Energy-Efficient Interactive Ray Tracing of Static Scenes on Programmable Mobile GPUs

by

Peter James Lohrmann

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

February 2007

APPROVED:

Dr. Emmanuel Agu, Advisor _____

Dr. Robert Lindeman, Advisor _____

Dr. Matthew Ward, Reader _____

Dr. Michael Gennert, Head of Department _____

Abstract

Mobile technology is improving in quality and capability faster now than ever before. When first introduced, cell phones were strictly used to make voice calls; now, they play satellite radio, MP3s, streaming television, have GPS and navigation capabilities, and have multi-megapixel video cameras. In the near future, cell phones will have programmable graphics processing units (GPU) that will allow users to play games similar to those currently available for top-of-the-line game consoles. Personal digital assistants enable users with full email, scheduling, and internet browsing capabilities in addition to those features offered on cell phones. Underlying all this mobile technology and entertainment is a battery whose technology has just barely tripled in the past 15 years, compared to available disk capacity that has increased over 1,000-fold.

Ray tracing is a rendering technique used to generate photorealistic images that include reflections, refraction, participating media, and can fairly easily be extended to include photon mapping for indirect illumination and caustics. In recent years, ray tracing has been implemented on the GPU using various acceleration structures to facilitate rendering. Until now, all studies have used build time and achievable frame rates to determine which acceleration structure is best for ray tracing. We present the very first results comparing both CPU and GPU raytracing using various acceleration structures in terms of energy consumption. By exploring per-pixel costs, we provide insight on the energy consumption and frame rates that can be experienced on cell phones and other mobile devices based on currently available screen resolutions.

Our results show that the choice in processing unit has the greatest affect on energy and time costs of ray tracing, followed by the size of the viewport used, and the choice of acceleration structure has the least impact on efficiency. For mobile devices enabled with a programmable GPU, whether it is a cell phone, PDA, or laptop computer, a bounding volume hierarchy implemented on the GPU is the most energy-efficient acceleration structure for ray tracing. Ray tracing on cellular phones with smaller screen resolutions is most energy-efficient using a CPU-based Kd-Tree implementation.

Acknowledgements

I would like to thank my advisors, Professor Emmanuel Agu and Professor Robert Lindeman, for their guidance throughout this thesis. Professor Agu has done countless hours of research behind the scenes to find the best papers to shape my learning and understanding of computer graphics and ray tracing. He has shown me the meaning of research and taught me how to approach future endeavors. Professor Lindeman has allowed me and several other students to make a home in the HIVE, where we spent late nights debugging traversal code and shader errors. His creativity and excitement for discovery has inspired me to seek out new challenges that I may have otherwise let slip away. My interests have only grown from working with these two individuals.

I would also like to thank Chen-Hao (Jason) Chang and Brandon Light for their help through the coding of ENCORE and the motivation that they unknowingly provided me. We've had lots of frustrating times, but it all came together when we needed it to.

Finally, my family – I cannot thank you enough for putting up with my busy schedule and weeks between visits and phone calls. You've always supported and encouraged me, and for that I am forever grateful.

Table of Contents

List of Figures

List of Tables

List of Equations

1 Introduction

Over the past decade, graphics research has focused on making physically-based rendering algorithms, such as ray tracing, interactive. Physically-based algorithms implemented on the CPU can take minutes or even hours to process the scene and generate high-resolution images. This is contrary to the demands of interactive environments, where minimum frame rates are typically between 2 and 15 frames per second depending on the level of interactivity desired. In recent years however, new, powerful Graphics Processing Units (GPUs), which can process millions of triangles per second, have helped bring ray tracing, the industry standard for photorealistic image rendering, into the realm of interactive graphics.

The rapid rate at which these graphics cards have been improving is evident in Figure 1-1, where the number of floating point operations per second (FLOPS) capable on ATI and NVIDIA graphics cards are compared to that of a Pentium 4 CPU [Buck 2004]. The GPU is a parallel pipelined system designed to perform similar operations on large amounts of input. Thus, it can process more data per second than the CPU. The rate at which these cards are increasing in performance, and the recently added capability to program them, has inspired much research into utilizing the capabilities of the GPU for general-purpose computations [GPGPU].

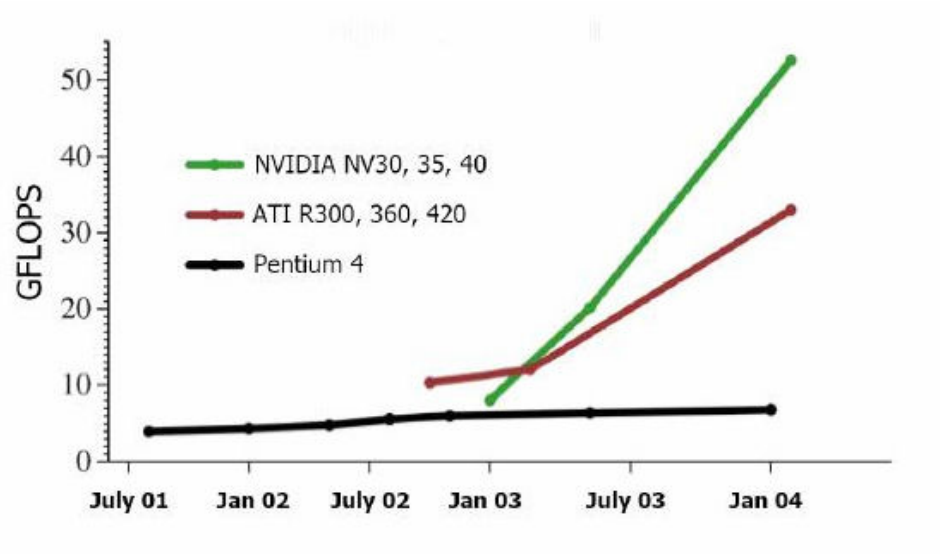

Figure 1-1: GPU vs. CPU Trends [Buck 2004]

As graphics processing speed increases, there is a corresponding decrease in the physical size of components based on the technology. These programmable GPUs are even being included in personal digital assistants (PDAs) and cell phones. Although these devices are faster and more powerful than in the past, they are still resource-limited and require that real-time graphics algorithms be reformulated. The per-pixel processing power necessary for real-time graphics is available on these mobile devices due to their low-resolution screens. With external memory cards that can store several gigabytes of information, the memory needed to store the geometric models is also not a limiting factor. In fact, the limiting resource is battery power. Intel [2002] suggests that nearly half of the total power consumption of a mobile device is due to the display and graphics card. In such an environment, the graphics software should be optimized to reduce power consumption while taking advantage of the card's capabilities.

Battery life has been among the slowest technologies to advance in mobile devices. Figure 1-2 shows the relative improvements in various PC technologies from 1990 through 2001. Where disk capacity and CPU speed have increased 1000 and 400 fold respectively, battery energy density has only doubled [Starner 2003].

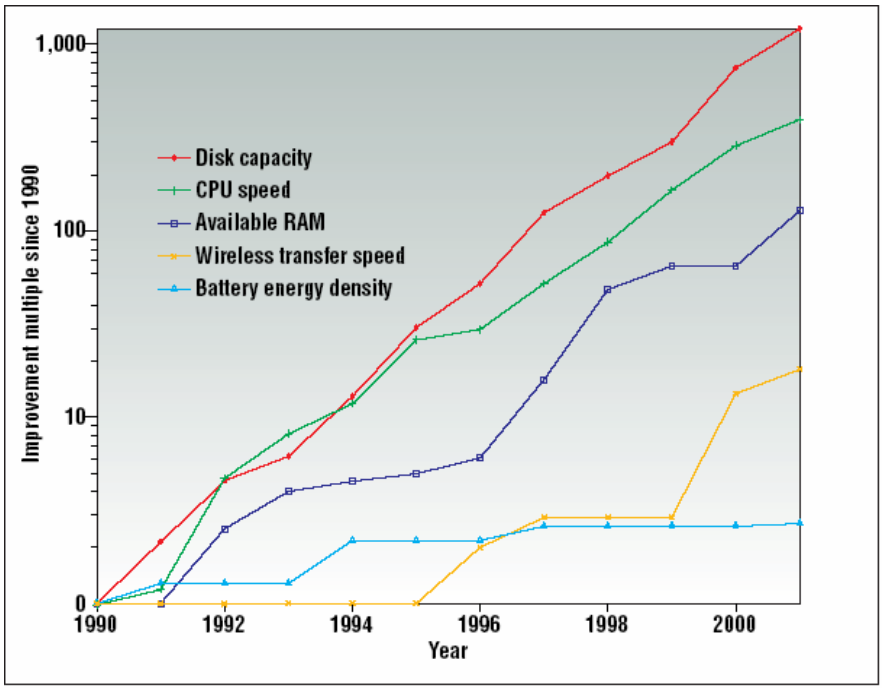

Figure 1-2: Technology trends from 1990 to 2001 [Starner 2003]

In the past, there has not been a strong push for improved battery-life of portable devices due, in part, to their history of limited functionality. However, the addition of email, internet, video conferencing, and music playing capabilities, turning previously single-feature devices into multi-functional tools, has increased the popularity of mobile devices. As new features are added, the hardware must become more flexible, yet maintain the speed that people have grown accustomed to. Increasing speed and reducing time costs has been part of the nature of research, as is evident in computer graphics, where the quality of rendering systems has only considered the number of frames per second that can be produced. On mobile devices, where resources are more limited than on a desktop computer, energy consumption should also be an important consideration. Mobile resources should be optimized while maintaining the ability to produce high quality images.

1.1 Goal of the Thesis

The programmability of new graphics processing units has allowed ray tracing, the industry standard for photorealistic images, to be performed at interactive frame rates [Purcell et al 2002]. We explore the use of ray tracing, implemented on both the CPU and GPU, as a means of producing high-quality images on mobile devices, where energy-efficiency is a concern. The use of acceleration structures to increase rendering speeds by reducing the number of ray-triangle intersection tests is a necessary extension to ray tracing. However, improved speed could also mean increased energy consumption. Research by Barr and Asanovic [2003] reported that a memory access on the CPU uses 200 times the power of a single computation. Thus, formatting the data into an acceleration structure, and accessing it from textures on the GPU, may result in unexpected levels of energy-consumption. We compare the energy-efficiency of uniform grids, Kd-Trees, and bounding volume hierarchies in CPU-based and GPU-based ray tracing systems implemented on a laptop computer. With the growing number of graphics-intensive resource-limited mobile devices in use today, an energy-efficient rendering engine will have widespread impact.

1.2 Organization of the Thesis

An understanding of the graphics processing unit, the different stages of the fixed-function and programmable pipelines, and the limitations imposed by the GPU are beneficial to understanding the approach we have taken. Chapter 2 describes the graphics processing unit along with the necessary features that enable it as a processor for ray tracing.

Ray tracing is a complex rendering algorithm to understand. Chapter 3 covers the basics of ray tracing, along with how it is performed on the CPU and the changes that must be made to implement it on the GPU. As mentioned, ray tracing was formerly an off-line rendering process and considered a very time-consuming algorithm, however, several acceleration structures were designed to make ray tracing faster. Among the structures that have been designed, the uniform grid, Kd-Tree, and bounding volume hierarchy are among the most common, and for this reason we have chosen to measure the energy-efficiency of these structures. The CPU and GPU-based implementations of these structures are described in Chapter 4.

Chapter 5 covers the scenes chosen to be included in the testing, along with the interface that enabled us to query the drain rate of the battery, and the steps taken to ensure that the results were not distorted by other processes running on the test machine. The results obtained from these tests and how they are applied to various mobile devices is presented in Chapter 6. Finally, Chapter 7 summarizes the conclusions that can be drawn from our results and suggests future work in the area of energy-efficient ray tracing.

2 Graphics Processing Units

A graphics processing unit (GPU) is the hardware behind a display device that physically controls the image that is being displayed. In an early form, the GPU was only capable of controlling pixel color, a process sometimes referred to as 2D rasterization or vector graphics. The enabling of pixels has to happen very quickly, and to do so, the GPU is based on a parallel architecture that can process many pixels at the same time. Early work on the GPU focused on scientific visualizations [SGI 2004], but as popularity grew and complex game development began, the GPU was enabled with 3D acceleration with the release of OpenGL 2.0 [SGI ARB 2004]. This allowed software to send groups of vertices to the GPU which would be positioned as points, lines, or triangles in a virtual three-dimensional space, and then projected onto the screen, coloring the appropriate pixels. Soon after, support for textures and lighting was added. This level of functionality became known as the fixed-function pipeline. Many graphics effects which make simple geometry appear much more complex can be implemented with the fixed-function pipeline using preprocessed textures. However, the desire for greater control over the processes performed in the GPU led way to the programmable pipeline.

2.1 Fixed-Function Pipeline

Although any graphics card can be considered to be a fixed-function pipeline, the most common pipeline used for gaming or advanced rendering techniques will have 3D acceleration. This allows an application to specify geometry with three-dimensional vertices to be rendered on the screen. The fixed-function pipeline (Figure 2-1) is a way of describing the general path that vertices follow as they travel through the graphics hardware and become colored pixels on the screen [Wright and Lipchak 2005].

Geometry data, including vertex positions, normals, colors, and texture coordinates, among other attributes, are transferred as arrays from the application into the GPU as *geometry data*. In this first stage, all the attributes arrays are broken apart and each vertex, along with its corresponding attribute data, is gathered together and sent along the pipeline. The next stage is referred to as the *Transform & Lighting* stage. It can be imagined that at this stage the vertices are positioned (transformed) into camera space

based on their XYZ coordinates. The vertices' material properties and orientation relative to lights in the scene are used to calculate the color of the geometry at each vertex. These color values, along with texture coordinates and other attributes are then interpolated along the edges of the geometry as the vertices are assembled into primitives in the third stage. The primitives that can be rendered using the fixed-function pipeline are points, lines, triangles, and polygons. At this same stage, the primitives are put into context with the camera position, and those primitives that are not in view of the camera are culled.

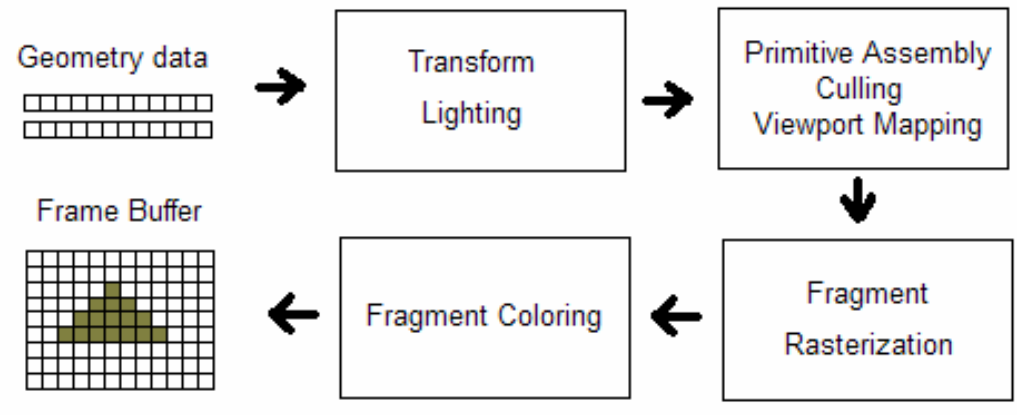

Figure 2-1: The fixed-function pipeline

Those primitives that are in view get projected (mapped) onto the viewing plane. *Fragment Rasterization* uses depth information of the projected primitives to identify the front-most primitive to the camera. Unless special depth settings are configured, primitives that are occluded from view by other primitives are not drawn. The pixels that will display a particular primitive are collectively referred to as a fragment. Now that a fragment is mapped to each visible primitive, the texture coordinates and other interpolated data are used for *Fragment Coloring*. Among other techniques, sampling textures, alpha blending, and stenciling can be used to produce the final pixel colors to be displayed on the screen. The color of each pixel is then copied into the *frame Buffer,* which is the output of the fixed-function pipeline, and displayed on the screen.

2.2 Programmable Pipeline

The programmable pipeline is similar to the fixed-function pipeline shown in Figure 2-1, but programmability is introduced into the *Transform & Lighting* and *Fragment Coloring* stages. At each of these stages, the programmer has the option to take advantage of the programmability or can default to using the standard fixed-function operations. The first of the two new stages is the Vertex Shader, which replaces the *Transform & Lighting* stage. The second area of programmability replaces the *Fragment Coloring* stage, and is referred to as the Fragment Shader (or Pixel Shader). Since the GPU is designed for displaying pixels, the Fragment Shader has a lot more processing power than the Vertex Shader. The nVidia GeForce GO 7800 used for these experiments has 24 fragment shader processors, compared to only 8 vertex shader processors.

The pieces of code that get loaded into the Vertex Shader and Fragment Shader on the GPU are called shaders. Since the GPU is a parallel architecture, a shader will actually be executed on many vertices or pixels simultaneously. The flexibility of these shaders has enabled many new rendering effects and has opened the doors for a new field of research called GPGPU, which is general purpose computation on the GPU [www.gpgpu.org]. This area focuses on taking advantage of the GPU's highly parallel architecture to perform calculations that do not involve triangle vertex processing in the manner in which the GPU was intended; such areas include financial calculations, signal processing, database querying, and image processing. The GPU is designed to process triangles and produce what is known as *raster graphics* by rasterizing triangles to the screen. Ray tracing on the GPU does not render images in this manner; for this reason, it is categorized under GPGPU. The triangle vertices are encoded in texture memory and the shaders are written to interpret the texture data as triangles; the shaders could be modified to interpret the data as spheres or other parametric surfaces to be ray traced on the GPU.

2.2.1 Vertex Shaders

The Vertex Shader (VS) replaces the *Transform & Lighting* section of the fixed-function pipeline and thus controls the method in which vertices are projected into camera space. Having control over this stage allows a programmer to algorithmically

relocate the vertices and can even decide which vertices should be visible in the final image. Additionally, it is possible to calculate vertex colors, normals, and texture coordinates among other values to be used throughout the remainder of the pipeline. Aside from the vertex attributes, the application can update variables between frames and send them into the VS as *uniforms*. A *uniform* variable is one that remains the same for all the vertices rendered in a given frame. Because the programmer has control over the use of each vertex attribute and *uniform* value, they can be used to send other data into the VS.

In our implementation of the GPU-based ray tracer, the fixed function pipeline is used in place of a programmable vertex shader. The only actual vertices that are rendered in the application correspond to two triangles that form a screen-aligned quad (a square that covers the whole window). Ray tracing is implemented in the fragment shaders to generate the image that is displayed on this square. Once the ray traced image is rendered to a texture, it can then be applied to a non-screen-aligned quad, as shown in Figure 2-2.

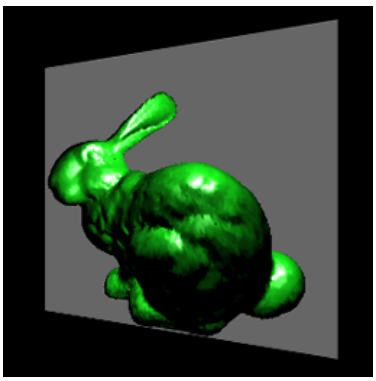

Figure 2-2: Ray traced image on a non-screen-aligned quad

2.2.2 Fragment Shaders

The Fragment Shader (FS) replaces the *Fragment Coloring* section of the fixed-function pipeline. The input to the FS comes from the output of the VS which are interpolated based on the position of the pixel on the fragment. Similar to the VS, input can also come from *uniform* variables passed in by the application. In addition to the *uniforms*, however, the FS can also get information via textures that are loaded into

texture memory. The output of the FS is either no color, or a set of one or more colors. The Fragment Shader is allowed to *discard* pixels, which means that a color will not be output for that pixel; this is particularly useful for GPGPU applications. Normally, the FS will output one or more colors which will be written to a render target. The render targets can include the back buffer (frame buffer), which will be displayed on the screen, or renderable textures, which can be used as input to other shaders.

The GPU-based ray tracer is implemented entirely using fragment shaders, with some minor control flow performed by the CPU. The details of the implementation are described in Section 3.3. Necessary textures (such as those containing the vertex information) are also constructed on the CPU. With the exception of the final fragment shader which calculates the final color of the pixel, all of the shaders have multiple output values that are written to renderable textures. The implementation of the renderable textures is based on frame buffer objects which are described in Section 2.3.

2.2.3 Limitations

The GPU currently has several technical limitations that have to be dealt with in implementing ray tracing. Although the GPU allows for conditional statements, looping, and claims to support infinite length shaders, there are still limitations in these regards. The most prominent limitation imposed by the GPU is the number of loops that can be performed in a single execution of the shader. A loop is only allowed to iterate 256 times before it is forced to stop. To get around this limit, Thrane and Simonsen [2005] suggest nesting the loop inside of a loop with the exact same conditions, thus forcing the inner loop to execute again. They claim that this will allow the loop to iterate 2^{16} times, however we found that the outer loop was only executed ten times, allowing 2560 iterations in a single pass. This discrepancy could be due to the use of different hardware and different drivers. We believe it to be the result of the shader compiler attempting to unroll the loops in hopes of optimizing the resulting code, but perhaps due to memory constraints, only a fixed number of loops can be unrolled. This is only speculation however, and we have been unsuccessful in finding support for the reason behind this limitation. Although iterating 2560 times was sufficient for some tests, in some cases the limitation prevented rays from fully traversing the acceleration structure. It is possible to

work around this issue by saving some state information, and using occlusion queries to determine whether or not the shader should be executed again [Purcell et al 2002].

Another GPU limitation that is worth noting is the maximum size of texture width and height. This limit is based on the hardware being used and is set at 4096x4096 for the nVidia GeForce GO 7800. This limitation is important because the geometry's vertex data is transferred to the GPU using textures. The implication of this maximum size with regards to ray tracing is that a limit exists on the number of triangles that can be handled, and the size of the acceleration structures. We did not experience any issues caused by this limitation, but it exists and could be an issue if we attempted to ray trace more detailed or larger scenes.

2.2.4 The Future

Available as part of the recently released DirectX 10 specification is another section of programmability called the Geometry Shader which resides after the *Primitive Assembly* stage, but before the Fragment Shader. Programs written for this section of the pipeline have control over entire pieces of geometry including lines, triangle strips, fans, and the connecting vertices. The ability to generate new geometry or to remove pieces of geometry allow for a wide variety of new techniques on the GPU.

2.3 Framebuffer Objects

The framebuffer object (FBO) was approved by ARB Working Group on January 31, 2005 [EXT_framebuffer_object 2005] as an extension to OpenGL version 1.1. It manipulates off-screen memory that provides a means of rendering to a texture that can then be used as input to another shader pass. Previously, an implementation using pixel buffers (pbuffers [ARB_pixel_buffer_object 2004]) was used, but requires that the pixels be copied from the framebuffer into texture memory using calls in the application. This texture copying was a very slow process, but using FBOs the output from one pass may be written directly to texture memory, allowing the data to quickly become the input of another pass. FBOs can also be generated on the CPU and filled with data from a one-dimensional array of float values. This provides us with a single interface for writing

acceleration structure and geometry data to texture memory and transferring data between shader passes.

FBOs also allow the use of non-power-of-two textures, which is important in being able to minimize texture memory consumption which may be limited on mobile devices. A power-of-two texture requires that the dimension along the X and Y dimension be powers of two; in a non-power-of-two texture, these dimensions can be any positive integer. In both cases, there is a limit on the maximum size that the texture may be and the texture must be at least rectangular, if not square. Figure 2-3 portrays how a power-of-two-texture requires additional unused data (shown in grey) to be inserted to pad the texture over a non-power-of-two texture. The padding that is performed on the non-power-of-two texture is done to ensure that the texture is rectangular.

