LUND UNIVERSITY

Faster data structures and graphics hardware techniques for high performance rendering

Ganestam, Per

2016

Link to publication

Citation for published version (APA):
Ganestam, P. (2016). *Faster data structures and graphics hardware techniques for high performance rendering*. [Doctoral Thesis (compilation), Department of Computer Science]. Lund University.

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Faster data structures and graphics hardware techniques for high performance rendering

Per Ganestam
Department of Computer Science
Lund University

ISBN 978-91-7623-655-0 (Printed)
ISBN 978-91-7623-656-7 (Electronic)
ISSN 1404-1219
Dissertation 49, 2016
LU-CS-DISS: 2016-01

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

Email: per.ganestam@cs.lth.se
WWW: http://fileadmin.cs.lth.se/cs/Personal/Per_Ganestam/

Typeset using LaTeX2ε
Printed in Sweden by Tryckeriet i E-huset, Lund, 2016
© 2016 Per Ganestam

Abstract

Computer generated imagery is used in a wide range of disciplines, each with different requirements. As an example, real-time applications such as computer games have completely different restrictions and demands than offline rendering of feature films. A game has to render quickly using only limited resources, yet present visually adequate images. Film and visual effects rendering may not have strict time requirements but are still required to render efficiently utilizing huge render systems with hundreds or even thousands of CPU cores.

In real-time rendering, with limited time and hardware resources, it is always important to produce as high rendering quality as possible given the constraints available. The first paper in this thesis presents an analytical hardware model together with a feed-back system that guarantees the highest level of image quality subject to a limited time budget.

As graphics processing units grow more powerful, power consumption becomes a critical issue. Smaller handheld devices have only a limited source of energy, their battery, and both small devices and high-end hardware are required to minimize energy consumption not to overheat. The second paper presents experiments and analysis which consider power usage across a range of real-time rendering algorithms and shadow algorithms executed on high-end, integrated and handheld hardware.

Computing accurate reflections and refractions effects has long been considered available only in offline rendering where time isn't a constraint. The third paper presents a hybrid approach, utilizing the speed of real-time rendering algorithms and hardware with the quality of offline methods to render high quality reflections and refractions in real-time.

The fourth and fifth paper present improvements in construction time and quality of Bounding Volume Hierarchies (BVH). Building BVHs faster reduces rendering time in offline rendering and brings ray tracing a step closer towards a feasible real-time approach.

Bonsai, presented in the fourth paper, constructs BVHs on CPUs faster than contemporary competing algorithms and produces BVHs of a very high quality.

Following Bonsai, the fifth paper presents an algorithm that refines BVH construction by allowing triangles to be split. Although splitting triangles increases construction time, it generally allows for higher quality BVHs. The fifth paper introduces a triangle splitting BVH construction approach that builds BVHs with quality on a par with an earlier high quality splitting algorithm. However, the method presented in paper five is several times faster in construction time.

Acknowledgements

First of all, I would like to thank my main supervisor Michael Doggett for all the discussions on and off topic and invaluable support and advice throughout my studies, paper writing and thesis work. I would also like to thank my assistant supervisor Tomas Akenine-Möller for his support and insights and for arranging an internship for me at Intel. I would like to thank the other members of the graphics group at Lund University for intriguing discussions and valuable insights, Rasmus Barringer, Björn Johnsson, Magnus Andersson, Carl-Johan Gribel and Pierre Moreau. I would also like to give my gratitude to the members of the Advanced Rendering team at Intel in Lund and abroad, Jacob Munkberg, Petric Clarberg, Jon Hasselgren, Robert Toth and Jim Nilsson.

I would like to thank my family for their support and for withstanding past and future years of scientific ranting.

I owe my deepest gratitude to my wife and my daughter, thank you.

Preface

I. Per Ganestam and Michael Doggett,
"Auto-tuning Interactive Ray Tracing using an Analytical GPU Architecture Model", in *GPGPU-5 Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*,
Pages 94-100, March 2012.

II. Björn Johnsson, Per Ganestam,
Michael Doggett and Tomas Akenine-Möller,
"Power Efficiency for Software Algorithms running on Graphics Processors", in *EGGH-HPG'12 Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, Pages 67-75, June 2012.

III. Per Ganestam and Michael Doggett,
"Real-time multiply recursive reflections and refractions using hybrid rendering", in *The Visual Computer*, Volume 31, Issue 10, Pages 1395-1403, September 2014.

IV. Per Ganestam, Rasmus Barringer,
Michael Doggett and Tomas Akenine-Möller,
"Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees", in *Journal of Computer Graphics Techniques*, Volume 4, Number 3, Pages 23-42, September 2015.

V. Per Ganestam and Michael Doggett
"SAH guided spatial split partitioning for fast BVH construction", to appear in *Computer Graphics Forum (Proceedings of Eurographics)*, Volume 35, Number 2, 2016.

Contents

1 Introduction 1
    1.1 Computer generated imagery 2
    1.2 Real-time rendering 5
    1.3 Offline rendering 7
2 Rendering algorithms and hardware 9
    2.1 Rasterization and the graphics processing unit 9
    2.2 Ray tracing and the central processing unit 11
3 Spatial data structures 14
    3.1 Variations 15
    3.2 Bounding volume hierarchy 15
    3.3 Combinatorial complexity 16
    3.4 Quality evaluation 17
    3.5 Geometric top-down 18
    3.6 Geometric bottom-up 19
    3.7 Space-filling curves BVH construction 20
    3.8 Inital tree optimization 22
    3.9 Triangle splitting 23
    3.10 SIMD efficiency 24
4 Methodology and Contributions 25
Bibliography 27

Paper I: Auto-tuning Interactive Ray Tracing using an Analytical GPU Architecture Model **33**

1 Introduction 35
2 Previous Work 36
3 GPU Performance Model 37
    3.1 Parameters for Different GPUs 40

4 Estimating Workload . . . 40
4.1 Ray Tracing Parameters . . . 41
4.2 Estimating Shader Cache Performance . . . 42
4.3 Instruction Counting . . . 42
4.4 Tuning Ray Tracing Parameters . . . 44
5 Results . . . 45
6 Conclusion . . . 48
Bibliography . . . 51

Paper II: Power Efficiency for Software Algorithms running on Graphics Processors **53**
1 Introduction . . . 55
2 Methodology . . . 57
3 Case Studies . . . 59
3.1 Case 1: Primary Rendering . . . 59
3.2 Case 2: Shadow Algorithms . . . 60
3.3 OpenGL ES . . . 61
4 Results . . . 61
5 Conclusions and Future Work . . . 67
Bibliography . . . 69

Paper III: Real-time multiply recursive reflections and refractions using hybrid rendering **71**
1 Introduction . . . 73
2 Related Work . . . 73
3 Algorithm . . . 76
3.1 Primary Visibility . . . 77
3.2 The Cube Map . . . 78
3.3 BVH Construction . . . 78
3.4 Ray Tracing: BVH and Cube Map Traversal . . . 79
3.5 Shadows and Deferred Rendering . . . 82
4 Results . . . 82
4.1 Limitations . . . 83
5 Conclusion . . . 84
Bibliography . . . 89

Paper IV: Bonsai: Rapid Bounding Volume Hierarchy Gener-

ation using Mini Trees **93**
1 Introduction . . . 95
2 Previous Work . . . 96
3 Background BVH Generation . . . 97
4 Our Implementation of Sweep SAH . . . 98
5 Bonsai BVH Algorithm . . . 99
5.1 Compute Midpoints . . . 100
5.2 Mini Tree Selection . . . 100
5.3 Mini Tree Construction . . . 101
5.4 Bonsai Pruning . . . 102
5.5 Top Tree Construction . . . 103
6 Implementation . . . 104
7 Results . . . 105
8 Conclusions and Future Work . . . 110
Bibliography . . . 113

Paper V: SAH guided spatial split partitioning for fast BVH construction **117**
1 Introduction . . . 119
2 Previous Work and Background . . . 120
3 Algorithm . . . 122
3.1 The Surface Area Heuristic . . . 122
3.2 SAH guided mid-point split partitioning . . . 123
4 Implementation . . . 124
4.1 Triangle Splitting and Recursively Growing Memory . . . 125
4.2 Bonsai and 8-wide Trees . . . 127
5 Results . . . 128
5.1 Binary Trees . . . 130
5.2 Triangle counts . . . 131
5.3 Splitting performance . . . 132
5.4 SAH cost . . . 132
6 Conclusion . . . 132
Bibliography . . . 137

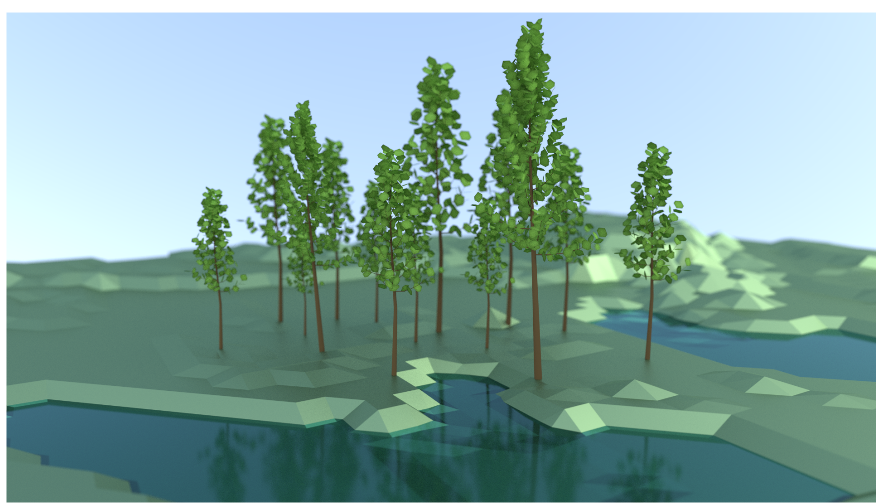


Figure 1: Path traced grove with ponds. Although a simple scene, this rendering displays a multitude of effects used in computer graphics. A glossy ground and water with reflections and refractions accompanied by diffuse inter-reflections. The leaves interact with light via subsurface scattering and the sky color is determined by a physically based sky model. The camera model creates a depth of field effect based on its aperture settings.

1 Introduction

Most people, perhaps without realizing it, experience computer graphics to some extent on a daily basis. It may be small things such as viewing an email on a smartphone or it may be an obvious feature at the cinema.

Computer Generated Imagery (CGI) is used in a multitude of disciplines and has become an important aid in medicinal visualization. Two closely related example applications where computer graphics and visualization techniques are used are Computed Tomography (CT) and Magnetic Resonance Imaging (MRI). Another modern usage of computer graphics in medicine is in visualizing three dimensional ultrasound data.

Architectural tools utilize advanced rendering and illumination techniques to allow architects a realistic view and a feeling of what their project may look like without having to construct it. Similar tools are also used in industry by, for example, automobile manufactures when designing new models or visualizing engine components, or when marketing the final products.

Although represented in many disciplines, the entertainment industry is by far the largest producer of CGI. Most feature films and television series utilize CGI to some extent. Even in cases where CGI is obvious, it may be difficult to differentiate real filmed objects from computer generated ones.

It is not uncommon to hide stunt actors' faces behind digital models of the actors they are covering for. Terminator 4 is a recent example of using a cover face of an actor when an aging robot had to battle a younger version of himself. The younger robot was played by a younger actor who had his face exchanged for a computer generated face that looked and behaved as the original robot's did three decades ago. Furthermore, entire landscapes may be added with exciting weather and perhaps wild animals. A feature film stage often consists of a mixture of real and computer generated objects and computer animated film is the extreme case where everything on screen apart from voices is generated digitally. The current trend in the creation of CGI in films is to use a rendering technique known as *ray tracing*.

Another field where CGI is extensively used is in computer and video games. Although some of today's game settings and scenes, at first glance, may be difficult to distinguish from reality, CGI used in games suffers from constraints that aren't as strict in film rendering. Everything that is displayed on screen in a real-time game has to be created within a few milliseconds. Due to the hard real-time constraint the computationally demanding ray tracing technique usually isn't fast enough. Instead, a real-time rendering method called *rasterization* is used.

The focus in this thesis is part real-time rendering hardware and part ray tracing and the underlying accelerating spatial data structures used by ray tracing algorithms to make them computationally feasible. Improving quality and construction times of spatial data structures is beneficial to offline rendering as it results in reduced overall rendering time. It also brings ray tracing a step closer to becoming a reasonable real-time rendering approach. To further reduce the offline to real-time gap this thesis also presents a real-time hybrid rendering approach that combines the speed of rasterization with the quality of ray tracing.

1.1 Computer generated imagery

The 3D World

Object representation is usually defined in a 3-dimensional cartesian space. Figure 2 describes a simple scene in computer graphics which consists of three abstract components. The objects to be rendered, one or more light sources that illuminate the scene, and an abstract camera model used to capture images. In addition to 3-dimensional space, a time dimension may be added. Time introduces motion and animation and each discreet image sample in time is referred to as a *frame*. Animation is not exclusive to geometric objects represented in a scene. The camera position and camera settings such as aperture, zoom and focus may also change with time. Light sources may also move or change properties between frames.

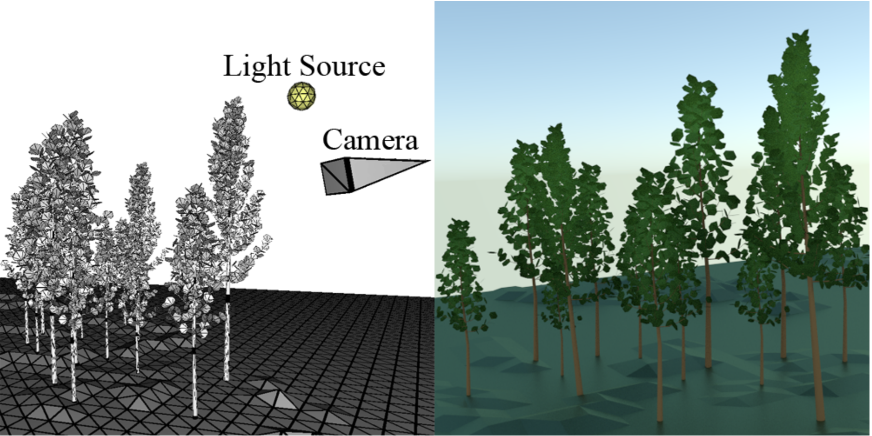


Figure 2: To the left, a wireframe rendering of a scene consisting of a ground plane, a few trees, a camera and a single light source. The right side is a ray traced image representing the setup presented in the left image.

The Triangle

The typical atom in computer graphics is the *triangle*. A triangle is represented as a set of three points in space and its edge vectors span a single plane in three dimensions. A triangle edge vector from point $\mathbf{p^0}$ to point $\mathbf{p^1}$ is computed as $\mathbf{e^0} = \mathbf{p^1} - \mathbf{p^0}$. The points representing a triangle are also known as *vertices*. The triangle is the simplest geometric entity that spans a plane in three dimensions and, if all vertices are different, is guaranteed to span precisely one plane. As an example, a *quad* which consists of four points may span a plane with three of its points, but it is not guaranteed that the fourth point lies on that plane. The property that a triangle is guaranteed to span precisely one plane simplifies visibility determination in computer graphics and is the main reason why triangles are favored rather than more complex polygons.

A collection of triangles where all triangle share at least one vertex with another triangle is known as a *mesh*. Meshes are used to model complex objects such as houses or trees in 3-dimensional space. A coarse grained mesh represented by only a few triangles is considered to be lowly tessellated and a more detailed and refined mesh is referred to as highly tessellated. Three unit spheres with different levels of tessellation are displayed in Figure 3.

Camera

The simplest camera model used in computer graphics is analog to the real world pin-hole camera. An image rendered with this type of camera

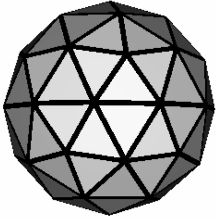
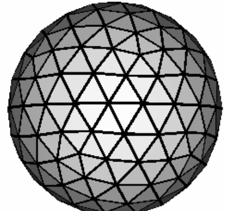
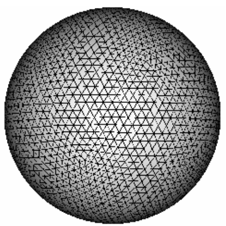




Figure 3: Three triangle meshes that represent the same parametric surface, a unit sphere. From left to right is a refinement in quality due to an increased level of tessellation. The sphere to the left coarsely approximates a sphere however renders faster then the highly tessellated sphere to the right.

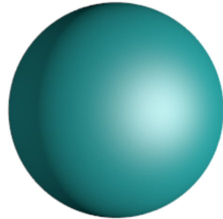
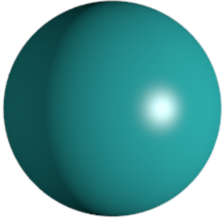



Figure 4: Phong shading of three unit spheres. To the left a matte material with no specular highlight. In the middle the specularity is increased but the highlight is still smooth. To the right the specular intensity is high and the sharpness of the specular highlight is increased.

may seem flat since all objects, regardless of distance to the camera, are in perfect focus. There are many possible extensions to the pin-hole camera model, either as real-time approximations or physically based concepts used in offline rendering. To achieve *depth of field*, a lens has to be simulated and to add the effect of *motion blur*, time has to be considered within a frame.

Illumination and shading

Without lighting computations a rendered object would seem plain and artificial. Typical early forms of illumination or *shading* methods in real-time graphics are Gouraud and Phong shading [13, 41]. Phong shading, illustrated in Figure 4, computes direct illumination from point lights and blends the result with a diffuse and an ambient term. Illumination with Phong shading can easily be computed per pixel in real-time on modern graphics hardware. It may be difficult to represent all types of lighting and materials using Phong shading only. In modern rendering systems, mathematical models form the basis of material and lighting computations.

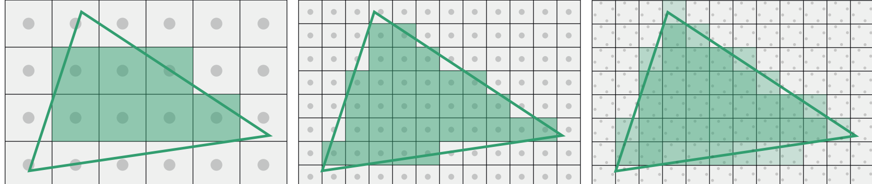


Figure 5: Three triangles with different level of aliasing artifacts. The triangle to the left suffer from under sampling because of too low resolution. The middle triangle is represented with a higher resolution but still only has one sample per pixel. The triangle to the right displays smoother edges and a visually adequate appearance thanks to four rotated grid MSAA samples.

Real-time rendering uses fast approximate techniques and in offline rendering lighting and material properties are often simulated using physically based models. Ray tracing with physically based material and illumination computations can represent properties of objects so that a rendered image sometimes is visually indistinguishable from a photograph.

Sampling and aliasing

A common artifact[1] is aliasing. Figure 5 presents three versions of a triangle where aliasing is affecting the visual quality in different ways. The first example is obviously under sampled in the sense that the resolution is too low. The second triangle looks better than the first and has four times higher resolution. However, aliasing is still present and the edges are perceived as rather jagged. The third triangle in Figure 5 utilizes a technique known as Multi-Sampled Anti Aliasing (MSAA) [2]. MSAA works by sampling each pixel in more than one place followed by a weighted blend of all triangles visible to any sample within a pixel. With more than one sample per pixel the aliasing artifact is reduced and the color of a pixel becomes a weighted blend of all visible triangles covering a pixel. Sampling patterns and techniques are important topics in computer graphics, both when it comes to real-time and offline rendering.

1.2 Real-time rendering

In real-time rendering the task at hand is to generate everything that is to be displayed on the screen many times every second. The requirement of a modern real-time application is to update the screen, and thus render a

[1]In computer graphics, an artifact is an undesired visual effect such as noise or a grainy image.

new image of the scene, at a frequency in the range 30–60Hz. Considering that time constraint, an image has to be rendered in 16–33ms.

The computational task to generate a fully realistic looking image using physically based lighting and materials is challenging. Consider a simple scenario, an enclosed room with a single light source, blue walls and a white floor. Light is emitted in a continuous spectrum with spikes at different wavelengths depending on the type of light source. When light hit the walls some of its spectrum is absorbed and some is reflected. Depending on the physical properties of the walls, the angle of reflection might be affected due to micro facets in the wall and the wavelength of the light. Reflected light with spectral properties different from the original light bounces from the walls, onto the floor, to the ceiling and back to the walls, before eventually passing through the optics of the eye of the observer and finally excites light sensitive receptors in the eye. In addition to these possible light paths in a simple environment, there is participating media such as smoke, fog or dust, that the light can pass through.

Naturally, it is not possible to perfectly simulate physically based lighting and materials within such a small time budget. Instead in real-time graphics almost all computations, shadows, lighting and materials are clever approximations. Although not all real-time applications seek photo-realism in rendering, for those that do, methods that reduce computational complexity while retaining as much realism as possible are used.

Figure 6 illustrates the benefit of *ambient occlusion*, an example algorithm that approximates a global illumination phenomena. In the real world, corners and creases are usually less illuminated with soft shadows which darkens the deeper the crease is. Adding the ambient occlusion effect has a positive impact on realism and an approximation can be computed quickly, adding only a few milliseconds to the overall per frame rendering time [34].

There have been many creative approximations in real-time graphics and they tend to push the available hardware to its limits. As dedicated hardware improves, from fixed function accelerators to fully programmable *graphics processing units* (GPU), so does the approximations, bringing real-time rendering closer to realistic looking images.

Adding shadows to rendering is important not only to improve realism but also necessary as a visual aid to understand scale and positioning of an object. There are several methods to compute shadows and there are also many modern improvements. A couple of well known basic techniques are *shadow volumes* [6] and *shadow maps* [52].

A popular technique to allow many light sources in real-time rendering is *deferred shading*. It is a multi pass algorithm that renders colors, normals and depth values to separate buffers. The collection of buffers are called a *G-buffer* [43]. Using the G-buffer, lighting computations are performed in

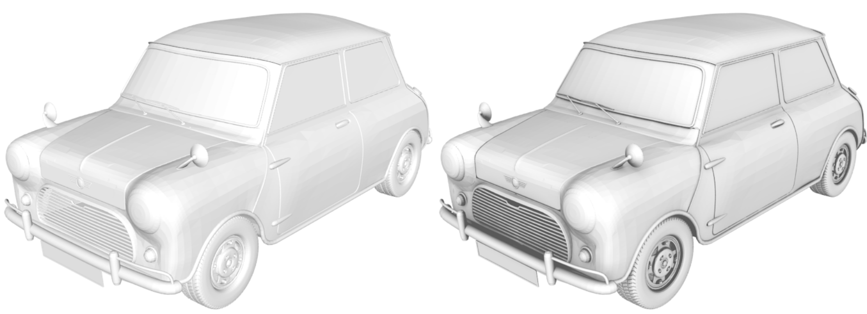


Figure 6: Both cars are shaded with the same simple diffuse lighting computations. The car to the right is rendered with ambient occlusion, which can be seen as darker details in cracks and other places where some parts of the car may limit the light that illuminates other parts of the car. Ambient occlusion enhances details and improves realism.

a second pass.

In Paper II deferred shading is one of the rendering techniques used when comparing energy efficiency of different rendering algorithms. Furthermore, energy efficiency of shadow algorithms is also investigated. In addition to shadow volumes and shadow maps, the technique *variance shadow maps* [7] is also included in the energy measurements.

Two effects difficult to achieve in real-time are refractions and specular reflections. Reflections and refractions can be computed with little overhead in screen space [32]. However, screen space techniques are limited to reflect what is currently visible to the camera and would miss objects reflected from the outside of the *view frustum*.

Paper III addresses the difficulty of representing global reflections and refractions in real-time by separating the scene into two regions. A smaller region close to the camera is rendered with Whitted ray tracing [51], allowing accurate reflection and refraction computations. The larger but further away region surrounding the ray traced region is approximated using rasterization techniques similar to screen space reflections. However, rather than limiting secondary visibility to screen space objects, in Paper III we utilize a G-buffer cube map that captures the entire scene from the origin of the camera. Instead of using the G-buffers for deferred shading, the G-buffers are used to approximate ray tracing in geometry far away from the camera.

1.3 Offline rendering

Offline rendering is the complement to real-time rendering, creating computer generated images often using physically based rendering techniques [40]

and trading longer rendering times for improved visual quality.

It is not uncommon to spend hours to generate a high quality image, even if it is just a single frame of a feature film. The goal is not always to create photorealistic imagery but lighting computations can still be physically based and the ultimate task is to solve the rendering equation [19]

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

In short, the rendering equation represents the outgoing light L_o, referred to as *radiance*, from any 3-dimensional point x in the scene in the direction ω. The radiance is computed as the sum of light emitted L_e from x in the direction ω and the proportion of the total incoming light on the hemisphere Ω that is scattered in the direction ω. The proportion and properties of the light scattered is determined by f_r, the Bidirectional Reflectance Distribution Function (BRDF). The BRDF can be thought of as a mathematical representation of the material properties of the surface.

Finding a solution, or approximation, to the rendering equation involves computing the light interaction between each and every primitive in the scene. Common methods to solve the rendering equation are Monte Carlo based ray tracing methods, such as *path tracing* [19], that gradually converge to a solution with each additional sample computed. Ray tracing and path tracing are further discussed in Section 2.2.

Looking back at the example discussed as impossible in real-time rendering in Section 1.2, when considerably longer rendering times are allowed and using techniques that at any time can interact with any part of the scene without loss of information, it is possible to consider all the complex light paths and material interactions. However, when the primitive count of a scene reaches tens or hundreds of millions, the computational needs grow. To be able to render high quality images of vast and detailed scenes in *only* minutes or hours, a huge amount of computational recourses are needed. In the visual effects industry large distributed systems called *render farms* are used, parallelizing image computations across hundreds or thousands of processors [37].

The computational task to produce an animated or effects heavy feature film can be difficult to comprehend. As an example, consider a 2 hour production that is required to be rendered at 24 frames per second. The total number of frames to render are 172800. Assuming an average frame time of only 3 minutes, total rendering time would reach almost one year of around the clock rendering. To be more precise, rendering such a film would take exactly 360 days.

2 Rendering algorithms and hardware

Although it is not exclusive for real-time rendering to use Graphics Processing Units (GPU) and vice versa for offline rendering methods to use Central Processing Units (CPU), the following sections will discuss two rendering algorithms, *rasterization* and *ray tracing*, together with the hardware they are commonly paired with.

Both rasterization and ray tracing are methods of visibility determination and the rendering pipeline using either of them is built up in stages, where finding which triangle is visible to a given pixel is at the heart of the algorithm.

2.1 Rasterization and the graphics processing unit

The foremost choice as a real-time rendering algorithm is rasterization. Rasterization has been pushing the development of graphics hardware and although modern GPUs are more versatile, GPU hardware was originally designed for real-time rendering using rasterization.

Early graphics hardware accelerators were designed solely as fixed function pipelines with no programmable stages. In addition to the hardware *rasterizer*, they also had hardware support to compute transforms and lighting. As graphics hardware matured, programmable *shader cores* were introduced. A shader core is a Single Instruction Multiple Data (SIMD) processor designed for high arithmetic throughput and to hide latencies when fetching data from memory. The SIMD width of a shader core may be as wide as 1024 or 2048 bits, as an example performing 32 or 64 floating point operations per SIMD instruction. A GPU is optimized for fine grained parallel execution, and high throughput is achieved by allowing thousands of threads to stay active at any time. When a high latency memory fetch is issued, the shader core switches tasks and performs arithmetic computations with other available threads. If there are enough active threads to switch between, it is possible to completely hide latency caused by fetching data from main memory. Although reducing bandwidth usage with a cache hierarchy is important, with the concept of task switching for latency hiding, GPUs are not as dependent on their caches and the cache hierarchy as CPUs to achieve high throughput.

In Paper I, a GPU hardware model was used to understand the expected workload of a given algorithm. The hardware model estimate was used to auto-tune a real-time rendering implementation to maximize rendering quality per frame subject to a limited time budget. Although this section's main focus is rasterization, it is worth mentioning that the rendering technique used in Paper I was ray tracing, which is further discussed in Section 2.2.

The graphics hardware pipeline as it is implemented in modern GPUs, in

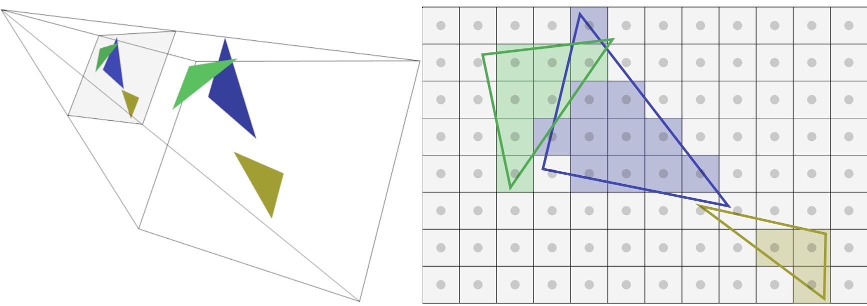


Figure 7: The left image illustrates how geometry is projected onto a view plane prior to rasterization. The right image is an example of the rasterization process with one sample per pixel. The z-buffer ensures that the green triangle is rendered in front of the blue triangle.

broad terms consist of three stages. The first stage is geometry processing and although this stage consists of additional optional components such as the geometry shader and the tessellation shader, the vertex shader is the most important component that cannot be omitted. The vertex shader executes a user defined program for each vertex in the scene's vertex array stream. Typically, the vertex shader is used to animate three-dimensional objects. Given a vertex, its transform and view projection represented as a $4x4$ matrix, the vertex shader outputs are the transformed vertices projected to the view plane together with some user specified attributes possibly needed later in the pipeline. Common attributes that are passed through the pipeline from the vertex shader are vertex normals and texture coordinates.

The second stage in the pipeline is rasterization. Rasterization is a fast technique to determine primary visibility in computer graphics and is implemented as fixed function hardware in modern GPUs. Figure 7 illustrates 2-dimensional rasterization. Given the transformed triangles projected to the view plane, the algorithm scans from left to right each pixel and tests which, if any, triangles are covering that pixel. In case of overlapping triangles a z-buffer [5] is used to know which triangle is closest to the viewer. The z-buffer stores the per pixel depth of the currently nearest triangle. If an overlapping triangle is found to be further away it can be culled without spending time on any lighting computations. A common improvement to the rasterization algorithm is to divide the grid of pixels into tiles. Working on tiles of pixels improves locality and thus improves cache behavior. There are many further improvements that are not covered in this brief algorithmic description.

Following geometry processing and rasterization is pixel processing. The last programmable stage in the pipeline is the *fragment shader*. The fragment shader is executed per pixel and outputs the final color to shade an object

with. It is not uncommon that fragment shading is the pipeline stage where most time is spent. The fragment shader is usually responsible for lighting computations, shadow computations and other visual effects that are added to the final image.

The last stage of the pipeline is blending. There may be multiple overlapping fragments and the blending stage, implemented in hardware, is tasked to composite overlapping fragments into a final color.

