overlap with our work. We note that sampling the camera line can be seen as a temporal movement of the camera, and therefore, we also include work done in temporal sampling and ray tracing.

A four-dimensional light field [4, 5] is described by the *plenoptic* function. Isaksen et al. [6] introduce dynamically reparameterized light fields, which is a technique that enables real-time focus and depth-of-field adjustments from a full four-

dimensional light field. Any depth in the scene can appear in focus, by using a *wide-aperture* reconstruction filter. Light field generation can be done using both texture mapping-based algorithms or using ray tracing. In the latter case, they simply iterate over the cameras. Our work focuses on a reduced light field with three dimensions, where two dimensions form the screen plane, and the third dimension is a line where the camera can be positioned. We also use adaptive sampling, back tracing, and multi-view silhouette edges, in contrast to previous algorithms.

A suitable kernel for recovering a continuous light field signal from (sparse) samples is proposed by Stewart et al. [7]. They discuss how the reconstruction filter should be modified to handle non-Lambertian BRDFs. In addition, low-pass filtering is done to avoid ghosting, and then they use the wide-aperture approach [6] to isolate features around the focal plane. A high-pass filter is used to extract these features and those details are finally added back to the low-pass filtered version.

Determining the minimal sampling rate needed for alias-free light field rendering is discussed by Chai et al. [8]. They formalize previous results concerning light field sampling to a mathematical framework. Given the minimum and maximum depth of the scene, a reconstruction filter with an optimal constant depth can be designed. Recently, Egan et al. [9] presented a similar analysis for motion blur. Zhang and Cheng [10, 11] present non-rectangular sampling patterns and extend previous work to non-Lambertian surfaces.

Halle [12] introduces the notion of *interperspective aliasing* and presents early work on band-limiting signals for holographic stereograms. Both image-space filters and 3D filters with depth information are discussed. By using a camera with a "wide" aperture when capturing the light field, it will be perspectively filtered. His PhD thesis presents a thorough overview of rendering for multi-view displays [13], mostly focused on rasterization-like algorithms. Contemporary displays using parallax barriers or lenticular lens arrays have a limited number of views, and are prone to interperspective aliasing. Zwicker et al. [14] propose a pre-filtering step to avoid high-frequency information that the display cannot handle. A general framework for automultiscopic display anti-aliasing is presented. They combine the reconstruction filters by Stewart et al. and display pre-filtering. Furthermore, by remapping/compressing the depth range, more details can be preserved.

Kartch [15] discusses efficient rendering and compression techniques for holographic stereograms. In order to achieve this goal, scene polygons are mapped into a four-dimensional space. He presents an extensive overview of current autostereoscopic and holographic techniques and their respective advantages and drawbacks. He also presents an overview of compact lumigraph representations, comparing DCT and wavelet-schemes for compression. Rendering is also discussed in detail. If the cameras are placed along a coordinate axis, v, and the image plane is parametrized with coordinates (x,y), the triangle will sweep out a volume in (x,y,v) space. He proposes a coherent rendering algorithm based on (clipped) simplex primitives in four dimensions.

View interpolation techniques [16, 17] can be used to approximate a large number of views by reprojecting pixels from one or a few images with depth information. The main problems are missing data due to occlusion, and incorrect handling of view-dependent effects and transparency. This can be improved by using deferred shading and multi-layer representations [18, 19], and interpolation-based techniques have gained interest for multi-view video compression [20]. In contrast, our goal is high-quality multi-view image generation by directly sampling and reconstructing the three-dimensional radiance function.

A lot of research has already been done in the field of ray tracing stereoscopic images. In most cases, however, the existing stereoscopic ray tracing algorithms reproject samples from the left eye to the right eye, and detect whether there were any occlusions [21, 22, 23]. These algorithms only use two discrete camera locations (one per eye location) instead of a full camera line. Havran et al. [24] divide shaders into a view-dependent and a view-independent part, and reuse the latter of these between frames in animation. This technique resembles our shader reuse over the camera line. Finally, we refer to the article on spatio-temporal anti-aliasing by Sung et al. [25], which includes a thorough survey of the field.

To reduce the number of samples needed, we use a variant of multi-dimensional adaptive sampling (MDAS) [3]. This is a two-pass algorithm where in the first pass, samples are generated in the multi-dimensional space, focusing on regions where the local contrast between samples is high. In the second pass, the image is reconstructed by integrating the multi-dimensional function along all dimensions but the image dimension. They also compute structure tensors from gradients in order to better filter edges. MDAS is used to render motion blur, soft shadows, and depth of field.

Bala et al. [26] use edges projected to the image plane to prevent sample exchange over edge discontinuities during reconstruction. We extend this technique to three-dimensional patches in *xyv*-space. Similar to our reconstruction, Bala et al. also uses the projected edges to divide pixels into regions to produce anti-aliasing.

3 Overview

In this section, we will briefly present an overview of our rendering system. The space we want to sample has three dimensions, denoted (x, y, v), where v is the *view* parameter along the camera axis, and (x, y) are the image space parameters. This is illustrated in Figure 1. In our subsequent sections, where our algorithms are described in detail, we often refer to the epipolar plane, which is visualized in Figure 2.

Our goal is to sample and reconstruct the light field, $L(x, y, v)$, in order to display it in high quality on an automultiscopic display. These displays conceptually have a grid of pixels, where each pixel can simultaneously display n distinct radiance values projected towards different positions, v_i, along the camera axis. Each such *view* is visible within a small range, and there is usually some overlap between views based on the optical properties of the display. As the number of views is

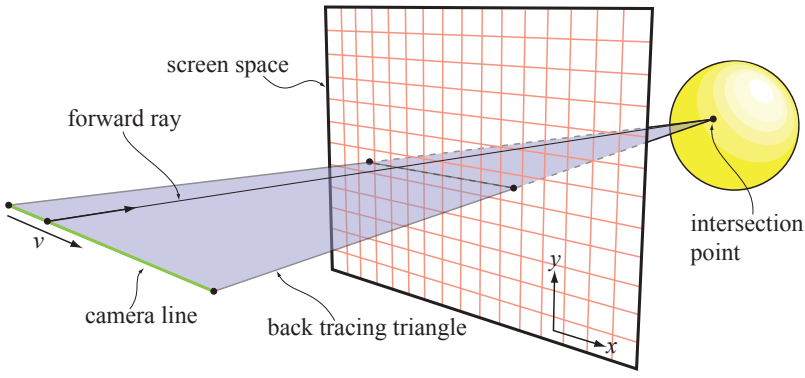


Figure 1: The three dimensions that we sample in our system consists of the image space, (x,y), and a continuous camera line, v. In our algorithm, we trace forward rays from a specific point on the camera line, i.e., for a specific v-value. From the intersection point on the hit surface, we create a back tracing triangle, which originates from the intersection point and connects with the entire camera line. This triangle is used for both analytical visibility and for detecting multi-view silhouette edges (Section 4).

limited (e.g., 8–64), the display is severely bandwidth-limited along v. To avoid interperspective aliasing, L is integrated against a view-dependent filter, g_i, in the reconstruction step to compute n distinct images, L_i:

$$L_i(x,y) = \int L(x,y,v)\, g_i(v) dv, \tag{1}$$

This effectively replaces the strobing effect seen when the human viewer moves, by blurring of all objects in front of and behind the focus plane. To determine final pixel values, L_i is integrated against a spatial anti-aliasing filter as usual.

Random sampling of L is expensive. For each sample, a ray has to be traced from v through (x,y), in order to find the intersection with the scene, and the shading has to be evaluated. We call these *forward rays*. There has, however, been little work done on efficient multi-view ray tracing. One approach that can be applied is multi-dimensional adaptive sampling (MDAS) [3], which focuses samples to regions of rapid change, and thus drastically reduces the total number of samples required. This is a general method, which can be seen as performing edge detection in the multi-dimensional space. Discontinuities in L have two main causes; abrupt changes in the shading, and geometrical edges in the scene. As each sample is shaded individually, the method does not exploit the shading coherence that is inherent in multi-view settings; the shading of a point often varies very slowly along the camera axis, and for the view-independent part, not at all.

We propose a specialized algorithm that extends upon MDAS for multi-view ray tracing, which addresses this inefficiency. Our adaptation still supports higher di-

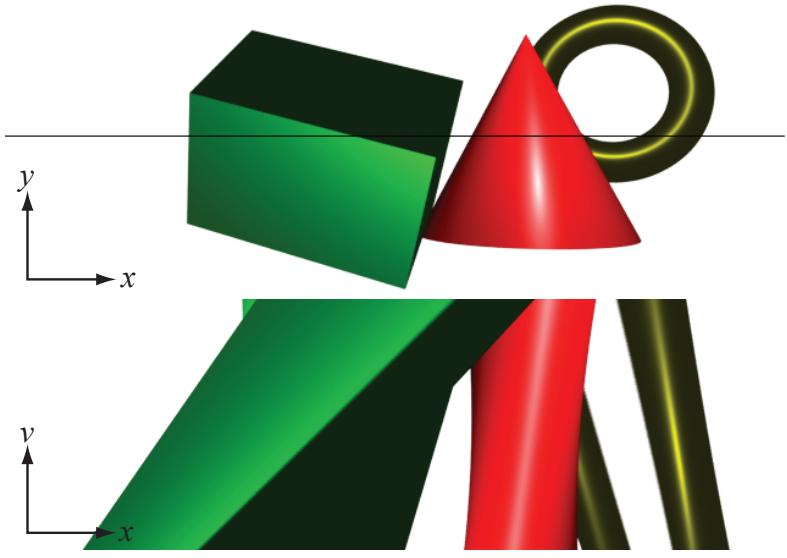


Figure 2: Top: a single image as seen from one point, v, on the camera line, viewed in the xy-plane. Bottom: epipolar image in the xv-plane for the black line (top) with a particular y-coordinate. We used extreme disparity of the camera to illustrate a wide range of events in the epipolar plane.

mensional sampling (such as area lights) to be handled as usual by MDAS, as long as they do not affect vertex positions (such as motion blur). In combination with sampling of L using adaptive forward rays, as described in MDAS, we also analytically detect geometric silhouette edges by tracing a triangle *backwards* from an intersection point towards the camera axis. This will be described in Section 4.1. In Section 4.2, we introduce a concept called *multi-view silhouette edges* to encode this data. For shading discontinuities, we rely on adaptive sampling.

The analytical detection of geometric edges provides the exact extent of the visibility between a shading point and the camera axis. Hence, we can insert any number of additional samples along the segments known to be visible, without any need for further ray tracing. The shading of these extra *back tracing samples* has a relatively low cost, as all view-independent computations can be re-used, e.g., the sampling of incident radiance. See Section 4.3 for details. In the reconstruction step, described in Section 5, a continuous function is created based on the stored samples and edge information. Whereas MDAS reconstructs edges by computing per-sample gradients, we know the exact locations of geometric silhouette edges. This significantly improves the quality of the reconstruction, and hence of the final integrated result.

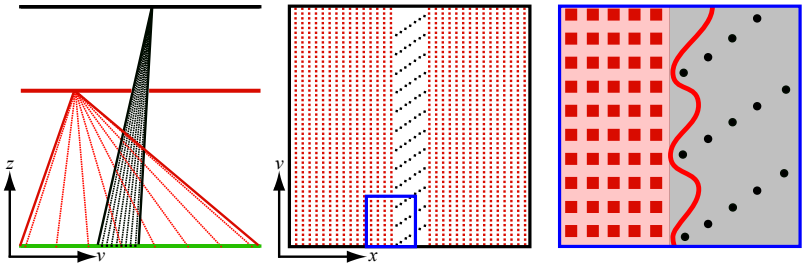


Figure 3: Non-uniform sample densities due to back tracing. The leftmost figure shows a scanline in the xz plane of a scene with a red horizontal line with a gap in focus, and a black horizontal line in the background. For each intersection point, a number of back tracing samples are distributed along the visible part of the camera line. In the middle figure, the epipolar plane is shown after a moderate number of forward rays has been traced and corresponding back tracing samples has been distributed. Note that the sample density is both significantly lower and less evenly distributed in the gray region. Finally, a close-up shows the incorrectly reconstructed edge as a red line, compared to the correct regions (light red and gray background). Using our approach, the correct result is obtained.

4 Sample Generation

When a forward ray, as seen in Figure 1, hits a surface, shading is computed and stored as a sample in the (x, y, v)-set in a *sample kD-tree*. We note that for the parts of the camera line, i.e., for the values of v, from which this intersection point can be seen, all view-independent parts of the shading can be reused. Our goal is to exploit this to add additional, low-cost samples to the sample set. However, as shown in Figure 3, this may cause sharp variations in the sample density of different areas. Unless care is taken during reconstruction, this may cause geometric edges to translate slightly. These sharp variations mainly occur around the geometric silhouette edges. We solve this using an edge-aware reconstruction filter (Section 5).

4.1 Analytical Back Tracing

The back tracing part of our algorithm commences after a forward ray has hit an object at an intersection point. The goal is to compute the parts of the camera line that are visible from the intersection point. This visibility is defined by the *silhouette edges*, and we find these analytically using an approach where the last part resembles shadow volume rendering [27] in the plane.