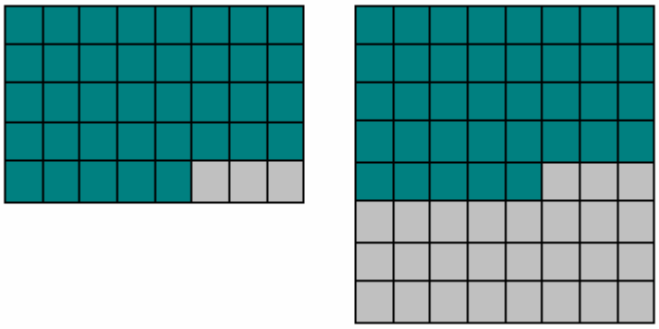

Figure 2-3: The difference in unused data (grey) between a non-power-of-two texture (left) and a power-of-two texture (right).

3 Ray Tracing

Ray tracing is a physically-based rendering technique that can produce photorealistic images by simulating the way light moves around a scene and is interpreted by the eye. It involves casting rays out from a virtual camera and tracing them through a scene to determine the image that the camera sees based on the materials and textures of the objects in view. Ray tracing can be extended with photon mapping to incorporate indirect lighting, atmospheric effects, soft shadows, and caustics. Other rasterization techniques exist that attempt to make these types of effects appear real, but usually they are approximations at best, or otherwise may be an artist's hand-painted texture that represents how they expect the shadow or caustic to appear.

Behind ray tracing are mathematical equations describing how light reflects off of surfaces and determining if a ray intersects with the objects in the scene. There are a relatively small number of steps that need to be utilized to perform ray tracing, and every ray has to perform these steps many times. Since the processing of each ray is almost identical, this rendering technique can be easily mapped from the CPU to the GPU, which has an advantageous parallel architecture.

3.1 Basic Ray Tracing

Ray tracing is similar to looking through a screened window and seeing the world on the other side. In real life, the light is emitted from the sun, bounces off all the objects outside, comes through the many holes in the screened window, and into your eye. The opposite processes, of looking from your eye through each of the holes in the screened window at all the sunlight bouncing from one object to another, is like ray tracing. The difference of course, is that the sunlight and the objects on the other side of the screen (or computer monitor instead of window) are just geometric, mathematical shapes. The origin of the rays can still be thought of as being your eye, but it can also be referred to as a camera, and instead of looking through each hole in the screened window, a ray is sent through each pixel of the monitor. The monitor becomes your window into a virtual world.

The rays that originate from the camera are called primary rays. As these primary rays travel through the pixels and into the scene, there are many objects that they could intersect. The color of a particular pixel is determined by the material properties of the closest object that a ray hits through that pixel. These material properties do not just include color, but can also describe the roughness, reflectivity, and translucence of the object. Based on these values, secondary rays may be spawned into the scene. These secondary rays determine the colors that might be reflected off of the intersected object, or that might be visible through the object if it is transparent. Similarly, the reflected object or background object could have material properties that cause it to be reflective or transparent, and thus more rays can be spawned. Additionally, at each of these ray-object intersections, the object may or may not be in shadow based on the location of emissive objects (i.e. objects which emit light) and other objects in the scene. To determine if it is in shadow, shadow rays are spawned in the direction of the lights and are solely used to identify if there is something obscuring the object from the light source.

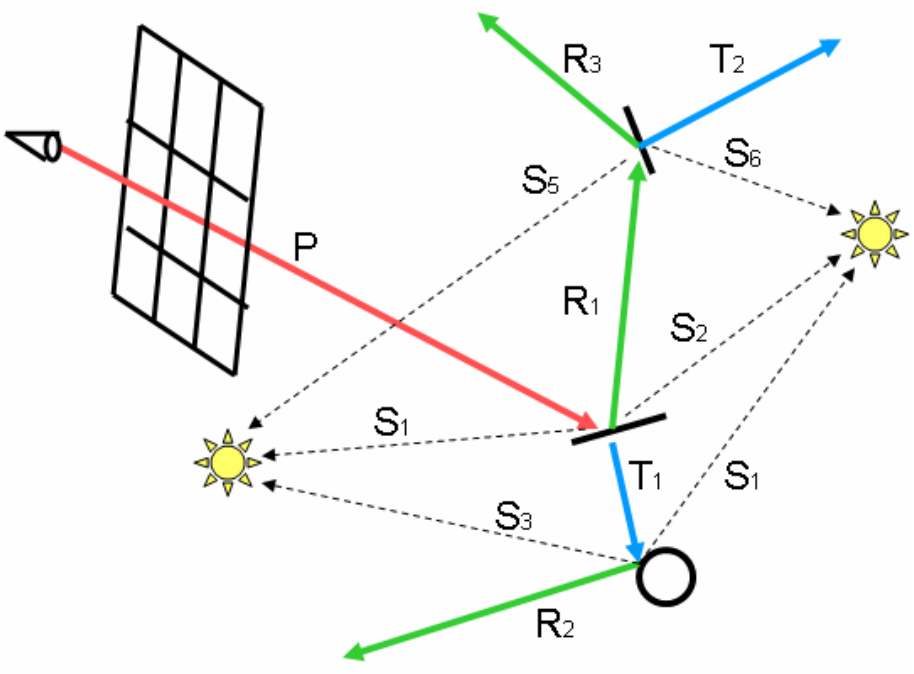

Figure 3-1: Primary, reflected, refracted, and shadow rays

Figure 3-1 visualizes the different types of rays that are used while ray tracing. The primary ray is labeled ‘P’ and is colored red, reflection rays ‘R’ are colored green,

translucence rays ‘T’ (also referred to as refraction rays) are blue, and shadow rays are labeled ‘S’. Each of the labels also has a subscript that indicates in what order each of the different types of rays is generated. Based on these numbers, the primary ray intersects with the center object (producing four secondary rays), the ‘T1’ ray then intersects with a ball which produces three ternary rays. Since the ‘R2’ ray does not intersect with anything, the intersection caused by the ‘R1’ ray is explored, which produces another four ternary rays. Neither ‘R3’ nor ‘T2’ intersect with objects, so shading for primary ray ‘P’ is complete. From this primary ray, a total of eleven other rays were spawned. The exponential growth in the number of rays spawned with each subsequent object hit demonstrates why ray tracing has been previously considered an off-line or preprocessed rendering technique.

To simplify the necessary calculations to determine a ray-object intersection, only triangles are supported in our implementation. We have implemented the ray-triangle intersection algorithm described by Moller and Trumbore [1997]. This algorithm can be ported for use on the GPU, taking advantage of the vector math, as shown by Thrane and Simonsen [2005]. Pseudocode for the algorithm is shown in Figure 3-2. In a naïve approach to ray tracing, the intersection algorithm will test every ray against every triangle in the scene. Due to the number of rays that can be spawned in a scene, this naïve approach can result in the ray-triangle intersection algorithm being performed millions of times for a relatively simple scene. Acceleration structures were designed to reduce the total number of ray-triangle intersections that must be performed by grouping triangles together into a bounding region. The computations to traverse a ray through these bounding regions are less complicated than the ray-triangle intersection algorithm, which computes several expensive dot products and cross products in both the CPU and GPU-based implementations.

```
HitInfo RayTriangleIntersect( vert1, vert2, vert3, RayOrigin, RayDir, TriIndex, PrevHit)
{
        edge1 = vert2 – vert1
        edge2 = vert3 – vert1

        Pvec = cross( RayDir, edge2 )
        Det = dot( Pvec, edge1 )

        Tvec = RayOrigin – vert1
        Qvec = cross( Tvec, edge1 )

        U = dot( Pvec, Tvec ) / Det
        V = dot( Qvec, RayDir ) / Det
        T = dot( Qvec, edge2 ) / Det

        bHit = ( U >= 0 ) and ( V >= 0 ) and ( U + V <= 1 ) and ( T > 0 )

        if bHit = true
                return ( T, U, V, TriIndex )
        else
                return PrevHit
}
```

Figure 3-2: Pseudocode of the ray-triangle intersection algorithm based on Thrane and Simonsen [2005]

3.2 CPU-Based Ray Tracing

The flexible design of CPUs is very advantageous for ray tracing. There are many rays traveling in different directions through the scene and additional rays may be spawned at any object intersection point. The CPU can handle this complexity because it allows for conditional branching, looping, recursion, and the use of complex data structures.

A ray tracer that supports reflections and refractions usually uses a recursive algorithm that traces rays over a fixed number of bounces. A bounce would be considered any time a new ray must be spawned due to an intersection with an object that has material properties that suggest that light would be reflected or refracted. The recursive portion of ray tracing pertains to the shading of rays. Figure 3-3 shows the standard recursive nature of the shade function. In order to return the color that is seen along a ray through a pixel (assuming a positive intersection with the scene), the color that is

reflected and refracted off the object must be found. In order to find those colors, however, other objects may be similarly reflected and refracted. The higher recursion level (or the number of bounces) of the primary ray, the more reflections and refractions will be calculated for each primary ray. We followed the CPU-based implementation of ray tracing described by Hill [2001].

```
Shade( Ray, RecurseLevel )
{
        HitInfo = IntersectWithScene( Ray )
        If ContainsHit( HitInfo )
                Color = calculatePhongLighting( HitInfo )
                ReflectedColor = 0
                RefractedColor = 0
                If ( RecurseLevel Not Equal 0 )
                        ReflectedColor = Shade ( ReflectRay ( Ray ), RecurseLevel – 1 )
                        RefractedColor = Shade ( RefractRay ( Ray ), RecurseLevel – 1 )

                ReflectedColor = ReflectedColor * reflectivity
                RefractedColor = RefractedColor * transparency
                Color = Color + ReflectedColor + RefractedColor

                ShadowRato = 0;
                For each Light
                {
                        If IntersectsWithScene( DirectToLight( Ray ) )
                        ShadowRatio = ShadowRatio + ( 1 / NumLights )
                }
        Else
                Color = backgroundColor;

        Return ( Color * ShadowRatio )
}
```

Figure 3-3: Pseudocode of standard recursive shade function

Shadows are also rendered as part of this algorithm. Although they do not rely on recursion, they require additional rays to be spawned for each intersection in the direction of each light to determine if another object obstructs the light from reaching the intersection point. For example, if half of the lights in the scene are obstructed, then the point is shaded with 50% of the color that it would have received. The CPU allows for easy creation and tracing of these additional rays, as well as adding or removing them from the algorithm.

In Figure 3-3, the *IntersectWithScene* function would be implemented as either the naïve approach or with an acceleration structure's traversal algorithm. Traversal algorithms that are based on a tree or hierarchical structure (in their most conventional implementation) take advantage of the CPU's flexibility with a data structure called a stack, which is currently not available on the GPU. The stack provides a place to set aside data that will need to be used in the future. When something needs to be stored, it is placed (pushed) onto the top of the stack, and when it needs to be used it is removed (popped) from the top of the stack. Accessed in this manner, the first data pushed onto the stack is the last data popped from it; likewise, the most recent data pushed onto the stack will be returned when data is popped from the stack.

3.3 GPU-Based Ray Tracing

As was mentioned in Chapter 2, the GPU is designed to process many pixels simultaneously. Typically rendering pixels only requires mathematical calculations and does not involve branching or iterating through loops. The fixed-function pipeline had no support for such instructions, but soon after the programmable pipeline was introduced, conditional statements and loops were added to the instruction set. Unfortunately these instructions are much slower than performing mathematical calculations. The GPU is also limited in the number of nested conditional statements and the number of iterations of a loop that can be performed in a single execution of a shader. Due to the memory complexity that is caused by support for recursion, recursive function calls are also not available on the GPU.

Although the instruction set on the GPU limits the number of loops that can be performed and does not allow for recursive calls, like those used to shade the rays on the CPU, ray tracing can still be performed using alternative approaches. Although reflections, refractions, and shadows are possible using the GPU (Figure 3-4 shows two screenshots from a version of our GPU-based ray tracer with shadows enabled), ray tracing without these additional rays removes the need for complicated switching between the CPU and GPU and removes the need for recursion. These effects were not used while generating results for this thesis.

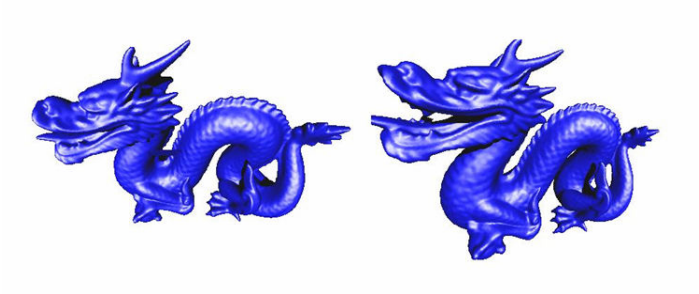

Figure 3-4: Screenshots from our GPU-based ray tracer with shadows enabled, as is most clearly evident above the back leg on the right image.

Purcell et al [2002] proposed the original organization for a GPU-based ray tracer; and did so before the programmable pipeline was even available in graphics hardware. The proposal executed fragment shaders on a screen-aligned quad so that the shader would be executed for every pixel in the viewport. This naturally leads to using one ray per pixel and the shaders could be written to handle one ray at a time.

They assume that loops would not be available in the instruction set, and provides a solution to work around this limitation. For each pixel, the fragment shader may either output a color value, or discard the pixel and output nothing. The application can perform an occlusion query to get the number of pixels for which a color value was output. The result of this occlusion query is automatically returned to the application on the CPU, which, based on the result, can choose which of the shaders to execute next. In this manner, the CPU is given the task of looping for the GPU.

There are four fragment shaders used in the proposed approach, each represented by a box in Figure 3-5; the arrows on the right side indicate the loops that the CPU controls. The first fragment shader uses the camera location and orientation to generate the primary rays that will intersect with the scene. Purcell et al [2002] assumes the use of a uniform grid as an acceleration structure, but the proposed approach is flexible enough to work with any acceleration structure. The second fragment shader traverses the ray through the acceleration structure one step at a time. The CPU loops this shader until all the rays either traverse through the structure or enter a voxel that contains triangles. The CPU then executes a shader that performs a ray-triangle intersection test for each ray. This is executed until all the rays either intersect or miss the triangles in the current voxel. If there are rays that failed the triangle intersection tests, then the traverser is executed

again until the rays enter another voxel containing geometry. This pair of shaders is continually looped by the CPU until all rays have either traversed through the acceleration structure, or have a positive intersection with a triangle. At this time, the fourth and final shader is executed which colors the pixel and, if necessary, generates reflection, refraction, and shadow rays, which can then be sent back through the traversal and intersection shaders. The proposed approach has full support for the spawned rays that generate the images achieved with ray tracing. Since the CPU performs all the looping, the reflections and refractions can be calculated as many times as desired.

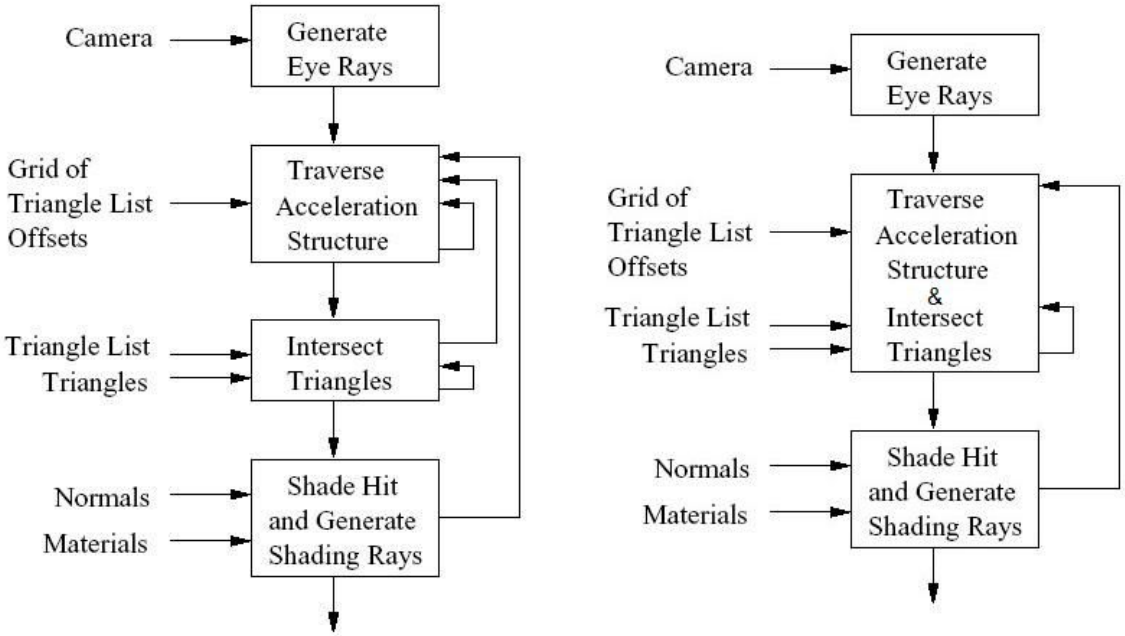

Figure 3-5: GPU-based ray tracer proposed by Purcell et al [2002] (left) and improved version by Thrane & Simonsen [2005] (right).

The approach we followed is based on Thrane and Simonsen [2005] which improved upon Purcell et al [2002] after the inclusion of loops in the shader instruction set (Figure 3-5). In their implementation, the traversal and intersection fragment shaders are combined into a single shader. The reason for the combination is that the traversal of a ray through the acceleration structure is very much tied to performing the ray-triangle intersection test on the triangles contained within the voxels of the acceleration structure.

By combining the two shaders, traversal and intersection can happen as needed until the ray has finished traversing.

The use of occlusion queries is still needed however, because the number of loops that can be performed in a shader is limited. The actual number that can be performed depends on the hardware being used and the version of the driver. In our case, a single loop could be executed 256 times, but by nesting the loop inside a loop with the same constraints, the inner loop can be executed a total of 2560 times. When this limit is reached, the shader exits the loop and continues processing the shader. In the case that a loop is unable to complete in that execution of the shader, data is written out to framebuffer objects and the CPU uses the result of an occlusion query to determine if the shader should be executed again.

The data shown on the left side of Figure 3-5 (including the voxel data, triangle lists, normals, and materials) need to be transferred from the CPU to the GPU through the use of textures. In order to put the data into texture memory, it must first be aligned in a one-dimensional array. A texture must then be created with a pixel format that uses between one (R) and four (RGBA) components. In the case of the voxel, triangle, and material data, all four components are used. Only three components are needed for loading the triangle normals. Figure 3-6 visualizes how the one-dimensional data is unpacked to fit the data into texels. Every four consecutive values in the array are mapped to the RGBA components respectively. The four components in the texel can then be retrieved at one time in the shader by sampling the texture.

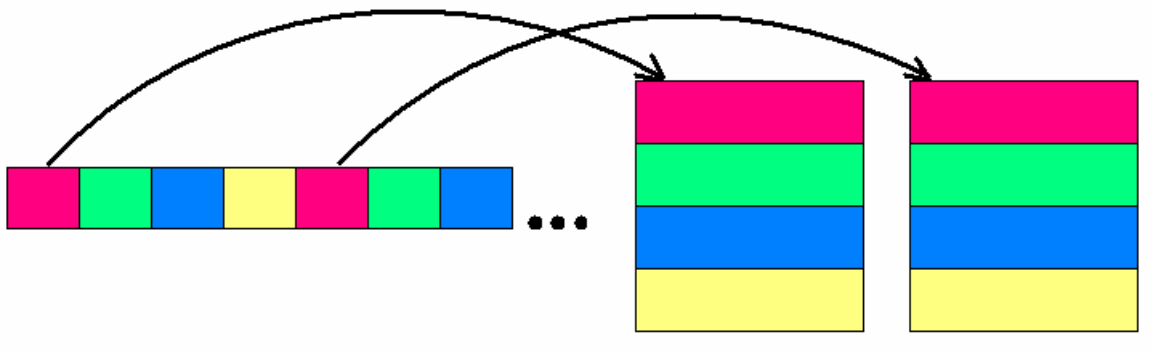

Figure 3-6: The data transformation from a float array (left) to texture memory (right)

While deciding on a texture representation to use for storing the triangle vertices, we had to consider the number of triangles that should be supported in a scene. The most common method of representing the triangle vertices uses two textures. The first stores the XYZ coordinates of the vertex into the RGB components of the texel respectively. Since the hardware and driver impose a maximum texture width and height of 4096, this implementation allows for 16,777,216 vertices to be supported by the scene. A second texture contains the triangle data, such that the RGB components of the texture each contain an index to the corresponding vertex of the triangle. Sampling of the vertex data is performed using the *Xcoord* and *Ycoord* that result from Equation 3-1. This equation must be solved for each of the three vertices. Storing the data in this manner, 16,777,216 triangles can be supported. We have devised an alternative approach, which uses three textures with corresponding texels containing the vertices pertaining to a single triangle. Organized in this manner, a single index can be used to retrieve all three vertices of a triangle (using Equation 3-1 to get the texture coordinates to sample from). Our representation also allows for 16,777,216 triangles, however for triangles that share a vertex, the vertex coordinates must be duplicated. We believe the cost of loading the additional texture is offset by the reduced number of modulus calculations that must be performed for each triangle that is accessed in the execution of the traversal/intersection shader. Additionally, there is one less texture access because a level of indirection has been removed. If all 16,777,216 triangles are accessed one time, our approach reduces complexity by 16,777,216 texture accesses, and 50,331,648 modulus, division, and multiplication calculations (based on each of the three calculations appearing once in Equation 3-1 and the equation only being performed once per triangle rather than four times).

$$Xcoord = \mathrm{mod}\, f(\frac{vertexIndex}{4096}, Ycoord) * 4096$$

Equation 3-1: Indexing and sampling into texture memory using the vertex index. This equation assumes a maximum renderable texture size of 4096 and use of integer texture coordinates.

For each of the triangles that are accessed, a ray-triangle intersection test is performed. Aside from the three texture accesses that are needed to obtain triangle vertex

information, twenty-seven additional GPU operations are needed (based on the ASM instruction count). The intersection implementation is based on Thrane and Simonsen [2005] which had improved upon the algorithm in Moller and Trumbore [1997] by taking advantage of the vector math available on the GPU. This test is uniform across all the acceleration structures, as it operates strictly on triangle vertices and is independent of the structures' traversal and geometry partitioning schemes.

4 Acceleration Structures

Acceleration structures are a means of dividing up a scene in order to simplify accessing of the geometry when it is needed for computations. These structures are used in ray tracing to reduce the number of ray-triangle intersections at the expense of additional build and traversal costs. The general concept is to group a set of spatially close triangles within a bounding region that has a simple intersection algorithm. Calculating if the ray intersects with this bounding region can quickly eliminate the need to perform intersections with each of the contained triangles. Each acceleration structure has a different method of partitioning the scene into such bounding regions. There are three categories of acceleration structure: those that partition the volume of space occupied by the scene, those that partition the geometry specifically, and hybrid approaches that partition both the geometry and the volume. Often times, the layout of the scene will be used to guide a partition of the spatial volume.

Some acceleration structures are well suited for dynamic scenes (those in which the geometry is moving over time) and can be updated quickly at runtime based on the motions. Other structures are built as a preprocessing step and are better suited for static scenes. In this thesis, the focus will be on static scenes which will not require updates to the acceleration structure. This limitation does not prevent the camera from moving about the scene, but applies strictly to the motion and deformation of the geometry.