2.2 Ray tracing and the central processing unit

Although recursive ray tracing as a rendering algorithm was first described by Whitted [51] in 1980, ray tracing is often used as an umbrella term describing a set of ray tracing algorithms. Unlike rasterization, ray tracing doesn't reduce dimensionally by projecting the geometry to a view plane to test which pixel a given triangle belongs to. Instead, ray tracing is thought of as *shooting* rays from each pixel and visibility is determined by finding the nearest intersections of rays and 3-dimensional scene geometry. The intersection closest to the origin of a ray is considered as the ray's *hit point* and the hit point becomes the source of lighting computations.

The primary rays produce exactly the same visibility and 2-dimensional simplifications as rasterization. However, with dedicated hardware available in GPUs these computations can be made significantly faster with rasterization. When it comes to secondary visibility such as shadows, reflections and refractions, ray tracing is inherently suitable but rasterization requires specialized methods to achieve visually acceptable approximations. Having said this, approximations developed for rasterization are often many times faster than ray tracing and are used extensively in real time applications. One high quality approximation is the earlier mentioned hybrid rendering approach presented in Paper III.

Ray tracing without the support of an accelerating data structure is computationally unfeasible, with linear time complexity per ray. Assuming only primary rays, rendering an image would exhibit $O(NM)$ time complexity where N is the number of pixels to render and M is the number of primitives in the scene. It is possible to overcome this unfortunate time complexity by the use of an accelerating spatial data structure. By encapsulating the scene geometry into a spatial tree structure, time complexity can be reduced to $O(logM)$ per ray. Accelerating data structures for ray tracing are further discussed in Section 3.

Computing shadows using ray tracing is performed by shooting a new ray from the hit point to each light source. If any intersection with geometry happens between the hit point and a light source, the hit point is considered to be in shadow from that light source. Furthermore, other secondary visibility such as rays originating from reflective or translucent materials

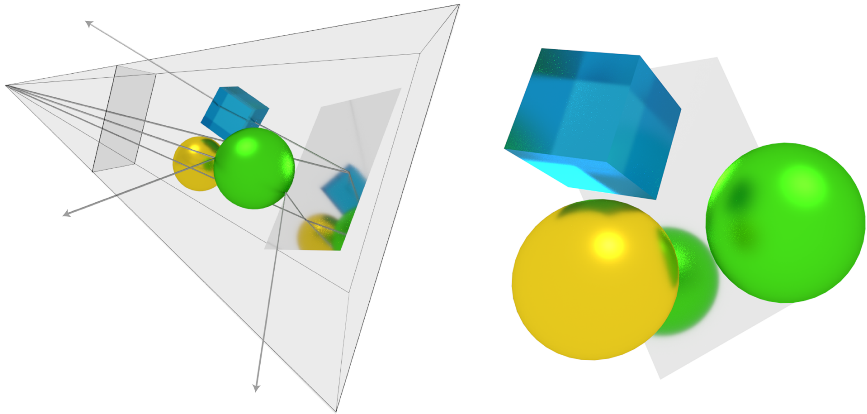


Figure 8: To the left is an illustration of three ray paths as they progress through the scene. Each primary ray passes through a pixel in the view plane and continues through the scene, creating new rays at each intersection point. Recursion is terminated when a ray finally leaves the scene without intersecting anything, alternatively a ray is terminated due to the probability of not producing another ray or because a maximum recursion depth bias has been reached. To the right is a path traced image from the point of view of the left image setup.

are computed recursively by shooting new rays in the direction of reflection or refraction of the original ray and hit point. In each recursive step the final color of a pixel is accumulated and refined. Recursive ray tracing that follows a ray from each pixel through a scene of various materials certainly improves realism, but does not converge towards a solution to the rendering equation described in Equation 1.

An improvement to Whitted's recursive ray tracing [51] is path tracing [19] which was developed in conjunction with the rendering equation and thus designed to solve the equation. Path tracing is a probabilistic approach based on the statistical Monte Carlo method. In path tracing many samples, i.e. rays, have to be computed from each pixel and the final color is the average of the samples. Instead of strictly spawning reflective or refractive recursive rays like it is done in Whitted's ray tracing, path tracing has a probability to spawn a ray in a random direction of the hemisphere covering the hit point. Eventually, the path will terminate, either due to the probability of not spawning a ray or because of a biased decision of a maximum path depth. The color contribution of an entire ray path is a weighted sum of each hit point's contribution and the contribution at a hit point is decided by the BRDF of the intersected object. In production rendering, path tracing or one of its derivatives are the algorithms commonly used for high quality and photo realistic CGI.

Although dedicated ray tracing hardware is being researched [30], ray tracing systems are often built completely in software. Ray tracing can be efficiently implemented on either GPU or CPU hardware. An example ray tracing frame work designed for GPU hardware is NVIDIA's OptiX [38] and an equivalent implementation for CPU hardware is Intel's Embree [49].

The modern x86 CPU [16], which is the most common CPU architecture available in laptops, desktops and servers, usually consists of more than one CPU core per chip and exhibits a possible linear performance improvement with the number of cores, less the serial execution limitation defined by Amdahl's law. Ray tracing, like rasterization, is an algorithm that exposes ample amounts of parallel workload where every pixel can be computed in parallel and each available hardware thread can be dedicated to a pixel or a tile of pixels. Due to deep and large cache hierarchies available in CPUs, divergent workload can be handled efficiently. Like GPUs, modern CPUs also implement fine grained parallelism in the form of SIMD hardware, although usually not as wide as its GPU counterpart. Current x86 CPUs are available with 256-bit SIMD width [16] and it is likely that upcoming hardware generations will be available with a 512-bit SIMD configuration.

Exploiting SIMD in ray tracing is less straight forward than implementing thread level parallelism. Although vector operations can be performed in a SIMD fashion, it is uncommon to see vectors with more than four elements in computer graphics. Rays can be traced in packets, however it has been found that divergence within ray packets, as a consequence of path tracing, quickly reduces SIMD efficiency [8]. Another approach to utilize SIMD in ray tracing is by adapting the spatial data structure to make it more SIMD friendly. Such an approach is discussed in Section 3.

Path tracing is one of the simpler ray tracing algorithms that can be used with physically based rendering which makes it a popular choice, but there are some shortcomings which are handled more robustly with other approaches.

Caustic effects such as the intricate patterns of light at the bottom of the seabed produced by sun light refracting through gentle waves are difficult to simulate with path tracing. The probability that a randomly directed ray from a hit point on the seabed will refract through the waves and hit the light source is extremely low and thus it may take a very long time for path tracing to converge to a solution to the rendering equation.

Bi-directional path tracing [28] improves *light transportation* by allowing rays to be traced both from the camera and from the light sources. Figure 9 illustrates caustic and dispersion effects simulated using bi-directional path tracing. Two pass methods such as photon mapping [18] and progressive photon mapping [15] are also more robust in situations like the caustic case. The first pass in photon mapping sends rays from the light sources through the scene and produces a photon map. The second pass shoots rays from

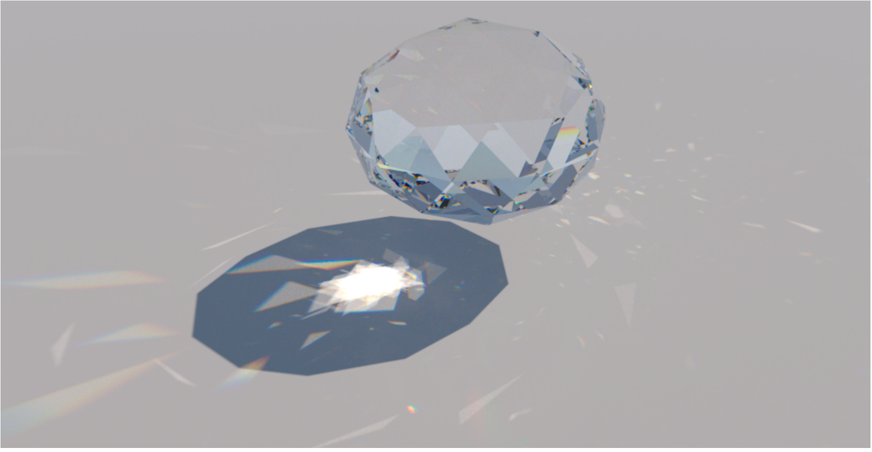


Figure 9: A simulation of light paths through a cut diamond using bi-directional path tracing. Without the possibility of tracing rays from the light source as well as the camera it would take a very long time for the caustic effects to converge.

the camera and gathers light from the photon map at each intersection.

3 Spatial data structures

As mentioned in the previous section, an important technique to reduce computational complexity for ray tracing algorithms is to utilize an accelerating spatial data structure. By representing scene geometry in an efficient spatial data structure, it is possible for a ray to quickly discard large segments of the scene, since it can be known that no ray-triangle intersections are possible in the discarded regions.

Choosing data structure and construction algorithm is not as simple as to always select the fastest or to always select the one resulting in the best ray tracing performance. Naturally, if one algorithm produces the same quality data structure at a reduced build time compared to another algorithm then there is no reason not to chose the faster one. However, sometimes it might be a better choice to use an algorithm that is faster in terms of build times but results in reduced rendering performance. If it is known that the number of rays to trace is relatively small and that rendering time will be short, then it might be beneficial to utilize a data structure algorithm that is optimized for build speed rather than ray tracing performance. On the other hand, if it is known that a large number of rays will be traced, for example when path tracing in a feature film, then it is probably worth spending some additional time on producing a higher quality data structure

that also improves rendering performance.

Paper IV presents a novel fast CPU based BVH algorithm that achieves ray tracing performance similar to, or sometimes even better than, contemporary BVH approaches. Paper V improves BVH quality compared to Paper IV by also allowing triangles to be split during BVH construction, without suffering from significantly increased build times.

3.1 Variations

There are several varieties of spatial data structures that have been used in computer graphics in one or another way. Among the simpler types are uniform grids [25, 20] where a scene is quantified into a predefined number of grid cells and each primitive is assigned to all grid cell that touches the primitive. Uniform grids are fast to construct but often inefficient in eliminating empty space and are sub-optimal in most ray tracing situations. A slightly more effective data structure is the quadtree [44] represented in three dimensions known as an octree [33]. An octree is a hierarchical approach which improves ray traversal time compared to uniform grids. However, grid cells are not adaptively refitted to the triangles enclosed by the cell and a ray may still have to traverse many grid cells that do not contain any geometry that could be intersected by the ray. Improving further upon octrees are kd-trees [3, 17]. A kd-tree adaptively separates primitives in a top down fashion and thus is able to reduce empty space efficiently. However, in a kd-tree, it is likely to duplicate primitive references since it is impossible to guarantee that there exists a plane that separates all primitives in two disjoint sets. Instead, primitives that intersect a split plane are referenced to both children of a tree node. Although kd-trees exhibit good ray tracing performance they have been considered impractical due their a priori unknown memory foot print.

3.2 Bounding volume hierarchy

One of the most popular contemporary data structures for ray tracing is the Bounding Volume Hierarchy (BVH). The BVH benefits from an a priori known upper bound in memory usage, efficient empty space elimination and very good ray tracing performance. The typical BVH is a binary tree structure where each node in the tree encapsulates the volume of its children. A 2-dimensional BVH of a few triangles is illustrated in Figure 10. Leaf nodes represent a set of primitives that each need to be tested against a ray that reaches the leaf during tree traversal. A BVH references each primitive only once and allows spatial overlap of sibling nodes. It may seem inefficient to allow node overlap, however, with a good build heuristic the overlap can be minimized and the benefits of the BVH as an accelerating data structure often amortizes its drawbacks.

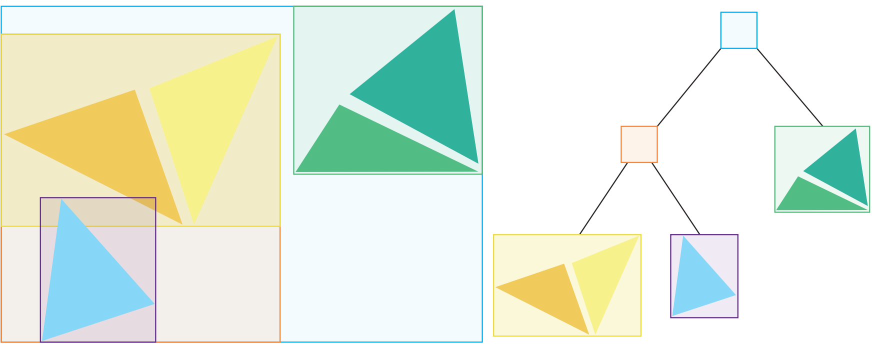


Figure 10: To the left a 2-dimensional analog of a BVH and to the right its corresponding tree structure. This example illustrates a simple spatial median object split build strategy with at most two triangles per leaf node.

A BVH is traversed recursively, starting by letting a ray intersect the root node. If a node is intersected its child nodes are tested against the ray, prioritizing the child node closest to the origin of the ray. The other child is pushed on a stack and may be processed later. Traversal continues as long as there are nodes on the stack or until an intersection with a primitive is found and no other nodes on the stack are closer than the intersected primitive.

3.3 Combinatorial complexity

Considering that scenes in computer graphics and especially in production rendering commonly consist of triangles counted in millions, it is obvious that there are many potential solutions to a BVH. The number of ways to build a full binary tree, one where each node has either two children or none at all, for $n+1$ elements is defined by the nth Catalan number [45]

$$C_n = \frac{1}{n+1}\binom{2n}{n}, n \geq 0. \tag{2}$$

The first few catalan numbers are relatively small and the full binary trees representing catalan number 3 are illustrated in Figure 11. However, already with as few elements as there are letters in the alphabet, the number of possible ways to build a binary tree grows to become a 13 digit number. The combinatorial possibilities to construct a BVH is even greater, since a leaf node may contain an arbitrary, relatively small, number of triangles.

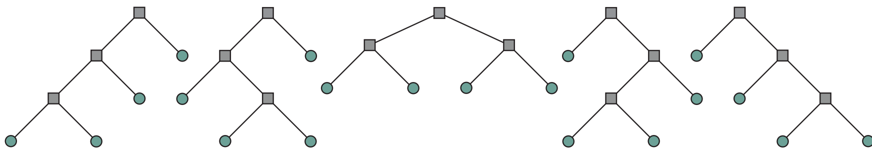


Figure 11: Five possible variations of binary trees with four leaf nodes. The number of combinations is represented by the 3rd catalan number. Circles are leaf nodes and squares represent internal nodes.

3.4 Quality evaluation

Although there is an immense number of solutions that result in functional BVHs, it is likely that most of these solutions would suffer from poor ray tracing performance. A good build strategy together with a BVH quality metric would quickly discard most poor solutions. Such a metric needs to account for certain properties that correlate to ray tracing performance. The by far most used BVH quality metric is the Surface Area Heuristic (SAH) [12], which incidentally is also used to define a common top-down build strategy for BVH construction [31]. The SAH cost of a BVH is evaluated as

$$C_i \sum_{n \in I} \frac{A(n)}{A(n_0)} + C_l \sum_{n \in L} \frac{A(n)}{A(n_0)} + C_t \sum_{n \in L} \frac{A(n)}{A(n_0)} N(n), \tag{3}$$

where the result depends on the surface area A of all nodes, the number triangles N referenced by leaf node $n \in L$ and the respective cost constants of traversing all internal nodes C_i, traversing leaf nodes C_l and intersecting leaf nodes C_t. Internal nodes are represented by the set I and leaf nodes by the set L. The root node is denoted n_0. Modern usage of the equation often omits the second sum by defining C_l as zero since most ray tracing implementations don't traverse leaf nodes but instead immediately compute intersections with the triangles represented in a leaf node.

The SAH metric represents the cost of tracing a randomly directed ray starting and ending outside of the BVH and $A(n)/A(n_0)$ is the conditional probability $P(R(n)|R(n_0))$ of a non-terminating randomly directed ray R to intersect node n provided that it has intersected the root n_0. Although minimizing the SAH cost is a common approach known to work well to construct high quality BVHs, it has been found that the SAH may not always correlate perfectly to ray tracing performance [10, 1]. The main issue with the SAH is its simple assumption that only randomly directed rays that begin and end outside the root node without actually intersecting any geometry are considered. In path tracing it is true that most rays are randomly directed with the exception of camera rays. However, it is not true that rays begin and end outside the root node. In fact, the majority of

rays will begin from a surface somewhere inside the BVH, and in the case of an enclosed scene such as an indoor environment, all rays will terminate inside the BVH.

One approach to improve the SAH metric is presented by Fabianowski [10] where ray origins uniformly distributed within the root node of a BVH are considered. Another approach by Aila et al. [1] introduced two new post-build metrics to improve BVH cost estimation, End Point Overlap (EPO) and Leaf Count Variability (LCV). EPO is similar to SAH but considers rays starting and terminating within the BVH and LCV is an approach to estimate SIMD utilization. Fabianowski's metric can be evaluated at build time and thus also improve ray tracing performance. The additional metrics by Aila et al. aren't evaluated at build time and don't improve ray tracing performance. Instead they are used to describe the performance of a particular BVH built with one of many available algorithms. Aila et al. found that the new BVH quality metric as a combination of SAH and EPO correlated well with ray tracing performance for non SIMD execution and by also considering the LCV term their metric correlated well with ray tracing performance on SIMD hardware as well.

3.5 Geometric top-down

The canonical approach of a top-down BVH builder is to initially consider the complete set of triangles and using a build heuristic decide how to divide the set into two subsets. The builder then continues recursively by considering each subset as an independent partition to further subdivide until a stopping criteria terminates the recursion.

The simplest build heuristics are object median and spatial median splits. Object median split requires the triangles to be sorted in the dimension that the split will occur and in each recursion simply divides the array of triangles in two halves. Spatial median split is slightly more sophisticated where the triangles are partitioned to one or the other side of a split plane with the plane defined as the spatial median of the parent BVH node. Since these techniques in different ways ignore important properties of the scene it is possible to create large BVH nodes containing spatially unrelated triangles. Although spatial median split is likely to produce a BVH with a tighter fit to the scene than object median split both approaches suffer from poor ray tracing performance when compared to more sophisticated build methods.

A BVH build strategy often considered the gold standard among BVH algorithms is Sweep SAH [31]. The typical implementation greedily searches for the best object split in all three spatial dimensions and minimizes the SAH cost at each level of recursion. As it is not computationally feasible to consider all possible partitions the sweep approach maintains the triangles in sorted order in x,y and z during construction and only considers object splits in the sorted arrays as candidate solutions. Sweep SAH is known to

produce high quality BVHs at the cost of longer build times. However, in Paper IV we present a Sweep SAH implementation with build performance competitive to faster BVH construction approaches.

An efficient approximation to Sweep SAH is the Binned SAH algorithm proposed by Wald et al [48] and further refined by Wald [47]. Instead of considering split candidates at each triangle in sorted x, y and z order, triangles are partitioned into bins defined by a set of uniformly distributed spatial split planes. The binned algorithm then evaluates the SAH cost of choosing a particular split plane and selects the one that results in the lowest SAH cost. Contrary to Sweep SAH, the binned algorithm does not need to sort or maintain a sorted order of triangles and only needs to perform one pass per dimension over the triangles to reference them to their respective bin. The binned approach is also thread friendlier than its predecessor. Sweep SAH doubles parallelism at each recursion, starting with only one thread. Binned SAH allows all available threads to perform binning even at the first level of recursion.

In addition to experiments in Paper IV and Paper V, both Sweep and Binned SAH have been thoroughly tested and found to be robust high quality BVH algorithms [47, 22, 1, 49].

Our BVH algorithm in Paper IV is mostly classified as a geometric top-down approach. However, the last stage of our method, where a set of pre-constructed and optimized *mini trees* are passed to a Sweep SAH builder, may be considered a bottom-up approach.

3.6 Geometric bottom-up

Bottom-up approaches in BVH construction are tempting due to the inherently parallel algorithmic structure similar to divide and conquer algorithms such as merge sort. Walter et al. [50] proposed a bottom-up BVH algorithm based on agglomerative clustering. Naive agglomerative clustering has an unfortunate time complexity $O(N^3)$ unsuitable even as a parallel algorithm [50]. Walter et al. devised a fast agglomerative clustering algorithm that uses a coarse un-optimized kd-tree of triangle mid points as an auxiliary data structure. Although with longer construction times than Sweep SAH, Walter et al. found that their approach reduced overall SAH cost and improved ray tracing performance of their test scenes.

Inspired by the work of Walter et al. the Approximate Agglomerative Clustering (AAC) algorithm was proposed by Gu et al [14]. Noting that initial clustering was a dominant factor of execution time Gu et al. spatially limited the clustering search space by initially arranging the triangles into a space filling curve known as Morton order. However, unlike the space-filling curve based builders described in Section 3.7, AAC does not implicitly utilize the Morton order for BVH construction. Rather than globally searching

for nearest neighbors while clustering using a kd-tree [50], the AAC algorithm only needs to search a pre-calculated Morton order based local subset to find nearest neighbors. By exchanging construction, update and search queries of a kd-tree for a once computed and sorted Morton order, Gu et al. found that their AAC algorithm could be executed in linear time to the number of triangles. They also showed that they had similar ray tracing performance as the original agglomerative clustering BVH by Walter et al.

Although the SAH cost of a BVH built with one of the agglomerative clustering algorithms often is on a par with, and sometimes lower than, the SAH cost of a Sweep SAH BVH, as discussed in Section 3.4, lower SAH costs does not always correlate to improved ray tracing performance. In the case of the agglomerative clustering methods there seems to be some discrepancy between BVH SAH cost and ray tracing performance [1].This discrepancy is also briefly discussed in Paper IV.

3.7 Space-filling curves BVH construction

A space-filling curve in broad terms is a continuous mapping from the unit interval onto a 2-dimensional unit square of a higher order n-dimensional hypercube. The first such curve was discovered by Peano in 1890 [39] and is known as the Peano curve. Another well known space-filling curve is the Z-order curve, also named Morton curve [35] after its inventor. For BVH construction the Morton curve is favorable due to its simple implementation and representation in 3 dimensions. A point in 3-dimensional space can be quantized into a 32-bit or 64-bit integers as a 30-bit or a 63-bit Morton code respectively, where x, y and z values are interleaved and every three bits represent a different level of the hierarchy where the most significant three bits are level zero. Strictly morton based BVH builders implicitly utilize the hierarchical 3-dimensional morton order to construct BVHs. Most notable work in space-filling curve BVH construction are the Linear BVH (LBVH) by Lauterbach et al. [29], Hierarchical LBVH (HLBVH) by Pantaleoni et al. [36] which was further improved by Garanzha et al. [11] and Karras' algorithmic improvement to LBVH construction [21]. All of the mentioned algorithms were originally designed for GPU BVH construction.

Lauterbach et al. noted that representing triangle mid points in sorted 3-dimensional Morton order implicitly defined a BVH structure. Their approach recursively builds a BVH top-down similar to a simple top-down spatial median builder. The difference is that instead of recursively partitioning at the spatial median, LBVH recursively partitions where the most significant bit of the Morton code differs. To finalize the BVH a linear time pass computes the *bounding boxes* of the LBVH nodes.

HLBVH improved the LBVH approach by utilizing two levels of sorting [36]. LBVH was implemented with a parallel radix sort to create the sorted Morton order and sorting was a large part of total build time. Pantaleoni et

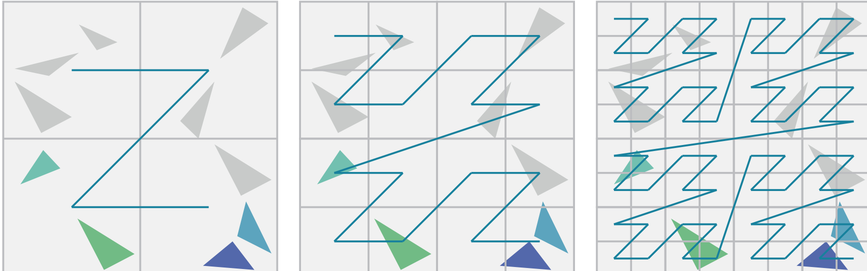


Figure 12: Three levels of a 2-dimensional Morton curve. One issue with space-filling curve based BVHs is the same as with uniform grids. Many triangles may be quantized to the same Morton code if the resolution isn't high enough. The two triangles in the bottom right corner aren't separated until the last level in the illustration.

al. found that improved build speed due to improved data locality could be achieved if only the n most significant bits were sorted with a global radix sort approach and the lower bits instead sorted with odd-even sorting [26] in GPU shared memory. HLBVH also made it possible to construct the top-level of the hierarchy represented by the n most significant bits simultaneously as the lower levels of the hierarchy.

Karras [21] used the sorted Morton order to build a binary radix tree representing a skeleton of the expected BVH, and each node represented in the radix tree in fact could be computed in parallel. Furthermore Karras presented a highly parallel bottom-up approach to compute the bounding boxes of each BVH node. The real-time hybrid rendering algorithm presented in Paper III uses Karras' [21] Morton based BVH algorithm to allow a per frame full rebuild of the BVH.

The space-filling curve algorithms discussed in this section share a couple of properties. They are all highly parallel algorithms that execute efficiently on GPU hardware. However, even though both LBVH and HLBVH present SAH optimization approaches, they also produce BVHs with a relatively poor ray tracing performance compared to the top-down algorithms [22].

Although ray tracing performance of Morton curve based BVHs are not as good as with other algorithms, there are still situations where faster builders are preferred. If it is known that the number of rays to trace will be relatively small, a faster builder, introducing less build time overhead, may result in an overall improved rendering performance. Fast to build low quality BVHs may also be used as initial trees further improved by iterative optimization techniques.

3.8 Inital tree optimization

A relatively new approach in producing high quality BVHs is to begin with a lower quality hierarchy and perform iterative optimization to improve tree quality and ray tracing performance. A precursor to such approaches was introduced by Kensler [23] where the initial BVH is optimized by tree rotations. Although Kensler initiated optimization with a high quality BVH he found that the SAH cost could be reduced further by recursively traversing the BVH and performing tree rotations in a depth-first order. Kensler noted that the simple and fast recursive optimization strategy had a tendency to get stuck at local minima in regards of SAH cost and also proposed a tree rotation approach in combination with Simulated Annealing [24]. Although requiring longer build times, occasionally forcing poor solutions made it possible to escape local minima in favor of continued optimization.

An efficient version of Kensler's tree rotations used to maintain high quality BVHs of animated scenes was introduced by Kopta et al. [27]. Instead of strictly optimizing an initial BVH, Kopta et al. found tree rotations to be an effective instrument to avoid BVH deterioration when the BVH is refitted due to animated objects.

Bittner et al. [4] took a different approach to BVH optimization and instead of using tree rotations optimized BVHs with node removal and insertions. Optimization is performed by iteratively selecting costly internal nodes from the initial BVH, removing those nodes from the BVH and reinserting them in positions that reduce the SAH cost the most. Bittner et al. noted that although tree rotations [23] technically could find the global minima, convergence was slow and the insertion based optimization resulted in higher quality BVHs in a significantly shorter time [4].

Considering the concepts and ideas of both rotation based and insertion based optimization Karras et al. [22] developed a massively parallel high quality BVH algorithm designed for GPU hardware. The primary observations of previous work was that although simple tree rotations could improve BVH quality, the inherent risk of getting stuck in local minima, unless using excessively time consuming optimization algorithms such as Simulated Annealing, made tree rotations less favorable. Insertion based optimization begins with a low quality BVH and produces superior BVHs, sometimes even of higher quality than top-down sweep SAH [22] but the algorithm is inherently serial and not likely to perform well on parallel hardware [4, 22]. The BVH construction approach by Karras et al. optimizes an LBVH initially built with Karras' LBVH algorithm [21]. Instead of simple tree rotations entire sub-trees called *treelets* are exhaustively optimized with a dynamic programming approach. The optimization approach by Karras et al. results in BVHs with a quality not far from top-down sweep SAH but build times are significantly reduced [22].

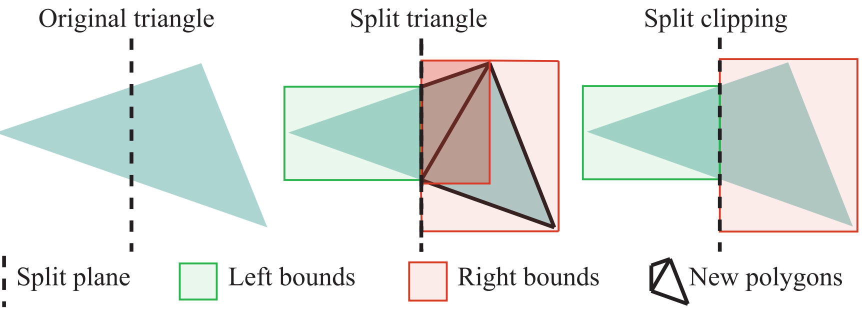


Figure 13: An illustration showing the benefits of performing split clipping rather than actual triangle splitting. In this case it is obvious that split clipping creates tighter bounds than triangle splitting.

3.9 Triangle splitting

As mentioned earlier in Section 1.1 the triangle can be considered as the atom of computer graphics and just like its real world counterpart, despite the meaning of the name, the atom of computer graphics may be split in half. An approach closer to the truth regarding triangle splitting is known as *split clipping* [17]. As Figure 13 shows, rather than actually splitting a triangle into multiple triangles, triangle bounding boxes can be split in two parts and both parts refitted to form a tight bound of each respective part of the triangle. Actually splitting a triangle may produce a triangle and a *quad*, i.e. a polygon with four vertices, rather than two triangles. Further refinement would eventually be required where the quad is divided into two triangles and consequently also two bounding boxes. Another benefit of split clipping compared to actual splitting is that split clipping is likely to produce a smaller summed area of the split bounding boxes.

Triangle splitting for BVHs can be categorized into two groups, *pre-split* and *intra-split* where pre-split represents a pass prior to BVH construction and intra-split is a splitting approach within a BVH algorithm.

Splitting triangles prior to or while building a BVH may improve ray tracing performance dramatically [46, 42, 22]. A scene with an even distribution of finely tessellated triangles will probably not benefit much from triangle splitting, but typical scenes consist of a mix of large and small triangles and such scenes are likely to result in improved ray tracing performance when using a triangle splitting BVH approach. Two modern robust approaches are the top down intra-split builder by Stich et al. [46] and the pre-split method by Karras and Aila [22]. The first, known as Split BVH (SBVH), is a combination of the greedy sweep SAH builder and a builder with a set of predefined spatial split planes as with binned SAH. However, the binned split planes actually split the triangle and reference the two parts to their

respective side of the split plane. In each recursion, the algorithm picks the better of sweep SAH and the best available predefined split plane. Thorough experiments with SBVH show that although other algorithms are getting closer, SBVH generally produces the highest quality BVHs [46, 22, 1]. A variant of SBVH is available in Embree [49] as a slow to build high quality BVH.

The other modern robust triangle splitting approach is the pre-splitting algorithm by Karras and Aila [22]. Inspired by Early Split Clipping (ESC) by Ernst and Greiner [9] where triangles are split recursively according to a cost heuristic, Karras and Aila introduce a split budget and employ a novel heuristic that ranks the importance of splitting a specific triangle. They also define a function of spatial split planes that are likely to improve the SAH cost rather than looking at uniformly distributed split planes or simply subdividing each triangle until a certain threshold of bounding box areas have been reached [9]. Karras' and Aila's pre-split approach reduces BVH build times significantly compared to SBVH but still results in ray tracing performance close to that of SBVH. The approach by Karras and Aila is designed for GPU hardware and is thus likely to perform better on a GPU than a CPU.