First, a *back tracing* triangle is created from the intersection point and the camera line, as seen in Figure 1. The back tracing triangle is intersection tested against the scene's *back facing* triangles. When an intersecting triangle is found, both of the triangle edges that intersects the back tracing *plane* are inserted into a hash table. If the edge is already in the hash table, this means that it is shared by two back

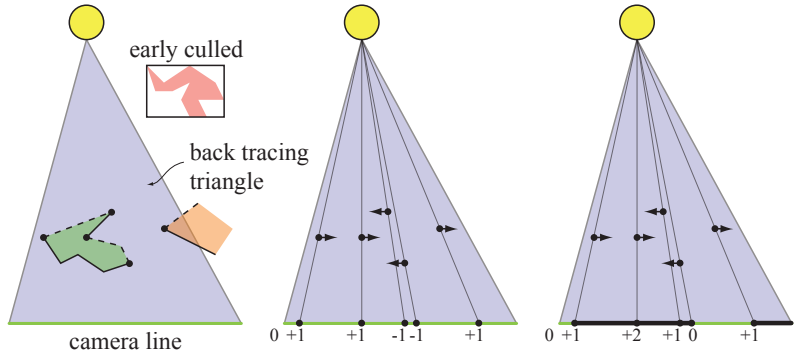


Figure 4: Illustration of how analytical visibility is computed from an intersection point (on the yellow circle) back to the camera line. The back facing lines, as seen from the intersection point, are solid. Left: the triangles of objects that cannot be early culled against the back tracing triangle are processed. A hash table is used to quickly find the silhouette points (black dots) on the remaining lines. Middle: the winding of the silhouette points are used to give them either a weight $+1$ or -1. Right: When the weights are summed from left to right, the occluded parts are the segments with summed weight greater than zero (black regions). The inner silhouettes are discarded from further processing.

facing triangles, and is removed from the table. For an edge to be a silhouette, as seen from the intersection point, it must be shared by one back facing and one forward facing triangle. This means that the edges that remain in the hash table after traversal are the potential silhouettes. This is illustrated to the left in Figure 4. Using this approach, closed 2-manifold surfaces can be handled, but by adding another hash table for front facing triangles, open 2-manifold surfaces can also be handled.

When the potential silhouette edges have been found, their intersection points with the plane of the back tracing triangle are projected to the camera line and are then processed from left to right. A counter is initialized to the number of silhouette edges whose triangles originates outside and to the left of the backtracing triangle. Each point has either weight $+1$ or -1 according to their *winding* order, (indicated with small arrows in Figure 4). Similar to shadow volume rendering [27], we compute the sum of the weights along the camera line, and when the sum is greater than zero, the camera line is occluded—otherwise, it is visible from the intersection point. In the end, we only keep the *outer* silhouette points, i.e., the end points of the black segments in Figure 4. Note that since the intersection point is visible from at least one point on the camera line, situations where the entire camera line is occluded cannot occur. The silhouette tests are executed on the edges of the primitives with no regards to the surface shader, so transparent or alpha-textured geometry will be handled as solid occluders, and are thus not supported. We leave this for future work.

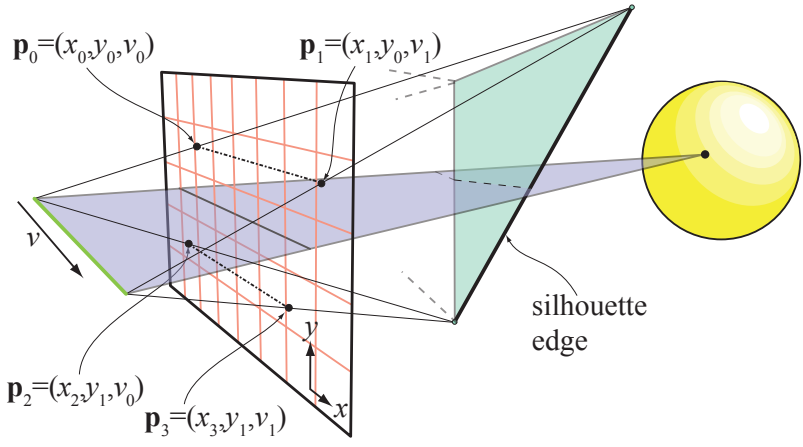


Figure 5: Multi-view silhouette edge generation. When the end points of a silhouette edge are projected towards the end points of the camera line, four points, $\mathbf{p_0}$, $\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_3}$, are obtained. These define the multi-view silhouette edge, which is a bilinear patch. The multi-view silhouette edges are used in our reconstruction algorithm to better preserve geometric edges. Note that the focus plane is identical to the screen-space plane in this figure.

4.2 Multi-View Silhouette Edges

In this section, we describe how we generate silhouettes in a multi-view setting. These *multi-view silhouettes* encode a subset of the geometric edges in the scene, and are used in Section 5 to substantially improve the final image quality in the reconstruction.

In the back tracing step described above, we have identified a number of *outer* silhouette points. Each such point is generated from a triangle edge, which is a silhouette for at least some v on the camera line and y in the image plane. Seen in the three-dimensional (x,y,v)-space, these lines will trace out a surface which is a bilinear patch. We call these patches *multi-view silhouette edges*, but we will use the term patch interchangeably. The geometrical situation is illustrated in Figure 5, where the multi-view silhouette edge is defined by the four points, $\mathbf{p_i}$, $i \in \{0,1,2,3\}$, obtained by projecting the triangle edge end points towards the end points of the camera line. Note that reducing the dimensionality by fixating y or v to a specific value, the patch will be reduced to a line segment in the xv- and xy-plane, respectively.

Each multi-view silhouette edge locally partitions the sampling domain into disjoint regions, which represent geometrical discontinuities in the scene, and this is what we intend to exploit in the reconstruction step. We note that it is only silhouettes that are visible from the camera line that are of interest. We know that at least one point on each patch is visible (the outer silhouette point found in the back tracing step), but other parts may very well be occluded by other patches. In

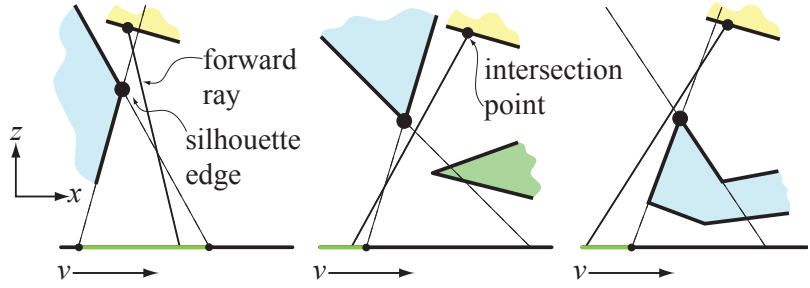


Figure 6: A multi-view silhouette is clipped in the v-dimension. We use the planes incident to the silhouette edge to limit the extents of the multi-view silhouette edge along the v-axis. The multi-view silhouette edges are clipped to the green intervals on the v-axis. In all three cases, we can only keep the part of the camera line where the edge is a silhouette, and where the intersection point is visible.

general, only a limited region of each patch will be visible from the camera line. The ideal solution would be to clip each patch against all others, in order to find the correct partitioning. We note that our multi-view silhouette edges resemble the EEE-events used when computing discontinuity meshes for soft shadows [28], however, our surfaces are more restricted and thus simpler, since there are only two different y-values and two different v-values. Despite this, full patch clipping is still much too expensive for our purposes.

To reduce the complexity of clipping, we lower the dimensionality of the problem by discretizing the y-dimension into *y-buckets*. Hence, in each y-bucket, the patch is reduced to a line segment, which lies in the epipolar plane, so this gives us a much more tractable 2D clipping problem. A silhouette edge in the (x,v) plane for some y can easily be retrieved by interpolating between the points $\mathbf{p_0}, \mathbf{p_2}$ and $\mathbf{p_1}, \mathbf{p_3}$ respectively. The depth of the silhouette can also be acquired by the inverse of the interpolation of $\frac{1}{z}$ for these end points. The retrieved silhouette edge is overly conservative since the edge is not necessarily an outer silhouette as seen from the entire camera line. By clipping the camera line with the two planes of the triangles that share the silhouette edge we can determine the range in v over which the edge is a true outer silhouette. The common case, and the two special cases are shown in Figure 6. Note that we only keep the portion of the multi-view silhouette in which there is a point known to be visible from the camera line (i.e., the part where the forward ray originated from). The silhouette is then inserted into an *edge kD-tree*, held by the y-bucket, and is clipped against the existing edges in that kD-tree.

When an existing edge and the new edge intersects, the edge with the largest depth is clipped against the other. The winding associated with an edge corresponds to which side of it is occluded. Hence, the part of the clipped edge that is in the non-occluded region of the other edge is kept. However, the hit point where the back facing triangle intersected the incoming silhouette edge is known to be a part of the new edge, so even if it is in the occluded region of an existing edge, we keep

it and rely on future back tracing to find the missing edge information. A typical edge clipping scenario is shown in Figure 7.

Instead of inserting the multi-view silhouette edge in to all buckets the patch overlaps, we insert only in the current y-bucket. We then rely on the back tracing from other intersection points to repeatedly find the same multi-view silhouette edge and insert it into the other buckets. This may seem counterintuitive, but otherwise occluded parts of the multi-view silhouette edge could be inserted, resulting in incorrect discontinuities in the light field. All per-patch setup (i.e., its corner points $\mathbf{p}_i$, depth, etc) is computed only once and stored in a hash table, keyed on the vertex id:s that make up the edge.

Finally, it should be noted that to find multi-view silhouettes, we only shoot forward rays through the *centers* of y-buckets in the current implementation. This avoids the problem of getting patches that are only visible along the forward ray's y-coordinate, but not from the y-bucket's center. In our renderings, we used four buckets per pixel, which did not give any apparent visual artifacts.

4.3 Shading Reuse

We define a *back tracing sample* as the radiance originating from the intersection point hit by a forward ray, as seen from a point, v, on the camera line. Once the analytical back tracing has identified a set of *visible* segments on the camera line, a set of these back tracing samples are generated very inexpensively in those segments, and these samples are inserted into the sample kD-tree holding all the samples. We exploit shader coherence by reusing shading computations for back tracing samples generated from the same intersection point. In addition, we can further reduce the shader evaluations using careful interpolation, since some parts of the shading equation vary slowly. Our technique is somewhat similar to approximate multi-view rasterization [29].

Each forward ray will hit an intersection point that needs to be shaded. In the simplest implementation, we divide the shader into a view-dependent (V_D) and a view-independent (V_I) part [24, 15]. For *every* back tracing sample, the V_D part of the BRDF is evaluated. This includes, for example, specular highlights, reflections, and refractions. The V_I part of the BRDF is only evaluated *once* for *all* the back tracing samples. Examples of this are irradiance gathering, visibility computations, and diffuse lighting. This alone results in a significant speedup. We also note that colors derived from anisotropic or mip mapped texture filtering belong to V_D. When looking towards a point on a surface, the texture lookups will get different footprints depending on the viewpoint.

Now, assume we have n_{bt} back tracing samples, where we want to evaluate *approximate* shading. Furthermore, assume that the V_D shading is computed at n_{vd} locations on the camera line, and texturing is done at n_{t} locations. We have observed that high-quality approximations of shading and texturing can be obtained using interpolation even if $n_{\mathrm{bt}} > n_{\mathrm{vd}} > n_{\mathrm{t}}$. This is illustrated in Figure 8. The type of quality that can be achieved using our technique is shown in Figure 9. Note

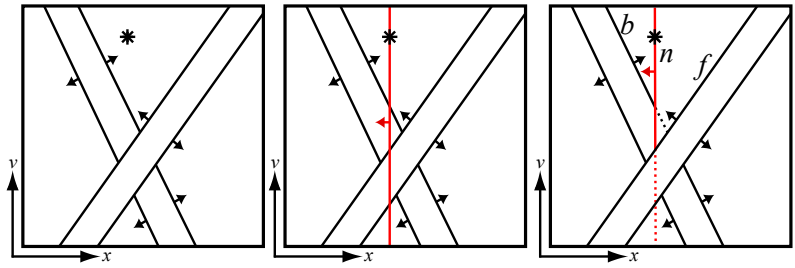


Figure 7: Insertion and clipping in a y-bucket. Left: the existing multi-view silhouette edges and the intersection (the star) between a new silhouette edge and the backtracing triangle. Middle: the 2D edge, which is in perfect focus (vertical), representing the new silhouette edge in this y-bucket, is added. Right: the clipped result, where the new edge, n, is clipped against the edges in the foreground, f, and clips one of the edges in the background, b. The winding (arrows) of the clipping edge determines which part to keep of the clipped edge.

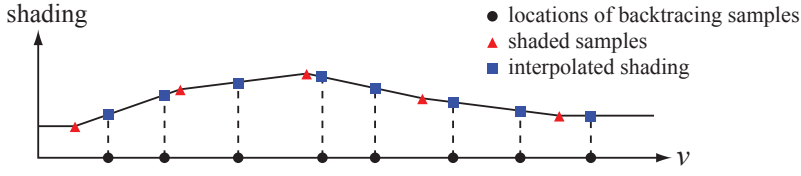


Figure 8: Illustration of our interpolation technique for shading. The locations of $n_{\mathrm{bt}} = 8$ back tracing samples (circles) and $n_{\mathrm{vd}} = 5$ view-dependent shading samples (triangles) are shown. The shading is evaluated (squares) at the back tracing samples using interpolation of the shading samples. Note that interpolation is replaced by clamping outside the shaded samples' range.

that our approximation resembles that of RenderMan-like motion blur rendering, where shading is evaluated less often compared to visibility [30]. For texturing, we have found that setting $n_{\mathrm{t}} = 1$ is sufficient. This introduces a small error, but this has been insignificant for all our images. In extreme cases, more texture samples may be required.

The number of view-dependent (n_{vd}) and texture (n_{t}) samples per forward ray is a variable parameter, and can be adjusted depending on the shader. The back tracing

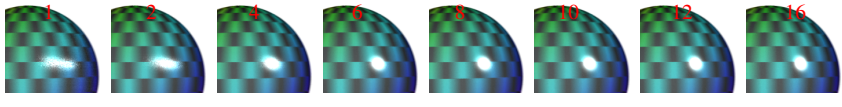


Figure 9: A specular sphere rendered with $n_{\mathrm{bt}} = 16$ back tracing samples (visibility) and a varying number of view-dependent shading samples (n_{vd}). Since the material used is highly view-dependent, the shading needs to be evaluated rather often for faithful reconstruction.