All of the acceleration structures take advantage of a ray-bounding box intersection algorithm (Figure 4-1) for different purposes. The uniform grid and Kd-Tree use the algorithm to determine if a ray enters the region containing the scene or misses it completely; the BVH uses this algorithm throughout traversal. We have implemented a ray-bounding box intersection algorithm introduced by Woo [1990] and written for use on the GPU by Thrane and Simonsen [2005].

```
RayBoxIntersect( MinExtent, MaxExtent, RayOrigin, RayDir )
{
        tMin = ( MinExtent – RayOrigin ) / RayDir
        tMax = ( MaxExtent – RayOrigin ) / RayDir

        RayMin = min( tMin, tMax )
        RayMax = max( tMin, tMax )

        MinOfMax = min( RayMax )
        MaxOfMin = max( RayMin )

        return ( MinOfMax >= MaxOfMin )
}
```

Figure 4-1: Pseudocode of the ray-bounding box intersection algorithm based on Thrane and Simonsen [2005]

4.1 Uniform Grid

The uniform grid is a volume partitioning structure; it was originally implemented by Fujimoto et al [1986]. This structure is based on the bounding box of the objects in the scene and simply divides each axis into equal-length segments. Traversal of the structure is almost as easy as counting the segments along each axis until the position of the ray is found. The uniform grid is the most commonly used acceleration structure because the cell and ray positions can be easily calculated. Because it evenly divides the volume of the geometry, ray tracing with the uniform grid usually provides consistent frame rates and there are rarely cases of unexpected performance.

4.1.1 Construction

As the geometry is loaded into memory, the minimum and maximum extents are tracked and used to determine the bounding box of the scene. The length of the bounding box along each axis is then equally divided into a predetermined number of segments. The smaller bounding boxes formed by the length of these segments along each axis is called a cell, or voxel. Although the dimensions of the voxels need not be square, [Pharr 2004] uses square voxels to simplify the traversal algorithm, as suggested by Woo [1992]. The optimal number of segments for a scene can vary based on the layout of the

geometry. Havran et al [2000] suggested that using $\sqrt[3]{d * n} + 0.5$ segments along each axis (given n triangles and scene density d) provided a reasonable voxel resolution across a wide variety of scenes. Commonly $d = 1$, which seems logical - given a set of uniformly distributed triangles each voxel would then contain a single triangle - however, Havran et al [2000] constructed their grid using $d = 3$ and Pharr and Humphreys [2001] used $3 * \sqrt[3]{n}$ which is similar to using $d = 9$ in the previous formula. We constructed the uniform grid using the formula by Havran et al [2000] with $d = 1$ to determine the number of voxels, and did not force them to be square.

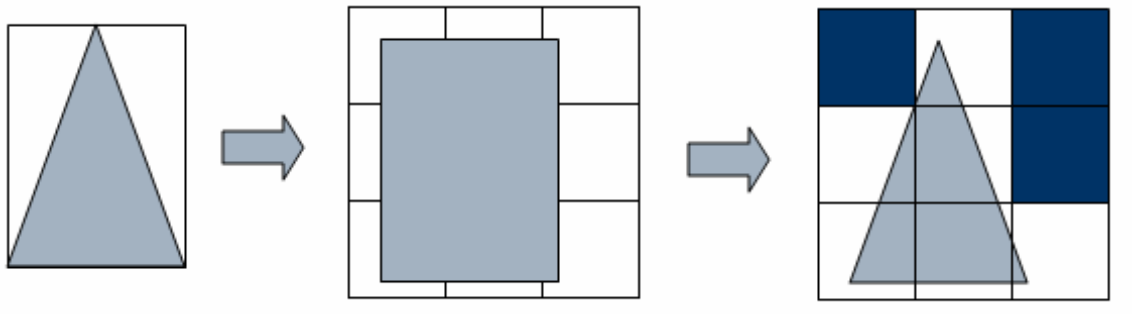

Figure 4-2: Three steps of putting geometry into a uniform grid. The right-most image represents the bounding box of the triangle. The center image represents determining which voxels are intersected by the bounding box. The left-most image represents the voxels which the bounding box intersected but the triangle does not (shaded dark blue).

Once the number of segments is determined, the geometry must be organized according to the volumetric partition; we followed a similar approach described by Bikker [2005]; visualized in Figure 4-2. The bounding box of each triangle is used to calculate which voxels the triangle could be in, taking advantage of a method proposed by Akenine-Moller [2001]. Once the list of possible voxels is determined, an actual triangle-box intersection is performed to identify the final containing voxels. The triangle is then stored in a list for each voxel. This also has the implication that a single triangle might be inserted into the structure multiple times depending on the number of voxels it intersects. For example, each of the four triangles in Figure 4-3 would be replicated for each voxel it resides in, resulting in a total of eleven triangles being stored. The impact of the repeated triangles can be reduced by adding a level of indirection. It is possible to

store each of the triangles once and have each voxel contain a list of triangle indices rather than a list of triangles. This is done on the CPU, as each list contains a pointer to the triangle instead of a copy of it.

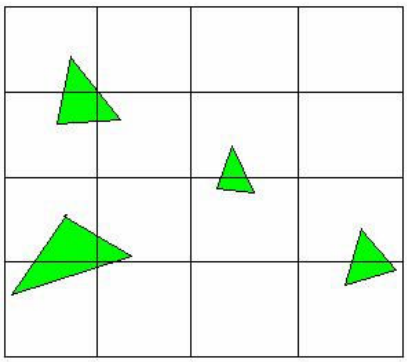

Figure 4-3: Uniform grid scene division

On the CPU, the structure is stored using a three dimensional array of lists, such that each voxel has a list of triangles intersecting with that voxel. Equation 4-1 shows how the minimum extent of the scene and the voxel dimensions are used to calculate the voxel that the ray is in at time *t*. This calculation is only performed after it has been confirmed that the ray actually intersects with the scene's bounding box.

$$floor((RayPosition(t) - MinExtent) / VoxelDimensions)$$

Equation 4-1: Calculating voxel index from ray position at time t

4.1.2 Traversal

The traversal algorithm used on the CPU is based on that presented by Amanitides and Woo [1987], which has also been previously implemented on the GPU by Karlsson and Ljungstedt [2004], Christen [2005], and Thrane and Simonsen [2005]. The first step is to intersect the ray with the bounding box of the scene to determine if the ray misses the scene completely and is immediately terminated. If the ray enters the scene, the time at which the ray enters and exits the bounding box is computed and used to calculate (using Equation 4-1) the first voxel that the ray enters. If the ray enters the

uniform grid as it does in Figure 4-4, (for a 2D example) the *tMax* values are calculated by setting the ray position to the blue dividing line (to calculate the value along the X axis) and the green line (for the Y axis). The earlier of these two times will indicate which direction the ray will traverse. After initially entering the structure (the center image), the *tMax* value along the X axis is less than the Y axis, so the ray traverses in the X direction (right image). After traversal, the *tMax* values are recalculated and once again, the ray will step along the X axis.

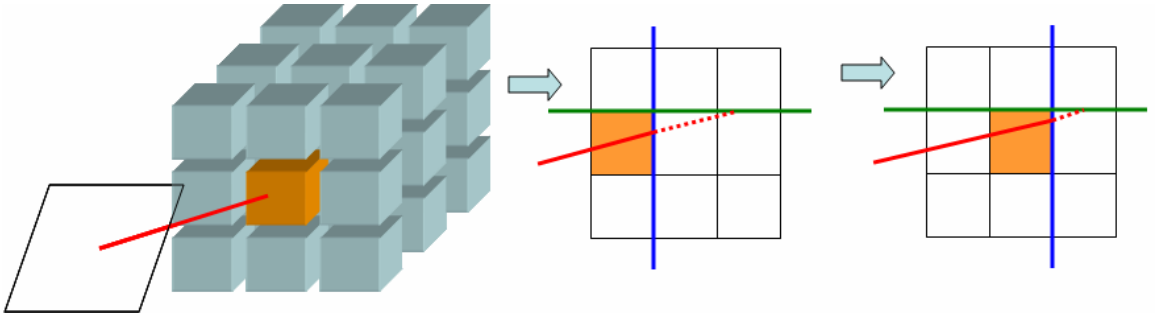

Figure 4-4: Uniform grid traversal starting with the ray entering the structure (left), intersecting with the first voxel (center), and traversing to the second voxel (right)

Special situations arise when the ray is traveling directly parallel with one of the axes, since the *tMax* value cannot be calculated under such conditions. Another special case is when the ray intersects with a corner of the voxel and could logically traverse into any of the touching voxels. In our implementation, the ray is traversed along each of the intersecting axes into the voxel that is further along the ray, avoiding the intermediate touching voxels.

4.1.3 GPU Implementation

There is little difference between the CPU and GPU implementations of the uniform grid. Although the traversal algorithm is the same, the data representation of the grid must change so that it can be accessed through textures. *Uniform* data types are used to transfer the extents of the uniform grid to the GPU, along with the number of voxels that exist along each axis, and the length of the edges of those voxels. This is all the information that is required to describe and traverse the uniform grid, with the exception of the location of the geometry. Purcell et al [2001] and Thrane & Simonsen [2005] chose

to represent the geometry data organized within a uniform grid using a total of five textures. Our approach is similar to theirs in the use of three textures to store the vertex data of each triangle, as described in Section 3.3. Thrane and Simonsen [2005] used two additional textures – one which contains a single index for each voxel of the uniform grid which is used to control access into the second texture, which contains a list of triangle indices (into the three triangle vertex textures) for each triangle that is contained in the corresponding voxel of the uniform grid. This approach avoids the need to repeat triangles that span multiple voxels. Our approach removes a level of indirection at the expense of having to store a triangle once for each voxel which it intersects. Our representation of the uniform grid stores two values per texel for each voxel of the uniform grid. These values correspond to the number of triangles contained in the voxel, and an index into the triangle vertex textures indicating the first triangle to perform the intersection test with. The grid texture is arranged such that there is one texel for each of the voxels in the uniform grid. The texture is formatted with the voxels along the X axis stored first, followed by each of the rows in the bottom layer, with each successive layer appended onto the texture. Sampling of this texture uses the *Xcoord* and *Ycoord* values from Equation 4-2 as texture coordinates.

$$Xcoord = \mathrm{mod}\, f\left(\frac{xIndex + yIndex * nSegments + z * nSegments * nSegments}{4096}, Ycoord\right) * 4096$$

Equation 4-2: Indexing and sampling into texture memory based on a three dimensional array. This equation assumes a max renderable texture size of 4096 and use of integer texture coordinates.

Traversal of the uniform grid on the GPU is performed in the same manner as on the CPU. As previously mentioned however, the limitation on the number of loops that can be performed on the GPU require some data to be stored so that the traversal shader is able to continue where it was when it reached the loop limit. Eight values need to be stored for each ray being processed (in addition to those that would normally be output containing intersection information), thus requiring two additional FBOs (Figure 4-5). The first stores the *tMax* values along each axis and a flag indicating whether the ray has finished traversing, is entering the acceleration structure for the first time (initial state), or

is either traversing or intersecting. If the ray is entering the structure, its initial voxel is calculated using Equation 4-1 and the *tMax* values are seeded. The second set of stored values is used if the ray is either traversing or intersecting. These hold the X, Y, and Z indices of the voxel containing the ray, along with a flag indicating that the ray is either traversing or intersecting (and which triangle the ray was intersecting).

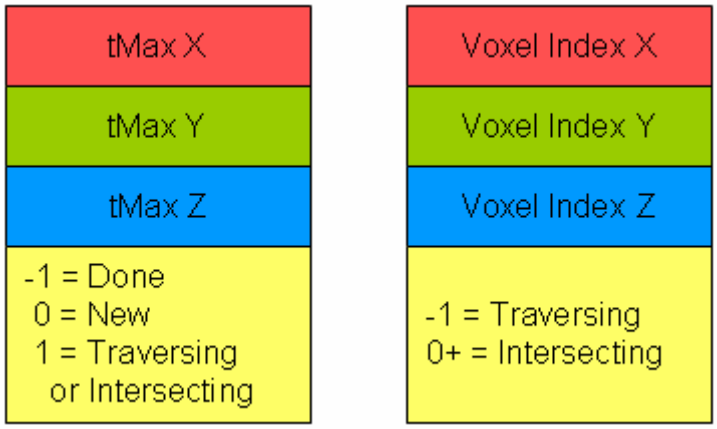

Figure 4-5: The two additional FBOs needed to store the state of the uniform grid traversal

4.2 Kd-Tree

K-dimensional trees (Kd-Trees) are another volume partitioning structure. The Kd-Tree also uses the bounding box of the scene as a basis, but then recursively divides the bounding box in half along one of the three axes. When a region is created that has few or no triangles, that region is no longer divided. In this manner, the volume occupied by the scene is divided with preference to dividing more where there is more geometry. This avoids the situation that the uniform grid may encounter where a ray is traversing through empty space. Havran et al [2000] showed that Kd-Trees are statistically among the best acceleration structures based on the number of traversals and intersections that are performed. Foley et al [2005] implemented a Kd-Tree based GPU ray tracer and reported rendering times up to eight times faster than with a uniform grid. They were using an optimized ray tracing engine that reduced memory usage, and they used an improved surface area heuristic (based on Havran and Bittner [2002]) to assist in constructing a more optimal Kd-Tree.

4.2.1 Construction

Similar to the uniform grid, the scene extents are tracked as the geometry is loaded into memory. This volume becomes the root node and is divided along one of the three axes at a point referred to as a split position. The split axis and split position are the two pieces of data that define each node of the Kd-Tree. By splitting the large volume into two smaller volumes, a single node branches into two smaller additional nodes. Each of these nodes is called an intermediate node. We used a naïve approach to building the Kd-Tree that divides each node into equal halves. The axis to split along is chosen based on the depth the tree, such that the root node (the scene's bounding box) is split along the X axis, and each successive depth rotates between the Y, Z, and X axes. As previously mentioned, a node which contains no or few triangles will not be divided. Instead of containing a split axis and split position, these leaf nodes will contain a list of triangles that are at least partially enclosed within that volume. Pharr and Humphreys [2001] suggests that using a maximum of 16 triangles per leaf node will result in an optimal tree; however preliminary studies of our test models showed better results using 10 triangles per leaf as a maximum. A tree with too many triangles per leaf node can be performance bound due to intersection computations, while a tree that is too deep can cause the traversal cost to out-weigh the reduced number of triangle intersections.

Another method of building the Kd-Tree uses a surface area heuristic to help determine where and along which axis the volume should be split. This compares the surface area of the geometry in each half of the volume and attempts to make the two halves contain equal surface area. Parameters can be set that also allow the surface area to tend towards creating empty nodes early on, so that fewer calculations are performed traversing a ray through empty space. A surface area heuristic was not used in generating our results.

One of the drawbacks to the Kd-Tree is that there is a tendency for many triangles to span across the split plane, similar to the repeated triangles that were encountered for the uniform grid. In this situation, the triangles are included in the triangle lists for both halves of that node. Since the volumes are recursively divided, a spanning triangle may be repeated several times.

The Kd-Tree is stored as a tree structure on the CPU, such that each node contains two float values (the split axis and split position) and pointers to two additional nodes. Figure 4-6 exemplifies the first few divisions in a Kd-Tree construction and how the resulting tree may appear. The original bounding box of the model is the root node of the tree and each half becomes a child node. Each of the halves is then divided along the Y axis to form the 3rd layer of the tree. Unfortunately, tree structures like this cannot be directly passed to the GPU, but rather must be passed in the form of textures, so a different representation must be used on the GPU.

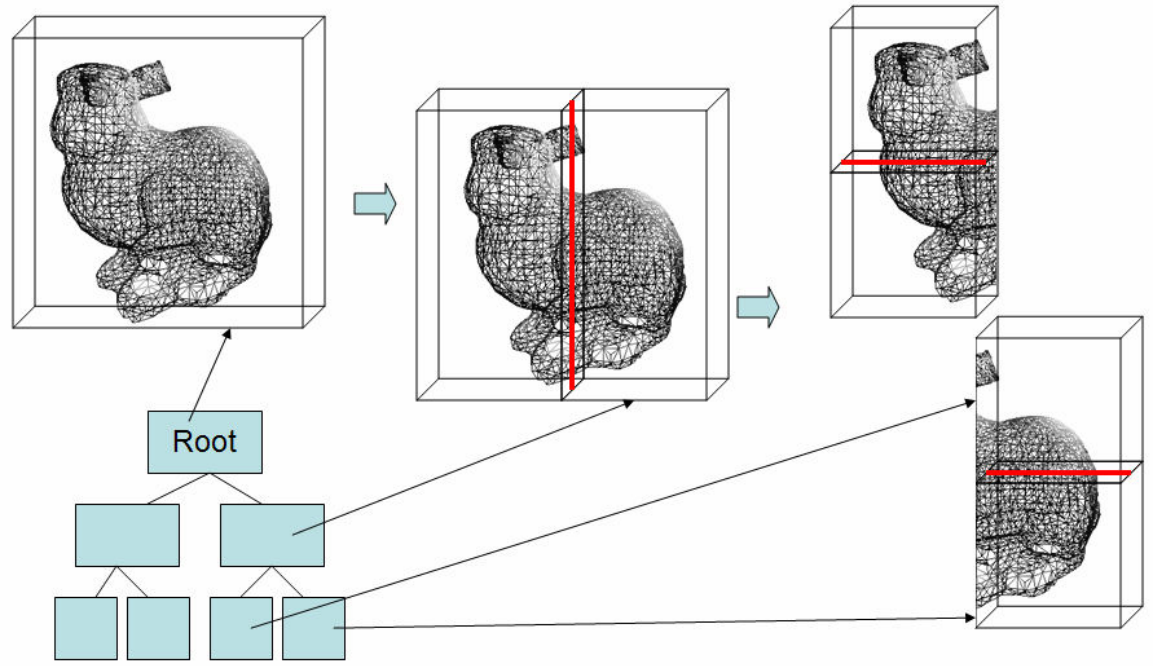

Figure 4-6: The building of a Kd-Tree

4.2.2 Traversal

The traditional traversal algorithm for a Kd-Tree is a stack-based approach. The nodes are pushed onto the stack when calculating a ray's intersection with the node, and are popped off of it to obtain the next node to intersect. All primary rays are tested against the scene's bounding box, and rays that do not intersect it are immediately terminated, all other rays will perform a ray-node intersection with the root node. It is

important to note that only nodes that are intersected by a ray will be traversed. Nodes that will not be intersected are never pushed onto the stack, and thus are never traversed.

Figure 4-7 shows a diagram of the three types of ray-node intersections that result in a traversal of at least one side of the node. The side(s) that will be traversed are based on three values that are obtained during the ray-node intersection test. The first value is represented as blue arrows in the figure. This value corresponds to the distance, or time, that the ray traveled before entering the node, and is often referred to as *tMin*. The next value shown in orange, referred to as *tSplit*, is the time at which the ray intersects with the split plane of the node. The final value, *tMax*, is the time at which the ray exits the node and is shown in green. Comparison of these three values determines which side of the node to traverse. If *tSplit* is the smallest of the three values (A), only the farthest node is traversed. If *tSplit* is the largest of the three values (C), then the closest node is traversed. Finally, if *tSplit* is between *tMin* and *tMax* (B), then both nodes are traversed. In this case, usually the closest node is traversed first, and the farthest node is pushed onto the stack. If a ray-triangle intersection results in a positive hit in the closest node, then traversal of the far node can be skipped and the traversal for that ray is finished.

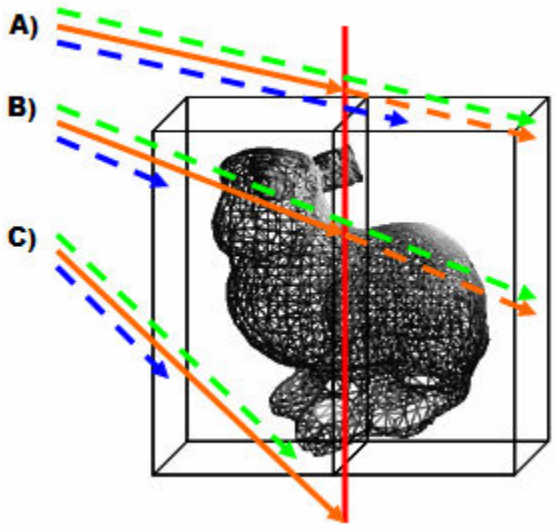

Figure 4-7: Three potential traversals of the Kd-Tree. A) The ray only travels through the right node. B) The ray travels through both nodes. C) The ray only travels through the left node.

Since each node is composed of two smaller nodes and leaf nodes are only constructed when there are less than 10 triangles in the node, several traversals must be performed in order to identify the leaf node that the ray first intersects. If all triangles in the leaf node fail their intersection test, then a node is popped off the stack and node traversal continues. If the stack is empty at the time that the node should be popped off, then the ray has traversed through the Kd-Tree and terminates as not having intersected a triangle.

4.2.3 GPU Implementation

The traversal algorithm on the GPU is significantly different from that on the CPU. The first limitation is that although data structures are available on the GPU, there is no means to directly transfer a complex structure on the CPU to a similar structure on the GPU. Small amounts of data can be loaded as *uniform* data types or large amounts of data can be manipulated on the CPU into arrays of float values, which can then be transferred to the GPU as a texture. This use of texture memory must be used to represent the nodes of the Kd-Tree on the GPU.

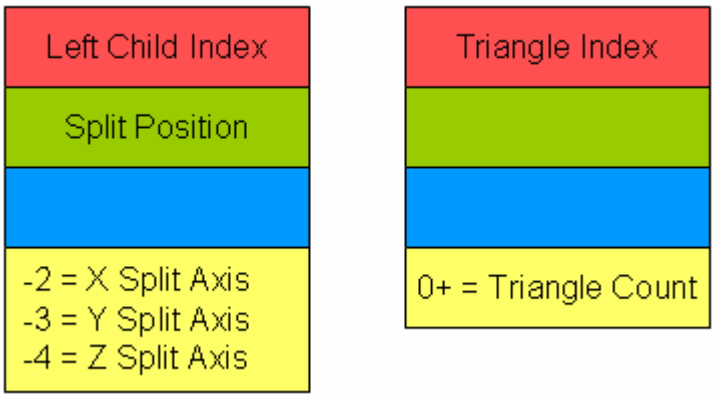

Figure 4-8: Representations of intermediate (left) and leaf nodes (right) of the Kd-Tree on the GPU

The two types of nodes that get generated during Kd-Tree construction get stored differently in the texture, as is shown in Figure 4-8. Since both types of nodes are stored in the same texture, there must be a means of distinguishing between texels that represent intermediate nodes, and those that represent leaf nodes. This is done using the alpha channel (yellow). A positive number indicates that the texel represents a leaf node

(shown to the right), and the value of the number indicates the number of triangles that exist in that node. The index of these triangles into the triangle vertex textures is indicated by the red channel; the green and blue channels remain unused (so the value in the channel does not matter). A negative number in the alpha channel indicates that the texel represents an intermediate node (shown to the left). The value of the number indicates which axis the node is divided along, with -2 being the X axis, -3 being the Y axis, and -4 being the Z axis; the split position along this axis is stored in the green channel. The indices to the nodes on either side of the split position are encoded both in the texel components and in the representation of the tree within the texture. The index to the left child is stored in the red channel and the texel representing the right child is always stored immediately following the texel for the left child. Using this encoding, the index of the right child is always one more than the index to the left child. In our implementation, the blue channel is not used.

In addition to the data representation being different on the GPU, the primary mechanism for tree traversal on the CPU, the stack, does not exist on the GPU. Foley and Sugerman [2005] present two alternative methods of traversal of a Kd-Tree. The first method is referred to as *Kd-restart* and the second is *Kd-backtrack*. The *Kd-restart* approach uses the same node intersection algorithm as the CPU, but instead of popping a node off the stack, the ray position is incremented in distance (or time), and the traversal restarts from the root node again. Since the ray is incremented, the traversal algorithm will follow a path into the further node than the ray was previously in, and the traversal will continue as necessary. The *Kd-backtrack* approach also offers an alternative to popping off the stack. Instead of restarting the ray from the beginning as in the *Kd-restart*, this algorithm updates the ray's position until it is out of the current node (and thus inside a further node) and stores additional data which allows it to walk back up the tree until a node is found that contains the updated ray. If such a node is not found, then the ray has traversed out of the tree; if it is found, then traversal down the tree continues as normal. The additional data that needs to be stored for the *Kd-backtrack* algorithm would have required a revised construction algorithm (to track indexes to parent nodes) and additional textures to be loaded (to transfer the data to the GPU). For these reasons, the *Kd-restart* algorithm was implemented for testing.