In Paper V we present a new approach for triangle splitting. Our algorithm is optimally integrated with a BVH builder as an intra-split approach and the Bonsai algorithm from Paper IV is favorable. However, we also show that our algorithm produces high quality BVHs when used as a pre-split approach together with sweep SAH. Furthermore, in Paper V we show that our algorithm is faster than competing CPU based robust triangle split builders yet producing BVHs of quality similar to SBVH.

3.10 SIMD efficiency

As mentioned in Section 2.2 an approach to improve SIMD utilization for ray tracing requires a slightly different topology of the data structure. Instead of building BVHs with two children per node, a multi-BVH [8] is used, with as an example four or eight children per node. With eight children per node it is possible to intersect all children at the same time using Intel's Advanced Vector Extensions (AVX), the current standard of x86 SIMD instructions. Intel's Embree [49] ray tracing framework currently implements branching widths of 4 and 8 and the binned BVH builder is biased towards packing 4 or 8 triangles per leaf node, depending on the SIMD width of the architecture compiled for.

4 Methodology and Contributions

This thesis presents topics on hardware modeling, algorithmic energy efficiency, hybrid rendering and BVH construction. My supervisor Michael Doggett has had an active part and co-authored all papers presented in this thesis. Paper II was also authored by Björn Johnsson and Tomas Akenine-Möller. Rasmus Barringer and Tomas Akenine-Möller co-authored Paper IV.

My early work was mostly hardware centric with focus on hardware modeling and energy efficiency. Paper I presents and extends an available GPU hardware model and tunes it to function with hardware slightly different from what the model was originally designed for. The paper sought to find means of utilizing a GPU model to predict the behavior and execution time of more complex algorithms than the simple benchmarks earlier presented by previous work. The end result is a performance predictive real-time ray tracing system that by the means of a hardware model and a feedback controller auto-tunes ray tracing quality to maintain a steady state rendering performance. In Paper I Michael Doggett implemented the original hardware model and I focused on its extension, the GPU based real-time ray tracer and the feedback system.

Paper II continues the hardware centric path and presents energy efficiency experiments and measurements of a set of different rendering and shadow algorithms executed on a broad range of GPUs. Energy efficiency data was gathered by tapping in to the power supplies of high-end GPUs, an integrated GPU and a mobile phone GPU. One significant difficulty in this paper was to understand measured data. The tool used to sample voltages and currents at a high frequency was decoupled from actual rendering and thus we had to find a way to detect when our rendering measurements actually started. My focus, other than running time consuming experiments, was on analyzing sampled energy data and finding ways of synchronizing and making sense of the large data sets gathered by the energy measuring tool. The first author, Björn Johnsson focused on implementing the various rendering algorithms used, setting up test scenes and also running experiments and analyzing data.

Following the first two papers, rendering focus changed from hardware to algorithms. Paper III presents a hybrid rendering approach that combines the visibility queries possible using ray tracing with the performance benefit of using dedicated hardware and rasterization. The hybrid rendering algorithm quickly builds a low quality BVH of geometry nearby the camera and represents geometry outside of the BVH with a G-buffer cube map. This approach allows for detailed and correct reflections and refractions near the viewer and approximate reflections and refractions by stepping through the G-buffer cube map outside the BVH. We found that our approach could render visually acceptable reflections and refractions in real-time on patho-

logical test cases such as animated scenes consisting of over two million triangles, all reflective or refractive.

Paper IV presents Bonsai, a novel approach to BVH construction that builds BVHs with a ray tracing performance on par with the sweep SAH algorithm discussed in Section 3. Furthermore, our approach is the fastest CPU based high quality BVH algorithm and it utilizes both thread level parallelism and SIMD instructions efficiently. In this paper we also present an implementation of sweep SAH that constructs BVHs with competitive build times. My focus was on overall algorithmic design and implementation. Rasmus Barringer implemented the fast sweep SAH approach as well as an highly efficient radix sorting approach. It is worth mentioning that I also implemented a proof of concept version of Bonsai into Embree and the source code is currently available at Embree's github repository. As the proof of concept implementation was found to scale well with many core hardware and also expected to scale well with wider SIMD instructions the Embree team has decided on implementing their own version of Bonsai.

My final paper is a continuation of Paper IV. In Paper V the Bonsai algorithm is implemented into Intel's Embree framework and most measurements are made using Embree. Paper V presents a novel and efficient method to build BVHs with triangle splitting. The algorithm can either be integrated with Bonsai, adding only a small increase in build times, or works as pre-split pass prior to construction using any other BVH algorithm. In the pre-split case the overhead in build times is slightly larger than with the Bonsai integrated version. Our splitting approach is the fastest CPU based BVH builder that utilizes triangle splitting and yields a ray tracing performance comparable to that of SBVH.

Credits

Various models credited to others have been used in my papers and I would like to extend my gratitude to those who created these models. Thanks to Veronica Sundstedt for the Kalabsha temple model and to Timo Aila for the Italian and Arabic City models. Thanks to Gilles Tran at `www.oyonale.com` for the Mini model and thanks to Martin Lubich at `www.loramel.net` for the Austrian Imperial Crown model. The San Miguel scene was created by Guillermo M. Leal Llaguno and the original Sibenik Cathedral and Atrium Sponza Palace was created by Marko Dabrovic. The extended version of Sponza used in the papers is copyrighted Frank Meinl at Crytek and the Hairball scene is contributed by NVIDIA Research.

Bibliography

[1] Timo Aila, Tero Karras, and Samuli Laine. On Quality Metrics of Bounding Volume Hierarchies. In *High-Performance Graphics*, pages 101–107, 2013.

[2] Kurt Akeley. Reality engine graphics. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 109–116, 1993.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[4] Jirí Bittner, Michal Hapala, and Vlastimil Havran. Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum,*, 32(1):85–100, 2013.

[5] Edwin Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, University of Utah, 1974.

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

[7] William Donnelly and Andrew Lauritzen. Variance Shadow Maps. In *Symposium on Interactive 3D Graphics and Games*, pages 161–165, 2006.

[8] M. Ernst and G. Greiner. Multi Bounding Volume Hierarchies. In *IEEE Interactive Ray Tracing*, pages 35–40, 2008.

[9] Manfred Ernst and Günther Greiner. Early Split Clipping for Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 73–78, 2007.

[10] Bartosz Fabianowski, Colin Flower, and John Dingliana. A cost metric for scene-interior ray origins. In *Eurographics Short Papers*, pages 49–52, 2009.

[11] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. Simpler and Faster HLBVH with Work Queues. In *High-Performance Graphics*, pages 59–64, 2011.

[12] Jeffrey Goldsmith and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics & Applications,*, 7(5):14–20, 1987.

[13] H. Gouraud. Continuous shading of curved surfaces. *IEEE Trans. Comput.*, 20(6):623–629, June 1971.

[14] Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. Efficient BVH Construction via Approximate Agglomerative Clustering. In *High-Performance Graphics*, pages 81–88, 2013.

[15] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. *ACM Trans. Graph.*, 27(5):130:1–130:8, December 2008.

[16] P. Hammarlund, A.J. Martinez, A.A. Bajwa, D.L. Hill, E. Hallnor, Hong Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R.B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro,*, 34(2):6–20, 2014.

[17] Vlastimil Havran and Jiri Bittner. On Improving KD-Trees for Ray Shooting. In *Winter School on Computer Graphics*, pages 209–217, 2002.

[18] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping.* 2001.

[19] James T. Kajiya. The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, volume 20, pages 143–150, 1986.

[20] Javor Kalojanov and Philipp Slusallek. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 23–28, 2009.

[21] Tero Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In *High-Performance Graphics*, pages 33–37, 2012.

[22] Tero Karras and Timo Aila. Fast Parallel Construction of High-quality Bounding Volume Hierarchies. In *High-Performance Graphics Conference*, pages 89–99, 2013.

[23] A. Kensler. Tree Rotations for Improving Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 73–76, 2008.

[24] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[25] Krzysztof S. Klimaszewski and Thomas W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Comput. Graph. Appl.*, 17(1):42–51, January 1997.

[26] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[27] Daniel Kopta, Thiago Ize, Josef Spjut, Erik Brunvand, Al Davis, and Andrew Kensler. Fast, effective bvh updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 197–204, 2012.

[28] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (COMPUGRAPHICS '93*, pages 145–153, 1993.

[29] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum,*, 28(2):375–384, 2009.

[30] Won-Jong Lee, Youngsam Shin, Jaedon Lee, Seok Joong Hwang, Soojung Ryu, and Jeongwook Kim. An energy efficient hardware multithreading scheme for mobile ray tracing. In *SIGGRAPH Asia 2014 Mobile Graphics and Interactive Applications*, SA '14, pages 1:1–1:3, 2014.

[31] David J. MacDonald and Kellogg S. Booth. Heuristics for Ray Tracing Using Space Subdivision. *Visual Computer,*, 6(3):153–166, 1990.

[32] Morgan McGuire and Michael Mara. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):73–85, December 2014.

[33] Donald Meagher. Octree encoding: a new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. Technical report, Rensselaer Polytechnic Institute, Image Processing Laboratory, 1980.

[34] Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages 97–121, 2007.

[35] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd., 1966.

[36] J. Pantaleoni and D. Luebke. HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry. In *High-Performance Graphics*, pages 87–95, 2010.

[37] Jacopo Pantaleoni, Luca Fascione, Martin Hall, and Timo Aila. PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes. *ACM Transaction on Graphics,*, 29(3):37.1–37.10, 2010.

[38] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics,*, 29(4):66:1–66:13, 2010.

[39] Giuseppe Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36(1):157–160, 1890.

[40] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation.* MKP, 2nd edition, 2010.

[41] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, jun 1975.

[42] Stefan Popov, Iliyan Georgiev, Rossen Dimov, and Philipp Slusallek. Object Partitioning Considered Harmful: Space Subdivision for BVHs. In *High-Performance Graphics*, pages 15–22, 2009.

[43] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, pages 197–206, 1990.

[44] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, June 1984.

[45] Richard P. Stanley. *Enumerative Combinatorics, Volume 2.* Cambridge Studies in Advanced Mathematics, 2001.

[46] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial Splits in Bounding Volume Hierarchies. In *High-Performance Graphics*, pages 7–13, 2009.

[47] Ingo Wald. On Fast Construction of SAH-based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 33–40, 2007.

[48] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1), January 2007.

[49] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics,*, 33(4):143:1–143:8, 2014.

[50] B. Walter, K. Bala, M. Kulkarni, and K. Pingali. Fast Agglomerative Clustering for Rendering. In *IEEE Symposium on Interactive Ray Tracing*, pages 81–86, 2008.

[51] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM,*, 23(6):343–349, 1980.

[52] Lance Williams. Casting Curved Shadows on Curved Surfaces. volume 12, pages 270–274, 1978.

Paper I

Auto-tuning Interactive Ray Tracing using an Analytical GPU Architecture Model

Per Ganestam   Michael Doggett

Lund University

Abstract

This paper presents a method for auto-tuning interactive ray tracing on GPUs using a hardware model. Getting full performance from modern GPUs is a challenging task. Workloads which require a guaranteed performance over several runs must select parameters for the worst performance of all runs. Our method uses an analytical GPU performance model to predict the current frame's rendering time using a selected set of parameters. These parameters are then optimised for a selected frame rate performance on the particular GPU architecture. We use auto-tuning to determine parameters such as phong shading, shadow rays and the number of ambient occlusion rays. We sample a priori information about the current rendering load to estimate the frame workload. A GPU model is run iteratively using this information to tune rendering parameters for a target frame rate. We use the OpenCL API allowing tuning across different GPU architectures. Our auto-tuning enables the rendering of each frame to execute in a predicted time, so a target frame rate can be achieved even with widely varying scene complexities. Using this method we can select optimal parameters for the current execution taking into account the current viewpoint and scene, achieving performance improvements over predetermined parameters.

GPGPU-5 Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units
Pages 94-100, March 2012

1 Introduction

Programming GPUs for high performance requires a careful balance of several hardware specific related factors that is typically only achieved by expert users through trial and error. GPUs are massively parallel devices with parallel compute capacity exceeding other single chip devices and are still the best device for high performance graphics [8]. There are currently many APIs for programming GPUs all with their respective advantages and disadvantages, but getting optimal performance from the GPU is still a challenging task that requires repetitive manual tuning. To reduce the amount of trial and error required to achieve optimal performance, general guidelines can be followed or different metrics can be considered to predict performance, but ultimately a trial and error process is still prevalent. In this paper, we present a method that makes this tuning process automatic using an analytical GPU model.

The current challenges of programming and getting efficient performance from GPUs is likely to increase in the future as new devices have increasingly complex architectures. While new features, such as shader cache hierarchies, make programming easier, getting the best efficiency is still difficult. Also as pointed out by Owens et al. [8], the cost of memory bandwidth in comparison to computational power is ever increasing. On future devices applications will need to switch to modes that require more compute processing, whereas on older devices a better trade off between memory and compute is necessary. Power usage is also an important consideration and Dally [4] points out that the power cost of a memory operation is an order of magnitude greater than compute operations. Sometimes it is important to tune parameters for power usage instead of just performance. Using a single set of parameters for all devices is therefore inadequate and per device tuning is also needed, resulting in an ongoing maintenance task.

We present a method for automatically tuning the parameters of a parallel application running on massively parallel devices, GPUs. Automatic tuning is performed by estimating the run time for a given set of parameters using a GPU analytical model [7] and then changing parameters and re-estimating until a chosen performance target is met. A major advantage of this system over using feedback from the previous frame is that the rendering load based on viewpoint can change dramatically between frames, hence a previous frame's feedback loop could become inaccurate and highly unstable. We model GPU performance across a range of GPU architectures including older generations and different vendors by using the OpenCL API. We evaluate our auto-tuning method with a ray tracing application, because it has a range of parallel properties including a large amount of parallel work, high memory bandwidth usage, and a workload that can be sometimes coherent and sometimes incoherent.

This paper contributes a new method for auto-tuning using an analytical

GPU model. Our method uses a feedforward controller to improve the performance estimate achieved over what is possible with just the GPU model. In addition, a simplified version of Hong and Kim's GPU model [7] is presented. This paper also presents GPU modeling across both GPU vendors, NVIDIA and AMD.

2 Previous Work

Early efforts to tune rendering performance include work by Funkhouser et al. [6] which tunes the rendering algorithm using predetermined geometry properties and previous frame rates to ensure a user-specified frame rate. By using a cost and benefit calculation for each object a target frame rate can be achieved by setting appropriate LOD levels and adjusting image quality for each object in the scene. In our approach we also control rendering parameters, but use a GPU model to get accurate estimates of the final performance for the current frame enabling much faster tuning.

Making efficient use of GPU hardware resources has always been a challenging task and a large amount of research is devoted to the discovery of improved tuning parameters for a particular algorithm that have been found through trial and error or developer tools. But recent efforts to find more general methods have been presented. Using metrics such as the number of instructions and threads running on a GPU at one time, Ryoo et al. [9] shows how measures of utilization and efficiency can be computed to predict which regions of the complete space of available optimizations need to be tested in order to find the optimal setting. This method reduces the search time considerably, but still requires iterating over variables to find the best optimizations. Their model also only works if the application does not have memory bandwidth issues.

In recent years several efforts have been made to understand and model GPU architectures. Hong et al. [7] propose an analytical GPU model that can be used to estimate the performance of algorithms by also taking into account the impact of memory operations. They count the number of memory transactions as well as the actual address to better model the amount of parallelism available when memory requests happen on GPUs. They measure GPU characteristics using micro-benchmarking and can then make performance estimates using their analytical model. We use this model in this paper to tune our application. Bakhoda et al. [3] presents a detailed GPU simulator which takes PTX instructions and executes them to analyze execution performance. Their simulator gives accurate performance estimates but does not provide quick estimates that can be used to rapidly tune performance before execution. Baghsorkhi et al. [2] present a GPU model that also models important properties such as scratch-pad memory access and control flow divergence using a work flow graph. Further details

of GPU architecture including cache sizing can also been determined via the use of micro benchmarking [10].

Recent results in auto-tuning show that using statically determined parameters for algorithms that run on GPUs always produce poorer results than tuning those parameters to the GPU capabilities [5]. Davidson et al. [5] also show that by running multiple benchmarks of an algorithm, optimal performance can be achieved. In this paper we improve upon this result by directly querying a GPU model to determine the best parameters. This enables a wider range of parameters to be checked, which is important for complex algorithms such as ray tracing.

In this paper we tune the ray tracing algorithm for image synthesis. Previous work in GPU ray tracing has also used statically determined parameters or algorithm changes to improve performance. Aila et al. [1] introduced the concept of persistent threading in order to improve GPU utilization beyond that achieved by the hardware work scheduler. They found that packet tracing was not much faster than per-ray tracing even though per-ray introduces more incoherent memory accesses. To improve performance they created scheduling threads on the GPU, called persistent threads, which improved the performance two-fold by moving scheduling into the GPU thread. Persistent threads are an example of a non-standard performance optimization that programmers may not be aware of, but could be incorporated into an auto-tuning system.

3 GPU Performance Model

To estimate the performance of an algorithm on the GPU, we use a GPU model based on Hong and Kim's model. [7]. We use Hong and Kim's model on AMD GPUs as well as NVIDIA GPUs. The model is deterministic and straight forward to implement. It also executes quickly with practically no overhead, hence it is well suited for real-time applications. In this section we present a shorten version of their analytical GPU model. The properties of the program and the device are measured as shown in Table 1. The number of cores, D_c, is the number of Streaming Multicores (SM) for NVIDIA GPUs and SIMDs for AMD GPUs.

To compute the total number of cycles, several intermediate values are also computed as shown in Table 2. These intermediate values are computed using the same equations as Hong and Kim [7] and the variable name from their model is also shown in the Table.

GPUs are throughput oriented architectures capable of running thousands of program threads in parallel [8]. When a thread requests data from memory the GPU switches to other threads to ensure the GPU continues to do work while it waits for memory to respond. This switching is done on a warp (called wavefront for AMD hardware) basis. In the worst case, when

Program	
i_c	Number of compute instructions
i_m	Number of memory instructions
t_b	Number of threads per block
b_t	Number of blocks to run
b_o	Maximum blocks per core (occupancy)
Device	
M_d	Device total memory bandwidth
D_c	Number of cores
D_{tw}	Device number of threads per warp
W_d	Device resource limited number of warps $= \frac{b_o t_b}{D_{tw}}$
C_i	Cycles to execute one instruction
Benchmarked	
c_{ml}	Averaged memory latency cycles for one memory operation

Table 1: Input variables used for the GPU model.

		Hong09 variable
M_w	Memory bandwidth per warp	*BW_per_warp*
c_c	Computation cycles	*Comp_cycles*
c_m	Memory cycles	*Mem_cycles*
c_t	Total execution cycles	*Exec_cycles_app*
c_{ml}	Averaged memory latency	*Mem_L*
c_{dd}	Averaged departure delay to issue a memory instruction	*Departure_delay*

Table 2: Variables computed by the GPU model.

data is not in any on-chip cache and must be read from off-chip global memory, the original thread must wait a memory latency period. This memory latency is measured using benchmarks for each chip and represented by the variable c_{ml} in graphics core clock cycles. While GPUs are capable of running multiple kernels, we assume that the device is running only multiple instances of the same kernel program. The maximum number of warps that are runnable on a core is determined by available resources such as OpenCL private memory, OpenCL local memory (NVIDIA shared memory) and other device specific features. We use the device specified parameter

that limits the number of blocks (groups of warps) per core to determine this maximum which is called W_d.

Our objective is to compute the number of graphics core clock cycles that it takes to execute the specified program. When a warp issues a global memory request, it is put to sleep and other warps are run instead. While waiting for the sleeping warp's memory request, the GPU can run the compute instructions from other warps. We can compute the number of warps that run only compute instructions while waiting as

$$w_c = \frac{c_m + c_c}{c_c}. \tag{1}$$

This is the maximum amount of compute that can be done measured in warps. This must be capped by the maximum number of warps, W_d so we take $\min(w_c, W_d)$ and call it *Compute Warp Parallelism* (CWP) [7].

Next, we consider how many memory operations could be performed. First, we compute the maximum number of parallel warps that can issue a memory operation while the first warp is sleeping. The number of warps that can concurrently issue a memory operation is computed as follows:

$$w_{md} = \frac{c_{ml}}{c_{dd}}. \tag{2}$$

Each warp that makes memory requests uses some of the limited memory bandwidth resulting in a maximum number of warps that can run. This maximum number of warps is computed as

$$w_{mb} = \frac{M_d}{M_w D_c}. \tag{3}$$

Again, these numbers of memory warps must be limited by the maximum number of warps of the device, W_d, so we take
$\min(w_{md}, w_{mb}, W_d)$ and call it *Memory Warp Parallelism*
(MWP) [7].

From this, we can compute the total number of cycles for the kernel to run by multiplying the total of compute and memory cycles per block by the total number of block repetitions required on the device:

$$c_t = \frac{b_t}{b_o D_c}(C_i w_t(i_c + i_M) + c_b). \tag{4}$$

The total execution equation requires two variables which are determined by the limiting case for parallelism, compute warps w_t, and memory cycles per block c_b. For these two variables, there are three possible cases. The first case is when $MWP > CWP$, i.e., only CWP warps will be able to run because of the amount of compute instructions, so the device is *arithimetic* bound. In this case, the maximum number of warps that can run in parallel

is determined by device properties, and this is the main contributor to the total number of cycles.

The second case is when $CWP > MWP$, i.e., only MWP warps will be able to run because of the amount of cycles memory instructions require, so the device is *memory* bound. The maximum number of parallel warps is determined by MWP. The third case is when the number of warps that can run in parallel for compute and memory instructions are the same and equal to the device limit of number of warps running, this case is referred to as *balanced*. Once the type of limitation is worked out, w_t and c_b are determined as shown in Table 3.

	Arithmetic	Memory	Balanced
w_t	W_d	$\frac{MWP-1}{i_m}$	$\frac{MWP-1}{i_m}+1$
c_b	c_{ml}	$\frac{W_d c_m}{MWP}$	c_m

Table 3: Total execution time variables.

3.1 Parameters for Different GPUs

Several of the model parameters are measured using a series of synthetic benchmarks with known numbers of compute and memory operations. The memory operations are either consecutive memory accesses so that they will be combined, called coalescing or random addresses. More details about these benchmarks can be found in Hong and Kim [7].

The measured memory parameters are memory latency, c_l, departure delay for coalesced memory access, c_{dc}, and for uncoalesced memory access, c_{du}. These three values are varied to find the best fit for the particular architecture. The coalesced memory access takes into account the lower memory access time required for shader loads and stores from different threads in the same warp. These memory values and also the architectural parameters of three different GPUs are shown in Table 4

In Table 4, for the Geforce580 architecture, we set C_i to 2 to account for the 2 instructions issued per SM resulting in a halving of the effective cycle time for each instruction. The blocks per core determined by occupancy is given as the maximum value for the architecture 'Max b_o', in Table 4.

4 Estimating Workload

Ray tracing renders an image of a 3D scene by tracing a path from the eye into the geometry, intersecting with objects in the scene. For a ray tracer

	GF 8800	GF 580	Radeon 5870
M_d	86	192	153
D_c	16	16	20
D_{tw}	32	32	64
C_i	4	2	4
Max b_o	8	8	24
F (GHz)	1.35	1.54	0.88
Benchmarked			
c_l	420	550	500
c_{dc}	4	0.08	64
c_{du}	9.8	0.66	1

Table 4: GPU dependent variables.

to handle large complex scenes, a hierarchical data structure is typically used to improve the performance of finding intersections. In particular we use a bounding volume hierarchy (BVH) constructed using a surface area heuristic. Each ray starts at the root node that contains a bounding box for the entire scene. For each node that is intersected, the two child nodes are read into memory, typically from global memory, and intersection testing is performed with their bounding boxes. This continues recursively until the leaf nodes are reached. Once the ray reaches a leaf node in the BVH, it intersects with the triangles contained there. To estimate performance of ray tracing the algorithm is divided into several major components such as the number of nodes traversed, the number of nodes read in, and the number of triangles intersected.

Since every scene is different and even viewpoints within a scene vary greatly, we use a low resolution ray tracer to estimate the ray tracing specific parameters. One of the benefits of working with OpenCL is that we can easily make use of the CPU to quickly estimate these values and not put extra load on the GPU.

4.1 Ray Tracing Parameters

Beyond the basic surface of the objects in the scene, the actual lighting of the scene requires computation as well. To create a more realistic scene, more complex computation is required. Different techniques can be incrementally added to increase the realism and in this paper, we control the level of realism based on the available hardware. We add shadow rays and ambient occlusion (AO). Shadow rays trace a ray from the surface to each light source to determine if the surface is in light or shadow. AO attempts to approximate the light that is reflected by the scene to a point by calculating how much of a white hemisphere around the point it can 'see'. The visibility

of the hemisphere is calculated by tracing a selected number of rays in random directions and terminating them at a set radius. We tune the performance of AO by adjusting the number of rays and the terminating radius.

4.2 Estimating Shader Cache Performance

The NVIDIA Fermi architecture used in the GeForce 4XX and 5XX series includes an L1 and L2 cache hierarchy for global memory loads and stores from a kernel program. This memory hierarchy improves performance significantly for our BVH based ray tracer, as the BVH nodes are frequently stored in this cache. When running the low resolution frame estimate, we store the final BVH node for each ray. We count the number of rays that end at the same node within a region of the screen and assume that rays that terminate at the same node are likely to have taken a similar path through the BVH and so when reading nodes from memory, the nodes are likely to be already in the cache. This estimate of cache hits is represented by the variable m and used to estimate the performance by modifying the instruction count.

4.3 Instruction Counting

We calculate the compute and memory instruction counts for our ray tracing kernel on NVIDIA GPUs, by using the ability to save the OpenCL kernel binary using clGetProgramInfo. The binary is compiled using `nvcc` for the target architecture and the assembler dumped using `cuobjdump`. For AMD GPUs we use the KernelAnalyzer application which compiles OpenCL directly into machine assembler code. To get the final number of instructions, we break the kernel into instruction counts inside loops and outside loops. These instruction counts are denoted by the variable i. Using the low resolution workload estimate, we compute an average of the number of times each loop runs and denote these variables as n.

The total number of compute instructions is calculated as:

$$i_c = i_{ct}(a_t a_r + p_t) + i_{ci}(p_i + a_w t_v a_r) + i_{cco},$$

where the variables are described in Table 5.

The estimated ratio m is used both to divide between coalesced and uncoalesced memory instructions and to measure cache performance. This is possible due to the similarity of probability to have a coalesced memory access and a cached memory fetch. The total number of memory instructions

Instruction count variables	
i_c	Total number of compute instructions
i_{ct}	Compute instructions per traversal
i_{ci}	Compute instructions per intersection
i_{cco}	Constant compute instructions
i_m	Total number of memory instructions
i_t	Initial number of memory instructions
i_{mt}	Memory instructions per traversal
i_{mi}	Memory instructions per intersection
i_{mco}	Constant memory instructions
i_{mu}	Uncoalesced memory instructions
i_{mc}	Coalesced memory instructions
m	Memory coalescing estimate
m^2	Memory cache performance estimate
p_t	Number of primary ray traversals
p_i	Number of primary ray intersections
t_v	Number of visible triangles
a_t	Number of AO node traversals
a_r	Number of AO rays
a_w	AO intersection cost
a_m	Total AO memory instructions
a_{ca}	AO cache performance estimate

Table 5: Compute and memory instruction count variables.

are calculated as follows:

$$\begin{aligned}
i_t &= (i_{mt}p_t + i_{mi}p_i + i_{mco})(1 - m^2), \\
a_{ca} &= 1 - \min(1, 2^{-(\frac{a_t+p_t}{23}-1)}), \\
a_m &= a_{ca}(i_{mt}a_t a_r + \frac{t_v i_{mi} a_r}{2}), \\
i_{mu} &= i_t(1 - m) + a_m, \\
i_{mc} &= i_t m,
\end{aligned}$$

The total number of memory instructions is $i_m = i_{mc} + i_{mu}$.

4.4 Tuning Ray Tracing Parameters

The GPU model is used as a feed forward controller which updates itself iteratively to find the best fitting parameters and then sends those to the actual ray tracer. The error is calculated between the current model estimated frame time and a target reference frame time and ray tracing parameters are adjusted to improve the error. Since the simulated ray tracer and model executes quickly they update several times within one frame of the ray tracer, hence it is possible to recover from sudden changes in view direction. As an example, viewing a plain wall with few node traversals and triangle intersections, the model adapts by increasing the number of AO rays. If the view is turned around 180 degrees to view some more complex geometry the actual frame rate would drop severely, but since the model updates several times before the frame is rendered the number of AO rays are matched so that frame rate stays constant. Ray tracing features that are adjusted to match the reference frame rate are shadow rays and the number of AO rays.

Figure 1 illustrates how the GPU model is utilized to tune the ray tracer as a feed forward controller. The model control loop runs several times per frame and iteratively updates x until the error e, from model output y to reference r, is as small as possible. Changes in x result in switching shadow and AO on or off and adapting the number of AO rays so that a fixed frame rate is maintained.

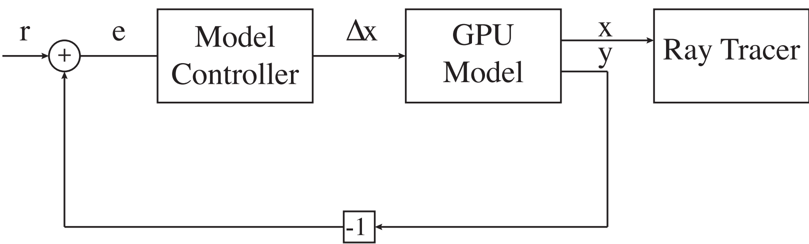


Figure 1: Auto tuning the ray tracer using the GPU model and a feed forward controller.

The feed forward model simulates the ray tracer with some error due to unmodeled behaviours. This error can be greatly reduced with help of a slow outer feedback loop. A controller compensates the model by comparing the model predicted execution time with real ray tracer execution time. In figure 2 the inner loop containing the GPU model also receives real execution time y_2 and a feedback controller compensates for the error e_2 between model execution time and real execution time reducing the error from model y_1 to reference r further.

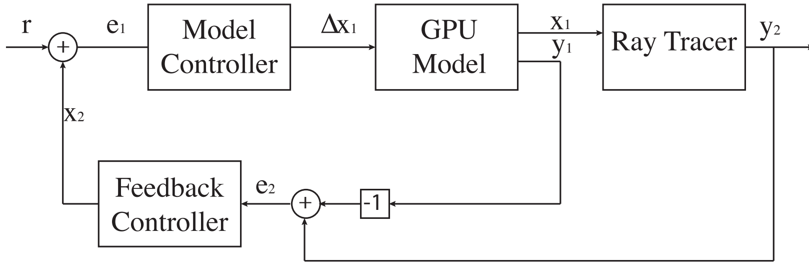


Figure 2: Model errors are improved by introducing a slow outer feed back loop.