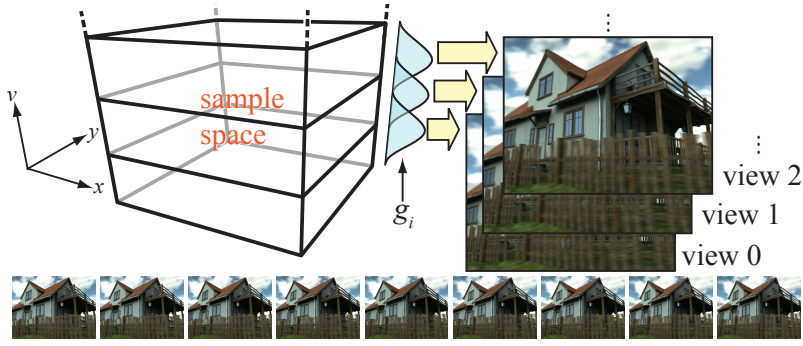


Figure 10: After the sample generation step, we have a three-dimensional sample set, (x, y, v), where v is the view dimension, i.e., the parameter along the camera line. From this sample set, n images are reconstructed. In this case, $n = 9$, where all nine images are shown at the bottom. An arbitrary filter kernel, g_i, along the v-axis can be used.

samples are jittered on the visible part of the camera line, and for the sub-segment where the forward ray originated in, the corresponding back tracing sample is removed. Instead the forward ray is used there, since it was generated by MDAS exactly where the error measure was largest. It is important to note that whereas MDAS always generates the next sample in the region with the highest error, the back tracing samples will, in general, be of lower importance since they are located in regions with lower local error estimates. Hence, there is no point in adding an excessive number of back tracing samples, as this will make sample kD-tree lookups substantially slower. However, for expensive shaders or when there are many views, the benefit of reusing shader computations and evaluating parts of the shaders at lower frequencies is still significant. Currently, we let the user set the n_{bt}, n_{vd}, and n_{t} constants. See Section 6.

5 Multi-View Image Reconstruction

In this section, we describe our reconstruction algorithm, which is, in many respects, similar to that of multi-dimensional adaptive sampling (MDAS) [3]. However, there is a fundamental difference, which is the use of multi-view silhouettes (Section 4.2) for more accurate reconstruction around geometric edges. Recall that all the samples are three-dimensional, i.e., have coordinates (x, y, v). In addition, each sample has a color. From this sample set, n different images will be reconstructed, where n depends on the target display. For example, for a stereo display, $n = 2$. See Figure 10 for an illustration.

The first steps of our reconstruction follow those of MDAS closely. First, the sample kD-tree is subdivided until it contains only one sample per node. Then, as each node is processed, the k nearest neighbor samples in Euclidean space are found and stored in the node. The samples found are representatives of the local

radiance that may ultimately affect the color in the kD-tree node region. Next, gradients are computed and used to calculate the structure tensors for each node.

After this initial reconstruction setup, the final color of a pixel is found by integrating along a line in the v-dimension, with (x,y)-coordinates fixed to the pixel center. In a multi-view setting, only a certain range of v needs to be processed (see Figure 10) for a particular view. The intersection of the line with the kD-tree nodes generate disjoint line segments. The clipped line segment's extent in the v-dimension determines the color weight of the node. The node color is computed by placing an integration point in the middle of the line segment $(x_p, y_p, v_{\text{mid}}^{\text{node}})$ and using an anisotropic nearest neighbor filter. This filter uses the node's structure tensor to find the sample, out of the k nearest neighbor representatives, with the shortest Mahalanobis distance to the integration point [3].

MDAS reconstructs high-quality edges when the sample kD-tree has a faithful representation of the true edge in itself, and MDAS sampling creates precisely that [3]. We modify MDAS reconstruction to exploit the multi-view silhouette edges. This gives an edge-aware reconstruction that is better suited the sample distribution generated by our algorithm. Our sample distributions will be less (locally) uniform (see Figure 3) due to insertion of many inexpensive back tracing samples (Section 4).

Therefore, we have developed an edge-aware reconstruction technique that uses the multi-view silhouettes to remedy this, and improves the reconstruction quality for the type of sample set that our algorithm generates.

Instead of directly using the Mahalanobis nearest neighbor, we now describe how to use the silhouette edges obtained in Section 4.2. Conceptually, we search for the two closest edges to the integration point in the closest y-bucket edge kD-tree. Recall that the intersection of a multi-view silhouette edge, which is a bilinear patch (Figure 5), with a plane $y = y_p$ is a line. This line is used to compute the x-coordinates at $v = v_{\text{mid}}^{\text{node}}$ for all the nearest multi-view silhouettes, and the two closest edges (if any) with $x < x_p$ and $x \geq x_p$ are kept. This situation is illustrated to the left in Figure 11.

Each patch, which is found in this process, is extrapolated in all directions to partition the (x,y,v) space into two disjoint sub-spaces. Each sample is then classified into one of these sub-spaces for each patch, giving four possible combinations. Samples that are separated by geometric edges in the scene are now classified into different sets, and the node color can be calculated by only considering the samples that are in the same sub-space as the integration point.

As a further improvement, when there are 1–2 edges overlapping a pixel for a sample kD-tree node we can produce high-quality anti-aliased edges in the images. Consider the cross section of the two patches at $v = v_{\text{mid}}^{\text{node}}$, which are lines in the xy plane. Clipped against the pixel extents in xy, this can result in the four different configurations as shown to the right in Figure 11, where the pixel contains 1–4 of the regions. The areas, a_i, of these clipped regions are computed, where the sum is equal to the area of a pixel. Most of the time, all but one $a_i = 0$, since the common

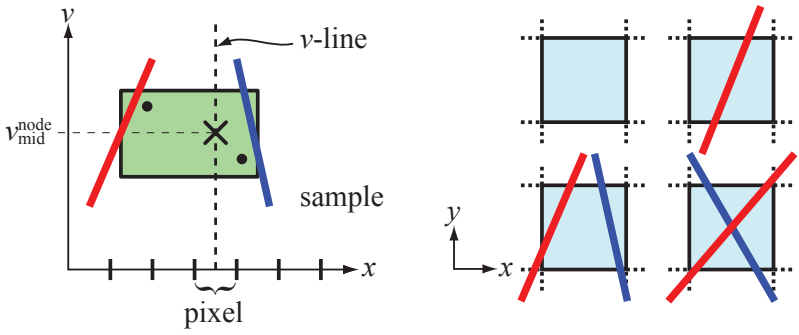


Figure 11: Left: a sample kD-tree node (green box) viewed in the epipolar plane, xv. The plus is the integration point on the integration line chosen for this kD-tree node, and black circles are samples. At $v = v_{mid}^{node}$, our algorithm searches in x for the two closest multi-view silhouettes (red and blue lines). Right: the closest multi-view silhouettes are evaluated for the y-value of the pixel center, and hence the edges are projected to screen space, xy. With two edges, four different cases can occur, where the pixel area is split into 1–4 regions. The color of each region is then computed using the closest samples in each region.

case is that the patches do not intersect the pixel extents at v_{mid}^{node}. When this does occur, though, we perform separate anisotropic nearest neighbor filtering for each region with $a_i > 0$. The final color contribution of a sample kD-tree node is then $\sum_{i=1}^{4} a_i \mathbf{c_i}$, where $\mathbf{c_i}$ is the color of region i.

In some rare cases, there will not be any samples in one or more regions overlapping a pixel. In this case, the reconstructed color cannot be calculated, and our remedy to this is to trace a new ray at a random location within that region, and use the color of that sample in the reconstruction. In order to avoid rebuilding the sample kD-tree, recomputing structure tensors etc., we simply discard the new sample after it has been used. This was found to be faster.

It should also be noted that while the silhouette patches greatly help by categorizing samples into different partitions, we only use a small subset of the local patches for performance reasons. Furthermore, there may even be patch information missing due to under-sampling. This, however, has not proven to be a problem in our test scenes.

As described in Section 4.2, the y-dimension was discretized to a small number of y-buckets per scanline. As a consequence, this approach is prone to aliasing in the y-direction. To counter this, we jitter the y-value of the integration point within the pixel, for each node and scanline during reconstruction, and use the y-bucket containing jittered integration point as a representative of the local geometry. This replaces aliasing with noise, given a sufficient number of y-buckets.

6 Results

We have implemented our algorithms as well as MDAS in our own ray tracer, which is similar to `pbrt` [31]. Our version of MDAS is at least as fast as Hachisuka et al's implementation. To be able to use a large number of samples, we split the image into rectangular tiles. Each tile is then sampled individually. The difference compared to Hachisuka et al. [3] is that our tiles are only split in the y-direction. Any split in the x- or v-directions would limit the extent of the epipolar plane, and thus reduce the advantage of our algorithm. Since all steps are independent for every tile, we can easily process the tiles in parallel using many threads. We have observed close to linear speedups with both our algorithm and MDAS using this approach.

Only Lambert and Phong shading were used in our test scenes. For irradiance gathering, we used 128 direct illumination rays and 16 indirect illumination rays with 8 secondary illumination rays per hit point (except for the dragon scene, where 64 were used), which makes the shading moderately expensive. All scenes were rendered with high-quality texture filtering [32], and using an overlapping box filter along the v-axis when filtering out the different images. Note that our target display is a Philips WoW 3D display with nine views, where each image has 533×400 pixels. In all our resulting images, only one out of these nine images is chosen. However, note that there is nothing in our algorithms that limits our results to this particular display.

All images were rendered on a Mac Pro with two 6-core Intel Xeon at 2.93 GHz with 8 GB RAM. To make it simpler to time different parts of the algorithms, we always use single-threaded algorithms. We split the y-dimension into 10 tiles to keep the memory usage down. Note that we find $k = 15$ nearest neighbors during reconstruction and allow six samples per node during sampling, which is a reasonable value shown previously [3]. The MDAS-scaling factor for the entire view axis, v, is set to 1.5. We use four y-buckets for all our images.

For all our test scenes, we used $n_{\text{bt}} = 12$ back tracing samples. For the house and fairy (top and middle in Figure 13) scenes, we found that using a single view-dependent shading sample ($n_{\text{vd}} = 1$) was enough to produce high quality images. All the materials used in these scenes (except for the waterspout in the house scene) have subtle or no specular components, and thus sparse shading works well. In contrast, the dragon in the bottom part of Figure 13 has a very prominent specular component. Here, we used $n_{\text{vd}} = 6$ view-dependent shading samples to better capture this effect. In Figure 12, we plot the image quality as a function of rendering time for two of our test scenes, using the logarithm of mean-square error (MSE) :

$$\log(\text{MSE}) = \log\left(\frac{1}{3pv}\sum_{v}^{views}\sum_{p}^{pixels}\Delta R_{pv}^2 + \Delta G_{pv}^2 + \Delta B_{pv}^2\right),$$

where each Δ is the difference in pixel color compared to the reference image. As can be seen, our algorithm renders images with the same quality as MDAS in about 10-20% of the time (for that particular quality).

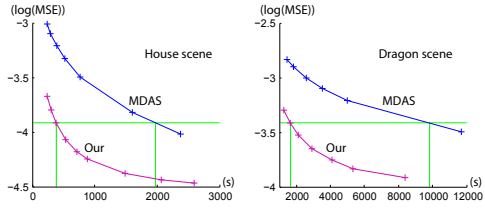


Figure 12: Error measurements for our algorithm (purple) and MDAS (blue) of the house and dragon scenes. The rendering time is on the x-axis and the log(MSE) over all nine views is on the y-axis.

Our algorithm and MDAS spend different amounts of time in the different parts of the algorithms. For our test scenes, MDAS typically spends most of its time in the sampling and shading step. Our algorithm is able to quickly produce more samples due to shader reuse and analytical back tracing, but on the other hand, our filter cost per sample is higher. As a consequence, more time is spent in the reconstruction step in our algorithm. Below is a table comparing the average times spent in each step for our algorithm and MDAS. The reconstruction setup involves the kD-tree subdivision, nearest neighbor (NN) lookup and tensor calculations, where the NN setup alone amounts to roughly 90% of the setup time.

	House		Fairy		Dragon	
Algorithm step	MDAS	Our	MDAS	Our	MDAS	Our
Sampling & shading	95%	66%	95%	65%	>98%	60%
Reconstruction setup	4%	22%	4%	22%	1%	8%
Filtering	1%	12%	1%	13%	<1%	32%

While the house and fairy scenes have relatively few silhouette edges, the grass in the dragon scene produces significantly more silhouettes. Consequently, our algorithm has to spend more time in the filtering step, since the cost of updating the active patches increases. There are also more pixels that use the improved filtering described in Section 5, and more regions lack samples and hence require retracing with forward rays.

We found that, besides silhouette patches, our algorithm benefits from the reused irradiance gathering. For the dragon scene, we only trace about half the number of forward rays compared to MDAS, but due to back tracing, we have roughly five times more samples than MDAS in the filtering step. As a consequence, we shoot only half the number of irradiance gathering rays.