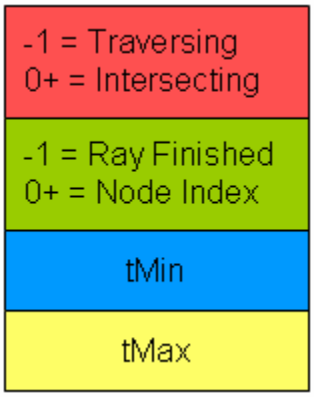

Figure 4-9: The additional FBO needed to store the state of the Kd-Tree traversal

Similar to the uniform grid, the limitation on the number of loops in the GPU affected the rendering of scenes using the Kd-Tree. In order to execute the shader again to allow the rays to complete traversal, an FBO was used to store the state information. Figure 4-9 shows the texel representation of this information. The first value is used as a flag to indicate if the ray is iterating over triangles (a positive number) or traversing the tree (-1). If *Kd-backtrack* had been used, a value (-2) can be used to indicate that the ray is walking up the tree. The second value indicates if the ray is finished (-1) or is an index into the texture (a positive number) representing the node that contained the ray. The final two values contain the *tMin* and *tMax* values of the ray.

4.3 Bounding Volume Hierarchy

The bounding volume hierarchy (BVH) is one of the few structures that partitions the geometry rather than the volume. A bounding region, typically an axis-aligned bounding box, is formed around pairs of spatially close triangle bounding boxes and then pairs of bounding regions are enclosed in a larger region, as shown in Figure 4-10. By dividing the geometry, the BVH ensures that there will be more traversal around highly detailed areas of the scene, and that empty regions will be traversed quickly. The Kd-Tree and BVH both separate the scene in similar manners, but since the BVH is based on dividing the geometry, the internal structure, and thus the traversal, is different. A big advantage of the BVH over the Kd-Tree is that there are no repeated triangles, thus a ray will never intersect with the same triangle more than once. On the other hand, the

organization of the scene, in particular the amount of occlusion that exists between triangles, has an impact on the speed at which the BVH can be traversed. However, in a comparison of acceleration structures, Thrane and Simonsen [2005] showed that the use of a BVH resulted in faster ray tracing on the GPU than the other structures.

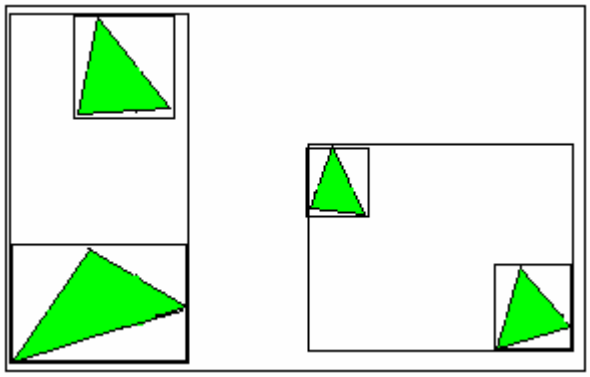

Figure 4-10: The BVH scene division

4.3.1 Construction

There are two different approaches that can be taken when building a bounding volume hierarchy. The first is a bottom-up approach, which inserts one triangle at a time into the tree [Goldsmith and Salmon 1987]. It uses a surface area heuristic and the current state of the tree to determine the best location (node) for the given triangle to be placed; as such, the order in which triangles are inserted into the tree can have an affect on the quality of the tree. The other approach is a top-down approach [Kay and Kajiya 1986], where the volume of the scene is divided and, in the process of dividing, bounding volumes are formed around the partitioned geometry. The latter of these approaches was used for constructing the BVH.

The division of the scene follows a format similar to the Kd-Tree division however triangles that span the split position are only inserted into one of the subdivisions. A visualization of this division is shown in Figure 4-11. The vertical split on the left divides a single triangle and a group of triangles. Division of a volume terminates if one or two triangles are contained within it, so the volume containing the single triangle is complete, but the group of triangles must still be divided. This next

division is chosen in a round-robin fashion (similar to the Kd-Tree naïve division). The horizontal split is performed at the mean of the bounding box extents. Given this split position, our implementation uses the relative location of the minimum vertex of the triangle to determine which side to place it in. This may result in all the triangles all being on one side of the split position, in which case the split-axis is incremented and the division repeated at most two more times in the round-robin fashion. If no division results in a successful split of the triangles, then they are divided evenly among the two halves.

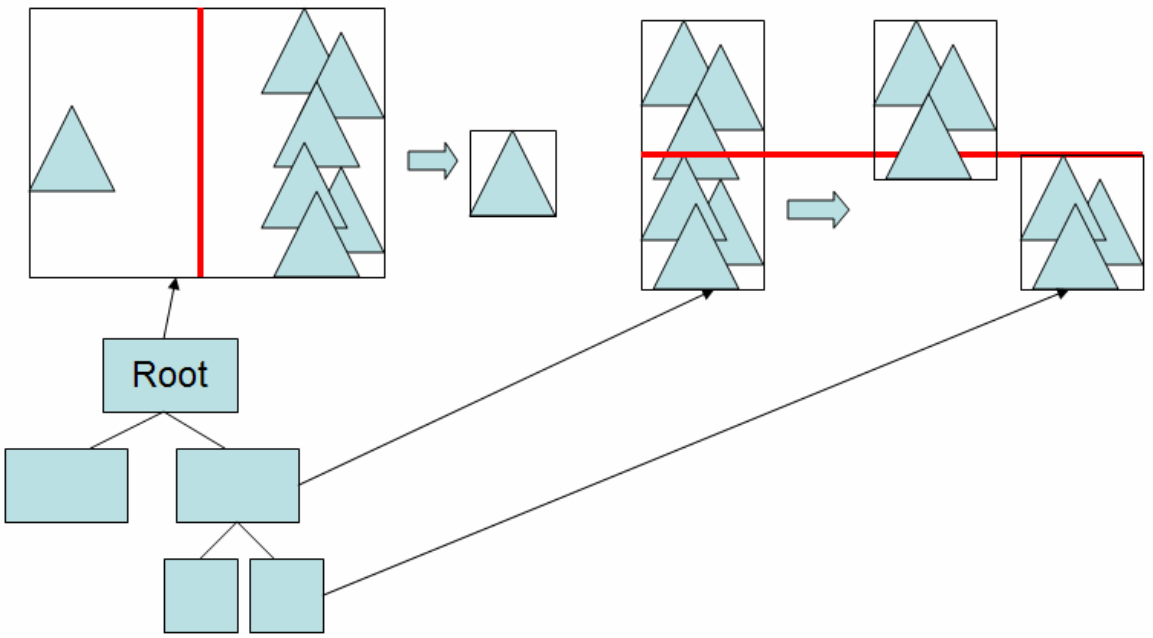

Figure 4-11: The construction of a BVH

If a triangle spans the selected split position, as it does in the figure, the triangle is inserted into only one of the two volumes. This has both positive and negative implications. On the positive side, this results in a binary tree with no repeated triangles. This fact allows us to pre-compute the size of the tree (given N triangles, the tree will have $2 * N - 1$ nodes) and perform memory optimization by pre-allocating all the necessary memory for the tree. It would also be possible to identify a worst-case scenario for ray tracing the scene. This leads into the negative aspect of not repeating the triangle. Due to the configuration of the geometry, bounding volumes may overlap where the

triangle spanned the split position. This means that rays that intersect the overlapping volumes must traverse through both bounding volumes.

The data storage needed for an intermediate node of the BVH is only 6 float values and two references to child nodes. The float values correspond to the coordinates of the min and max points of the axis-aligned-bounding-box. Intermediate nodes are constructed until only one or two triangles are in the volume, at which time a leaf node containing references to the triangles is constructed.

4.3.2 Traversal

Traversing a bounding volume hierarchy on the CPU is a stack-based approach; in this manner, it is very similar to the Kd-Tree traversal algorithm. When a ray intersects with both child nodes of a node, one child is pushed onto the stack and the other is traversed. Traversal of nodes is performed in a recursive manner until the ray reaches a leaf node, at which time ray-triangle intersections are performed and then the top node is popped off the stack, and traversal continues. The ray is done traversing when there is nothing more on the stack to pop.

The main difference between the Kd-Tree traversal and the BVH lies in the ray-node intersection test. The main mechanism for the BVH ray-node intersection is a fast ray-box intersection test provided by Woo [1990]. This test also provides the time that the ray enters and exits the bounding-box. If traversal of one node results in a ray-triangle intersection that is closer than the node that is popped off the stack, the ray can stop traversing early. Figure 4-12 shows a visual of this situation. The orange ray intersects with both bounding boxes. Since the *tHit* of the left bounding-box (blue line) is earlier than the *tHit* of the right box (green line), the node corresponding to the left bounding-box is traversed. The maroon ray indicates how far the ray has traveled until it intersects with a triangle in the left bounding-box and the black ray indicates how far the ray travels before intersecting with the second node. Since the *tIntersect* with the triangle is closer than the *tHit* represented by the green line, the ray cannot find a closer triangle intersection in the right node, and thus the ray can terminate traversal.

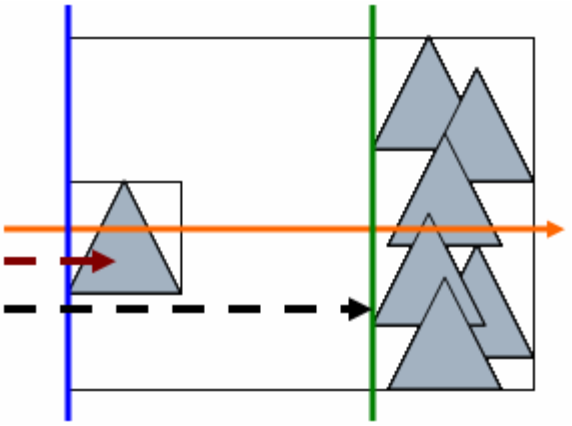

Figure 4-12: Traversal of a BVH showing that the *tHit* of the child nodes can be used to determine which node to traverse first

4.3.3 GPU Implementation

Implementing the BVH on the GPU requires overcoming the same two hurdles as was presented with the Kd-Tree. A texture-friendly format for the node data and a means of traversing the structure without a stack must be identified. Thane and Simonsen [2005] present a solution to both of these problems.

They suggest that the intermediate node data naturally leads to a straightforward texture format, such that the minimum extents of the bounding box are stored in one texel and the maximum extents in another. The leaf nodes do not contain bounding-box data, but instead are just triangles. Instead of using one or two texels to store the index of the triangle vertices, Thrane and Simonsen [2005] suggest actually storing the triangle vertex data in place of the node data. The basis for this suggestion is that as an index to the node data is being incremented, a flag in the texel data can be used to indicate that the data contained at that node is actually a triangle. This method of compact encoding removes the need for extra textures to store the triangle data and the additional texture accesses to gather the data. Two texture accesses need to be performed to get intermediate node data, after one of the accesses, a flag is checked to indicate whether the data represents a triangle or a node and if the data is a triangle, then one additional texture access is performed to get the third vertex of the triangle.

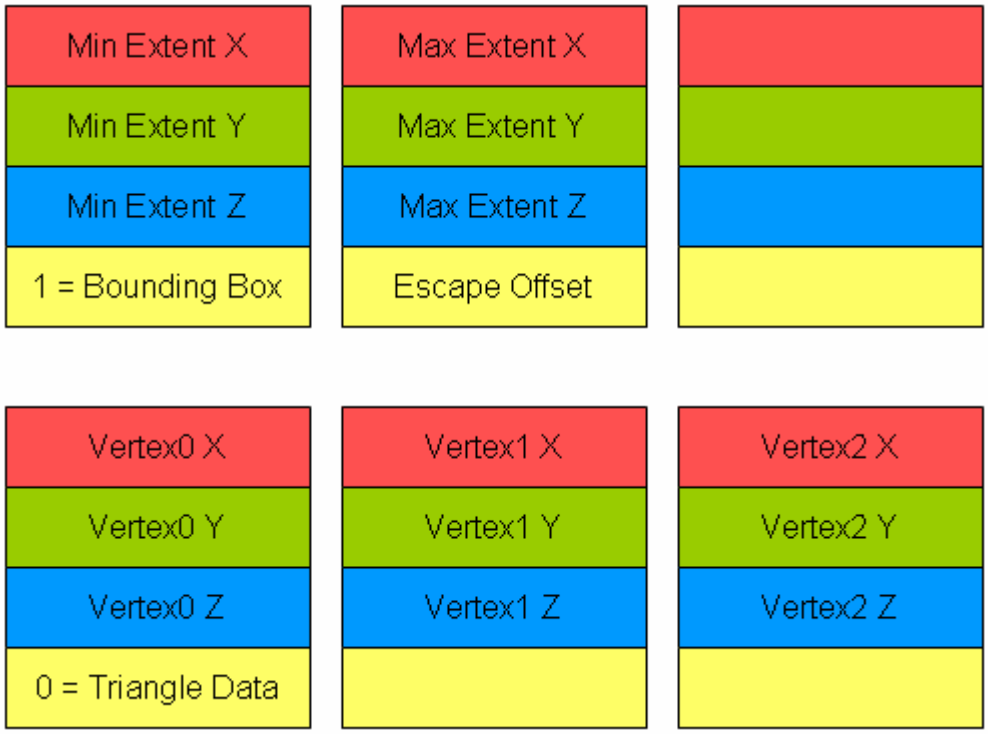

Figure 4-13: Texel components used to store intermediate nodes (top) and leaf nodes (bottom) of a BVH

Our implementation of the GPU-based ray tracer uses three textures to encode the vertex data and so the algorithm was modified slightly to account for this. Figure 4-13 displays the encoding of the intermediate nodes and leaf nodes of the BVH. For intermediate nodes, the RGB components of *Texture1* hold the XYZ coordinates of the minimum bounding box extent; the A component is a flag indicating that the data pertains to a triangle (0) or a bounding-box (1). *Texture2* contains the maximum extent and an escape index offset that indicates where to go if the ray does not intersect with the bounding box. In the case of intermediate nodes, *Texture3* is unused. Leaf nodes may contain one or two triangles; each triangle is stored in the texture memory such that *Texture1* has one vertex, *Texture2* has a second vertex, and *Texture3* has the third vertex.

The solution to the second hurdle, of traversing the structure without a stack, is solved in the encoding of the data into the texture. Thrane and Simonsen [2005] observe that all rays traverse the tree in a depth-first, right-to-left order. Since the rays already have a fixed-order traversal, this order can be encoded in the storage of the structure in texture memory. The only issue is identifying which node the ray must jump to if it fails a

ray-box intersection test. The secret here lies in the fact that this is a binary tree and given N triangles, a node will have $2 * N - 1$ children. The escape index offset shown in Figure 4-13 is simply this value based on the number of triangles in that node. The offset is added to the current index to get the index of the next node to traverse. The ray finishes traversing when the index is equal to the maximum number of nodes. Figure 4-14 (based on a figure from Thrane and Simonsen [2005]) shows a diagram of how this approach traverses the acceleration structure. Dotted lines represent the path taken if the ray fails the ray-box intersection test and the escape offset is used.

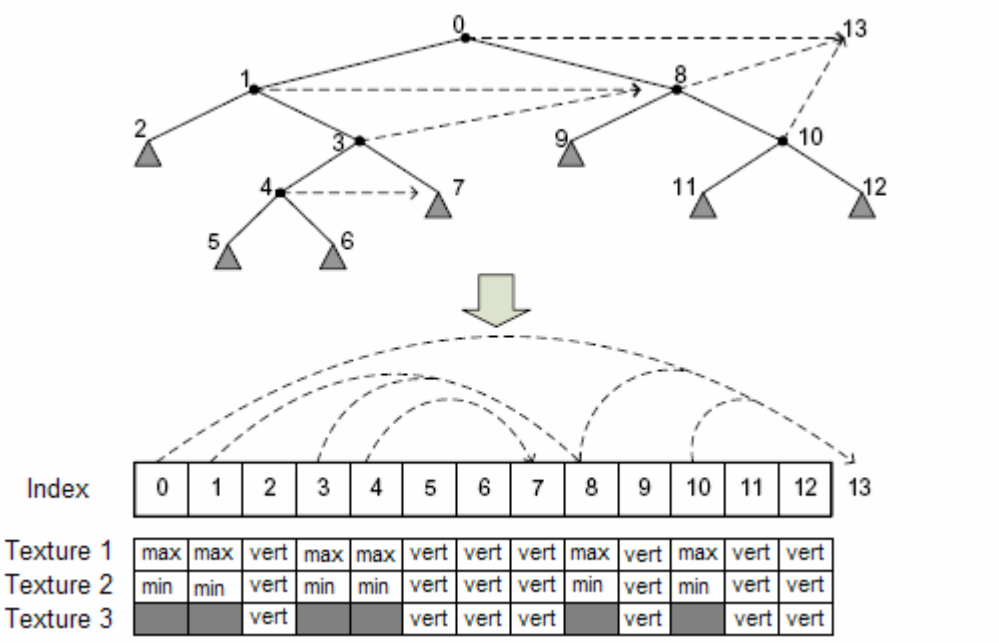

Figure 4-14: Short example of the texture organization and traversal order of the GPU-BVH (Based on a figure from [Thrane and Simonsen 2005])

Our original choice to use three textures to store the triangle vertices shows a drawback here, in that unused texture memory is consumed. As can be seen in Figure 4-14, the third texture is packed with unused texels (gray) for each of the intermediate nodes to ensure that the vertex data of the triangles remains aligned. The benefit of this approach, however, is that scenes with more triangles can be encoded through the use of three textures (allowing 8,388,608 triangles) rather than just one (only 3,355,443 triangles).

As with the other acceleration structures, the limitation regarding the number of loops that can be performed in a single execution of a shader requires that additional data be stored so that traversal can continue on the next execution of the shader. In the case of the bounding volume hierarchy, only one value needs to be stored – the index into the texture that the ray should continue traversing at (Figure 4-15). Since the node and triangle information are stored in the same texture, there is no need to further distinguish between traversing nodes and intersecting triangles, and unlike the other acceleration structures, the ray is not marching through a volume, so there are no *tMin* and *tMax* time segments that need to be stored. The current implementation stores the one necessary index in the R component of a four-component (RGBA) texture, but this could be optimized to hold just one value per texel.

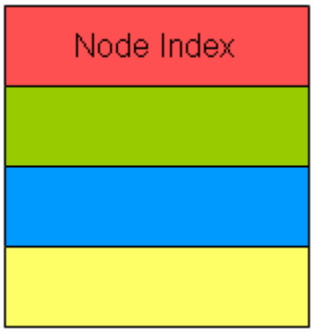

Figure 4-15: The additional FBO needed to store the state of the BVH traversal

5 Experimental setup

The drain rate of the battery can be largely impacted by the system configuration on which the measurements are being taken. The results presented in this thesis were gathered on a Dell Inspiron E1705 laptop with a duo core Intel T2300 processor operating at 1.66 Ghz. The machine has 512 MB of system memory and an NVidia GeForce GO 7800 PCI Express x16 graphics card with 256 MB of video memory running ForceWare version 83.60. The battery powering the system is a Dell Rechargeable Li-ion Battery; type D5318. It has a capacity of 53W and rating of 11.1V. The video card was configured with vertical sync on at a 60Hz refresh rate and a power setting of 'maximum power savings'. The graphics card in the laptop was designed for mobile devices, so several 'tricks' that hardware manufacturers use to reduce energy consumption are done on this system. For example, the card is designed to run on lower watt signals than desktop graphics cards. These same types of modifications will be performed on graphics chips for cell phones and PDAs. Thus, the results achieved should be representative of those that would be obtained by executing the code on other mobile devices.

Other factors that can impact battery drain rate involve the system devices that are enabled while measuring energy consumption and the level of activity that those devices are performing. As these tests target CPU and GPU based ray tracing, there should be little network activity and disk access. The level of activity on the CPU and GPU will be affected by the acceleration structures used and the geometry that is being ray traced. This thesis aims to identify an energy-efficient acceleration structure, so the choice of scenes is important.

5.1 Scenes

Our ray tracing engine only used primary rays while ray tracing the scenes. The use of only primary rays is called *ray casting*, as rays are only cast into the scene and are not traced along bounces throughout the scene. This was done intentionally to allow the results to be applied more generally. When only sending one ray through each pixel into the scene, the number of rays is easily calculated and the per-ray (or per-pixel) costs

become more applicable to anti-aliased ray tracing, or to a worst-case estimation if any secondary rays are used. However, if secondary rays had been included in the tests, geometry layout would have had a much greater impact on the number of rays in the scene and the material properties would have to be taken into consideration in determining the number of rays that were cast into the scene. The addition of secondary rays in the tests would have made the results much more difficult to analyze.

One of the difficult aspects of identifying an optimal acceleration structure for ray tracing is that the organization of the geometry in the scene can have a big impact on the traversal order of an acceleration structure. There have been attempts at creating a benchmark set of scenes that stress the weak points of different acceleration structures. The most complete set of benchmark scenes comes from the Benchmark for Animated Ray Tracing (BART) [Lext et al 2001]. Our ray tracing engine was not optimized for the areas that BART stresses, including cache performance, large datasets, and bounding volume overlaps. Also, BART uses hierarchical animation to position various parts of the geometry, but our engine does not have support for such a hierarchy. Although none of the BART scenes are used in these tests, the models and scenes chosen for testing were intended to provide various levels of complexity. If a trend exists such that one acceleration structure is better than another, it should be evident based on the selection of scenes used for these tests.

Models and scenes (collectively referred to as scenes) with a range of triangle counts were chosen for ray tracing and it is arguable that a better approach would have been to use multiple levels of detail of each of the scenes. In accordance with the aims of this thesis, an efficient acceleration structure should be consistently efficient across a collection of scenes regardless of the level of detail or configuration of the geometry. Occasionally we differentiate between models and scenes within the remainder of the thesis based on the following distinction: models are a single object (which may contain any number of triangles), and scenes contain multiple models.

After traversing through any of the structures, a ray will eventually either leave the acceleration structure, or will intersect with a triangle. The first scene (Figure 5-1) is referred to throughout this thesis as the Single Triangle scene, as it contains only one

triangle. This extremely simple scene was intended to compare the acceleration structures behavior at the most basic level.

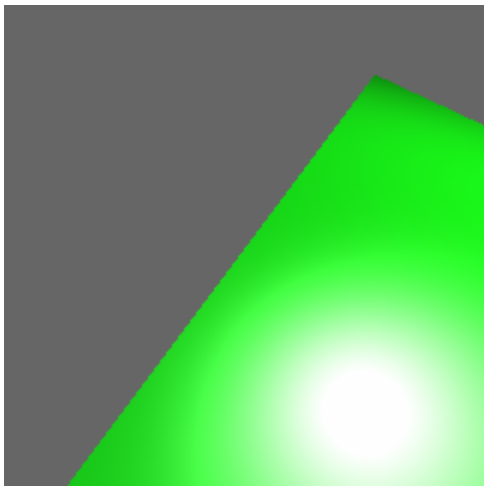

Figure 5-1: Single Triangle Model (1 triangle)

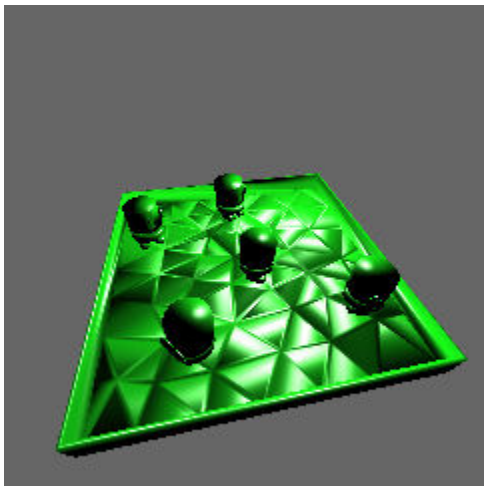

Figure 5-2: Toy Scene (11,141 triangles)

The Toy Scene (Figure 5-2) contains 11,141 triangles and is representative of a simple, typical scene that may be ray traced. There is a relatively low polygon floor with five more-detailed objects (wind-up toys) scattered about. This scene was originally introduced at SIGGRAPH 2006 by Wald et al [2006]. The complexity in this scene is related to the empty space that exists above the floor and between the toys. For this reason it is expected that the uniform grid may result in less efficient ray tracing of this

scene than other acceleration structures. There is also a significant amount of overlap in this scene as the toys occlude the floor behind them and there is a solid bottom on the underneath side of the floor. This may cause the BVH to have some difficulty, as it requires that rays fully traverse through the scene before identifying the closest intersection to the ray origin.

