ENCORE: Energy-Conscious Rendering for Mobile Devices

Chen-Hao Chang, Peter J. Lohrmann, Emmanuel O. Agu, and Robert W. Lindeman

***Abstract*—The recent emergence of programmable GPUs on mobile devices means that mobile ray tracing at interactive rates is imminent. Previous work has focused mainly on speeding up real-time ray tracing. However, on a mobile device, the most constraining resource is frequently its available battery energy. In addition to maintaining reasonable frame rates and image quality, ray tracing on mobile devices must also be energy efficient. We present results of a comprehensive measurement study that investigates the energy efficiency of mobile ray tracing. We compared the energy efficiency of uniform grid, k-d tree, and bounding-volume hierarchy (BVH) acceleration structures on scenes ranging in size from 1lk to 990k triangles when rendered on both the CPU and GPU. Our results show that from an energy perspective, several scene characteristics such as the number and distribution of triangles, mobile display size, and the rendering processor (CPU vs. GPU) can affect which acceleration structure is most energy efficient. With the exception of the SAH k-d tree, the build energy of all acceleration structures was much smaller than the rendering energy, making the SAH k-d tree a bad choice energy wise for highly dynamic scenes. For small screen sizes (e.g., cell phone resolutions), rendering on the CPU using a naïve k-d tree uses less energy than any other processor-acceleration structure combination. On the GPU, the BVH is the most energy-efficient acceleration structure for larger screen sizes (e.g., PDA and laptop resolutions), regardless of model size.**

***Index Terms*— Computer Graphics, Energy Efficiency, Real-Time Ray Tracing, Acceleration Data Structures, Spatial Data Structures**

I. Introduction

GRAPHICS on mobile devices is becoming popular because mobility increases the productivity of workers. Additionally, many graphics applications are more convenient when users are untethered. Global illumination rendering algorithms such as ray tracing are essential for photorealism. With advances in Graphics Processing Units (GPUs), ray tracing at interactive rates has become feasible. However, previous real-time ray tracing research has mostly used rendering speed (time) as the main measure of *efficiency* [1], [2]. On a mobile device, available battery power is frequently the most limiting resource because, while processor speeds and available memory have grown exponentially from 1990-2000, battery energy capacity has only doubled [3]. As mobile devices begin to incorporate GPUs, mobile real-time ray tracing is imminent, and photorealism and rendering speed must be balanced with the *efficient use of limited battery power*.

The Energy-Conscious Rendering (ENCORE) project is focusing on balancing three competing attributes of successful mobile graphics applications: image quality, frame rate, and energy consumption. Our initial focus is to measure how algorithm design and implementation choices affect energy consumption during ray-tracing. Using empirical data collection, we compare the energy efficiency of ray tracing on the CPU and GPU with uniform grids, k-d trees, and bounding-volume hierarchies (BVH) while varying scene complexity and display size. Understanding how these factors affect energy consumption can inform rendering trade-offs on mobile devices.

Two important trends motivated this work. First, the phenomenal growth rates of GPU technology has made real-time ray tracing feasible. Secondly, because it has been estimated that nearly half of the total energy consumption of mobile devices is due to the display and graphics hardware [4], it is imperative that graphics software be optimized to reduce energy consumption while taking advantage of the GPU's capabilities.

To the best of our knowledge, this is the first comprehensive study comparing the *energy consumption* of both CPU and GPU ray tracing using the most popular acceleration structures. We found the BVH to be the most energy-efficient acceleration structure when rendering on the GPU, regardless of the number of triangles in the scene. On an energy-per-frame basis, the GPU was more efficient than the CPU especially for larger scenes. Also, with the exception of the k-d tree, the build energy of all acceleration structures was much smaller than the rendering energy. Finally, for small screen sizes (e.g., cell phone resolutions), rendering on the CPU was more energy-efficient than the GPU and the k-d tree was the best performing acceleration structure. The BVH on the GPU is the most efficient for larger screen sizes (e.g., PDA and laptop resolutions). The remainder of the paper is structured as follows. Section 2 discusses related work, Section 3 gives required background, Section 4 describes our experimental setup, Section 5 presents our results, Section 6

Manuscript received September 24, 2007.