7 Conclusions and Future Work

We have presented a novel ray tracing algorithm, where we have exploited the particular setting of multi-view image generation. This was done by computing analytical visibility back from the intersection point of a forward ray, and by quickly generating back trace samples, which reuse common calculations for shading. In

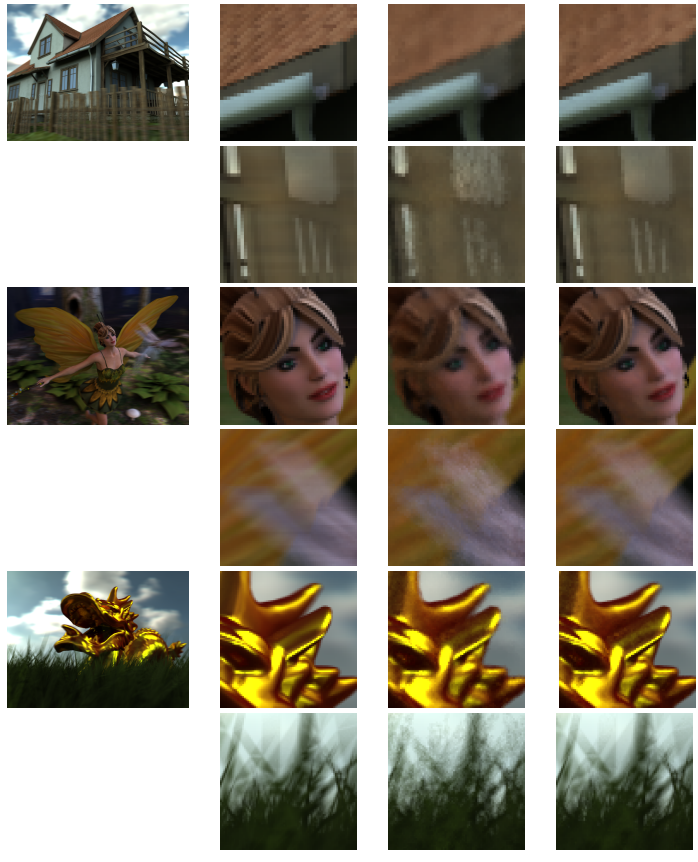


Figure 13: Image quality comparison for equal time renderings. The noise in our images is greatly reduced compared to MDAS. The images are rendered with 16k initial random samples, after which adaptive sampling kicks in. **Top:** House scene (93.6k tris) after 676s rendering with our algorithm and 712s with MDAS. **Middle:** Fairy scene (194k tris) after 504s with our algorithm and 519s with MDAS. **Bottom:** Dragon scene (301k tris) after 4114s with our algorithm and 4973s with MDAS.

addition, we detected multi-view silhouette edges, and used these during final reconstruction for improved image quality around geometrical edges. Our results indicate that we can render images with the same quality as generated by MDAS up to 13 times faster. An interesting result of our research is that it opens up new avenues for future work. We have shown that we can exploit geometrical information about edges to drastically reduce the sample rate needed to generate an image, and we believe that this can be explored also in other contexts, such as depth of field and motion blur.

Bibliography

[1] B. Javidi and F. Okano, *Three-Dimensional Television, Video, and Display Technologies*. Springer-Verlag, 2002.

[2] V. Ramachandra, M. Zwicker, and T. Nguyen, "Display Dependent Coding For 3D Video on Automultiscopic Displays," in *IEEE International Conference on Image Processing*, pp. 2436–2439, 2008.

[3] T. Hachisuka, W. Jarosz, R. Weistroffer, G. H. K. Dale, M. Zwicker, and H. Jensen, "Multidimensional Adaptive Sampling and Reconstruction for Ray Tracing," *ACM Transactions on Graphics,*, vol. 27, no. 3, pp. 33.1–33.10, 2008.

[4] M. Levoy and P. Hanrahan, "Light Field Rendering," in *Proceedings of ACM SIGGRAPH*, pp. 13–42, 1996.

[5] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen, "The Lumigraph," in *Proceedings of ACM SIGGRAPH*, pp. 43–54, 1996.

[6] A. Isaksen, L. McMillan, and S. Gortler, "Dynamically Reparameterized Light Fields," in *Proceedings of ACM SIGGRAPH*, pp. 297–306, 2000.

[7] J. Stewart, J. Yu, S. Gortler, and L. McMillan, "A New Reconstruction Filter for Undersampled Light Fields," in *Eurographics Symposium on Rendering*, pp. 150–156, 2003.

[8] J.-X. Chai, X. Tong, S.-C. Chan, and H.-Y. Shum, "Plenoptic Sampling," in *Proceedings of ACM SIGGRAPH*, pp. 307–318, 2000.

[9] 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, p. article no 93, 2009.

[10] C. Zhang and T. Chen, "Generalized Plenoptic Sampling," Tech. Rep. AMP01-06, Carnegie Mellon, 2001.

[11] C. Zhang and T. Chen, "Spectral Analysis for Sampling Image-Based Rendering Data," *IEEE Transactions on Circuits and Systems for Video Technology,*, vol. 13, no. 11, pp. 1038–1050, 2003.

[12] M. Halle, "Holographic Stereograms as Discrete Imaging Systems," in *Practical Holography VIII (Proceedings of SPIE)*, vol. 2176, pp. 73–84, 1994.

[13] M. W. Halle, *Multiple Viewpoint Rendering for Three-Dimensional Displays*. PhD thesis, MIT, 1997.

[14] M. Zwicker, W. Matusik, F. Durand, and H. Pfister, "Antialiasing for Automultiscopic 3D Displays," in *Eurographics Symposium on Rendering*, pp. 73–82, 2006.

[15] D. Kartch, *Efficient Rendering and Compression for Full-Parallax Computer-Generated Holographic Stereograms*. PhD thesis, Cornell University, 2000.

[16] S. E. Chen and L. Williams, "View Interpolation for Image Synthesis," in *Proceedings of ACM SIGGRAPH*, pp. 279–288, 1993.

[17] W. R. Mark, L. McMillan, and G. Bishop, "Post-Rendering 3D Warping," in *Symposium on Interactive 3D Graphics*, pp. 7–16, 1997.

[18] N. Max and K. Ohsaki, "Rendering Trees from Precomputed Z-Buffer Views," in *Eurographics Rendering Workshop*, pp. 45–54, 1995.

[19] J. Shade, S. Gortler, L.-w. He, and R. Szeliski, "Layered Depth Images," in *Proceedings of ACM SIGGRAPH*, pp. 231–242, 1998.

[20] M. Zwicker, S. Yea, A. Vetro, C. Forlines, W. Matusik, and H. Pfister, "Display Pre-filtering for Multi-view Video Compression," in *International Conference on Multimedia (ACM Multimedia)*, pp. 1046–1053, 2007.

[21] S. Adelson and L. F. Hodges, "Stereoscopic Ray-tracing," *The Visual Computer,*, vol. 10, no. 3, pp. 127–144, 1993.

[22] S. J. Badt, "Two Algorithms for Taking Advantage of Temporal Coherence in Ray Tracing," *The Visual Computer,*, vol. 4, no. 3, pp. 123–132, 1988.

[23] J. D. Ezell and L. F. Hodges, "Some Preliminary Results on Using Spatial Locality to Speed Up Ray Tracing of Stereoscopic Images," in *Stereoscopic Displays and Applications (Proceedings of SPIE)*, vol. 1256, pp. 298–306, 1990.

[24] V. Havran, C. Damez, K. Myszkowski, and H.-P. Seidel, "An Efficient Spatio-Temporal Architecture for Animation Rendering," in *ACM SIGGRAPH Sketches & Applications*, 2003.

[25] K. Sung, A. Pearce, and C. Wang, "Spatial-Temporal Antialiasing," *IEEE Transactions on Visualization and Computer Graphics,*, vol. 8, no. 2, pp. 144–153, 2002.

[26] K. Bala, B. Walter, and D. P. Greenberg, "Combining Edges and Points for Interactive High-Quality Rendering," *ACM Transactions on Graphics*, vol. 22, pp. 631–640, 2003.

[27] F. Crow, "Shadow Algorithms for Computer Graphics," in *Computer Graphics (Proceedings of ACM SIGGRAPH)*, pp. 242–248, July 1977.

[28] G. Drettakis and E. Fiume, "A Fast Shadow Algorithm for Area Light Sources using Backprojection," in *Proceedings of ACM SIGGRAPH*, pp. 223–230, 1994.

[29] J. Hasselgren and T. Akenine-Möller, "An Efficient Multi-View Rasterization Architecture," in *Eurographics Symposium on Rendering*, pp. 61–72, 2006.

[30] A. Apodaca and L. Gritz, *Advanced RenderMan: Creating CGI for Motion Pictures*. MKP, 2000.

[31] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. MKP, 2004.

[32] H. Igehy, "Tracing Ray Differentials," in *Proceedings of ACM SIGGRAPH*, pp. 179–186, 1999.

Paper VI

Design and Novel Uses of Higher-Dimensional Rasterization

J. Nilsson P. Clarberg B. Johnsson J. Munkberg J. Hasselgren R. Toth
M. Salvi T. Akenine-Möller

Lund University

ABSTRACT

This paper assumes the availability of a very fast higher-dimensional rasterizer in future graphics processors. Working in up to five dimensions, i.e., adding time and lens parameters, it is well-known that this can be used to render scenes with both motion blur and depth of field. Our hypothesis is that such a rasterizer can also be used as a flexible tool for other, less conventional, usage areas, similar to how the two-dimensional rasterizer in contemporary graphics processors has been used for widely different purposes other than the original intent. We show six such examples, namely, continuous collision detection, caustics rendering, higher-dimensional sampling, glossy reflections and refractions, motion blurred soft shadows, and finally multi-view rendering. The insights gained from these examples are used to put together a coherent model for what a future graphics pipeline that supports these and other use cases should look like. Our work intends to provide inspiration and motivation for hardware and API design, as well as continued research in higher-dimensional rasterization and its uses.

High Performance Graphics pages 1–11, 2012.

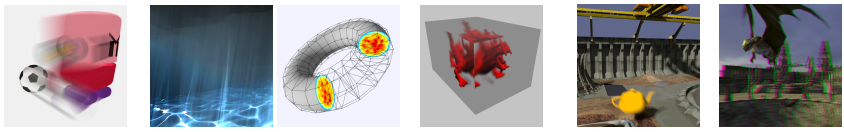


Figure 1: We demonstrate a number of novel use cases for (hypothetical) higher-dimensional rasterization hardware. In the first three, we exploit the volumetric extents of time-continuous and defocused triangles to perform geometric computations, i.e., occlusion and collision detection, caustic rendering, and use the rasterizer as a flexible tool for sampling. In the second set of applications, we render a scene from multiple viewpoints simultaneously, in order to achieve effects such as glossy reflections/refractions, soft shadows, and multi-view rendering.

1 Introduction

The two-dimensional rasterizer in a graphics processor is a highly optimized fixed-function unit. Conceptually, it is just a loop over samples bounded by a geometric triangle, with the capability to execute a program for each covered sample/pixel. Besides using it for rendering three-dimensional geometry, the rasterizer has been employed as a tool for a wide variety of topics, including, e.g., shadow algorithms [1, 2], constructive solid geometry [3], Voronoi diagrams [4], collision detection [5], caustics [6], ray tracing for global illumination [7], curved surfaces [8], and more.

We note that *higher-dimensional rasterization* is currently a very active research topic, and many efficient algorithms are emerging [9, 10, 11, 12, 13], as well as hardware studies [14]. The conventional usage area is correct visibility for motion blur and depth of field. The assumption in this paper is that there will be an efficient fixed-function 5D rasterizer, including efficient shading [15], in future graphics processors. Our hypothesis is that this new machinery can be used for many other purposes beyond motion/defocus blur. We have identified six such unconventional uses, of which some are dependent on shading and some only utilize time, lens or depth bounds produced by the rasterizer.

The applications can broadly be divided into two categories. The first set of algorithms exploit the fact that a moving and/or defocused triangle *sweeps out a volume* in three to five-dimensional space. This capability opens up for a number of less intuitive use cases. We show examples in *i*) continuous collision detection, *ii*) caustic rendering, and *iii*) higher-dimensional sampling. The stochastic rasterizer can also be used to simultaneously render a scene from *many different viewpoints*, through appropriate choice of lens, focus plane, samples, and use of the time dimension. The benefits are increased shader/texture coherence and fewer geometry passes. To illustrate this, we focus on *iv*) glossy reflections/refractions, *v*) motion blurred soft shadow mapping, and *vi*) stereoscopic and multi-view rendering with motion and defocus blur. Some screenshots are shown in Figure 1.

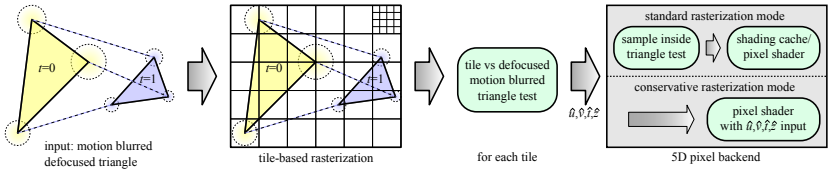


Figure 2: An overview illustration of our five-dimensional rasterization pipeline. The input is defocused, motion blurred triangles, rasterized in tile order. For each tile, a triangle-against-tile test is performed, and only tiles with non-empty intervals $(\hat{u},\hat{v},\hat{t})$ continue to the next stage. In the backend (top right), sample-inside-triangle testing is done, and surviving samples shaded. In conservative rasterization mode (bottom right), a tile size of 1×1 pixels is used, and the sample-inside-triangle test is bypassed. In this case, the pixel shader is executed once per tile and receives the tile intervals as inputs.

Based on the insights gained from these examples, we present a complete model for an efficient and flexible stochastic rasterization pipeline, including novel additions such as conservative depth computations. Our work is rather long term and speculative research in the sense that it assumes the existence of something that is not (yet) readily available. We provide a first set of algorithms for each topic, but we want to emphasize that we do not explore these fully. The core contributions are that our work:

1. Sketches out a space of new applications for higher-dimensional rasterization.

2. Presents the first coherent model for efficient stochastic rasterization, including some novel additions.

3. Highlights many practical design aspects, e.g., the importance of conservative rasterization and flexible sampling.

4. Provides motivation for further research in stochastic rasterization and its uses, as well as hardware and APIs.

We will start by describing our pipeline, and then present the specific novel use cases that have motivated our design.

2 Our Five-Dimensional Rasterization Pipeline

In this section, we describe the 5D stochastic rasterization pipeline that is used throughout this paper. An illustration of our pipeline can be found in Figure 2. The input is a triangle with defocused vertices at $t=0$ and at $t=1$ in order to handle both motion blur and depth of field. It is assumed that the vertices move linearly

in world space, and we assume the thin lens model for depth of field rasterization, similar to other work [9, 11, 12]. Furthermore, our pipeline can efficiently disable depth of field and only render motion blur, and vice versa. This is useful in several applications, e.g., continuous collision detection and caustics (3D), and for glossy effects (4D), see Sections 3, 4, 6.

