The Study of Energy Consumption of Acceleration Structures for Dynamic CPU and GPU Ray Tracing

By

Chen Hao Chang

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

January 2007

APPROVED:

____

Professor Emmanuel Agu, Principal Thesis Advisor

____

Professor Robert W. Lindeman, Thesis Advisor

____

Professor Mark Claypool, Thesis Reader

____

Professor Michael Gennert, Head of Department

Abstract

Battery life has been the slowest growing resource on mobile systems for several decades. Although much work has been done on designing new chips and peripherals that use less energy, there has not been much work on reducing energy consumption by removing energy intensive tasks from graphics algorithms. In our work, we focus on energy consumption of the ray tracing task because it is a resource-intensive, global-illumination algorithm. We focus our effort on ray tracing dynamic scenes, thus we concentrate on identifying the major elements determining the energy consumption of acceleration structures. We believe acceleration structures are critical in reducing energy consumption because they need to be built inexpensively, but must also be complex enough to boost rendering speed.

We conducted tests on a Pentium 1.6 GHz laptop with GeForce Go 6800 GPU. In our experiments, we investigated various elements that modify the acceleration structure build algorithm, and we compared the energy usage of CPU and GPU rendering with different acceleration structures. Furthermore, the energy per frame when ray tracing dynamic scenes was gathered and compared to identify the best acceleration structure that provides a good balance between building energy consumption and rendering energy consumption.

We found the bounding volume hierarchy to be the best acceleration structure when rendering dynamic scenes with the GPU on our test system. A bounding volume hierarchy is not the most inexpensive structure to build, but it can be rendered cheaply on the GPU while introducing acceptable energy overhead when rebuilding. In addition, we found the fastest algorithm was also the most inexpensive in terms of energy consumption. We propose an energy model based on this finding.

Acknowledgements

First and foremost I would like to thank my thesis advisors, Professor Emmanuel Agu and Robert W. Lindeman, for their guidance and inspirational advice. Emmanuel Agu has always been very encouraging throughout the entire thesis work. Robert W. Lindeman provided helpful technical tips and interesting ideas. They are both very friendly and always open to new ideas and questions. I am especially grateful for the chance Professor Emmanuel Agu has given me to work on my Masters thesis.

My thanks also go to my reader, Professor Mark Claypool, who gave me valuable feedback and suggestions on the thesis. I would also like to thank my partner, Peter James Lohrmann, who worked tirelessly with me to make the ray tracing implementation work well.

Last but not least, I would like to thank my uncle for offering me a place I can call home in Massachusetts. I also want to thank my mother and sister for being supportive even though they are in Taiwan. Special thanks to my friend Mike Chen and his wife who kindly invited me to dinner countless times so I didn't die from hunger. My deepest appreciation goes to Robert W. Lindeman, Emmanuel Agu, Beth, and Amanda N McCullough who provided tremendous help on my English grammar.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Introduction

Photorealistic images are essential in today's movies; interactivity is also desirable in applications such as computer games. Combining photorealism with interactivity has been a challenging research problem in computer graphics. Over the past decade, research has focused on making global illumination algorithms such as ray tracing, photon mapping, and radiosity, run at interactive frame rates. New, powerful Graphics Processing Units (GPUs), which can process billions of triangles per second, provide new processing platforms for global illumination algorithms. This means interactive global illumination calculation is possible by utilizing GPUs. The demand for high-quality graphics on mobile devices is growing as well, such as playing 3D games on cell phones, or allowing real-estate customers to take a virtual tour of a new house. Although mobile devices are faster and more powerful than in the past, they are resource limited especially in terms of energy. As shown in Figure 1, the energy capacity has only grown by a factor of three, while CPU speed, Disk Capacity, and available RAM have grown by factors of more than a hundred since 1990.

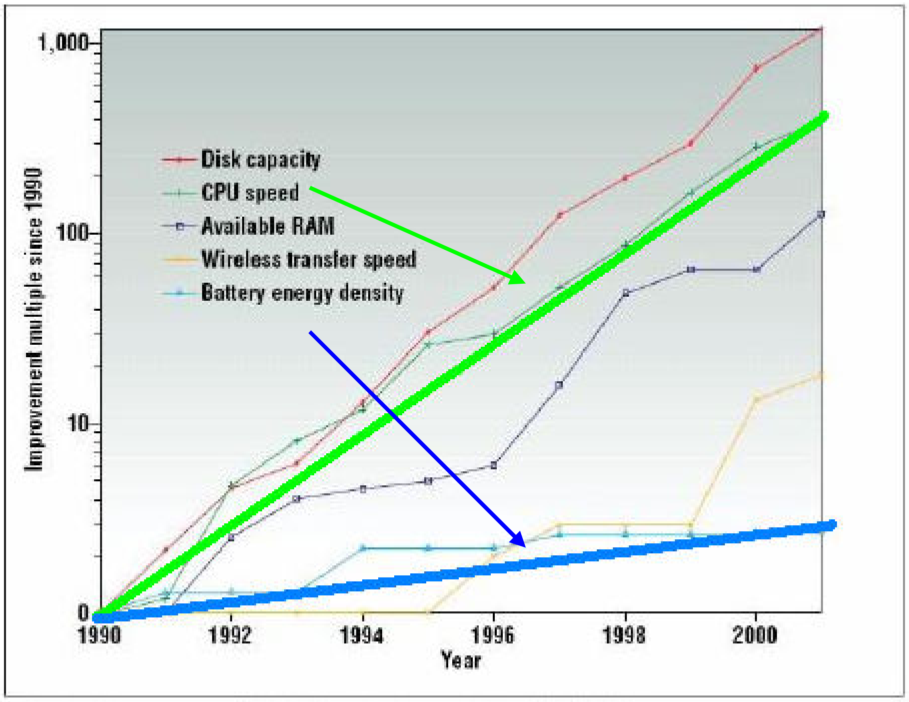


Figure 1 Technology for Laptop from 1990 to 2001, adapted from [Starner 2003]

The battery energy density curve from Figure 1 shows that more energy equals heavier and bigger batteries, which goes against the current trend towards smaller and thinner mobile devices. Researchers have tried to solve this problem by designing new, smarter CPUs and GPUs that consume less energy while retaining the relatively same performance compared to older generations of chips. The introduction of mobile GPUs, such as the GeForce Go series from nVidia and the Mobility Radeon series from ATI for notebooks, are good examples of GPUs that feature power management technology to reduce energy consumption. Newer dual-core CPUs from Intel and AMD also emphasize energy saving features. This shows chip designers are aware of the energy limitations, and more work is being redirected to address the energy issue.

Recent work on ray tracing with GPUs and SIMD (Single Instruction, Multiple Data) CPUs, where ray tracing takes advantage of the SIMD instruction set, allows ray tracing systems to achieve interactive rendering with shadows, reflections, refractions, motion blur, and more. It has even been shown that it is possible to ray trace animations at interactive rates. However, can it be done using less battery energy? Knowing that battery energy will continue to be the limiting resource for some time in the future, we are interested in finding out the primary components of ray tracing that consume the majority of energy. We focus our effort on the acceleration structures used to speed up ray tracing because ray tracing engines typically spend most of their processing time building, traversing acceleration structures, and calculating intersections.

1.2 Thesis Goal

Recent work on the CPU and GPU have shown interactive ray tracing is possible. With the continuous advances in speed of the CPU and GPU, we believe one day that the same algorithm that today can only achieve five frames per second will eventually be able to run at 30 frames per second or higher, and become a suitable rendering technique for interactive 3D applications such as video games. The only problem left is energy; the algorithms are more likely to consume more energy as the processors get faster. We might be able to have interactive ray traced images, but we will not be able to view them long enough to enjoy them because of the battery limitation on the mobile devices. Thus, we focus on identifying the major components in the building of acceleration structures that stress the battery the most.

Furthermore, we study the energy consumption of today’s hardware with CPU and GPU ray tracing rendering of static and dynamic scenes. In doing so, we hope to allow future studies to improve the energy efficiency of ray tracing.

2 Background

In this section, basic ray tracing will be described. After introducing ray tracing, we will describe GPU-based ray tracing and recent work that has enabled ray tracing to render at interactive rates for both static and dynamic scenes.

2.1 Ray Tracing

Ray tracing is a global-illumination algorithm that can easily generate physically correct reflections, refractions, and shadows. The basic idea for global illumination is to capture all the light properties in the environment. One way to do this is to shoot rays from the lights. If a light ray hits an object, it will light the object based on its surface properties. A light ray that lands on that object can bounce off in different directions, creating shadows, reflections, and refractions. The light ray will eventually end up in the viewer's eye. At this point, the viewer should see the color gathered by this light ray, hence the term ray tracing. This approach can be very computationally intensive because not all the light rays will end at the viewer's eye and the light rays can be infinitely long. Therefore, ray tracing usually traces rays from the viewer's eye to the lights because only the light rays visible to the viewer are generated. Figure 2 illustrates the process of ray tracing.

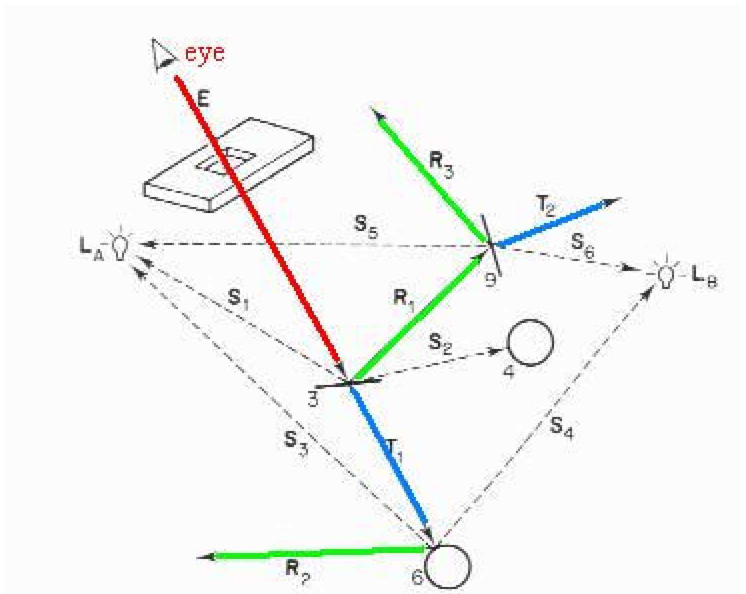


Figure 2 Ray Tracing Illustration, adapted from [Glassner 1989]

The ray coming from the eye, *E*, is the primary ray. This ray determines the color and shape of the objects a viewer will see. In Figure 2, the primary ray intersects with a plane with reflective and refractive properties; therefore, reflection and refraction rays are generated, labeled R1 and T1 in the figure. The reflection ray hits a plane labeled 9, and the refraction ray hits a ball labeled 6. This means the viewer should see the color of plane 9 reflected on plane 3 and the refracted color from ball 6. In addition, the visibility of the primary ray needs to be tested by shooting a shadow ray towards the light. The shadow rays are shown as dotted lines in the figure. If the shadow ray is blocked by another object, the point must be in shadow; otherwise, the viewer should see the surface with the color gathered from reflection, refraction and primary rays. This process is repeated for all the visible points from the current viewer's viewing angle. Furthermore, the reflection and refraction rays are not limited to one bounce; they can bounce forever until the viewer is satisfied with the final image. However, the computational costs grow exponentially as more bounces are allowed.

The process of only shooting the primary rays from the eye into the scene (and not shooting reflection, refraction, and shadow rays) is called *ray casting* and it can only produce direct illumination lighting effects similar to the scan-line algorithm used on commercial graphics cards. It is the ability to cast additional reflection, refraction, and shadow rays that makes ray tracing able to produce photorealistic images. Ray tracing can also be easily extended to produce motion blur, camera lens focal effects, caustics, and more.

Another advantage of ray tracing over the traditional scan-line algorithms is that it is not limited to trianglular geometries. It can be extended to recognize spheres, planes, tetrahedrons, and various shapes defined by mathematical equations. This allows more flexibility in representing models and saves the trouble of approximating analytical objects with triangles.

Ray tracing has traditionally been used as an off-line technique because it has not been possible to render the images at interactive rates due to the massive number of computations required. The majority of computations come from ray-triangle intersection tests. This intersection test reports the location and the object intersected for the ray in question. Knowing the location of all the intersection points, the algorithm can color the point based on the objects' surface properties and shoot reflection, refraction, and shadow rays. Given 1,300 triangles in a scene, drawing this model at a 1024^2 screen resolution with only primary rays requires 1.3 billion ray-triangle intersections. With the addition of shadow, reflection, and refraction rays, a total of 5.2 billion ray-triangle intersections are required to finish this image assuming one bounce for reflection and refraction. Assuming each intersection test can be completed in one nanosecond, the image needs 5.2 seconds to render. Furthermore, this is for a model with only 1,300 triangles. A typical 3D game or animation can contain more than 50,000 triangles in a scene. This translates to roughly three minutes per frame, an unacceptable time to be considered as interactive.

One solution for speeding up ray tracing is to use acceleration structures which help by reducing the number of ray-triangle intersections per ray. In the ideal case, every ray only performs one ray-triangle intersection, which adds up to one million ray-triangle intersection tests. If each intersection test can be completed in one nanosecond, the scene with 1,300 triangles can now be completed in 0.5 milliseconds, and the image can be rendered at 2,000 frames per second. Acceleration structures partition the triangles in the scene to help the rays avoid unnecessary ray-triangle intersection tests, thus boosting ray tracing performance tremendously. It is typically not possible to build an acceleration structure that achieves one ray-triangle intersection per ray for all possible geometry arrangements; nevertheless, every acceleration structure strives to achieve this goal. The most commonly used acceleration structures are the uniform grid, Kd-Tree and Bounding Volume Hierarchy (BVH). They are described in more detail in Section 3. Besides acceleration structures, clever implementations that make efficient use of CPU caches and Single Instruction Multiple Data (SIMD) instructions have been shown to further improve ray tracing performance [Wald 2004].

2.2 GPU Assisted Ray Tracing

Graphics processing units (GPUs) are the main processing chips residing on commercial graphics cards. They are designed to process a large number of triangles quickly in parallel to present interactive 3D images. Graphics cards implement the scan-line algorithm in hardware and are getting faster every year.

Figure 3 shows that GPUs are faster than CPUs on floating point calculations and that GPU performance grows by a factor of 30 or more each year. This suggests that the GPU is a good working platform for floating-point-intensive tasks such as ray tracing where ray-triangle intersection testing is a floating-point task. Therefore, it is desirable to bring ray tracing onto the GPU to take advantage of the GPU's processing power. With the introduction of programmable

GPUs, it is now possible to utilize the GPU for non-traditional graphics tasks, with certain limitations.

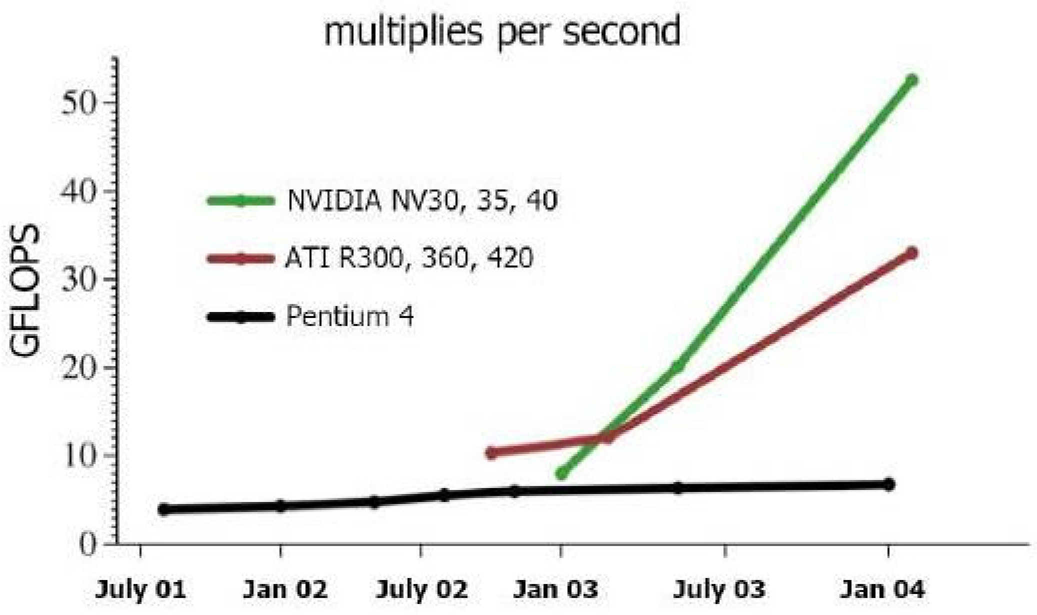


Figure 3 GPU Growth Rate [Buck 2004]

The GPU is designed to process batches of data at once, but it processes each individual data element with similar computations in parallel. This is also known as *stream processing*. This means the GPU can only process one "kernel" at a time but many instances of them in parallel. A kernel represents a set of operations that are identical across each individual data element. Basically, the GPU provides data parallelism, and is best suited for large data sets with minimal dependency between data elements that require the same computations with minimal memory access.

Ray tracing is highly parallel but requires frequent memory accesses. Triangles cannot be accessed from the traditional geometry pipeline in scan-line algorithms because each pixel needs to access multiple triangles. The triangles must be packed into textures and accessed in a random access fashion for GPU ray tracing. This goes against the design philosophy of the GPU because each pixel requires varying numbers of triangle accesses via textures. The algorithm cannot guarantee minimal memory access and the GPU texture caches might not be utilized effectively because triangles are accessed in a random fashion.

Another limitation is that the GPU cannot do complex logic control as well as the CPU. In fact, earlier GPU models could not do looping at all; they could perform a limited amount of looping by unrolling loops. Recent GPUs can perform loops a limited number of times but still not very efficiently. Unfortunately, ray tracing requires frequent looping control, therefore, hindering GPU performance.

The last general limitation is the speed of the traffic between the CPU and the GPU. The GPU cannot access CPU memory directly and vise versa. Therefore, we must pay the cost of sending data from CPU memory to GPU on-board memory, and the transfer rate can be limited by the bus technology on the motherboard. This can become a major bottleneck when the GPU is not receiving new data fast enough and spends some time idle. Despite these limitations, GPU ray tracing has been attempted in the past four years and is still viewed as a feasible route for performing ray tracing.

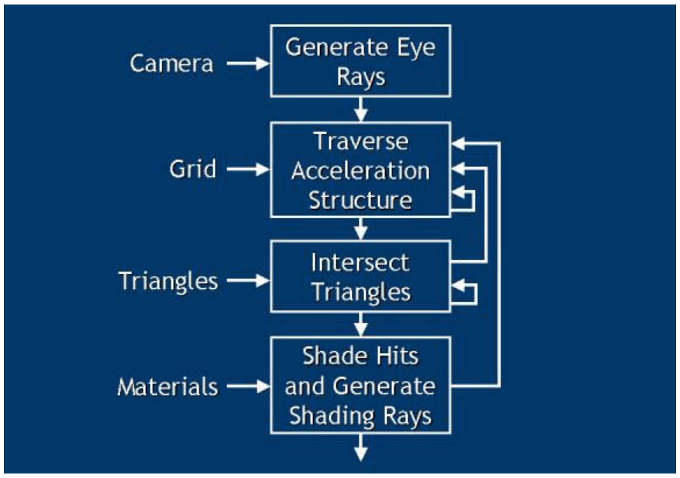
2.3 Related Work

In this section, we will describe the previous work done on ray tracing on both GPU and CPU platforms. Some work concentrated on improving ray tracing static scenes and some looked at dynamic scenes. The work related to rendering dynamic scenes usually focused on the building of the acceleration structures; they are the driving forces that directed us to concentrate our experiments on acceleration structures.

2.3.1 GPU-based Ray Tracing

GPU-based ray tracing started in 2002 with the Ray Engine [Carr et al. 2002] and Purcell et al.'s state-based GPU ray tracer [Purcell et al. 2002]. The Ray Engine had the GPU handle computationally intensive ray-triangle intersections and the CPU fed buckets of coherent rays and proximate geometry to the GPU. This division aimed to maximize the advantage of both processors, but was bottlenecked by the transfer speed over the bus.

This communication bottleneck can be avoided by directly implementing all the stages of ray tracing on the GPU. Tim Purcell at Stanford University decomposed ray tracing into four GPU kernels where each kernel is a fragment shading program that handles a different aspect of ray tracing: generating eye rays, traversal, intersection, and shading. He was able to achieve 114 million intersection tests per second with an ATI Radeon GPU, which outperformed the best CPU implementation at the time. Figure 4 shows the kernels used for his GPU Ray Tracer. His approach was innovative but still limited at the time because the GPU was not capable of doing true looping logic besides loop unrolling, and so could not fully utilize the GPU effectively.


Camera
Generate Eye Rays
Grid
Traverse Acceleration Structure
Triangles
Intersect Triangles
Materials
Shade Hits and Generate Shading Rays
Figure 4 Purcell's kernels for GPU ray tracing [Purcell et. al. 2002]

Nevertheless, Purcell's result was very promising and showed there is still a lot of room for improvement. This was in fact the case with the follow up implementation of two GPU-based ray tracers from two different Masters theses. Christen implemented a GPU ray tracer using both OpenGL and DirectX to demonstrate the implementation was feasible using different graphics APIs [Christen 2005]. Karlsson and Ljungstedt implemented a proximity-cloud uniform grid on the GPU and obtained a 37-50% speed up on some scenes [Karlsson and Ljungstedt 2004]. Both implementations used a uniform grid because it is the easiest data structure to implement on the GPU. Furthermore, Purcell suggested the uniform grid is probably the best acceleration structure on the GPU.

Researchers have also implemented other acceleration structure algorithms on the GPU such as the Kd-Tree and the BVH. Both the Kd-Tree and the BVH require stack operations on the CPU; however, it is not feasible to implement a stack on the GPU. Thus, GPU-friendly traversal algorithms should not rely on the stack. Foley and Sugerman implemented two stackless GPU Kd-Tree traversal algorithms: kd-restart and kd-backtrack [Foley and Sugerman 2005]. Knowing that the Kd-Tree had been shown to be the best overall acceleration structure for ray tracing static scenes at the time [Havran et al. 2000], an algorithm to allow the Kd-Tree to run on the GPU was unavoidable. While their work showed that hierarchy traversals other than a simple uniform grid were feasible, they did not achieve a performance comparable to an optimized Kd-Tree CPU implementation. Nevertheless, they demonstrated that the GPU Kd-Tree implementation outperforms the GPU uniform grid implementation on scenes with high variation in scene triangle density. Thrane and Simonsen did a performance comparison study of different GPU acceleration structures and implemented the BVH traversal algorithm on the GPU [Thrane and Simonsen 2005]. They concluded that the BVH was the best acceleration structure at the time on the GPU, and that the BVH could outperform other acceleration structures by a factor of nine in some cases. In 2006, Carr et al. implemented a hybrid approach using a BVH with geometry images on the GPU and demonstrated competitive performance against other acceleration structures on the GPU [Carr et al. 2006]. Furthermore, Carr's implementation could handle deforming models on the GPU.

We based our GPU ray tracer on the related work described above. Our GPU uniform grid implementation follows Purcell et al.'s paper [Purcell et al. 2002]. The Kd-Tree implementation is based on Foley's paper [Foley and Sugerman 2005]. Lastly, we implemented the GPU BVH traversal according to Thrane's paper [Thrane and Simonsen 2005] and we improved our existing GPU implementations with the provided shader code from Thrane's paper [Thrane and Simonsen 2005].

2.3.2 CPU-based Ray Tracing

Traditional ray tracing can only perform one ray-triangle intersection test, and traverse a single acceleration structure node, at a time. With the introduction of packet traversal and intersection test by Wald in 2004 [Wald 2004], we could traverse several rays in parallel on the CPU with SIMD instructions. Unlike the GPU, where the usage of complex logic and data structures is limited, the CPU does not have these limitations, but offers less-powerful parallel floating-point computation with SIMD instruction sets. Wald's implementation achieved 92-100 million intersection tests per second with packet ray-triangle intersection tests on the CPU. His system can render a static scene with 43 thousand triangles at four frames per second, and two frames per second for a dynamic scene of the same model. Wald continues to work on better ray tracing systems using both single CPUs and clusters of CPUs. In 2006, Wald published a coherent grid traversal method which was able to achieve 29 frames per second with pure ray casting and seven frames per second with full ray tracing effects on an 11,000-triangle, animated scene at a 1024^2 screen resolution [Wald et al. 2006]. His approach allowed the uniform grid to achieve high rendering performance by traversing several rays in parallel into the cells with a frustum-packet traversal. Since the uniform grid can be rebuilt quickly for all types of models, Wald's method can adapt to any triangle movements, as well as abrupt changes in the number of triangles in the scene.

Wald and Havran looked at how the Surface-Area-Heuristic (SAH) Kd-Tree can be built faster on the CPU. They proposed a method to build the Kd-Tree in O(n log n) time and their method was faster than the usual O(n $\log^2$ n) or O(n^2) implementations [Wald and Havran 2006].

Besides speeding up the Kd-Tree, hybrid tree structures have also been investigated. Havran presented the H-trees, a combination of the spatial Kd-Tree with bounding volumes, in his paper [Havran et al. 2006] and showed that H-trees can be built 2.4 to 11.7 times faster than Kd-Trees, and can perform as well as the Kd-Tree in terms of traversal and intersection testing.

Although a carefully optimized Kd-Tree is the best acceleration structure for static scenes, it is not the best acceleration structure for dynamic scenes because it cannot be updated or rebuilt fast enough to maintain adequate performance. On the other hand, the BVH has been shown to be more adaptable to dynamic scenes with deforming models. Lauterbach et al. proposed a simple BVH update algorithm that modifies the bounding volume as the triangles move in dynamic scenes [Lauterbach et al. 2006]. His method will gradually degrade the performance of the BVH, and he detects the degradation and rebuilds the BVH at that point. His ray tracing system can render a 40k-triangle dynamic scene at 12 frames per second at a 512^2 screen resolution. Wald also proposed a BVH implementation using a variant of SAH to render deformable models and achieved 8.5 frames per second for a 78k-triangle dynamic scene at a 1024^2 screen resolution [Wald et al. 2006a].

We were not able to implement all the latest work on acceleration structures because many of them are so new; however, we implemented the BVH update algorithm based on Lauterbach et al.'s paper because it is simple to understand. We do not follow his implementation completely and the differences are discussed in Section 3.3.2. We hope to incorporate more-recent improvements to acceleration structures in our future work.

3 Energy-Conscious Ray Tracing (ENCORE)

In this section, the high level implementation details of ENCORE will be described. ENCORE was developed with scalability and extensibility in mind, so it is generally not optimized for speed, and is a ray casting system. The three major components of ENCORE are the Scene Manager, the Accelerator and the Renderer, as shown in Figure 5.