5 Results

We auto tune ray tracing parameters for three different GPUs, namely, NVIDIA Geforce 580, NVIDIA Geforce 8800 and the AMD Radeon 5870. We use two data sets, a fairy scene shown in Figure 3 and a cabin scene shown in Figure 4. With our implementation of surface area heuristics BVH the fairy scene contains 174,117 triangles and requires a BVH depth of 28. The cabin scene contains 422,635 triangles and requires a BVH depth of 34. The increased complexity of the cabin scene results in more traversal iterations and triangle intersections resulting in longer rendering times. In particular on the Radeon 5870 the cabin scene results in ray tracing stack nodes spilling from the shader's on-chip registers out to global memory, resulting in slower performance.

Figure 3: Fairy scene.

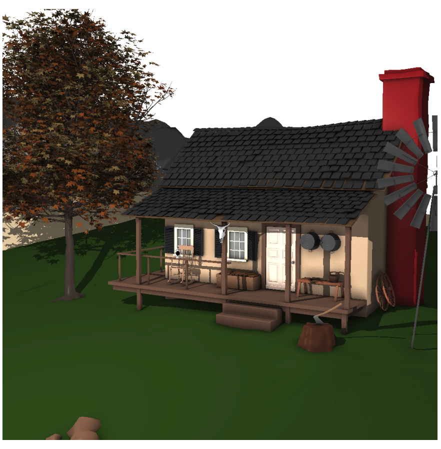


Figure 4: Cabin scene.

Figure 5 shows the frame time results of an animation of 100 frames of our two scenes. The times shown in the graphs are the actual measured frame time (red), the GPU model predicted frame time (green) and the outer feedback loop corrected model time (blue). For each GPU and scene a different reference frame time is set. This reference frame time is user selected and we set it to values to enable a reasonable number of AO rays. Lower reference times are possible, but if the frame time is too low, auto-tuning will switch to the lowest possible settings. The GPU model times follow the curve shape of the actual times accurately, but in some cases with a significant offset. The average model to execution time error for the fairy scene is 6.3, 4.8 and 2.1 percent and for the cabin scene it is 11.1, 12.8, and 19.9, for the three GPUs used. The GPU model alone does reasonable well with the fairy scene, but the error increases with the cabin scene. The offset between frame time and model time is removed when the outer loop feedback is used resulting in the new 'Model FB' estimated time. The average error of this improved feedback model time compared to the original execution time for the fairy scene is 1.1, 0.7, and 0.2 percent and for the cabin scene it is 1.7, 0.2, and 1.3, for the three GPUs used. Now with the feedback the error is reduced significantly for both scenes.

As a comparison we also ran the animation on both scenes on the GF 580 without any auto-tuning. Before measuring we manually tuned the first frame to execute at 100 ms. These settings were then kept during the animation. For both the fairy and the cabin scene the initial errors from target to real execution time were within 1 percent, however as the animation progressed the errors changed and for the fairy execution time increased and for the cabin execution time decreased. At the last frame

both scenes had an error close to 20 percent and the average errors over the animation are for the fairy 10.5 percent and for the cabin 13.2 percent. These measurements are only a comparison for this animation since without auto-tuning the errors can grow arbitrarily large depending on the initial view direction and tuning.

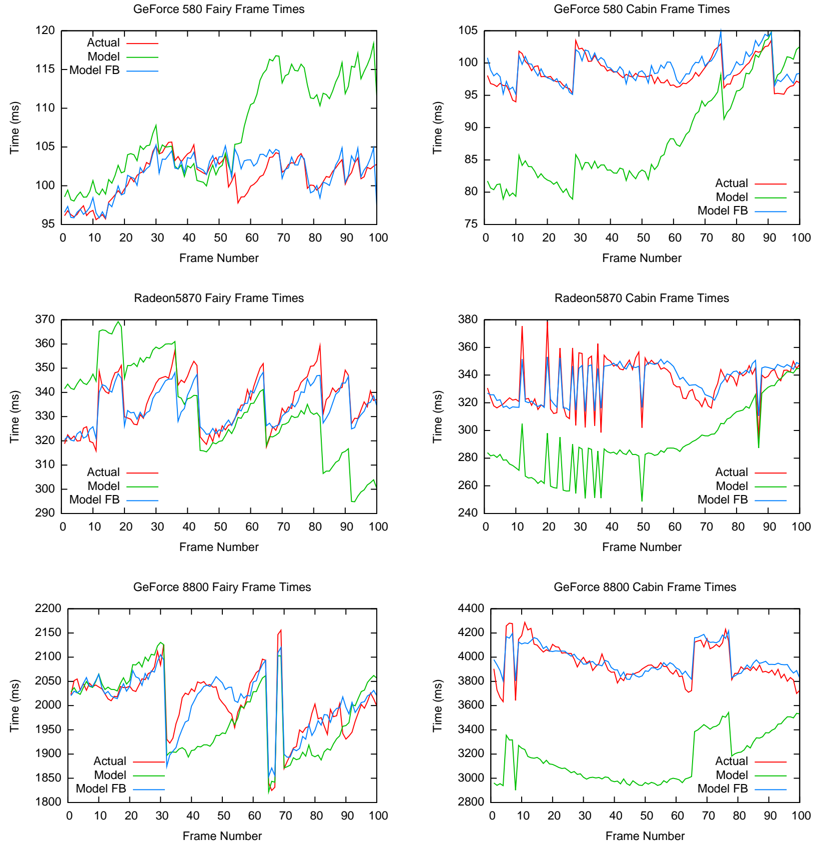
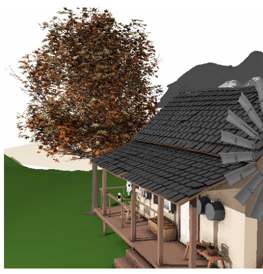


Figure 5: Frame times for different GPUs and scenes for a 100 frame animation. The reference time for the graphs for the fairy scene in the left column are 100ms, 333ms and 2000ms, and for the cabin scene in the right column, they are 100ms, 333ms and 4000ms.

Three frames from 1st, 50th and 100th frame of the two animations are shown in Figure 6 and Figure 7. Both animations rotate around the scene while moving in towards the center. The cabin animation starts with the tree filling a small part of the scene and finishes with the camera right next to the tree. When rendering the tree, the prediction of the rendering works well, due to the very constant distribution of primitives across the screen.

Figure 6: Frames from fairy animation.



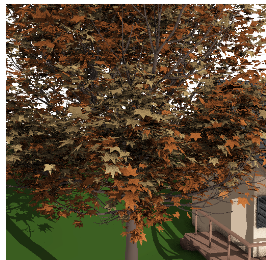


Figure 7: Frames from cabin animation.

Figure 8 shows the auto-tuned number of AO rays parameter, GPU model to execution time percentage error and the outer loop feedback corrected model percentage error. The number of AO rays is tuned in order to ensure the target frame rate as specified in Figure 5. The number of AO rays is similar across the different GPUs because the rendering time is determined by the current view point which is the same for each GPU. The GPU model error follows a similar curve for the two scenes even though different GPUs are used, showing that the model works well, but estimating the workload of ray tracing is still challenging for some views. This could be improved by increasing the resolution of our low resolution pre-sampling pass, which on multi-core CPUs would not affect the performance of the GPU rendering time if run in parallel with the previous frame rendering on the GPU. The outer feedback loop corrected time improves upon the original GPU model estimate by removing un-modeled behavior in the GPU.

6 Conclusion

We have used an analytical GPU model to tune a complex application, ray tracing, on a variety of GPU hardware. The model was originally designed

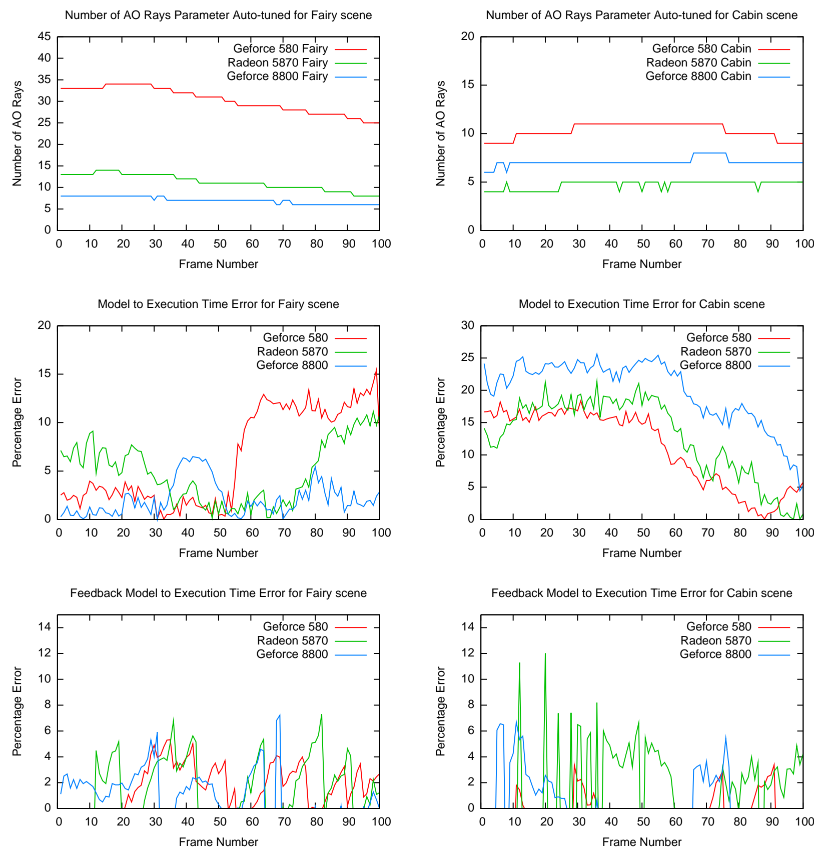


Figure 8: Auto-tuned number of AO rays, and error for different GPUs and scenes over a 100 frame animation. The left column is for the fairy scene and the column for the cabin scene. The top row graphs show the number of AO rays auto-tuned over the animation, the graphs in the middle show the percentage error between the GPU model and actual execution time and the bottom row graphs show the percentage error between the GPU model and the outer loop feedback corrected model time.

only to target NVIDIA GPUs but with its general construct we have managed to estimate performance on AMD GPUs as well.

Using the model and a feed forward controller we have shown that it is possible to estimate GPU workload and to tune a complex application using the workload information. We also introduced a slow outer feedback loop that can be used to improve the GPU models errors by compensating for unmodeled behaviors. Using this approach it is possible to estimate and auto-tune applications with different levels of complexity, given a model of

the application's run time.

We believe that our approach should be applicable to other complex compute applications as well. Performance tuning is possible as long as the number of instructions executed can be estimated for the application. Even if the problem space is large with many tuning parameters, the cost of executing the model is low and can easily be executed hundreds or more times per iteration.

Several areas for future work arise from our initial work. More ray tracing features such as reflection and refraction can be added to the ray tracer and their parameters auto-tuned. We modeled the Fermi shader cache architecture inside our workload estimates. A more general model of the cache architecture in the GPU model would make it useful for other applications. The GPU model could also be generalized to work with multi-core CPUs and Intel CPU SIMD extensions such as AVX.

Acknowledgements

We acknowledge support from the Intel Visual Computing Institute, Saarbrücken, Germany and the ELLIIT Excellence Center at Linköping-Lund in Information Technology. We thank NVIDIA and AMD for the generous donation of GPUs. Thanks to Andrew Kin Fun Chan and Dan Konieczka for allowing us to use 'The Cabin' model.

Bibliography

[1] Timo Aila and Samuli Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *High-Performance Graphics*, pages 145–149, 2009.

[2] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. *SIGPLAN Not.*, 45:105–114, January 2010.

[3] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[4] William Dally. Power Efficient Supercomputing. Accelerator-based Computing and Manycore Workshop (presentation), 2009.

[5] Andrew Davidson, Yao Zhang, and John D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, May 2011.

[6] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings ACM SIGGRAPH*, pages 247–254, 1993.

[7] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, 2009.

[8] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[9] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization Principles

and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[10] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

Paper II

Power Efficiency for Software Algorithms running on Graphics Processors

Björn Johnsson, Per Ganestam, Michael Doggett and
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.

EGGH-HPG'12 Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics
Pages 67-75, June 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 [9]. 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 [3]. 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 [5], and for CPUs, logic tends to use more power than caches [3]. 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 [9]. Recently, Esmaeilzadeh et al. [7] 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 [21]. 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 [2]. For GPUs, simulators can be used to model power and leakage [16], and different low-level hardware optimization techniques can be developed and studied [17]. Power gating techniques can also be used [18]. It is also possible to save energy by reducing the precision in the computations in the vertex shader unit [12, 13] and in the pixel shader cores [14]. 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 [10] 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

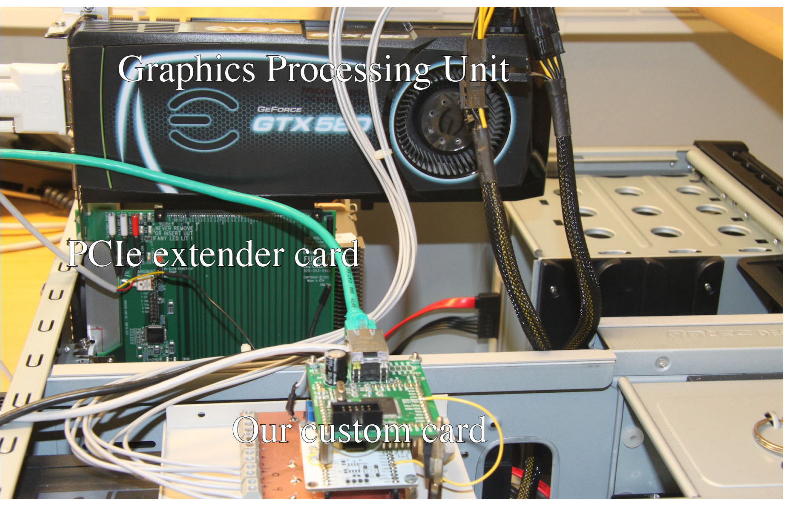


Figure 1: We have built a power measurement station, which measures power on the PCI express bus (which can deliver up to 75 W) and on the graphics card's two power connectors, which in this case can deliver up to 75+150 W. This sums to a max of 300 W.

rate goal has been reached.

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

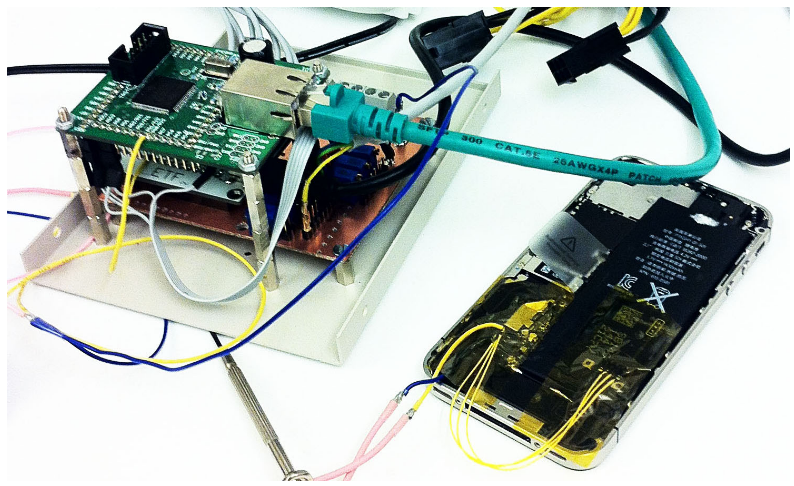


Figure 2: Our power measurement station, also seen in Figure 1, connected directly on the battery power connects on an iPhone 4S.

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 measured 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?

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

2. Is energy directly proportional to frame time?

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 [15], 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 4.

3.2 Case 2: Shadow Algorithms

As our second case study, we have chosen three different shadow algorithms, namely, shadow volumes (SV) [4], shadow mapping (SM) [19], and variance shadow mapping (VSM) [6]. 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) [1], 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 [20], 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 4.

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 4, 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

[2] In all fairness, it should be noted that NVIDIA recently released the Kepler architecture [11], 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.

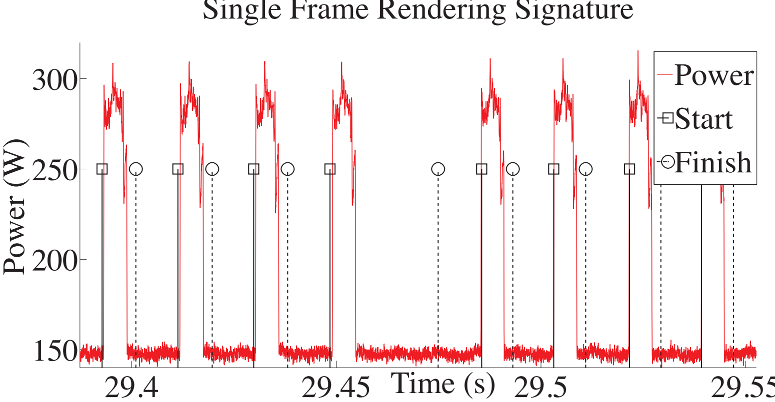


Figure 3: Power signature of eight frames of forward rendering on the AMD Radeon 7970, where the fourth frame is followed by a delay which also affects our time stamp measurements.

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 3. 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 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).

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

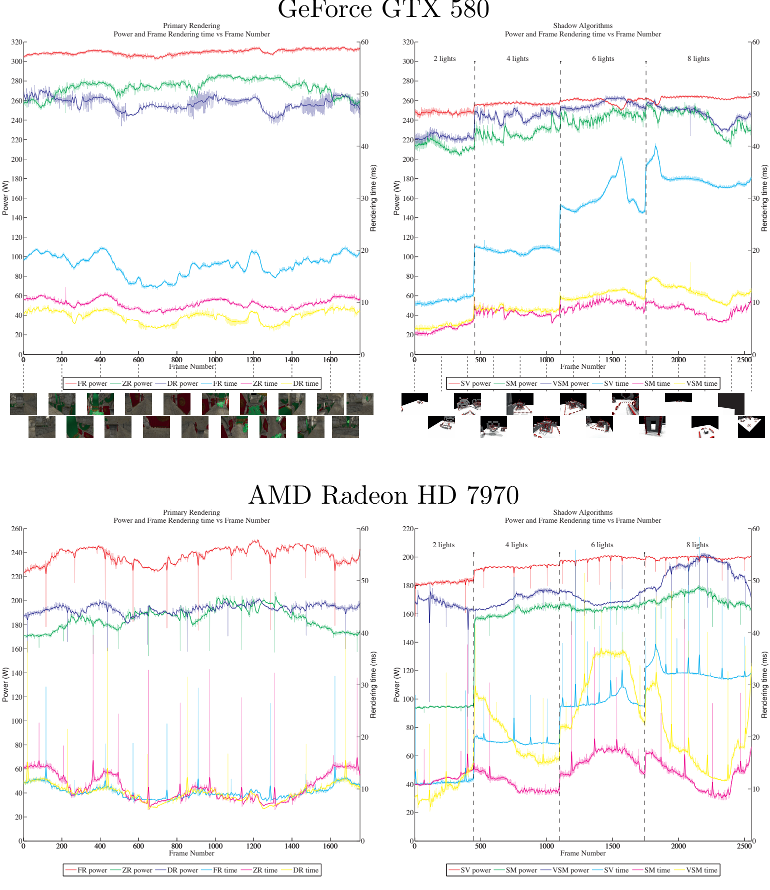


Figure 4: Frame times and power consumption for primary rendering (left) and the different shadow algorithms (right) on an NVIDIA GTX580 (top) and on an AMD Radeon HD 7970 (bottom). Abbreviations: FR (forward rendering), ZR (Z-prepass), DR (deferred), SV (shadow volumes), SM (shadow mapping), and VSM (variance shadow mapping).

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

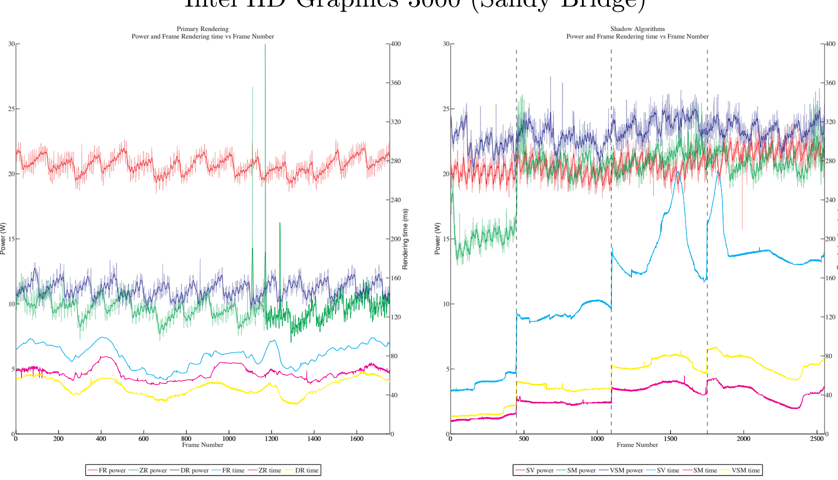


Figure 5: Frame times and power consumption for primary rendering (left) and the different shadow algorithms (right) on an Intel Sandy Bridge integrated graphics HD3000.

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. 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

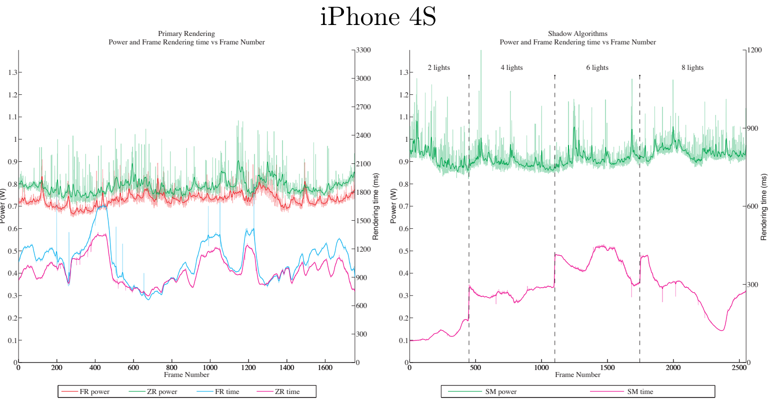


Figure 6: Frame times and power consumption for primary rendering (left) and the different shadow algorithms (right) on an iPhone 4S. Note that the frame time axis is different in the two graphs.

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 [8], which essentially performs a pre-Z pass in hardware before shading, and yet our ZR, which 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.

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:

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

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

	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.

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

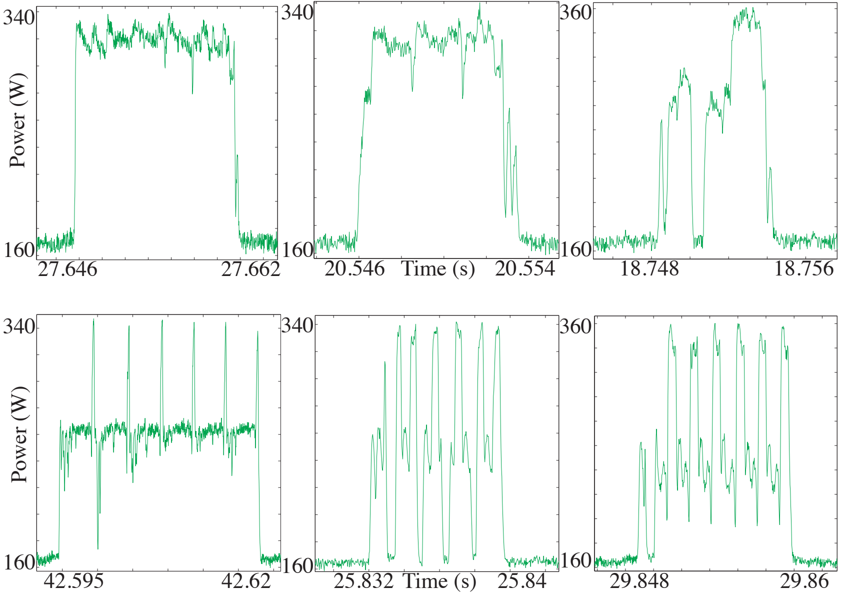


Figure 7: Single frame rendering power signature for the NVIDIA GeForce GTX580. The top row contains measurements for (from left to right) FR, ZR, and DR, while the bottom row shows SV, SM, and VSM. All shadow algorithms use 6 light sources.

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

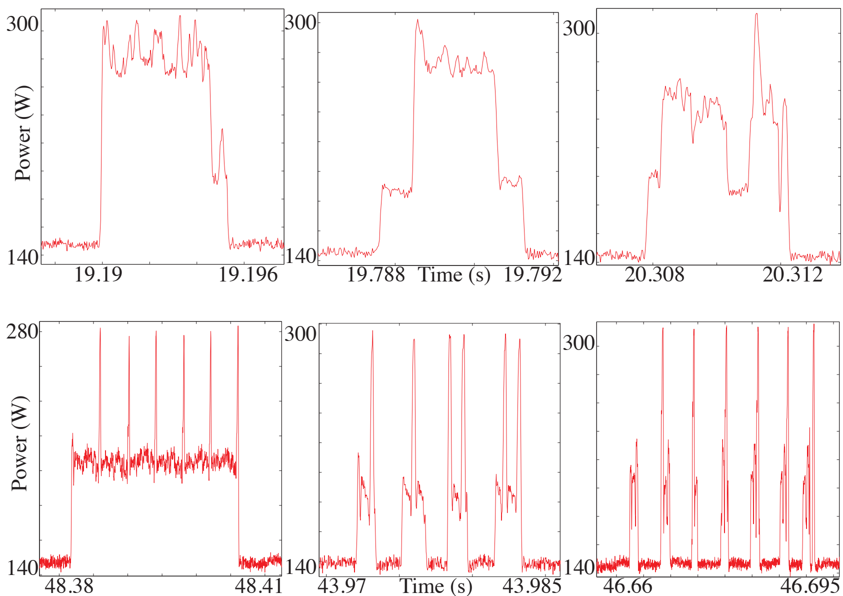


Figure 8: Single frame rendering power signature for the AMD Radeon HD7970. The top row contains measurements for (from left to right) FR, ZR, and DR, while the bottom row shows SV, SM, and VSM. All shadow algorithms use 6 light sources.

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.

Acknowledgements

We acknowledge support from the Swedish Research Council, Intel Visual Computing Institute, Saarbrücken, Germany, the ELLIIT Excellence Center at Linköping-Lund in Information Technology, and in addition, Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation.

Bibliography

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

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

[3] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Communications of the ACM,*, 54(5):67–77, 2011.

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

[5] William Dally. Power Efficient Supercomputing. Accelerator-based Computing and Manycore Workshop (presentation), 2009.

[6] William Donnelly and Andrew Lauritzen. Variance Shadow Maps. In *Symposium on Interactive 3D Graphics and Games*, pages 161–165, 2006.

[7] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *38th International Symposium on Computer Architecture*, pages 365–376, 2011.

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

[9] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro,*, 31(5):7–17, 2011.

[10] Raja Koduri. “Power” of Realtime 3D Rendering. In *Beyond Programmable Shading (SIGGRAPH course)*, 2011.

[11] NVIDIA. GeForce GTX 680. Technical report, 2012.

[12] Jeff Pool, Anselmo Lastra, and Montek Singh. Energy-Precision Tradeoffs in Mobile Graphics Processing Units. In *International Conference onf Computer Design*, pages 60–67, 2008.

[13] Jeff Pool, Anselmo Lastra, and Montek Singh. Power-Gated Arithmetic Circuits for Energy-Precision Tradeoffs in Mobile Graphics Processing Units. *Journal of Low Power Electronics,*, 7(2):148–162, 2011.

[14] Jeff Pool, Anselmo Lastra, and Montek Singh. Precision Selection for Energy-Efficient Pixel Shaders. In *High-Performance Graphics*, pages 159–168, 2011.

[15] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, pages 197–206, 1990.

[16] J. W. Sheaffer, D. Luebke, and K. Skadron. A Flexible Simulation Framework for Graphics Architectures. In *Graphics Hardware*, pages 85–94, 2004.

[17] 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*, pages 54–65, 2005.

[18] Po-Han Wang, Chia-Lin Yang, Yen-Ming Chen, and Yu-Jung Cheng. Power Gating Strategies on GPUs. *ACM Transactions on Architecture and Code Optimization*, 8(3):13:1–13:25, 2011.

[19] Lance Williams. Casting Curved Shadows on Curved Surfaces. volume 12, pages 270–274, 1978.

[20] Lance Williams. Pyramidal Parametrics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, volume 17, pages 1–11, 1983.

[21] Yan Zhang, Dharmesh Parikh, Karthik Sankaranarayanan, Kevin Skadron, and Mircea Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. Technical Report CS-2003-05, University of Virginia, March 2003.

Paper III

Real-time multiply recursive reflections and refractions using hybrid rendering

Per Ganestam and Michael Doggett

Lund University

Abstract

We present a new method for real-time rendering of multiple recursions of reflections and refractions. The method uses the strengths of real-time ray tracing for objects close to the camera, by storing them in a per frame constructed *bounding volume hierarchy* (BVH). For objects further from the camera, rasterization is used to create G-Buffers which store an image based representation of the scene outside the near objects. Rays that exit the BVH continue tracing in the G-Buffers' perspective space using ray marching, and can even be reflected back into the BVH. Our hybrid renderer is to our knowledge the first method to merge real-time ray tracing techniques with image based rendering to achieve smooth transitions from accurately ray traced foreground objects to image based representations in the background. We are able to achieve more complex reflections and refractions than existing screen space techniques, and offer reflections by off screen objects. Our results demonstrate that our algorithm is capable of rendering multiple bounce reflections and refractions, for scenes with millions of triangles, at 720p resolution and above 30 FPS.

The Visual Computer
Volume 31, Issue 10, Pages 1395-1403, September 2014

1 Introduction

Reflective and refractive objects are an important component of reality found in ray traced imagery, but rarely found in real-time rendering. When these objects do appear in real-time rendering, they typically only demonstrate a single bounce using rendered or pre-rendered environment maps. These reflective and refractive objects can relay to the viewer information about the composition of the scene, such as what is hiding behind an object, or what can be seen through refractive objects.

For example, in modern real-time games, being able to see movement behind the player in a mirror would add to the gameplay experience. Also, real-time reflections are listed by Andersson [1] as a major challenge for real-time rendering. Modern real-time rendering scenes have a high triangle count, and most triangle meshes are highly detailed. Storing this data can take a large amount of memory, and computing complex visibility with reflections is quite challenging, if all geometry is taken into account.