Each pixel is assumed to have n samples, where each sample has a fixed screen space position, (x,y), fixed lens parameters, (u,v), and a time, t. Although most applications need well-distributed stochastic samples, which may be procedurally generated [16], we have found several cases where other sampling patterns are beneficial or even required. This includes glossy rendering, soft shadows, and stereoscopic or multi-view graphics (Sections 6, 7, 8). We thus propose the hardware uses a reasonably large programmable sample table, coupled with a scrambling method [17] for increased randomness when desired.

Our traversal order is tile-based since such a rasterization order has been shown to be beneficial in several different aspects [10, 11, 12, 13]. This includes reduced memory bandwidth usage to buffers and textures, efficient handling of widely varying triangle sizes, and the possibility of efficient depth buffer compression [18]. In addition, primitive setup is only done once, and tile tests allow for arbitrary sample positions, which makes for higher quality samples. Consequently, as current graphics processors already employ a tile-based traversal order, the transition from two to five dimensions does not require a major pipeline re-design in this respect.

The output of a tile test is a set of intervals, $(\hat{u},\hat{v},\hat{t})$, where an interval is described as $\hat{t} = [\underline{t},\bar{t}]$, which is all t such that $\underline{t} \leq t \leq \bar{t}$. These intervals are conservative estimates of the samples that may overlap with the triangle being rendered. Details on how to compute them in 5D can be found in previous work [11, 13]. In some cases, e.g., caustics rendering (Section 4), we have found tighter bounds for the time interval [19] to be beneficial, so the exact implementation remains to be decided. In addition, we support optional, axis-aligned user clip planes, which can be specified to further limit the extents in *uvt* of a triangle. Each clip plane is given as a lower or upper boundary, e.g., $\bar{t}_{\text{clip}}$, and min/max operations are used to find the final set of sampling intervals. This is a minor extension, but it allows more flexible use of the rasterizer as a sampling engine (Section 5), and it is consistent with current APIs that support clip planes in x,y. We also introduce a way to compute a conservative interval, $\hat{z}$, of the depth over the tile in order to perform $z_{\max}$-culling [20] and $z_{\min}$-culling [21]. See Appendix A. The depth interval is also directly used in several applications, including 5D occlusion queries and continuous collision detection (Section 3).

As described in Figure 2, only tiles where all of the intervals, $\hat{u},\hat{v},\hat{t}$, are non-empty continue down the pipeline, which is common practice. In the *standard* mode (top right in the figure), each sample within those intervals is inside-tested against the triangle, and surviving samples shaded. Depending on the application, we use a shading memoization cache [15], hereafter referred to as a shading cache, or execute the pixel shader per sample. The shading cache is generally preferable,

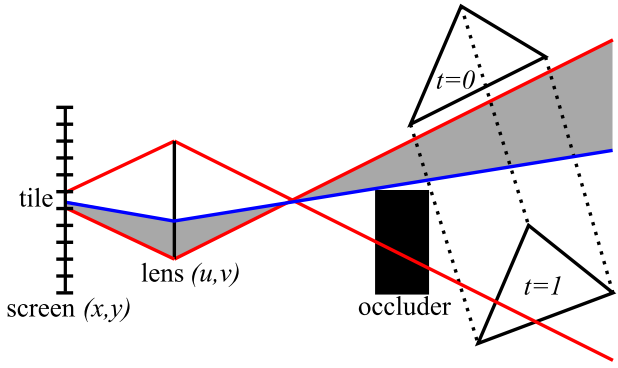


Figure 3: The gray area is the visible (i.e., not occluded) region in world space seen from a particular tile, where samples will pass the depth test and report the triangle as visible.

but some use cases like caustics rendering and sampling (Sections 4, 5) require a 1:1 mapping between samples and shader executions.

By considering a large set of potential use cases, we have found that it is often extremely valuable to be able to *disable* sample testing altogether, and directly feed the intervals $(\hat{u},\hat{v},\hat{t},\hat{z})$ output by the tile test to the pixel shader. In this case, the pipeline behaves as a *conservative* rasterizer at the granularity of the tile size [22] in five dimensions, and the pixel shader is executed per tile. We show examples of using this for 5D occlusion queries, collision detection, caustics, and sampling (Sections 3, 4, 5).

In the following sections, we will present each of our example applications for five-dimensional rasterization in more detail, in order to motivate the above design choices.

3 Five-Dimensional Occlusion Queries

Here, we generalize standard occlusion queries to five dimensions, which can be used for motion blurred and defocused occlusion culling. In addition, we also show that a time-dependent occlusion query (no defocus) can be used for continuous collision detection. Most contemporary graphics processors and 3D APIs support a feature called *occlusion query* [23]. An occlusion query (OQ) can be used to find out how many fragments, n_f, generated by a set of polygons pass the depth test. For example, the faces of a bounding box around a complex character can be used as an occlusion query, and if n_f is zero then the character is occluded with respect to the contents of the depth buffer.

A *sample-based* way to extend occlusion queries to account for motion and defocus blur, is to render the object to the depth buffer using the samples of the 5D rasterizer and count the number of fragments that pass the test, i.e., in the tradi-

tional sense. Figure 3 shows a moving triangle that is in part occluded by other geometry during the shutter interval. The gray area represents the time and lens intervals during which the moving triangle is visible for a particular tile. In a sampled approach, samples in the gray area will pass the depth test and report the triangle as visible.

For motion blur and defocus blur, a relatively large number of samples per pixel (e.g., $16-64$) may be needed for sufficient occlusion query precision. Each occlusion sample requires additional computation and bandwidth, but in many cases, the number of fragments that pass is not needed. We therefore propose to use a five-dimensional *interval-based* occlusion query, which reduces these requirements substantially. An interval-based occlusion query is similar to motion blur z-culling techniques [24, 25], where $z_{\min}/z_{\max}$-values for tiles are stored in a fast memory. A tile may also have several $z_{\min}/z_{\max}$-values for different time intervals. In a 5D setting, this concept is extended to the u and v lens parameters, i.e., $z_{\min}/z_{\max}$-values are stored for disjoint boxes covering the entire uvt-space for the tile. Consequently, the occlusion rate is dependent on the number of $z_{\max}$-values per tile and also the tile size. With more subdivisions in time, efficiency for moving occluders increases. Thus, as a complement to sample-based OQs, the user can trade off occlusion rate for OQ speed.

When performing the query, the rasterizer provides intervals, $\hat{u}$, $\hat{v}$, $\hat{t}$, and $\hat{z}$, for each primitive and visited tile. The $\hat{u}$, $\hat{v}$, and $\hat{t}$ intervals are used to decide which boxes in uvt-space need to be considered when doing the depth comparisons, while the $\hat{z}$ intervals (conservative, not generating any false positives) are used in the actual comparisons. This means that individual samples for each pixel in a tile need not be touched, i.e., no expensive sample-inside-triangle tests or depth computation have to be performed. As soon as one comparison fails, the geometry is not fully occluded, and further rasterization and testing can be aborted.

Continuous Collision Detection Collision detection (CD) algorithms can be broadly divided into *discrete* and *continuous* methods [26]. A drawback of discrete methods is that they may miss collisions involving fast-moving objects, e.g., a bullet shot through a thin paper. Continuous collision detection (CCD) algorithms avoid these problems, sometimes called "tunneling" effects, by computing the first time of contact between objects.

To accelerate CD, many image-based approaches running on the GPU have been proposed, starting with Shinya and Forgue's work [27]. Govindaraju et al. [5, 28] presented an algorithm which computes the *potentially colliding set* (PCS) for a set of objects. Pairs of objects not in the PCS will definitely not collide. Below, we show how to compute the PCS for *CCD* using a three-dimensional rasterizer to determine whether two objects do not collide in a certain period of time. A generalization to many objects follow the lines of previous work [5].

We have chosen to describe our approach in relation to Govindaraju et al.'s work, which works as follows. To detect whether two objects do not (conservatively)

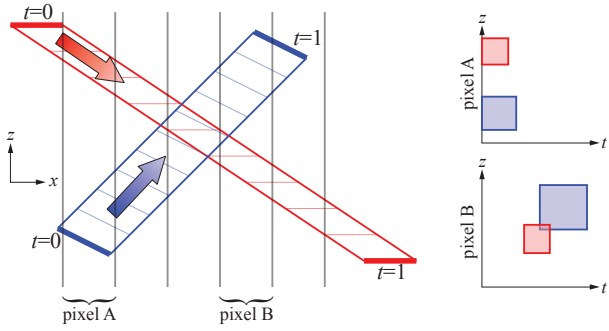


Figure 4: A two-dimensional example of continuous collision detection between two lines. The red (moving down) and blue (moving up) lines move from $t=0$ to $t=1$. For pixels A and B, the conservative bounds for time, t, and depth, z, are visualized to the right. The lines can never overlap in pixel A, since the red and blue boxes do not overlap in the tz-plane. For pixel B, the lines do overlap.

collide at a certain instant of time, the first object is rendered to the depth buffer. In a second pass, the depth test is reversed, and the second object is rendered with an occlusion query. If no fragments pass, then there is no collision between the two objects. We call this an overlap test. This is done orthographically in the xy, xz, and yz planes, and also from opposite directions, which sums to six overlap tests. If at least one of these tests indicates that there is no collision, then the pair does not belong to the PCS. As motion is introduced, then the sample-based OQ, described above, can replace the "static" OQ, but a conservatively correct PCS is not computed when sampling at discrete times.

Instead, we propose a substantially different approach for CCD, based on interval-based occlusion queries, which use a conservative 3D rasterizer. We start with the case of two triangles. For each pixel and triangle, where a 1×1 tile test indicates overlap, we insert the time and depth intervals ($\hat{t}=[\underline{t},\overline{t}]$, and $\hat{z}=[\underline{z},\overline{z}]$), into a per-pixel buffer, e.g., let RGBA $=[\underline{t},\underline{z},\overline{t},\overline{z}]$ be an axis-aligned bounding box in tz. If the tz-boxes of two triangles overlap, they potentially collide in that pixel. This is illustrated in Figure 4.

To handle many triangles per object, we initialize each pixel to represent an empty bounding box. Then, for each triangle in the first object, the union of the current bounding box and the incoming fragment's box is computed using min/max blending. For the second object, a similar buffer is generated. When both buffers have been created, a box vs box overlap test is performed for pixels with the same xy-coordinates (Figure 5, left). For a particular pixel, no collision can occur if the boxes do not overlap. Alternatively, when the second object is rasterized, each tz-interval can be directly tested for overlap with the first object's bounding box (Figure 5, right).

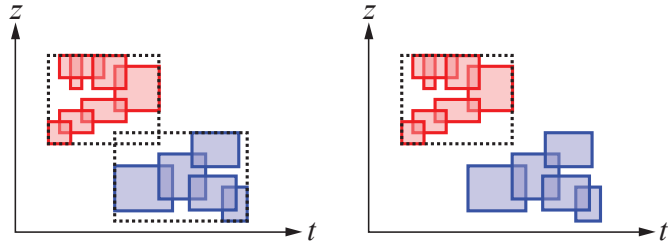


Figure 5: Collision detection in a single pixel between an object whose tz-fragments are red, and one object whose fragments are blue. Left: bounding boxes of the union of all tz-fragments for the red and blue objects are accumulated, and overlap tested. Right: alternatively, test blue tz-fragments individually for overlap against the bounding box of all the red fragments. This would be more efficient in this example, since it would not detect any overlap.

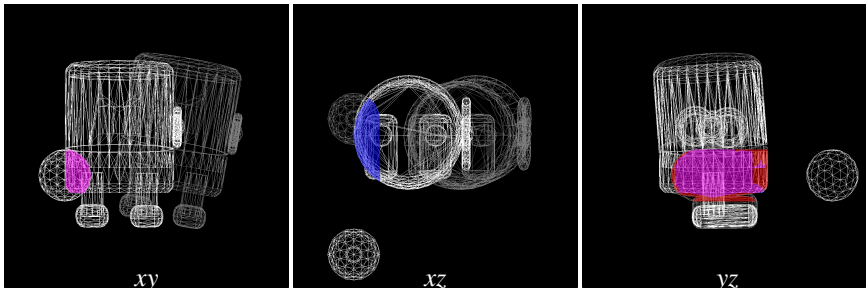


Figure 6: From left to right: orthographic xy, xz, and yz visualizations of pixels overlapping in time and in depth. For each pixel that overlaps in depth, the pixel's blue channel is set, and if there is an overlap in time, the red channel is set. Hence, only purple pixels overlap in time and in depth. Current algorithms only use overlap in depth, while we use both time and depth. In this case, we detect that the objects do not overlap in the xz projection (middle image), while previous algorithms would put them in the potentially colliding set, since the two objects overlap in depth in all views.

We have implemented our algorithm in a simulated rasterization pipeline. An example is shown to the left in Figure 1, where a "toaster" is dodging an approaching soccer ball. For CCD, we use orthographic 3D rasterization in the xy, xz, and yz planes, i.e., we use only three passes compared to six passes using previous algorithms [5]. We visualize the pixels that overlap in depth and in time in Figure 6. As can be seen, our method detects that these two objects do *not* overlap, as opposed to traditional techniques using overlap in depth only.