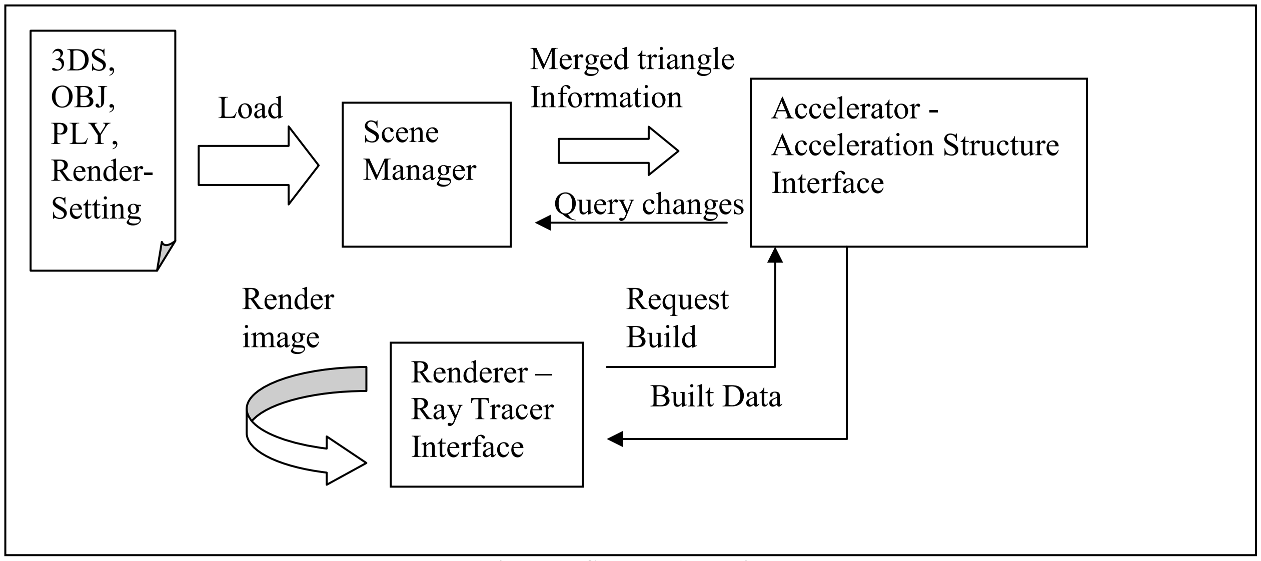

3DS, OBJ, PLY, Render-Setting
Load
Scene Manager
Merged triangle Information
Query changes
Accelerator - Acceleration Structure Interface
Render image
Request Build
Renderer – Ray Tracer Interface
Built Data

Figure 5 System Overview

The Scene Manager is responsible for loading 3DS (3D Studio Max file), PLY (a file format developed by Stanford University for their 3D scan repository), and OBJ files (a file format developed by Autodesk & Alias for Wavefront's Advanced Visualizer application) specified in description file (in house format). It stores the geometry data and creates a single list containing all the triangles in the scene.

The Accelerator is the interface for acceleration structures. Its main job is to provide and call a virtual build function for all acceleration structures implemented in the system. This allows new acceleration structures to be added in the future without changing the main system code. ENCORE currently supports three acceleration structures: uniform grid, Kd-Tree and BVH. Each acceleration structure queries the scene for changes before rebuilding. If there are no changes in the scene, the acceleration structure does nothing; otherwise, it requests a new list of triangles from the scene manager and rebuilds. In addition, all acceleration structures can be converted into textures which can be used to render on the GPU.

The Renderer is the interface for the rendering algorithm. Its main job is to ensure that every renderer implemented in the system has a render function. It also passes the triangle list and the acceleration structure into the renderer using an init function. The interface and virtual function declarations can be found in Appendix B.

In the following section, the uniform grid, Kd-Tree and BVH implementations will be explained in detail. The transition from CPU ray tracing to GPU ray tracing will be explained along with the uniform grid. Since the process is similar for the Kd-Tree and the BVH, only the uniform grid section contains an explanation of GPU ray tracing. To aid the discussion, Table 1 describes the notation used in the descriptions.

Table 1 Short hand notations for ENCORE Implementation

Short-hand Notation	Description
#T	Number of triangles in the scene
AABB	Axis-aligned bounding box
Voxel	Individual uniform cell in the uniform grid
Model	The geometry that makes up the scene. Scene and model mean the same in the context of this discussion
BVH	Bounding Volume Hierarchy
Texture	2D image to map onto 3D geometry
Grid	Short hand for uniform grid
Subscript s	Scene bounding box
Subscript t	Triangle bounding box

3.1 Uniform Grid

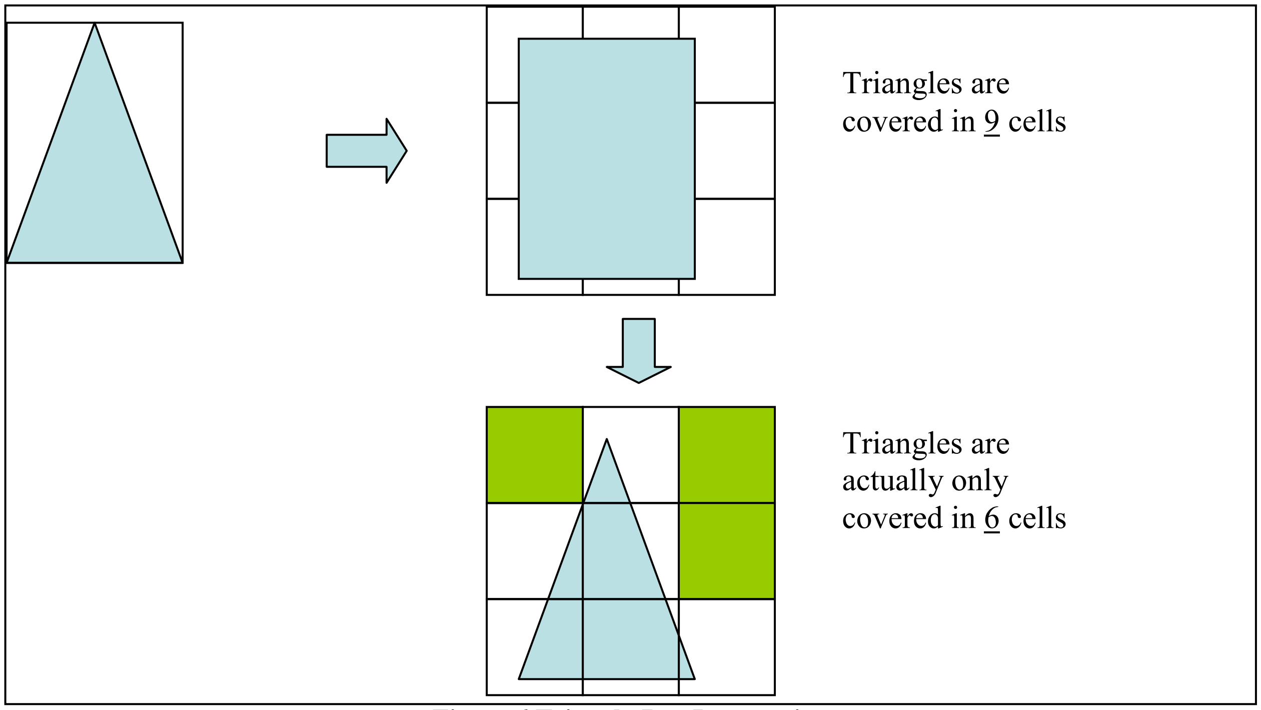
3.1.1 Build

Partitioning the space into uniformly distributed cells is the main idea behind a uniform grid. The cells in the grid can be uniform in size, same length in x, y, and z axes, or uniform in number where the number of cells along the x, y, and z axes are the same. The former creates uniform-sized cells but uneven cell numbers along each dimension. The latter creates non-uniform length across the different axes, but the cell length along a single dimension is uniform. The ENCORE implementation uses the later approach. There are numerous ways to determine the grid division in the x, y, and z directions, and the ENCORE implementation uses the cube root of #T to determine the number of grid divisions. Given a scene with 7,532 triangles, for example, $\sqrt[3]{7{,}532} = 19.6 \approx 20$ segments, assuming the size of bounding box enclosing the scene is 100x80x120. This results in a 20x20x20 uniform grid with a cell size of 5x4x6.

The next step is to insert triangle references into the cells containing the triangles. The bounding regions between triangles and cells can be calculated using algebra. Continuing from the above example, the scene bounding box has a minimum point at (0,0,0) and maximum point at (100,80,120). Given a triangle with a bounding box starting at (13,55,30) and end at (24,60,50), the cell indices that overlap this bounding box can be calculated with the following equations for each dimension.

Starting x cell index = (x-min$_t$ – x-min$_s$) / cell size in x
Ending x cell index = (x-max$_t$ – x-min$_s$) / cell size in x

The numbers are rounded down fractional results. The y and z values can be found by replacing the x with y or z in the above calculations. This example would yield an x index at (2,4), y index at (13,15), and z index at (5,8). This method is simple and fast but not entirely accurate. Figure 6 illustrates the reason.



Triangles are covered in 9 cells
Triangles are actually only covered in 6 cells

Figure 6 Triangle-Box Intersection

Figure 6 shows that a triangle bounding box can overlap cells not covered by the triangle. This introduces cells with false triangle references, leading to unnecessary ray-triangle intersection tests when rendering the uniform grid. The exact triangle coverage in each cell can be found by performing a triangle-box intersection test outlined by Akenine-Moller [Akenine-Moller 2001]. Doing the triangle-box intersection test not only produces a more-physically accurate allocation of the triangles in the grid, it also creates a more-compacted grid that uses less memory. We started the initial uniform grid implementation using code from Bikker [Bikker 2005] and modified the implementation. The pseudo-code for the uniform grid build follows.

```
1. let bbox = AABB enclosing the scene
2. Divide bbox into M x M x M cells
3. for every triangle
4.     count triangles AABB –  bbox overlap
5. allocate memory on the number computed in step 4
6. For every triangle
7.    Find triangle AABB – bbox overlap
8.        For every voxel
9.            If  triangle-box overlap
10.               Insert triangle reference in the voxel
```

This algorithm still functions if Steps 3 to 5 are removed. Steps 3 to 5 introduce redundant calculations that calculate the bounding region between triangles and cells because the bounding region is calculated again in Step 7. The redundant step computes the maximum memory needed to store all the triangles in the grid. Doing so avoids the usage of dynamic data structures such as C++ standard template library vector, list, or queue during Step 10. This algorithm builds faster and produces consistent build times compared to the algorithm using dynamic data structures. This idea is described by Haines [Haines 1999]. Removing Step 9 increases the build speed by roughly 250% and we name this algorithm the non-triangle-box intersection build. The energy consumption ratio between this coarser build and an accurate build (the uniform grid implementation with the triangle-box intersection) allows us to compare the benefits of triangle-box intersection against the impact on rendering time later on.

3.1.2 Traversal

The goal of an acceleration structure is to reduce the ray-triangle intersection tests by avoiding them if possible. An acceleration structure does so by replacing the ray-triangle intersection with the acceleration structure traversal; therefore, the traversal computation needs to be much cheaper than the ray-triangle intersection to speed up ray tracing. We implemented a fast voxel-traversal algorithm outlined by Amanatides and Woo [Amanatides and Woo 1987]. Figure 6 illustrates the algorithm in 2D.

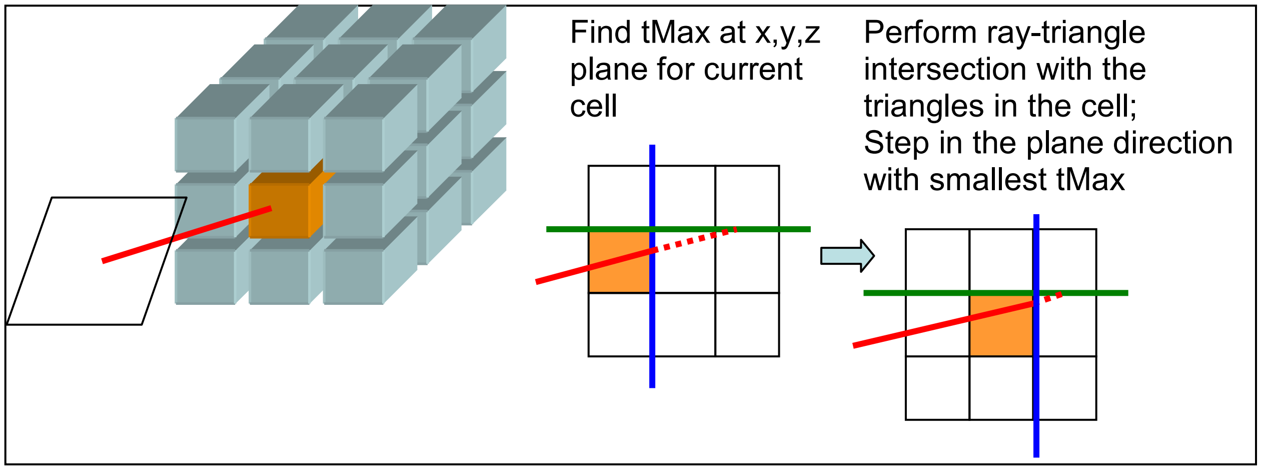

Find tMax at x,y,z plane for current cell
Perform ray-triangle intersection with the triangles in the cell; Step in the plane direction with smallest tMax

Figure 7 Uniform Grid Traversal Illustration

A ray enters the middle cell of a uniform grid in Figure 7. Assuming no triangles are in the first cell, the algorithm calculates the maximum hit time (tMax) for the ray to hit the boundary of the cell in x and y axes. The smallest tMax is used to determine the cell that the ray should traverse next. In Figure 7, the smallest tMax lies on the x axis so the ray steps in the x direction. If there are triangles in the second cell, the algorithm will perform ray-triangle intersection tests on all the triangles in the cell. If a valid triangle hit is found and the hit time is smaller than the tMax of current cell's boundary, the algorithm returns with the hit information. Otherwise, the algorithm continues the traversal. This algorithm costs six additions and three multiplications which is much cheaper than the cost of a ray-triangle intersection test. The ray-triangle intersection test is implemented using Moller and Trumbore. [Moller and Trumbore 1997]. An optimized version of the ray-triangle algorithm can be found in Wald [Wald 2004], however, his approach requires additional pre-computations for each triangle so we chose not to implement it. Overall, we found the uniform grid simple to implement and understand; it is an ideal example for a beginner to learn acceleration structures.

3.1.3 Moving to GPU

Unlike CPUs, GPUs do not have data structures such as arrays, lists, and stacks. Access to GPU memory is limited, so only viable option for inputting non-vertex information into the GPU is via textures, because textures can be used as random access memories in the GPU. Each individual pixel in a texture can be read in random order in GPU shaders, and this enables a texture to act as a random access memory. The conventional method of inputting triangle vertices, normals, and texels into the GPU is not suitable for GPU ray tracing because we need to access triangles in random order. Therefore, we store the triangles into textures. Figure 8 illustrates the conversion of CPU data into GPU textures. We break a vertex array into three separate vertex textures, holding the value of first, second and third vertices of the triangles, respectively. This allows a maximum of 16 million vertices in memory with a maximum texture size at 4096^2. The information in the uniform grid must also be translated into a texture. We are allowed to store a maximum of four values into a texel. Since we cannot use any dynamic data structures on the GPU, the data in each uniform grid cell needs to be represented in another fashion. The triangles referenced by the uniform grid are stored in the vertex texture in the order they appear in the uniform grid. We do not use a triangle index texture to reference repeated triangles; they are simply stored into the textures again. With the vertex texture set up in this fashion, the uniform grid texture can store the beginning vertex index in each cell and the number of triangles in each cell into the R and G components of the texels. We leave the B and the alpha components empty in the uniform grid texture, but they can be utilized in some way to maximize texture utilization in future work.

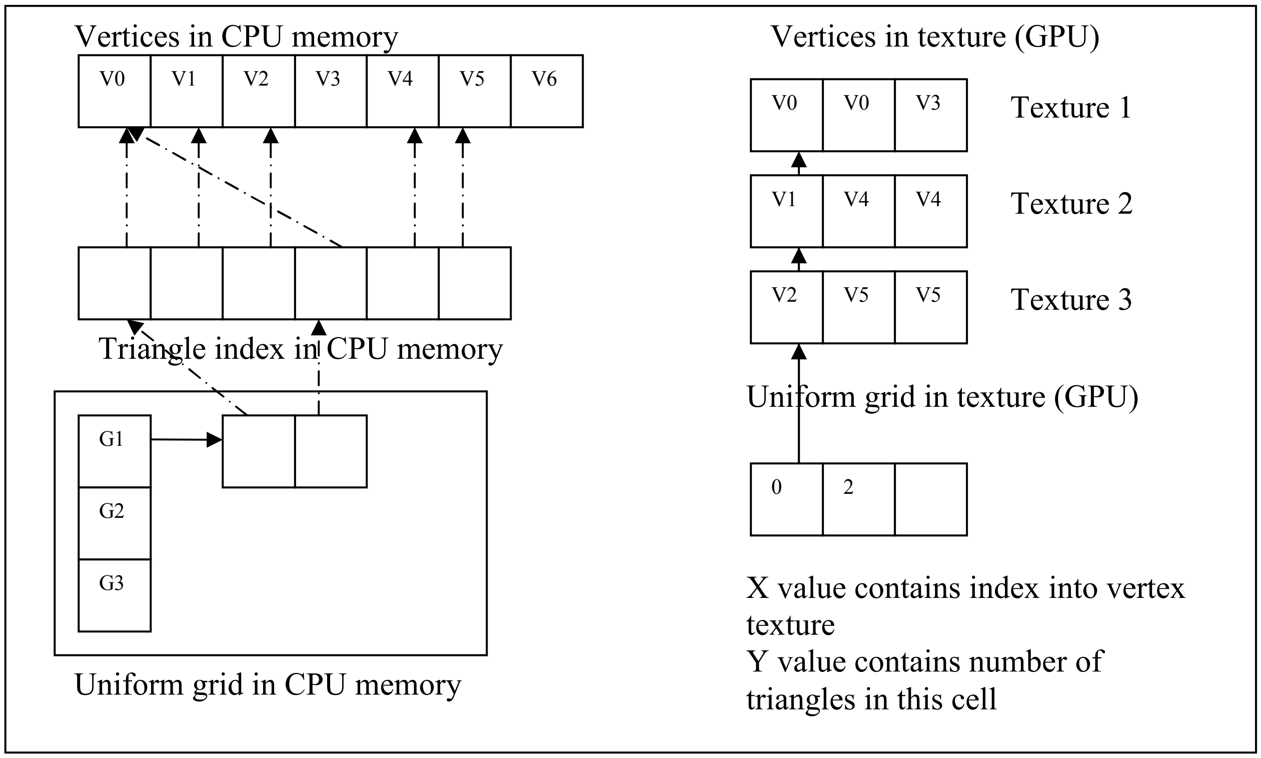

Vertices in CPU memory
V0
V1
V2
V3
V4
V5
V6
Triangle index in CPU memory
G1
G2
G3
Uniform grid in CPU memory
Vertices in texture (GPU)
V0
V0
V3
Texture 1
V1
V4
V4
Texture 2
V2
V5
V5
Texture 3
Uniform grid in texture (GPU)
0
2
X value contains index into vertex texture
Y value contains number of triangles in this cell

Figure 8 CPU Memory to GPU Texture

Purcell et al. use another level of indirection that requires another texture to represent the triangle index on CPU [Purcell et al. 2002]. This implementation produces smaller vertex textures at the cost of an additional texture access. We chose to repeat the triangle data in the vertex textures to avoid the additional texture access. With the textures set up properly, we could begin the execution of the ray-tracing shaders. A *shader* is the execution code for programmable GPUs. There are three main shaders in ENCORE: the Ray Generator, the Traversal-Intersection shader, and the Phong-Lighting shader. Figure 9 shows the execution flow of the shaders in ENCORE. Again, we follow the implementation described Purcell et al. [Purcell et al. 2002].

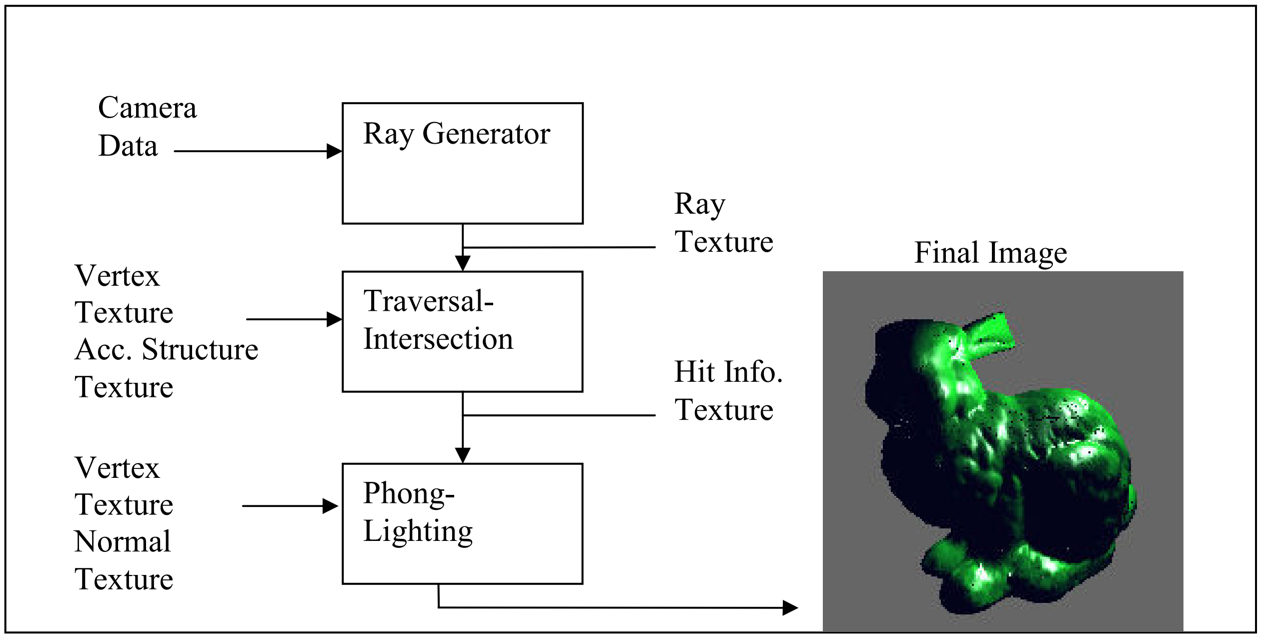


Figure 9 Kernel Diagram for ENCORE GPU Ray Tracer

The Ray Generator generates eye-ray textures where each ray shoots at a pixel location on the screen. The Traversal-Intersection shader computes the ray-triangle intersection with the same algorithm used on the CPU using textures as random access memories. Purcell separated the uniform grid traversal and the ray-triangle intersection into two different shaders because he needed to control the looping of shaders with the CPU. With shader model 3.0, programmable GPUs can perform up to 65,536 iterations in a nested for-loop. Thrane utilized this new feature to combine the traversal and the ray-triangle intersection shaders into one shader [Thrane and Simonsen 2005]. His approach eliminates the need to swap shader executions between the traversal and the ray-triangle intersection, thus improving performance. Ultimately, the Traversal-Intersection shader produces triangle-hit information at each pixel as a texture and passes the texture down the pipeline. The Phong-Lighting shader computes the color with the triangle-hit information texture and displays the image on the screen. The shader code for the Ray Generator, uniform grid traversal and Phong-lighting are provided in Appendix C. The shader code for the Kd-Tree and BVH are not included; see Foley and Sugerman [Foley and Sugerman 2005] and Thrane and Simonsen [Thrane and Simonsen 2005] for more detail.

3.2 Kd-Tree

3.2.1 Build

The Kd-Tree and uniform grid are both spatial subdivision algorithms. A uniform grid organizes the space into uniformly distributed cells. A Kd-Tree takes a non-uniform approach and organizes the space into a binary tree (Figure 10). The ENCORE Kd-Tree implementation is based on Pharr and Humphries [Pharr and Humphries 2004, page 198]. The algorithm builds the tree in O(n $\log^2$ n) time. We changed some parameter values in the algorithm to speed up the rendering of our test scenes, but their custom memory allocation method is not implemented.

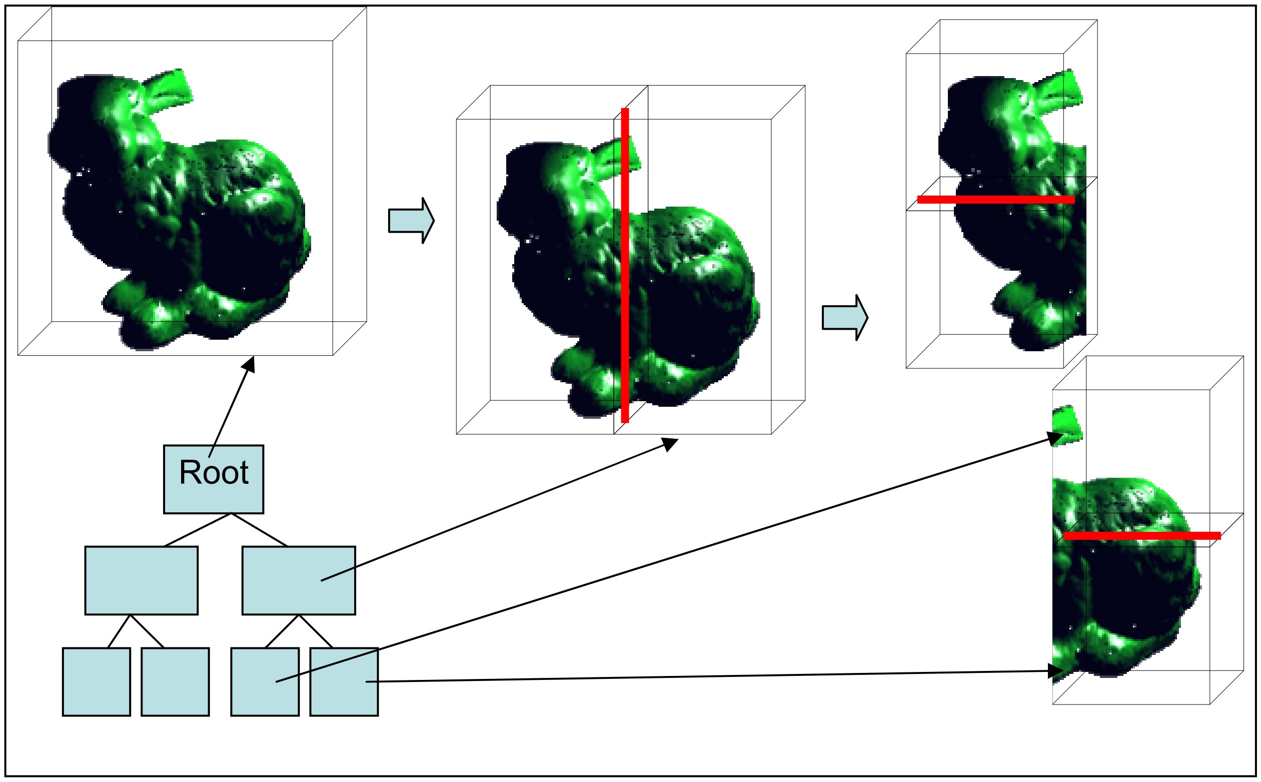


Figure 10 Kd-Tree

The Kd-Tree stores the split locations in interior nodes and lists of triangles in leaf nodes. Figure 10 is misleading because the root node actually stores the split location indicated in the second box pointed to by the second arrow. However, the arrows in Figure 10 represent the space corresponding to the nodes and not the data stored in the nodes. The decision of where to split the space can vastly change the topology of the tree, as well as its ray-tracing performance. Thus, it is critical to employ a good splitting criterion. The ENCORE Kd-Tree is implemented using two methods: a Surface-Area Heuristic (SAH) and Spatial Median Split (SMS). SMS is a very simple approach that always splits the axis in half on the current bounding volume and the splitting axis is chosen in round-robin fashion. This method can build the tree 2-4 times faster than the SAH approach, however, it does not always produce a balanced tree.