The contribution of this paper is a general framework that enables real-time reflection and refraction rays to traverse multiple bounces, while running on graphics hardware. Our method enables complex reflections and refractions with multiple recursions to be computed within a region near the camera, and more limited interaction in the remainder of the scene. We use a hybrid approach that starts with a rasterization of the scene to compute primary ray hit points. We also generate a cube map of G-Buffers, that store depth, color, normal and material, creating an image based representation of the scene from the camera's view point. In the area close to the camera, a bounding volume hierarchy is constructed every frame to enable fully deformable objects. For the primary ray hit points that require further tracing, rays are traced into the BVH, and the G-Buffers. To trace rays in the G-Buffer we present a new approach to ray marching in the scalable, geometry insensitive G-Buffer that represents an entire scene. Rays traced into the image-based G-Buffer that intersect with reflective objects, can spawn new rays that trace back into the BVH volume or into other G-Buffers. The different types of rays that are traced are illustrated in Figure 1. An important objective of our system is to integrate complex viewing rays into large scenes, e.g. where complex foreground reflections and refractions are integrated with non-reflective, faster to render, backgrounds.

2 Related Work

Real-Time reflections and refractions have long been possible using environment maps and graphics hardware using the technique introduced by Blinn and Newell [2]. This method is limited to a single bounce and so immediately loses much of the realism that reflections and refractions provide.

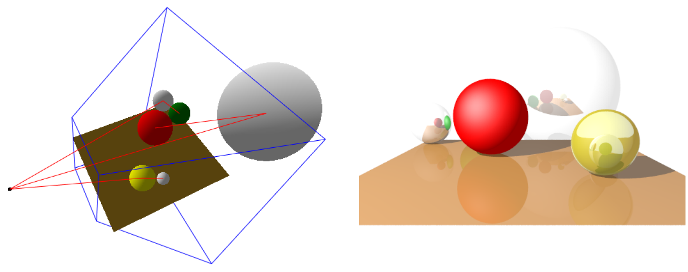


Figure 1: A scene showing the type of complex reflections possible in real-time using our system. The left image shows the camera and it's viewing frustum from a distance and the right shows the camera image. On the left three reflection and refraction ray paths are shown as red lines. The top ray reflects off a mirror sphere and reveals a green sphere that is occluded in screen space. The middle reflection bounces off a large mirror sphere that is behind our BVH region close to the camera, but the viewing ray is still traced back into the BVH to show the back of the red sphere. The lower ray traces through a refractive yellow sphere and is then reflected back into the yellow sphere. All these complex ray paths are not possible with existing screen space techniques.

True photo realism requires more complex rendering of visibility to capture realistic reflections and refractions. Realistic reflections and refractions can be achieved using ray tracing [24], but ray tracing needs to be integrated carefully into real-time systems in order to ensure high performance.

Real-time rendering in games has used environment mapped reflection and refraction for many years. Recent game engines such as Unreal Engine [3] supports features such as billboard reflections, where imposters are used to improve the accuracy of reflections.

More realistic screen space reflections are created in CryEngine 3 [19] using ray marching in screen space to create accurate reflections, but are limited to objects which appear in the view frustum. Recent research on using non-pinhole cameras for reflections is presented by Rosen [16].

Wyman [25] makes real-time refraction look more realistic by also representing a second surface. Sun et al. [20] present a technique for simulating light transmission through refractive objects using the GPU, but at much lower frame rates than we target in this work.

Our approach also captures the surrounding scene by rendering a cube map, similar in nature to image based methods such as presented by McMillan [11] where images are warped to create the final rendering. Hakura and Snyder [4] present a hybrid system that uses environment maps and ray tracing to generate realistic reflections and refractions. Their system creates multiple

environment maps from different directions and distances from the reflective or refractive object in a preprocessing stage that would take significantly more time than our method, which creates maps of the entire scene's environment using the rasterization pipeline, which is significantly faster, even without modern graphics hardware. Our algorithm also improves accuracy by tracing rays at each pixel.

Cube maps have been used extensively in real-time rendering to capture lighting. Games, such as Half-Life 2 [12], assign individual cube maps to each object to create local lighting. Sebastien et al. [18] present techniques for recent games that solve the parallax issues present in the cube map technique. Szirmay-Kalos et al. [21] render a cube depth map, similar to our G-Buffer Cube Map used here, and uses it for parallax corrected access to lighting in the environment map. Their approximation to the intersection point is calculated using the point where the cube maps where created from, and the currently intersected point. In our work we use similar cube and depth maps, but use ray marching to compute accurate intersection points and trace secondary rays from those points, rays that also traverse the depth map to enable much more accurate visibility from our cube map. Knecht et al. [8] also use G-Buffers to capture illumination and relight reflective and refractive objects.

In recent work, Mara et al. [10] use a two-layer deep G-Buffer to achieve low frequency lighting effects. They also show how to use their deep G-Buffer to compute mirror reflections. Although their two-layer G-Buffer captures objects hidden to the viewer (such as objects behind a wall), the reflected object still has to reside within the view frustum. Our approach offers reflective rays in any direction, even opposite to the view direction, and with a higher quality if the reflected object is located within the BVH region.

We use a screen space approach based on deferred shading [17] to find the hit points of the primary rays and then use real-time ray tracing to trace secondary rays through the foreground objects. There is a great deal of work in real-time ray tracing which will not be reviewed here that includes data-structure construction such as BVHs and kD-trees [27] and optimizations of ray tracing performance, but this paper focuses on combining ray tracing with image based rendering to compute complex reflection and refraction effects. Recent work on real-time ray tracing on GPUs [6], using voxelization and A-Buffers to create a representation of the scene, is capable of global illumination at interactive rates using a full GPU pipeline. The algorithm presented targets higher frame rates and resolutions in order to fit more easily into modern real-time rendering engines.

Interactive global illumination attempts to accurately model the interaction of light and matter by rendering frames in less than a second. Ritschel et al. [15] survey the current state of the field and we take a few highlights from that area that are related to our current work.

Screen space techniques improve upon basic environment mapping by using the representation of objects in the scene in screen space, but still screen space does not handle objects occluded from the view point or objects outside the view frustum. Reinbothe et al. [14] voxelize the entire scene so that ambient occlusion can be more accurately calculated in screen space.

Thiedemann et al. [23] also voxelize the scene to avoid illumination errors. Ritschel et al. [15] point out that interactive global illumination approaches approximate the geometry and lighting in the scene to reduce the complexity, because low frequency representations are sufficient for lighting. For accurate reflections and refractions, accurate geometry is required and for this we use real-time BVH construction. Recent improvements in accurate indirect illumination using BRDFs [26], demonstrate future directions for improving the image quality of our work, but since their performance is limited, they are beyond the scope of this paper.

3 Algorithm

Our rendering algorithm is based on a hybridization of existing rasterization and ray tracing techniques. It balances visual quality and performance in such a way that multiple bounce reflections and refractions of complex scenes are possible in real-time on current graphics hardware.

Distant geometry in the scene is represented by image-based maps that reduce scene complexity, but still allow rays to recursively reflect and even refract if the refraction goes to a sky box. The maps are a set of six G-buffers arranged in a cube map style. Geometry close to the view camera is represented in a BVH that is traversed with a real-time ray tracer. In order to facilitate fully dynamic scenes in real-time, a full rebuild of the BVH around objects close to the camera is performed each frame. An overview of the partitioning of the scene geometry is shown in Figure 2.

Each frame of our algorithm performs the following steps:

1. Rasterize primary visibility
2. Render a G-buffer cube map
3. Build the BVH of geometry near the view camera
4. Perform primary shading and generate secondary rays
5. Recursively traverse the BVH and G-buffer cube map

These steps are further explained in the following sections.

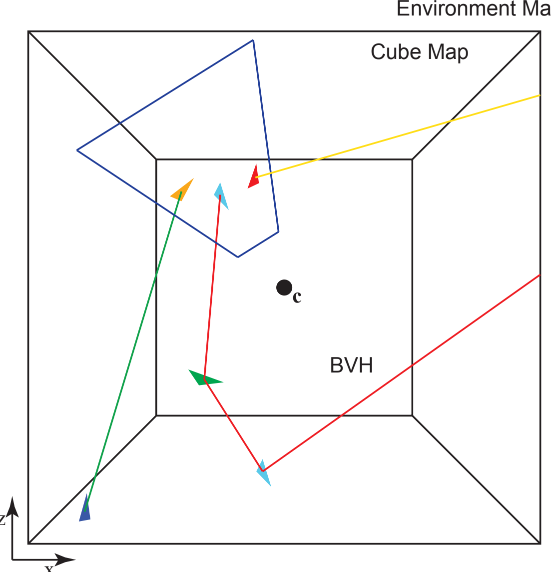


*Figure 2: An X-Z 2D diagram of our rendering setup shows the different regions we break the scene into and how they are represented for rendering. Objects near to the camera (**c**) are inside a BVH. Objects further away are rendered into 6 G-Buffers that are stored as a Cube Map. Objects beyond the Cube Map will be represented by the typical sky box in an outer Environment Map. The Cube Map faces are rendered with the camera at the center point **c**. A blue trapezoid shows the view frustum where three paths are traced from the primary view G-Buffer into the BVH and intersect with objects in the BVH (red ray), objects in the Cube Map (green ray) and going through two Cube map faces (yellow ray). The red ray shows an example where a ray can trace in and out of the Cube Map and BVH regions.*

3.1 Primary Visibility

The first pass of the algorithm is the same as the first step of a standard deferred renderer. Primary visibility depth values and normals are stored in 2D textures that are used as a G-buffer. The only variation with our G-buffer compared to one commonly used in deferred rendering methods is that a material index rather than a specularity value is stored in the alpha component of the normal texture. These material indices are later used as material identifiers by the ray tracer.

3.2 The Cube Map

The cube map in our algorithm is an image-based data structure that is updated once every frame. The cube map is used to reduce scene complexity for ray tracing by rasterizing distant geometry. It stores the same kind of G-buffers as the primary visibility pass does, but instead of having one G-buffer, an individual G-buffer is stored per cube face. The cube origin is set to the view camera's world space position and the near planes of the cube map cameras define the boundary between BVH ray traversal and image-based ray marching. To avoid issues when a ray travels parallel to the diagonal planes of the view frustums of the cube map camera faces, the cube map cameras' field of view (FOV) is slightly wider than 90 degrees. In our implementation the FOV is set to 90.2 degrees (Figure 3). By widening the FOV, a ray existing in one of the cube's diagonal planes will always belong to at least one of the cube sides when entering the cube map. It is not important which side a ray belongs to, the first side a ray is tested positively against is chosen for ray marching.

3.3 BVH Construction

Our BVH implementation builds an LBVH, the linear BVH approach by Lauterbach et al. [9], where tree construction is reduced to a sorting problem. Parallelism is further improved by applying a tree and axis aligned bounding box (AABB) construction algorithm similar to the one by Karras [7].

Only geometry residing inside the cube defined by the cube map cameras' near planes is represented in the BVH. Before the BVH is constructed, a per object culling pass is performed. If an object's AABB overlaps with the cube then all triangles of that object are transformed to world space and a second culling pass is executed. The second pass performs per triangle culling. This is motivated since any ray that leaves the BVH boundary will enter the cube map and not continue in the BVH, even if the BVH contained triangles from partially overlapping objects. By culling triangles from objects partially overlapping the BVH region, no time is wasted on building a tree that includes triangles that would never be intersected anyway. Per triangle culling also enables a more balanced BVH of the triangles that actually are inside the BVH region. Since a lot of geometry is represented in the cube map, the BVH becomes much smaller, with the benefits of both reduced construction time and faster ray traversal.

By introducing a small overlap of the BVH and the cube map the possibility of a gap at the boundary due to precision errors is removed. In the overlap, an intersection occurs either in the map or in the BVH.

It is also possible for objects to have individual pre-computed BVHs, rather than a per-frame full BVH rebuild, and that rays intersecting an object

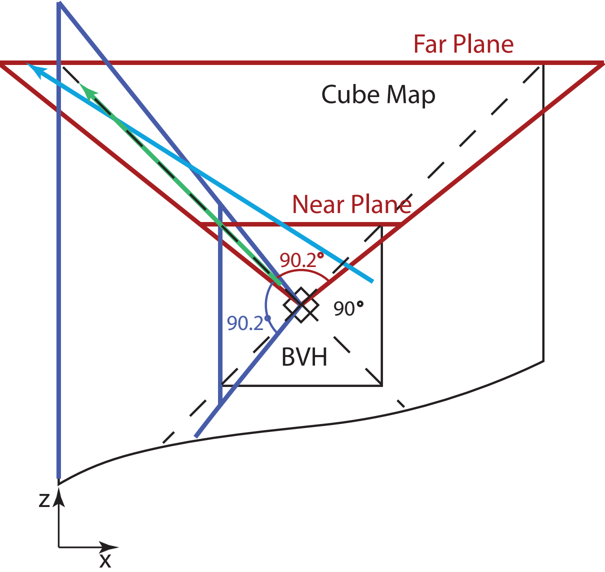


Figure 3: It is always possible to get a ray that goes along the diagonal planes (green ray) that separate the 90 degrees frustums of the cube map's faces. A drawback of having the view camera centered in the cube is that this is a frequent event. In our approach we widen the FOV of the cube map cameras so that the green ray always belongs to one or the other cube map side, and once the side to traverse is picked, the ray will continue in that side and not risk repeatedly switching sides. The blue ray displays a case where the ray first only belongs to the red frustum and then enters the overlapped region. Entering the overlapped region doesn't mean that the ray will switch side. The blue ray continues in its current side for as long as it is within the red frustum. A ray aligned with the diagonal plane of the red view frustum would only be able to switch to the blue frustum once, since the new switching planes would then be the blue view frustum's diagonal planes.

are transformed to the local frame of the object before continuing traversal. However, this method only works when applying rigid body transformations. Our method is capable of handling any types of transformations, as an example, procedurally animated meshes.

3.4 Ray Tracing: BVH and Cube Map Traversal

Our ray tracer approximates Whitted ray tracing by letting rays recursively traverse through the two different data structures, the BVH and the cube map. In fact, the cube map can itself be considered an approximation of a BVH. And as it is an image based data structure, once rasterized, ray

traversal time in the cube map is constant in relation to scene complexity. The size of the BVH and cube map can be chosen arbitrarily and as the BVH is increased in size, to cover a larger part of the scene, our method converges towards a complete Whitted ray tracer.

Before ray tracing begins, the view camera G-Buffer is used to compute shading of the primary intersections and to determine whether a visible object should generate secondary rays or not. A spawned recursive ray continues to traverse the scene until it hits a diffuse surface, exits through the far planes of the cube map (and intersects a sky box in an outer environment map), or the maximum recursion depth is surpassed. The first recursion of a ray may start either in the BVH or in the cube map. Where it starts is simply decided by testing if the origin of the ray is inside the BVH bounding box or not. A recursive ray is initially defined in world space as

$$r_w(t) = o_w + d_w t, \;\; \{o_w, d_w\} \in \mathbf{R}^3, \;\; |d_w| = 1, \;\; t > 0.$$

If a ray currently traversing the BVH doesn't intersect any geometry, then it is instead sent to intersect with the world space representation of the cube map cameras' near planes. Given an intersection in the cube map side $i \in [1, 6]$ at the parameter value $t_i > 0$, the two points

$$p_0 = r_w(t_i) \;\; and \;\; p_1 = r_w(t_i + \varepsilon)$$

can be computed along the ray, where the offset $\varepsilon > 0$ is a small value that extends the ray slightly into the cube map side. By multiplying the points p_0 and p_1 with the view projection matrix M_i given by the camera that corresponds to cube map side i, the transformed points

$$p_0' = M_i p_0 \;\; and \;\; p_1' = M_i p_1,$$

are computed and further used to define the map ray used for ray marching as:

$$r_m(t) = \underbrace{p_0'}_{o_m} + \underbrace{\frac{p_1' - p_0'}{|p_1' - p_0'|}}_{d_m} t$$

with the components of o_m and d_m in the range $[-1, 1]$.

Rays that leave the BVH and enter the cube map use a similar ray marching technique to that of per-pixel displacement mapping [5] and Parallax Occlusion Mapping [22]. An important difference is that where previous ray marching techniques expect an orthogonal height map to traverse, in our method, each face of the cube map is represented in perspective. A second difference, and an result of the perspective projection, is that a ray can exit one face of the cube map but still be an active ray, and enter a neighbouring side of the cube map. A ray can also bounce between the cube map and the

BVH and back again, as many times as the recursive traversal needs before reaching one of the terminating conditions.

If the map ray, r_m, currently traversing the cube side i doesn't intersect anything in the map, then there are five alternatives to exit the cube map sides for continued traversal and one alternative which would terminate the ray. If the z-component $r_{mz} > 1$, then the ray exits through the far plane of the cube and is sent to intersect the sky box as a final traveral step. If the ray traverses in the negative z-direction and $r_{mz} < -1$, then traversal continues in the BVH, using the original ray r_w. The other four alternatives are when the ray exits through the x- or y-axis and continues in the cube map side j. Given the side i and the map exit condition it is possible to directly pick the side j to traverse. Before the ray can continue in cube side j, r_m is transformed back to world space and further transformed to the cube side j's space using the jth view projection matrix. The matrices from any side i to any other possible side j can be precomputed to speedup the transition from one cube side to another.

The sampling rate n while traversing a cube map side depends on the angle between the normal N_w of the intersected plane and the ray direction r_w and is computed in a similar way as it is done by Tatarchuk [22], $n = n_{min} + N_w \cdot r_w(n_{max} - n_{min})$. However, once the ray is transformed to normalized device coordinates (from r_w to r_m) the transformed normal will always be directed along the z-axis and $N_m \cdot r_m$ simply becomes the z-component of r_m.

To avoid stretching artifacts when a ray that is close to parallel to the current cube map side intersects an object, and since the cube map only stores one layer of depth values, objects represented in the cube map can be considered to be thin or thick. The thickness value of an object is proportional to the amount of stretching permitted by that object when intersected in the cube map. If a ray, currently traversing the cube map, intersects an object that is considered thin, instead of stretching the object, the ray simply misses and continues directly to the sky box. Whether an object is thin or thick is a per material property which can be chosen arbitrarily by an artist. The thickness value ranges between 0 and 1, where a thickness of 0 represents a perfectly thin object and a thickness of 1 represents an object that stretches to the far plane. Smaller moving objects seem to visually benefit from being considered rather thin, and static objects, such as walls (which shouldn't let rays pass behind them anyway), should preferably be considered thick. Refractive objects in the cube map are always considered thin and once intersected, ray traversal is cancelled and the refracted ray is sent to the sky box.

3.5 Shadows and Deferred Rendering

It would be possible to compute accurate ray traced shadows inside the BVH, complying to the restriction that the light sources also reside within the BVH. If the light sources and thus possible occluders are positioned outside the extent of the BVH it is no longer possible to guarantee correct shadows using conventional ray traced shadow rays. However, since our method is highly compatible with the deferred rendering pipeline, it is straight forward to incorporate shadow maps, or any other effect or post processing filter commonly used with deferred rendering, to our method. Computing shadows using shadow maps has an insignificant impact on rendering performance.

4 Results

We implemented our method using OpenGL and CUDA 5 on a 32-bit Windows 7 PC with an Intel Core i7 and an Nvidia GeForce 680, and tested it on several scenes. For the larger San Miguel scene, we used 64-bit Windows and an Nvidia Quadra K5000 to generate the full BVH ray traced images. Figure 4 shows two images from each of our test scenes in comparison to an accurately rendered image, using a BVH for the entire scene, and a colored coded image that shows the size of the BVH near the camera. The San Miguel model is from PBRT [13]. The San Miguel scene uses only per triangle frustum culling, since per object frustum culling is not possible because of its file format. We would expect much better performance if per object culling was implemented as it is for the other scenes, even so, our method still manages to render The San Miguel scene at an average speed of 10 frames per second.

Figure 5 shows the frame time for rendering a 200 frame sequence in the Sponza-Buddha-Bunny model. Each frame is broken down into the CUDA kernels that are used for rendering each frame. The breakdown shows that the ray tracing kernel is the dominant part of rendering, with the cube map rasterization of the scene taking little of the overall rendering time. The results show that on average our algorithm is four times faster than using a BVH for the entire scene.

The Chess scene is considered a pathological case when it comes to computing approximated reflections. This is because of its many reflective convex objects (288 chess pieces and 9 spheres) where many of them reflect and interreflect each other. Yet, our method accurately computes reflections nearby the view camera and successfully approximates reflections far away. This can be compared to what is possible in, as an example, unreal engine

[3], where reflections of dynamic objects are only achieved in screen space (and only one recursion is possible), and off screen reflections have to be pre-computed and stored in reflection environment maps. Since only static objects are visible in unreal's reflection environment maps, but all objects in the Chess scene are dynamic, no reflections at all would be possible from the reflection environment maps. Figure 6 shows the performance of our method for the Chess scene compared to the full BVH version. The results show a similar characteristic for both methods as the rendering time is dominated by ray tracing for both our method and the BVH version. But our method always shows significant improvement in performance due to the use of the cube maps for storing the scene.

To fairly compare the performance of our method versus a full BVH ray tracer, we have chosen to rasterize primary visibility in both methods, which brings the full BVH ray tracer closer to real-time performance. Even so, our method always performs better than a full BVH ray tracer.

4.1 Limitations

While accurate reflections and refractions are achieved inside the BVH, this is not possible in the cube map. The G-Buffers only represent a single depth value without any thickness. So objects that have some thickness, details that are behind the front or objects that are hidden behind this depth value are not represented in the depth map and appear missing in some rays. This results in some artifacts, but our BVH close to the camera ensures that the artifacts are in distant geometry and are only present for secondary rays. Even with this limitation, the resulting images in real-time applications have a more realistic look when rendering multiply recursive reflections and refractions.

The first row of figure 7 presents a minor artifact in the Chess scene where the reflected chess piece on the board is slightly stretched (dark pixels to the right of the chess piece) due to its thickness value and that the reflected rays' origins aren't shared with the cube map cameras' origin.

Another artifact, also displayed in figure 7, is when an object is covering a reflective object in the cube map. This artifact has a lower probability to appear near the camera (and can't appear inside the BVH) than further away, due to the increased possibility of having objects covering each other in the cube map the further away they are represented. A reflected ray can detect that it is behind another object, but there is no information about what to intersect, and as a fallback, the ray is sent to do a look-up in the sky box.

Rendering performance is greatly affected by the type of materials in the scene and also by the size of the BVH. If a scene contains many reflective and refractive materials, performance is naturally reduced due to the high

recursive ray count. The reduced performance in scenes containing a lot of reflective materials can be mitigated by adapting the size of the BVH, thus a trade off between performance and image quality is made.

A carefully sized BVH is, in some cases, vital to minimize the presence of possible artifacts using our method. One artifact that would be too interfering had it not been pushed to the background by a, for this scene, suitably sized BVH is displayed in figure 8. The refractive objects (water pitcher and glasses) in the back of the San Miguel scene are only stored in the cube map, and thus no information about what is behind them exists. Instead of computing accurate refractions, a typical real-time refraction approximation is used where the objects are considered thin and rays simply refract only once and do a look-up in the cube map.

The G-Buffers require a reasonable amount of memory. If needed, the G-Buffers may be rendered at a lower resolution in order to reduce memory usage. However, a reduced G-Buffer resolution would also affect image quality.

5 Conclusion

We have presented an approach for rendering multiple bounce reflections and refractions in real-time using rasterization and ray tracing on modern graphics hardware. Our technique is capable of rendering objects typically not seen in previous real-time screen based techniques at real-time rates of between 30 and 60 FPS for 720p images. Since the BVH can be arbitrarily sized, our technique is highly customizable to scene or performance requirements.

Since our approach is highly compatible with current rendering approaches, such as deferred rendering, we hope it will impact future applications and enable new types of interactions and improved visibility in real-time rendering.

Acknowledgements

To ELLIIT and Intel Visual Computing Institute for funding. Thanks to TurboSquid artist cjx3711 for the chess piece models.

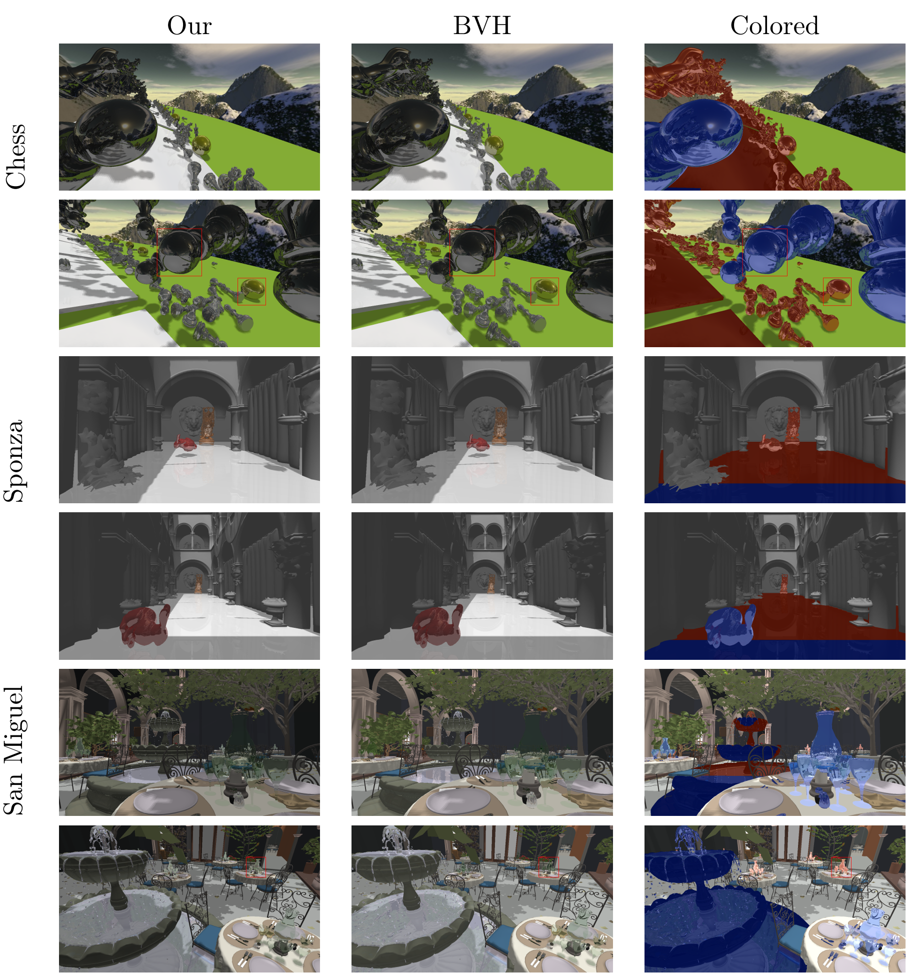


Figure 4: Three test scenes rendered from left to right with our algorithm (Our), using the BVH only (BVH), and with geometry inside the BVH colored blue and geometry in the cube map colored red (Colored). For the colored image, only pixels that contain reflective material that starts a ray are colored according to which area of the scene the ray is started in. The number of triangles for each scene is Chess 2,149,944, Sponza Buddha Bunny 1,354,743 and San Miguel is 10,500,551.

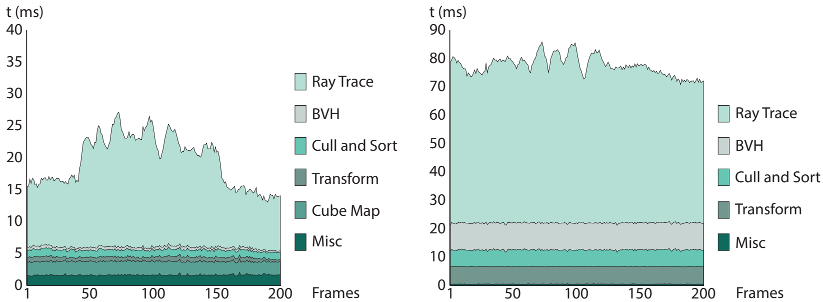


Figure 5: Rendering time breakdown of the 200 frames Sponza-Buddha-Bunny animation. Our method to the left compared to full ray tracing to the right. The scene is rendered with a maximum of four ray recursions. The improved performance is mostly due to the improved ray traversal in the cube map over the cost of BVH ray tracing. For this scene the BVH has been optimized to give high image quality and good performance. But since the size of the BVH used is chosen arbitrarily, performance is dependent on this trade-off, and the scene.

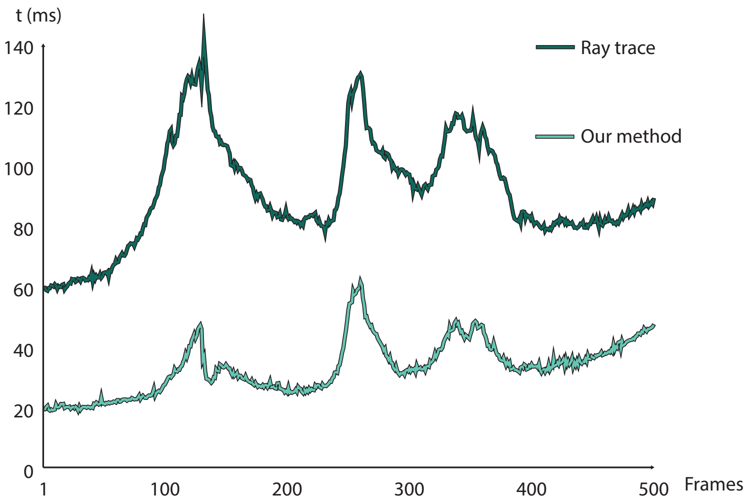


Figure 6: Performance comparison for the Chess scene between our method and using a full BVH for the scene. Both methods rasterize primary visibility. Our method always out performs full BVH ray tracing by being at least twice as fast and up to 4 times faster.

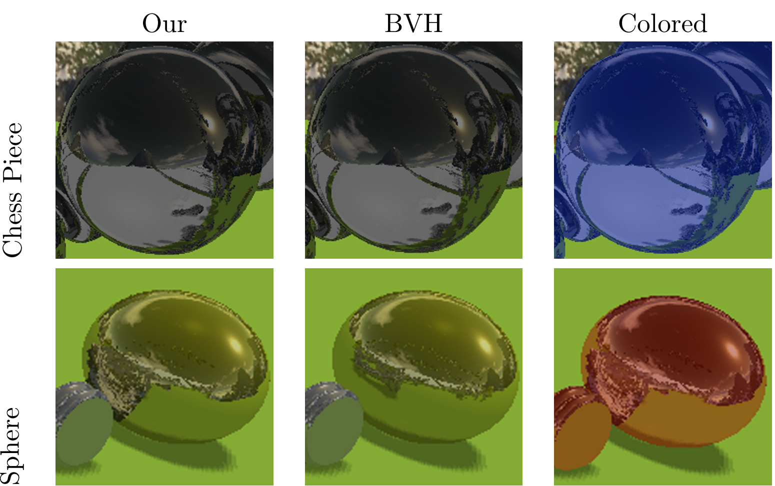


Figure 7: In the first row a minor stretching artifact is visible in the reflected chess piece on the board (visible as darker pixels to the right of the chess piece). The second row displays an artifact where an object (chess piece) is covering a reflective object (sphere) in the cube map and thus important scene information between the two objects is lost. Neither of the two artifacts can take place inside the BVH and so only occur in the background where the G-buffer cube map is used to store scene information. The magnified regions can be located by red boxes in the Chess scene images in figure 4.

Figure 8: In our method, refractive objects residing in the cube map cannot truly refract incoming rays due to the lack of information behind them, and as a fallback method a typical real-time refraction method is used, where rays simply look-up a value in the cube map. The magnified region presented can be located as red squares in the results image (figure 4) of San Miguel.