To summarize, for many applications, continuous (as opposed to discrete) CD is indeed required. To that end, we argue that the method of Govindaraju et al. does

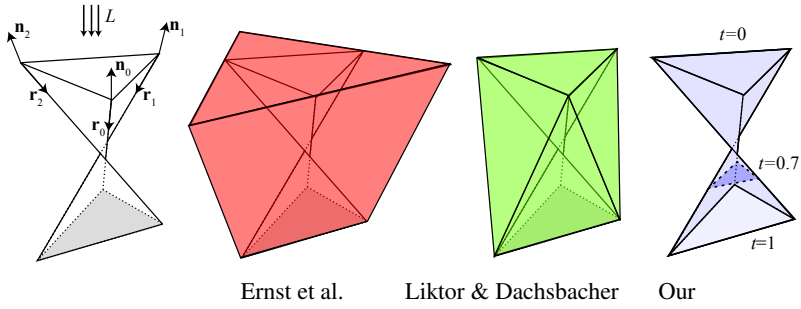


Figure 7: The caustic volume from a generator triangle, where the incoming light L is refracted based on the normals $\mathbf{n}$ along directions $\mathbf{r}$ is shown to the left. Then follows three methods for bounding the volume. Note that the rasterized areas differ substantially.

not take time into account and adds all pairs with overlapping trajectories to the PCS (and incur higher cost). Our contributions are (i) an improved z-interval estimate (appendix), (ii) a first description of continuous CD for triangle soups on the GPU, and (iii) an inexpensive detection test (2D AABB).

We believe that occlusion queries for higher dimensional rasterization will prove to be very useful, and we foresee a wide range of uses. This includes, for example, modifications to hierarchical occlusion culling [29] with occlusion queries so that motion blur, depth of field, and their combination can be even more efficiently rendered. For time-continuous collision detection, there are many avenues for further research. This includes self-collision, finding the time of contact, and different hierarchical strategies for improved performance.

4 Caustics Rendering

Caustics are the formation of focused patterns from light refracted and reflected in curved surfaces or volumes. These patterns add realism to a scene, e.g., the beautiful, shimmering light on the bottom of a swimming pool. The effect can be created by simulating the traversal of photons, calculating the local light density at the receiving surface. Physically correct methods [30], which inherently handle caustics as well as other phenomena, are still too costly for real-time use. Using graphics hardware, several approaches that trace light beams have been proposed [31, 32, 6, 33], as well as splatting-based methods [34].

In beam tracing techniques, *generator* triangles represent the origin of caustic beams, e.g., an ocean surface. The beams are created by letting incoming light be refracted or reflected in the generator triangles. Ernst et al.'s algorithm [6], later refined by Liktor and Dachsbacher [33], generates surface or volume caustics by intersecting the caustic beams with receiving surfaces or eye rays, respectively. In

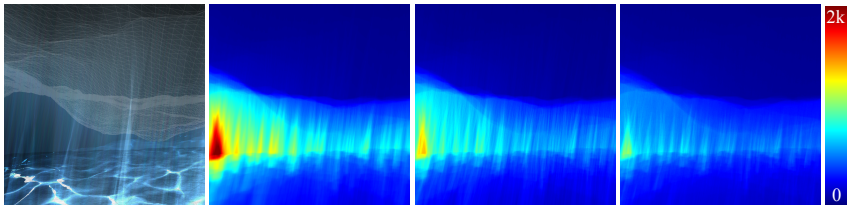


Figure 8: A visualization of the number of pixel shader executions relative to the method by Ernst et al. for a volume caustics example. Dark red represents 2,200 pixel shader executions. We use a motion blur rasterizer to render the caustic volumes as moving triangles, which results in less fill-rate than the two bounding volume methods.

dynamic scenes, the reflected/refracted rays and the set of caustic beams are generated in each frame. The generated caustics volumes are then rasterized, similar to shadow volume rendering, and for each covered pixel, the light contribution of each volume is computed. A tight enclosure of each beam is essential to decrease its screen space area and reduce the fill rate when accumulating the light intensity. As the sides of the caustic beams describe bilinear patches, such fitting is not clear-cut. Ernst et al. use a bounding prism approach with touching planes, while Liktor and Dachsbacher refine this for heavily warped volumes.

A motion blur rasterizer can in this setting be used to model the caustic volume. Instead of bounding the bilinear patch sides of the caustic beam with a volume (see Figure 7), the moving triangle is directly rasterized with the generator triangle at $t = 0$, and the end triangle at $t = 1$ (with backface culling disabled). The pixel shader is invoked for pixels covered by the moving triangle at *any* time, $t \in [0,1]$. To find this overlap, and to reduce the fill-rate as much as possible, we compute the exact time interval when the pixel is inside all three moving triangle edges [19].

With our approach, a bounding volume is not needed and the resulting fill rate is, in general, lower. Figure 7 shows a warped triangle, and compares the screen space coverage of the bounding volume methods with the area of the motion blurred triangle. In Figure 8, the three approaches are compared in an ocean scene with 130,000 caustic volumes. Here, our approach has lower fill-rate than the other two, and avoids bounding volumes altogether.

The modeling of beams as moving triangles may enable the use of rasterizer output in various other algorithms, e.g., to simplify pixel shaders. It would be interesting to explore this concept in a broader sense. For instance, for volumetric caustics, modeling of light attenuation through scattering and absorption could possibly be approximated by the rasterizer's depth interval; $\hat{z} = [\underline{z}, \overline{z}]$, i.e., $\overline{z} - \underline{z}$ can be used as an approximation to the distance the eye ray travels in the beam. For surface caustics, $\hat{z}$ may be used to cull parts of the beams that are completely occluded, and parts of the beams that are definitely in front of the geometry in the depth buffer

for the current tile. By doing so, it is expected that the majority of expensive pixel-in-volume tests can be avoided.

5 Sampling

Many applications in computer graphics, scientific and medical visualization, and engineering need to draw random samples over an arbitrary domain. Examples in graphics include rendering, texture synthesis, and object placement [35]. A standard approach is to divide the sampling domain into simpler shapes, pick a shape at random with the correct probability, and place a sample in it. Although not trivial to parallelize, several papers have shown it can be done in general GPGPU code [36, 37, 38].

Our core insight is that a higher-dimensional rasterizer performs many similar operations, including scheduling/dispatching of the work, random selection, and rejection sampling against complex shapes in 3D–5D. Rasterization also naturally decouples the sample placement from the choice of tessellation, i.e., a sample that misses one primitive falls into an adjacent, which is an added benefit compared to methods that sample shapes independently. All this ends up being a fair amount of code otherwise. The underlying assumption is that a hardware rasterizer would be more power-efficient, even if the algorithms have to be tweaked a bit. Naturally, this is currently hard to prove, so we will focus on sketching the idea and a few applications.

The 5D Rasterizer as a Volumetric Sampler The stochastic rasterization process is usually illustrated in clip space by moving/shearing a triangle's vertices, but it can equivalently be seen as the triangle carving out a volumetric shape, $\mathscr{S}$, in 5D *xyuvt*-space. This domain is filled with uniformly distributed samples, and the rasterizer quickly finds which ones are inside $\mathscr{S}$ through tile tests as described in Section 2. The analogy in 2D is a triangle in screen space, which cuts out a set of uniform samples in *xy*. The volume of $\mathscr{S}$ directly controls the *expected* number of samples, N, placed in it, i.e., $E[N] = \rho V(\mathscr{S})$, where ρ is the sampling density.

In 3D *xyt*-space, $\mathscr{S}$ is a generalized triangular prism with the triangular end caps at $t = 0$ and $t = 1$ (or a tighter range if user clip planes in t are used). Note that due to varying per-vertex motion and perspective foreshortening, the edges connecting the end caps may be *curved* and the sides are usually non-planar in *xyt*. This is non-intuitive, as the edges are always straight lines in clip space, although the sides may be bilinear patches [24]. When one extra dimension, u, is added, each vertex is sheared in x as u varies. The shear may be non-linear as it is a function of depth, which is time-dependent. The carved out hypervolume has 12 vertices, with end caps being generalized triangular prisms in 3D. Finally, in 5D, $\mathscr{S}$ is a complex shape with 24 vertices.

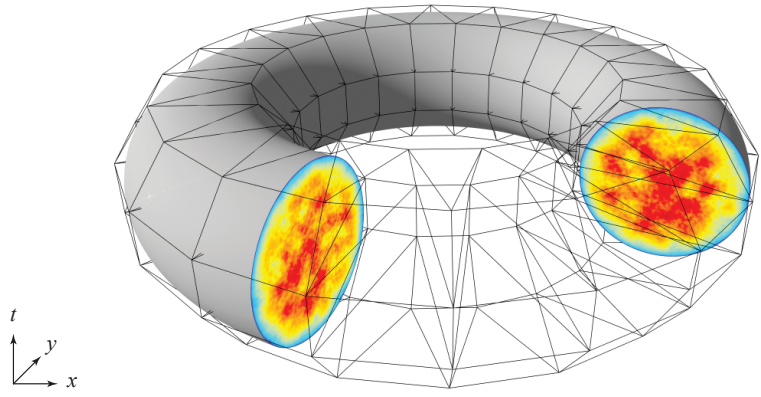


Figure 9: The stochastic rasterizer can be used as a flexible sample generator in, e.g., numerical integration. In this example, generalized triangular prisms conservatively bound the integration domain (here a torus). The prisms are rasterized as motion blurred triangles in xyt-space (note the t-axis pointing upwards). At each sample, the integrand is evaluated and accumulated in the pixel shader. Similar strategies apply to other sampling problems.

To sample an arbitrary domain, $\mathscr{D}$, we first construct a conservative bounding volume, $\mathscr{B}$, so that $\mathscr{D} \subseteq \mathscr{B}$. The bounding volume is then tessellated into a number of non-overlapping, adjacent primitives, $\mathscr{S}$, which are individually rasterized. In 2D, this corresponds to tessellating the interior of an arbitrary bounding polygon into triangles. Due to rasterization tie-breaking rules, any sample is guaranteed to be placed in at most one primitive. Finally, the pixel shader performs an analytical test per sample (in $\mathscr{B}$), to reject any remaining samples outside of $\mathscr{D}$. Figure 9 shows an illustrative example in 3D. The exact sample placement is given by the built-in low-discrepancy sample generator (Section 2). Alternatively, the conservative mode can be used and any number of random samples generated over the bounds $\hat{u}\hat{v}\hat{t}$. In this case, the samples have to be manually tested against $\mathscr{S} \cap \mathscr{D}$, not just $\mathscr{D}$, since the bounds may overlap. Collectively, the result is a uniform random sampling of $\mathscr{D}$; the hardware rasterizer performs an initial fast, but coarse sample culling, and the pixel shader performs a final fine-grained test (which can be skipped if $\mathscr{B} = \mathscr{D}$).

To make $\mathscr{S}$ easier to work with, we can restrict the vertex locations in such a way that all sides of the primitive are planar. For example, in the 3D case, if we restrict each vertex to not move in depth, and each pair of edges in the triangular end caps of the prism to be parallel, all edges will be straight and all sides planar. The result is a tapered triangular prism. In general, we can place the vertices so that $\mathscr{S}$ is an n-polytope (a geometric object with flat sides in n dimensions), where $n \leq 5$. The class of polytopes generated by the stochastic rasterizer in 3D, 4D, and 5D,

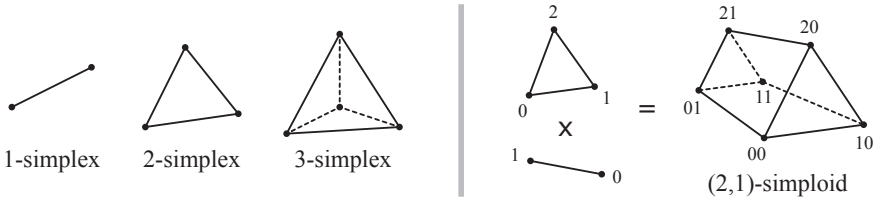


Figure 10: The n-simplex is the simplest possible n-dimensional polytope, consisting of n+1 vertices, with all pairs connected by edges. The product of m simplices makes an $(n_1,\ldots,n_m)$-simploid in $\sum n_i$ dimensions. The figure shows simplices in up to three dimensions, and an $(2,1)$-simploid, i.e., triangular prism.

will formally be $(2,1)$, $(2,1,1)$, and $(2,1,1,1)$-simploids [39], respectively. See Figure 10 for examples.

Accelerated Dart Throwing Poisson-disk sampling using dart throwing [40] is one of the intriguing use cases for our framework. The resulting blue noise samples are ideal for many applications, e.g., stippling and texture synthesis. In its basic form, dart throwing generates a large number of random candidate points over $\mathscr{D}$, but only keeps the ones that are separated by a certain minimum distance. Modern algorithms accelerate the process by tracking the voids, V, between samples using, e.g., octree cells or general polytopes [36, 38]. Conceptually, the sampling domain, $\mathscr{D}$, is first subdivided into voids, which are put in an "active" list, and the following operations are performed:

1. Select a void, V, from the active list with probability according to its volume.
2. Choose a random candidate point, p, in the void.
3. Check if p meets the minimum distance criteria for the neighboring points, and if so, add it to the point set.
4. Check if V is completely covered, and if not, split it into smaller voids that are added to the active list.

With a stochastic rasterizer, we can perform steps (1) and (2) on a large number of voids in parallel. Each void is represented as a single (or union of) simploids compatible with the rasterizer. Figure 11 shows some examples. The expected number of candidate points in each void is controlled by uniformly scaling it, and the voids are independently and randomly displaced to avoid bias. All voids in the active list are then rasterized with the depth test disabled, and the generated candidate points stored to a linear array (e.g., using an append buffer). A compute pass finally processes the points to eliminate conflicts, and updates the active list. When the active list is empty, a *maximal* distribution has been achieved, i.e., no more points can be inserted.