The SAH uses the area ratio of parent and child nodes to find the best possible splitting location. Our SAH implementation is outlined in Pharr and Humphries [Pharr and Humphries 2004, page 206] and we will not go into the implementation detail for the SAH. The SAH Kd-Tree usually produces a fairly balanced tree that speeds up ray tracing performance, however, it is slower to build.

Besides the split location, it is also important to specify the termination criteria for a build; otherwise, the build algorithm can go on splitting the space forever. Typical termination criteria limit the depth of the tree and the maximum number of triangles in the leaf nodes. Pharr and Humphries set the maximum depth of the tree equal to 8 + 1.3 * log(#T) and the maximum number of triangles to 16 for leaf nodes [Pharr and Humphries 2004, page 213]. They allow three retries when a better splitting location is not found by the SAH, and then create a leaf node, ignoring the triangle count. We use 11 + 1.3 * log(#T) for the maximum tree depth, 10 for the maximum triangle size in leaf node, and five for the number of retries. If the scene has less than 5,000 triangles, the maximum triangle size for the leaf node is set to two. The pseudo-code for the ENCORE Kd-Tree follows. The 'left' and 'right' variables are global arrays allocated to have size #T in the scene before the build algorithm starts. Build is a recursive function.

```
Build(id, depth, numRetry, triangle_count, prev_triangle_count )
1.  If(prev_triangle_count – triangle_count <= 3 )
        numRetry++
2.  If ( depth >= maxDepth or numRetry >= 5 or triangle_count <= 10 )
3.      Create a leaf node, return
4.  determine the split axis and find the split location
5.  If( id equals 0 ) // indicates this is the root node
6.      for each triangle in the scene // all the triangles in the scene
```

```
7.          if ( the triangle intersects with the left cell )
8.              insert the triangle into left          // replace the old value
9.          if ( the triangle intersects with the right cell )
10.             push the triangle into right          // append to the old values
11. If( id equals left ) // indicate this is for left node
12.    for each triangle in left array // only the triangle in left array
13.        do step 5-9
14. If( id equals right ) // indicate this is for right node
15.     for triangle_count triangles on top of right array   // only the triangle for the current node
16.         do step 5-9
17  If ( tree-node array is too small )
18.    allocate new array with size = 2 * tree-node array size
19.    copy the old array into new array and deallocate old array
20. create parent node
21. build(left, depth+1, numRetry, number of triangles inserted in left, triangle_count)
22. build(right, depth+1, numRetry, number of triangles inserted in right, triangle_count)
```

The above implementation uses two global arrays to store the triangles. However, a simpler implementation can eliminate Steps 11 to 16 and create left and right arrays locally using dynamic data structures. The simpler implementation would also need to pass the local array in Step 21 and 22. Allocating dynamic data structures slows down the algorithm because new memory allocations are needed in each recursive build function call. Each leaf node requires a dynamic allocation to hold the triangle references as well. The use of global left and right arrays in Step 5 to 16 eliminates the need to allocate more memory to hold the triangle references. It does not eliminate the need to allocate memory for the leaf nodes. The use of a global array to avoid memory allocation is not part of Pharr and Humphries [Pharr and Humphries 2004] and we have yet to read any literature using this technique. A cleaner approach would be the use of a custom memory pool.

Since the build function always builds left first (Step 21), the left array can be reused on every recursive call because the triangles in the left array are guaranteed to be redistributed by Step 11 to 13. The right array is treated like a stack that contains batches of triangles. The top batch of triangles contains the triangles used by the first build(right) function call. Step 15 shows that the algorithm can only use the number of triangles intended for the working node at the time. The right array needs to allocate more memory when the array is full because we are adding the triangle references to the array on every recursive call. It is not reflected in the pseudo-code, but we use the C++ standard library vector for the right array and we use the reserve function to allocate the desired memory. Figure 11 illustrates the use of global arrays for the ENCORE Kd-Tree build.

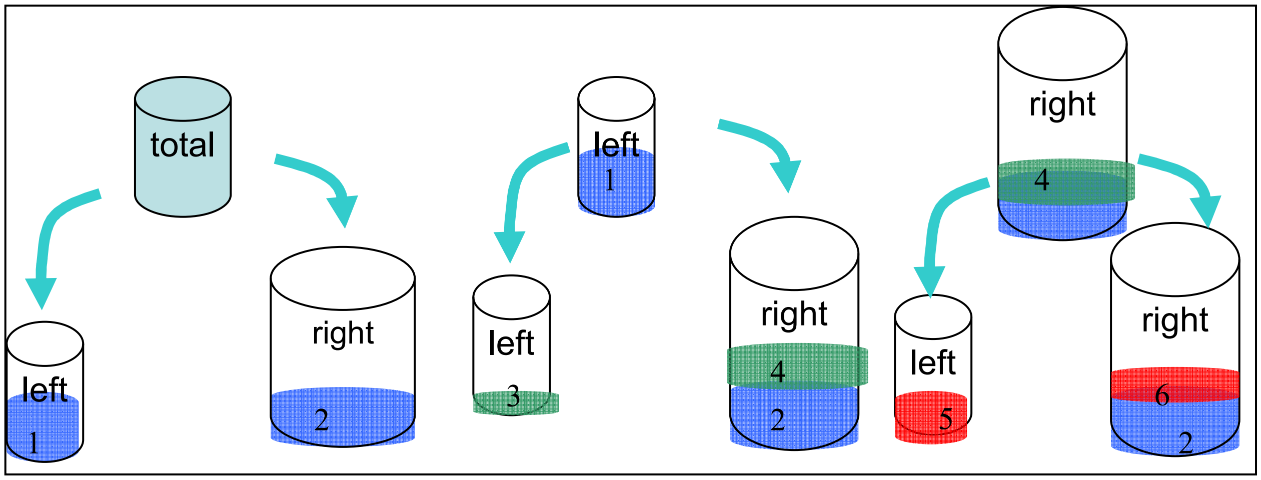


Figure 11 Global left/right arrays for Kd-Tree Build

In Figure 11, the triangles are first split into two sets, 1 and 2. The build(left) function-call at Step 21 is executed and the left array is used, therefore, set 1 is separated into sets 3 and 4. Set 3 replaces the original content in the left array, but set 4 is appended on top of the right array. Assuming the next build(left) function-call produces a leaf node, the algorithm reaches Step 22, and build(right) is executed. Only the set 4 data in the right array are used in Step 15 and the new set, 6, is appended on top of the right array again. The build algorithm is 2 to 3 times faster than the original implementation that used the C++ standard library list to store the triangles. We expect a greater speed up can be achieved by eliminating the memory allocation in the creation of the leaf nodes.

The most complex part of the Kd-Tree algorithm is the SAH implementation. Since it is covered in Pharr and Humphries [Pharr and Humphries 2004] and Wald also goes into extensive length in describing how to build a good SAH Kd-Tree [Wald 2004], the SAH implementation is not described in this thesis.

3.2.2 Traversal

The Kd-Tree traversal is much cheaper than the uniform grid traversal. It requires only one subtraction and one multiplication for each traversal operation. The first step in the traversal process is determining if the ray hits the bounding box of the scene (Figure 12). The traversal algorithm is implemented using Pharr and Humphries [Pharr and Humphries 2004, page 215].

The ray can return immediately if the ray misses the scene completely. The smallest hit time when the ray entered the scene AABB is recorded in the variable tMin. tMax stores the smallest hit time when the ray exited the scene AABB. The algorithm starts at the root node and the time for the ray to hit the split axis (tPlane) is calculated. Furthermore, the ray direction is used to determine the order of traversal. A positive ray direction means the ray must visit the left node first then the right node. A negative ray direction means right then left. The algorithm can only traverse one node at a time. If the ray visited both nodes, the farther node is pushed onto the stack and the closer node is traversed. The traversal step continues until a leaf node is found and the algorithm performs ray-triangle intersection tests on all triangles in the node. The algorithm pops a node off the stack after the intersection tests and continues on.

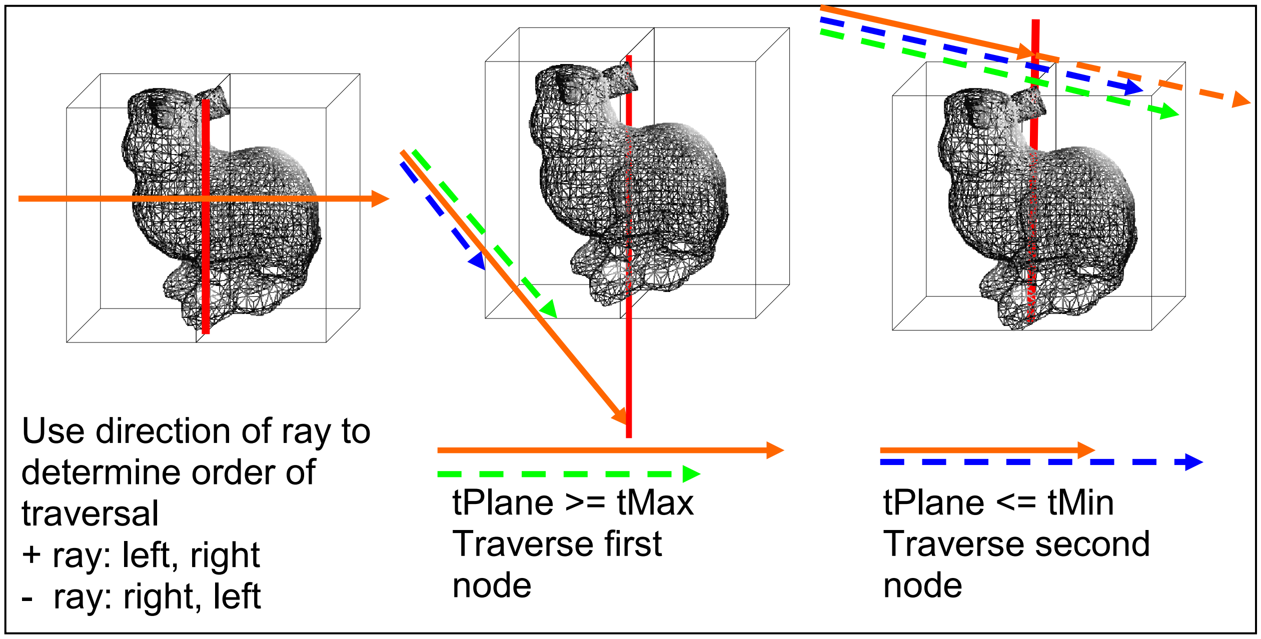


Figure 12 Kd-Tree Traversal

Figure 12 shows that tPlane can be used to determine the next node for traversal. Similar to the uniform grid, the Kd-Tree is traversed in front-to-back order, and the triangle hit time can be returned when the first valid hit is found in a leaf node.

The Kd-Tree is stored into the texture in a similar fashion as the uniform grid. An interior node is stored with (left child index, split position, none, split axis/leaf node indicator). A leaf node uses (start index, none, none, triangle count). The traversal on the GPU is similar to the CPU implementation. The ray traverses down the nodes until a leaf node is found. If the ray misses all triangles in one leaf node, the tMin and tMax of the ray are moved forward and the ray restarts from the root node again. Since the tMin value is moved forward, the ray takes a different path down the tree and ends up in the next leaf node.

3.3 Bounding Volume Hierarchy (BVH)

3.3.1 Build

Unlike the previous two acceleration structures, the BVH is a geometry-partition algorithm. The algorithm partitions the geometry and not the space around the geometry (Figure 13). The BVH is also a binary tree structure like the Kd-Tree, but stores the bounding volume enclosing the triangles in the scene. The bounding volumes are collapsed or expanded to exactly enclose the triangles in the target area, so the tree will never have a node containing no triangles. Each triangle is represented only once in the BVH, because every split operation divides the geometry.

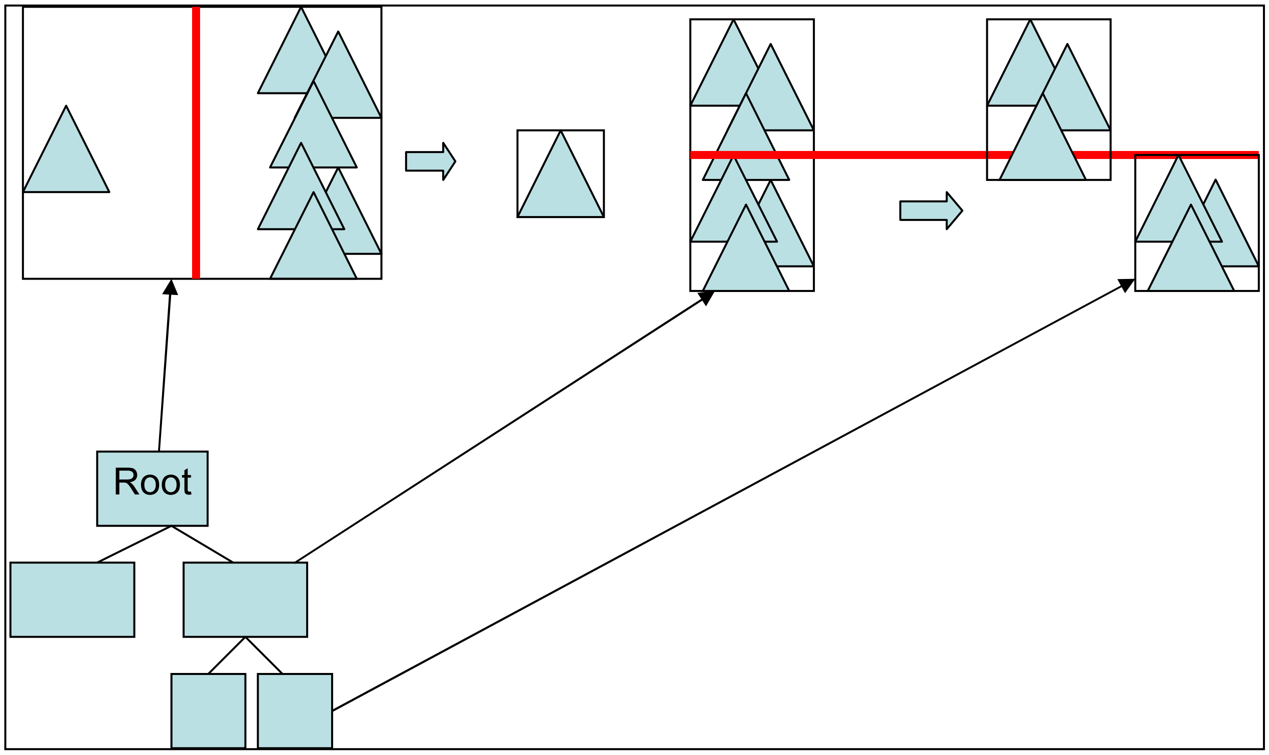


Figure 13 BVH

Figure 13 shows the bounding boxes are resized to fit the triangles, even if the split position leaves some space. The picture also shows that a triangle is not represented in two nodes if the split location lies in the middle of the triangle. The CPU implementation of the BVH is based on the description provided by Lauterbach et al. [Lauterbach et al. 2006]. They uses SMS as the splitting criteria because it is the simplest and fastest approach. They continue to split the tree until there is only one triangle in all leaf nodes, resulting in a tree with 2*#T – 1 maximum nodes. The tree-node array can be pre-allocated based on this computation. and there is no dynamic allocation needed for each leaf node because it will always contain exactly one triangle. This approach is fast to build and can perform as well as a SAH Kd-Tree. We did not modify any parameters for the BVH. The pseudo-code for the ENCORE BVH follows.

```
build(id, triangle_count)
1.  if( triangle_count equals 2 )
2.     make 2 leaf nodes // left and right child node of current node
3.      return
4.  if( triangle_count equals 1 )
5.      make current node leaf , return
6.  choose axis in round-robin fashion and find spatial median as the split location
7.  if( triangle bounding box min. point less than the splitting location )
8.      insert in the left array
9.  else
10.     insert in the right array
11.  if ( left or right is empty )  // mean the split can't produce two child at this location
12.     try other two axes
13.  if( left or right is still empty ) // not possible to split them, so force it into two halves
14.     insert half of the triangles in the left array
15.     insert other half in the right array
16. build(left)
17. build(right)
```

We used the same memory preallocation technique for the BVH, but it is not shown in the pseudo-code. Step 13 divides the triangles in the current node in half because the spatial median split point cannot guarantee the left and right have an equal number of triangles. If the algorithm used the geometry median, where the split location is the median of the triangles in the current bounding volume, Step 13 to 15 can be avoid.

3.3.2 Update

Since every leaf node contains only one triangle, we can update the BVH without rebuilding it from scratch when the triangle locations change (Figure 14). We can loop through all leaf nodes and check the stored AABB against the corresponding triangle’s AABB when rendering a dynamic scene. If the AABBs are not the same, the AABB in the leaf node is updated as well as all its parent nodes. This is the update algorithm proposed by Lauterbach et al. [Lauterbach et al. 2006].

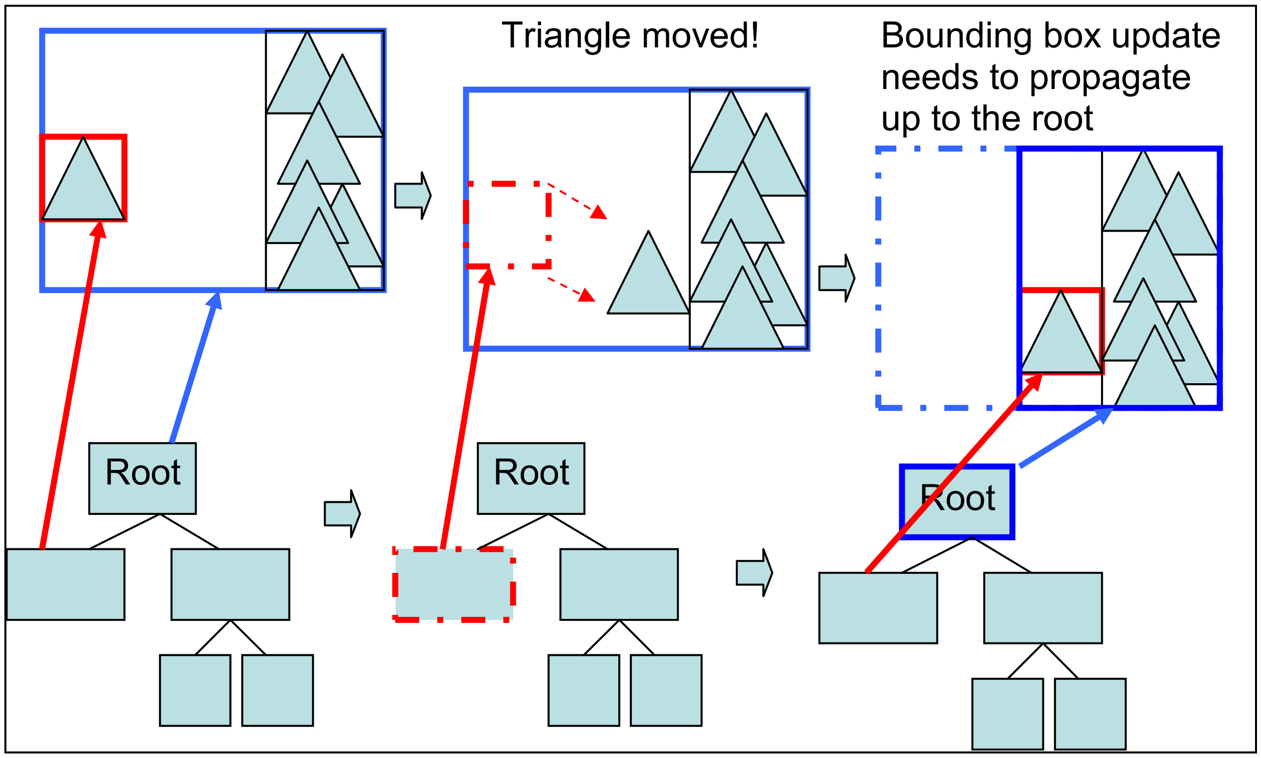


Figure 14 BVH Update Method

This method is surprisingly simple yet effective. The only problem is that the rendering performance will be degraded if the new triangle location does not fit well with the existing tree topology. Thus, the BVH needs to be rebuilt when the triangle movement passes beyond some threshold. We have yet to implement the automatically rebuilt mechanism described by Lauterbach et al., and it is left as future work. The update method only works on deforming models, animated models that do not increase in triangle count, and per-frame rebuild is needed if the testing scene contains non-deforming models.

3.3.3 Traversal

The BVH traversal algorithm is similar to the Kd-Tree, however, it uses two ray-AABB intersections per node to determine the path to walk down the tree. Since the nodes are not guarantee to be stored in front-to-back order, early termination is not possible without additional calculations to insure traversal in front-to-back order. Lauterbach et al. [2006] described a method to determine 'near' and 'far' child nodes by storing the maximum distance between the child nodes. We did not follow the approach because we did not fully understand the algorithm at the time, and instead use the hit time information returned from ray-AABB intersection to determine if the node should be traversed further. When a ray-triangle hit time is found after examining a leaf node, the value is stored in the variable bestHit. If at tMin, the time when the ray entered the AABB, is greater than bestHit, we can skip that parent node and its children completely because all hit times found within that path are behind the bestHit. This method will not speed up the BVH traversal if the first bestHit found happens to be the farthest triangle in the scene. The BVH rendered much faster with this approach for all of our test scenes, so we did not search for better BVH early termination techniques. Our approach is based on our observations of the behavior of BVH traversal and was not found from any literature. However, we believe this approach must already have been used in the past. The BVH GPU traversal did not implement this early termination check.

4 Test Environment

The purpose of this thesis is to identify the major elements that stress the battery during ray tracing. Therefore, the battery discharge rates are measured for a wide range of scenes with different acceleration structures at different screen resolutions. In this section, we describe the hardware environment, machine specifications, software settings, and operating system environment for the tests. The test scenes and the methods used to measure power and time are presented here as well.

4.1 Hardware Settings

The test machine was a Dell Inspiron 9300 laptop. It had a 1.6GHz Intel® Pentium® M processor and 1.25GB of RAM. It was equipped with a PCIe x16 nVidia GeForce Go 6800 graphics card with 256 MB of video memory. The stock battery was a Dell rechargeable Li-ion Type D5318, Rating 11.1V, 4800mAh, with 53WH capacity. For all the tests, the laptop monitor was set to have 50% brightness to reduce the energy used by the monitor. Doing so allowed more tests to run to completion before the battery ran down. In addition, on-board network devices were disabled to reduce energy consumption and unstable battery discharge rates. The battery was recharged back to 98% or more after each batch of tests (see below).

4.2 Software Settings

The operating system on the machine was Windows XP Professional Version 2002 Service Pack 2. The graphics driver is nVidia ForceWare version 83.60. The ENCORE executable was built using Microsoft Visual Studio 2003 in release mode. The tests were run using Windows batch files, where each batch file contained six to nine tests. Several versions of ENCORE executables were built to allow easier batch file control by changing the executable names in the batch file.

Each batch file contained a list of statements in of the form: <ENCORE executable> <description file>. The description file specified the setting for the ENCORE executable. It set the render screen size, the acceleration structure, the renderer (CPU, GPU, or OpenGL), the scene files, the maximum running time of the application, the information to print, and the power measurement setting. We will not go into detail on the content of the description file.

Each test was set to run for three minutes and 10 seconds (see below). A single batch file executed for 20-30 minutes, leaving the battery life at around 40-60% when the execution ended.

At this point, the battery was recharged back to 98% or more before running the next batch file. We did not wait until the battery was recharged back to 100% because we believe 98% or more is good enough to be considered fully charged. In addition, we did not run tests until the battery capacity dropped to 0% because we found the battery tended to discharge faster when it had a low capacity. The tests conducted with low battery life always had higher discharge rates even for identical tests. We worked around this issue by recharging the battery back to full before the battery life drops below 40%. If the battery life was below 40% at the end of a batch file execution, the last three test data were discarded. The battery was recharged and the last three tests in the batch file were tested again.

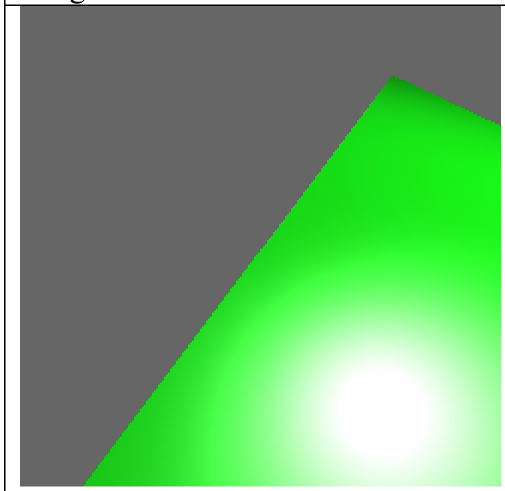
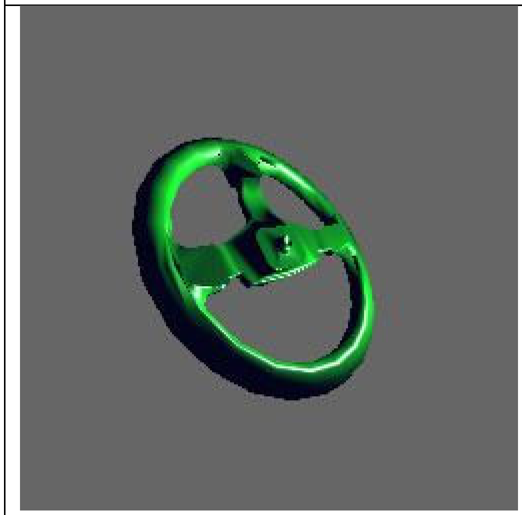
We ran a total of 225 tests and no test was repeated, unless the data seemed unexpected and we used the second run to double check the data. The data for the repeated run was not saved; therefore, we do not have variances for each individual test. We do have the variances of the discharge rates during the execution of each test and these data are in given Appendix A.