C.-H. Chang was with Worcester Polytechnic Institute, Worcester, MA 01609 USA (e-mail: ninjitaru@gmail.com).

P. J. Lohrmann was with Worcester Polytechnic Institute, Worcester, MA 01609 USA. He is now with the GPG Developer Tools Group, AMD, Marlborough, MA 01752 USA (phone: 508-303-3996; e-mail: Peter.Lohrmann@amd.com).

E. O. Agu is with the Computer Science Department, Worcester Polytechnic Institute, Worcester, MA 01609 USA (e-mail: emmanuel@wpi.edu).

R. W. Lindeman is with the Computer Science Department, Worcester Polytechnic Institute, Worcester, MA 01609 USA (e-mail: gogo@wpi.edu).

summarizes our conclusions, and Section 7 discusses future work.

A. Fundamental System Energy Relationships

A mobile device's battery-drain rate depends on several factors, including the energy consumption of hardware components and peripherals, and the operations performed by running applications. The total energy expended is measured in Joules, while the rate at which the battery energy is drained (also called *Power*) is measured in Joules per second (or Watts). Thus, the total energy consumed by an application can be expressed as:

$$\text{Total energy (Joules)} = \text{Average drain rate (Watts)} * \text{Running time (Sec.)} \quad (1)$$

To determine the average drain rate in Eq. 1, the battery drain rate was sampled every second while executing our ray tracer, and then averaged over the entire time interval the application was run. **Figure 1** shows the battery drain rate for both OpenGL and our ray tracer running on the same GPU for five minutes. Our tests show that a laptop running a GPU-intensive task consumes energy at a rate of about 45 Watts compared to the laptop's idle drain rate of 30 Watts.

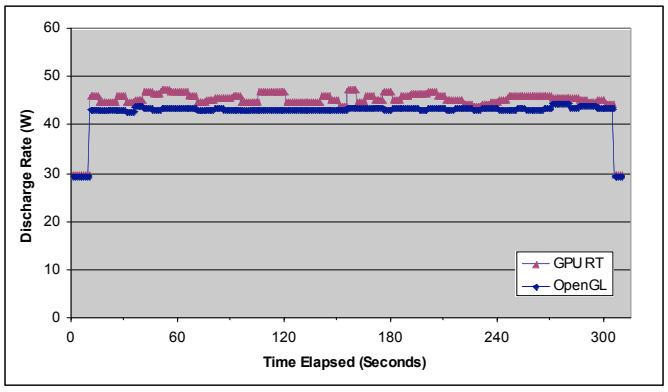


Figure 1: *Comparing Battery Discharge Rate of OpenGL Rendering to Ray Tracing on the GPU*

On a given processing unit, the total energy consumed depends mostly on the average drain rate of the rendering technique. OpenGL utilizes the fixed-functionality of the GPU, has a lower average drain rate, and renders more frames per unit time. It consumes less energy per frame and less total energy based on Eq. 1 than the GPU ray tracer, at the expense of image quality. This shows that the rendering algorithm also affects the average drain rate since different algorithms have different constituent operations and spend different fractions of time performing CPU/GPU operations, memory accesses, and screen updates. We decompose ray tracing into build, transfer, and rendering steps in order to understand their energy implications, and hence make optimizations.

II. Related Work

While graphics research has not focused on optimizing rendering algorithms for power consumption, the measurement and tracking of energy usage is an active area of research within the mobile-computing community. PowerScope is a hardware tool developed by Flinn and Satyanarayanan [5] that enables easy power measurement on mobile devices. While PowerScope uses an external hardware multimeter to measure energy consumption, PowerSpy, a software-only solution proposed by Banerjee and Agu [6], profiles the energy usage of the various threads, hardware components, and functions of mobile software applications. Barr and Asanovic [7] optimized the energy consumption of data compression, by focusing on reducing the number of bytes and hence the energy consumed in wireless data transfer. Several authors have proposed hardware techniques for measuring graphics card *power measurement* [8], *energy management* [9], network card energy *consumption* [10], and operating system *accounting techniques* to prolong battery life [11]. Several authors have also proposed either new energy-efficient components for GPUs [12], [13] or novel energy-efficient hardware architectures for graphics [14]-[16].