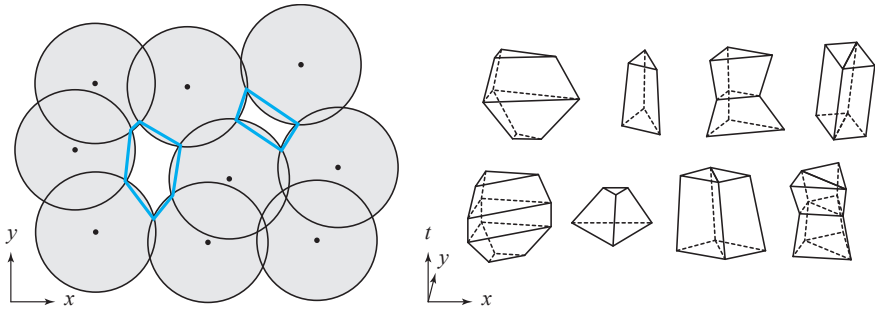


Figure 11: Optimized dart throwing keeps track of the voids, i.e., unsampled regions, in the sampling domain to guide the insertion of new points. We represent voids as polygons/polytopes, as shown in blue on the left. Candidate points are generated in parallel by stochastically rasterizing all voids, after random displacement and appropriate scaling. Some examples in 3D are shown on the right.

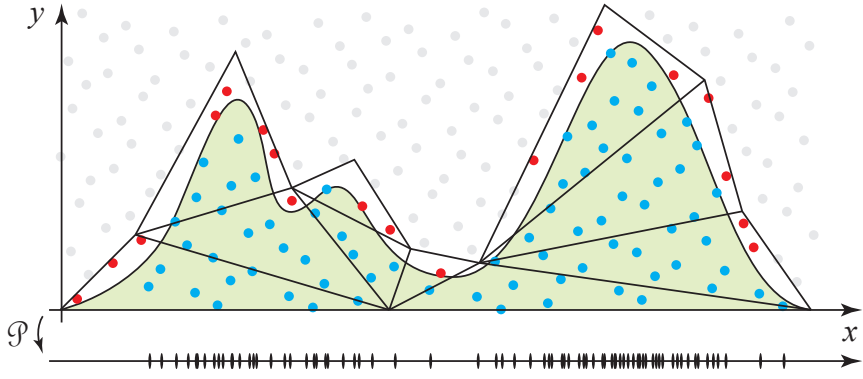


Figure 12: Non-uniform samples can be created by sampling the region under the density function (shown in green), $\rho(\mathbf{x})$, uniformly and projecting the generated samples on $\mathbf{x}$. The rasterizer quickly rejects all samples (gray) outside the rasterized triangles, and in the pixel shader, the remaining samples are tested against ρ to remove outliers (red). The same technique applies in higher dimensions.

Adaptive White Noise Non-uniformly distributed samples in n dimensions can be achieved by sampling uniformly over an appropriate domain in n+1 dimensions, and orthographically projecting the samples back to n dimensions. Intuitively, the shape of the sampling domain in $\mathbb{R}^{n+1}$ is defined by viewing the density function, $\rho(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^n$, as a height field. By generating samples under the height field $(\mathbf{x}, \rho(\mathbf{x})) \in \mathbb{R}^{n+1}$, and projecting on $\mathbf{x}$, we effectively get samples distributed according to $\rho(\mathbf{x})$.

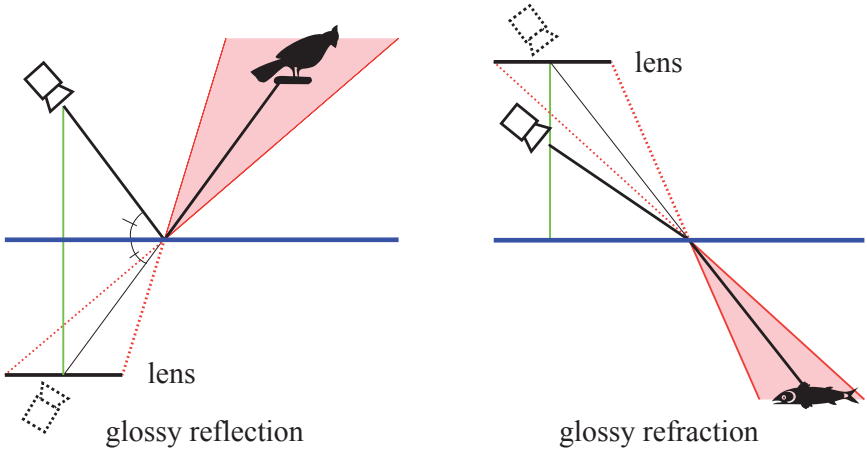


Figure 13: The dotted cameras in the figure are used for rendering the reflection map (left) and the refraction map (right), respectively. The green lines indicate the direction in which the primary camera is translated to find these positions. The frustums that our method sample using 4D rasterization are shown in red for a single texel. Note that the reflecting/refracting plane (blue line) becomes both the focus plane and the near plane.

This is not fundamentally new, but the stochastic rasterizer gives us an efficient way of sampling the height field $(\mathbf{x}, \rho(\mathbf{x}))$ in up to five dimensions. This allows non-uniform sampling in up to 4D. Figure 12 shows a simple example of a 1D density function, which is lifted to 2D, bounded and tessellated, and sampled by rasterizing the resulting triangles. Note that samples generated using this method have white noise characteristics, as due to the projection, it is difficult to ensure a minimum point distance (i.e., blue noise).

6 Planar Glossy Reflections and Refractions

To render glossy effects, it is necessary to sample the scene in multiple directions from the surface point being shaded. For planar surfaces, this can be implemented as a two-dimensional shear of the reflected/refracted geometry. This has been exploited in previous multi-pass approaches based on accumulation buffering [41], or stochastic motion blur rasterization [24]. Our solution is instead to use the 4D rasterizer to perform the entire 2D shear in a single pass.

For each frame, our method first generates a reflection/refraction map. The map is stochastically rendered with correct occlusion into a multisampled render target and then pre-filtered into a 2D texture, which is used as texture when the scene is rendered from the original viewpoint. For reflections, we set the camera at the reflected position of the primary camera, as illustrated in Figure 13 (left). Alter-

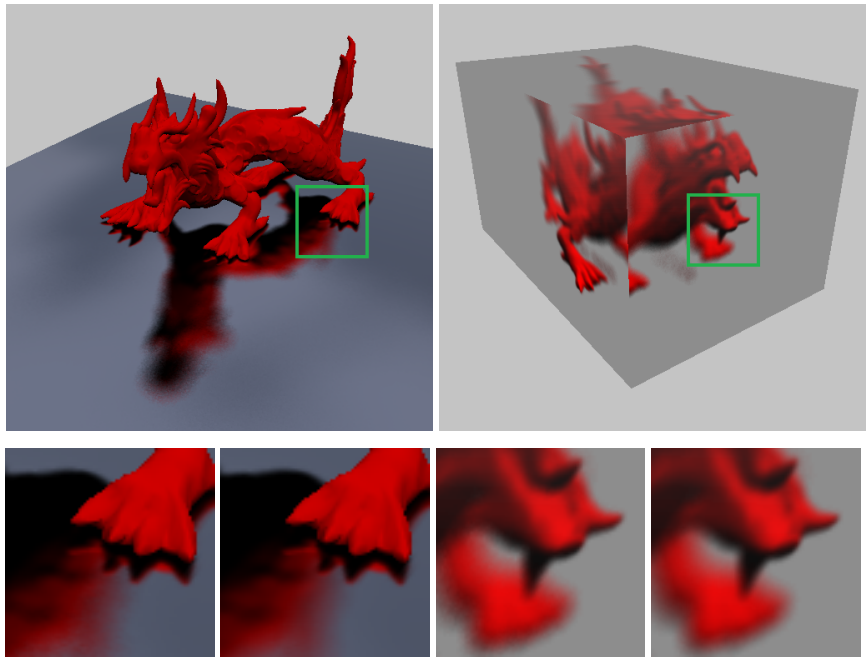


Figure 14: Two renderings with 16 samples per texel for the reflection/refraction maps, using a simple box resolve filter. Left: glossy reflection. Right: glossy refraction. The bottom row compares images with 16 and 64 samples per texel.

natively, to compute a refraction map, we choose a single ray from the primary camera and refract it correctly. We then translate the camera, along the refraction plane normal, to intersect this ray. In our implementation, the central ray of the primary camera is used (right in Figure 13).

Our way of translating the camera results in an approximate index of refraction that is correct at the chosen central ray. However, it is progressively less accurate, but still plausible, toward the edges of the image. For both reflection and refraction, the viewport is chosen to tightly enclose the glossy surface, which is also set as the focus plane. The center of the lens is located at the reflection or refraction camera position as described above, and the lens plane is parallel to the reflection or refraction plane. This gives us a camera setup with an off-center viewport, which can be used with four-dimensional rasterization.

In our implementation, we use a uniform, stratified sampling of a circular camera lens. This results in a visually plausible BRDF, and although it is not physically based, it is possible to control the roughness of the surface by setting the size of the lens. For future work, it would be interesting to examine different BRDFs, either by adjusting the sample distribution or by calculating a weight for each sample. The multi-sampled render target is finally resolved into a texture and no specialized filter is required when sampling. Our reflection and refraction maps

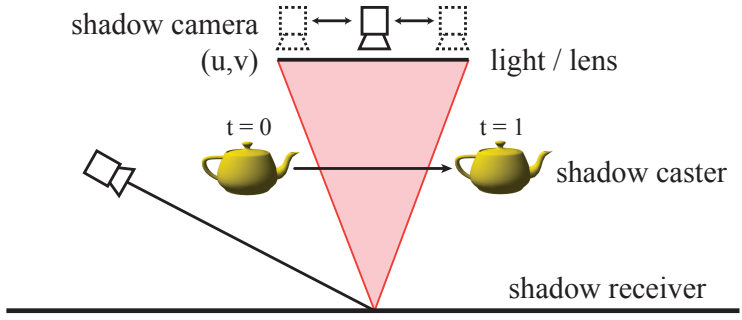


Figure 15: The camera setup used to create a 5D shadow map. The (u,v) parameters position the shadow camera over the area of the light source, and t is used for object motion. For simplicity, we set the focus plane to infinity.

are generated in such a way that they map directly to the glossy surface and can, if desired, be weighted together according to the Fresnel term when rendering the surface. Examples can be seen in Figure 14.

7 Motion Blurred Soft Shadow Mapping

We show that a five-dimensional stochastic rasterizer can be set up to generate motion blurred soft shadows through a variation of the shadow mapping algorithm [2]. We note that the (u,v) sample parameters may be used to stochastically vary the sample position on an area light source, as shown in Figure 15. Furthermore, the time parameter may be used to interpolate object motion similar to what is described by Akenine-Möller et al. [24]. Using the stochastic rasterizer, we can thus create a 5D visibility field as seen from an area light source, i.e., a 5D shadow map. This gives us visibility data similar to what is used by Lehtinen et al. [42], which needs to be filtered to give a low-noise estimate of visibility for every screen pixel.

As lookups in a 5D data set are inherently difficult (offline algorithms often use 5D kD-trees, for example), we generate a 5D shadow map restricted to a small set of fixed (u,v,t)-samples. This gives results equivalent to accumulation buffering for soft shadows [43], but our 5D shadow map is generated in a *single* render pass. The shadow map is queried in the subsequent shading pass, which uses motion blur rasterization with the same set of t parameters. With this setup, the shadow map lookups can be made very efficient. The data structure is an array of 2D shadow maps, which is indexed by t. Thus, in constant time, each sample's t parameter determines which shadow map and corresponding projection matrix to use, and we perform a traditional shadow map lookup using these. This implies that the 5D shadow map needs to be queried per sample for correct motion blurred shadows, as the shadow map lookup depends on the parameters of the samples seen from the camera. An example of soft shadows with motion blur is shown in Figure 1,

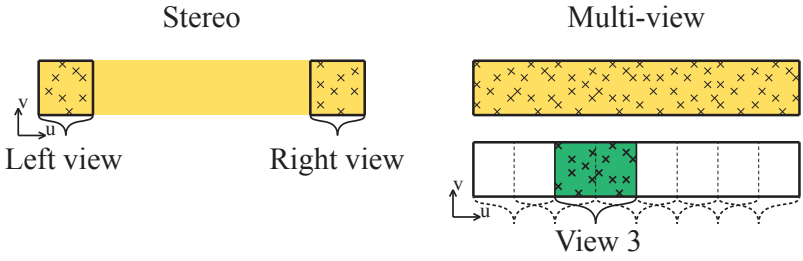


Figure 16: Sampling for multi-view and stereoscopic rendering. The yellow areas show the full extent of the lens. The green area shows a single view's part of the lens.

generated with a set of 64 unique (u,v,t) values.

The most interesting venue for future work lies finding hardware-friendly data structures that allow efficient lookups in higher dimensions. This would enable more well-distributed samples and produce noise instead of the banding artifacts associated with accumulation buffering. We note similarities to the work by Lehtinen et al. [42], but much work remains before such filtering approaches can be used in the real-time domain. Another interesting area of future work is lossy compression of visibility data to make 5D shadow lookups more efficient at reasonable loss of quality; for instance transforming the visibility data for more efficient lookup by extending the work of Agrawala et al. [44].

8 Stochastic Multi-View Rasterization

Stereoscopic and multi-view rendering are seeing widespread use due to the increased availability of 3D display technology. Popular usage areas include films, games, and medical applications. Stereo involves rendering separate images for the left/right eyes, while multi-view displays may need between five and 64 different images [45].

We use a stochastic 5D rasterizer to generate multi-view images with motion blur and depth of field in a *single* pass. This extends Hasselgren and Akenine-Möller's [46] work for static multi-view rendering. Their shading cache is directly applicable, but in our pipeline originally targeting motion blur and depth of field, we use a memoization shading cache [15], allowing us to get shader reuse not only between views, but also over time and the entire lens. This is similar to what was suggested by Liktor and Dachsbacher [47], and we believe this will be a very important use case for higher-dimensional rasterization.