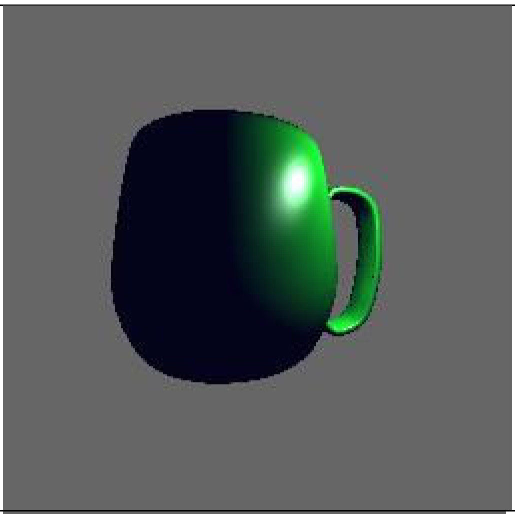
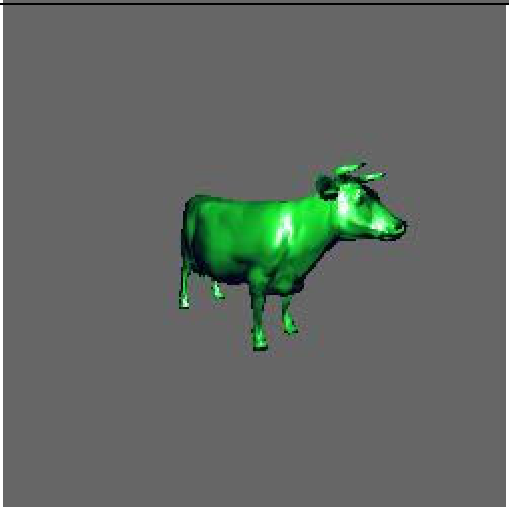
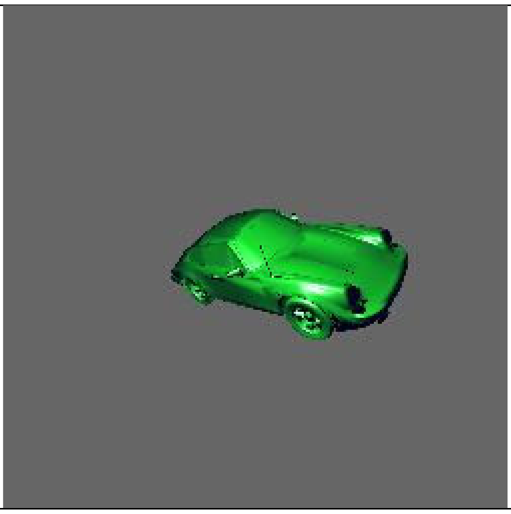
The machine was left untouched during the duration of the test. The monitor auto shutdown option was off and the screen saver was disabled. There was no keyboard or mouse inputs either. All user-mode background software, such as anti-virus and firewall software, were shutdown before the test began, and the Windows auto-update option is disabled. We did not tamper with the system processes, and cannot guarantee that no other system tasks were scheduled by the operating system during the duration of the test. However, we tried to insure that all tests ran under the same software conditions.

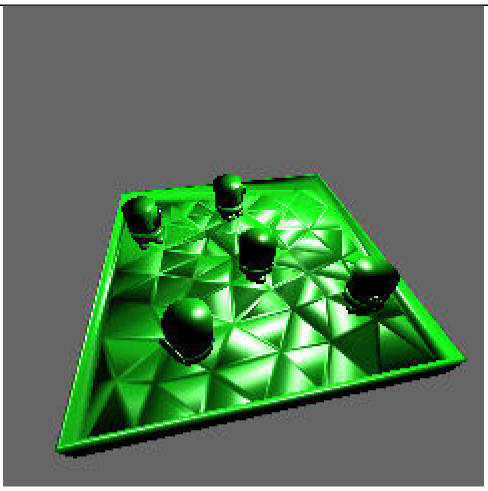
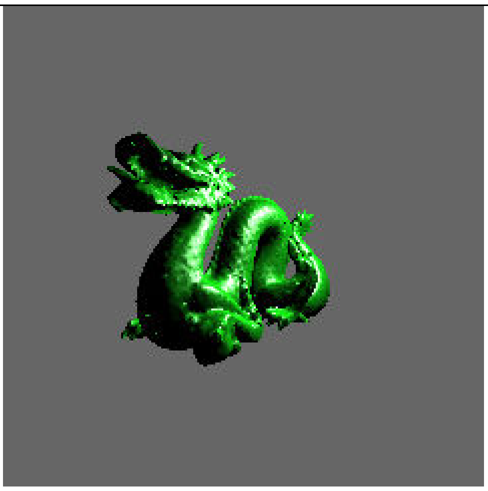
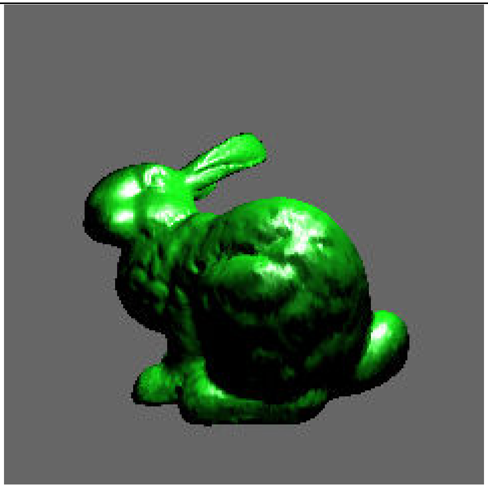
4.3 Test Scenes

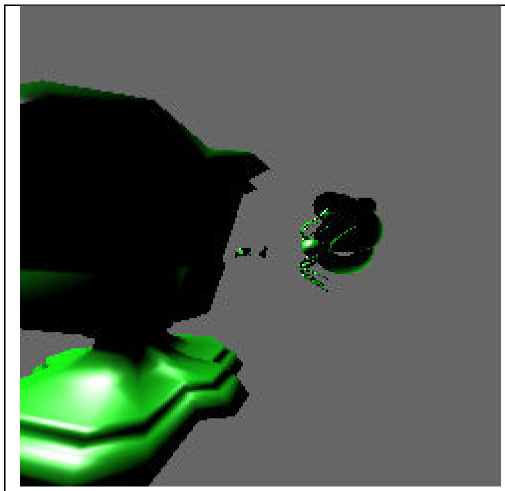
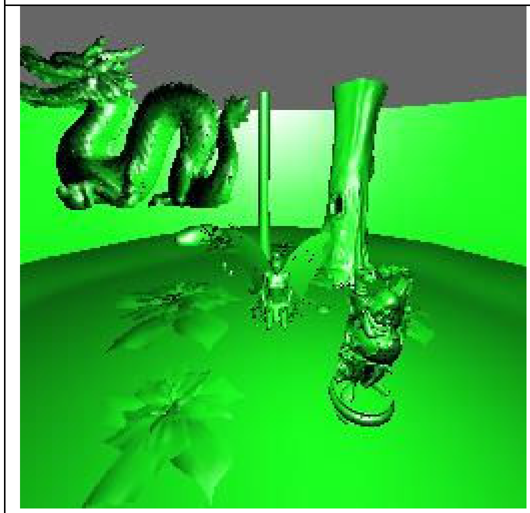
There are 14 scenes that were used for our experiments. Table 2 shows the image, the name, the number of the triangles, the model file names, the file source, and the test conducted for all the scenes. Most models are publicly available by going to the source Website. A blank source means the source is unknown but we can make the model available if needed.

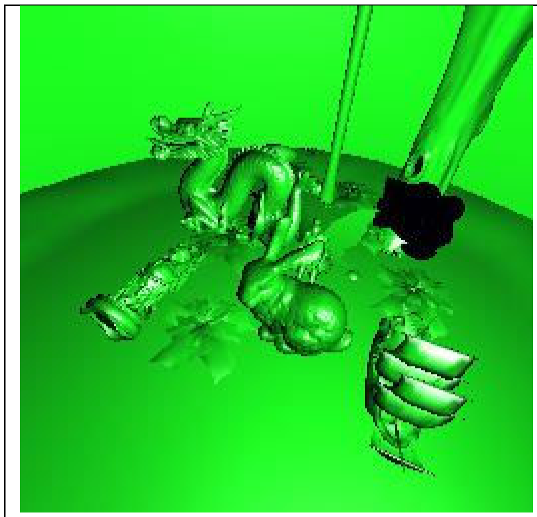
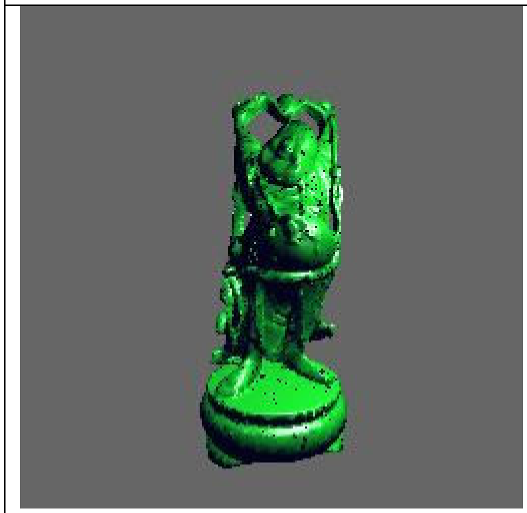
Table 2 Test Scenes

Image	Detail	
	Name	SingleTri
	Model File Name	SingleTri.ply
	Source	A (-1,-1,1), (1,1,-1), and (1,-1,1) triangle made by us
	Number of Triangles	1
	Used in (section)	5.1.1, 5.1.4, 5.1.5
	Name	Scissors
	Model File Name	Scissors.ply
	Source	
	Number of Triangles	604
	Used in (section)	5.1.2
	Name	Wheel
	Model File Name	Steeringweel.ply
	Source	
	Number of Triangles	1,368
	Used in (section)	5.1.2

	Name	Mug
	Model File Name	Mug.ply
	Source	
	Number of Triangles	3,450
	Used in (section)	5.1.2
	Name	Cow
	Model File Name	Cow.ply
	Source	
	Number of Triangles	5,804
	Used in (section)	5.1.2
	Name	Porsche
	Model File Name	Big_porsche.ply
	Source	
	Number of Triangles	10,474
	Used in (section)	5.1.2

	Name	Toaster
	Model File Name	Toasters004.obj
	Source	The Utah 3D Animation Repository
	Number of Triangles	11,141
	Used in (section)	5.1, 5.2, 5.3
	Name	Sdragon
	Model File Name	Dragon3.ply
	Source	The Stanford 3D Scanning Repository
	Number of Triangles	47,794
	Used in (section)	5.1, 5.2, 5.3
	Name	Bbunny
	Model File Name	Bunny1.ply
	Source	The Stanford 3D Scanning Repository
	Number of Triangles	69,451
	Used in (section)	5.1.1, 5.1.4, 5.1.5

	Name	Complex
	Model File Name	Urn2.ply, torus.3ds, big_spider.ply bunny1.ply, big_dodge.ply
	Source	Stanford 3D Scanning Repository, RenderMonkey
	Number of Triangles	98,867
	Used in (section)	5.1.1, 5.1.4, 5.1.5
	Name	200k
	Model File Name	Dragon2.ply
	Source	The Stanford 3D Scanning Repository
	Number of Triangles	202,520
	Used in (section)	5.1.1, 5.1.4, 5.1.5
	Name	400k
	Model File Name	F000.obj, dragon3.ply, bundha2.ply
	Source	Stanford 3D Scanning Repository, Utah 3D Animation Repository
	Number of Triangles	436,942
	Used in (section)	5.1.1, 5.1.4, 5.1.5

	Name	Fairy
	Model File Name	F000.obj, bunny1.ply, dragon2.ply, galleon.ply, ElephantBody.3ds, bundha2.ply
	Source	Stanford 3D Scanning Repository, Utah 3D Animation Repository, RenderMonkey
	Number of Triangles	679,531
	Used in (section)	5.1.1, 5.1.4, 5.1.5
	Name	990k
	Model File Name	Buddha1.ply
	Source	The Stanford 3D Scanning Repository
	Number of Triangles	1,087,716
	Used in (section)	5.1.1, 5.1.4, 5.1.5

Some scenes were composed of several models, such as the 'Complex' and 'Fairy' scenes. The models were chosen based on the number of triangles, but their distribution was not considered. The camera angle and position were not important for any scenes except the Toaster and the Sdragon because they were the only two scenes used in rendering tests. The camera had a 90 degree viewing angle and was stationed at (0, 0, 25) from the origin. The Toaster and the Sdragon were scaled to cover approximately 70% of the rendering windows during rendering tests.

4.4 Software Measurement Tool

4.4.1 Power

Power was measured using the CallNTPowerInformation function in the Windows API. For a detailed description of this function, please see MSDN at http://msdn.microsoft.com/library

/default.asp?url=/library/en-us/power/base/callntpowerinformation.asp. We used this function to retrieve battery information into the SYSTEM_BATTERY_STATE structure. The information for the structure is referenced in the MSDN Website as well. There are 11 variables in the structure, however, we only used the information in the Rate and RemainingCapacity variables. Rate returns the rate of discharge of the battery in milliWatts. RemainingCapacity returns the estimated remaining capacity of the battery in milliWatthours. The CallNTPowerInformation function has a limited update rate of 3-6 seconds. If the function is called more often than this, the variables will not change, which means the information returned is the same when the function is called a hundred times within one second and when it is called one time within two seconds. The hundred calls case will simply give us a hundred repeated values. This is problematic because all our tests can finish execution within one second. The function will report there are no changes in the discharge rate of the battery because the application does not run long enough.

To work around this problem, we ran the tests for more than three seconds. In fact, we had to run the tests much longer than three seconds to gain enough discharge rate samples. The running length was set to three minutes for all the tests. Three minutes might seem too short because it only gives us roughly 45 meaningful samples due to the limited power sampling resolution. A longer testing length was considered and tested. Five minutes of testing was used, but we found little variance compared to three-minute testing. Therefore, we settled on a three-minute testing length because shorter running time allowed more tests to complete before the battery had to be recharged.

4.4.2 Power Test Settings

The CallNTPowerInformation ran in a different thread and the power data was queried every second. With a sampling rate of 3-6 seconds, we had roughly four repeated samples on average. The three minute 10 second testing length gave 190 samples and the average discharge rate was calculated from these samples. The data were output to a comma-separated values (CSV) file just prior to application termination.

The tests were divided into three major categories, the build test, the static rendering test and the dynamic rendering test, as explained in Section 5. The build tests and dynamic rendering tests started after the application loaded the triangles into the system memory. The static rendering tests started after the application built the acceleration structures. The processing before the testing target is referred to as the *preprocess*. We wanted to isolate the power used by the preprocessing from the target so the application was set to sleep for 10 seconds before the test began, allowing the power draw to drop back down to the idle stage so the preprocessing power did not show up in the measurement. The power measurement started after the 10-second sleep. There was another 10-second sleep after the test ended so the fall of the discharge rate back down to the idle stage can be plotted. This gives us a clear picture of how the discharge rate rose and fell over the duration of a test. Figure 15 shows a representative test result.

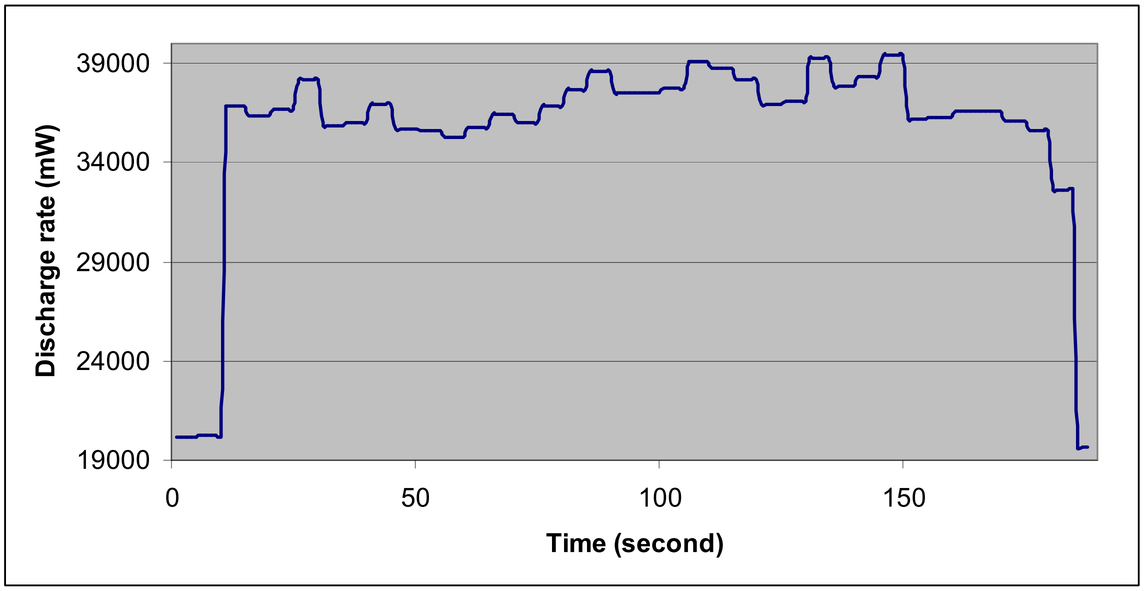


Figure 15 Discharge Rate Graph

Figure 15 shows how the discharge rate rises above 34W in the beginning and drops back down to 19 W in the end. The tiny plateaus are clear evidence of repeated discharge rate values. The 10-second sleep time is added to the testing length; therefore, we have a total running length of 3 minutes and 10 seconds. In addition, the average discharge rate is calculated by removing the samples below 24W in the CSV file because they represent the power used by the system when the application was in a sleep state. The average discharge rates and the standard deviations are reported in Appendix A.

4.4.3 Time

Time was measured with the timeGetTime function in the Windows API. A detailed description of this function can be found on the MSDN Website as well. The function returns time in millisecond resolution. This function is used to manage all the timers in the application such as the 10-second sleep and three-minute running length for the power measurement tests. If the application is in the middle of a loop when the three-minute timer is triggered, it will wait for the loop to complete before terminating the program. The acceleration structure build time and the rendering time are calculated by taking the total run time of the application divided by the number of completed loops where the total run time refers to the running length of the target operations such as the build or the rendering. When rendering dynamic scenes, the total run time includes the build time, transfer time from CPU to GPU, and the rendering time.

5 Measurement Results and Discussion

In this section, we present three categories of power measurement, power measurement results for building acceleration structures, rendering results without rebuilding acceleration structures, and rendering results with acceleration structures rebuilt per frame. They will be referred to as build results, static rendering results, and dynamic rendering results. Each category is further broken down into smaller sub-categories which are organized by different testing parameters.

Table 3 lists all the values gathered or derived from the experiment results. The term "operation" in this table refers to the targeted energy; if the test is conducted while rendering, then the operation is rendering. If the test is for building acceleration structures, then operation refers to build. It is used to aid the explanation of the experiment results in this section.

Table 3 Measurement Values

Name	Short Notation	Type	Unit	Equation	Note
Triangle #	T#	Given			
Completed Operations	Loop	Measured			
Time to Completion	T	Measured	Millisecond (ms)		
Discharge Rate	Rate	Measured	milliWatt (mW)		
Malloc Count	MC	Measured			
Malloc Size	MS	Measured	Byte		
Triangle-Box Intersection		Measured			uniform grid
Total Tree Node		Measured			Kd-Tree, BVH
Total Tree Leaf		Measured			Kd-Tree, BVH
Standard deviation	stdev	Calculated	mW		Appendix A
Time/Operation	TpO	Calculated	Ms	T/Loop	
Energy/Operation	EpO	Calculated	milliJoule (mJ)	Rate*(TpO/1000)	
Energy/Triangle	EpT	Calculated	mJ	EpO/T#	
Standard Error	STDEV	Calculated	mW		
Grid Size		Calculated			uniform grid
Max. Tree Depth		Calculated			Kd-Tree, BVH
Total Energy Loss	EL	Calculated	milliWatthour (mWh)	Rate*(T/1000)/3600	

5.1 Acceleration Structure Energy Measurement

The energy measurements for building the uniform grid (UG), Kd-Tree (KdT) and Bounding Volume Hierarchies (BVH) are presented in this section. They are further analyzed by considering model size, memory allocation size, triangle-box intersection counts and build time. The build energy used by the acceleration structure is very important because it distinguishes a static rendering task from a dynamic rendering task. The acceleration structure needs to be rebuilt every frame in the dynamic rendering task and its topology affects the performance of rendering. Thus, identifying the major elements that stress energy consumption in the building of the acceleration structure is the goal of this section. We tested nine scenes with a range of one to one million triangles, which we believe provided enough data samples to demonstrate the energy usage trend when building acceleration structures.

5.1.1 Model Size

Model size is an important factor to look at because it affects the rendering and building speed. Building the acceleration structure for larger models can take more time, and they can have a longer rendering time. Since model size and time have a linear relationship, we were interested in finding out the relationship between energy and time.

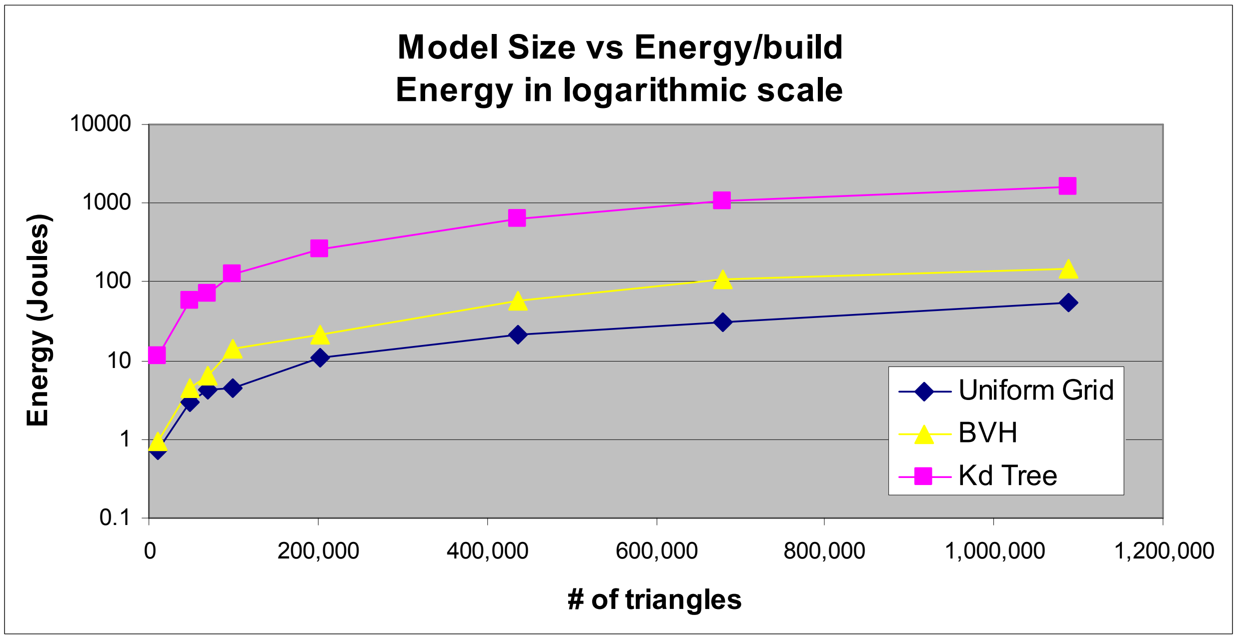


Figure 16 Data for Joule per Build with each acceleration structure

Figure 16 shows the energy (Joules) per build data for eight different scenes. The smallest scene has eleven thousand triangles and the biggest scene has about one million triangles. The data represented in the diamond markers are the energy measurements of uniform grid with triangle-box intersection tests. The square markers represent data for Kd-Tree (KdT) using SAH. The triangular markers are BVH data. The y axis, energy per build, is in logarithmic scale, but the x axis is not in logarithmic scale. Building KdT uses about ten times more energy than the other two data structures; furthermore, KdT energy usage also increases more quickly than in the other two data structures. On the other hand, BVH and UG use much less energy per build. They start off with similar energy usage but UG uses less energy per build after the model size increases over 100k.

Tables 4 and 5 show the average discharge rates (ADR) and standard deviations (STDEV) for each test. The singleTri results are not plotted in Figure 16 but it is shown in the tables. To find the energy (Joules) per build data from these tables, use the equation labeled Energy/Operation in Table 3.

Table 4 Build Data Standard Deviation, Part 1

	Uniform grid		Kd-Tree (SAH)		Kd-Tree (median)	
	ADR	STDEV	ADR	STDEV	ADR	STDEV
singleTri	33,119	218	34,901	238	35,176	216
Toaster	33,733	714	34,539	191	35,054	262
Sdragon	33,716	507	34,053	216	34,628	176
Bbunny	34,871	316	33,758	351	34,939	448
Complex	32,801	453	33,068	349	33,926	421
200k	32,786	397	33,848	393	34,446	342
400k	32,351	421	33,135	642	33,917	382
Fairy	33,913	258	35,075	357	35,613	317
990k	34,504	299	33,862	846	34,912	514

Table 5 Build Data Standard Deviation, Part 2

	BVH		BVH update (best case)		BVH update (average case)		BVH update (worst case)	
	ADR	STDEV	ADR	STDEV	ADR	STDEV	ADR	STDEV
singleTri	34,669	332	34,705	265	34,303	1428	34,217	583
Toaster	34,600	288	33,751	346	34,742	295	34,787	374
Sdragon	33,622	275	32,473	390	34,494	289	34,219	413
Bbunny	33,172	302	32,010	329	33,960	389	33,480	364
Complex	31,818	430	30,718	260	32,649	474	32,275	365
200k	33,032	579	32,143	245	34,106	296	33,914	482
400k	32,539	322	31,636	320	33,594	374	33,014	322
Fairy	34,161	469	33,077	296	35,269	288	34,983	400
990k	33,642	523	32,575	180	34,751	342	34,471	342

The data show very little variance in Tables 4 and 5. Their standard deviations are small and all of them deviate about 1% from their averages, thus, we believe our average data is accurate.

We take another look at the model-size versus energy trend by plotting the energy per Triangle graph for each acceleration structure. The naïve KdT (KdT with the spatial median split method) and the BVH update data are added to the graph as well. We are interested in the amount of energy that can be saved from using the naïve KdT and BVH update when compared to their complete build counterpart. The BVH update results are further broken down into best, average, and worst case scenarios. This is necessary because the BVH update algorithm does not depend on the size of the model, but rather on the amount of bounding volume update due to the triangle movement. This dependency can be reflected by performing the test on animated scenes; however, the animation process itself consumes additional energy that we would need to isolate. This additional energy is the energy used to update the triangles per frame and we do not wish to include that energy usage in the equation. One solution is that the program can sleep for few seconds before and after all the triangles updated their positions. The sleep allows the isolation of the energy on updating the triangles and updating the BVH, so only the data for updating the BVH is gathered. The second method is to emulate the update dependency with random number. Each leaf node is assigned a random chance to update, even though the triangle position does not change. We pick the emulation method because it is easier to control and produces reproducible results. We plot the best, worst and average cases for the BVH update. The best case represents the scenario where no update is necessary. This is equivalent of calling the BVH update on the same set of triangles. The average case has 60% chance that a leaf node will update itself and its corresponding parent nodes. The worst case is the scenario where all the triangles in the scene moved and all the BVH nodes have to update their bounding volumes.