The Small Dragon model (Figure 5-3) is a popular model used in ray tracing due to its intricate shape and complex geometry. The Small Dragon is available at several levels of detail through The Stanford 3D Scanning Repository [Stanford 3D]; the one rendered here is the third most complex, with 47,794 triangles. The configuration of the dragon's body creates many empty regions within the dragon's bounding volume; for this reason, this model may be more difficult to ray trace with the uniform grid than with other structures.

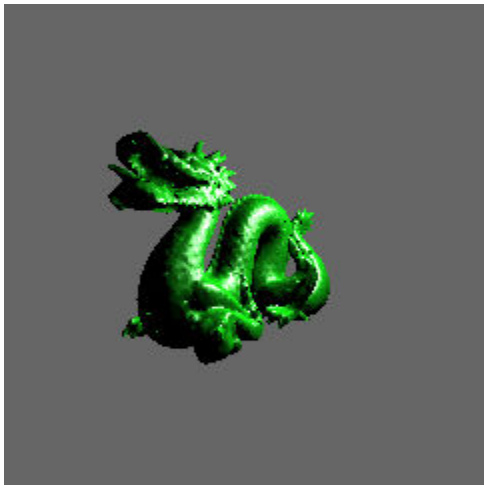

Figure 5-3: Small Dragon Model (47,794 triangles)

Another popular model that was included for testing is the Stanford Bunny [Stanford 3D], shown in Figure 5-4. The Stanford Bunny is also available through the Stanford Repository in multiple levels of detail; here the highly tessellated version with 69,451 triangles is used. The Stanford Bunny has a fairly even distribution of triangles throughout, with slightly more-detailed areas around the eyes and ears. There is not much overlap within this model, nor are there very many empty regions. Each acceleration structure should be able to ray trace this model without difficulty.

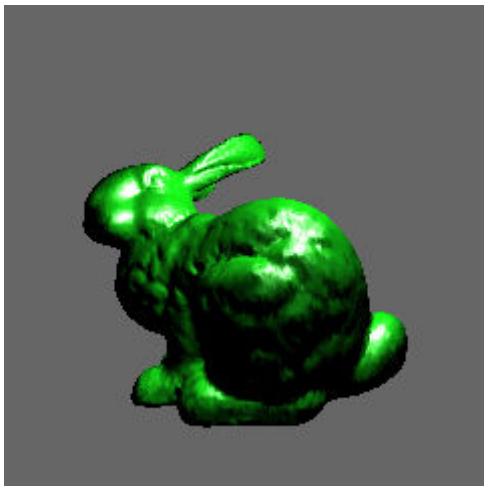

Figure 5-4: Stanford Bunny Model (69,451 triangles)

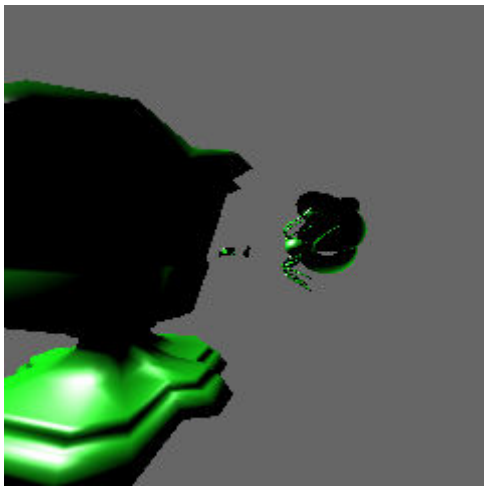

Figure 5-5: Complex Scene (98,867 triangles)

The fifth and final scene (Figure 5-5) included for testing is referred to as the Complex Scene throughout this thesis. This scene was intended to create a very difficult "teapot-in-a-stadium" situation with highly detailed models towards the center of the scene and low detail models towards the outskirts. Located in the center of the scene is a 69,451-triangle Stanford Bunny that is $1/15^{th}$ the volume of that shown in Figure 5-4; its silhouette is barely visible in the center of the figure. The slightly larger object to the left of the Stanford Bunny is a 16,646-triangle antique car which has very complicated wheels and a less complex body. The left-most object in the scene is a 604-triangle urn.

On the opposite side of the scene from the urn is a 2,880-triangle torus knot. These two objects have the lowest polygon count of the five models composing the scene. The last model is a 9,286-triangle spider. All together there are 98,967 triangles in the Complex Scene. The number of triangles and the inconsistent triangle density throughout this scene are likely to make it difficult for all the acceleration structures.

5.2 Measuring Energy Consumption

There are many factors that can affect the energy consumption of a laptop including, but not limited to, the monitor, network cards, and hard disk specifications. Barr and Asanovic [2003] made a strong case for the negative affects that network transmission can have on energy consumption. Pilot tests of energy consumption with the wireless LAN card enabled resulted in drastically inconsistent battery drain rates; so disabling the wireless network card was identified as an important experimental decision. The wired network on the test machine is automatically disabled when running on the battery, so that also would not interfere with our data gathering. In regards to disk access, some access was required by our ray tracing engine to load the models from file, however the energy consumption measurements begin immediately before the scene is rendered. The disk accesses for model loading must happen before this time. In attempts to minimize other programs access to the disk, tests were run with only our ray tracing engine running in the foreground. There may have been background processes or OS tasks that start or stop during the testing period which would have affected our results, but based on the measured results this is not likely to be the case. Figure 5-6 compares the discharge rate between rendering a scene with OpenGL and ray tracing the same scene on the GPU. The left-most and right-most data points are samples taken before and after the testing to provide insight into the discharge rate of the test machine in an idle state. The figure makes it evident that both rendering (using OpenGL) and ray tracing (using the GPU) increased the drain rate by nearly 50% of idle rate. In the presence of additional processes running on the machine, the discharge rate would have fluctuated greatly during sampling. Although there are minor fluctuations in the GPU ray tracing measurements in the figure, the consistency of the OpenGL rendering suggests that if any background tasks were being executed on the test machine during testing, they were

minimal and did not interfere with gathering results. The minor shifts in the GPU ray tracing are likely due to changes in the number of traversals that rays had to take as the camera rotated around the scene.

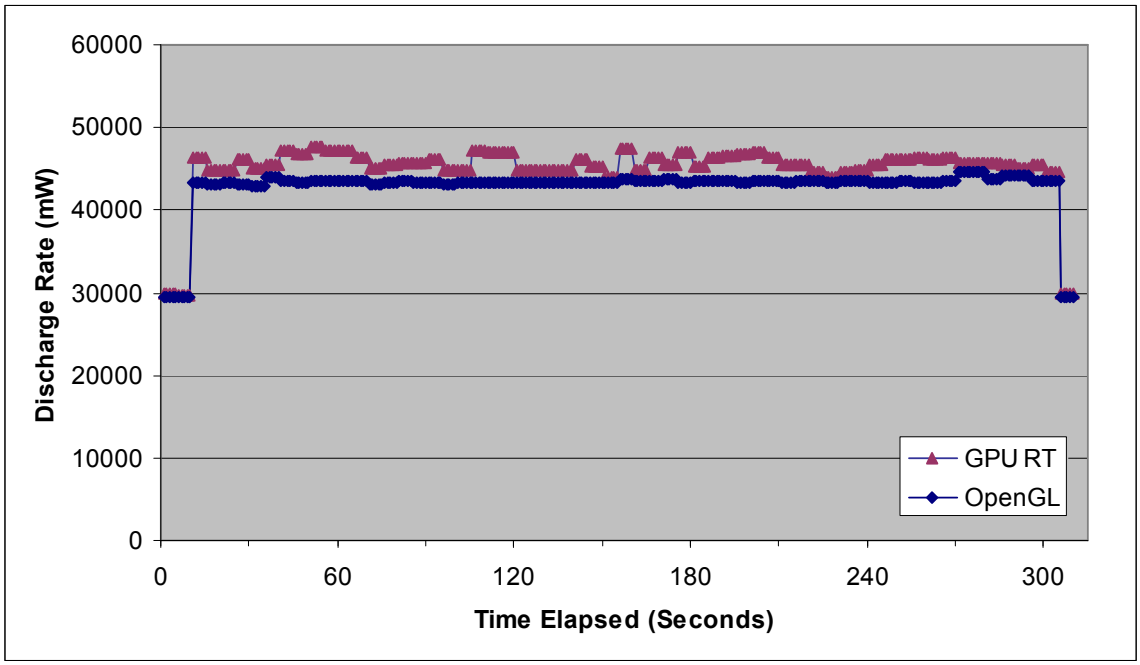

Figure 5-6: Comparing battery discharge rate of OpenGL rendering to GPU-based ray tracing

If we had tested larger scenes, there could be a concern for additional energy consumption in an occasional memory swap writing data to disk. Such costs would be dependent on the amount of memory on the system and the amount of data that had to be swapped. If this had occurred however, it would likely be a cost associated with the attempts at ray tracing a large dataset and the cost of the memory swap should be included in the measurements.

According to Intel, powering the pixels on a mobile device can account for 50% of the battery consumption [Intel 2002]. In attempts to minimize the impact of this, the tests were executed with monitor brightness at its lowest setting. Not only did this enable us to confirm consistency across the tests, but it extended the battery life enough to allow a greater number of tests to complete on a single charge of the battery.

Most electronics, the test machine included, have batteries which can quickly charge a majority of capacity (90% for example) in a short time and then may take the

same amount of time to charge to full capacity (the final 10%). A pilot study showed that a similar behavior does not exist on the discharge of the battery, and that the level to which the battery was charged at the start of the test did not affect the results obtained. A test run at 90% capacity had similar results to the same test run at 30% capacity.

The graphics research community has not focused on optimizing rendering algorithms for power consumption. Barr and Asanovic [2003] worked towards optimizing the energy consumption of data compression, motivated by the energy consumed during wireless data transfer. They recognized that significant savings could be made by using better encryption to minimize the number of bytes sent over a wireless connection. Tscheblockov [2004] used shunts and a multimeter to measure and compare the power consumption of various graphics cards from different manufacturers. Their tests were performed strictly to compare the consumption of graphics cards in 'idle' and 'burn' states. PowerScope is a hardware tool developed by Flinn and Satyanaranan [1999] which enables easy power measurements to be performed on mobile devices. A similar software-based solution called PowerSpy was proposed by Banerjee and Agu [2005] as means to measure how software applications are using various hardware devices and the corresponding affects on power usage.

These solutions are all limited in various ways that prevented them from being useful in our research. We chose to use the Advanced Configuration and Power Interface (ACPI) that was developed between Microsoft and several other hardware and software manufacturers which exposes, among other information, battery discharge rates in mW. The frequency at which ACPI updates energy usage information varies depending on the devices and operating system installed. Microsoft Windows exposes the information through a function called *callNtPowerInformation* and updates the battery statistics every five seconds, although the information can be sampled more frequently than that. Preliminary tests showed that sampling more frequently than twice per second had a negative impact on performance. It was decided to sample the battery information once per second. The ACPI samples are stored until after the testing is complete at which point the information is processed and written to file. This was done to avoid additional processing and affecting the battery drain rate while testing.

Given a scene and a screen resolution, a test can be executed using one of the three acceleration structures on either the CPU or GPU or the scene can be rendered using OpenGL. This results in seven structures that can be tested. The test period lasted five minutes, during which time the camera would rotate 1 degree around the object after each frame was rendered. Figure 5-7 shows the total energy lost after rendering for the five minute duration with each of the acceleration structures. The acceleration structure is only built once prior to the start of testing, so the energy consumed during construction is not included in the results. Each test was only performed once, but the five minute duration should have allowed a sufficient number of frames to render that the average energy-cost-per-frame would not be biased in the presence of minor noise in the sampling.

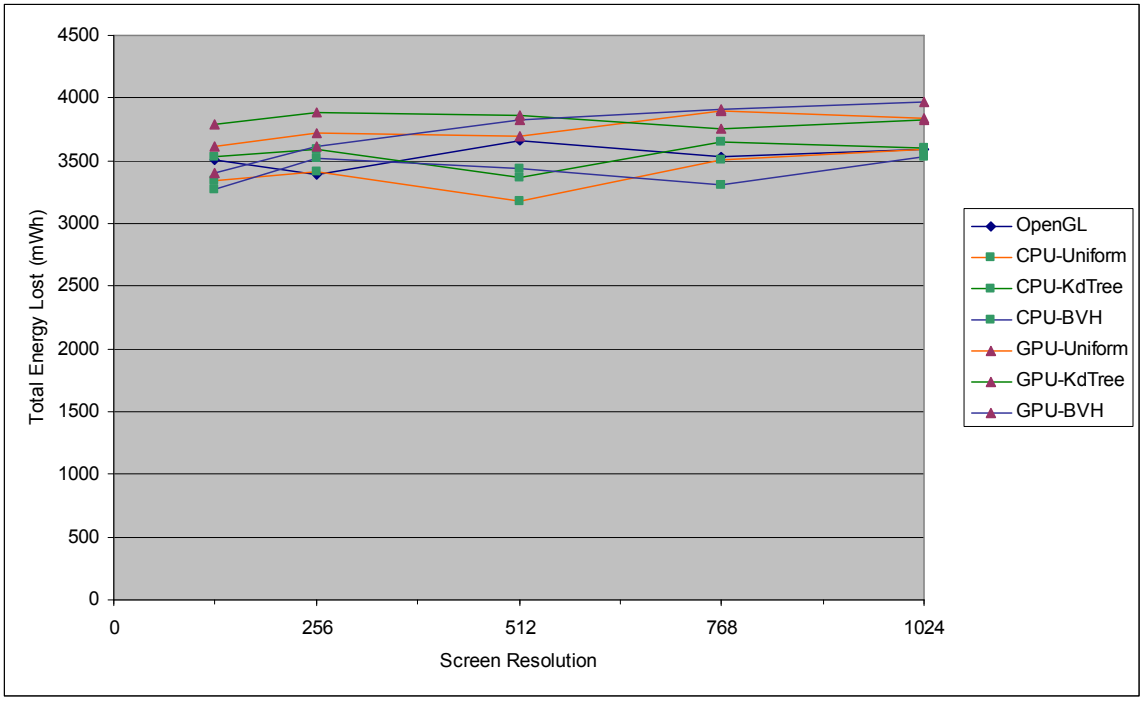

Figure 5-7: Comparing total energy lost by each of the seven acceleration structures

The OpenGL rendering is represented throughout this thesis by a blue dot and blue line. The CPU-based acceleration structures are represented by a green square and the GPU-based structures are represented by a purple triangle. This is done to assist in comparing the CPU to GPU-based implementations. Within each of the processing units, an orange line is used to represent the uniform grid, a green line is used for the Kd-Tree,

and a blue line is used for the bounding volume hierarchy. In some figures a pink line is also used to represent a naïve ray tracing implementation that does not use any acceleration structure, which is labeled as 'BasicAS'.

The data represented in Figure 5-7 is incomplete, however, as it does not provide any indication as to the number of frames rendered during the five minute period. We aimed to identify an acceleration structure that was efficient in terms of energy cost per frame, so the number of frames rendered is important. Figure 5-8 presents the total number of frames rendered on a logarithmic graph. Although the total energy cost may have been relatively consistent across all the structures, the number of frames rendered is quite different for each of the acceleration structures. For this reason, the results discussed in Chapter 6 are presented from the standpoint of energy per frame.

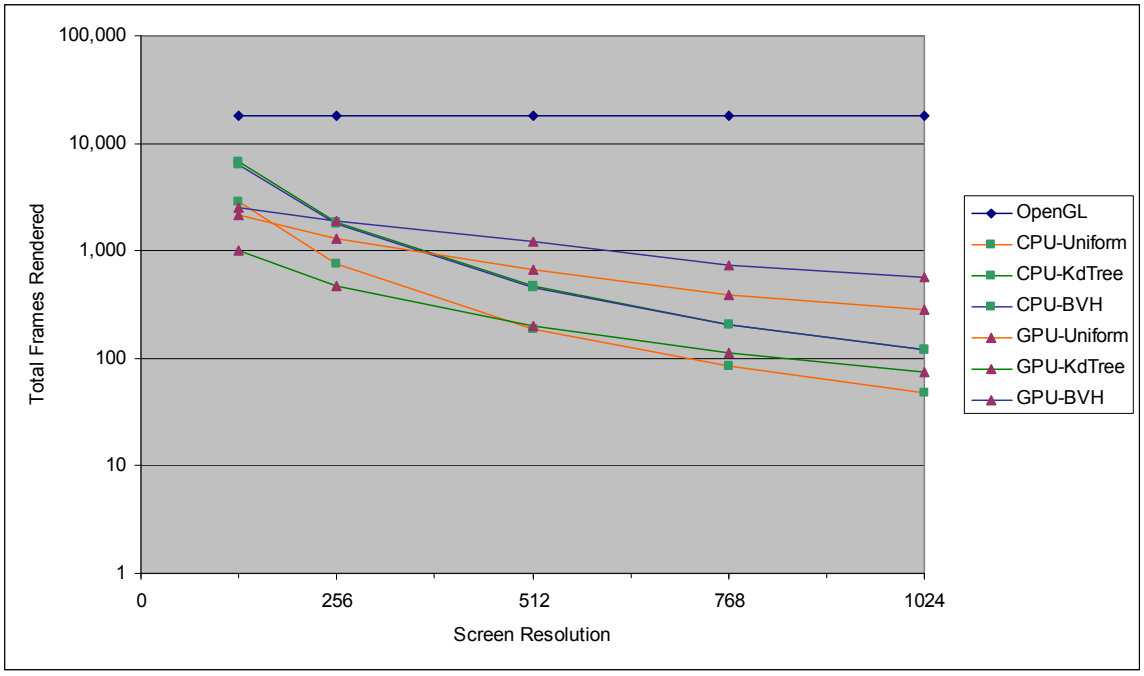

Figure 5-8: Comparing total number of frames rendered by each structure (logarithmic scale)

6 Results and Applicability

The results obtained from our tests can be analyzed from several viewpoints. First and foremost, the objective is to determine which acceleration structure and processing unit results in the least amount of energy loss while rendering a scene. This may be dependent on the number of triangles in the scene, the layout of the scene, or the viewport resolution. The tests included in this thesis targeted triangle count and viewport resolution as primary variables, although a byproduct of changing the number of triangles in the scene is that scene layout is also affected. We present the results looking first at the energy and time costs of the different structures while ray tracing a given scene and how they vary as the viewport resolution increases from 128x128 to 1024x1024. We then look at a given resolution and compare how the energy and time costs change with increasing triangle counts. These results can then be used to estimate the energy and time costs per frame that can be achieved on mobile devices.

6.1 Results

These results show how different acceleration structures compare between the two processing units and how they change as the viewport resolution increases. Results generally show that given a model the energy and time costs are influenced most by processing unit, then by viewport resolution, and finally by the acceleration structure chosen.

Since the model rendered is consistent across all screen resolutions, data is presented in the form of line graphs. Where specified, the results are normalized against the CPU implementation of the uniform grid because the combination is considered the simplest ray tracing implementation that makes use of an acceleration structure.

6.1.1 Ray Tracing a Single Triangle

As a scene is divided through the construction of an acceleration structure, there comes a point when there is only a single triangle left to enclose. Similarly, as a ray is traversing an acceleration structure, it is likely to intersect with a triangle. This test model was intended to show the efficiency of each structure in the trivial situation of having to

accelerate the ray tracing of a single triangle. In the testing of this scene we include the use of a naïve ray tracing implementation, where every ray is intersected with the triangle. Depending on the acceleration structure, some will perform only the ray-triangle intersection test, while others will attempt to cull the ray based on a bounding box intersection prior to intersecting with the triangle. The data shows (Figure 6-1) that on the CPU, the uniform grid and the Kd-Tree both consume more energy and time than the other CPU ray tracing schemes due to this extra intersection test. Unfortunately, these results don't reflect how the acceleration structures will perform every time a ray must intersect with a triangle; the reason for this is that some structures, such as the Kd-Tree, will create a leaf node that contains several triangles. In this case, there might be an overhead cost to intersect with the bounding box, but that cost would be amortized over all the triangles in the leaf node.

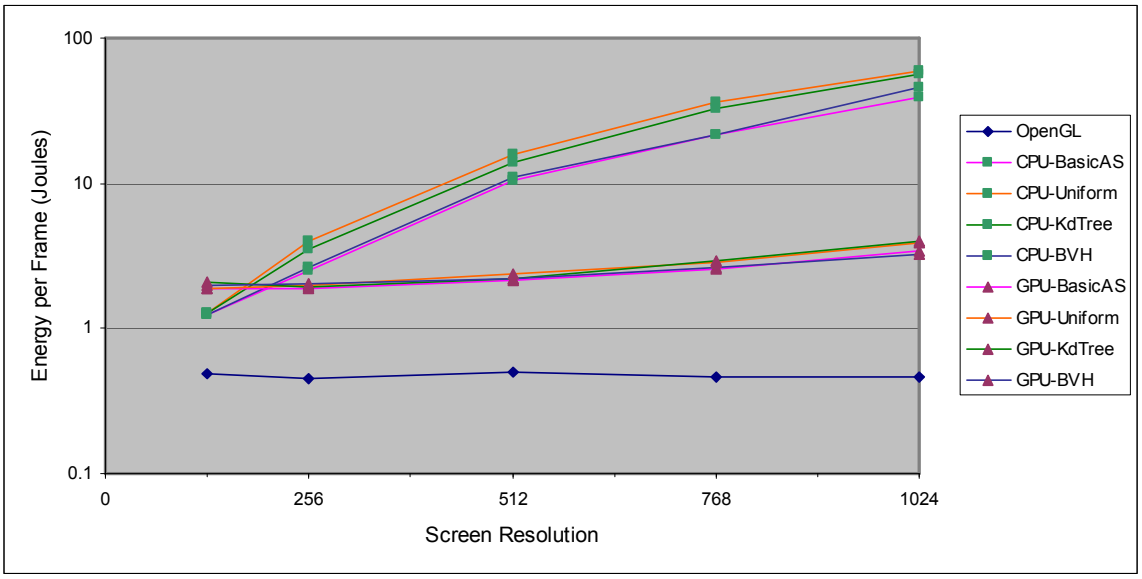

Figure 6-1: Energy cost per frame of ray tracing the Single Triangle

Strictly looking at energy cost per frame, there is a very clear separation between the CPU and GPU. For such a simple model, the choice of processing unit is more important in terms of energy efficiency than the choice of acceleration structure. At a resolution of 128x128 the CPU consumed less energy than the GPU, but at the next tested resolution of 256x256 the GPU becomes the more-efficient processing unit.