III. Background

To compare the energy consumption on both the CPU and GPU, the ENCORE system allows an acceleration structure and a rendering processor (CPU vs. GPU) to be selected at runtime. Variations of several common acceleration structures have been implemented, including two variants of the uniform grid (UG), two variants of the k-d tree, and two variants of the BVH.

A. Ray Tracing on the GPU

Using the GPU for ray tracing first appeared in 2002 with the Ray Engine [17], where the GPU was used solely for performing ray-triangle intersection tests. Purcell *et al.* [18] proposed the first GPU-based ray tracer, which decomposed the ray tracing process into four GPU fragment-shader kernels, each handling a different aspect of the process: eye-ray generation, traversal, intersection, and shading. The approach was successful, but limited at the time because GPUs lacked loop logic. ENCORE uses the ray tracing implementation described by Purcell *et al.* [18], but with a unified traversal-intersection shader [2]. Two other implementations of GPU-based ray tracers [19], [20] used the idea of kernels, but added new acceleration structures, such as a proximity cloud UG [19], reporting a 37-50% speed up for some scenes.

The k-d tree implementation from Foley and Sugerman [21] outperformed GPU-UG implementations on scenes with high variation in triangle density, but did not achieve performance comparable to CPUs. The GPU-BVH implementation by Thrane and Simonsen [2] showed the BVH to be the best acceleration structure for GPUs, outperforming other structures by a factor of nine in some cases.

Although current GPUs allow for conditional statements and looping, and claim to support infinite-length shaders in Shader Model 3.0, some significant limitations exist compared to CPUs. The most prominent limitation is the maximum number of loop iterations (256) that can be performed in a single execution of a shader before it is forced to stop. To get around this, Thrane and Simonsen [2] suggest nesting a loop inside another loop with the same termination conditions. In practice, however, we found that the outer loop was only allowed to execute ten times, providing 2,560 iterations in a single pass. This was sufficient for most of our tests, but prevented rays from fully traversing the k-d tree for large scenes. It is possible to work around this issue by saving some state information, and using

occlusion queries to determine whether or not the shader should be executed again [2], [18], [21].

A second GPU limitation is the maximum size of texture dimensions (4,096x4,096 on the nVidia GeForce Go 6800 used in our tests), which sets the maximum number of triangles per scene at 5.6 million, and also limits the size of acceleration structures. Though this limitation did not affect our work, it could become an issue if we attempted to ray trace more-detailed scenes.

Finally, the primary mechanism for tree traversal on the CPU, the stack, does not exist on the GPU. Foley and Sugarman [21] present two alternative methods for GPU traversal of a k-d tree, *k-d-restart* and *k-d-backtrack*. Only the k-d-restart algorithm was implemented for our tests because of the required memory overhead of the k-d-backtrack algorithm.

B. ENCORE Acceleration Structures

An acceleration structure is a means of partitioning scene geometry in order to simplify accessing it for computations. Assuming highly dynamic scenes, two main steps are required at each frame: rebuilding of the acceleration structure, and traversing the structure for rendering. Implementation on the GPU requires the additional step of transferring the resulting acceleration structure from main memory to texture memory on the GPU.

ENCORE builds the selected acceleration structure on the CPU, regardless of where rendering will take place. Our GPU-based ray tracer is implemented using fragment shaders, with some minor control flow (based on occlusion queries) performed by the CPU. For formatting triangle data on the GPU, we used an approach similar to Purcell *et al.* [18], where three textures are used to store the first, second, and third vertices of each triangle; we encode the acceleration structure in one additional texture. On both the CPU and GPU, ENCORE uses the ray-triangle intersection test initially described by Woo [22] and improved by Moller and Trumbore [23].