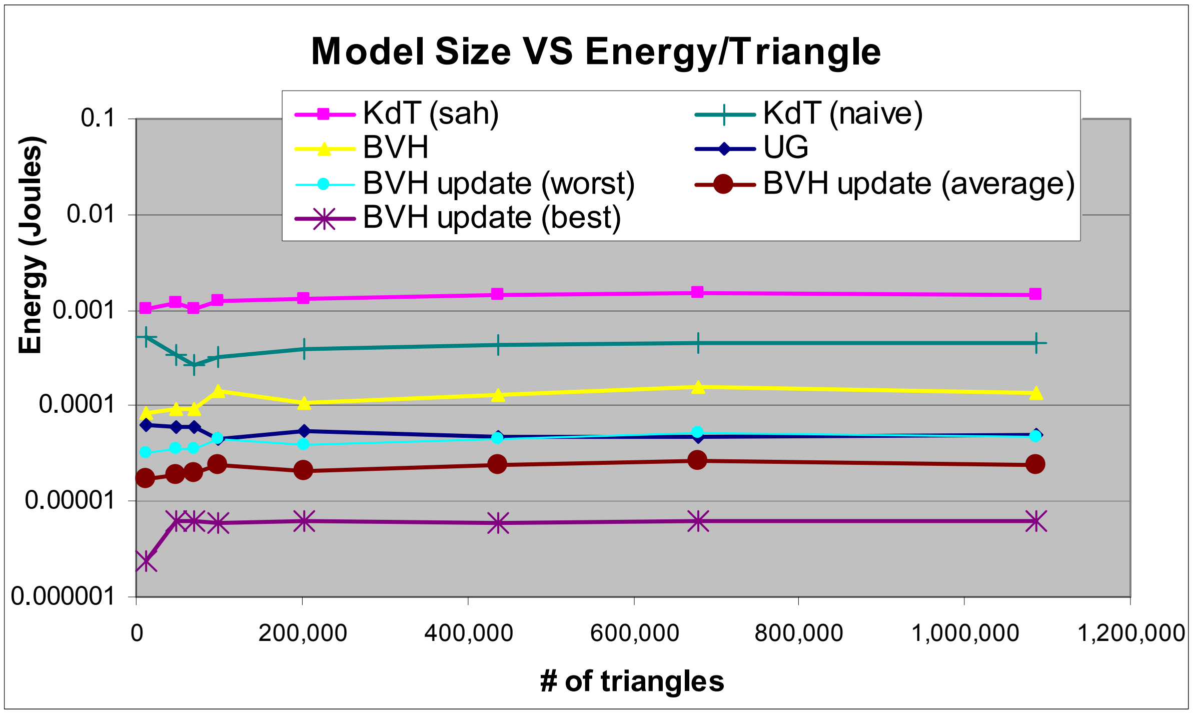


Figure 17 Energy (Joules) spent per Triangle Versus Model Size

Figure 17 shows energy spent per triangle for eight different scenes. The y axis is in logarithmic scale. The energy per triangle is plotted instead of the energy per build because we want to investigate whether the energy per triangle changed as the model size increases. Most of the lines in the graph are almost horizontal, thus suggesting that the energy spent on each triangle does not vary as the model size increases. This also means more triangles in the scene equals more energy per build. The naïve KdT has a disappointing improvement where it used 50% less energy than the SAH KdT, but still uses more energy than the BVH even though both algorithms use the median split method. BVH update is the most energy efficient build method in this graph. Its worst case performance results are similar to the uniform grid energy usage. Its best case results represent the scenario where no updates are necessary but they still have the overhead of checking the triangle position at each node. The best case results do not represent the results of not calling the update algorithm because the algorithm still loops through all the leaf nodes to check if updating the bounding volumes is necessary. Overall, the BVH update algorithm uses the least amount of energy per triangle; however, it is not suited for all types of models. Specifically, the BVH update does not work when the scene suddenly introduces new triangles. Furthermore, it requires occasional BVH rebuild if the triangles move beyond a certain threshold in order to maintain a suitable tree for rendering. In comparison, the uniform grid can be built from scratch every time and uses fairly low amount of energy to do so. The uniform grid is more energy efficient when building without triangle-box intersections (UG-N). We do not have the complete build data for UG-N but here are two measured energy per build results for the Toaster and Sdragon scenes.

Table 6 Joule per Build for Two Cases of UG-N

	UG	UG-N	Worst BVH update	Average BVH update
Toaster (11k)	0.7	0.3	0.4	0.2
Sdragon (48k)	2.9	1.1	1.7	0.9

Table 6 shows that UG-N energy consumption lies between the worst and average BVH update cases. Although this table only represents two data point, we believe this is enough to make the point that UG-N can be competitive against BVH update in term of energy. The UG-N requires slightly more energy than the average BVH update method. Future work should include complete UG-N results on the same test scenes.

5.1.2 Memory Allocation

Memory Allocation was found to affect build speed if handled naïvely as described in Section 3. The memory allocation issues are briefly discussed here to serve as a reminder. A typical UG implementation requires dynamic allocation of memory when inserting triangles into their associated cells. Kd-Tree (KdT) and BVH have recursive build functions that require memory allocation per function call. Furthermore, each KdT leaf node requires a dynamic array to hold any triangle references associated with the node. The memory issue can be solved with a custom memory pool. Doing smart memory allocation speeds the build process up by a factor of two or more; therefore, it is interesting to see if doing so gives similar benefits in term of energy.

In this experiment, we emulated the benefit of a custom memory pool without implementing it. A custom memory pool is a memory management class that allocates a large amount of memory from the system. The class assigns memory via pointer to the application so no further memory allocation is required from the system. Since we are looping the build function for three minutes, we can have the program reuse the memory allocated in the initial loop to avoid further memory allocations. The energy measurement starts after the initial loop and the measured energy is the build method without memory allocations.

The memory allocation experiments were conducted on smaller models. The model size ranges from ~600 to ~10,000 triangles. The scenes in Section 5.1.1 are not used because we suspect there is a crossover in the energy usage between BVH and UG when the model size is small. The result shows there is no crossover; therefore, it is not discussed or graphed. Figure 18 shows the Joule per build before and after the custom memory pool emulation.

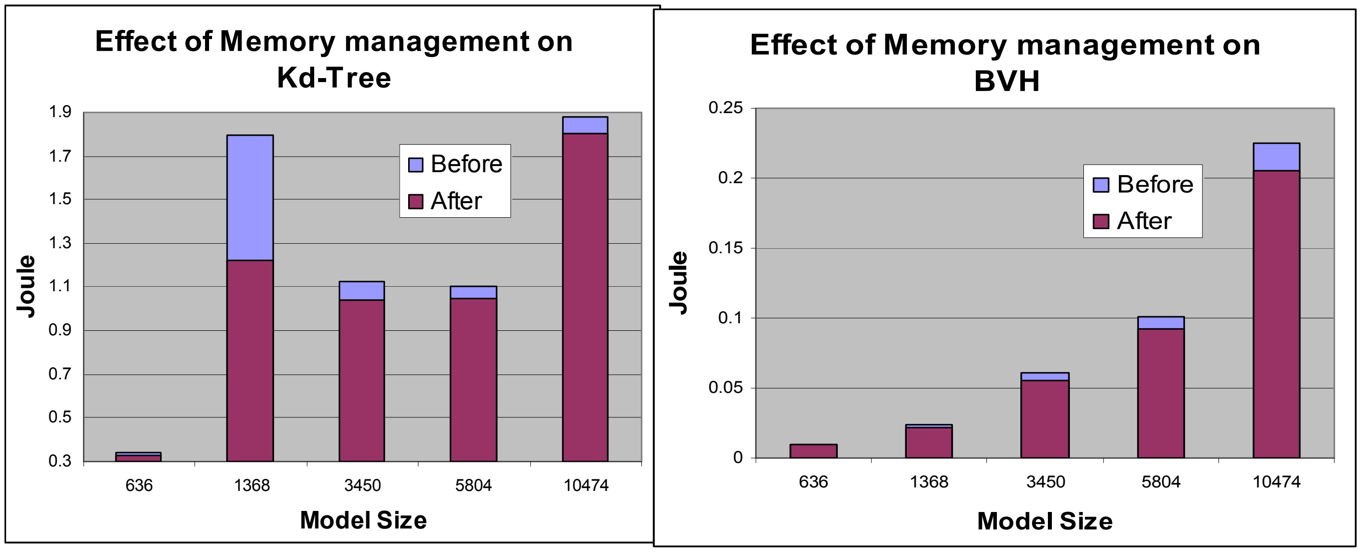


Figure 18 Energy Usage Reduction for Kd-Tree and BVH

Eliminating memory allocation does reduce build energy. The BVH result shows consistent improvement as the model size increases in Figure 18. The KdT result shows an irregular energy usage pattern and it does not grow linearly as the model size increases. Furthermore, the energy reduction is not consistent across different models. The model with 1368 triangles uses a lot less energy after the emulation while the others do not have the same amount of energy saving. The cause of this irregularity was not investigated and should be looked at in future work. Nevertheless, both graphs show memory allocation does contribute to the energy for building the acceleration structure. The average discharge rates and the standard deviation of the results are in Appendix A. Although the BVH and KdT are both tree structures, the BVH does not have significant energy saving when compared to the KdT. We believe this is related to the total memory size allocated so the memory allocation size is graphed in Figure 19.

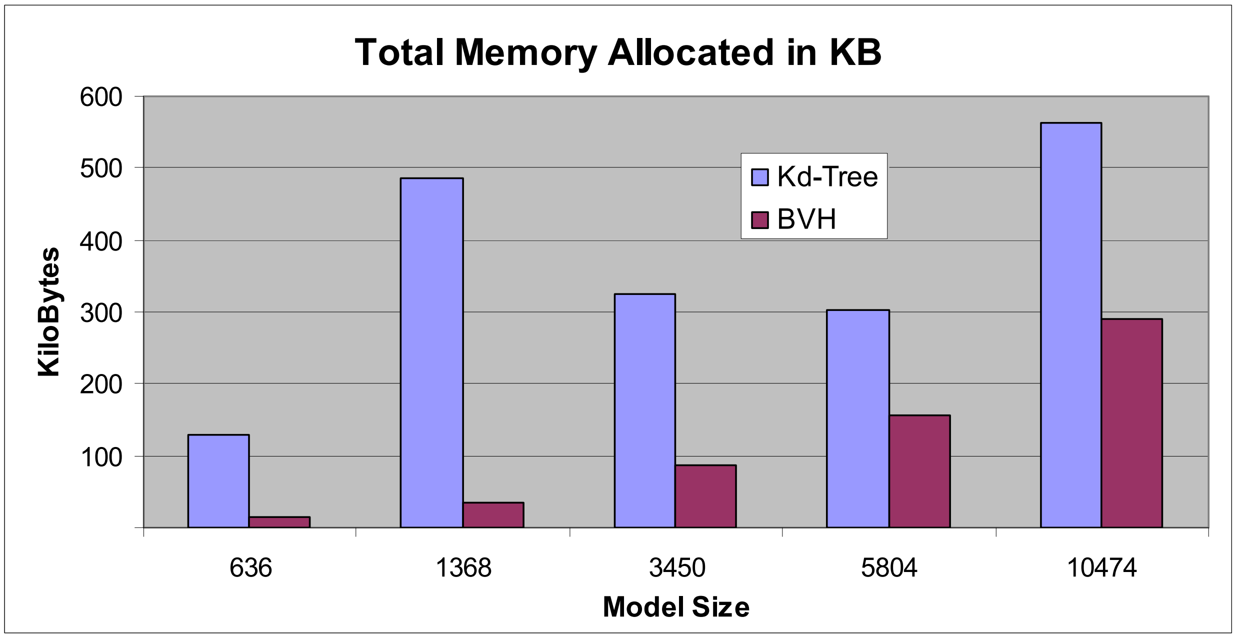


Figure 19 Total Memory Allocated Chart for Kd-Tree and BVH

Figure 19 shows that the BVH allocated less memory than the KdT and this explains why the BVH does not have significant energy saving with the emulation. The BVH allocated more memory as the model size increases which explains the linear pattern for the BVH shown in Figure 18. On the other hand, the KdT does not always allocate more memory as the model size increases. The model with 1368 triangles allocates 500 KB but the model with 5804 triangles allocates only 300 KB. Figure 19 suggests that memory allocated size might be the cause of the irregular energy usage pattern for the KdT; however, it does not answer why the '1368 model' saves the most energy. This issue is discussed further in the following section.

5.1.3 Memory Allocation Test Result Discussion

From Figure 19, the KdT always uses more memory than the BVH. The size of the memory allocated has been discussed; we now focus on the frequency of memory allocation. The number of memory allocation function calls might reveal more information about the effect of removing memory allocation. Figure 20 shows the number of memory allocation function calls during the building of KdT and BVH. For the purpose of this discussion, malloc will be used to represent the functions allocating memory from the system.

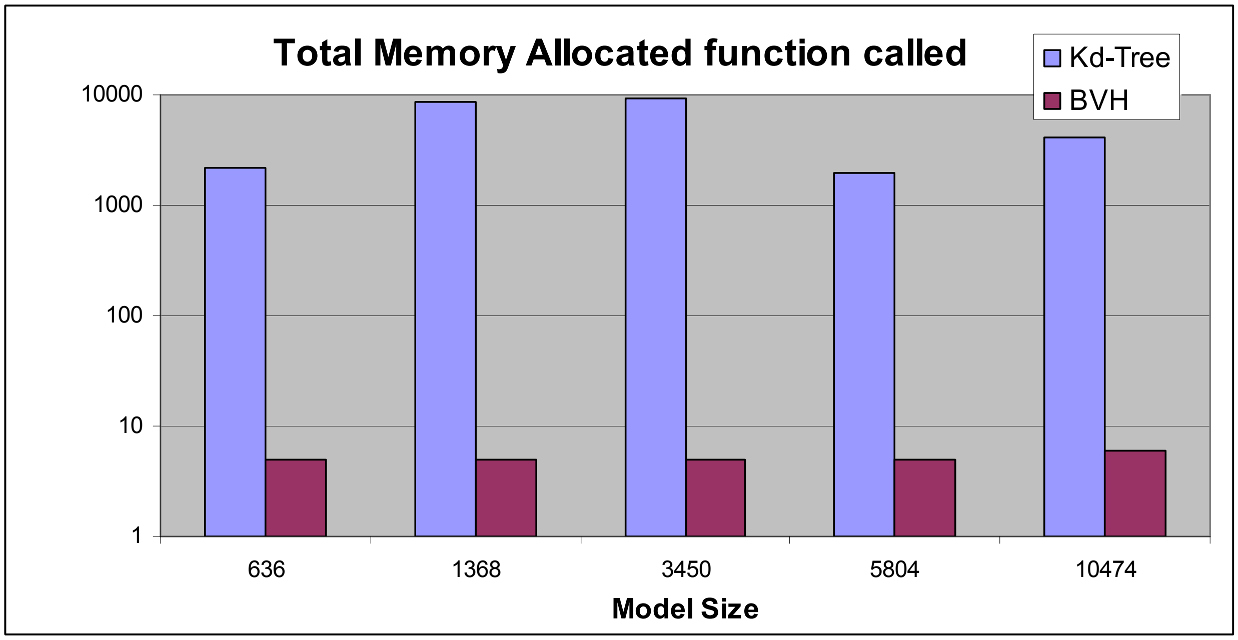


Figure 20 Memory Allocation Request Graph

From Figure 20, we see that the BVH never uses more than six malloc calls, but the KdT uses thousands. From Figures 20 and 19, we can conclude that the KdT allocates smaller memory blocks frequently and the BVH allocates bigger memory blocks only occasionally. This suggests that eliminating thousands of small malloc calls can lower energy consumption more effectively than erasing a few large malloc calls where 'small' and 'large' refer to the size of the memory allocated.

However, the reason for the significant energy saving from the KdT '1368 model' is still unanswered. We believe there are other elements that influence the energy savings when memory allocations are removed from the KdT build. We hypothesize that these could be cache usage, contingency of the memory allocated, and deallocation which should be investigated further in future work. Nevertheless, Eliminating or reducing memory allocations does improve energy efficiency.

5.1.4 Uniform Grid Triangle-Box Intersection

Uniform grid (UG) is not in section 5.1.2 because it already used custom memory pool. UG build mostly involves computation and the most intensive computation is the triangle-box intersection test. Section 3.1.1 shows that the triangle-box intersection test is necessary to produce an accurate UG. Furthermore, running the UG algorithm through a profiler has shown that the most expensive function is the triangle-box intersection. This makes triangle-box intersection test an important factor to look at when determining the energy per build for the UG. We hypothesize that if two scenes have the same number of triangle-box intersection tests, they will have similar energy consumption. Figure 21 graphs the energy per build against the number of triangle-box intersections per build.

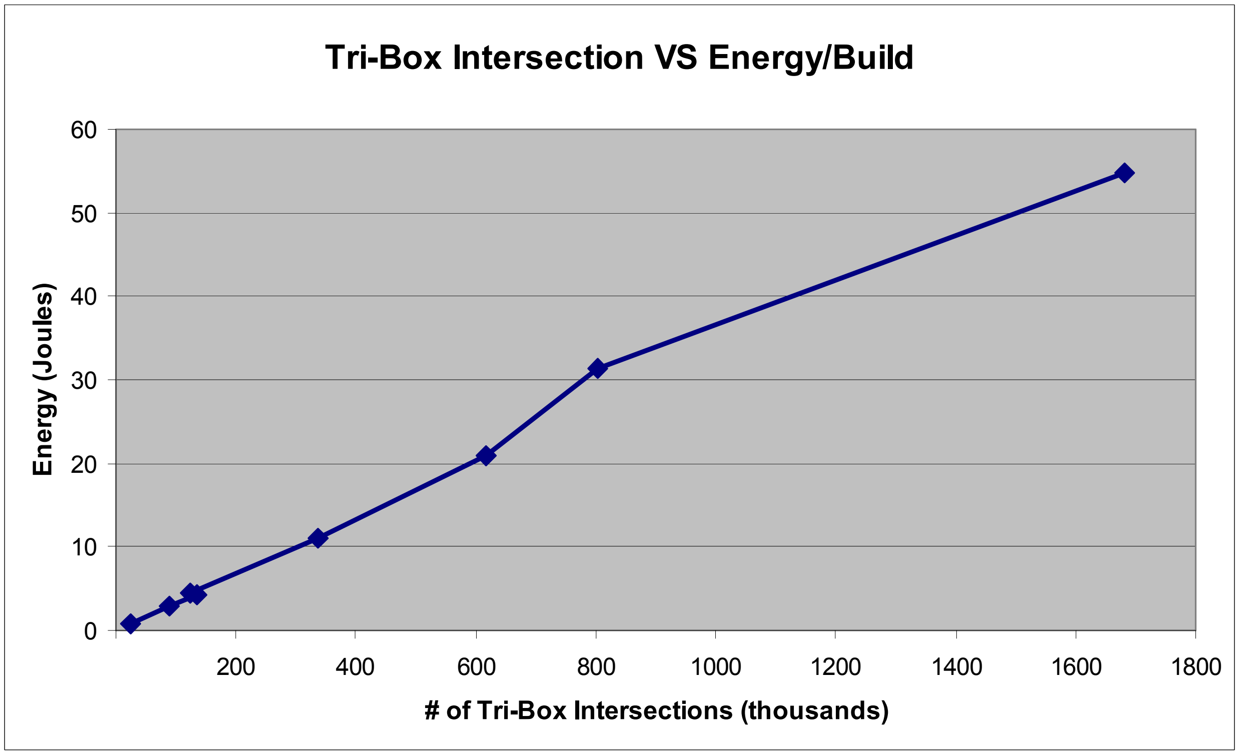


Figure 21 Triangle-Box Intersection Versus Energy per Build

Figure 21 shows a near-linear growth and the energy per build increases as the number of triangle-box intersection increases. The line has a slope about 3/100,000 and this suggests every 100k intersection equals 3 Joule of energy. The data does not reveal the amount of energy used by the triangle-box intersection tests and should be included in the future work. There are only two scenes, Toaster and Sdragon, which have data for non-triangle-box intersection UG build. The results are in Table 6 and they each show an energy reduction of around 61% and 66%. This suggests coarser non-triangle-box intersection UG build should be used if the rendering overhead is smaller than the energy saved from the build.

Finding models that produce the same number of triangle-box intersections is not trivial; bbuny (69k) and complex (98k) are the only scenes with similar triangle-box intersection counts in our data. They are the two dots overlaying each other near the 200,000 mark on the x axis. Table 7 shows the two models in more detail.

Table 7 Energy Comparsion on Intersection Count

Model Size	Build Time (ms)	# of intersections	Build Energy (J)
69k	123	13k	4.2
98k	137	12k	4.5

Table 7 shows the 98k model has fewer triangle-box intersections than the 69k model. The 98k model uses more energy despite the fact that it has fewer triangle-box intersections. The effect of triangle-box intersection is not as dominant as we had believed because the bigger model still uses more energy to build. Going from 123 ms to 137 ms is an 11% increase, but 4.2 to 4.5 only represents a 7% increase. Thus, this data suggests the 8% decrease in the number of triangle-box intersection does lower energy consumption in minor percentage. We are not able to prove our hypothesis on the triangle-box intersection because we do not have enough samples and future work should be investigated on more models with the same amount of triangle-box intersection.

5.1.5 Build Time

It is intuitive to think that less running time equals less energy consumed. Everyone in the ray tracing community had always tried to minimize the running time and if less time means less energy; it is another reason to reduce the running time more aggressively. If time does not relate to energy, it is still a major finding because no one has looked at the energy consumption for ray tracing before. We expected the graph to come out somewhat linear because the test results have similar discharge rates and the equation for energy per build is build time multiplied by the discharge rate for the given operation. Figure 22 is the energy versus time graph.

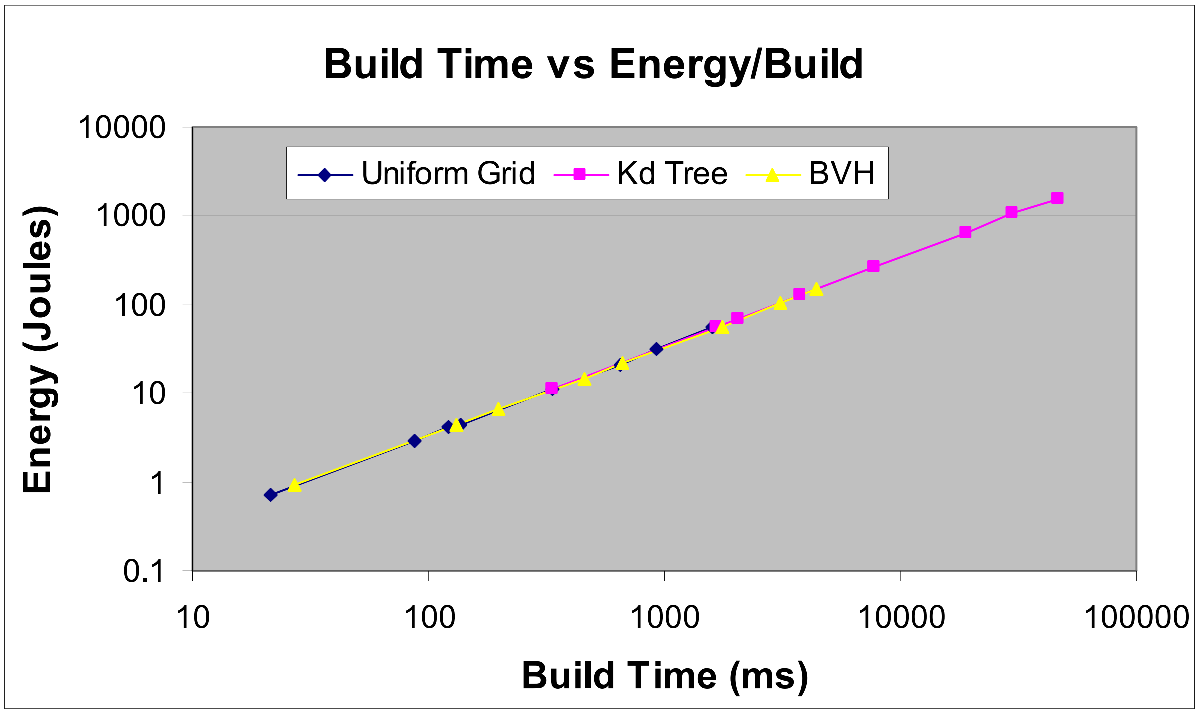


Figure 22 Build Energy versus Build Time

Both axes are in logarithmic scale in Figure 22. The line has a slope of about 34 J/S which is the average discharge of the system at 100 % CPU utility when running the build tests. Figure 22 implies that less time equals less energy because the energy per build increases as the build time increases. The calculated average discharge rate for all the build tests is 33.6 J/S and has a standard deviation of 1 J/S. The confidence interval of 95% is 0.25 J/S with 63 samples. The equation for energy per build is average discharge rate * build time. Since we know the average discharge rate is 33.6 J/S, it becomes the constant for the equation. The equation becomes 33.6 * build time. Since the build time is the only variant, it determines the energy per build.

We were expecting different acceleration structures to have very different discharge rate, thus they will have different rate of change; however, they turned out to have similar discharge rates. This suggests that all build algorithms have similar CPU utilization which has a 33.6 ± 0.25 J/S discharge rate on the testing machine.

5.1.6 Build Energy Discussion

The simple relationship between time and energy is surprising. It suggests that work should be focused on reducing the running time, and that reduction in energy will be an additional benefit. Our original hypothesis states that energy is not directly related to time; however, we found a strong correlation between energy consumed during the build and build time. Faster algorithms are not necessarily more complex or use more power. The build algorithm complexity does not affect the battery discharge rate; therefore, faster algorithms consume less energy overall.

Looking at acceleration structures in general, they are algorithms that partition data so that unnecessary triangle-ray intersections can be avoided. This type of algorithm typically involves the allocation of memory before the algorithm begins. The algorithm does computation to find the triangle's corresponding location in the structure and inserts the triangle reference. The major difference is the amount of energy required during the computation stage to find the location for each triangle. In another words, we are only comparing the amount of computation required for each build algorithm. The build algorithms do not fit well with our hypothesis because they are not complex algorithms that run to completion quickly. The KdT is the most complex to build but it requires a lot of time to complete. The UG is the simplest and takes little time to complete. We do not see the real difference between each algorithm, but also fail to define how the complexity should be measured. It is possible that beyond a certain level of complexity, the CPU runs at almost 100% utilization. So if two algorithms run at almost 100% utilization, their battery drain rate will be similar. Since all three build algorithms are computation intensive algorithms, they can be seen as having the same complexity. Therefore, they have similar discharge rates. If the build algorithms have different operations, we expect to see different discharge rates. For example, we can compare a memory allocation heavy algorithm to a computation intensive algorithm. Our preliminary studies show an algorithm doing memory allocation and deallocation of 340 KB has a 32.5 J/S discharge rate, while an algorithm doing two additions and multiplications requires 29 J/S.

Another important concept is that improvements carried out to minimize time tend to reduce wasted work by avoiding expensive computations and increasing memory efficiency. Avoiding work means saving the energy required to carry out work. Increasing memory efficiency means less paging and cache fetches, thus reducing work and wasted energy as well.

The fact that the testing machine has a 33.6 J/S discharge rate is not discouraging. The energy consumption can be predicted if the build time is known. The 33.6 J/S is the overall discharge rate of the system when running the build algorithms. The energy consumed by the algorithm alone can be calculated by subtracting the idle discharge rate from 33.6 J/S. The idle discharge rate is the discharge rate when the system is only running the energy measurement algorithm and the operating system. The average discharge rate will not be the same if the tests are conducted on a different machine; however, we believe the following equation will apply by finding the average discharge rate of any acceleration structure on the machine. The energy per build can be calculated using the following formula.

EpB = (33.6-IdleB)*bt
where
bt = build time in second.
IdleB = discharge rate in the idle mode
EpB = energy (Joule) per build

To conclude, time is the most dominant factor affecting the build energy. Uniform grid is the cheapest structure to build in terms of energy and time. It is scalable because it has low energy consumption per triangle. In addition, it can handle animated scenes that change their triangle counts per frame. BVH is the best choice when the scene only contains deformable models because it can update the tree with low cost. It is not suited for all general scenes because the update algorithm is dependent on the movement of the triangles in the scene. The Kd-Tree should be avoided if one plan to build the structure repeatedly, and an efficient update mechanism would does not penalize the rendering performance needs to be investigated. Lastly, memory allocation and computation reduction should be explored to optimize the energy consumption of the acceleration structure build algorithm.