Bibliography

[1] Johan Andersson. Five Major Challenges in Real-Time Rendering. In *Beyond Programmable Shading course*, SIGGRAPH, 2012.

[2] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542–547, October 1976.

[3] Epic. Epic games unreal engine.

[4] Ziyad S. Hakura and John M. Snyder. Realistic reflections and refractions on graphics hardware with hybrid rendering and layered environment maps. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 289–300, 2001.

[5] Johannes Hirche, Alexander Ehlert, Stefan Guthe, and Michael Doggett. Hardware accelerated per-pixel displacement mapping. In *Proceedings of Graphics Interface 2004*, GI '04, pages 153–158, 2004.

[6] Wei Hu, Yangyu Huang, Fan Zhang, Guodong Yuan, and Wei Li. Ray tracing via GPU rasterization. *The Visual Computer*, 30(6-8):697–706, 2014.

[7] Tero Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In *High-Performance Graphics*, pages 33–37, 2012.

[8] Martin Knecht, Christoph Traxler, Christoph Winklhofer, and Michael Wimmer. Reflective and Refractive Objects for Mixed Reality. *IEEE Trans. Vis. Comput. Graph.*, 19(4):576–582, 2013.

[9] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum,*, 28(2):375–384, 2009.

[10] Michael Mara, Morgan McGuire, and David Luebke. Lighting Deep G-Buffers: Single-Pass, Layered Depth Images with Minimum Separation Applied to Indirect Illumination. Technical Report NVR-2013-004, NVIDIA Corporation, December 2013.

[11] Leonard McMillan and Gary Bishop. Plenoptic modeling: an image-based rendering system. In *Proceedings of SIGGRAPH 95*, Annual Conference Series, pages 39–46, 1995.

[12] Gary McTaggart. Half-life 2 shading. In *Direct3D Tutorial*, GDC, 2004.

[13] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation.* MKP, 2nd edition, 2010.

[14] Christoph Reinbothe, Tamy Boubekeur, and Marc Alexa. Hybrid ambient occlusion. *EUROGRAPHICS 2009 Areas Papers*, 2009.

[15] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. The state of the art in interactive global illumination. *Computer Graphics Forum*, 31(1):160–188, 2012.

[16] P. Rosen, V. Popescu, K. Hayward, and C. Wyman. Nonpinhole Approximations for Interactive Rendering. *Computer Graphics and Applications, IEEE*, 31(6):68 –83, nov.-dec. 2011.

[17] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, pages 197–206, 1990.

[18] Lagarde Sébastien and Antoine Zanuttini. Local image-based lighting with parallax-corrected cubemaps. In *ACM SIGGRAPH 2012 Talks*, pages 36:1–36:1, 2012.

[19] Tiago Sousa, Nickolay Kasyan, and Nicolas Schulz. Secrets of CryENGINE 3 graphics technology. In *ACM SIGGRAPH 2011 Courses, Advances in Real-Time Rendering in 3D Graphics and Games*, 2011.

[20] Xin Sun, Kun Zhou, Eric Stollnitz, Jiaoying Shi, and Baining Guo. Interactive Relighting of Dynamic Refractive Objects. *ACM Transactions on Graphics*, 27(3):35:1–35:9, August 2008.

[21] László Szirmay-Kalos, Barnabás Aszódi, István Lazányi, and Mátyás Premecz. Approximate ray-tracing on the gpu with distance impostors. *Computer Graphics Forum*, 24(3), 2005.

[22] Natalya Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, pages 63–69, 2006.

[23] Sinje Thiedemann, Niklas Henrich, Thorsten Grosch, and Stefan Müller. Voxel-based global illumination. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 103–110, 2011.

[24] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM,*, 23(6):343–349, 1980.

[25] Chris Wyman. An approximate image-space approach for interactive refraction. *ACM Trans. Graph.*, 24(3):1050–1053, July 2005.

[26] Kun Xu, Yan-Pei Cao, Li-Qian Ma, Zhao Dong, Rui Wang, and Shi-Min Hu. A Practical Algorithm for Rendering Interreflections with All-frequency BRDFs. *ACM Transactions on Graphics*, 33(1):10:1–10:16, February 2014.

[27] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-Time KD-tree Construction on Graphics Hardware. *ACM Transactions on Graphics,*, 27(5):126:1–126:11, 2008.

Paper IV

Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees

Per Ganestam[1,2], Rasmus Barringer[1], Michael Doggett[1] and Tomas Akenine-Möller[1,2]

Lund University[1] Intel Corporation[2]

Abstract

We present an algorithm, called Bonsai, for rapidly building bounding volume hierarchies for ray tracing. Our method starts by computing midpoints of the triangle bounding boxes, and then performs a rough hierarchical top-down split using the midpoints, creating triangle groups with tight bounding boxes. For each triangle group, a mini tree is built using an improved sweep SAH method. Once all mini trees have been built, we use them as leaves when building the top tree of the bounding volume hierarchy. We also introduce a novel and inexpensive optimization technique, called mini tree pruning, that can be used to detect and improve poorly built parts of the tree. We achieve a little better than 100% in ray tracing performance compared to a "ground truth" greedy top-down sweep SAH method, and our build times are the lowest we have seen with comparable tree quality.

Journal of Computer Graphics Techniques
Volume 4, Number 3, Pages 23-42, September 2015

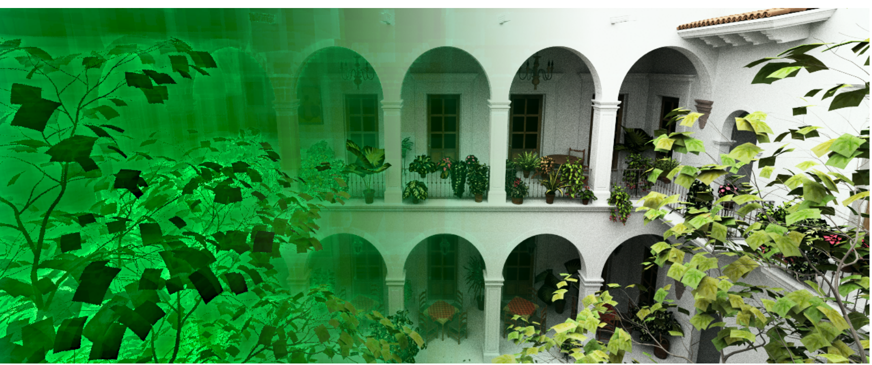


Figure 1: The San-Miguel scene (7,842,744 triangles) overlaid with a visualization of its Bonsai bounding volume hierarchy (BVH). The Bonsai BVH of San-Miguel is constructed in 478 *ms using a 2.6G Hz quad core laptop CPU (Intel 4950HQ) and rendering performance is* 107% *compared to the rendering performance of the same scene using a BVH built with the sweep SAH method.*

1 Introduction

In order to ray trace [27] a scene with path tracing [10], for example, a spatial acceleration data structure [13, 19] needs to be built. The task of this structure is to speed up the determination of what a ray intersects in a three-dimensional scene. One of the most popular spatial acceleration data structures is the bounding volume hierarchy (BVH). For animated scenes, the entire BVH, or parts of it, needs to be rebuilt every frame, and therefore, the BVH generation needs to be fast. However, it is also important that the generated trees are of high quality so the subsequent ray tracing process becomes as fast as possible.

Top-down, greedy sweep surface area heuristic (SAH) methods [16], simply abbreviated *sweep SAH* here, are known to generate high-quality trees. We present a highly efficient implementation of the sweep SAH method, and use that as a building block in our new algorithm for generating BVHs. Our algorithm is surprisingly simple, parallelizes well, and is easy to implement. As we will show in our results, our BVHs can be built faster than binning SAH methods [23], and our tree quality is better in that the subsequent ray tracing is faster.

Next, we review previous work and BVH generation background. In Section 4, we present our implementation of the sweep SAH method with some extra optimizations, followed by our novel BVH generation algorithm. Implementation details are described in Section 6 and results are presented in Section 7. Finally, we offer some conclusions.

2 Previous Work

An important component of light transport simulation performance is the time it takes a ray to find the surface intersection. Tremendous gains in ray tracing performance have been achieved through improved traversal algorithms and improved data structures. One of the first uses of hierarchical storage was presented by Clark [3], who used them to improve the determination of visible surfaces. Later Rubin and Whitted [21] developed this idea using parallelepipeds for ray tracing. To ensure the best hierarchy was constructed, Goldsmith and Salmon [7] presented the surface area heuristic (SAH), that computes the surface area for new nodes to find the best potential split of a bounding volume. MacDonald and Booth [16] later formalized the SAH. Walter et al. [26] used SAH to build trees using a bottom-up, node merging approach, but this approach requires long execution times. Recently, Gu et al. [8] demonstrated a real-time, multi-threaded CPU approximation to Walter et al.'s agglomerative clustering algorithm. While showing impressive results, we show in our results, using their provided source code, that our top down algorithm running on a multi-threaded CPU can build and trace scenes faster.

Aila et al. [1] extended the SAH metric by proposing additional quality metrics for tree construction, and hence improved ways to measure ray tracing performance. They introduced two terms where the first term accounted for the reality that many rays start or terminate inside the scene, whereas SAH assumes they do not. They called the first term end-point overlap (EPO) and it takes into account the area of the surfaces within each node. Second, they showed how to model SIMD performance by taking into account the number of leaf nodes intersected by a ray, using their leaf count variability (LCV) term. LCV is computed as ray tracing is performed, which makes it a good measure for explaining performance, but impractical for BVH construction.

The construction of BVHs typically follows a top-down approach where a bounding volume of the entire object is split into two child volumes. These child volumes are recursively split, and before splitting, the SAH is used to estimate the cost of each potential split. While this type of exhaustive search can generate trees with very low SAH cost, it can take a very long time, so faster methods are often used. A popular approximation is binned SAH [23, 24], which limits the number of potential split planes to a fixed number.

To further improve performance and utilize the parallel capacity of GPUs, Lauterbach et al. [15] presented a technique called linear BVH (LBVH), which constructed a BVH by first generating a Morton code for each primitive, then using a parallel GPU algorithm to sort them, and then recursively bucketing primitives based on the bits in their Morton codes. HLBVH [17] improved this technique by using a 2-level hierarchical sort that used the

upper bits of the Morton code to do an initial sort. Pantaleone et al. [18] used a similar 2-level build approach in their stream-based out-of-core BVH construction algorithm. Garanzha et al. [5] simplified the bookkeeping for the HLBVH algorithm and used work queues and binary search. Karras [11] improved parallel construction time of this group of algorithms by creating node indices and keys using a binary radix tree that allowed creation of connections between parent and child nodes. A related hierarchical GPU based approach is presented by Garanzha et al. [6]. In their work, a hierarchical grid is computed over the scene and used to construct the BVH using SAH.

Triangle splitting is an important technique to handle difficult scenes with a wide variety of triangle sizes. Havran and Bittner [9] presented the idea of split clipping, where the bounding box of an object is split to reduce empty overlap between object bounding boxes and *kd*-tree nodes. Ernst and Greiner [4] applied a similar concept to BVHs, by splitting triangle bounding boxes in a preprocess, before using a typical BVH construction pass. Stich et al. [22] and Popov et al. [20] proposed similar ideas, where primitives are considered for splitting into both children during BVH construction, which resulted in tighter bounding boxes on a larger range of triangles than previous approaches.

Further performance can be achieved by optimizing existing trees. Kensler [14] presented a method of improving a BVH by locally rearranging nodes or using tree rotations. Bittner et al. [2] also refined existing BVHs by selecting expensive SAH nodes for optimization, removing them, and reinserting their children at locations with minimal cost. Karras and Aila [12] selected groups of nodes in treelets and performed an exhaustive search for the optimal treelet in parallel on GPUs.

3 Background BVH Generation

As background, we first review the surface-area heuristic (SAH) [7, 16], which is used extensively in spatial data structure generation. The SAH cost for a bounding volume hierarchy (BVH), with similar notation as Karras and Aila [12], is

$$C_I \sum_{n \in I} \frac{A(n)}{A(root)} + C_L \sum_{n \in L} \frac{A(n)}{A(root)} + C_T \sum_{n \in L} \frac{A(n)}{A(root)} N(n). \tag{1}$$

This formula expresses the expected cost of traversing a random ray through the BVH, such that the ray does not terminate inside the scene geometry. The set of internal nodes is denoted by I, and L is the set of leaf nodes. The function A computes the surface area of a node's bounding volume and the function N represents the number of triangles in a leaf node. The constants C_I and C_L are the traversal costs of an internal node and a leaf

node respectively, and C_T is the cost for intersecting a triangle. Karras and Aila use $C_I = 1.2$, $C_L = 0$, and $C_T = 1$.

To determine the SAH cost of a node, n, we use the standard formulation, where we again use a similar notation as Karras and Aila [12], i.e.,

$$C(n) = \begin{cases} C_I A(n) + C(n_l) + C(n_r), & n \in I, \\ C_T A(n) N(n), & n \in L. \end{cases} \tag{2}$$

The left and right child nodes are denoted n_l and n_r, respectively. Note that the first row is an expression of splitting n into a left and a right child, while the second row represents the cost of making a leaf node of the triangles.

4 Our Implementation of Sweep SAH

The sweep SAH BVH algorithm was introduced by MacDonald and Booth [16], and one often uses a top-down, greedy approach to build such trees. Sweep SAH is commonly used as a comparison algorithm due to its high quality trees. However, most implementations seem relatively slow. In this section, we will adapt a partitioning trick from kd-tree building to BVHs, and then describe a very efficient implementation.

Sweep SAH is a top-down recursive algorithm that, at each recursion, tries to partition a set of primitives into two subsets that minimizes the surface area heuristic. The initial set to be partitioned is all primitives in the scene, and recursion stops when a partition cannot improve the cost of the tree. The SAH metric is minimized by sweeping over primitives along the x, y, and z axis. For this sweep to work, primitives need to be sorted along each coordinate axis. Typical implementations do this by sorting the primitives along the axis to test before each sweep. However, we have discovered that this is unnecessary work.

In the spirit of previous work on kd-tree building [25, 29, 28], we sort all primitives *once* along each coordinate axis before any recursion takes place, and keep these three arrays sorted within each subset during recursion. This allows us to improve performance without sacrificing tree quality, which is in contrast to the binned SAH approach [23], where quality is often reduced. The sweep part of the algorithm simply performs a sweep over the correctly sorted array, so no additional sorting is required. Once the partition that minimizes the SAH metric is found, we need to ensure that all three arrays are correctly partitioned and sorted within the two subsets.

Without loss of generality, we assume that x is the coordinate axis that we chose to partition the primitives along. This means that the y and z arrays need to have the same primitives in each subset as the x array, but ordered by the y and z axis within each subset. We do this by flagging all triangles depending on which side of the pivot they are on along x. Then, using this flag, a partition of the primitives in y and z is performed, while preserving

order. Partitioning is a fast and simple operation that runs in $O(n)$ time. This is in contrast to any comparison-based sorting algorithm, which would take $O(n \log n)$ time at best. Thus, the sweep recursion is improved from running in $O(n \log^2 n)$ time to instead execute in $O(n \log n)$ time.

In addition to the algorithmic improvements, we have found that many parts of the SAH algorithm lends itself well to vectorization and threading. Instruction-level parallelism and SIMD is exploited during SAH minimization by sweeping over multiple triangles at the same time. In our implementation, we successfully utilize 8-wide AVX2 instructions for the sweep. Thread-level parallelism is achieved by branching off the two subsets as new thread tasks at each recursion. Threads are then coordinated using a work list with task information. Since each subset can be processed independently, little synchronization is required. Additionally, the initial sorting along each coordinate axis can be performed using a parallel sorting algorithm. We currently use a radix sorter, where each axis (x, y, z) is sorted in a separate thread. Further details relevant to the implementation are presented in Section 4.

5 Bonsai BVH Algorithm

The basic idea of our approach to rapidly building bounding volume hierarchies (BVHs) is illustrated in Figure 2. Very briefly, the triangles are partitioned into groups with a user-defined size, and then a *mini tree* is built for each group, and finally, the mini trees can be seen as leaf nodes in a top tree build. If a bounding box is computed for each triangle group as part of the grouping, then the top tree and all the mini trees can be built in parallel. To improve tree quality further, we have developed a novel mini tree pruning algorithm (Section 5.4), which can be applied before the top tree is built. However, the pruning algorithm is dependent on the mini trees, and therefore, the top tree must be built after the mini trees. We make extensive use of mini trees and pruning, and hence decided to call the entire algorithm *Bonsai*.

The Bonsai algorithm is summarized by the following list of operations:

1. Compute the midpoint for each triangle.
2. Mini tree selection: split the set of midpoints hierarchically into groups of triangles.
3. Use efficient implementation of sweep SAH (Section 4) to build a mini tree per triangle group.
4. [optional] Mini tree pruning, i.e., find and optimize mini trees with subtrees that cause less optimal ray tracing performance.

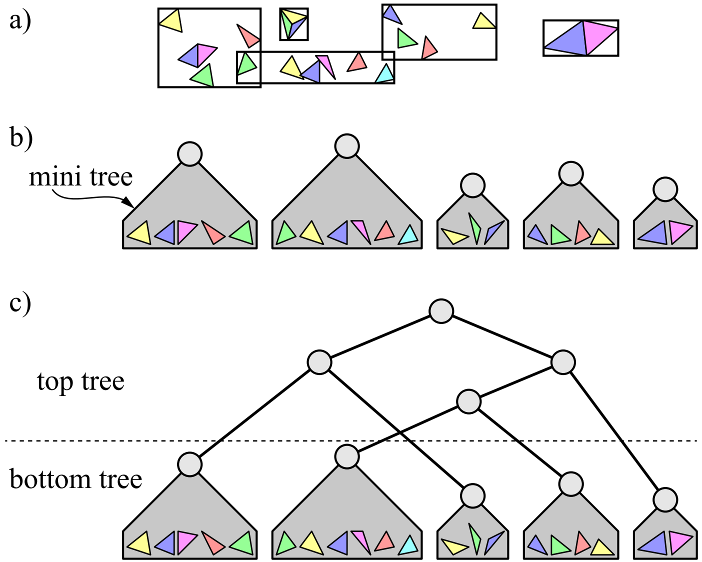


Figure 2: Illustration of our BVH tree builder in two dimensions. a) In an initial pass, groups of triangles are generated. b) For each group of triangles, an SAH-optimized mini tree is built using the algorithm in Section 4. c) The top tree is built using an SAH-optimized builder as well. Pruning is not shown in this illustration.

5. Top-tree construction using the mini trees as leaves.

These five steps are described in more detail in the following subsections.

5.1 Compute Midpoints

Initially we loop over all triangles, where the *midpoint*, i.e., the center point of a triangle's axis-aligned bounding box, is computed for each triangle. Computing midpoints maps well to both thread- and instruction-level parallelism.

5.2 Mini Tree Selection

The purpose of the second step is to find a number of relatively small groups of triangles, where the triangles of each group is spatially coherent.

It is common to use triangle bounding box midpoints to determine the sorted order of triangles in the x, y, and z dimensions as well as to determine whether a triangle is to the left or to the right of a plane, but also to determine which bin a triangle belongs to. However, in many algorithms,

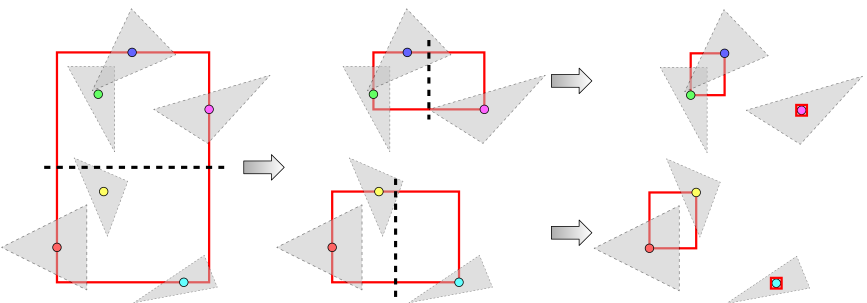


Figure 3: Our hierarchical split uses the triangle midpoints (colored circles) to generate triangle groups using a top-down approach. In each step, the bounding box of the triangle midpoints in the set is computed and used in the next split. We always split along the longest axis and in the middle.

the bounding boxes of triangles enlarges the bins or left and right bounds, and then a hierarchical top-down split follows using these boxes. We have found that it is considerably faster to use only the triangle midpoints rather than the minimum and maximum of the bounds of all vertices. In addition, long sliver triangles are simply treated as points, which avoids problems where long boxes do not become subdivided. In Figure 3, we illustrate our hierarchical split using triangle midpoints. The bounding box of the current set of triangle midpoints is calculated, and the set of triangle midpoints are simply split into two subgroups using the center point of the longest axis of the parent box. Each subgroup computes its own bounding box of the triangle midpoints, and then the hierarchical split continues until fewer or equal than N triangles are located in each triangle group. The threshold N is a parameter that can be chosen for a particular platform, depending on cache sizes, etc. Again, we achieve a high computational efficiency by exploiting thread-level parallelism at each hierarchical subdivision and instruction-level parallelism when computing the triangle midpoint bounds.

5.3 Mini Tree Construction

In the third step of our mini tree BVH algorithm, we compute an SAH-optimized subtree, called a *mini tree*, for the triangles in each group using our implementation of sweep SAH (Section 4). In theory, any method, such as, for example, LBVH [15], HLBVH [17, 11], and binned SAH [23], could be used here. However, our sweep SAH implementation results in the same tree quality as the greedy, top-down sweep SAH [16], and it is important to generate high-quality trees for the mini trees in order to get good overall tree quality. It is also possible to introduce triangle splitting techniques [4, 22, 12] here, but this is out of the scope of our work, and is

something we want to investigate in the future.

When building mini trees, we get even better hardware utilization and thread-level parallelism compared to using our implementation of sweep SAH (Section 4) for all triangles in the scene, since each mini tree is built with only one thread each. In addition, situations like sorting the three index arrays for a full sweep SAH build, using only three threads are avoided. As long as there are mini trees to build, all threads will have completely parallel tasks to process.

5.4 Bonsai Pruning

Mini tree pruning is a novel technique that we introduce in order to recover tree quality lost due to potentially poorly chosen mini tree triangle groups in the selection algorithm (Section 5.2). Since the mini tree selection does not take SAH into account, the separation of large triangles from groups of smaller triangles will be rather arbitrary. So even though each mini tree is SAH optimized, the initial choice of triangles for a mini tree may be quite poor. As a result, both the top tree and the mini trees may suffer from reduced tree quality. Although, for some scenes, such as Hairball, midpoint split works surprisingly well as a BVH build heuristic.

Our pruning algorithm searches for mini trees that have a surface area larger than some user defined threshold, T. For each such mini tree, a depth first traversal searches for the first nodes that are smaller than T, and such nodes become new mini trees that are used in the top tree construction (Section 5.5). All nodes between the found nodes and the mini tree root node are deleted, and the remaining nodes are added back as mini trees for the top tree construction. This is illustrated in Figure 4. By pruning mini trees, we will find misplaced triangles (or entire misplaced subtrees), and just add them to the top tree index list as mini tree roots. The effect is that difficult (large) triangles or difficult regions of a mini tree will be pushed up in the hierarchy, and re-built with the top tree builder among equally sized nodes. The threshold value, T, is simply a fraction of the average surface area of all the original mini tree root boxes. We present results with $T = 0.1$ and $T = 0.01$.

The pruning algorithm relies solely on the already computed mini trees (e.g., it does not split triangles or build any new data structures), and all it really does is traversing a mini tree at most once, so its addition to the overall build time is very small for most scenes. There are exceptions to all rules, and the Hairball is one such case. Since it is evenly tessellated and has an even distribution of triangles, the variance of mini tree root node areas is small, and thus Hairball build time is quite sensitive to pruning. For future work, we plan to use a fraction of the standard deviation of the mini tree root box areas as threshold in order to avoid this.

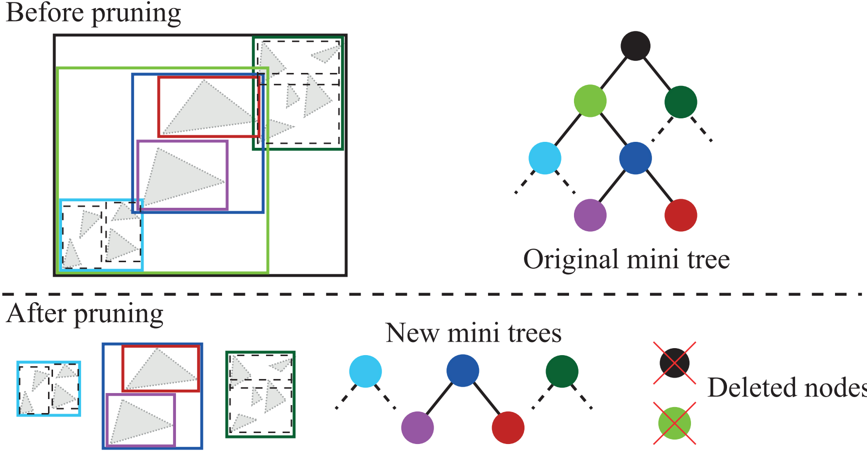


Figure 4: In mini trees, whose root box has a surface area larger than a threshold, our pruning algorithm performs a depth-first search in the mini tree to find better nodes to use as mini tree root nodes. Nodes that have a surface area smaller than the threshold (light blue, dark blue, and dark green) are found, and their subtrees are selected as new mini trees. The old mini tree root node (black) and all nodes between it and the new mini tree root nodes (light green) were parts of a tree caused by poor mini tree selection, and can be safely deleted. The remaining nodes (light blue, dark blue, and dark green) are used as new mini tree root nodes in the subsequent top tree build.

It could be worth mentioning that mini tree pruning is not exclusive to the mini tree BVH algorithm. Any already constructed BVH could potentially benefit from pruning. As an example, pruning could start at any node with less than a user defined number of triangles referenced by the sub tree.

5.5 Top Tree Construction

The top tree construction is similar to sweep SAH, except that now we have a set of mini trees, each with an axis-aligned bounding box, and a fully built subtree, and we need to build the top part of the tree based on these. Also, while performing the sweeps, there is no need to weigh in the SAH cost to the parent surface area as it is done in normal sweep SAH, since the top tree nodes will always be split as far as possible. Building the top tree can be done using any appropriate method. We use our implementation of sweep SAH (Section 4).

6 Implementation

We have implemented both our sweep SAH (Section 4) and Bonsai BVH (Section 5) with as much focus as possible on both thread-level and instruction-level parallelism.

The first consideration to achieve good parallelism is data layout. We store our vertex data in three arrays of vertices where each vertex has four values, $[x, y, z, 0]$. A triangle is composed of the i:th vertices from the three arrays where i is the triangle index. This layout maps well to SIMD execution when computing midpoints and triangle bounds. The midpoints are stored in three float arrays, one for each dimension. We keep the triangle bounding boxes in an array of eight values per bounding box, $\mathbf{b} = [x_{min}, y_{min}, z_{min}, 0, x_{max}, y_{max}, z_{max}, 0]$, and all bounds are arranged in an array as $[\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}]$, where n is the number of triangles in the scene. With this layout, each triangle bounding box can be loaded into a single 256-bit AVX2 register.

In our implementation of sweep SAH, denoted *SweepSAH* from now on, each recursion in the sweep algorithm spawns a new thread task to the left child, while the right child is constructed using the current thread. The sweep loop from left to right operates on eight triangle indices per iteration, that is, eight triangle bounds are loaded and accumulated to eight potential left side bounding boxes. The left to right sweep is quite similar. One situation when SIMD instructions are not used is the sorted order preserving index partitioning described in Section 4. Memory reads and writes while partitioning are simply too scattered to benefit from SIMD instructions and the partitioning algorithm does not have good SIMD features. However, partitioning consumes only a small part of the total running time, and so is generally not a problem.

In Section 5.1(midpoint computation), we compute four midpoints per loop iteration using 256-bit AVX2 registers. Even though triangle vertices are loaded in order to compute the midpoints, using the triangle bounding boxes, it does not pay off to save the bounding boxes to memory at this stage of the algorithm. A serial loop operating on independent data lends itself well to thread-level parallelism and our threading scheduler simply assigns the available hardware threads to 1024 sized segments of the loop range. Once a thread finishes computing its segment, it is assigned a new segment of 1024 iterations of the loop.

The mini tree selection (Section 5.2) is designed in a recursive fashion, where we spawn new thread tasks at each recursion. We compute the bounding boxes around the midpoints using 8-wide AVX2 registers and the 8-wide min and max intrinsic functions. Since we work with midpoints and not bounding boxes, we can read 8 of them per iteration and find the min and max of all eight midpoints using only six SIMD instructions (one min and one max AVX2 instruction per dimension). Without 8-wide AVX2, the same

computation would require 16 min and max operations per dimension.

Triangle bounding boxes are computed and stored during mini tree construction (Section 5.3). The reason why this is not done earlier is that now it is known which triangles belong to which mini tree and all data necessary is gathered and computed in pre-allocated thread local memory. Operating in thread local memory improves caching and removes false sharing between threads. Each mini tree is built by only one thread, and so there are no idle threads as long as there are mini trees left to construct. This is slightly different from SweepSAH, since there is no need to spawn new thread tasks while recursing down the tree.

It is not straightforward to map the Bonsai pruning algorithm to SIMD instructions, since the algorithm basically just traverses the constructed mini trees. The average surface area computation of mini tree root nodes benefits from SIMD instructions, but is a tiny part of pruning. However, the traversal has thread-level parallelism just as SweepSAH and mini tree selection, and in addition, each mini tree can be pruned in parallel. The only synchronization is when a new mini tree root is found and the new mini tree root node bounds are written to the top tree's bounds array.

The top tree (Section 5.5) is built in a similar manner as SweepSAH.

7 Results

All our results have been generated on a Macbook Pro laptop with Iris Pro 5200 integrated graphics processor. More specifically, the CPU is a 4950HQ, which has eight hardware threads at 2.6 GHz. All BVH building is done entirely on the CPU cores, while ray tracing is done using both the CPU cores and the GPU. More precisely, BVH traversal and triangle intersections are done using the GPU while shading computations are done using the CPU. Note that for all our comparisons, we use a path tracer, which means that the rays are highly incoherent after a few bounces. Our baseline algorithm for comparison is the efficient implementation of sweep SAH as described in Section 4. This method is denoted *SweepSAH*, and note that it generates the same high-quality trees as a standard sweep-based SAH BVH algorithm. Furthermore, we compare to binned SAH [23], referred to as *binSAH*, to *Bonsai* (Section 5), and to two versions of *Bonsai P* (Section 5.4), where Bonsai P is our algorithm with pruning. For binSAH, we use Intel's Embree 2.2 implementation of binned SAH BVH. However, Embree's fastest BVH builder is designed to build 4-wide trees, but we only compare to 2-wide trees, since our GPU traversal is faster for these trees. Although older versions of Embree implement binary tree builders, those implementations were not as fast, and we found it fairest to modify the faster 4-wide builder to construct 2-wide trees. We also compare to approximate agglomerative clustering (AAC), using the authors' source code [8], where

we used both the high quality (HQ) and the low quality version (LQ), where the latter is faster, but generates lower quality trees. The provided source code is single threaded, so to generate fair build times, giving AAC the benefit of the doubt, we have divided the single-threaded performance by 4, since we have 4 hardware cores and because the authors claim linear speedup with number of cores.