Comparatively, the GPU continues to get better as the screen size increases. At the highest resolution of 1024x1024, the acceleration structures on the CPU consumed roughly 10 times the amount of energy as their GPU counterpart. On a per-frame basis, the energy cost of ray tracing on the CPU at a resolution of 256x256 is comparable to that of the GPU at a resolution of 1024x1024. Although the number of rendered pixels has a 16:1 ratio, the efficiency of the GPU allows the energy cost to be similar and renders more frames in the same amount of time.

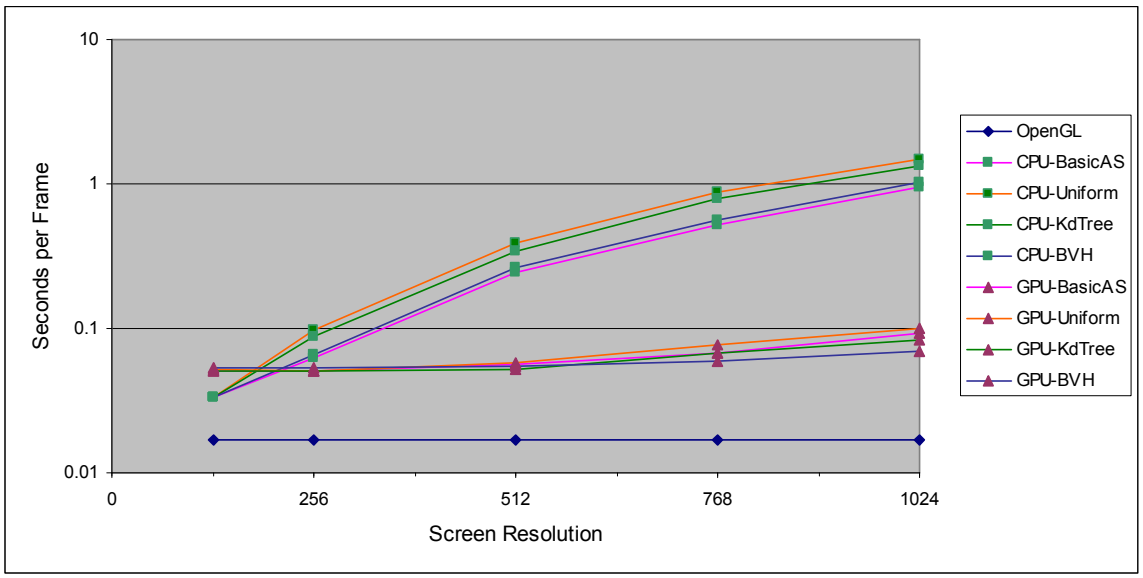

Figure 6-2: Time cost per frame of ray tracing the Single Triangle

Comparing seconds per frame (Figure 6-2) to the energy cost shows that *the time per frame and energy per frame are directly related*. To get a better look at that relationship, the ratio of energy per second is presented in Figure 6-3. Although energy per second could be a measure of energy efficiency, it should not be used in terms of computer graphics because it does not take into consideration the number of frames rendered. This measure is strictly being used to compare the relationship between energy per frame and seconds per frame. The OpenGL rendering costs ~28 W, while the ray tracers range between 36 and 47 W. This separation of 8-19 Watts suggests that ray tracing uses more energy than not ray tracing. This finding is not of particular interest as the objective is to find the most efficient acceleration structure and processing unit pair.

At resolutions of 512x512 and 1024x1024, it is easiest to see that for this scene, the BasicAS (the naïve approach to ray tracing) is generally the most efficient structure. Since this scene has only one triangle which covers a majority of the screen, this result is not surprising, as most acceleration structures will add the ray-box intersection test in attempts to cull the ray. If the triangle was small, these results probably would have changed slightly and the BasicAS may not have been the most efficient.

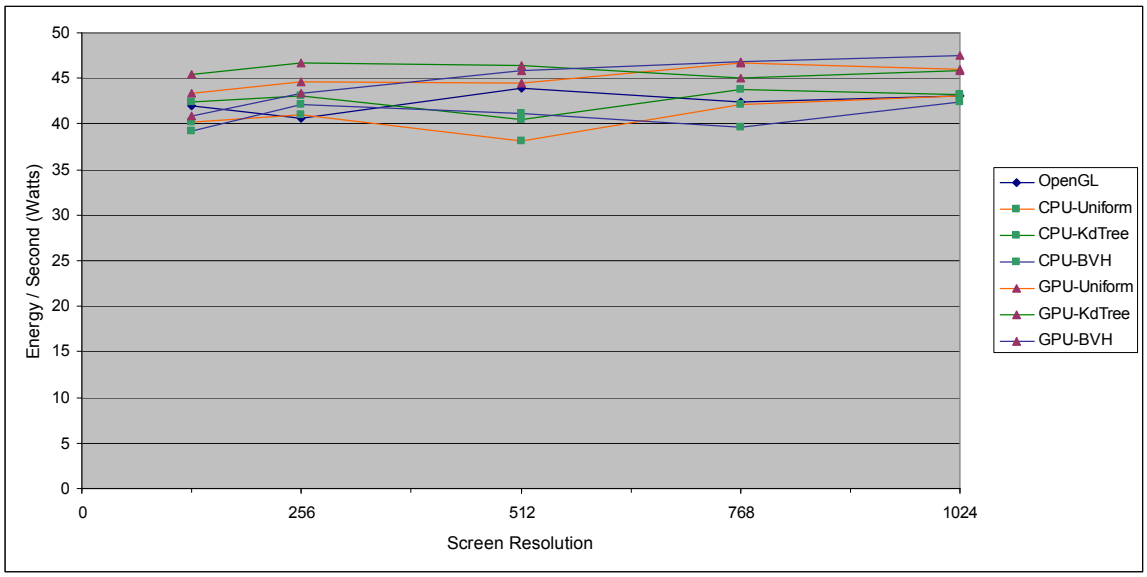

Figure 6-3: Power expended (in Watts) while ray tracing the Single Triangle

Normalized by the amount of energy consumed per frame against the CPU-uniform grid, we get better insight into the relative efficiency of the structures on the CPU (Figure 6-4). The simple case where every frame attempts to intersect with the triangle actually ends up being the most efficient of the CPU-based implementations, consuming only 60% of the energy as the uniform grid in the best case. The bounding volume hierarchy also reaches the same 60% efficiency, whereas the Kd-Tree is at best 90%. The reason for this separation between the structures is due to a bounding box intersection test that must be performed for the Kd-Tree and uniform grid to determine if the ray is entering the region inhabited by the triangle. In the case of the BasicAS and BVH, the only intersection test performed is with the single triangle. Similar efficiencies in rendering times on the CPU are also evident. At the lowest screen resolution, the GPU

ray tracer consumes almost 1.6 times the amount of energy and time per frame. However, at the highest resolution of 1024x1024, the GPU is more than 10 times more efficient.

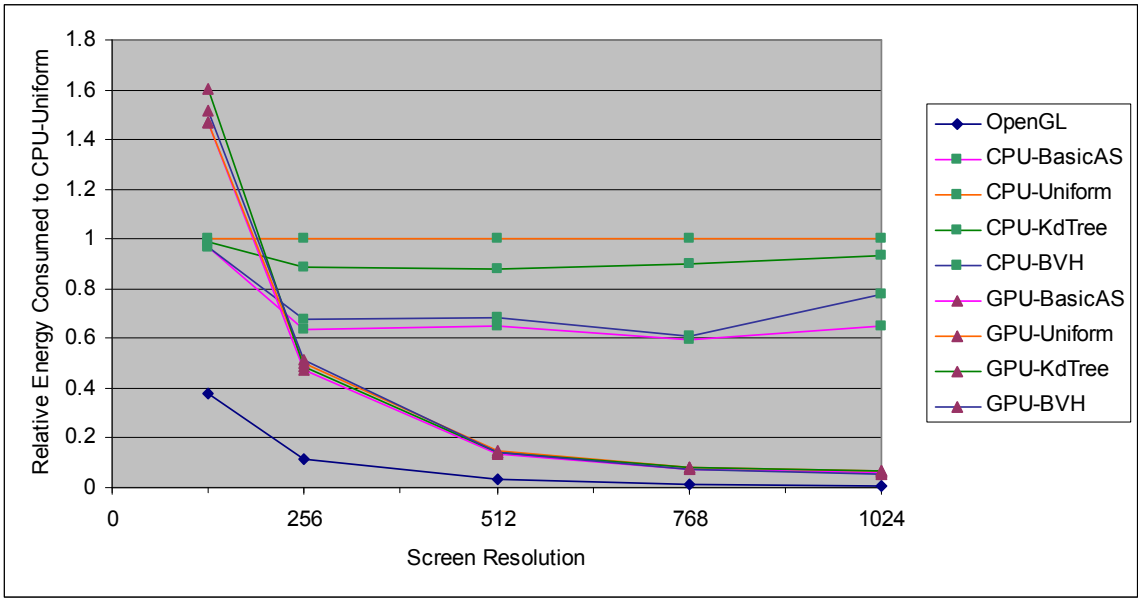

Figure 6-4: Relative energy consumed while ray tracing the Single Triangle compared to the CPU-uniform grid

6.1.2 Ray Tracing the Toy Scene (11k Triangles)

With over 11,000 triangles, this scene would not finish ray tracing using the BasicAS during the five minute test duration. It became clear that scenes of this size and greater would require the use of an acceleration structure in order to be ray traced at an interactive frame rate.

Strictly looking at the energy consumed per second of rendering, the GPU is less efficient at ray tracing than the CPU (Figure 6-5) at resolutions higher than 256x256. The CPU ranges from 37 to 43 W, while the GPU ranges from 40 to 47 W. Although the CPU may consume less energy per second of ray tracing, this does not account for the number of frames that are rendered.

Figure 6-6 shows the energy cost per frame of ray tracing. The Kd-Tree implementation on the GPU has also begun to stray slightly from the other acceleration structures, but maintains a similar trend as the other GPU implementations. The K-Tree

aside, there still exists a crossover where the GPU becomes more energy efficient than the CPU, however it is significantly less dramatic than before. For the resolutions where the CPU is the more efficient processing unit the Kd-Tree is the more efficient structure. At higher resolutions the GPU is more efficient, with the uniform grid being just slightly better than the bounding volume hierarchy.

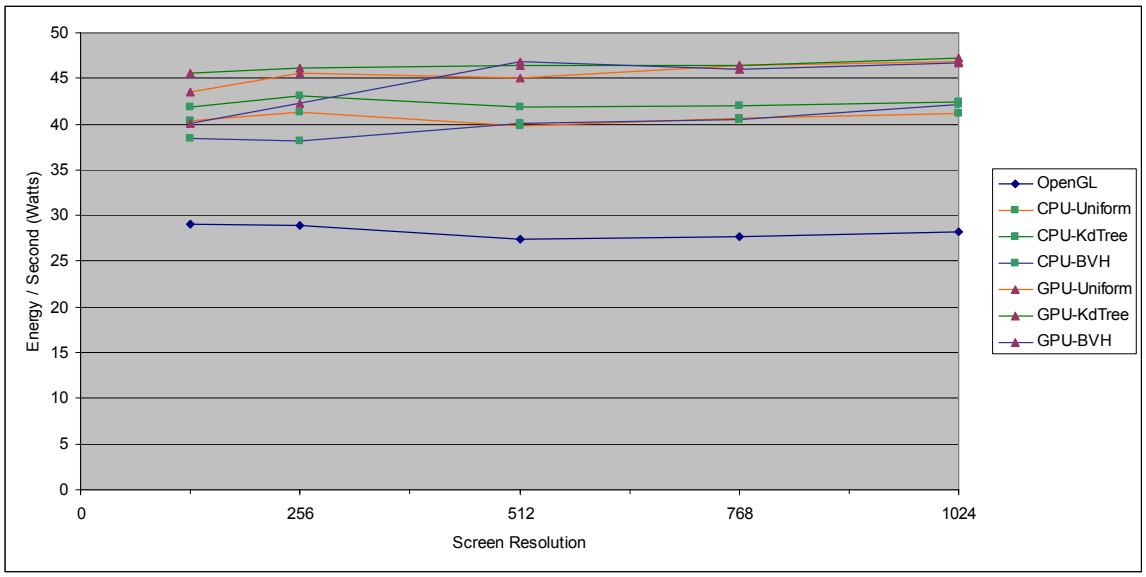

Figure 6-5: Power expended (in Watts) while ray tracing the Toy Scene

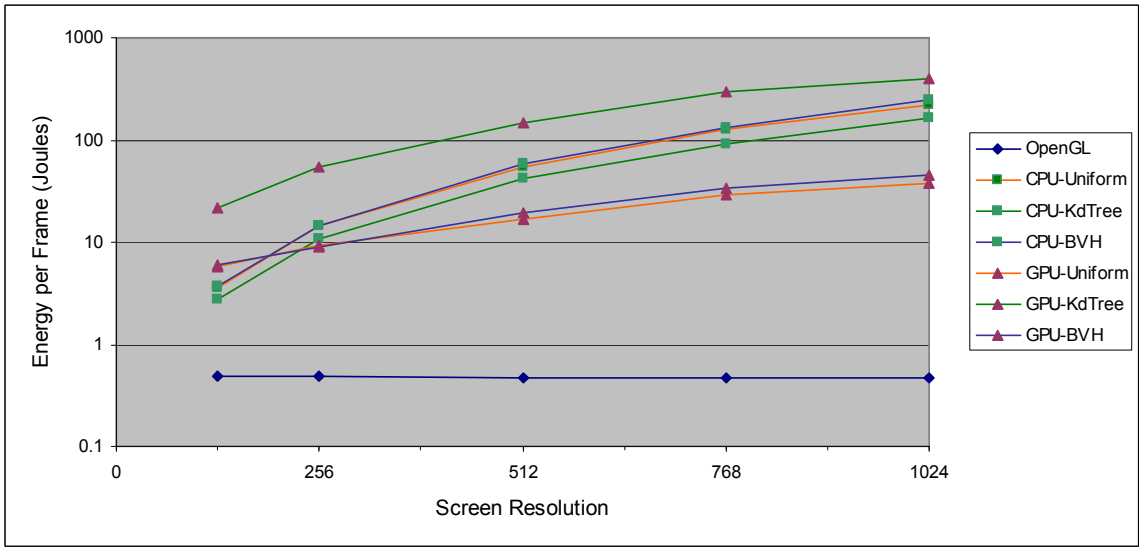

Figure 6-6: Energy cost per frame of ray tracing the Toy Scene

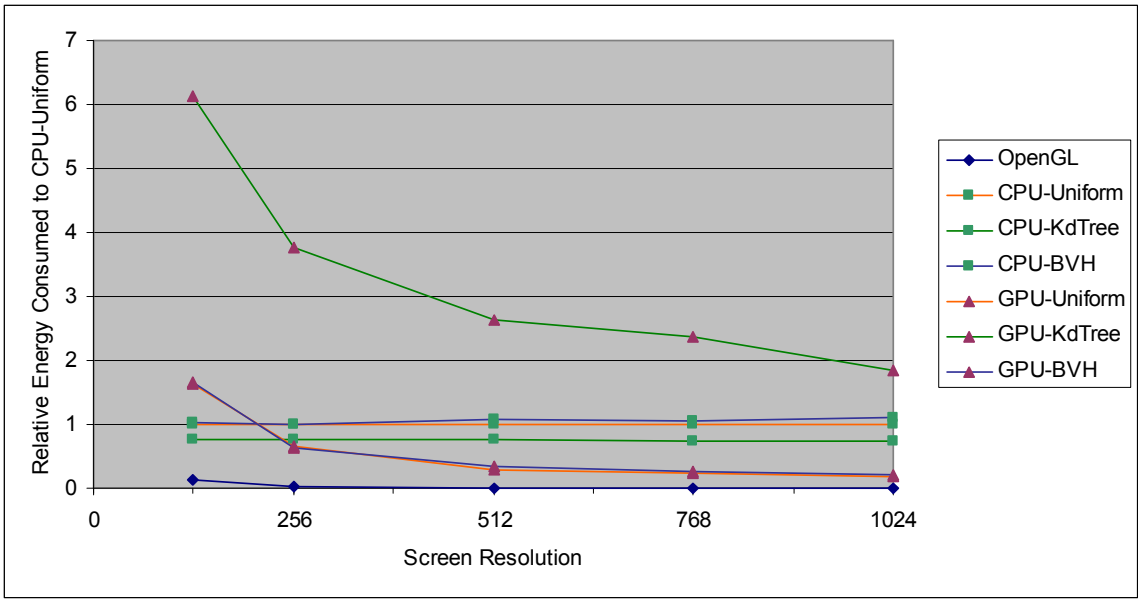

Figure 6-7: Relative energy consumed while ray tracing the Toy Scene compared to the CPU-uniform grid

When normalized against the CPU-based uniform grid (Figure 6-7), the first observation is that the GPU-Kd-Tree is not as efficient as the other structures, performing between 2 and 6 times worse than the CPU-Uniform at every resolution. At the smallest resolution, the other GPU-based implementations are once again ~1.6 times the CPU-uniform grid and are now ~10 times more efficient at high resolutions.

6.1.3 Ray Tracing the Small Dragon Model (48k Triangles)

The separation between CPU and GPU that started to appear with the previous scene is not as clear for this model (Figure 6-8), however the ray tracers do maintain similar ranges. The GPU averages ~5 W more expensive than the CPU. Interestingly, the OpenGL rendering is now comparable to the ray tracers, consuming between 40 and 44 W.

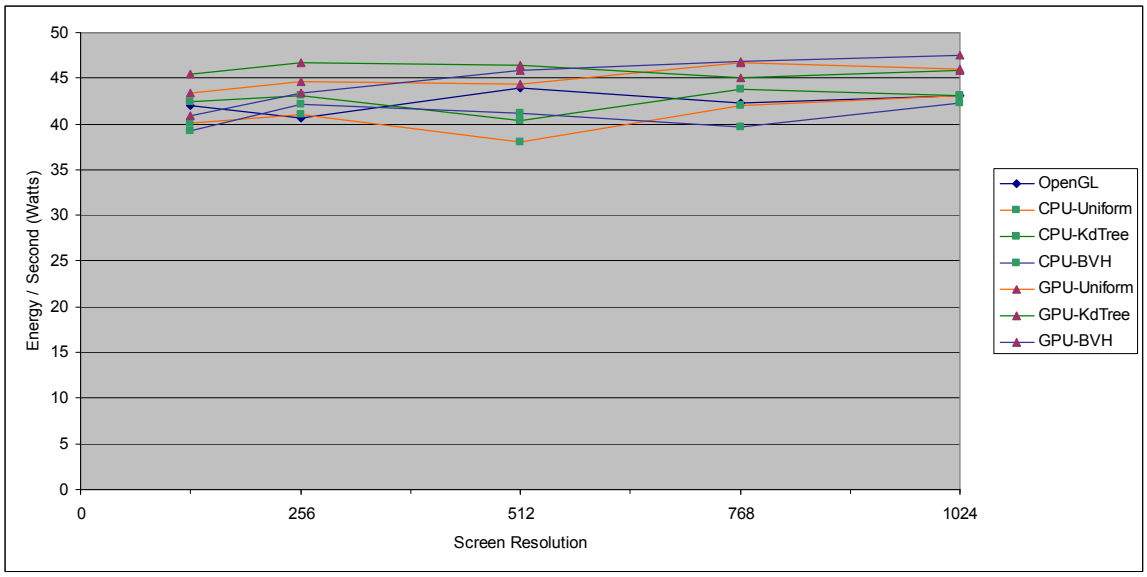

Figure 6-8: Power expended (in Watts) while ray tracing the Small Dragon Model

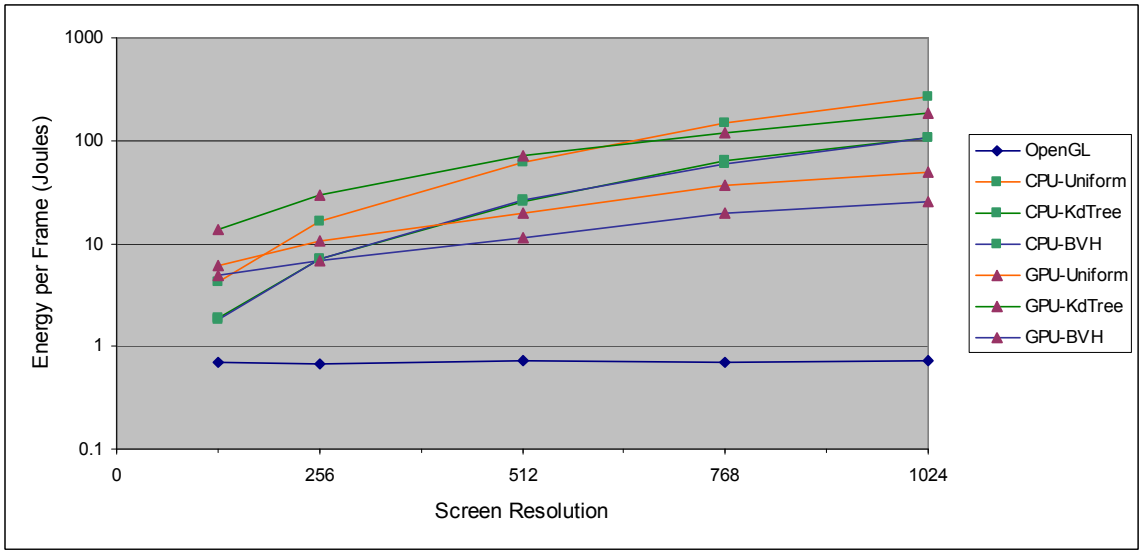

Figure 6-9: Energy cost per frame of ray tracing the Small Dragon Model

Taking the number of frames rendered into consideration (Figure 6-9), the costs of the GPU-based acceleration structures have begun to separate more than they had before, and the Kd-Tree implementation seems to fit more closely with the other structures. On the CPU, the uniform grid has fallen off from the BVH and Kd-Tree implementations, which now have almost identical energy (Figure 6-9) and time (Figure 6-10) costs.