5.2 Static Rendering Energy Measurement

Static rendering energy consumption is not the focus of this paper, but the data is necessary to isolate energy consumption during the dynamic rendering test. The only difference between dynamic and static rendering is the building acceleration structure step. Dynamic rendering requires per frame acceleration structure rebuild whereas static rendering does not. In static rendering, we measure the energy per render of the Toaster (11k) and Sdragon (48k) scenes. The tests are conducted in two different resolutions: 256x256 and 768x768. Every scene is rendered using 5 different acceleration structure builds: the naïve Kd-Tree, the SAH Kd-Tree, the uniform grid with the triangle-box intersection, the uniform grid without the triangle box intersection, and the Bounding Volume Hierarchy. The short notations used for each structure are the following:

KDN: Kd-Tree with naïve split (spatial median)
KDS: Kd-Tree with SAH split (surface area heuristic)
UGT: Uniform grid with the triangle-box intersection
UGN: Uniform grid without the triangle-box intersection
BVH: Bounding Volume Hierarchy

The BVH does not have an alternative build method because the BVH update does not produce a different structure if the triangles do not move. The alternative build methods for Kd-Tree and UG produce different topologies of the acceleration structure that are likely to impact upon the rendering time; therefore, they are included as testing parameters. Furthermore, the benefit of the coarser builds and the decrease in rendering performance can be compared.

The tests are conducted on both CPU and GPU. CPU represents the algorithm running completely on the CPU and GPU represents the rendering algorithm running on the GPU. The results are presented per scene in each sub-section. The average discharge rate and the standard deviation can be found in Appendix A.

5.2.1 11K Model Results

The 11k model result is the rendering energy consumption of the Toaster scene. The picture of this model can be found in Section 4.3. The Toaster scene is a box with a few toys inside. The original coordinate has the box lying flat on the z axis. The test has the camera rotate around the model as described in Section 4.3. This makes static scene testing non-trivial because the rendering engine has to generate new eye rays and traverse different paths into the acceleration structure per frame. Due to the rotating camera, the rendering time per frame is not the same for every frame and the results have more variance. We believe that a better test decision would be not to rotate the camera so that a more accurate average discharge rate and rendering time could be obtained. Figure 23 shows the energy per frame for each acceleration structure on the CPU and the GPU at 256x256 screen resolution. Figure 24 represents the result at 768x768 screen resolution.

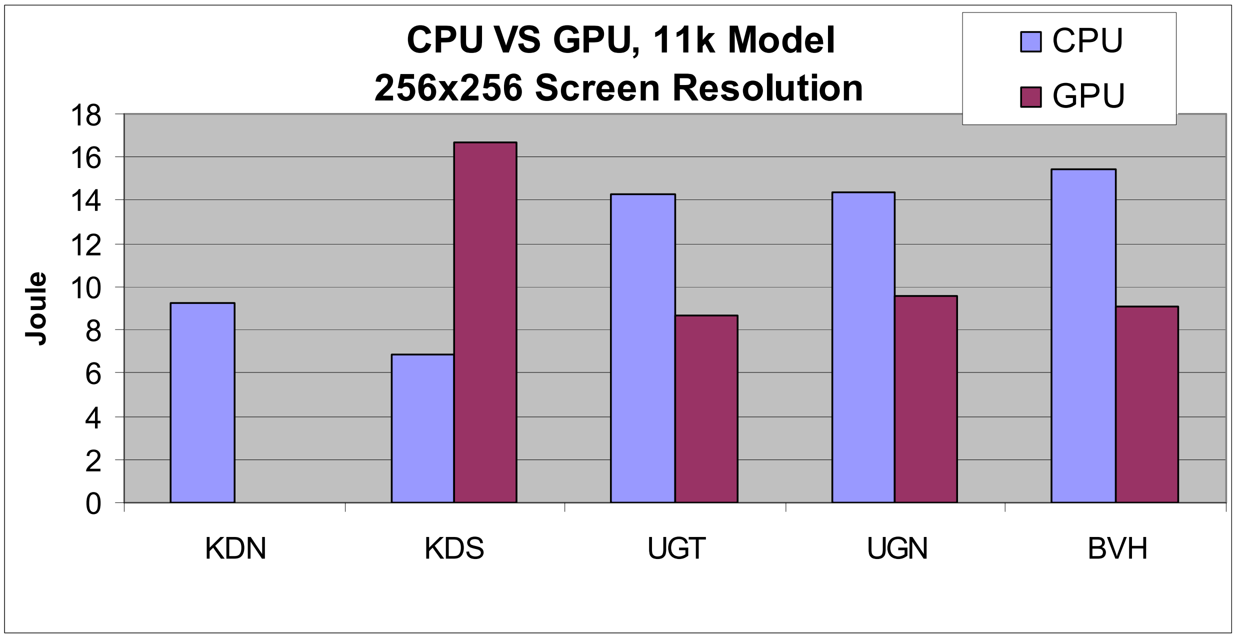


Figure 23 Energy Comparison Rendering 11k Model at 256x256 Resolution

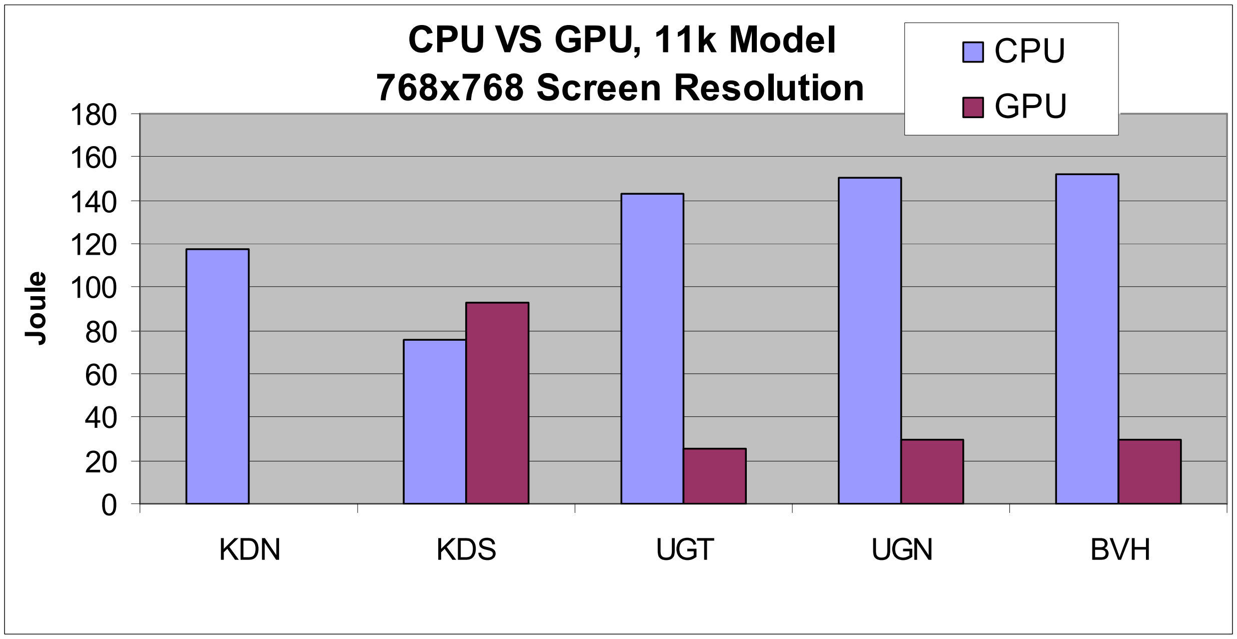


Figure 24 Energy Comparison Rendering 11k Model at 768x768 Resolution

The terms lower and higher resolution are used to refer to 256x256 and 768x768 screen resolutions, respectively, in this section. As shown in Figures 22 and 23, KDS uses the least amount of energy per frame in the CPU tests; however, KDS perform poorly on the GPU. UG and BVH use similar energy per frame and they use the least amount of energy on the GPU in the higher resolution. At the lower resolution, the GPU UG and the GPU BVH only differ from the CPU KDS by 2 Joules. This confirms the effectiveness of GPU ray tracing. The GPU KDN data are not presented because the algorithm did not render the image correctly due to a limitation on the number of loop iterations in the shader code that limited the traversal depth; therefore, the data is removed from all the graphs. Future work should correct this seek ways to overcome this limitation.

Table 8 Energy Reduction from Coarser Build for 11k Model

	CPU Kd-Tree		CPU UG		GPU UG	
Screen Resolution	lower	higher	lower	higher	Lower	higher
Energy reduction from coarser build (Joule)	20.8		0.5		0.5	
Energy increased in rendering (Joule)	2.3	42	0	7.3	0.8	3.6
Total energy saved (Joule)	18.5	-21.2	0.5	-6.8	-0.3	-3.1

Table 8 shows the benefits of coarser builds and the rendering overhead. The KDN and the UGN use more energy at higher resolutions. They received no benefit from the coarser build methods. The CPU KDN and UGN exhibited some energy saving at lower resolutions because the rendering overheads are minor. The table suggests that a coarser building method is not beneficial in general; however, more samples are required to confirm this conclusion. Future work should include more rendering results to verify this.

Figures 23 and 24 do not give a clear picture of the percentage increases when the screen resolution increases. Figure 25 presents a bar graph with the percentage increase in energy per frame, comparing the lower resolution to the higher resolution. From Figure 25, we can see that CPU algorithms adapt poorly as the screen resolution increases. The CPU KDN requires 12 times more energy and the CPU UGT needs 10 times more. The higher resolution requires 9 times more pixels to be painted than the lower resolution. The CPU rendering algorithms grows linearly as the screen resolution increases. On the other hand, the GPU algorithms only use 3 to 5 times more energy. This results show that the GPU was not fully utilized at the lower resolution. Overall, the GPU is more adaptable to screen resolution changes.

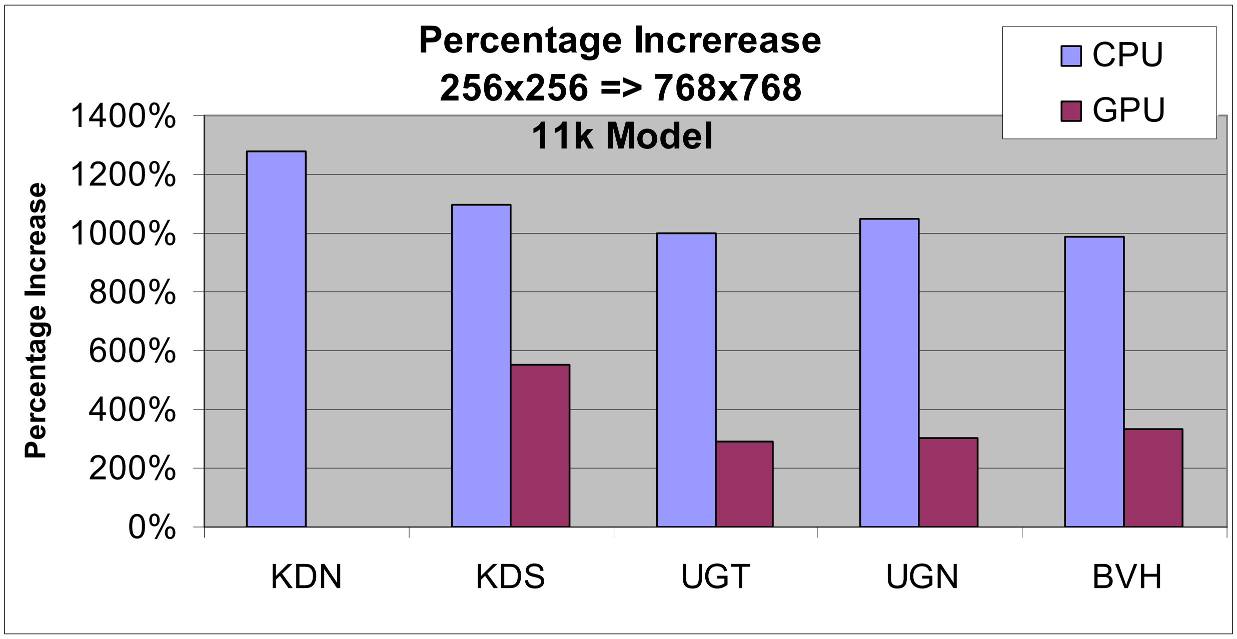


Figure 25 Percentage Energy Usage Increase from 256 to 768 Resolution 11k Model

Knowing that the GPU is more efficient at higher resolutions, we want to know how much improvement in energy the GPU can provide at the same resolutions. Figure 26 shows the percentage of energy saved from moving CPU rendering into the GPU.

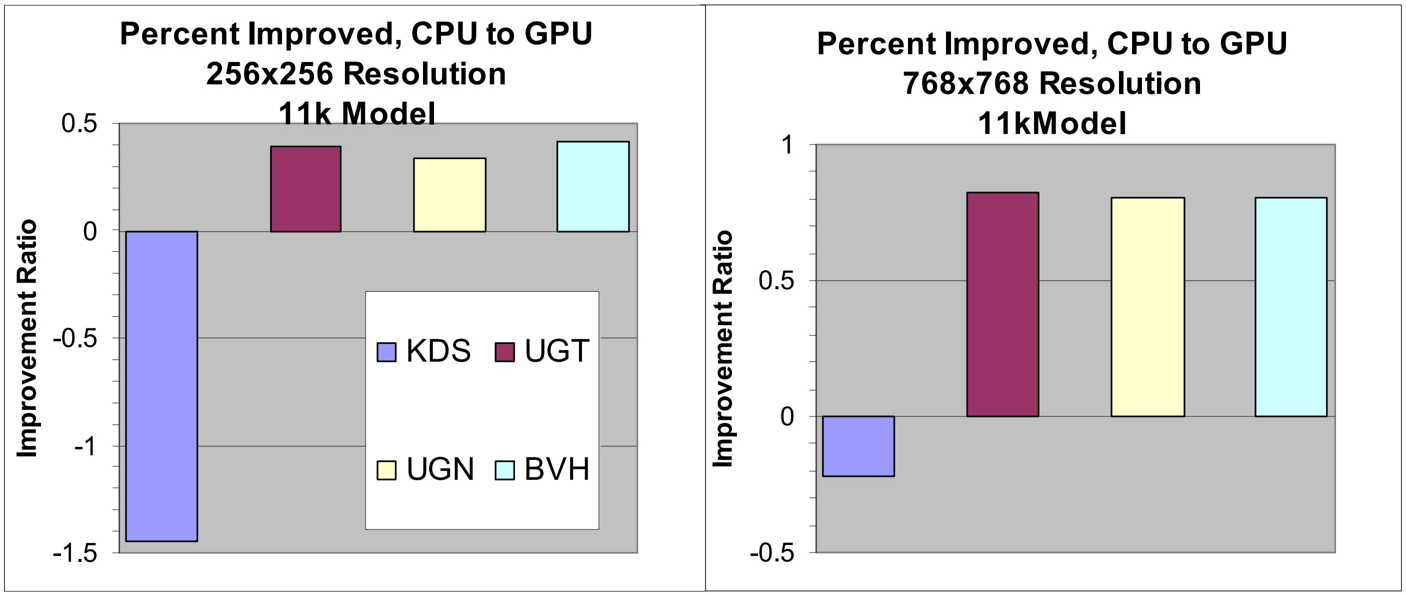


Figure 26 Percentage Energy Reduction in Moving Rendering Task to GPU (11k Model)

Figure 26 shows that the Kd-Tree does not benefit from the GPU and its performance is better than that of the CPU. On the other hand, UG and BVH reduce their energy consumption about 40 percent with the GPU in the lower resolution and 80 percent in the higher resolution.

Nevertheless, the GPU has been shown to be a good platform for ray tracing. It is scalable in terms of screen resolutions and it requires less energy to render a frame with the UG and the BVH. CPU does better on the lower resolution and KDS can render an image with the least amount of energy.

5.2.2 48K Model Results

In this section, the Sdragon scene (48k model) results are presented. The same sets of the graphs similar to Section 5.2.1 are drawn. Table 9 shows the benefit of the coarser builds for the Sdragon scene.

Table 9 Energy Reduction from Coarser Build for 48k Model

	CPU Kd-Tree		CPU UG		GPU UG	
Screen Resolution	lower	higher	lower	higher	lower	higher
Energy reduction from coarser build (Joule)	73		1.8		1.8	
Energy increased in rendering (Joule)	3.1	26.9	0	2.9	1.2	4.4
Total energy saved (Joule)	69.9	46.1	1.8	-1.1	0.6	-2.6

Unlike in the 11k results, the Kd-Tree benefits from the coarser KDN build for the 48k model in both screen resolutions. The coarse UG build still consumes more energy in the higher resolution setting. The trade off between the coarse build and rendering performance is interesting and remains a question for future work.

Figure 27 shows the energy per frame for each acceleration structure on the CPU and the GPU with 256x256 screen resolution. Figure 28 is the result with 768x768 screen resolution.

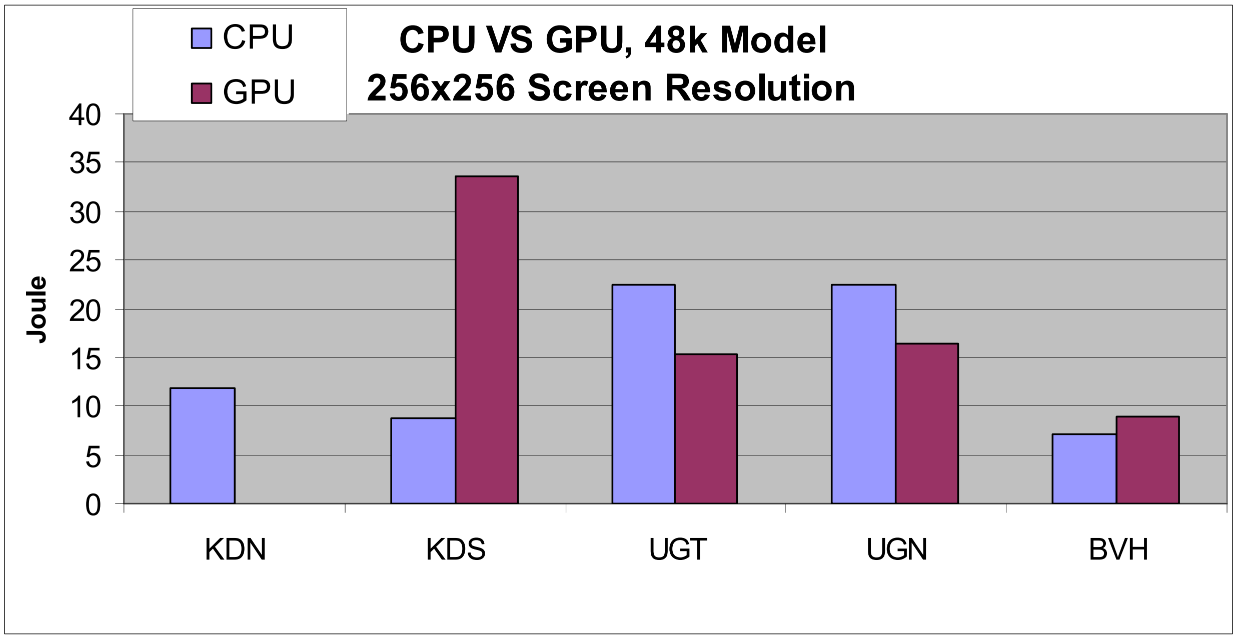


Figure 27 Energy Comparison Rendering 48k Model at 256x256 Resolution

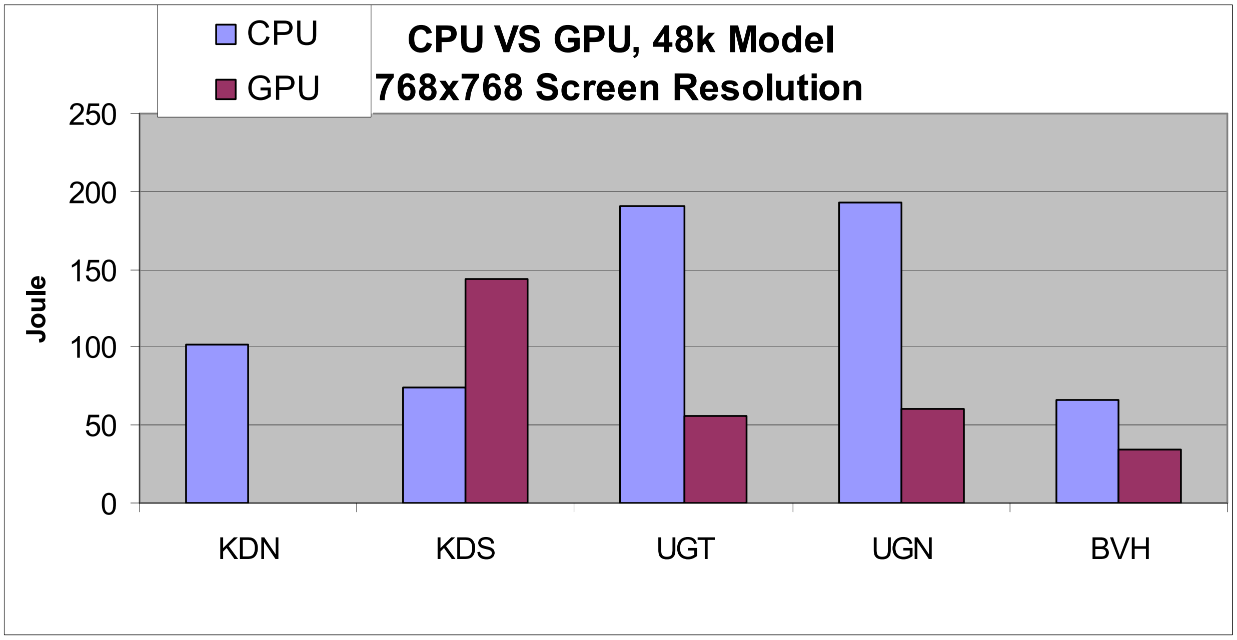


Figure 28 Energy Comparison Rendering 48k Model at 768x768 Resolution

Not surprisingly, the KDS is very efficient in terms of energy; however, the BVH actually use least amount of energy for the 48k model. Similar to the 11k model result, the GPU UG and the GPU BVH use the least amount of energy at the higher resolution. The major difference between the 11k result and 48k result is that CPU BVH is as energy efficient as its GPU counterpart.

Figures 29 and 30 do not exhibit significant differences when compared with the 11k model results; thus Figures 29 and 30 are not discussed.

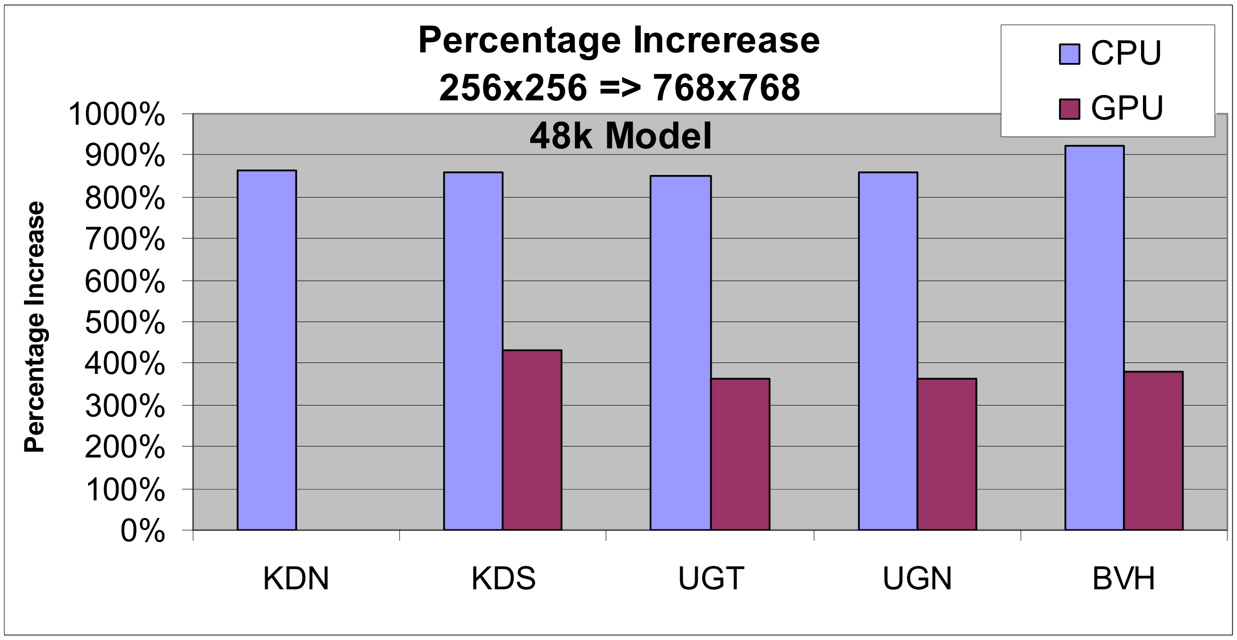


Figure 29 Percentage Energy Usage Increase from 256 to 768 Resolution 48K Model

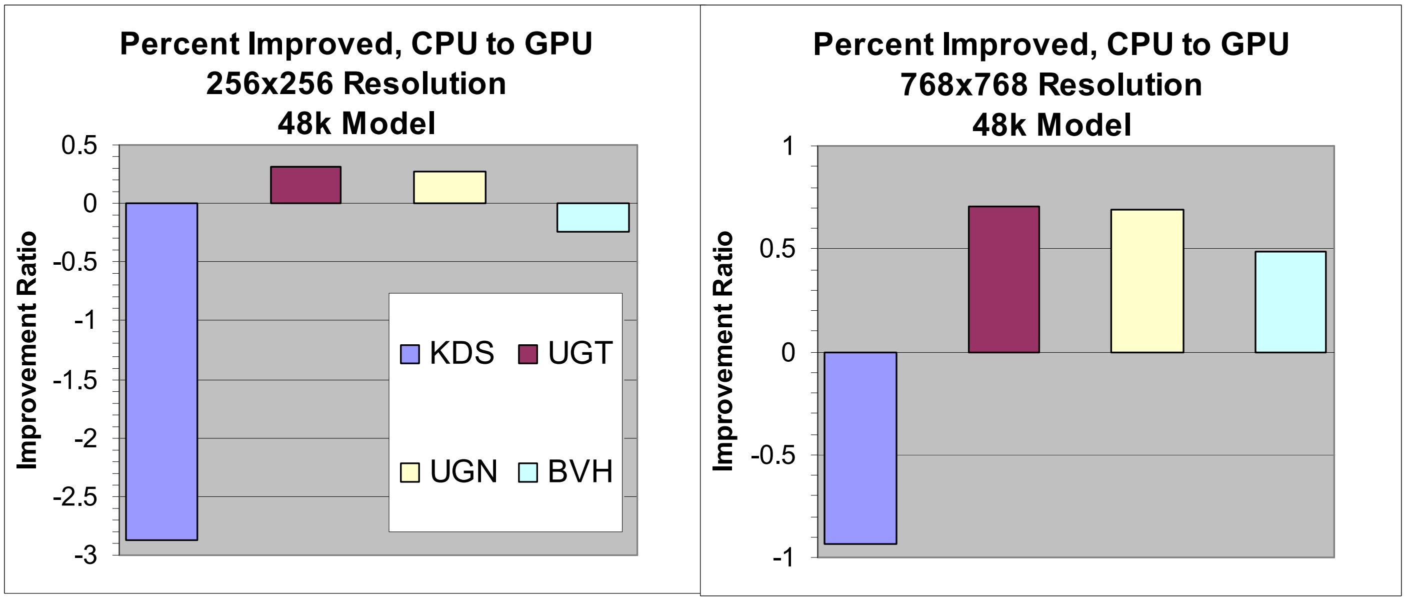


Figure 30 Percentage Energy Reduction in Moving Rendering Task to GPU (48k Model)

5.2.3 Static Rendering Discussion

From the data, we can say the static rendering energy consumption depended on both the acceleration structure (AC) used to render and the screen resolution (SR). Rendering a model with the wrong AC will cost more energy. Even though there is no universal AC that always gives the best rendering performance, the data show that SAH Kd-Tree is a good choice on the CPU platform. This confirms the statement from Wald in his PhD dissertation [Wald 2004] that a good SAH Kd-Tree is generally the best acceleration structure for static scenes. Unfortunately, the SAH Kd-Tree does not have the same performance on the GPU cases. The BVH and UG do much better on the GPU. The GPU BVH outperforms every other AC and it does almost as well as the CPU KDS at the lower resolution. Table 10 summarizes these findings. It is important to realize that these are just general guidelines, as there are always models that will heavily favor a particular AC. This only applies to single-ray ray tracing.