Our first contribution in terms of results is to show the results of our implementation of the sweep SAH algorithm (Section 4). While it is difficult to compare against others' implementations of the same algorithm, we simply note that the Hairball often takes at least 15× longer to generate [12, 2] than when using our implementation. In fairness, there are differences in CPUs and likely also in the ambition level of optimization for sweep SAH. Since we will release our source code, we believe that this is a small but important contribution since the community will get access to a highly efficient implementation of sweep SAH for BVHs.

All major results are shown in Figure 5 for 14 different scenes, where Bonsai and Bonsai with pruning (Bonsai P in the table) were generated with a maximum mini tree size of 512 triangles and Bonsai P* with a maximum of 4096 triangles. The pruning threshold constants are 0.1 for Bonsai P and 0.01 for Bonsai P*.

	Arabic City		Battlefield		Bentley		Conference		Crown		Dragon		Fairy Forest	
Triangles	411,563		64,256		2,346,676		331,177		4,868,924		7,349,988		174,117	
	Build	Trace	Build	Trace	Build	Trace	Build	Trace	Build	Trace	Build	Trace	Build	Trace
SweepSAH	67.3	**100%**	8.9	**100%**	560	**100%**	57.8	**100%**	1585	**100%**	3074	**100%**	29.9	**100%**
binSAH	46	91%	9.5	92%	271	89%	38.7	93%	584	90%	948	89%	24.5	94%
AAC HQ	76.8	84%	13.2	75%	502	69%	60.9	91%	1033	73%	1624	67%	34.7	76%
AAC LQ	32.3	72%	6.2	69%	209	67%	23.7	87%	415	71%	632	66%	14	79%
Bonsai	**19**	81%	**3.3**	78%	**117**	90%	**16.3**	86%	**251**	90%	**403**	92%	**8.8**	87%
Bonsai P	20.2	96%	3.7	94%	119	94%	16.9	96%	278	96%	415	92%	9.5	96%
Bonsai P*	23.6	**100%**	4.3	**101%**	133	**97%**	19.7	**98%**	315.4	**97%**	463	**93%**	11	**97%**

	Hairball		Italian City		Kalabsha		Sala		San Miguel		Sibenik		Sponza	
Triangles	2,850,000		368,322		4,542,545		395,725		7,842,744		79,306		262,266	
	Build	Trace	Build	Trace	Build	Trace	Build	Trace	Build	Trace	Build	Trace	Build	Trace
SweepSAH	590	**100%**	57.3	100%	1603	100%	67.9	100%	2386	**100%**	10.5	100%	51.4	100%
binSAH	275	81%	40.71	92%	571	96%	43.3	98%	947	94%	10.7	**100%**	32.3	100%
AAC HQ	625	49%	68.3	87%	954	82%	76.6	97%	1649	91%	15.9	84%	50.4	**105%**
AAC LQ	276	50%	29	82%	367	74%	32	93%	688	87%	6.6	84%	21.5	**105%**
Bonsai	**119**	95%	**17.3**	75%	**267**	81%	**19.3**	91%	**409**	78%	**4.3**	94%	**12.6**	90%
Bonsai P	156	**100%**	18.5	**101%**	271	**102%**	20	**104%**	424	**106%**	4.9	**100%**	13.4	**105%**
Bonsai P*	203	**103%**	21.5	**108%**	298	**108%**	23	**103%**	478	**107%**	5.8	**108%**	15.4	**108%**

Figure 5: Build times are in milliseconds and ray tracing (path tracing) performance is relative to SweepSAH. Note that the fastest build time is marked with bold text, and so are the two fastest ray tracing performance numbers per column. The binned SAH algorithm is from Intel's Embree 2.2 and uses a modified version of its fast BVH4 builder to construct binary trees.

For Bonsai, ray tracing performance ranges from 75% up to 95% compared to SweepSAH. With Bonsai P, ray tracing performance is increased to range between 92% to 105%, with an average of 98.5%. The most difficult scene to build for Bonsai P and P* is Dragon, which with P* is ray traced at 93% performance compared to SweepSAH. The other scenes are in the range 97% to 108% for P* and the total average ray tracing performance is 101.5% to that of SweepSAH. Depending on the scene, the increase in build time for pruning can range from very little (Bentley) to nearly double (Hairball). Bonsai build times vary with different mini tree sizes, and without loss of ray tracing performance, improved build times can be made for some scenes by selecting other mini tree sizes. However, with no a priori knowledge regarding which scene would benefit from which mini tree size we have found that a size of 512 is a reasonable compromise across the test scenes. Bonsai P and P* build times are also affected by the pruning constants, and we base our choices on empirical observations in regards of both ray tracing performance and build times. It is actually not pruning itself that adds to the build time, but it is the increased number of leaf nodes for the top tree, caused by pruning, that affects build time. Simply put, more leaf nodes result in more work for the top tree. This effect can be seen in Figure 6, where we show the timings of each step of the Bonsai and the Bonsai P algorithms, relative to Bonsai build times of the San Miguel scene.

We found that scenes with a uniform distribution of finely tessellated triangles with roughly the same size are more sensitive to the pruning algorithm. This is because we use a fraction of the average surface area of mini tree root nodes as a threshold. If there is little variance among mini tree root nodes sizes, then too many mini trees may be larger than the threshold, and they simply get over pruned. However, if a scene has a larger variance in triangle sizes and triangle distribution, then it is likely that just a smaller number of all the mini trees have a surface area larger than the threshold, and these mini trees are exactly the ones that need to be pruned.

Average build times, SAH costs, and ray tracing times for our 14 test scenes are presented in Figure 7. Build performance for Bonsai compared to SweepSAH is on average 4× faster and compared to binSAH a little more than 2× faster. Bonsai, Bonsai P and P* all have faster build times than AAC LQ and, in addition, a significantly higher ray tracing performance. The SweepSAH implementation is on par with the build times of AAC HQ, but ray tracing is often much faster for SweepSAH. Note that we use a full path tracer, with highly incoherent ray distribution after a few bounces, for all comparisons and we measure wall clock rendering time, while Gu et al. [8] used 16 diffuse rays and counted BVH node traversal and intersections tests in their paper. As can be seen, our algorithm is significantly faster than the others at building, and at the same time, it has the best ray tracing performance.

Although it is very difficult to make comparisons to algorithms aimed at

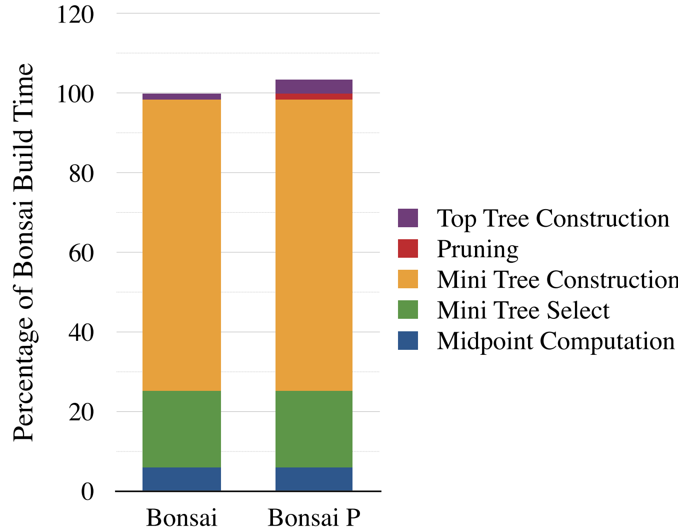


Figure 6: As pruning increases ray tracing performance significantly, it also has an adverse effect on BVH build times. In the two bars, we show the time of each step of the Bonsai and Bonsai P algorithms relative to Bonsai's full build time. We chose San Miguel as a representative scene. Even though pruning (red) does not take up much more than 1% of the total build time, the top tree build time (purple) increases from 1.5% of the total build time without pruning to 3.3% with pruning.

completely different hardware, it might be worth noting, that on average, in the cases where we share test scenes, our Bonsai BVH build times are less than 1.4× the build times of the GPU BVH algorithm by Karras and Aila [12]. Note that the latter was executed on an NVIDIA GTX Titan, which has 10× more FLOPS than the CPU cores that we use. We have similar behavior with Bonsai P, where the shared scenes have an average build time less than 1.6× to that of Karras and Aila.

It is well known that SAH does not correlate perfectly to ray tracing performance. However, it is generally assumed that better ray tracing performance can be the result when SAH cost is lowered. Figure 7 shows the average SAH costs for all the involved algorithms and Figure 8 shows the SAH cost details across all scenes. Although it is not possible to deduct any clear conclusions how SAH cost reflects ray tracing performance between different algorithms (AAC HQ often has a lower SAH cost than SweepSAH but most of the time also lower ray tracing performance), there seem to be a clear intra-algorithmic correlation for both Bonsai and AAC, where reduced SAH costs correlate well to improved ray tracing performance.

	Build Times	SAH Cost	RT Performance
SweepSAH	100%	100%	100.0%
binSAH	61%	106%	92.6%
AAC HQ	101%	106%	80.8%
AAC LQ	42%	108%	75.2%
Bonsai	**25%**	112%	85.7%
Bonsai P	27%	101%	98.5%
Bonsai P*	32%	99%	**101.5%**

Figure 7: Average build times, SAH costs, and ray tracing performance compared to SweepSAH. The algorithms with the best build times and best ray tracing performance are highlighted respectively.

	Arabic City	Battlefield	Bentley	Conference	Crown	Dragon	Fairy Forest
SweepSAH	113.5	33.9	4.2	53.6	36.6	25.5	47.8
binSAH	119.0	36.4	6.1	57.5	37.5	26.3	47.6
AAC HQ	111.0	41.5	6.1	49.8	41.8	29.1	54.5
AAC LQ	124.5	41.8	6.1	50.9	41.6	28.2	53.5
Bonsai	129.1	39.4	6.0	63.9	37.5	26.0	49.9
Bonsai P	109.3	34.1	5.9	55.9	36.8	25.9	46.6
Bonsai P*	108.2	33.9	5.8	55.4	36.7	26.1	46.3

	Hairball	Italian City	Kalabsha	Sala	San Miguel	Sibenik	Sponza
SweepSAH	620.5	90.7	10.4	53.9	95.6	74.9	121.3
binSAH	663.4	96.2	10.8	54.2	96.7	74.2	122.9
AAC HQ	767.1	86.9	8.2	52.3	88.5	80.2	109.5
AAC LQ	785.8	97.5	8.0	52.0	89.0	82.2	112.7
Bonsai	640.5	107.6	10.3	58.3	111.0	79.3	136.5
Bonsai P	610.7	86.2	10.1	51.2	92.6	71.6	114.2
Bonsai P*	595.9	85.2	8.6	51.6	93.0	71.0	113.4

Figure 8: SAH costs, calculated using Equation 3, of all scenes and across all algorithms. Traversal cost constants used are $C_I = 2$, $C_L = 0$, and the triangle intersection cost constant is $C_T = 1$.

8 Conclusions and Future Work

Bonsai is a highly efficient and simple to implement algorithm for building bounding volume hierarchies. When measuring ray tracing performance, our algorithm, supplemented with the pruning optimization technique, meets or comes close to, but in many cases surpasses, the tree quality of SweepSAH. We have shown that Bonsai maps well to both thread-level and instruction-level parallelism. An observation is that it may well be that CPU hardware is a better fit to construct high quality bounding volume hierarchies than GPU hardware. This is based on the comparisons of BVH build times

between Bonsai (with and without pruning) on a laptop CPU and Karras and Ailas [12] GPU BVH algorithm. The latter is executed on an NVIDIA GTX Titan with 10 times more compute capabilities, while Bonsai with pruning only takes 60% longer to execute on a laptop CPU.

Acknowledgements

Thanks to the whole Advanced Rendering Technology group at Intel, and especially to Charles Lingle for help with internships and to Jacob Munkberg for help in the beginning of this project. Also, Tomas is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation. Thanks to Veronica Sundstedt for the Kalabsha temple model and to Timo Aila for the Italian and Arabic City models. Thanks to Carsten Benthin for sharing information about Embree. Per and Michael are funded by ELLIIT and the Intel Visual Computing Institute.

Bibliography

[1] Timo Aila, Tero Karras, and Samuli Laine. On Quality Metrics of Bounding Volume Hierarchies. In *High-Performance Graphics*, pages 101–107, 2013.

[2] Jirí Bittner, Michal Hapala, and Vlastimil Havran. Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum,*, 32(1):85–100, 2013.

[3] James H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM,*, 19(10):547–554, 1976.

[4] Manfred Ernst and Günther Greiner. Early Split Clipping for Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 73–78, 2007.

[5] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. Simpler and Faster HLBVH with Work Queues. In *High-Performance Graphics*, pages 59–64, 2011.

[6] Kirill Garanzha, Simon Premoze, Alexander Bely, and Vladimir Galaktionov. Grid-based SAH BVH construction on a GPU. *The Visual Computer,*, 27(6-8):697–706, 2011.

[7] Jeffrey Goldsmith and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics & Applications,*, 7(5):14–20, 1987.

[8] Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. Efficient BVH Construction via Approximate Agglomerative Clustering. In *High-Performance Graphics*, pages 81–88, 2013.

[9] Vlastimil Havran and Jiri Bittner. On Improving KD-Trees for Ray Shooting. In *Winter School on Computer Graphics*, pages 209–217, 2002.

[10] James T. Kajiya. The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, volume 20, pages 143–150, 1986.

[11] Tero Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In *High-Performance Graphics*, pages 33–37, 2012.

[12] Tero Karras and Timo Aila. Fast Parallel Construction of High-quality Bounding Volume Hierarchies. In *High-Performance Graphics Conference*, pages 89–99, 2013.

[13] Timothy L. Kay and James T. Kajiya. Ray Tracing Complex Scenes. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, volume 20, pages 269–278, 1986.

[14] A. Kensler. Tree Rotations for Improving Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 73–76, 2008.

[15] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum,*, 28(2):375–384, 2009.

[16] David J. MacDonald and Kellogg S. Booth. Heuristics for Ray Tracing Using Space Subdivision. *Visual Computer,*, 6(3):153–166, 1990.

[17] J. Pantaleoni and D. Luebke. HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry. In *High-Performance Graphics*, pages 87–95, 2010.

[18] Jacopo Pantaleoni, Luca Fascione, Martin Hall, and Timo Aila. PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes. *ACM Transaction on Graphics,*, 29(3):37.1–37.10, 2010.

[19] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation.* MKP, 2nd edition, 2010.

[20] Stefan Popov, Iliyan Georgiev, Rossen Dimov, and Philipp Slusallek. Object Partitioning Considered Harmful: Space Subdivision for BVHs. In *High-Performance Graphics*, pages 15–22, 2009.

[21] Steven M. Rubin and Turner Whitted. A 3-dimensional Representation for Fast Rendering of Complex Scenes. In *Computer Graphics (Proceedings of ACM SIGGRAPH 80),*, volume 14, pages 110–116, 1980.

[22] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial Splits in Bounding Volume Hierarchies. In *High-Performance Graphics*, pages 7–13, 2009.

[23] Ingo Wald. On Fast Construction of SAH-based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 33–40, 2007.

[24] Ingo Wald. Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics,*, 18(1):47–57, 2012.

[25] Ingo Wald and Vlastimil Havran. On Building Fast Kd-trees for Ray Tracing, and on Doing that in O(N log N). In *IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006.

[26] B. Walter, K. Bala, M. Kulkarni, and K. Pingali. Fast Agglomerative Clustering for Rendering. In *IEEE Symposium on Interactive Ray Tracing*, pages 81–86, 2008.

[27] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM,*, 23(6):343–349, 1980.

[28] Zhefeng Wu, Fukai Zhao, and Xinguo Liu. SAH KD-tree Construction on GPU. In *High-Performance Graphics*, pages 71–78, 2011.

[29] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-Time KD-tree Construction on Graphics Hardware. *ACM Transactions on Graphics,*, 27(5):126:1–126:11, 2008.

Paper V

SAH guided spatial split partitioning for fast BVH construction

Per Ganestam and Michael Doggett

Lund University

Abstract

We present a new SAH guided approach to subdividing triangles as the scene is coarsely partitioned into smaller sets of spatially coherent triangles. Our triangle split approach is integrated into the partitioning stage of a fast BVH construction algorithm, but may as well be used as a stand alone pre-split pass. Our algorithm significantly reduces the number of split triangles compared to previous methods, while at the same time improving ray tracing performance compared to competing fast BVH construction techniques. We compare performance on Intel's Embree ray tracer and show that BVH construction with our splitting algorithm is always faster than Embree's pre-split construction algorithm. We also show that our algorithm builds significantly improved quality trees that deliver higher ray tracing performance. Our algorithm is implemented into Embree's open source ray tracing framework, and the source code will be released late 2015.

To appear in *Computer Graphics Forum (Proceedings of Eurographics)*, Volume 35, Number 2, 2016.

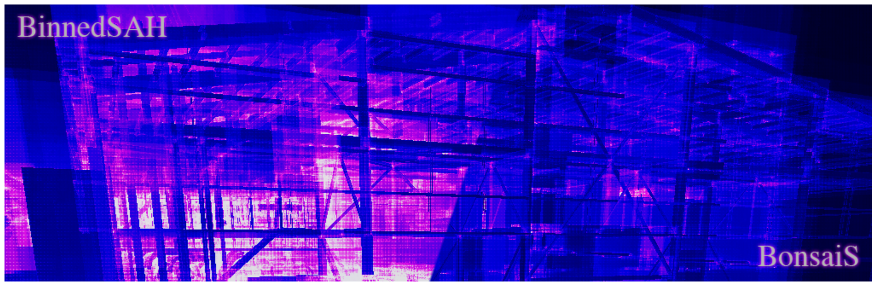


Figure 1: Traversal cost visualization of the Power Plant model (12,759,246 triangles). The binned SAH builder is represented by the left half of the figure and our splitting approach together with the Bonsai BVH algorithm is represented by the right half. A brighter color means a higher traversal cost. The binned BVH is built in 1311ms and the Bonsai BVH using our triangle split approach is built in 1881ms. However, traversal cost is significantly reduced and rendering performance is improved by 60% when using our triangle split BVH.

1 Introduction

For ray tracing techniques [20], such as path tracing [10], to be a feasible choice of rendering it is necessary to accelerate triangle intersection computations with an underlying accelerating spatial data structure [12, 15]. This data structure, often arranged as a binary tree, is used to speed up traversal of a three-dimensional scene to determine which triangle a ray intersects. Common types of data structures for ray tracing are kd-trees, grids, and the Bounding Volume Hierarchy (BVH), where the latter has gained a lot of popularity in recent years.

It is also important that the data structure is of high quality, in the sense that a higher quality data structure results in better ray tracing performance. A commonly used algorithm to construct high quality BVHs is a greedy top-down approach called sweep SAH, that uses the Surface Area Heuristic (SAH) [14]. Another crucial aspect of BVH construction, especially for animated scenes and real-time ray tracing, is that the data structure must be constructed quickly, since it may need to be rebuilt or updated every frame.

An often overlooked, but still important aspect for high quality tree construction is triangle splitting. Triangle splitting creates multiple references to a single triangle enabling the axis aligned nodes in a BVH to have a tighter fit, resulting in improved ray traversal times.

Analyzing the results in Section 5 reveals some disadvantages of current triangle splitting approaches which may be either a lack of robustness or significantly increased BVH construction times and memory usage.

We propose a new fast top-down triangle splitting algorithm that produces BVHs with a quality similar to those of Split BVH (SBVH) [17] but with build times close to those of the fast binned [18] BVH construction approach. Our algorithm may be used as a preprocess to construction, or in conjunction with other top-down BVH approaches. By pairing our method with top-down construction algorithms, our splitting approach can take advantage of the partitioning stage already present in the algorithm. We evaluate our new algorithm by implementing it into the industry grade Embree [19] ray tracing system. We use our method both with sweep SAH as a pre-split pass and integrated with the Bonsai BVH construction algorithm [7] and compare our results with some of the high performing and high quality BVH builders available in Embree.

Our main contribution is the fastest CPU based triangle split BVH builder to achieve ray tracing performance comparable to SBVH. Furthermore, our triangle split approach can be used as a preprocess to any BVH builder, and thus improve ray tracing quality by simply adding a pre-split module to existing frameworks. To attain our results we have developed a new simple but effective recursive triangle split approach and paired it with a novel in-place parallel partitioning scheme for recursively growing data.

2 Previous Work and Background

Havran and Bittner [9] presented the idea of split clipping, where the bounding box of an object is split to reduce empty overlap between object bounding boxes and *kd*-tree nodes. Rather than actually splitting triangles and storing the resulting polygons, or tessellating the polygons into several triangles, only the bounding boxes of triangles are actually split and re-computed, to create tighter axis-aligned bounds around the triangles. By performing split clipping rather than actual triangle splitting both computational complexity and the memory footprint are reduced.

Ernst and Greiner [5] applied a similar concept to BVHs by splitting triangle bounding boxes in a preprocess, called Early Split Clipping (ESC), before using a typical BVH construction pass.

Dammertz and Keller [3] suggested splitting triangles based on the tightness of the bounding boxes on each triangle edge and introduced a heuristic based on the volumes of the triangles' bounding boxes, the Edge Volume Heuristic (EVH).

Karras and Aila [11] introduced a new heuristic to the pre-splitting approach and the concept of a split budget. Their heuristic significantly improves split candidate selection and consequently improves the BVH quality compared to earlier methods. By using a split budget they also reduce the risk of producing too many triangle splits.

Although computing triangle splits prior to BVH construction is a tempting approach, it also comes with certain impediments. As noted by Karras and Aila [11], both ESC [5] and the EVH [2] fail in robustness. With the lack of knowledge of which triangle is valuable to split, some triangles may be split when they shouldn't, and some that should may not be split enough, or not split with the right split plane; sometimes resulting in worse ray tracing performance than a BVH without triangle splitting. Another shortcoming shared by the earlier pre-split approaches is that an a priori choice has to be made. Since ESC only relies on a predefined constant, it has to be hand tuned for every scene. The heuristic of EVH tries to mitigate the hand-tuning issue, however, it still has a constant that needs to be chosen. Although the pre-splitting heuristic by Karras and Aila [11] produces very good BVHs and is robust across a wide range of scenes, there is still the choice of split budget size. Sometimes a split budget of 10% of the triangles is enough, however, in other scenes a split budget of 50% is necessary to reach the potential increase in ray tracing performance. Thus the risk of performing too many splits is still inherent, since the BVH of some scenes cannot be improved by triangle splitting.

Stich et al. [17] and Popov et al. [16] proposed similar ideas, where primitives are considered for splitting into both children during BVH construction, which resulted in tighter bounding boxes on a larger range of triangles than previous approaches. The top-down approaches by Stich et al. [17] and Popov et al. [16] have the potential of producing superior BVHs. However, at the mercy of long build times. At each level of recursion, SBVH by Stich et al. [17] picks the better of either using the best object split computed by sweep SAH, or the best spatial split among 256 equidistant split planes. To reduce the risk of performing too many triangle splits, and potentially running out of memory, SBVH incorporates a bias so that it may choose sweep SAH object split, even though spatial splits in fact could reduce SAH costs further. As of yet, the SBVH algorithm by Stich et al. [17] produces the highest quality BVHs.

Since we make extensive use of Bonsai, a fast CPU based BVH construction algorithm presented by Ganestam et al. [7], it is worth mentioning a few details about the algorithm. Initially, Bonsai employs a fast approximate partitioning routine, using triangle mid-points only, to divide the scene into smaller spatially coherent groups of triangles. The triangle groups are passed to a BVH builder, in Bonsai a fast sweep SAH routine is used, and in parallel constructed into *mini-trees*. Prior to the last stage where the mini-trees are considered as leaf nodes in a sweep SAH pass, a *pruning* algorithm is applied to each mini-tree. Thus tightening the bounds of each mini-tree and correcting potential flaws to the full BVH caused by the initial partitioning stage.

Domingues and Pedrini [4] improve upon the performance of GPU based BVH construction by using an agglomerative process that merges nodes in

treelets based on selecting the minimum surface area of the bounding boxes. They build on previous treelet reordering work that searched exhaustively for the best treelet [11]. Their techniques is based on an initial tree being built using the LBVH technique [13]. Domingues and Pedrini don't present any results with triangle splitting. However, their BVH builder may as well be combined with the pre-splitting approach by Karras et al. [11], or for that matter, our approach as a preprocess.

Previous triangle splitting algorithms, although producing high quality BVHs, are either slow in construction [17, 16], or in need of hand tuning or manual decisions prior to construction [5, 2, 11]. Motivated by the little, or none, *a priori* knowledge needed for BVH construction and the prospect of producing superb trees, we continue in the fashion of the recursive triangle split approaches. However, to reduce build times, we pair our splitting method with the Bonsai BVH algorithm.

3 Algorithm

Our SAH guided mid-point split partitioning can be either integrated with the Bonsai BVH construction algorithm or work as a stand alone triangle pre-split method for any other BVH algorithm. Both with Bonsai and as a sweep SAH pre-split pass, our split partitioning approach results in improved ray tracing performance and when paired with Bonsai, ray tracing performance is similar to that of Embree's SBVH based spatial split builder.

3.1 The Surface Area Heuristic

Although the Surface Area Heuristic [8, 14] for BVH construction in recent years has been found not to correlate perfectly with improved ray tracing performance [6, 1, 7], it is still an advantageous approach in the construction of high quality BVHs. With this in mind we will briefly explain the SAH cost equations and how the SAH cost is evaluated and minimized in a top-down BVH construction algorithm.

The total SAH cost of a BVH can be computed as

$$C_i \sum_{n \in I} \frac{A(n)}{A(root)} + C_l \sum_{n \in L} \frac{A(n)}{A(root)} + C_t \sum_{n \in L} \frac{A(n)}{A(root)} N(n), \tag{1}$$

where the equation expresses the expected cost of a random ray traversing the BVH, however in such a way, that it does not terminate within the scene geometry. Internal nodes are in the set I and the set L represents leaf nodes. The operator A represents the surface area of a node's bounding box and N represents the number of triangles in a node. The constants C_i, C_l, and C_t represent the costs of traversing an internal node, a leaf node, and intersecting a triangle, respectively.

Using SAH to optimize a top-down BVH builder is done by at each level of recursion evaluating and minimizing the equation

$$E_r = C_i A(n) + C_l(A(n_l)N(n_l)) + A(n_r)N(n_r)), \quad (2)$$

which represents the cost of proceeding with the recursion, where n_l and n_r are the potential left and right child nodes. The result is compared to

$$E_t = C_t A(n)N(n), \quad (3)$$

which represents the cost of terminating the recursion and using the current node n as a leaf node in the BVH. If $E_t < E_r$ an SAH optimal leaf node is found and the recursion terminates. Otherwise the recursion continues with n_l and n_r as child nodes. In our triangle split algorithm we use Equation 2 in a similar way to how it is used in recursive BVH construction.

3.2 SAH guided mid-point split partitioning

In addition to computing the mid-point bounds used to find the split plane while partitioning, to know whether triangles should be subdivided or not, we need to compute the complete left and right bounding boxes. While partitioning, we accumulate six sets of different triangle categories. We create two completely *disjoint* sets of left and and right triangles, in relation to the mid-point split plane, where the two sets are denoted as D_L and D_R and $D_L \cap D_R = \emptyset$. We also store two *overlap* sets, O_L and O_R, of those triangles that have their mid-points to the left or right side of the split plane but with bounding boxes overlapping the split plane. The last two sets, the *split* sets, contain the left and right halves of subdivided triangles respectively, and are denoted as S_L and S_R. The triangles in each of the split sets are exactly the same as the triangles in the overlap sets, i.e. $O_L \cup O_R = S_L = S_R$. However, due to split clipping [9], the left and right subdivided triangles have different bounding boxes. This also implies that the split sets contain twice as many triangles as the overlap sets.

Figure 2 shows the bounding boxes and their associated groups of triangle indices created around the split plane. As in the Bonsai algorithm [7], to avoid empty partitions, we use the mid-point bounds rather than full bounds when choosing the split plane. As a consequence of this, the split plane used to subdivide triangles isn't the spatial median of the complete bounding box of a partition, but instead chosen as the spatial median of the mid-point bounds. We choose the largest axis for partitioning.

Once the bounds of the six sets are computed we evaluate the benefit of splitting triangles by computing the SAH cost when using the overlap sets

$$C_O = A(D_L \cup O_L)|D_L \cup O_L| + A(D_R \cup O_R)|D_R \cup O_R|$$

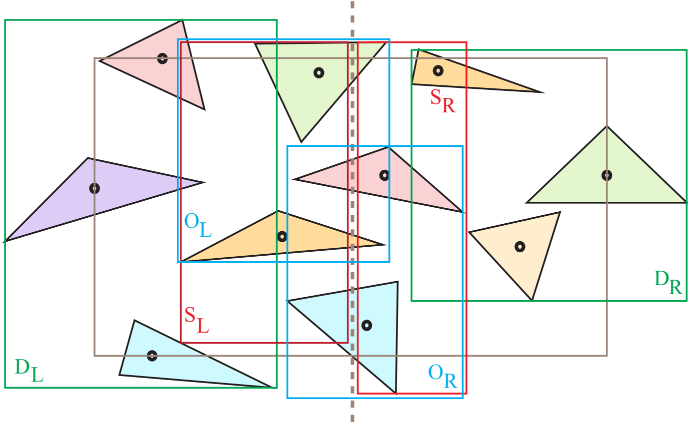


Figure 2: The bounding boxes of the different sets of triangles used by our triangle split method. The brown box represents the mid-point bounds of all triangles and its spatial median is used to create the split plane. The blue bounds represent the overlap sets, the red bounds represent the split sets, and the green bounds represent the left and right disjoint sets.

and comparing the result to the SAH cost when using the split sets, evaluated as

$$C_S = A(D_L \cup S_L)|D_L \cup S_L| + A(D_R \cup S_R)|D_R \cup S_R|,$$

where A is the surface area of a set, C_O is the SAH cost of keeping the original triangles, and C_S is the SAH cost of using the split triangles. Split partitioning is continued by choosing whichever set has the lowest SAH cost.

Our split partitioning algorithm continues the recursion until a threshold of triangle count has been reached. The threshold is based on the Bonsai algorithm's mini-tree size and can be set arbitrarily. However, we have found that for smaller scenes ($< 100,000$ primitives) a smaller threshold of 512 is necessary to perform enough triangle splitting, and for larger scenes ($> 4,000,000$ primitives), a value of 8192 as a threshold is enough to achieve a high quality BVH. We linearly interpolate the threshold value between its minimum and maximum values. Other than the mini-tree size threshold, our method does not require any other bias or tuning variables to guarantee enough splitting or to avoid over-splitting.