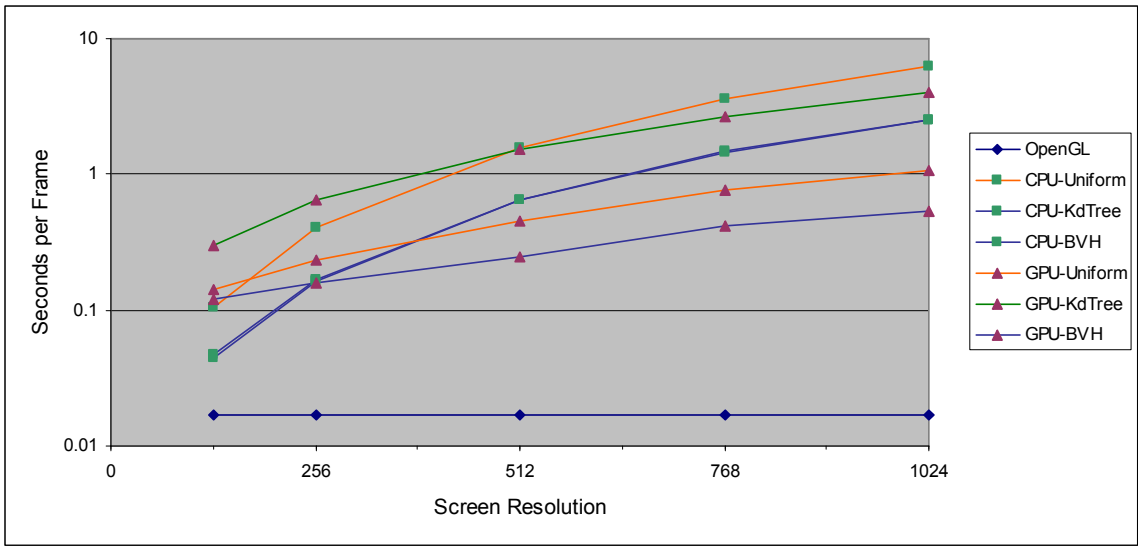

Figure 6-10: Time cost per frame of ray tracing the Small Dragon Model

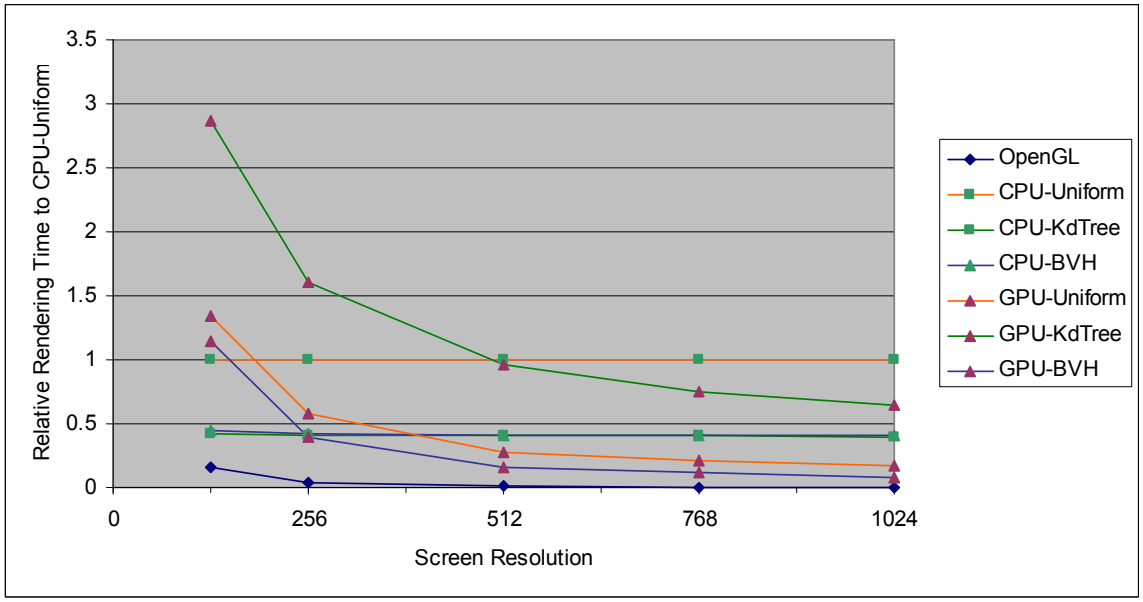

Figure 6-11: Relative energy consumed while ray tracing the Small Dragon compared to the CPU-uniform grid

Comparing the accelerations to the CPU-uniform grid (Figure 6-11), the CPU-BVH and CPU-Kd-Tree consume less than half the amount of energy per frame than the uniform grid. At small resolutions, the GPU-based structures consume between 1.1 and 3.25 times the amount of energy, but all are more efficient at resolutions of 768x768 and

above. The best of the GPU-based structures is the bounding volume hierarchy, consuming less than 1/10 the energy of the CPU-based uniform grid.

For rendering the small dragon model, the CPU-based BVH and Kd-Tree were the most efficient acceleration structures for resolutions less than 256x256 and the GPU-based BVH was the most efficient for larger resolutions.

6.1.4 Ray Tracing the Stanford Bunny Model (70k Triangles)

The trend of the GPU-Kd-Tree as an outlier continues here, as the geometry prevented it from being able to properly render. As the screen resolution increases, more rays are being directed into the same area of the model. This raises the likelihood of rays grazing the edge of the model and requiring many more traversals. Due to the traversal mechanism of the GPU-Kd-Tree, each time the ray moves into a new node, its position must be recalculated from the root node of the tree. The limitation in the number of loops that can be performed on the GPU is exceeded by the number of traversals that must occur in order to fully traverse a model that is this complex. As such, data for resolutions higher than 256x256 could not be fairly collected.

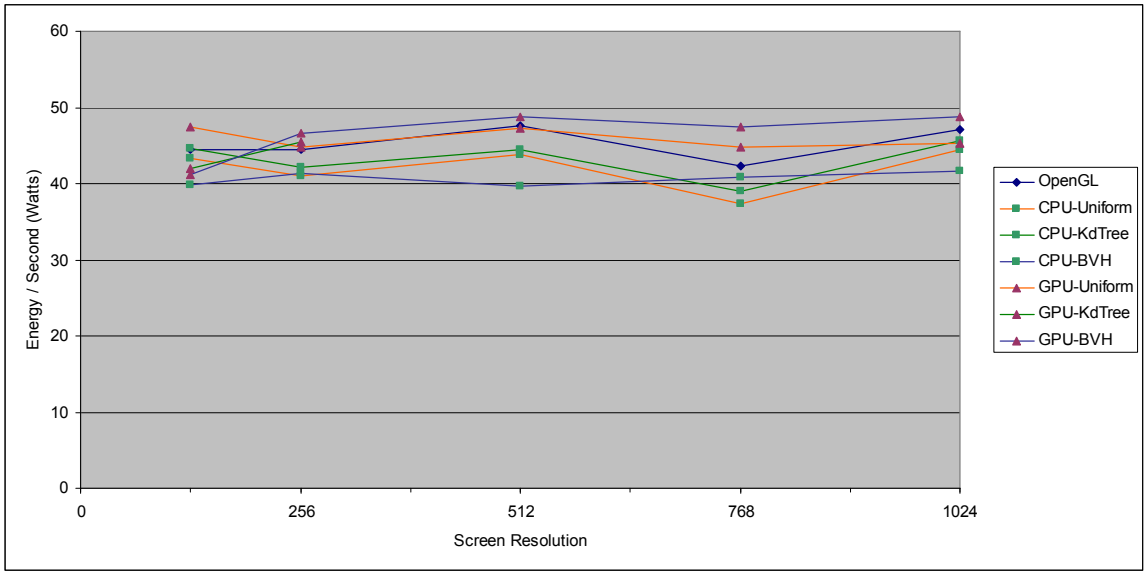

Figure 6-12: Power expended (in Watts) of ray tracing the Stanford Bunny Model

Looking at the ratio of energy per second of rendering (Figure 6-12), the GPU is still slightly more expensive than the CPU at resolutions greater than 256x256, ranging

from 45 - 49 W compared to the CPU range of 37 - 45 W. Once again, the OpenGL rendering consumes similar power as ray tracing. For the first time we see that there is a consistent difference in power consumption in rendering at a specific resolution. At a resolution of 768x768 every structure consumed less power than it did at the surrounding two resolutions, with the exception of the CPU-BVH. Although it does not represent that trend, it consumed the least amount of power at all other resolutions. It was also the most consistent structure, only fluctuating 2 W across all of the resolutions.

Figure 6-13 shows the energy consumed in rendering a single frame. We see that the CPU-Uniform grid and Kd-Tree are generally the least efficient structures for ray tracing. At the smallest resolution, the GPU-Uniform grid is the least efficient structure while the CPU-Kd-Tree is the most efficient, differing by roughly 15 Joules per frame. At resolutions of 256x256 and higher, the GPU-BVH makes a strong case for being the most efficient structure for ray tracing.

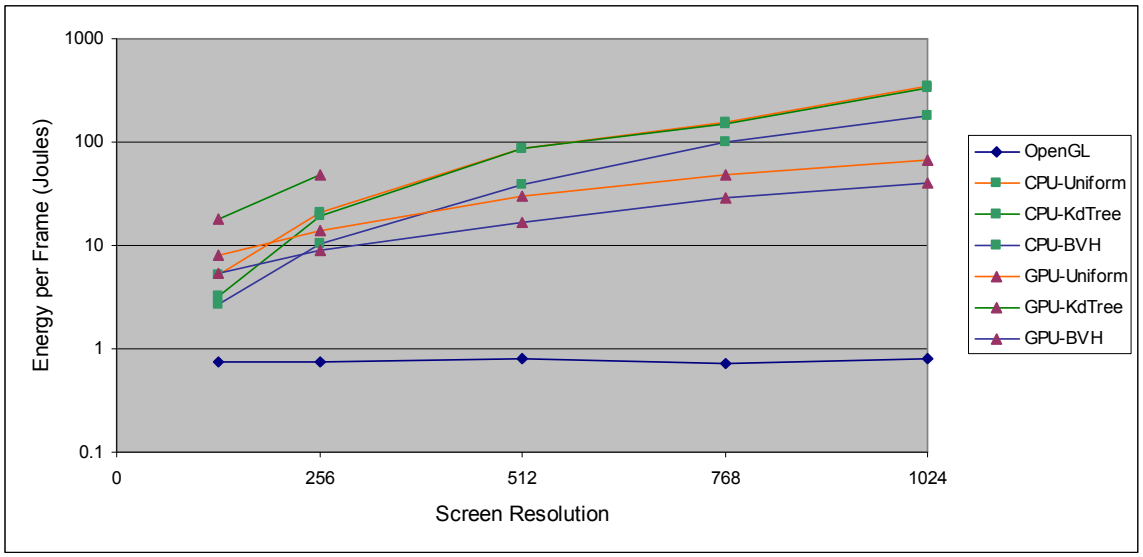

Figure 6-13: Energy cost per frame of ray tracing the Bunny Model

At a resolution of 128x128, the CPU-based structures are more efficient than the GPU-based structures, with the most efficient being the CPU-BVH using 50% of the energy of the CPU-based uniform grid (Figure 6-14). At resolutions of 256x256 and larger, the GPU-BVH becomes the most efficient structure, consuming only 11% of the energy compared to the CPU-based uniform grid at the largest resolution. With a model

of this size, it becomes apparent that the BVH is the most efficient structure regardless of the processing unit used.

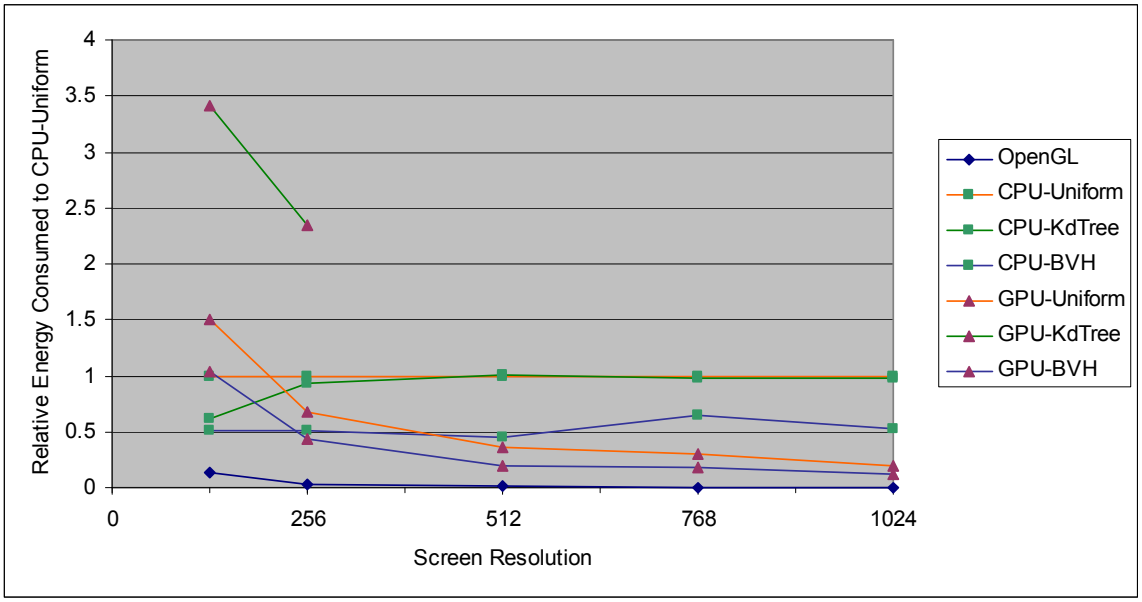

Figure 6-14: Relative energy consumed while ray tracing the Bunny Model compared to the CPU-uniform grid

6.1.5 Ray Tracing the Complex Scene (99k Triangles)

The Kd-Tree on the GPU was unable to fully traverse the scene at any screen resolution and thus does not appear on any of the results. The energy-time ratio (Figure 6-15) appears much different for this scene than it did for previous ones. The CPU-based structures used between 38.5 and 44 W, while the GPU-based structures used between 40 and 53.5 W. The OpenGL rendering followed a similar trend as the GPU-based uniform grid.

Considering the energy cost per frame (Figure 6-16), the GPU appears as the preferred processing unit for resolutions of 256x256 and larger. The GPU-BVH was consistently the most efficient structure consuming 7.6 Joules per frame at a resolution of 256x256 and 39.6 Joules per frame at the largest resolution. The CPU-Kd-Tree is the most efficient structure at the smallest resolution of 128x128, consuming only 3.2 Joules per frame.

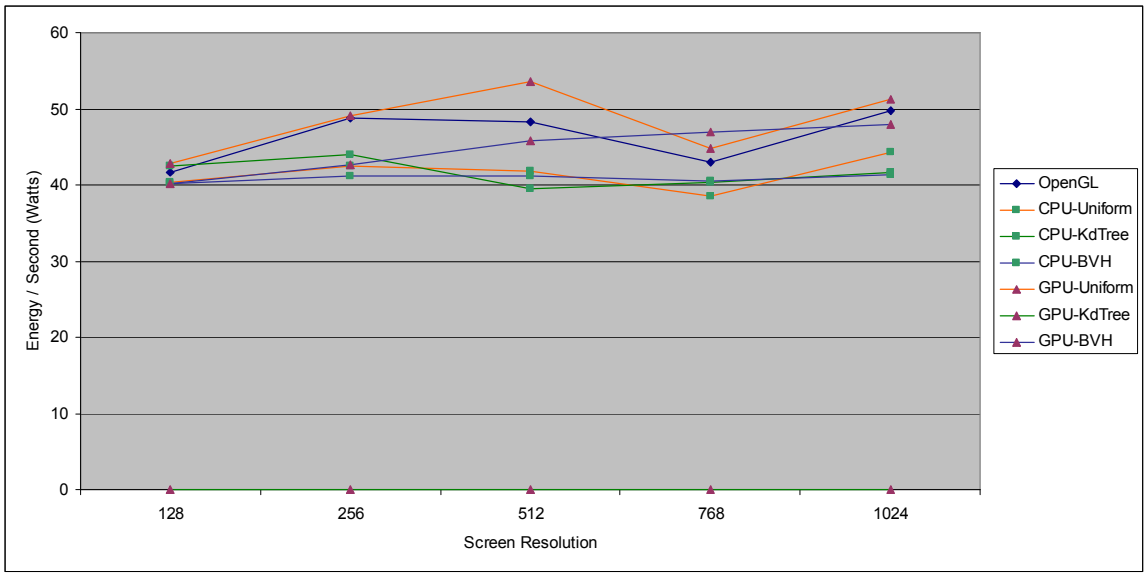

Figure 6-15: Power expended (in Watts) for ray tracing the Complex Scene

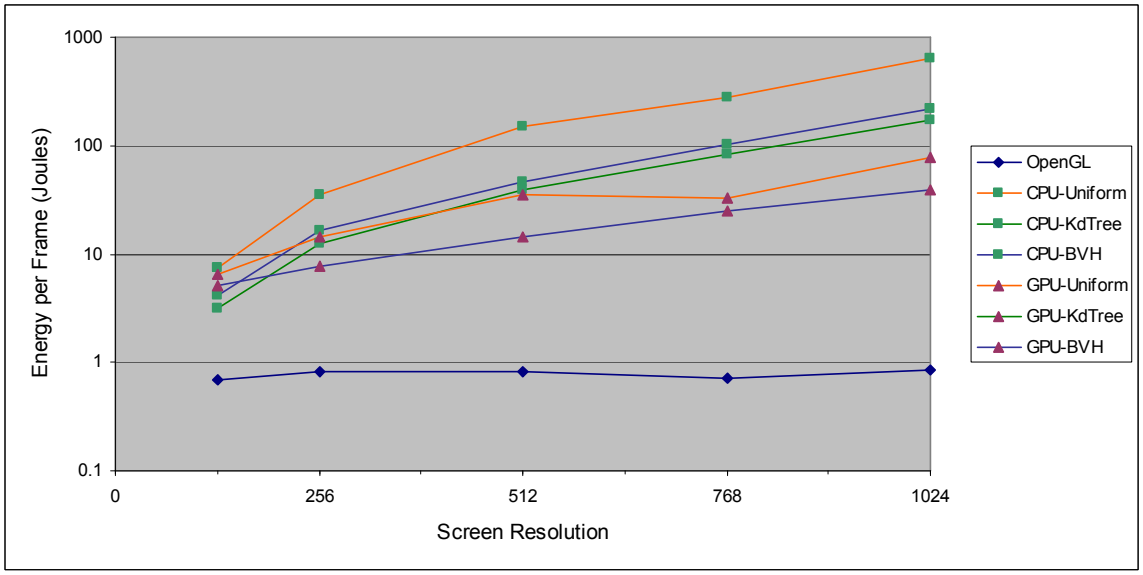

Figure 6-16: Energy consumption of ray tracing the Complex Scene

Figure 6-17 shows how each of the structures compares to the CPU-based uniform grid; at resolutions of 256x256 and greater, all the structures consume less than half the energy. The most efficient structure at these resolutions was the GPU-BVH consuming 20% the energy of the CPU-based uniform grid at a resolution of 256x256 and only 6% at a resolution of 1024x1024. At the smallest resolution, the CPU-Kd-Tree was the most efficient, consuming 42% as much energy.

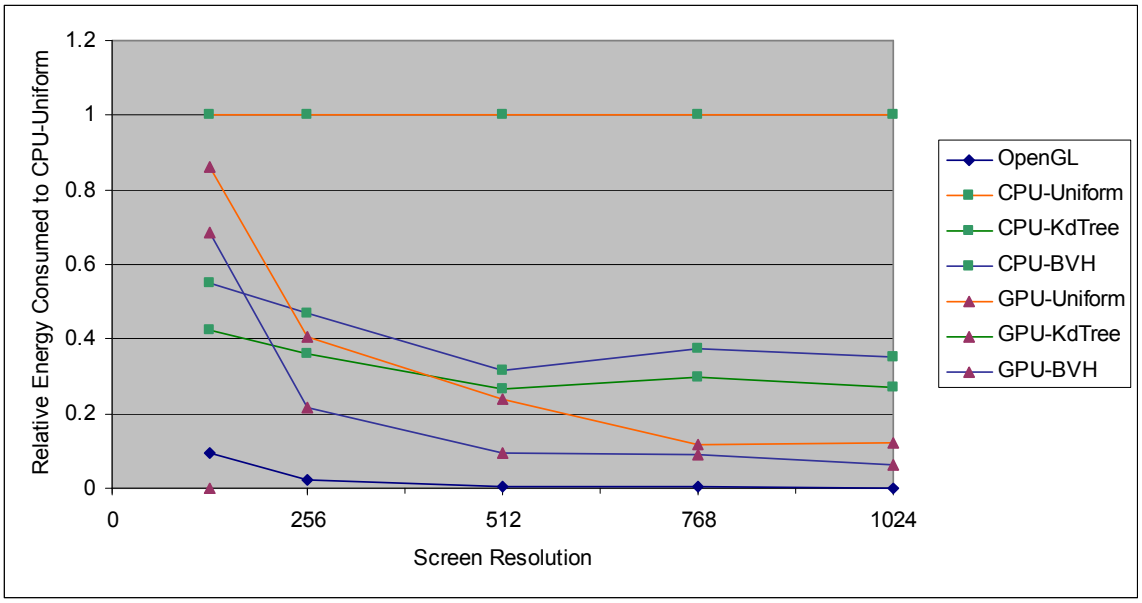

Figure 6-17: Relative energy consumed while ray tracing the Complex Scene compared to the CPU-uniform grid

6.2 Applicability to Mobile Devices

We aimed to identify an efficient acceleration structure for cell phones, personal digital assistants (PDAs), and laptops. In order to better target a particular mobile device, we find the viewport resolutions used during testing that most closely matches the screen resolution of the target device. Given the set of test results that correspond to the resolution, we compare each acceleration structure and processing unit over the set of test scenes. The energy-efficient acceleration structure for that resolution is found by comparing the worst result for each of the structures. The structure whose worst result is better than the other structures' is labeled the most energy-efficient. Our test results can then be used to approximate the expected performance of ray tracing on the target device using Equation 6-1.

$$ExpectedPerformance = \frac{AverageTest\,\mathrm{Re}\,sultForTheEfficientStructure}{PixelsInTest\,\mathrm{Re}\,solution} * PixelsInDevice$$

Equation 6-1: Expected performance of a mobile device based on the test results

6.2.1 Cell Phones

Although there are a wide variety of cellular phones available, as of this thesis, many of them have a screen resolution of 128x160 and do not contain programmable GPUs. Our results showed that at a resolution of 128x128, the CPU was the most efficient processing unit, so these results can be used to approximate the expected performance of ray tracing on a common cell phone. High-end cellular phones that are currently available have a programmable GPU and a primary screen resolution of 240x320 pixels. We use the results of rendering at a resolution of 256x256 to approximate ray tracing on a GPU-enabled cell phone.

Figure 6-18 compares the energy cost per frame of ray tracing the test scenes at a viewport resolution of 128x128. There is a positive linear trend between energy cost and triangle count on the CPU-based structures, all of which have a lower average than their GPU-based counterparts. Using the highest energy cost of each structure to rank them, the CPU-Kd-Tree is labeled as the most energy-efficient structure. It was able to render the Stanford Bunny Model (70k triangles) and the Complex Scene (99k triangles) with approximately the same energy cost of 3.2 Joules-per-frame; this cost is only slightly more than 2.5 times the energy cost of rendering a Single Triangle (1.27 Joules-per-frame).

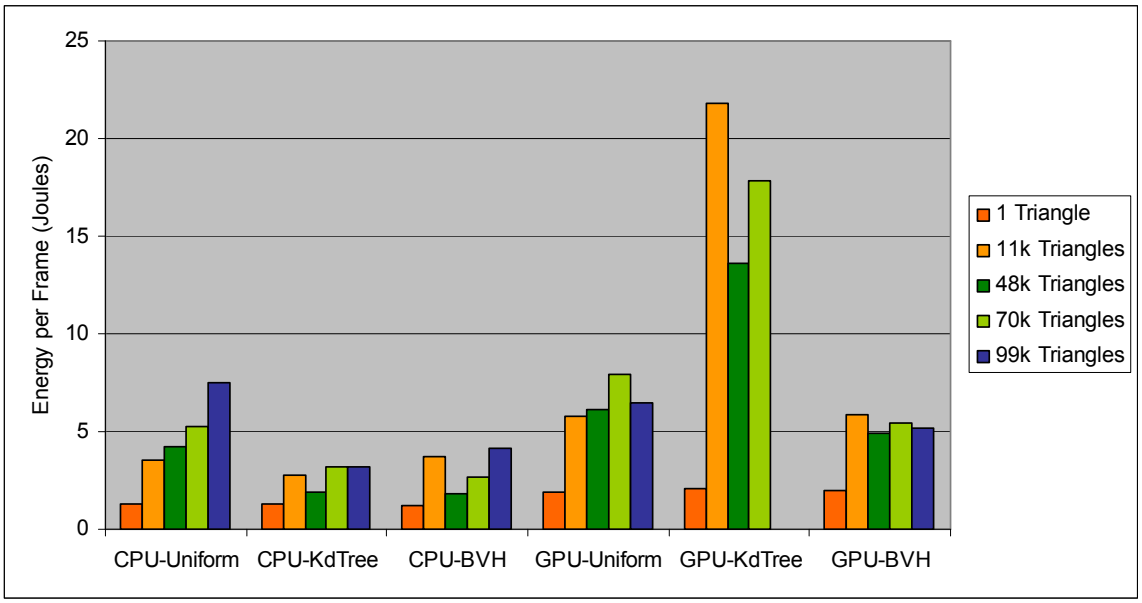

Figure 6-18: Comparing energy-efficiency of ray tracing at a resolution of 128x128

The CPU-Kd-Tree was able to ray trace the scenes with an average of 2.76 Joules per frame. This suggests a per-pixel (or per-ray) cost of $1.68 * 10^{-4}$ Joules-per-pixel for each of the 16,384 pixels. On a cell phone with a 128x160 resolution (20,480 pixels) we can expect ray tracing to consume 3.45 Joules-per-frame. The CPU-Kd-Tree was able to maintain an average frame rate of ~15.56 frames-per-second (fps) while ray tracing the test scenes. Appling Equation 6-1 with this value, it can be expected that a low-end cell phone could ray trace the scenes while averaging 12.45 fps.