1) Uniform Grid

Purcell *et al.* [18] proposed the original organization of a GPU-based ray tracer and the approach assumes the use of a uniform grid (UG) as an acceleration structure. For building the UG, we use the approach described by Bikker [24], with the memory-allocation optimizations suggested by Haines [25]. Our implementation divides each edge of the scene bounding-box into an equal number of segments. Havran *et al.* [1] suggest that given n triangles, using $\sqrt[3]{d*n}+0.5$ segments along each axis, with scene density d (commonly with $d=1$), provided a reasonable voxel resolution across a wide variety of scenes. We use this formula, and the triangle-box intersection method described by Akenine-Möller [26] to assign scene triangles to appropriate voxels. The traversal algorithm is the same for both the CPU and GPU, and is based on that presented by Amanatides and Woo [27].

2) k-d Tree

Havran *et al.* [1] showed that k-d trees are statistically among the best acceleration structures based on the number of traversals and intersections that are performed. Foley and Sugerman [21] implemented a k-d tree-based GPU ray tracer and reported rendering times up to eight times faster than with a UG. They used an optimized ray-tracing engine that reduced memory usage, and an improved surface-area heuristic (based on [28]) for constructing a more-optimal k-d tree. The ENCORE k-d tree implementation is based on Pharr and Humphreys [29], without their memory-allocation optimizations. We implemented both spatial median split (SMS) and surface-area-heuristic (SAH) [30]. We have found that the SMS method can build 2-4 times faster than the SAH approach, but that the SAH often produces more-balanced trees. We used a combination of maximum tree depth and maximum triangle quota per leaf-node to terminate the building process [29]. Given n triangles, the maximum tree depth is set to 11 + 1.3 * log(n), and the maximum number of triangles allowed per leaf-node is two when n is less than 5k, and 10 otherwise.

3) Bounding-Volume Hierarchy

The bounding-volume hierarchy (BVH) is an object-space partitioning structure that partitions the geometry rather than the space. We use a top-down approach [31], where the bounding volume for the scene is recursively subdivided into bounding sub-volumes around partitioned geometry. Thrane and Simonsen [2] compared a BVH to k-d trees and uniform grids, and showed that it performed better on the GPU than the other structures. We based our BVH construction and CPU-BVH traversal implementation on the RT-DEFORM system [32], which shows promising performance of 5.6 FPS on a 70k-triangle animated model. We also implemented a variation of their BVH with updates, but rather than basing the decision to perform a complete rebuild on the movement of the triangles, we use a probabilistic method described later.

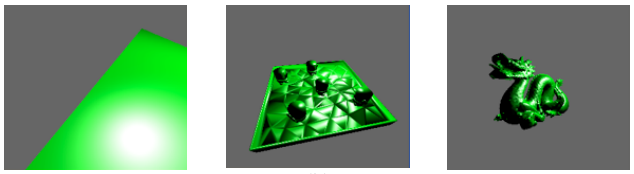
Like the k-d tree, there are different criteria for determining a good split axis. We use SMS and then accordingly shrink the resulting bounding boxes, and exit the build when a leaf-node encloses exactly one triangle. This results in a maximum of (n * 2) - 1 nodes. If a split location cannot split the triangles into left and right nodes, we keep the tree balanced by assigning one half of the triangles to the left node, the other half to the right, and then continue to recursively partition the scene.

IV. Experimental Setup

All tests were conducted under Windows XP, SP2, on a Dell Inspiron 9300 laptop, with an Intel Pentium-M 1.6GHz CPU, 1GB of DDR2-533 RAM, an nVidia GeForce Go 6800 (PCI Express 16x, 256MB VRAM) GPU, and a Dell Li-ion Battery (TYPE D5318) with a capacity of 53WH, rated at 11.1V and 4800mAh. For all the tests, the LCD display was set to 50% brightness. In addition, network devices were disabled to reduce extraneous power discharge. No other applications were active while the tests were running, though OS housekeeping was not controlled.