4 Implementation

Since we have implemented our splitting algorithm in Embree, we also make use of some of their design choices. The type *primitive reference*, denoted

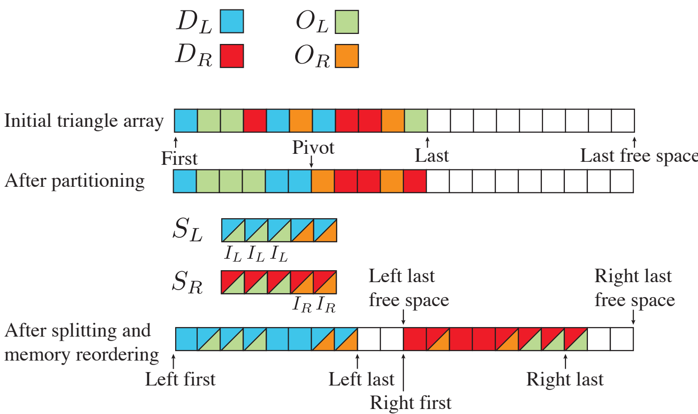


Figure 3: Triangle array memory management in each recursion. The initial array contains disjoint left and right triangles (blue and red) and overlap triangles (green and orange). After in-place partitioning, the left side of the split plane contains $D_L \cup O_L$ and the right side its counterparts. Two separate arrays of temporary left and right split triangles contain their respective parts of split triangles. An indexed triangle, one that can be inserted where an overlap triangle currently resides, in the left split set S_L is colored blue and green and marked with I_L. An indexed triangle in the right set S_R is colored red and orange and marked with I_R. If the SAH metric finds the split sets favorable, indexed triangles are inserted where the overlap triangles are and a small segment of the right partition is moved as the last part of the diagram illustrates. Note that this is an extreme case were almost 50% of the triangles are split, and it is merely for illustrative purposes.

primref, is a structure of six floats representing a triangles bounding box and two integers used as references to the original mesh and triangle. Since the size of a *primref* is $(6 + 2) \cdot 32$ bits, it fits into a 256-bit AVX register, which is useful for fast bounding box computations. As an initial pass, all algorithms we present create an array of *primrefs* containing the original bounding boxes and indices of each triangle. It is the *primref* array that is used in BVH construction and triangle splitting. A split triangle is simply two *primrefs* referencing the same original triangle, but having different bounding boxes. To avoid confusion, in the following section we simply talk about triangles, although all operations are actually done on *primrefs*.

4.1 Triangle Splitting and Recursively Growing Memory

One of the most difficult tasks with a recursive algorithm that needs to expand memory wise, is efficient memory management. In practice, keeping the different sets (split, overlap, and disjoint) in separate arrays and then

merging them when a decision whether to split or not is taken, although much easier to implement, is inefficient memory wise.

We perform a quick-sort style in-place partitioning with the split plane as the pivot. To minimize memory traffic, we track the index of any triangle that belongs to one of the overlap sets instead of making an actual copy of the triangle. When partitioning discovers an overlap triangle, it places that triangle to the left or right of the split plane depending on its bounding box midpoint, as is done with a non-split partitioning. Thus there is no need for any auxiliary storage of the overlap sets as it is with the split sets. Since we store the index of the overlap triangles, it is possible to later substitute an overlap triangle for a split triangle, if the SAH-cost calculation determines that it is more beneficial to keep the split sets of triangles than the overlap sets. The partitioning performed at each recursion is illustrated in Figure 3.

A memory problem arises when it is decided that the split sets should be used. Since there are twice as many triangles in the split sets as there are in the overlap sets, and only half of the split triangles are indexed and can be inserted into the original triangle array. In the first level of split partitioning, before any recursive calls have been made, it is possible to simply append the non-indexed triangles of the right split set S_R to the end of the original partition. However, it doesn't solve the problem with the non-indexed left split triangles. There is no space to place them in-between the pivot and the triangles that belong to the right of the split plane. Thus, as many right triangles as there are non-indexed left split triangles have to be moved to the end of the original partition as well. Then there would be enough space for both the left and right split sets.

Due to the recursive fashion of our algorithm, it becomes more problematic with the succeeding recursions. In most cases, there won't be any empty space available at the end of the partition, and all triangles of one of the halves would have had to been moved to new memory every time the split sets are better than the overlap sets. This memory overhead can be solved by always rebalancing the available empty space relative to the partition sizes and making sure that enough empty space is created in each partition as the algorithm recurses.

Whether the split sets produce a lower SAH cost than the overlap sets or not, we always perform a rebalancing of empty space. In each recursion of the left and right partitions, they are rearranged to supply each side with a fraction of the available empty space in proportion to the current size of the partitions. As an example, after splitting and partitioning, if there is space for 15 additional triangles at the end of the right partition but no space between the left and the right partitions, and the left partition is twice as large as the right partition, then after rebalancing there would be space for 10 additional triangles at the end of the left partition and 5 at the end of the right partition.

If there isn't enough space for both S_L and S_R after split partitioning, then the whole right set $D_R \cup S_R$ has to be moved to a new memory region, with an additional 20% extra padding space as well. The left side gets all the memory left behind by the right side.

We empirically found that the 20% additional space for the partitions works in a robust way for all of our scenes. It is rare that a partition grows with more than a few percent of its size.

In a pathological worst case scenario where all triangles in a scene need to be split, the size of the temporary split set arrays S_L and S_R would need to be as large as the entire scene, but such a scene would cause trouble to any split builder. We found that the actual worst case space needed for the split set arrays across all our scenes is 1% of the triangle count in size. Often they are much smaller than that. For San Miguel the largest temporary split set array is only 0.05% in size relative to the number of triangles of the scene.

Bonsai [7] permits gaps between each final partition and some unused space at the end of the partitions will not affect other stages of the builder. However, when using our splitting algorithm as a pre-split pass, a compaction of the triangle array prior to BVH construction is necessary.

Task parallelism is implemented by recursively spawning new tasks as more partitions are created. We utilize data parallelism when swapping positions of triangles and when computing new bounding boxes.

The total memory allocations needed for our split partitioning algorithm additional to the initial triangle array are a dynamically growing S_L and S_R pair per thread and the dynamically growing memory of the triangle array. Our in-place growing memory partitioning scheme allocates at worst about twice as much additional memory than needed. As an example, the Power Plant scene is built with 19% additional split triangles but allocated space is increased by 39%.

4.2 Bonsai and 8-wide Trees

To maximize ray tracing performance on modern CPUs Embree utilizes 8-wide SIMD instructions (AVX) when ray tracing. However, to efficiently gain data level parallelism while traversing a BVH, Embree builds 4-wide or 8-wide trees for SSE or AVX hardware. We had to modify the Bonsai algorithm to also build 8-wide trees. The initial partitioning of Bonsai doesn't store any BVH node information and is not affected by the fact that the BVH nodes will have eight children. However, mini-tree construction and top tree construction must be adapted to build 8-wide trees. Since both the top tree and the mini-trees use the sweep SAH algorithm, they are modified in the same way. Just like in Embree's binned SAH builder, instead of creating a node once an SAH optimal pivot has been found, the

two halves are placed in a priority queue. While the queue is smaller than the maximum number of children of a node, the element with the largest SAH cost in the queue is chosen for further partitioning. Once the queue has reached the maximum branching size, a BVH node is created and the children are further partitioned recursively.

One issue with the top tree in 8-wide BVH construction that doesn't exist for binary trees, is that it isn't guaranteed that the top tree gets precisely the maximum number of children (which are mini-trees) in its leaf nodes, causing partially filled nodes in the middle of the tree. Initially we thought it would be better to move the gaps to the leaves of the BVH and when a partially filled node was created, the empty space was propagated down the tree. However, we didn't see any tree quality improvement by moving the empty nodes from the middle of the tree to the leaf nodes. It was simply wasted CPU cycles, thus we decided to leave the partially filled nodes as they were.

Triangle splitting is integrated into Bonsai simply by exchanging the mini-tree partitioning with our split partitioning.

5 Results

Our main results have been generated using an Apple Macbook Pro laptop with a quad core Intel 4850HQ CPU. Our primary results are produced by running our triangle split implementations in the Embree ray tracing framework and using the path tracer available in Embree, in benchmark mode, for tree quality measurements. We compare several different construction algorithms with and without splitting and the results are shown in Table 1. For non-split algorithms we include a standard sweep SAH, SweepSAH, the original non-split version of Bonsai (Bonsai), and the binned SAH algorithm included in Embree, BinnedSAH. Our new splitting algorithm is integrated into the Bonsai construction algorithm and denoted BonsaiS. To demonstrate the ability to use our splitting algorithm with other construction techniques, we apply it as a pre-split pass to sweep SAH, SweepPre. We also compare to the two splitting algorithms available in the Embree. The first is a pre-split builder BinnedPre, and the second is a spatial split algorithm BinnedS, based on SBVH [17].

BVH performance comparisons are done using Embree version 2.7.1. It is worth noting that our implementation of SweepSAH is not optimized for build time, although, Ganestam et al. [7] showed that SweepSAH can achieve competitive build performance.

In Table 1 build times are presented in milliseconds and rendering performance is the average time in milliseconds of 10 frames rendered using the path tracer supplied by Embree. A frame is simply a rendering pass with one sample per pixel and many frames need to be accumulated for a final

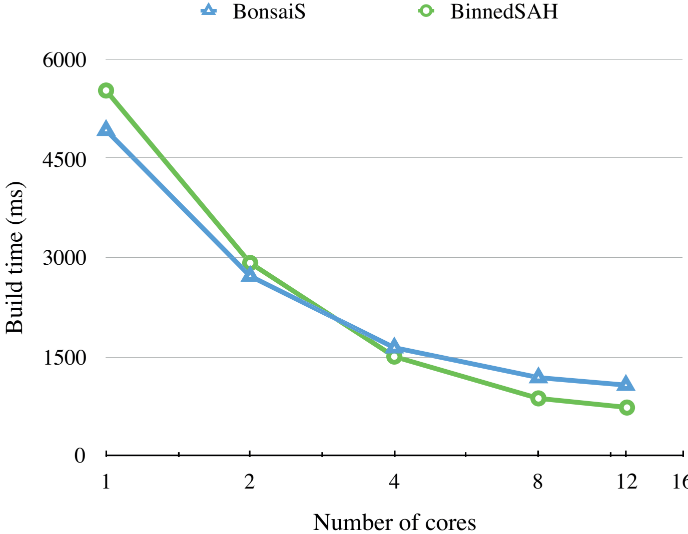


Figure 4: Multicore performance scaling of the Power Plant scene. Although BonsaiS *is close,* BinnedSAH *presents slightly better multicore scaling. Scaling measurements were conducted on a dual socket 12 core CPU. Note the logarithmic x-axis scale.*

image to converge. The more samples and longer time an image need to converge the more beneficial it is with a high quality BVH.

We have marked the two algorithms resulting in the best rendering performance in bold. If more than two algorithms share the best performance percentage, they are all marked. Among the algorithms with the best rendering performance we also marked the one with the fastest build time.

Across our twelve test scenes, BonsaiS is always the fastest triangle split algorithm. Additionally, BonsaiS is among the two best performing algorithms 10 out of 12 times and in four occasions, Bonsai with integrated triangle splitting even out performs BinnedS. SweepPre is among the top performing algorithms in two occurrences, and tends to have a ray tracing performance in between BinnedS and BinnedPre on scenes where triangle splitting is beneficial. BinnedS improves rendering performance on all scenes but one compared to SweepSAH. However, on scenes that don't benefit much from splitting, the performance gain is small, and at the cost of significantly longer build times.

An example of a scene where triangle splitting does not improve rendering performance is Dragon. The reason is that Dragon only contains finely tessellated and evenly distributed triangles, and non-split builders like BinnedSAH can easily minimize the SAH cost.

Figure 4 illustrates multicore performance scaling of the Power Plant scene with BonsaS and BinnedSAH. Scaling measurements were performed on an Intel E5-2643V3 dual socket 12 core CPU. Although BinnedSAH scales slightly better with an increasing number of CPU cores than BonsaiS, it is worth noting that BonsaiS build times are close to BinnedSAH build times and ray tracing performance on the 12 core CPU using BonsaiS is $105ms$ per frame and with BinnedSAH $162ms$ per frame. Since BinnedSAH and BonsaiS present significantly better build performance than BinnedS we omit BinnedS in the multicore scaling comparisons.

Although it is easy to measure memory performance of BonsaiS it is difficult to make thorough comparisons in regards of build time memory usage to the BinnedS algorithm. The source code of BinnedS presents many small memory allocations and memory free instructions. We added a counter to see whether the actual number of allocations were significant or not and found that a large number of allocations were made. However we didn't have the means to measure the maximum memory allocated at any time. To further investigate memory usage we executed both BonsaiS and BinnedS through the well known memory analysis tool Valgrind. Valgrind couldn't tell us what the largest amount of memory allocated at any time was either, but it could inform us about the total number of allocations made by both algorithms. In the tests we made, BinnedS allocated $34\times$ more memory in total compared to BonsaiS and for Arabic City BonsaiS allocated 735MB of memory and BinnedS allocated 25GB of memory, not counting any free instructions.

5.1 Binary Trees

Since Embree only implements 4-wide and 8-wide BVHs we also present results on a subset of our scenes using a standard binary BVH. In this comparison we also use SweepSAH as a baseline but present only one additional algorithm, BonsaiS. The ray tracer used in the second comparison performs ray traversal on the GPU but computes shading on the CPU. The GPU calculations are done on an Intel Iris Pro 5200 integrated graphics processor.

In our Embree implementation, we noticed that the improved rendering times using triangle splitting didn't always match the expected rendering improvement we had seen prior to integrating our methods to Embree. One observation that partially explains the gap in rendering performance between the 8-wide Embree implementation and the 2-wide GPU implementation is that it may be less advantageous for wider trees to perform triangle splitting. We found that if we forced Embree to actually build binary BVHs, although the ray tracer would still use its 8-wide AVX implementation, on some scenes, the reduced rendering time benefit from triangle splitting compared to no splitting would differ with about 5% compared to the splitting

benefit with 8-wide trees. This doesn't fully account for the discrepancy, where as an example Arabic City, in Table 1 using 8-wide trees BonsaiS performs at 84% of SweepSAH, but in Table 2 using 2-wide trees, the rendering time is reduced to 65% of SweepSAH. The rest of the discrepancy can then only be explained by using different hardware or a different ray tracer, or more likely, a combination of the two.

Due to this discrepancy we also present results of a sub set of our test scenes by ray tracing standard binary BVHs. We do this with our own GPU based ray tracer, with shading computations done on the CPU. The improved rendering times in Table 2 can also be put in relation to the rendering speed improvements reported by Karras et al. [11]. For Arabic City their SBVH implementation reduces rendering times to 62% compared to SweepSAH. The same comparison results in 71% for Sponza and 73% for San Miguel. The rendering times of the same three scenes of the fast triangle split builder by Karras et al.[11] are reduced by 74%, 73% and 77% respectively compared to SweepSAH. For Arabic City and Sponza the triangle counts are increased by 50% and for San Miguel the count is increased by 30%. On the same scenes, BonsaiS reduces rendering times to 65% for Arabic City, 75% for San Miguel, and 60% for Sponza, compared to SweepSAH, while having an adaptive split count that is increased by 34% for Arabic City, 6% for San Miguel, and 23% for Sponza. Since the algorithms are designed for and executed on different hardware it is difficult make direct build time comparisons. However, BonsaiS and the triangle split builder by Karras et al. [11] exhibit similar performance improvements, but BonsaiS results in a significantly smaller increase of triangles.

5.2 Triangle counts

Table 3 presents the increase in triangle counts due to splitting triangles. Our splitting algorithm and the binned spatial split builder BinnedS both split triangles adaptively and thus find triangle splits that greedily minimize the SAH costs. The pre-split approach used in Embree depends on a pre-defined splitting budget and thus may split too much or too little in some scenes. Generally, BonsaiS creates fewer additional triangles than the two triangle split methods available in Embree. The pre-split approach in Embree tends to increase the original triangle count by close to 50%. This is because of the split budget that allows a maximum increase of 50%. The algorithm will continue to split large triangles until it is close to its split budget, even though it may not always further improve rendering performance. For the Power Plant model, BinnedS creates more than twice the number of additional triangles than BonsaiS, even so, rendering performance is identical. For San Miguel, the difference in triangle splits is even greater, where BonsaiS only adds 6% additional triangles compared to 45% with BinnedS but BonsaiS results in slightly better rendering per-

formance. Again, we see how dragon isn't affected much by splitting and most of its split computations are discarded.

5.3 Splitting performance

In Table 4 we present the actual time our splitting algorithm takes as a stand alone pre-split approach and the percentage of the BonsaiS build time that is spent on split partitioning. When our method is used with Bonsai the splitting time is merged with the initial mini-tree partitioning. As an example, when using our triangle splitter as a pre-split on San Miguel, the extra build time added is 248ms. Since the splitting algorithm is integrated into Bonsai, and the original partitioning pass of Bonsai takes 133ms for San Miguel, the added build time for BonsaiS in relation to Bonsai is $248 - 133 = 155$ms. Any other extra build time added for BonsaiS and other algorithms using our split method would be from the fact that there are more triangles to process while building the BVH.

5.4 SAH cost

Table 5 shows the SAH cost of the competing algorithms. BonsaiS consistently produces low SAH costs, not always the lowest, but close. Table 5 also demonstrates that SAH cost doesn't necessarily correlate with rendering performance [6, 1, 7]. This is clearly seen on Fairy Forest and San Miguel, where BinnedPre greatly increases the SAH cost without significantly reducing ray tracing performance compared to BinnedSAH.

6 Conclusion

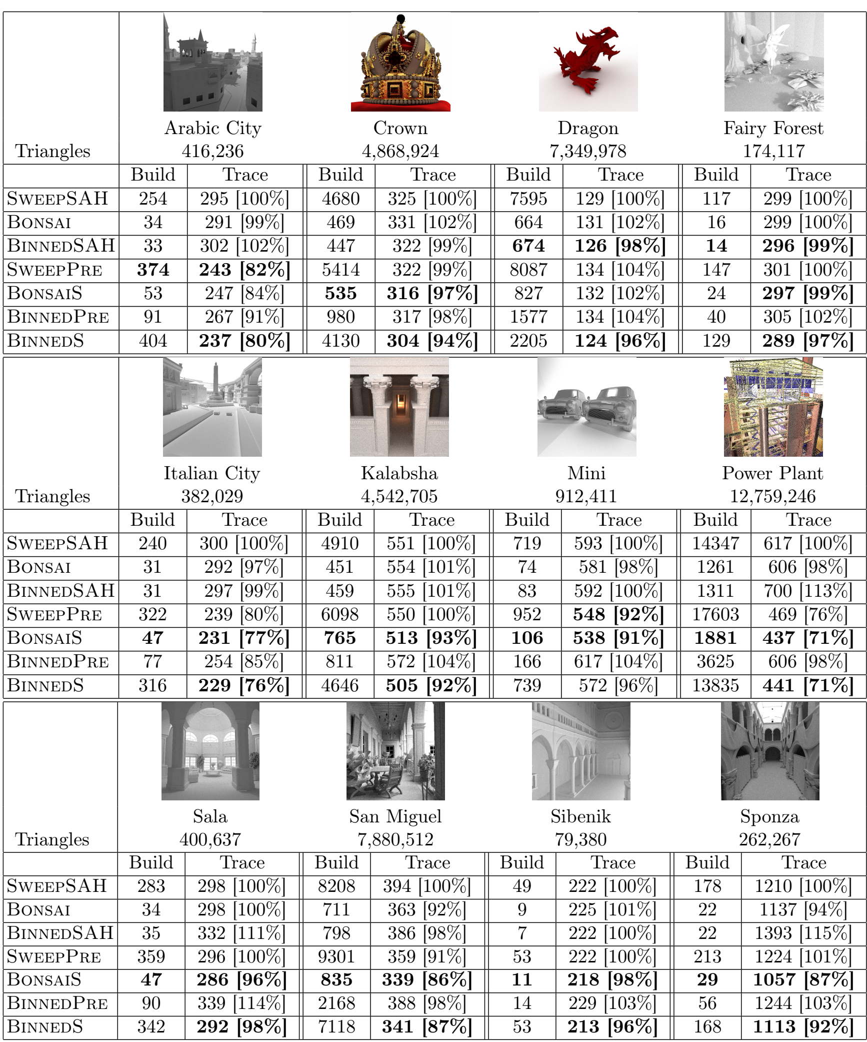
We have presented a new triangle split algorithm for fast BVH construction on multicore CPUs using the latest SIMD extensions. Our triangle split approach paired with Bonsai is always the fastest triangle split builder among the presented algorithms and achieves a rendering performance similar to, and some times better than, the SBVH based spatial split builder available in Embree. We achieve fast build times by utilizing a parallel in-place partitioning scheme for recursively growing data, and improved BVH quality by employing an SAH guided triangle split technique while partitioning. Our algorithm reaches its full potential in regards of both build times and rendering performance when paired with Bonsai. We have also showed that our method works well as a pre-split pass prior to BVH construction, and can easily be added to existing BVH builders without any invasive procedures.

We consider our contributions as continued work towards a high quality real-time BVH construction.

As future improvement, our algorithm could benefit from a more sophisticated parallel approach in the early stages of partitioning. Rather than using only one thread in the first level of recursion, all available threads could work on the same partition until the number of partitions equals the number of hardware threads available.

Acknowledgements

Thanks to ELLIIT and the Intel Visual Computing Institute for funding. We would like to thank Veronica Sundstedt for the Kalabsha temple model and Timo Aila for the Italian and Arabic City models. We would also like to thank Martin Lubich, `http://www.loramel.net`, for the Crown model and Gilles Tran, `http://www.oyonale.com`, for the Mini model.



Triangles	Arabic City 416,236		Crown 4,868,924		Dragon 7,349,978		Fairy Forest 174,117	
	Build	Trace	Build	Trace	Build	Trace	Build	Trace
SWEEPSAH	254	295 [100%]	4680	325 [100%]	7595	129 [100%]	117	299 [100%]
BONSAI	34	291 [99%]	469	331 [102%]	664	131 [102%]	16	299 [100%]
BINNEDSAH	33	302 [102%]	447	322 [99%]	**674**	**126 [98%]**	**14**	**296 [99%]**
SWEEPPRE	**374**	**243 [82%]**	5414	322 [99%]	8087	134 [104%]	147	301 [100%]
BONSAIS	53	247 [84%]	**535**	**316 [97%]**	827	132 [102%]	24	**297 [99%]**
BINNEDPRE	91	267 [91%]	980	317 [98%]	1577	134 [104%]	40	305 [102%]
BINNEDS	404	**237 [80%]**	4130	**304 [94%]**	2205	**124 [96%]**	129	**289 [97%]**

Triangles	Italian City 382,029		Kalabsha 4,542,705		Mini 912,411		Power Plant 12,759,246	
	Build	Trace	Build	Trace	Build	Trace	Build	Trace
SWEEPSAH	240	300 [100%]	4910	551 [100%]	719	593 [100%]	14347	617 [100%]
BONSAI	31	292 [97%]	451	554 [101%]	74	581 [98%]	1261	606 [98%]
BINNEDSAH	31	297 [99%]	459	555 [101%]	83	592 [100%]	1311	700 [113%]
SWEEPPRE	322	239 [80%]	6098	550 [100%]	952	**548 [92%]**	17603	469 [76%]
BONSAIS	**47**	**231 [77%]**	**765**	**513 [93%]**	**106**	**538 [91%]**	**1881**	**437 [71%]**
BINNEDPRE	77	254 [85%]	811	572 [104%]	166	617 [104%]	3625	606 [98%]
BINNEDS	316	**229 [76%]**	4646	**505 [92%]**	739	572 [96%]	13835	**441 [71%]**

Triangles	Sala 400,637		San Miguel 7,880,512		Sibenik 79,380		Sponza 262,267	
	Build	Trace	Build	Trace	Build	Trace	Build	Trace
SWEEPSAH	283	298 [100%]	8208	394 [100%]	49	222 [100%]	178	1210 [100%]
BONSAI	34	298 [100%]	711	363 [92%]	9	225 [101%]	22	1137 [94%]
BINNEDSAH	35	332 [111%]	798	386 [98%]	7	222 [100%]	22	1393 [115%]
SWEEPPRE	359	296 [100%]	9301	359 [91%]	53	222 [100%]	213	1224 [101%]
BONSAIS	**47**	**286 [96%]**	**835**	**339 [86%]**	**11**	**218 [98%]**	**29**	**1057 [87%]**
BINNEDPRE	90	339 [114%]	2168	388 [98%]	14	229 [103%]	56	1244 [103%]
BINNEDS	342	**292 [98%]**	7118	**341 [87%]**	53	**213 [96%]**	168	**1113 [92%]**

Table 1: BVH build time and rendering performance measurements across all scenes and algorithms. Build times are reported in milliseconds and ray tracing performance in milliseconds per frame. The reported frame time is the average of ten frames rendered with the path tracer available in Embree, set to benchmark mode. The reported build times are generated by taking the minimum build time when running each builder 20 times. In regards of both build times and rendering performance, lower is better. The two algorithms that result in the highest ray tracing performance are marked, and so is the fastest build time among those with the best rendering performance.

	Arabic City	Crown	Italian City	Kalabsha	San Miguel	Sponza
SweepSAH	240 [100%]	191 [100%]	129 [100%]	455 [100%]	667 [100%]	630 [100%]
BonsaiS	155 [65%]	183 [96%]	86 [67%]	330 [73%]	499 [75%]	380 [60%]

Table 2: Rendering performance comparison of SweepSAH *and* BonsaiS *using binary BVHs and on a GPU based ray tracer. Measurements are reported in milliseconds per frame.*

	BonsaiS	BinnedPre	BinnedS
Arabic City	34%	47%	61%
Crown	9%	46%	34%
Dragon	2%	46%	1%
Fairy Forest	19%	46%	24%
Italian City	27%	48%	59%
Kalabsha	11%	22%	46%
Mini	24%	46%	42%
Power Plant	19%	47%	50%
Sala	19%	45%	44%
San Miguel	6%	50%	45%
Sibenik	19%	46%	29%
Sponza	23%	45%	25%

Table 3: The triangle count increase due to splitting. BonsaiS *represents the counts for* SweepPre *as well, since they use the same splitting algorithm.* BinnedSAH *with pre-split always stays close to a 50% increase, since this is the allowed splitting budget. Our method and* BinnedS *are both adaptive split techniques, and reduce the probability of keeping unnecessary splits.*

	Split time (ms)
Arabic City	13 [25%]
Crown	151 [28%]
Dragon	294 [36%]
Fairy Forest	11 [46%]
Italian City	11 [23%]
Kalabsha	394 [52%]
Mini	30 [28%]
Power Plant	844 [45%]
Sala	17 [36%]
San Miguel	248 [30%]
Sibenik	4 [36%]
Sponza	8 [28%]

Table 4: The time in milliseconds our triangle splitting approach takes when used as a pre-split pass. In brackets we show the time spent on split partitioning relative to full BonsaiS *build times. On split friendly scenes, such as Italian City, the relative time spent on splitting is lower than on less split friendly scenes. This is inherent, since all triangles that cross the split plane have to be considered as a split candidate, even if the split never is used. However, if the number of triangles used by the BVH builder isn't increased, the build times are lower than they would be if the splits were used. Thus increasing the ratio between triangle split time and BVH build time.*

	SweepSAH	SweepPre	Bonsai	BonsaiS	BinnedSAH	BinnedPre	BinnedS
Arabic City	23.23	17.21	22.43	17.10	23.83	20.38	**16.80**
Crown	7.91	**7.48**	7.77	7.77	7.93	7.79	7.58
Dragon	6.22	6.16	**6.02**	6.48	6.50	7.44	6.50
Fairy Forest	9.50	9.89	**9.42**	9.63	9.76	15.17	9.82
Italian City	17.94	13.83	17.49	**13.64**	19.05	17.27	13.66
Kalabsha	2.31	2.34	**2.24**	2.42	3.42	6.55	3.39
Mini	5.10	5.16	**4.94**	5.31	6.19	9.57	5.61
Power Plant	10.02	8.34	9.83	8.17	10.67	9.37	**7.32**
Sala	10.43	10.41	10.67	**10.00**	12.05	13.08	10.61
San Miguel	18.86	16.92	19.12	16.00	19.75	24.45	**15.99**
Sibenik	15.03	14.38	14.59	14.06	14.78	16.02	**13.45**
Sponza	22.53	22.25	22.61	**20.78**	24.59	29.26	21.65

Table 5: SAH costs of all scenes and construction methods. For each scene, the builder with the lowest cost is marked with bold text. BonsaiS *and* BinnedS *tend to have the lowest SAH costs, but no algorithm is consistently lowest. The C_i and C_t constants used to evaluate Eq. 1 are set to one and C_l is zero.*

Bibliography

[1] Timo Aila, Tero Karras, and Samuli Laine. On Quality Metrics of Bounding Volume Hierarchies. In *High-Performance Graphics*, pages 101–107, 2013.

[2] H. Dammertz, J. Hanika, and A. Keller. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum,*, 27(4):1225–1233, 2008.

[3] Holger Dammertz and Alexander Keller. Edge volume heuristic - robust triangle subdivision for improved BVH performance. In *Proc. 2008 IEEE/EG Symposium on Interactive Ray Tracing*, pages 155–158, 2008.

[4] Leonardo R. Domingues and Helio Pedrini. Bounding Volume Hierarchy Optimization through Agglomerative Treelet Restructuring. In *High-Performance Graphics*, pages 13–20, 2015.

[5] Manfred Ernst and Günther Greiner. Early Split Clipping for Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 73–78, 2007.

[6] Bartosz Fabianowski, Colin Flower, and John Dingliana. A cost metric for scene-interior ray origins. In *Eurographics Short Papers*, pages 49–52, 2009.

[7] Per Ganestam, Rasmus Barringer, Michael Doggett, and Tomas Akenine-Möller. Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees. *Journal of Computer Graphics Techniques,*, 4(3):23–42, September 2015.

[8] Jeffrey Goldsmith and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics & Applications,*, 7(5):14–20, 1987.

[9] Vlastimil Havran and Jiri Bittner. On Improving KD-Trees for Ray Shooting. In *Winter School on Computer Graphics*, pages 209–217, 2002.

[10] James T. Kajiya. The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, volume 20, pages 143–150, 1986.

[11] Tero Karras and Timo Aila. Fast Parallel Construction of High-quality Bounding Volume Hierarchies. In *High-Performance Graphics Conference*, pages 89–99, 2013.

[12] Timothy L. Kay and James T. Kajiya. Ray Tracing Complex Scenes. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, volume 20, pages 269–278, 1986.

[13] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum,*, 28(2):375–384, 2009.

[14] David J. MacDonald and Kellogg S. Booth. Heuristics for Ray Tracing Using Space Subdivision. *Visual Computer,*, 6(3):153–166, 1990.

[15] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation.* MKP, 2nd edition, 2010.

[16] Stefan Popov, Iliyan Georgiev, Rossen Dimov, and Philipp Slusallek. Object Partitioning Considered Harmful: Space Subdivision for BVHs. In *High-Performance Graphics*, pages 15–22, 2009.

[17] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial Splits in Bounding Volume Hierarchies. In *High-Performance Graphics*, pages 7–13, 2009.

[18] Ingo Wald. On Fast Construction of SAH-based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 33–40, 2007.

[19] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics,*, 33(4):143:1–143:8, 2014.

[20] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM,*, 23(6):343–349, 1980.