The viewpoints are located on a *camera line* with small distances in between. To avoid inter-perspective aliasing [48], it is important to filter over a region of the

Figure 17: Top: a pair of stereoscopic images with motion/defocus blur rendered using our 5D rasterizer with eight samples per view. These images were constructed for parallel-eye viewing, and hence the stereo effect can be seen when the human viewer focuses at infinity. Bottom: multi-view rendering with seven views, each filtered from a subregion of the lens, in this case using overlapping box filters.

camera line when generating a view. With a standard 2D rasterizer, multi-view images with motion blur and depth of field can be generated by rendering a large set of images at different positions on the camera line, lens and time [43] and filter them together. This is impractical for many views, has low shading and texture cache coherence, and requires many geometry passes. In contrast, we render all views stochastically in one pass with very high shading reuse.

If we add depth of field to a multi-view rendering setup, the camera line becomes an oblong region/lens. We sample over this region in a single four-dimensional rasterization pass, as seen in Figure 16. The fifth dimension of the stochastic rasterizer is used for motion blur, as usual. For stereoscopic rendering, the samples are divided into two disjoint regions on the lens, which effectively creates two smaller lenses, as can be seen to the left in Figure 16. The images in Figure 17 were rendered using this approach. For more efficient rasterization, the separate lens regions could share the tile test for the v-axis, but perform separate tests for the u-axis.

We have implemented stereoscopic and multi-view rendering with motion and defocus blur in our pipeline equipped with a shading cache (see Section 2). Rendering the two stereo views in Figure 17 results in just a 10% increase in shader invocations, compared to rendering a single view only, using a shading cache with 1024 entries. For multi-view rendering with seven views, the total increase in pixel shader invocations is just 15% in this example.

9 Conclusion

We have presented a number of future-looking use cases for higher-dimensional rasterization, as well as several improvements to the pipeline itself, which we hope will help guide future hardware and API design. Our inspiration came from the analogy with the traditional rasterization pipeline, which has been used for a wide range of tasks that were not immediately obvious. We have shown that the same will likely hold true for a stochastic rasterization pipeline.

One of our core insights is that the time and lens bounds provided by recent tile-based traversal algorithms have a much wider applicability than just sample culling. Seeing the moving/defocused triangle as a volumetric primitive and utilizing its extents in the different dimensions, opens up for many interesting use cases. Our second set of applications are more straightforward, but they highlight many practical aspects of the design and present a useful model for thinking about stochastic rasterization as a tool for efficiently rendering a scene from many different viewpoints. Besides the presented applications, it may also be possible to use higher-dimensional rasterization in other related topics, such as computer vision and computational photography.

The next step is to further analyze and improve the most promising applications. We have focused on looking broadly at what *can* be done with a higher-dimensional rasterizer, but detailed simulations are necessary to reveal the true winners. Other than that, the most obvious future work would be to implement the full pipeline in hardware. We believe and hope the graphics research community will continue to explore topics along these lines, in which case our research will have truly long term value.

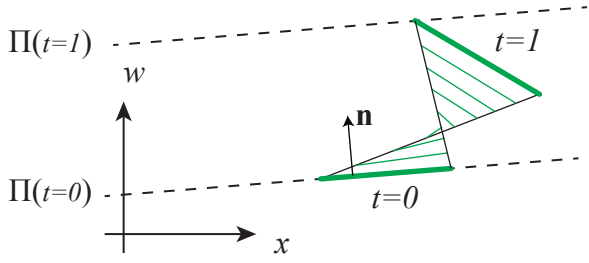


Figure 18: A conservative maximum depth bound from our second test is expressed as a plane equation $\Pi(t)$ that moves linearly in time, which has larger (or equal) depth than the triangle (as seen from the camera) at all times, $t \in [0,1]$.

A Appendix: Conservative Depth Computations

For efficient $z_{\min}/z_{\max}$-culling, it is important to conservatively estimate the min/-max depth of a triangle inside a tile. This is straightforward for static triangles, but considerably harder for motion blurred triangles. The only known method [24] reverts to using min/max of vertices when the triangle changes facing during $t \in [0,1]$, for example. This leads to poor bounds. In this discussion, the vertices at $t = 0$ are denoted $\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$, and $\mathbf{r}_0\mathbf{r}_1\mathbf{r}_2$ at $t = 1$. Our approach is based on two types of tests, which gives tight estimates of minimum and maximum depth over a motion blurred triangle inside a tile. The first test is similar to the moving bounding box around the triangle [10], except that for the max depth computation, we only use the farthest rectangle of the box, and vice versa.

The idea of the second test is to compute a plane equation through the triangle at time $t = 0$ ($t = 1$), and move that linearly in t along the plane's normal direction, such that it always has the moving triangle on one side of the plane. This is illustrated in Figure 18. For example, the moving plane equation for maximum depth (based on the triangle at $t = 0$), can be constructed as follows. The normal, $\mathbf{n}$, of the triangle at $t = 0$ is computed, and negated if $n_w < 0$ in order to ensure that it moves in the right direction. The plane equation is then $\mathbf{n} \cdot \mathbf{x} + d = 0$, where $d = -\mathbf{n} \cdot \mathbf{q}_0$, and $\mathbf{x}$ is any point on the plane. Next, we compute the maximum signed distance from the plane over the vertices of the moving triangle at $t = 1$:

$$v = -\max_i(\mathbf{n} \cdot \mathbf{r}_i + d), \quad i \in \{0,1,2\}. \tag{1}$$

At this point, we have created a moving plane equation, which moves in the direction of the plane normal: $\Pi(t) : \mathbf{n} \cdot \mathbf{x} + d + vt = 0$. As can be seen, we have added the term vt, which ensures (by construction) that the triangle is "below" the moving plane, $\Pi(t)$. If $\Pi(t)$ does not sweep through the camera origin, e.g., $d + vt \neq 0, \forall t \in [0,1]$, then $\Pi(t)$ has greater depth than the moving triangle at all times, $t \in [0,1]$. Similarly, a moving plane equation can be derived from the triangle normal at $t = 1$.

For each moving plane equation, the maximum depth is computed over a tile by evaluating the plane equation at the appropriate tile corner using the time interval, $\hat{t}$, from the tile test. The minimum of these two evaluations and the maximum depth from the first test is used as a conservative maximum depth over the tile. Similar computations are done for evaluating the minimum depth.

This approach can also be extended to handle depth of field. Briefly, the depth estimations need to be done at the four corners of the two-dimensional bounding box of the lens shape, and similar to before, the second test has to be disabled if the plane sweeps over any part of the lens (and not only a single camera point).

Bibliography

[1] F. C. Crow, "Shadow Algorithms for Computer Graphics," in *Computer Graphics (Proceedings of SIGGRAPH 77)*, vol. 11, pp. 242–248, 1977.

[2] L. Williams, "Casting Curved Shadows on Curved Surfaces," in *Computer Graphics (Proceedings of SIGGRAPH 78)*, pp. 270–274, 1978.

[3] T. F. Wiegand, "Interactive Rendering of CSG Models," *Computer Graphics Forum,*, vol. 15, no. 4, pp. 249–261, 1996.

[4] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver, "Fast Computation of Generalized Voronoi Diagrams using Graphics Hardware," in *Proceedings of SIGGRAPH 1999*, pp. 277–286, 1999.

[5] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha, "CULLIDE: Interactive Collision Detection between Complex Models in Large Environments using Graphics Hardware," in *Graphics Hardware*, pp. 25–32, 2003.

[6] M. Ernst, T. Akenine-Möller, and H. W. Jensen, "Interactive Rendering of Caustics using Interpolated Warped Volumes," in *Graphics Interface*, pp. 87–96, 2005.

[7] T. Hachisuka, *High-Quality Global Illumination Rendering Using Rasterization*, pp. 615–634. 2005.

[8] C. Loop and J. Blinn, "Real-Time GPU Rendering of Piecewise Algebraic Surfaces," *ACM Transactions on Graphics,*, vol. 25, no. 3, pp. 664–670, 2006.

[9] 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*, pp. 59–68, 2009.

[10] J. Munkberg, P. Clarberg, J. Hasselgren, R. Toth, M. Sugihara, and T. Akenine-Möller, "Hierarchical Stochasic Motion Blur Rasterization," in *High Performance Graphics*, pp. 107–118, 2011.

[11] S. Laine, T. Aila, T. Karras, and J. Lehtinen, "Clipless Dual-Space Bounds for Faster Stochastic Rasterization," *ACM Transaction on Graphics,*, vol. 30, no. 4, pp. 106:1–106:6, 2011.

[12] T. Akenine-Möller, R. Toth, J. Munkberg, and J. Hasselgren, "Efficient Depth of Field Rasterization using a Tile Test based on Half-Space Culling," *to appear in Computer Graphics Forum*, 2012.

[13] J. Munkberg and T. Akenine-Möller, "Hyperplane Culling for Stochastic Rasterization," in *Eurographics Short Papers Proceedings (to appear)*, 2012.

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

[15] 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.

[16] S. Joe and F. Y. Kuo, "Constructing Sobol' Sequences with Better Two-Dimensional Projections," *SIAM Journal on Scientific Computing,*, vol. 30, no. 5, pp. 2635–2654, 2008.

[17] T. Kollig and A. Keller, "Efficient Multidimensional Sampling," *Computer Graphics Forum,*, vol. 21, no. 3, 2002.

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

[19] C. J. Gribel, M. Doggett, and T. Akenine-Möller, "Analytical Motion Blur Rasterization with Compression," in *High Performance Graphics*, pp. 163–172, 2010.

[20] N. Greene, M. Kass, and G. Miller, "Hierarchical Z-Buffer Visibility," in *Proceedings of SIGGRAPH 1993*, pp. 231–238, 1993.

[21] 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.

[22] T. Akenine-Möller and T. Aila, "Conservative and Tiled Rasterization Using a Modified Triangle Set-Up," *Journal of Graphics Tools,*, vol. 10, no. 3, pp. 1–8, 2005.

[23] M. Segal and K. Akeley, "The OpenGL Graphics System: A Specification, v 4.2," August 2011.

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

[25] 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*, pp. 11–18, 2010.

[26] S. Redon, Abderrahmane, and S. Coquillart, "Fast Continuous Collision Detection between Rigid Bodies," *Computer Graphics Forum,*, vol. 21, no. 3, pp. 279–287, 2002.

[27] M. Shinya and M.-C. Forgue, "Interference Detection through Rasterization," *The Journal of Visualization and Computer Animation,*, vol. 2, no. 4, pp. 132–134, 1991.

[28] N. K. Govindaraju, M. C. Lin, and D. Manocha, "Fast and Reliable Collision Culling Using Graphics Hardware," *IEEE Transactions on Visualization and Computer Graphics,*, vol. 12, no. 2, pp. 143–154, 2006.

[29] 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.

[30] H. W. Jensen and P. Christensen, "Efficient Simulation of Light Transport in Scenes with Participating Media using Photon Maps," in *Proceedings of SIGGRAPH*, pp. 311–320, 1998.

[31] M. Watt, "Light-Water Interaction using Backward Beam Tracing," in *Proceedings of SIGGRAPH 1990*, pp. 377–385, 1990.

[32] K. Iwasaki, Y. Dobashi, and T. Nishita, "An Efficient Method for Rendering Underwater Optical Effects using Graphics Hardware," *Computer Graphics Forum,*, vol. 21, no. 4, pp. 701–712, 2002.

[33] G. Liktor and C. Dachsbacher, "Real-Time Volume Caustics with Adaptive Beam Tracing," in *Symposium on Interactive 3D Graphics and Games*, pp. 47–54, 2011.

[34] M. Shah, J. Konttinen, and S. Pattanaik, "Caustics Mapping: An Image-space Technique for Real-time Caustics," *IEEE Transactions on Visualization and Computer Graphics,*, vol. 13, no. 2, pp. 272–280, 2007.

[35] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2nd ed., 2010.

[36] L.-Y. Wei, "Parallel Poisson Disk Sampling," *ACM Transactions on Graphics,*, vol. 27, no. 3, pp. 20:1–20:9, 2008.

[37] M. N. Gamito and S. C. Maddock, "Accurate Multidimensional Poisson-Disk Sampling," *ACM Transactions on Graphics,*, vol. 29, no. 1, pp. 8:1–8:19, 2009.

[38] M. S. Ebeida, A. A. Davidson, A. Patney, P. M. Knupp, S. A. Mitchell, and J. D. Owens, "Efficient Maximal Poisson-Disk Sampling," *ACM Transactions on Graphics,*, vol. 30, no. 4, pp. 49:1–49:12, 2011.

[39] D. Moore, *Understanding Simploids*, pp. 250–255. Graphics Gems III, 1992.

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

[41] P. J. Diefenbach and N. I. Badler, "Multi-Pass Pipeline Rendering: Realism For Dynamic Environments," in *Symposium on Interactive 3D Graphics*, pp. 59–70, 1997.

[42] 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.

[43] P. Haeberli and K. Akeley, "The Accumulation Buffer: Hardware Support for High-Quality Rendering," in *Computer Graphics (Proceedings of SIGGRAPH)*, pp. 309–318, 1990.

[44] M. Agrawala, R. Ramamoorthi, A. Heirich, and L. Moll, "Efficient Image-Based Methods for Rendering Soft Shadows," in *Proceedings of SIGGRAPH*, pp. 375–384, 2000.

[45] N. A. Dodgson, "Autostereoscopic 3D Displays," *IEEE Computer,*, vol. 38, no. 8, pp. 31–36, 2005.

[46] J. Hasselgren and T. Akenine-Möller, "An Efficient Multi-View Rasterization Architecture," in *Eurographics Symposium on Rendering*, pp. 61–72, 2006.

[47] G. Liktor and C. Dachsbacher, "Decoupled Deferred Shading for Hardware Rasterization," in *Symposium on Interactive 3D Graphics and Games (to appear)*, 2012.

[48] M. Zwicker, W. Matusik, F. Durand, and H. Pfister, "Antialiasing for Automultiscopic 3D Displays," in *Eurographics Symposium on Rendering*, pp. 73–82, 2006.