The Advanced Configuration and Power Interface (ACPI) [33], developed by a consortium of hardware and software manufacturers, provides battery usage information including discharge rates in mW. Microsoft Windows exposes an interface to ACPI through a function called *callNtPowerInformation*. To maintain accuracy without negatively impacting system performance, we sampled the battery drain rate once per second.

Our ray-tracing engine only used primary rays while ray tracing the scenes. When only sending one ray through each pixel into the scene, the number of rays is constant for a given screen resolution and the per-ray (or per-pixel) costs can be easily calculated and applied to estimate other effects, such as anti-aliasing or various material properties.



(a) 1 Triangle (b) 11k Triangles (c) 48k Triangles

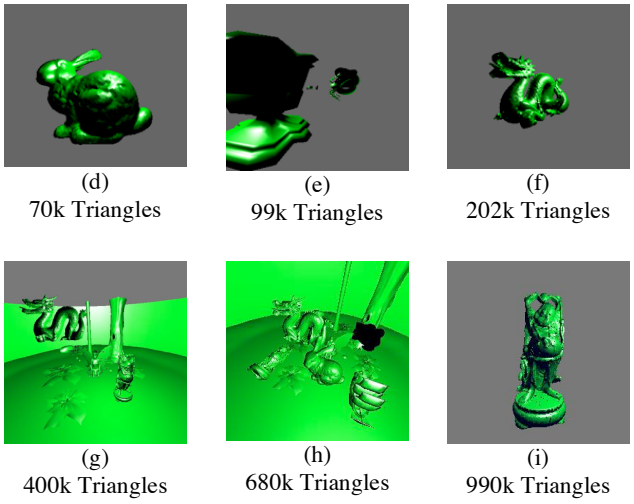


Figure 2: *Test Scenes.*

In order to remove any bias of one acceleration structure over another, we chose a set of nine sample scenes to run our comparisons (**Figure 2**). The set contains a range of numbers of triangles (1 to approximately 1 million), as well as scenes with the triangles tightly grouped (c, d, f & i) and more-uniformly distributed (e, g & h). The models and scenes chosen for testing were intended to provide various levels of complexity.

V. Results and Analysis

A. *Energy Consumption of Building Acceleration Structures*

As previously discussed, ray tracing using acceleration structures involves several distinct phases including building the acceleration structures, scene traversal, intersection calculation, and shading. In this section, we focus on comparing the energy required to build the uniform grid, k-d tree, and BVH acceleration structures. It should be noted that even when the GPU is used for rendering, the acceleration structure is built on the CPU, and then transferred to the GPU. Hence, we only measure the energy required to build the acceleration structures on the CPU. First, we investigate how energy usage varies with increasing model size, and then describe two optimizations to improve energy usage.

Figure 3 shows *energy spent per build* for the UG, BVH, and k-d tree (using SAH) on a logarithmic scale as the number of scene triangles increases. Larger scenes generally consume more energy to build the acceleration structures. The UG has the lowest energy consumption for all scene sizes. For scenes with fewer than 50k triangles, the BVH and UG have similar energy usage. For scenes with over 100k triangles, the BVH's build energy is about twice that of the UG. The k-d tree's build energy is about 20 times the UG's build energy for all scene sizes.

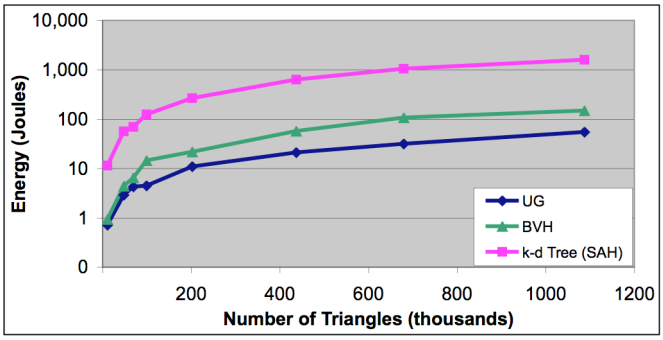


Figure 3: *Energy Consumed (in Joules) per Build for Each Acceleration Structure*

Next, we compute the *build energy on a per-triangle basis* by dividing the total build energy consumed by the number of scene triangles. **Figure 4** plots the build energy (Joules) consumed per triangle by each acceleration structure. In addition to the naïve (SMS) k-d tree build strategy, we also include results for the k-d tree build using the SAH, as well as results for the BVH update.