Table 10 Rendering Platform Recommendation

	Best Combination
Low Resolution	CPU-KDS or GPU-BVH
High Resolution	GPU-BVH

The rendering data with CPU has an average discharge rate of 34.6 J/S ± 0.4 J/S. The confidence interval is calculated with 95% confidence and 20 samples. The GPU data has an average discharge rate of 40.2 J/S ± 0.8 J/S. The confidence interval is calculated with 95% confidence and 16 samples. The data can be reproduced using the average discharge rates and the standard deviations for the static rendering result in Appendix A. The GPU rendering algorithms have much higher variance. Figure 31 demonstrates the average discharge rate for each acceleration structure when rendering the 11k model. KD represents Kd-Tree in Figure 31.

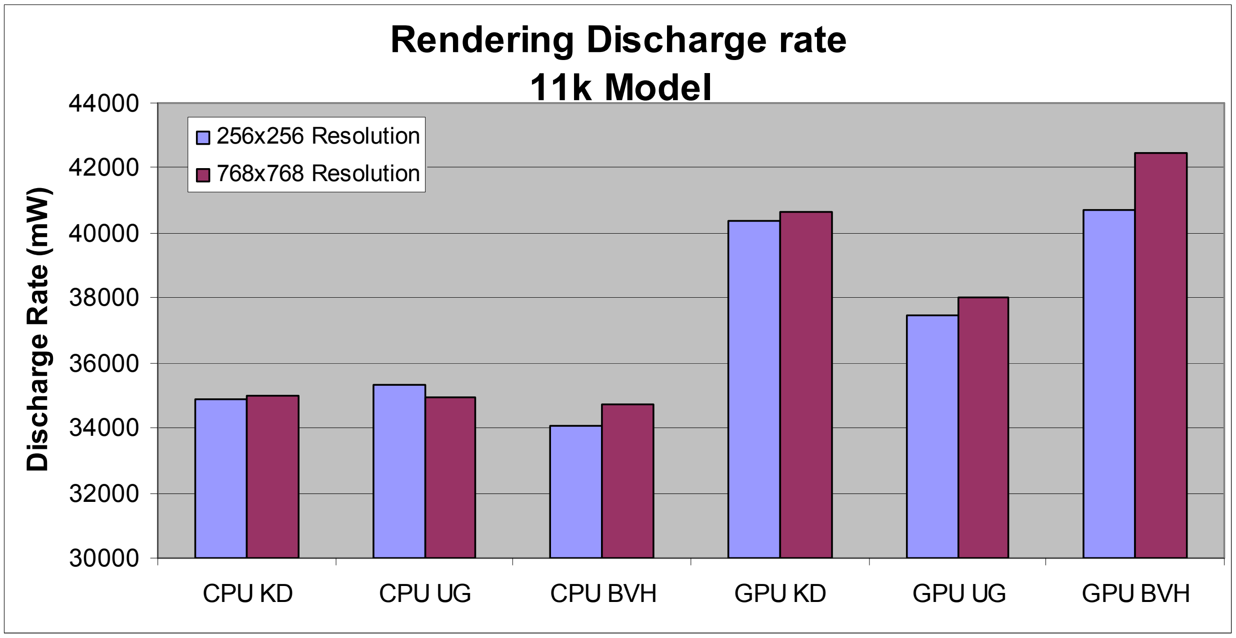


Figure 31 Average Discharge Rate with 11k Model

Figure 31 shows that BVH and Kd-Tree use 40-42 J/S when rendering but UG requires only 38 J/S. The GPU average discharge rates exhibit greater variance, suggesting that the GPU is not utilized to its full capacity. Future work could focus on improving the efficiency of the GPU algorithms. The difference in the GPU average discharge rate suggests that the GPU rendering cases do not have the same rate of energy growth. If all three acceleration structures can complete a model in the same amount of the time, the results should show that the GPU UG uses the least amount of energy. The GPU UG does match up closely with the energy consumption of the GPU BVH in Figure 23, 24, 27 and 28. Knowing that the UG is the cheapest acceleration structure to build, GPU UG might be the best method for rendering most dynamic scenes.

5.3 Dynamic Rendering Energy Measurement

Dynamic rendering tests use scenes, screen resolutions and acceleration structures from static rendering tests. The two models are the Toaster model from the Utah Animation Repository and the Dragon model from the Stanford 3D Scanning Repository. Dynamic scenes have a typical execution cycle consisting of updating the triangles, building the acceleration structure and rendering. These models are not animated, and the triangles do not update from frame to frame. The acceleration structures are set to rebuild per frame even if there are no changes in the scene. This emulates the effect of running an animated scene without the overhead of updating all the triangles. We do this to avoid adding the energy used to update the triangles into the energy measurement, as it will be hard to distinguish the energy used to update the triangles from the build and rendering. Also, the amount of energy used to update the triangles is small when compared to the build and the rendering. Finally, identifying the energy used to update the triangles is not likely to result in additional information. Therefore, the method of forcing the acceleration structure to rebuild is chosen to represent the dynamic rendering energy.

Like the static rendering tests, the tests were divided into GPU and CPU rendering tests. They are further divided into 6 tests with different acceleration structures. The following is a list of all the acceleration structures with their short-hand notations and descriptions:

UG-A – Uniform grid with the triangle-box intersection.

UG-N – Uniform grid without the triangle-box intersection.
KD-M – Naïve Kd-tree (spatial median split)
KD- S – SAH Kd-tree
BV- F – BVH that rebuild per frame
BV- U – BVH that updates every node and does not rebuild

BV-U represents the BVH update worst case scenario from Section 5.1. It forces all leaf nodes to execute the update function. Since the triangles in the scene do not move, BV-U will not produce a different tree topology. The acceleration structure will always rebuild itself and the rendering algorithm will always traverse the same structure. The camera is set to rotate around the model. As discussed in Section 5.2.1, the camera should be fixed to generate a more accurate energy measurement. Since the resolution of the energy measurement function is 3-5 seconds, the camera should not rotate before the energy sampling function samples the energy at a particular camera angle for more than 5 seconds. Future work using the same energy sampling function should take this into consideration.

We hypothesize that the dynamic rendering energy on the CPU will represent the addition of the build and the static rendering energy. The energy per frame on the GPU, on the other hand, will not be equal to this simple addition because GPU rendering ray tracer requires an additional process before the GPU can render the image, specifically, the translation of data in array format into texture. Described in Section 3.1.3, the triangles and the acceleration structures need to be converted into texture so the GPU algorithm can use the textures as random access memory. In addition, textures are not created in the GPU memory, and therefore need to be copied from the CPU memory to the GPU memory. This step requires additional time and energy. The translation into texture and the transfer of the texture into the GPU memory will be referred as the *transfer* in the following section. The energy and time used by the *transfer* will be referred as transfer energy and transfer time. We hypothesize that the transfer process can become a major bottleneck, causing additional energy usage when rendering with the GPU.

The average discharge rate of dynamic rendering is measured by looping the build and static rendering functions. The energy per frame is calculated by the multiplication of the average discharge rate and the time per loop. The energy per build found in section 5.1 is subtracted from the energy per frame, meaning that the remainder is the energy per rendering. The transfer energy can be obtained with subtraction as well.

Transfer energy per frame = energy per frame – energy per build – energy per render

The equation to determine transfer energy only applies to the GPU data. Rendering on the CPU does not require the transfer step. The accuracy of this simple subtraction can be checked by adding the energy found in build tests (section 5.1) to that found in the static rendering tests (section 5.2). We found the measured and the calculated results to be similar but not the same. This is expected because we are using averages. The standard deviations and average discharge rates for the dynamic rendering test can be found in Appendix A.

5.3.1 11K Model Measurement Results

The data is normalized to the worst data, the data with biggest energy per frame, across the screen resolutions. Figures 32 and 33 provide the data for the 11k model in two screen resolution.

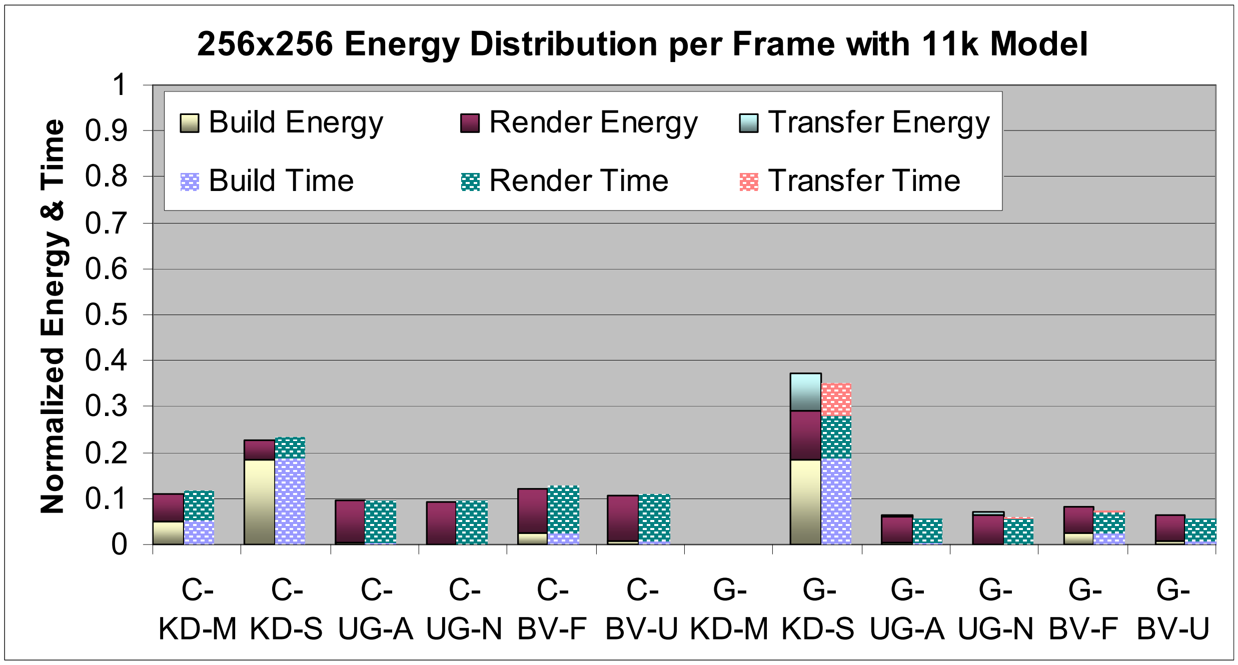


Figure 32 Normalized Energy and Time Chart for 11k Model at 256x256 Resolution

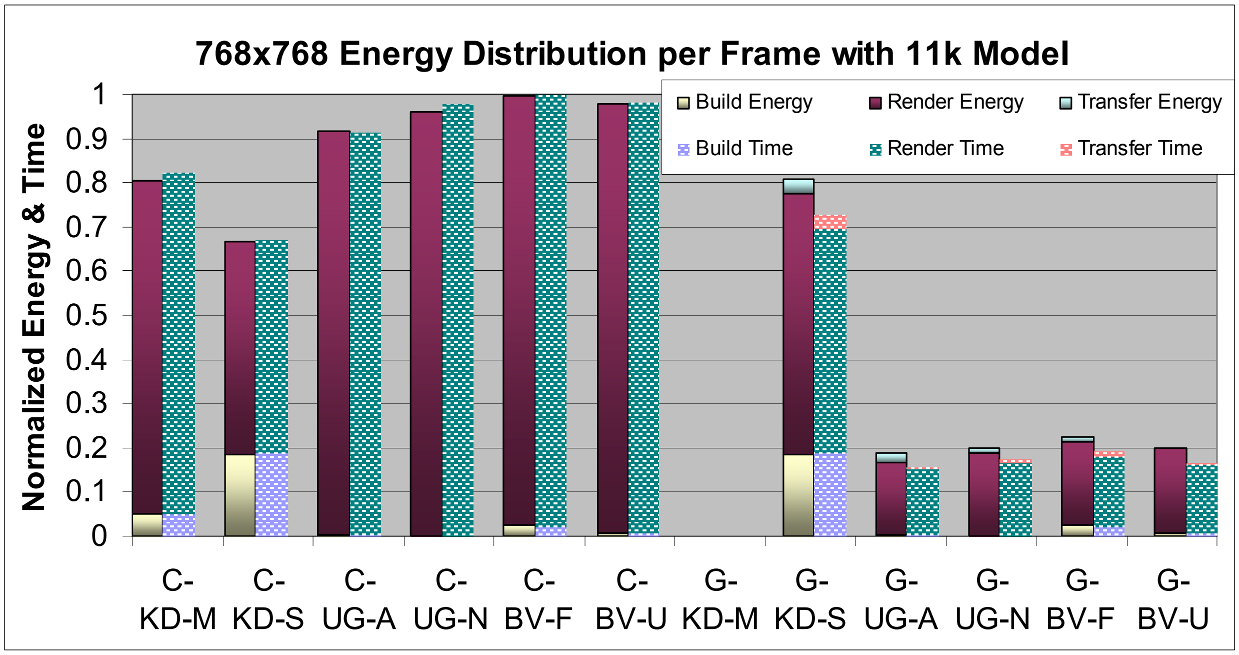


Figure 33 Normalized Energy and Time Chart for 11k Model at 768x768 Resolution

The stack represents data with the following order from top to bottom: transfer, render and build. The CPU results do not include transfer energy and time so there are only two stacks. The dotted bar represents time and the solid bar represents energy. From Figures 31 and 32, it can be seen that the time and energy columns are parallel, suggesting that energy increases as time increases. From Section 5.2.3, we know that rendering on the GPU has a higher discharge rate. This is reflected here again because the GPU energy bars are usually higher than the time bars, whereas the CPU energy bars have same height as the time bars. This suggests if both CPU and GPU rendered a scene in the same amount of time, the CPU would use less energy than the GPU.

Kd-Tree is not the best option on the CPU platform because of the high build energy requirement; however, it is still the best CPU option at the 768x768 screen resolution where the rendering energy is significantly higher than the build energy for all CPU tests. The *transfer* process does not have a significant impact on the GPU energy consumption as we had hypothesized. It uses some additional energy but not enough to penalize the GPU performance on all the GPU tests. Overall, GPU-BVH and GPU-UG are the best combination in rendering the 11k model with per frame acceleration structure rebuilds.

From Figures 32 and 33, we can see that build energy contributes to less than 20% of the energy when rendering a dynamic scene. The majority of the energy and time are used to render the image. The only exception is Kd-Tree where the build takes more energy and time than rendering the image at the 256x256 screen resolution. This suggests that balancing the energy used to build the acceleration and the rendering is important. Attempting for an extreme amount of power conservation on one end of the scale can cause more energy consumption on the other end. The Kd-Tree data for 256x256 screen resolution is a good example of this. It is reflected in Table 8 in Section 5.2.1, where the coarse KD-M build shows an energy saving of 18.5 Joules per frame. Figure 32 confirms this because KD-M uses less energy per frame than KD-S.

5.3.2 48K Model Energy Measurement Results

The results are normalized to the highest energy per frame across both screen resolutions for the 48k model. Figures 34 and 35 represent the results for the 48k model.

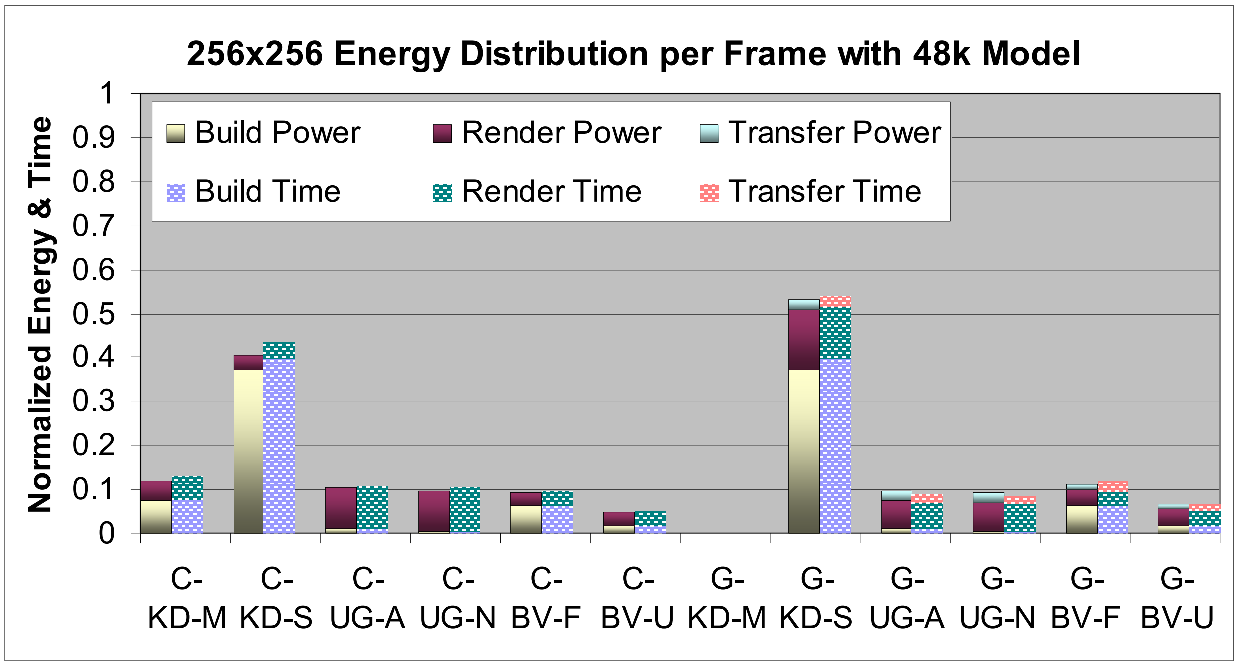


Figure 34 Normalized Energy and Time Chart for 48k Model at 256x256 Resolution

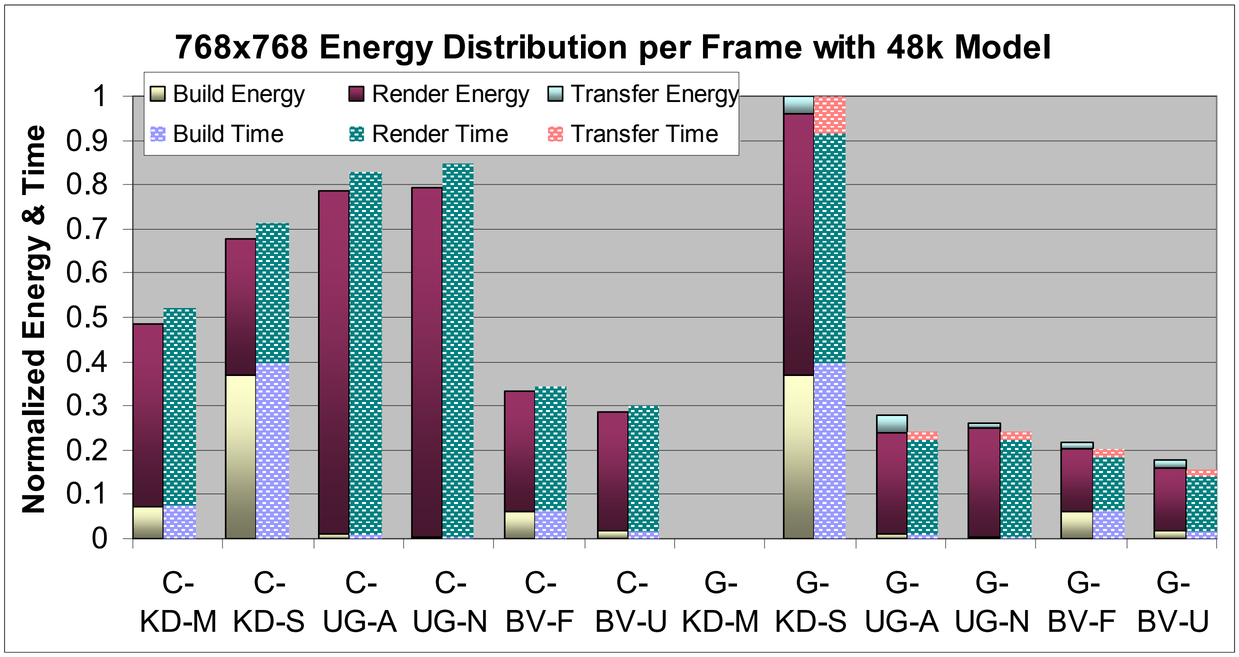


Figure 35 Normalized Energy and Time Chart for 48k Model at 768x768 Resolution

In term of the 256x256 resolution, the CPU BV-U uses the least amount of the energy per frame. Unlike the 11k model result, the BVH build for this model is more expensive and needs more energy than the rendering on the lower resolution. Despite this, BVH still uses the least amount of energy overall in both screen resolutions. The KD-M uses less energy than the KD-S in both resolutions, verifying the finding in Table 9.

There is no all-round best acceleration structure for the CPU when rendering a dynamic scene. The 48k data shows that CPU BVH uses the least amount of energy while the 11k model performs best using the CPU Kd-Tree. The CPU UG looks promising at the lower resolution but does not scale well on the higher resolution. GPU BVH and GPU UG remains the most scalable solution with consistent performance. GPU BVH can usually finish a frame faster with the updating algorithm. Thus, we conclude that GPU BVH is the best choice for single-ray ray tracing on dynamic scenes in general.

5.3.3 Dynamic Rendering Energy Measurement Discussion

The average discharge rate for all the CPU dynamic rendering results is 34.2 J/S ± 0.37 J/S. This discharge rate is the same as the static rendering results in Section 5.2.3 because their confidence intervals overlap. The average discharge for all the GPU results is 39.1 J/S ± 0.97 J/S. The discharge rate for the GPU has a larger variant because of the large difference in discharge rates between the build and the GPU rendering. Figure 36 provides a graph of the dynamic rendering discharge rate for BVH at 256x256 screen resolution.

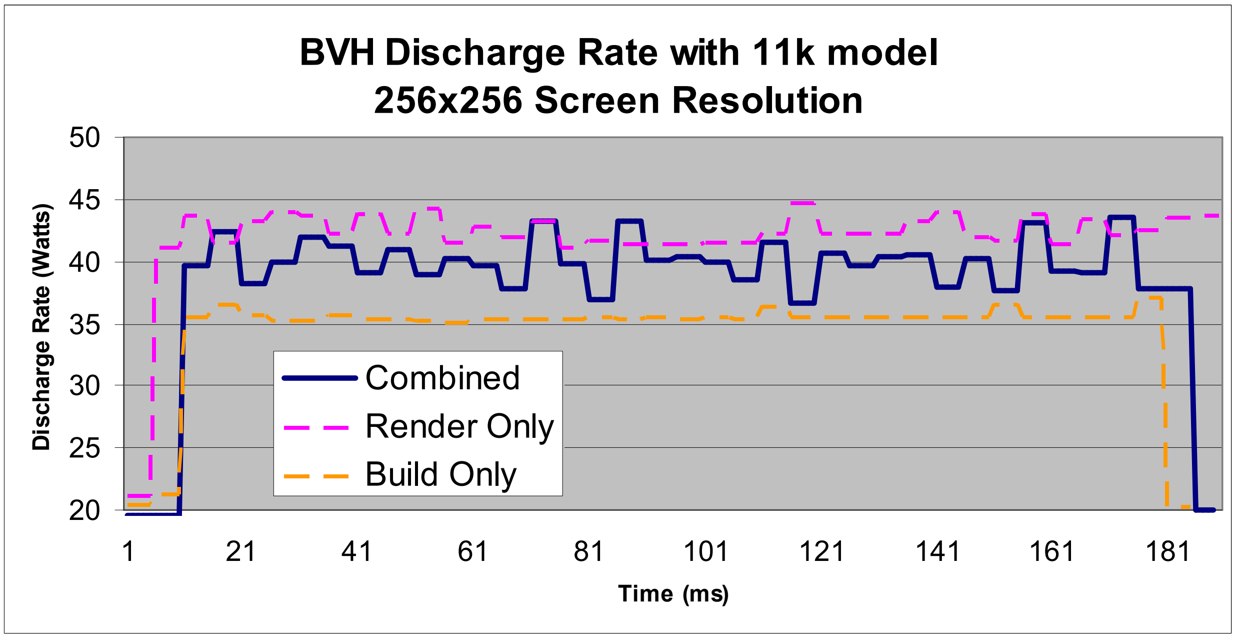


Figure 36 Discharge Rate Comparison for 11k Model Rendered with BVH

Figure 36 shows that the discharge rate of the dynamic test is between the build and the render discharge rates. We suspect that the combined discharge rate is the addition of render and builds at different biases. GPU BVH in Figure 36 spends 200 ms on rendering and 100 ms on building and the combined discharge rate is closer to the rendering line. This leads us to believe that the combined average discharge rate is biased towards rendering because the rendering takes more time. If this is true, the discharge rate when rendering a dynamic scene with the GPU can be approximated with the following equation:

The average discharge rate = cpuJ*Bt/(Bt+Rt) + gpuJ*Rt/(Bt+Rt)
where
Bt = build time, Rt = render time
cpuJ = build discharge rate, gpuJ = render discharge rate

The above discharge rate approximation ignores transfer power. A more accurate method should incorporate the discharge rate and run time during the transfer process. Further tests are needed in future work to confirm this discharge rate approximation for the GPU.

Rendering with the GPU requires additional energy for the GPU to complete its execution of the shaders. As a result, more energy was used, and the uneven column between time and energy on the GPU bars in Figure 33 are good evidence of this. It might be intuitive to think that using additional energy to drive more devices on a PC consumes more energy; however, it is shown that even with additional energy fed into the GPU, it is still more energy efficient to render a single-ray ray traced scene with the GPU because the benefit of a shortened runtime outweighs the increase in discharge rate.