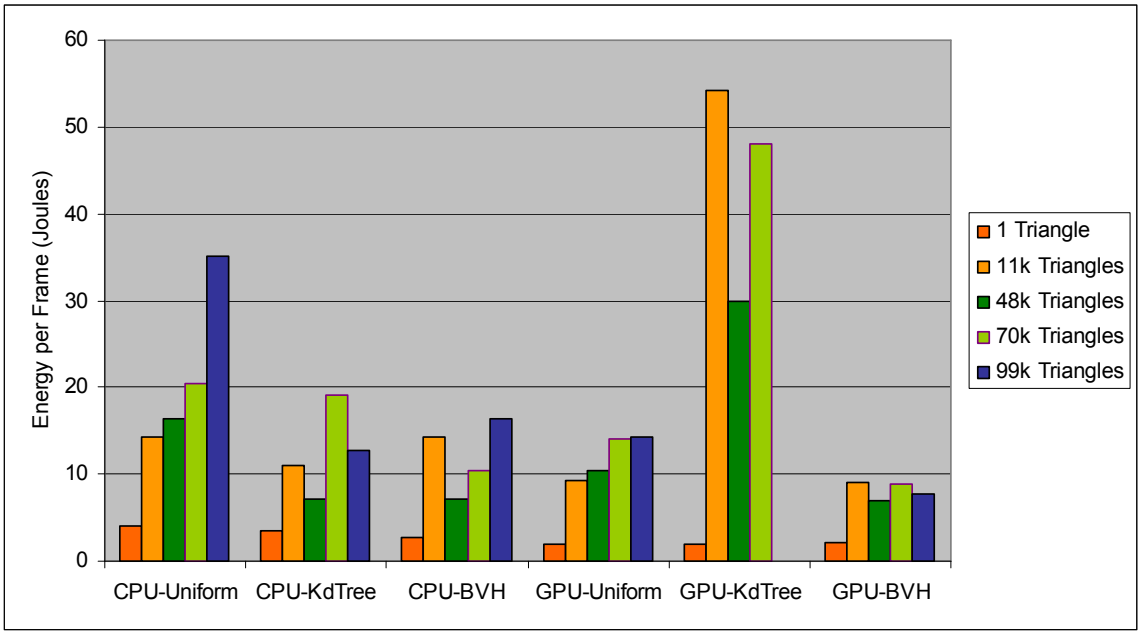

Figure 6-19: Comparing energy-efficiency of ray tracing at a resolution of 256x256

High-end cell phones that contain programmable GPUs can best be approximated with the test resolution of 256x256. A comparison between each of the acceleration structures' efficiency at ray tracing the test scenes is available in Figure 6-19. The GPU-Kd-Tree is the most energy-hungry of the acceleration structures; this was also evident in Figure 6-18. At first glance, there does not appear to be a significant savings ray tracing with one acceleration structure over another, but there is a trend that is consistent for the uniform grid on both processing units. There is a positive relationship between energy consumption and triangle count for the uniform grid; this trend is also mostly true for the

CPU-based Kd-Tree and BVH, but they occasionally have a higher energy cost associated with a lower triangle count. For the 256x256 resolution, the GPU-BVH is labeled the most energy-efficient structure, as it maintains an almost constant energy usage (within 2.2 Joules) for all scene sizes. The Toy Scene was the most expensive scene for the GPU-BVH to ray trace, but it managed to do so more efficiently than any of the other structures.

We can better approximate the results for a cell phone by averaging the GPU-BVH results for all scenes (not including the Single Triangle) and then normalizing over the number of pixels (65,536). The average cost for rendering the scenes was 8.09 Joules per frame, or $1.23 * 10^{-4}$ Joules-per-pixel. Given a cell phone's resolution of 240x320 (76,800 pixels), we can expect ray tracing to cost 9.45 Joules-per-frame. Applying Equation 6-1 to the time costs per frame, we expect to ray trace at 4.61 frames per second. This rate is slightly slower than the average frame rate of 5.4 fps achieved during testing.

6.2.2 Personal Digital Assistants

At the time this thesis was written, a PDA which had a programmable GPU was available with a 480x640 resolution screen. We use the results of rendering at a resolution of 512x512 to approximate ray tracing on a PDA.

Figure 6-20 displays the energy cost per frame of rendering at a resolution of 512x512. There is still an increasing trend in energy costs as the number of triangles increases. This trend is most evident on the CPU-based implementations and on the GPU-uniform grid. The GPU-based Kd-Tree remains an outlier consuming a drastically different amount of energy depending on the scene it is ray tracing. Once again, the bounding volume hierarchy on the GPU defies the other trends. It was still able to ray trace the larger scenes between 2.6 - 8 Joules-per-frame more efficiently than the Toy Scene. Although the GPU-uniform grid was 2.7 Joules-per-frame more efficient at rendering the Toy Scene, the BVH was more efficient than all structures for the larger scenes. Despite the ~900% increase in triangles between the Toy Scene and the Complex Scene, the GPU-BVH was able to ray trace the Complex Scene using 73.7% the energy per frame as it did while ray tracing the Toy Scene.

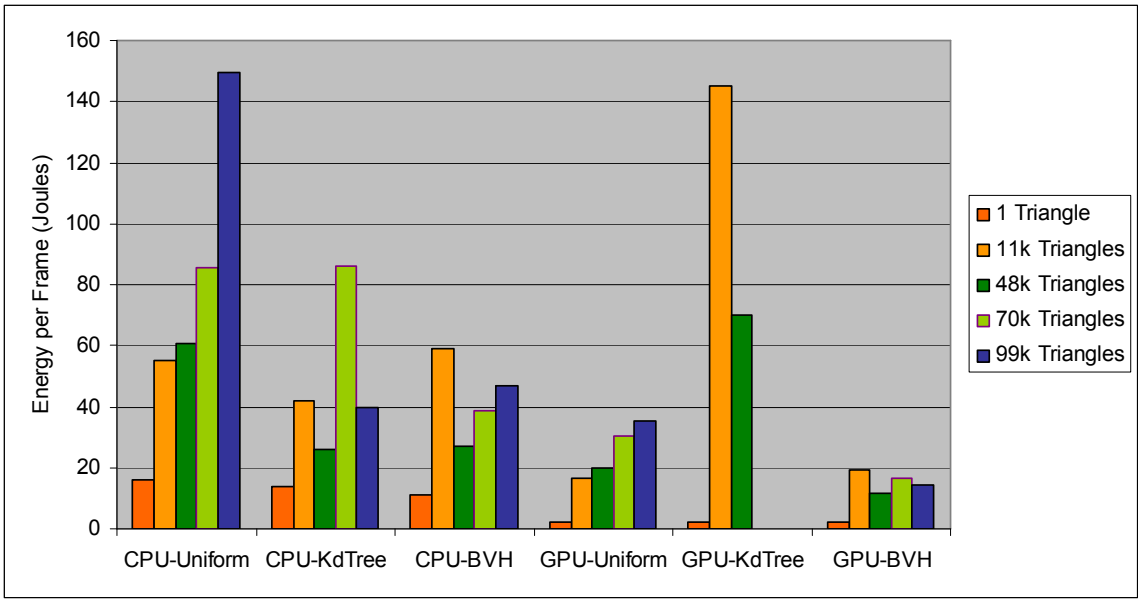

Figure 6-20: Comparing energy-efficiency of ray tracing at a resolution of 512x512

The average energy cost of rendering with the GPU-BVH was 15.42 Joules-per-frame, or $5.88 * 10^{-5}$ Joules-per-pixel. This expands out to rendering on a PDA with resolution 480x640 at 18.07 Joules-per-frame. Frame rates on the PDA are expected to be 2.60 fps. Although the resolution is four times larger than a cell phone, the energy cost on the PDA is less than half and frames are rendered at slightly more than half the speed.

6.2.3 Laptops

Laptop screens come in a variety of sizes and there are a variety of programmable GPUs available for them. At this point in time, most laptops with programmable GPUs have either 15" or 17" screens; a typical 15" screen has a resolution of 1280x800 and a typical 17" screen has a resolution of 1440x900. For the purposes of this thesis, we will use the 1024x1024 test resolution to approximate the performance on a typical 17" laptop screen. This is in accordance with the fact that even larger laptop displays are available, and that in the future, either the displays will be bigger, or the pixels will become smaller, fitting more pixels into the same size screen.

Similar trends exist at a resolution of 1024x1024 (Figure 6-21) as they did for smaller resolutions. The CPU-based structures and the GPU-uniform grid show a strong positive trend with triangle count; the GPU-Kd-Tree is also very inconsistent. The GPU-

based implementations are much more efficient at this resolution than the CPU-based ones. Once again, the GPU-BVH continues to out-perform the other acceleration structures, with the exception of being 7.3 Joules-per-frame more expensive than the GPU-uniform grid for the Toy Scene. However, the BVH consumes only half the energy per frame than the uniform grid for the Complex Scene.

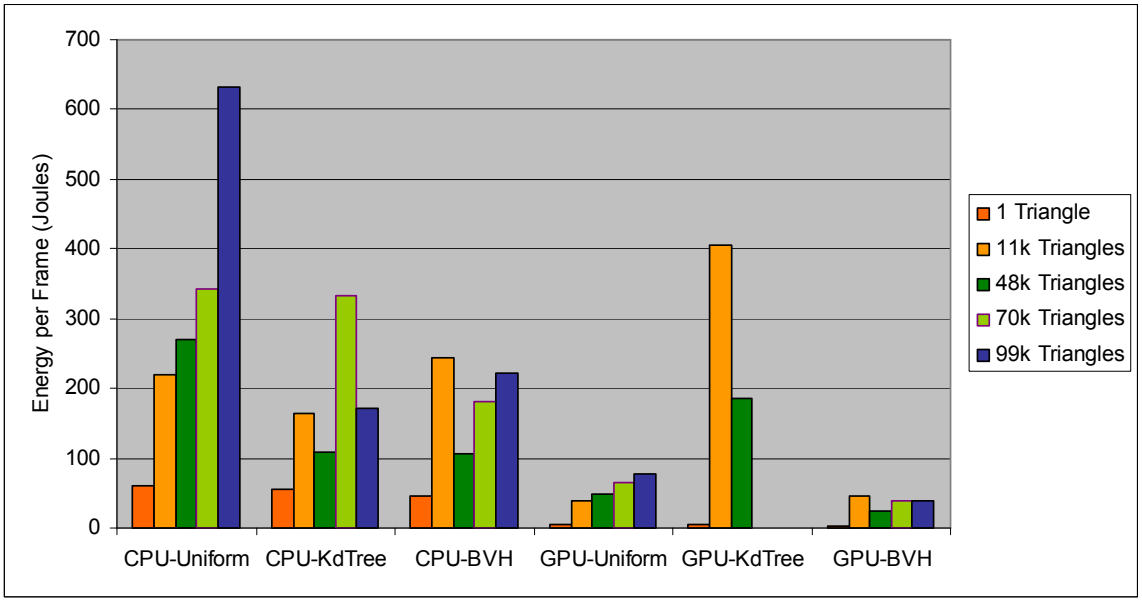

Figure 6-21: Comparing energy-efficiency of ray tracing at a resolution of 1024x1024

The GPU-BVH has a range of 20.1 Joules-per-frame depending on the scene being ray traced, with an average energy cost of 37.39 Joules. This energy cost per frame corresponds to $3.57 * 10^{-5}$ Joules-per-pixel. Over all 1440x900 pixels on the laptop screen, ray tracing would cost 46.21 Joules-per-frame at a rate of 1.42 fps.

7 Conclusions

The results show that at small resolutions (~128x128) the CPU is the most efficient processing unit for ray tracing. The Kd-Tree and BVH were both more energy efficient than the uniform grid. Comparing the structures on an average-energy-cost-per-frame basis, the Kd-Tree appears as the most efficient structure. At resolutions of 256x256 and higher, the GPU becomes the most efficient processing unit. The Kd-Tree implementation on the GPU was unable to render some scenes due to the limitations of the GPU, but the uniform grid and BVH both provided promising results. Depending on the scene, the uniform grid was more efficient than the BVH, however, as the number of triangles in the scene increased, the uniform grid's energy consumption per frame also increased. This trend did not exist for the BVH. In fact, as the number of triangles increased, the BVH had an energy-efficiency that remained within a small threshold of being constant.

Several trends were apparent throughout the tests. Given the decision to render using ray tracing, the choice of screen resolution has the largest impact on energy consumption and frame rate. This is partly because the screen size determines the total number of rays (or work) that has to be done. The efficiency can then be targeted based on the choice of using the CPU or the GPU. Finally, the choice of acceleration structure has the smallest impact on the efficiency of ray tracing (assuming the implementation can fully render the scene). In general, at a given screen resolution, the relative efficiency of the different acceleration structures was not affected by the number of triangles being ray traced. Similarly, none of the acceleration structures were significantly impacted when rendering a scene versus a single model.

Using the various test resolutions, the energy cost per ray can be calculated and used to estimate the cost of energy-efficient ray tracing on various mobile devices. Table 7-1 shows a summary of these estimations. A typical cellular phone has a screen resolution of 128x160 and does not have a programmable GPU. It is convenient that at this resolution, the CPU-based Kd-Tree is the most energy-efficient acceleration structure for ray tracing. It is expected that a frame rate of 12.44 fps can be obtained while expending only 3.45 joules-per-frame. Higher-end cell phones are available with a

programmable GPU and have a screen resolution of 240x320. A GPU-based BVH is the most energy-efficient acceleration structure for one of these phones, and an energy cost of 9.45 Joules per frame can be expected at a frame rate of 4.61 frames per second.

Table 7-1: Summary of estimated energy costs and frame rates of ray tracing on various mobile devices using the most energy-efficient acceleration structure

	Screen Resolution	Acceleration Structure	Energy / Frame	Frame Rate
Cell Phone	128x160	CPU-KdTree	3.45	12.44
Cell Phone	240x320	GPU-BVH	9.45	4.61
PDA	480x640	GPU-BVH	18.07	2.60
Laptop	1440x900	GPU-BVH	46.21	1.42

A personal digital assistant that has a screen resolution four times larger than a high-end cell phone (480x640) can perform ray tracing using a GPU-based BVH at 2.6 frames per second at 18.07 Joules per frame. A typical laptop with a screen size of 17" has a maximum resolution of 1440x900. Our results estimate ray tracing at such a resolution to cost 46.21 Joules per frame at a rate of 1.42 frames per second using a GPU-based BVH. Despite the fact that this resolution is 16.875 times as large as the high-end cell phone, the energy consumption of the laptop is only 4.9 times that of the cell phone.

7.1 Future Work

We have presented the first results of using energy efficiency to compare acceleration structures and processing units for ray tracing. Our analysis has shown the bounding volume hierarchy implemented on a programmable graphics processing unit to be the most energy-efficient acceleration structure. Additional testing and analysis can still be performed that relates energy consumption of the acceleration structures to the power consumption of the processing unit. The traversal algorithms and the architecture of the system may be further improved by gaining insight into the energy costs of loading textures and of performing different types of computations on the GPU. Given this knowledge, new algorithms can be developed that further improve energy efficiency, or support for current algorithms can be strengthened.

The system can be further improved by dividing the screen-aligned quad into smaller quads before ray tracing the scene, as suggested by Purcell et al [2002]. By

performing ray tracing on smaller quads, the cache can be better utilized and groups of rays that terminate early will not act as a bottleneck for those rays that need further processing with additional executions of the shaders. Although this is understood to improve the rendering time, perhaps having to switch between shaders will increase the energy consumption of the system. Ray tracing multiple quads would have an effect similar to performing packet tracing on the CPU [Wald et al 2001], where each packet would be represented by a quad.

An easy extension to our system would be to execute the experiments on GPUs from different manufacturers and different models in order to identify if the relative efficiency of the acceleration structures is different and to identify the range of energy consumed for each of the structures. It would also be interesting to port the GPU-based ray tracing code from OpenGL to DirectX. Since the two APIs handle texture memory differently, comparing the energy-efficiency of the graphics API could lead to further improvements to our system.

Improved support for mobile devices could be incorporated into the system also. One example would be to store the models, construct the acceleration structures on a server and stream the data, as either data structures or textures, over the network to the mobile devices. Alternatively, the energy consumed in transmitting and receiving the data over the network may exceed that of storing and accessing the model data and constructing the acceleration structures.

Since our work measured the energy consumption of ray casting with the use of primary rays, an interesting extension to the research would be to use multiple rays per pixel to perform anti-aliased ray tracing. This would only require casting more primary rays into the scene and would yield a higher quality image. The results of such research would validate that our results can be extended to reflect the expected results at a different screen resolution based on per-pixel or per-ray costs.

8 References

Akenine-Moller, T. 2001. “Fast 3D Triangle-Box Overlap Testing”. Retrieved May 2006, from http://www.cs.lth.se/home/Tomas_Akenine_Moller/pubs/tribox.pdf

Amanitides, J. and Woo, A. 1987. “A Fast Voxel Traversal Algorithm for Ray Tracing”. Eurographics Conference Proceedings. pp. 3-10.

ARB_pixel_buffer_object. 2004. Retrieved February 2006, from http://www.oss.sgi.com/projects/ogl-sample/registry/ARB/pixel_buffer_object.txt.

Barr, K. and Asanovic, K. 2003. “Energy Aware Lossless Data Compression”. The First International Conference on Mobile Systems, Applications and Services.

Banerjee, K. and Agu, E. 2005. “PowerSpy: Fine-Grained Software Energy Profiling for Mobile Devices”. IEEE WirelessCom Conference Proceedings.

Bikker, J. 2005. “Raytracing: Theory & Implementation Part 4, Spatial Subdivisions”. Retrieved February 2006, from http://www.devmaster.net/articles/raytracing_series/part4.php.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P. 2004. “Brook for GPUs: Stream Computing on Graphics Hardware”. SIGGRAPH 2004. Retrieved March 2006, from http://graphics.stanford.edu/papers/brookgpu/brookgpu.pdf

Christen, M. 2005. “Ray Tracing on GPU”. University of Applied Sciences Basel. Diploma Thesis. Retrieved February 2006, from http://gpurt.sourceforge.net/DA07_0405_Ray_Tracing_on_GPU-1.0.5.pdf

EXT_framebuffer_object. 2005. Retreived February 2006, from http://oss.sgi.com/projects/ogl-sample/registry/EXT/frambuffer_object.txt.

Flinn, J. and Satyanaranan, M. 1999. “PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications”. Second IEEE WMCSA. New Orleans, LA.

Foley, T. and Sugerman, J. 2005. “KD-Tree Acceleration Structures for a GPU Raytracer”. Stanford University. Proceedings of the ACM SIGGRAPH/Eurographics, Workshop on graphics hardware, pp. 15-22.

Fujimoto, A., Tanaka, T., Iwata, K., 1986. “ARTS: Accelerated Ray Tracing System”, IEEE Computer Graphics and Applications, Vol. 6, Issue 4, pp. 16-26.

General Purpose Computations on the GPU. http://www.gpgpu.org.

Goldsmith, J. and Salmon, J. 1987. “Automatic Creation of Object Hierarchies for Ray Tracing”. IEEE Computer Graphics and Applications, Vol. 7, Issue 5, pp. 14-20.

Havran et al. 2000. “Statistical Comparison of Ray-Shooting Efficiency Schemes”. Technical Report TR-186-2-00-14. Institute of Computer Graphics, Vienna University of Technology.

Havran, V. and Bittner, J. 2002. “On Improving KD-Trees for Ray Shooting”. WSCG conference, pp. 209-216.

Hill, F.S. 2001. Computer Graphics using OpenGL. Second Edition. Prentice Hall, Upper Saddle River, NJ, USA.

Intel. 2002. “PC Energy-Efficiency Trends and Technologies”. Retrieved January 2006, from http://cache-www.intel.com/cd/00/00/10/27/102727_ar024103.pdf.

Karlsson, F., Ljungstedt, C. J. 2004. “Ray Tracing Fully Implemented on Programmable Graphics Hardware”. Chalmers University of Technology. Master’s Thesis. Retrieved March 2006, from http://www.ce.chalmers.se/~uffe/xjobb/GPURT.pdf.

Kay, T. and Kajiya, J. 1986. “Ray Tracing Complex Scenes”. ACM Computer Graphics, Vol. 20, Issue 4, pp. 269-278, ACM Press, New York, USA.

Lext, J., Assarsson, U., Moller, T. 2001. “BART: A Benchmark for Animated Ray Tracing”. IEEE Computer Graphics and Applications, Vol. 21, Issue 2, March, pp. 22-31. IEEE Computer Society Press, Los Alamitos, CA, USA.

Moller, T. and Trumbore, B. 1997. “Fast, Minimum Storage Ray/Triangle Intersection”. Journal on Graphic Tools, Vol.2, No.1, pp.21-28, 1997.

Pharr, M., Humphreys, G. 2004. Physically Based Rendering. Morgan Kauffman. ISBN 012553180X.

Purcell, T., Buck, I., Mark., W., Hanrahan, P. 2002. “Ray Tracing on Programmable Graphics Hardware”. ACM Transaction on Graphics 21 (3), pp. 703-712.

SGI. 2004. “The Power of Scalable Visualization”. Retrieved January 6, 2007, from http://www.sgi.com/pdfs/3620.pdf.

SGI, and OpenGL ARB. 2004. Retrieved January 6, 2007, from http://www.sgi.com/company_info/newsroom/press_releases/2004/august/opengl.html.

Stanford 3D Scanning Repository, The. Retrieved February 18, 2006, from http://graphics.stanford.edu/data/3Dscanrep/.

Starner, Thad. 2003. “Batteries and Possible Alternatives for the Mobile Market”. IEEE Pervasive Computing. October-December, pp. 86-88. Retrieved March 2006, from http://doi.ieeecomputersociety.org/10.1109/MPRV.2003.1251172.

Thrane, N. and Simonsen, L.O. 2005. “A Comparison of Acceleration Structures for GPU Assisted Ray Tracing”. Retrieved April 2006, from http://www.larsole.com/files/GPU_BVHthesis.pdf.

Tscheblockov, T. 2004. “Power Consumption of Contemporary Graphics Accelerators”. Retrieved January 2006, from http://www.xbitlabs.com/articles/video/display/ati-powercons.html.

Wald, I., Slusallek, P., and Benthin, C. 2001. “Interactive rendering using coherent ray tracing”. Computer Graphics Forum 20, 3, pp. 153-164.

Wald, I., Ize, T., Kensler, A., Knoll, A., Parker, S. 2006. “Ray Tracing Animated Scenes using Coherent Grid Traversal”. SIGGRAPH, 2006. pp. 485-493.

Woo, A. 1990. “Fast Ray-Box Intersection”. Graphics Gems I. pp. 395-396. Academic Press Professional, Inc., San Diego, CA, USA.

Woo, A. 1992. “Ray Tracing Polygons using Spatial Subdivision”. Proceedings of the Conference on Graphics interface. pp. 184-191.

Wright, R. Jr., Lipchak, B. 2005. OpenGL Superbible. 3rd Edition. Sams Publishing. Indianapolis, IN, USA.