5.4 Discussion of System and Experiment Limitations

The experiment results suggest that energy increases as running time increases. This is reasonable because less work equals less energy. However, we feel that the results do not reveal the true energy consumption in the algorithms at which we have looked. The data reveal the average discharge of the algorithm but do not disclose the energy usage trends during the execution of the algorithm. We do not know which parts of the operation inside the algorithm use more or less energy. Since the CallNtPowerInformation function offers limited resolution, we cannot obtain energy consumption information for each individual component of the algorithm. An energy measurement tool with a finer resolution is required if we are to further understand energy behavior when the system executes an algorithm.

Another way to solve this problem would necessitate finding the energy expands of different CPU operations individually. The base case is the energy needed when running the empty loop. Each individual operation in an algorithm can then be added gradually to observe the changes in the discharge rate.

The results in the static rendering and the dynamic rendering sections are interesting, but more data is needed to verify the finding. We only have data for two different models in two different resolutions. More models need to be tested to confirm that GPU BVH is really the best overall rendering method for dynamic scene. Furthermore, the testing scenes need to include more models. Real world 3D applications typically have more than ten models on the screen at once. It is more meaningful to test scenes that match closely with real world scenes than rendering a single complex model. In the experiment, each test runs once for three minutes and we do not run the test again. The same test should run multiple times so that the information on the variance between each run can be gathered.

Another possible error is the rotating camera. Mentioned in Section 5.2 and 5.3, the camera should be kept still for the rendering test. The rotating camera introduces more variance to the rendering energy because each frame is not rendered in the same amount of time after the camera position changes. The variance in the rendering energy per frame causes inaccuracy in the calculation for transfer energy because the transfer energy is derived from the rendering energy. Figures 32, 33, 34 and 35 show the transfer energy; however, we cannot derive too much information from the data because we suspect that they do not correctly represent transfer energy. We will only mention on the trend that suggests that transfer energy does not significantly increase the energy usage on the GPU.

Lastly, the rendering data discussed in this paper only applies to traditional single-ray ray tracing. Our implementation is not the fastest, but it is sufficient for energy measurement purposes. The SIMD implementations published in recent papers will likely speed up the rendering process. Utilizing SIMD instructions on the CPU might produce a different discharge rate, and this is an interesting topic for future work.

6 Modeling Energy Usage

Knowing that time can translate directly to energy, we can predict the energy consumption during ray tracing if the average discharge rate is known. Assuming the algorithms, both build and rendering, have similar CPU utilization, the average discharge rate and the frame per second of the ray tracing application are enough to predict the energy per frame. The following is an example of the process:

Assuming the following data is gathered when running dynamic scenes.
Let CJ = current battery capacity in Joule
cpuJ = discharge rate of CPU at 100% utilization
gpuJ = discharge rate of GPU when running shaders
tranJ = discharge rate when transferring data to GPU
Bt = build time, Rt = render time, Tt = transfer time
Fps = frame per second
Fpj = frame per Joule
Jpb = joule per build
Jpr = joule per render

When rendering on the CPU:
Bt and Rt are measured in seconds.
Fps = 1/(Bt+Rt)
Jpb = cpuJ*Bt, Jpr = cpuJ*Rt
Fpj = 1/(Jpb+Jpr) = 1/(cpuJ*(Bt+Rt)) = Fps * 1/cpuJ = Fps/cpuJ

The frame per Joule is simply the frame per second divided by the average discharge rate.
We used CPU UG data with the 11k model data rendered at 256x256 screen resolution to verify this equation.

Measured variables:
Bt = 0.02s, Rt = 0.4s
cpuJ = 34 Watts

Derived values:
Fps = 2.38
Fpj = 2.38 / 34 = 0.07

CPU UG needs 14.3 Joule per frame and the calculated value is 1/0.07 = 14.28 J, which is fairly close to the measured value. The maximum number of frames that a system can render with its current energy is CJ*Fpj. If the confidence interval is known for the average discharge rate, this can be used to derive the interval for the calculated Joule per frame.

The energy needed when rendering on the GPU can be calculated with the frame per second divided by the average discharge rate as well. This is not as accurate as the CPU because the GPU discharge rate has a larger variance. The calculation for the average discharge rate is more involved but more error prone in the GPU case. The average discharge rates for the build, GPU rendering and transfer data to the GPU need to be measured. Their corresponding runtimes are required as well. Knowing these values, we can extend the equation in Section 5.3.3 to include the transfer energy and time. The equation is as follows:

average discharge rate = Rate = cpuJ*Bt/(Bt+Rt+Tt) + gpuJ*Rt(Bt+Rt+Tt) + tranJ*Tt(Bt+Rt+Tt) and Fpj = Fps/Rate

The equation calculates the overall system energy consumption and it does not take into account the other miscellaneous tasks running on the system. The equation will likely fail if there are other applications sharing resources with the ray tracer application.

Lastly, the equation is a hypothesis and needs to be validated on different models and on different machines. We do not validate the equation in this paper but this is a necessary experiment in future work.

7 Conclusion and Future Work

7.1 Conclusion

After testing the energy usage of three popular acceleration structures for ray tracing, and the energy usage when rendering dynamic scenes using the CPU and the GPU as rendering platforms, we found time to be the dominant factor affecting energy usage, and the trend is that energy increases as time increases. We have shown rendering on the GPU with the BVH uses the least amount of energy in most of our test cases. We found that dynamic scenes are mostly bottlenecked by rendering performance. However, balancing the build time and the render time is essential for saving energy. The Kd-Tree is not well suited for rendering dynamic scenes because the building of the Kd-Tree is not fast enough. We show that the GPU can render scenes faster than the CPU with less energy at higher screen resolution. However, a good acceleration structure on the CPU is comparable to the GPU implementation at lower screen resolutions. In addition, we found that our GPU implementations do not fully utilize the GPU, and that more work can be done to improve the efficiency of our GPU algorithm. With this finding, we firmly believe that the GPU will continue to play an important role in boosting global-illumination algorithm performance and extending the battery life on mobile devices in the future.

7.2 Future Work

We feel more work can be added to extend our work. First, we will discuss topics that will make our work more complete. Second, we will talk about ray-tracing implementation improvements, and lastly, other possible future work. In our experiments, we did not have enough data samples for build-energy measurement for the non-triangle-box uniform grid; this should be included in the future to complete the build data. The GPU Kd-Tree is not implemented fully, as it is not able to render every scene correctly, and it should be fixed in the future. The equation in Section 6 needs to be validated through further data collection, and we hope to include this verification in the future. As mentioned in Section 6, we do not have enough data samples for the static and dynamic rendering tests and more samples are needed to validate our claim. More samples are needed for the uniform grid triangle-box intersection test as well. More samples should make the result statistically more accurate.

The implementation can be improved to better utilize the CPU and the GPU. We feel the ENCORE implementation is not complete, as it can only do ray casting, and the ability to render reflections, shadows and refractions should be added. The CPU ray tracer can be improved by implementing the SIMD traversal and SIMD ray-triangle intersection testing to speed up the rendering. In addition, the rending image should be broken up into smaller tiles to allow better cache efficiency on both the CPU and GPU. We can possibly speed up the building by incorporate threading [Lauterbach et al. 2006]. There are also papers looking at hybrid approaches to building the Kd-Tree [Havran et al. 2006]. Coherent ray tracing, supporting multiple rays per pixel, or multi-level ray tracing is also interesting future work.

In terms of energy measurements, a more-accurate measurement might be obtainable through a programmable multimeter that can record data through a USB port to a PC. With the aid of profilers, we can look at different elements, such as cache usage, memory allocation patterns, and CPU-instruction usage to aid in the understanding of energy behavior. Each acceleration structure can be examined in more detail by comparing the data structure topology, triangle density in the scene, and triangle access patterns. We can test other acceleration structures such as hierarchical uniform grids and octrees. Scenes more closely resembling commercial 3D applications could be used in addition to our simple ones. The energy consumption of a multithreaded SIMD ray tracing algorithm running on a dual-core CPU is interesting as well. The list of possible future work is infinite, but we hope our work inspires new research and continued interest in ray tracing and energy aware computing.

References

Akenine-Moller, T. 2001. Fast 3D triangle-box overlap testing. *J. Graph. Tools* 6(1), 29-33.

Amanatides, J. and Woo, A. 1987. A Fast Voxel Traversal Algorithm for Ray Tracing. In Eurographics '87. Eurographics Association, 3-10.

Bikker, J. 2005. "Raytracing: Theory & Implementation Part 4, Spatial Subdivisions", www.devmaster.net/articles/raytracing_series/, 2005. (Accessed Feb 21 2006).

Buck, I. 2004. "Brook for GPUs", SIGGRAPH 2004, talk slides.

Carr, Nathan A., Hall, Jesse D. and Hart, John C. 2002. The Ray Engine. Proc. Graphics Hardware 2002, Sep. 2002

Carr, Nathan A., Hoberock, J., Crane, K. and Hart, John C. 2006. Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images, In Proc. Graphics Interface, 2006.

Christen, M. 2005. Ray Tracing on GPU, Masters thesis, University of applied sciences, Basel, 2005.

Foley, T. and Sugerman, J. 2005. KD-tree acceleration structures for a GPU raytracer, in Proc. SIGGRAPH/Eurographics Workshop on graphics hardware 2005, pp. 15-22.

Glassner, Andrew S. 1989. An Overview of Ray Tracing. *An Introduction to Ray Tracing*. Morgan Kaufmann.

Haines, E. 1999. Quicker Grid Generation via Memory Allocation, Ray Tracing News, Volume 12, Number 1, 1999

Havran, V., Prikryl, J., Purgathofer, W. 2000. Statistical comparison of ray-shooting efficiency schemes, Tech. report TR-186-2-00, Institute of Computer Graphics, Vienna University of Tech, July 2000.

Havran, V., Herzog, R. and Seidel, H.-P. 2006. On Fast Construction of Spatial Hierarchies for Ray Tracing. *Submitted to RT'06*, 2006.

Karlsson, F. and Ljungstedt, Carl J. 2004. Ray tracing fully implemented on programmable graphics hardware, masters thesis, Chalmers university of technology, 2004.

Lauterbach, C., Yoon, S.-E., Tuft, D. and Manocha, D. 2006. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. University of North Carolina at Chapel Hill, 2006

Moller, T. and Trumbore, B. 1997. Fast, minimum storage ray triangle intersection. *JGT* 2(1), 21-28.

Pharr, M. and Humphreys, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2004.

Purcell, Timothy J., Buck, I., Mark, William R., and Hanrahan, P. 2002. Ray Tracing on Programmable Graphics Hardware, in Proc. SIGGRAPH 2002, 703 - 712.

Starner, Thad E. 2003. "Powerful change part 1: batteries and possible alternatives for the mobile market", IEEE Pervasive Computing, Vol 2, No. 4, Oct.-Dec. 2003 Pages:86 - 88.

Thrane, N. and Simonsen, Lars O. 2005. A Comparison of Acceleration Structures for GPU Assisted Ray Tracing, Masters thesis, University of Aarhus, 2005.

Wald, I. 2004. Realtime Ray Tracing and Interactive Global Illumination, PhD thesis, Saarland University, 2004.

Wald, I. and Havran, V. 2006. On building good kd-trees for ray tracing, and on doing this in O(N log N). Tech. Rep. UUSCI-2006-009, SCI Institute, University of Utah.

Wald, I., Ize, T., Kensler, A., Knoll, A. and Parker, Steven G. 2006. Ray Tracing Animated Scenes using Coherent Grid Traversal, in ACM Transactions on Graphics, 2006, 485 – 493.

Wald, I., Boulos, S. and Shirley, P. 2006a. Ray Tracing Deformable Scene using Dynamic Bounding Volume Hierachies. *ACM Transactions on Graphics (conditionally accepted, to appear)*.

Appendix A

Average Discharge Rate and Standard Deviation Data

ADR = average discharte rate (mW)
STDEV = standard deviation for ADR

Table – build standard deviation, part 1

	Uniform grid		Kd-Tree (SAH)		Kd-Tree (median)	
	ADR	STDEV	ADR	STDEV	ADR	STDEV
singleTri	33119	218	34901	238	35176	216
Toaster	33733	714	34539	191	35054	262
Sdragon	33716	507	34053	216	34628	176
Bbunny	34871	316	33758	351	34939	448
Complex	32801	453	33068	349	33926	421
200k	32786	397	33848	393	34446	342
400k	32351	421	33135	642	33917	382
Fairy	33913	258	35075	357	35613	317
990k	34504	299	33862	846	34912	514

Table – build standard deviation, part 2

	BVH		BVH update (best case)		BVH update (average case)		BVH update (worst case)	
	ADR	STDEV	ADR	STDEV	ADR	STDEV	ADR	STDEV
singleTri	34669	332	34705	265	34303	1428	34217	583
Toaster	34600	288	33751	346	34742	295	34787	374
Sdragon	33622	275	32473	390	34494	289	34219	413
Bbunny	33172	302	32010	329	33960	389	33480	364
Complex	31818	430	30718	260	32649	474	32275	365
200k	33032	579	32143	245	34106	296	33914	482
400k	32539	322	31636	320	33594	374	33014	322
Fairy	34161	469	33077	296	35269	288	34983	400
990k	33642	523	32575	180	34751	342	34471	342

M – no memory pool emulation is done. The data structures calls malloc function or new operator when new memory is needed and releases them when deleted.
N – memory pool emulation is in effect. The data structures reuse previous allocated spaces.

Table – memory test build standard deviation

	Kd-Tree (M)		Kd-Tree (N)		BVH (M)		BVH (N)	
	ADR	STDEV	ADR	STDEV	ADR	STDEV	ADR	STDEV
Scissor	32902	382	33078	1180	34131	354	34535	495
Wheel	34886	332	34051	478	33803	240	34003	273
Mug	34406	395	33699	279	33497	293	33757	293

Cow	34270	238	33477	667	32968	310	33255	306
Porsche	32646	365	32090	299	32617	493	32940	308

C-KD-M	= CPU Kd-Tree with median split
C-KD-S	= CPU Kd-Tree with SAH split
C-UG-A	= CPU uniform grid with triangle-box intersection test
C-UG-N	= CPU uniform grid without triangle-box intersection test
C-BV-F	= CPU BVH without update
C-BV-U	= CPU BVH update
G-	= GPU

Table – Build Data for Combine Test

		C-KD-M	C-KD-S	C-UG-A	C-UG-N	C-BV-F	C-BV-U
11k							
	ADR	33806	33838	33838	32820	35560	34553
	STDEV	226	1342	131	335	417	346
48k							
	ADR	34250	34371	34770	33803	35920	34973
	STDEV	433	412	246	260	434	282

Table – Render Data for Combine Test, CPU

		C-KD-M	C-KD-S	C-UG-A	C-UG-N	C-BV-F	C-BV-U
256x256							
11k	ADR	34037	35256	35304	33770	32731	
	STDEV	321	167	384	264	362	
256x256							
48k	ADR	34380	35662	35430	34537	33059	
	STDEV	265	303	398	591	152	
768x768							
11k	ADR	34068	34971	34927	34219	34722	
	STDEV	293	253	298	435	523	
768x768							
48k	ADR	34448	35275	34868	34319	35168	
	STDEV	280	209	153	279	201	

Table – Render Data for Combine Test, GPU

		G-KD-M	G-KD-S	G-UG-A	G-UG-N	G-BV-F	G-BV-U
256x256							
11k	ADR		40387	37475	38282	40693	
	STDEV		1305	906	845	1089	
256x256							
48k	ADR		41788	38280	39938	41258	
	STDEV		2777	712	873	2908	
768x768							
11k	ADR		40659	38001	39561	42450	

	STDEV		1090	484	839	1146	
768x768							
48k	ADR		41716	39269	41016	42540	
	STDEV		1380	1022	1678	1030	

Table – Combine Test Data, CPU

		C-KD-M	C-KD-S	C-UG-A	C-UG-N	C-BV-F	C-BV-U
256x256							
11k	ADR	33559	33784	33852	33915	34609	31512
	STDEV	339	431	361	227	404	497
256x256							
48k	ADR	34493	34387	34064	34242	35391	32654
	STDEV	361	508	357	418	590	326
768x768							
11k	ADR	35739	34245	34215	33918	34807	33783
	STDEV	315	331	339	389	689	356
768x768							
48k	ADR	35970	34501	34315	34276	35276	34393
	STDEV	520	410	334	578	719	585

Table – Combine Test Data, GPU

		G-KD-M	G-KD-S	G-UG-A	G-UG-N	G-BV-F	G-BV-U
256x256							
11k	ADR		36894	39428	39575	36919	37950
	STDEV		2822	912	1376	1352	1025
256x256							
48k	ADR		36254	39869	39878	34663	37029
	STDEV		1783	1998	1416	988	1503
768x768							
11k	ADR		38617	42627	39362	40944	41242
	STDEV		2611	1509	581	1792	2744
768x768							
48k	ADR		36763	42478	40018	39306	42158
	STDEV		3012	2040	968	3122	1565

Appendix B

ENCORE Interface Code

Only the interfaces for acceleration structure (AccelerationStructure) and renderer are included. The implementation files (.C files) are not shown for these header files

```
/***************
* AccelerationStructure Class
***************/
class AccelerationStructure
{
public:

    virtual ~AccelerationStructure() {}

    // usage:
    // prints out the options that the AccelerationStructure can read from
       the config file
    virtual void usage( void ) = 0;

    // configure:
    // set of options that are read in from the config file
    virtual void configure( Options* l_pOptions ) = 0;

    // build:
    // builds the AccelerationStructure from the Scene
    virtual void build(std::list<IModel*> &modelList) = 0;

    // update:
    // implement this if the AC can update itself
    // otherwise, this method simply call build method
    virtual void update(std::list<IModel*> &modelList);

    // buildGPU:
    // builds the accelerationStructure into the GPUAccelerationStructureData
       reference
    virtual void buildGPU(std::list<IModel*> &modelList, std::list<Triangle*>
    &triangleList, GPUAccelerationStructureData& l_pASD ) = 0;

    // setGPUParameters
    // set the shader parameters that the accel struct needs to pass in
    virtual void setGPUParameters( CShader& l_Shader,
    GPUAccelerationStructureData& l_ASD ) = 0;

    // intersect:
    // returns the HitInfo of the first successful intersection
    // of the ray with the Primitives in the AccelerationStructure
    virtual HitInfo intersect( Ray& l_Ray ) = 0;

    // keyboard:
    // defines how the AcceleratioStructure should react to keyboard input
    virtual void keyboard( unsigned char key ) = 0;
};
```

```
/***************
* Renderer Class
***************/
class Renderer
{
public:

    Renderer();
    virtual ~Renderer();

    // usage:
    // prints out the options that the Renderer can read from the config file
    virtual void usage( void ) = 0;

    // init:
    // uses the Scene and the Camera to build the AccelerationStructure
    // along with any data structures needed for the renderer
    virtual void init( Scene* l_pScene, AccelerationStructure*
    l_pAccelStruct, Camera* l_pCamera ) = 0;

    // configure:
    // a map of options and values that are read in from the config file
    virtual void configure( Options* l_pOptions ) = 0;

    // render:
    // renders the scene
    virtual void render( void ) = 0;

    // deinit:
    // removes any AC that were built during the running of the Renderer
    // turns off any OpenGL options that were needed to render
    virtual void deinit( void ) = 0;

    // keyboard:
    // defines how the Renderer should react to keyboard input
    virtual void keyboard( unsigned char key ) = 0;

protected:
    Scene* m_pScene;
    Camera* m_pCamera;
};
```

Appendix C

Uniform Grid Shader Code

The shader code for Traversal and intersection is shown below. The shader codes for eye ray generation and Phong light are omitted.

```
// voxel .w : 0 mean keep going, 1 mean done
// hitinfo  : x = t, y = u, z = v, w = index
struct stepinfo
{
    float4 hitinfo  : COL0;
    float4 voxel    : COL1;
    float4 tMax     : COL2;
};

//code adapted from thrane's thesis paper
// determine if the ray-box intersect
bool ray_box_intersect(float3 rayD, float3 gmin, float3 gmax, float3 eyePos,
out float t_hit)
{
    float3 tmin, tmax;

    tmin = (gmin-eyePos)/rayD;
    tmax = (gmax-eyePos)/rayD;

    float3 r_min = min(tmin, tmax);
    float3 r_max = max(tmin, tmax);

    float minmax = min(min(r_max.x, r_max.y), r_max.z);
    float maxmin = max(max(r_min.x, r_min.y), r_min.z);
    t_hit = maxmin;
    return minmax > maxmin;
}

// return correct voxel index in the correct range
float3 getvoxelindex(float3 p, float3 gmin, float3 gridsize, float3 len)
{
    return clamp(floor((p-gmin)/len),float3(0.0,0.0,0.0), (gridsize-
    float3(1.0,1.0,1.0)));
}

// return the voxel intersect by the ray
stepinfo getvoxel(float3 rayD, float3 gmin, float3 gmax, float3 eyePos,
float3 cell_width, float3 resolution, out float time)
{
    stepinfo o;
    float t;
    float3 gridOrig = eyePos;
    o.tMax = float4(INF(),INF(),INF(), 1);

    // if the ray hit a cell in the grid
    if( ray_box_intersect( rayD, gmin, gmax, eyePos, t ) )
    {
        gridOrig = time > 0.0f ? eyePos+rayD*t : gridOrig;
        o.voxel.xyz = getvoxelindex(gridOrig, gmin, resolution, cell_width);
        o.voxel.w = 0;
    }
    else
    {
        // ray outside of box, so it'll never intersect
        o.tMax.w = -1;
    }

    time = t > 0.0f ? t : 0;
    float3 cell_min, cell_max;

    cell_min = gmin+cell_width*o.voxel.xyz;
    cell_max = cell_min + cell_width;

    float3 t1 = (cell_min-gridOrig)/rayD;
    float3 t2 = (cell_max-gridOrig)/rayD;

    float3 p = sign(rayD.xyz) == float3(1,1,1);
    float3 n = sign(rayD.xyz) == float3(-1,-1,-1);

    // calculate tMax for the intersected cell
    o.tMax.xyz = t1*n + t2*p;

    if(rayD.x < EP && rayD.x > -EP) o.tMax.x = INF();
    if(rayD.y < EP && rayD.y > -EP) o.tMax.y = INF();
    if(rayD.z < EP && rayD.z > -EP) o.tMax.z = INF();

    // add the time from ray origin to the uniform grid
    o.tMax.xyz += time;

    return o;
}

// v0, v1, v2 = 3 vertex of triangle
// rayD, rayStart is self explain
// lasthit is a hit from last intersect that is valid
// index is index of this triangle
// adapted from thrane's thesis paper
float4 intersect(float3 v0, float3 v1, float3 v2, float3 rayDir, float3
rayStart, float4 lasthit, float index)
{
    float3 edge1, edge2;
    float3 pvec, tvec, qvec;
    float det, inv_det, t, u, v;

    edge1 = v1 - v0;
    edge2 = v2 - v0;

    pvec = cross(rayDir, edge2);
    det = dot(pvec, edge1);
    bool isHit = det > EP;

    inv_det = 1/det;

    tvec = rayStart - v0;
    u = dot(pvec, tvec)*inv_det;

    qvec = cross(tvec, edge1);
    v = dot(qvec, rayDir)*inv_det;
    t = dot(qvec, edge2)*inv_det;

    isHit = (u >= 0) && (v >= 0) && (u+v <= 1.0)
             && (t > 0.0) && (t < lasthit.x) ;

    return isHit ? float4(t,u,v,index) : lasthit;
}

// cell_info contain index to triangle, number of triangle, 0 ,0
// voxel status: 2 mean there is a hit, 0 mean has not been traversed at all,
1 mean its out of bound, dont check
stepinfo main(
    uniform samplerRECT rayDirMap,
    uniform samplerRECT rayStartMap,
    uniform samplerRECT cellData0,
    uniform samplerRECT hitInfoMap,
    uniform samplerRECT trav0Map,
    uniform samplerRECT trav1Map,
    uniform samplerRECT v0t,
    uniform samplerRECT v1t,
    uniform samplerRECT v2t,
    uniform float len,
    uniform float gridsize,
    uniform float gmin,
    uniform float gmax,
    uniform float maxloop,
    float2 texc : TEXCOORD0)
{
    stepinfo o;

    float4 timeInfo = texRECT( trav1Map, texc );
    // x = tMax x
    // y = tMax y
    // z = tMax z
    // w = -1: finished
    //      0: initial state
    //      1: traversing or intersecting

    // finished, then don't process
    if ( timeInfo.w == -1 )
        discard;

    // get some information
    float3 rayD = texRECT(rayDirMap, texc).xyz;
    float3 resolution = float3(gridsize,gridsize,gridsize);
    float3 cell_width = float3(len,len,len);

    // figure out step direction and boundary
    float3 eyePos = texRECT(rayStartMap, texc).xyz;
    float3 step = sign(rayD);
    float3 delta = abs(cell_width/rayD);
    if(rayD.x < EP && rayD.x > -EP) delta.x = INF();
    if(rayD.y < EP && rayD.y > -EP) delta.y = INF();
    if(rayD.z < EP && rayD.z > -EP) delta.z = INF();
    float t;

    // if first pass, then generate some data
    if ( all( timeInfo == float4(0,0,0,0) ) )
    {
        o = getvoxel(rayD, gmin, gmax, eyePos, cell_width, resolution, t);
    }
    else
    {
        // read it in from textures
        o.tMax = timeInfo;

        o.voxel = texRECT( trav0Map, texc );
        // x = voxel x
        // y = voxel y
        // z = voxel z
        // w = 0 : initial state (traversing)
        //     1+: intersecting with index w-1 triangle
    }

    // hit information is always read in from texture
    o.hitinfo = texRECT( hitInfoMap, texc );

    float maxloops = 2500;
    float3 v0,v1,v2;
    float2 index;

    while(o.tMax.w != -1 && maxloops > 0)
    {
        // find the correct texture index
        index.x = o.voxel.x + o.voxel.y*gridsize +
        o.voxel.z*gridsize*gridsize;
        index.x = modf(index.x*0.000244140625, index.y)*4096;

        // format : triangle index, triangle count,0,0
        float4 info = texRECT(cellData0, index);

        // if there are triangles in this voxel
        // then intersect with them
        float start = info.x + o.voxel.w;
        float end = info.x+info.y;
        while(start < end && maxloops > 0) // start intersect test
        {
            index.x = modf(start*0.000244140625, index.y)*4096;

            v0 = texRECT(v0t, index).xyz;
            v1 = texRECT(v1t, index).xyz;
            v2 = texRECT(v2t, index).xyz;
            o.hitinfo = intersect(v0,v1,v2,rayD,eyePos,o.hitinfo,start);
            start++;
            maxloops--;
        } // end intersect test

        // do we still have loops left?
        if ( maxloops > 0 )
        {
            float tMin = min( o.tMax.x, min( o.tMax.y, o.tMax.z ) );

            // we have a hit
            if ( o.hitinfo.w >= 0 )
            {
                // did not check against the tMax in the cell
                // possible to return incorrect hit time but rare
                o.tMax.w = -1; // for now indicate we are done
            }

            //traverse
            float3 mask = float3(tMin, tMin, tMin) == o.tMax.xyz;

            // update voxel and t value
            o.voxel.xyz = o.voxel.xyz + step*mask;
            o.tMax.xyz = o.tMax.xyz + delta*mask;

            // find out if we stepped outside the grid
            float3 lt = o.voxel.xyz >= resolution;
            float3 gt = o.voxel.xyz < float3(0.0,0.0,0.0);
            if(any(lt) || any(gt))
                o.tMax.w = -1;
        }

        maxloops--;
    }

    return o;
}
```