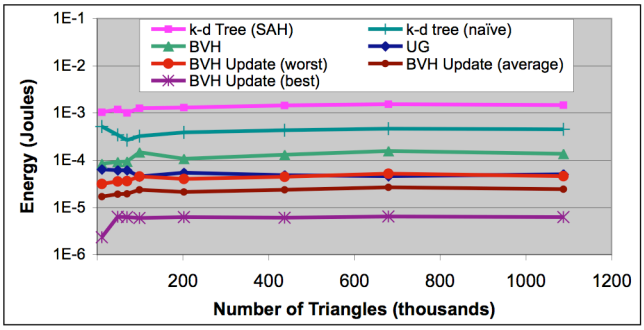


Figure 4: *Joules Spent per Triangle Versus Model Size*

The RT-DEFORM algorithm [32] for dynamic scenes avoids the large time and energy overheads associated with a complete BVH rebuild by simply updating the BVH if object motion has affected relatively few BVH nodes. The energy required to update the BVH depends on how many nodes require updates and not on the number of triangles in the scene. Thus, variance in scene geometry, as well as object trajectory and speed in dynamic scenes, can lead to a wide variance in the BVH update energy. We implemented RT-DEFORM and measured its energy consumption when 60% of its nodes were randomly chosen and updated (average case), no nodes updated (best case) and all nodes updated (worst case). The results in **Figure 4** show that the build energy spent per triangle is almost constant for all tested acceleration structures. The k-d tree build using the SAH has the worst energy usage per triangle, probably due to the surface area computations required. The naïve k-d tree uses 50% less energy than the SAH k-d tree. The BVH update displays the best energy savings. Its worst-case performance (every node has to be updated) is roughly equivalent to the UG and the best case (no updates required) uses 10 times less energy than the UG. However, it is important to bear in mind that the BVH update only works for deformable models, and occasionally a complete BVH rebuild is still required when the scene's triangle distribution changes beyond a certain threshold, incurring a higher energy consumption. Thus, considering model size alone, the UG is the most energy-efficient structure. Our build experiments had very little variance after many repetitions, with a standard deviation of less than 1%.

1) *Memory Management to Optimize Build-Energy Consumption*

While building acceleration structures, dynamic memory allocation is a fundamental operation. Our pilot experiments showed that naïve memory allocation could increase the build time of our acceleration structures. In this section, we attempt to reduce the energy consumed by memory allocation. Specifically, we reduced the number of system-level memory allocations by pre-allocating larger blocks of memory (known as memory pooling).

A typical UG build requires dynamic memory allocation when inserting scene triangles into their associated cells. The k-d tree and BVH have recursive build functions that require memory allocation to store triangles in leaf nodes. Each leaf node of the k-d tree also requires a dynamic array to hold the triangles that it references because there are an unpredictable number of triangles intersecting the leaf node's volume, which further complicates memory allocation. To mimic memory pooling, we exploit the fact that our tests on dynamic scenes repeatedly rebuild the test acceleration structures on the same model from frame to frame. We can therefore allocate memory once for the first build and reuse it during subsequent builds, thus reducing the need to re-allocate memory.

We found that reducing the frequency of memory allocation reduced energy consumption in all cases (**Figure 5**). We found that memory pooling especially benefited acceleration structures that allocated memory frequently during their build. The BVH graph on the left of **Figure 5** shows consistent improvements as the model size increases. Our experiments showed that compared to the BVH, the k-d tree typically allocates smaller blocks of memory about 200 times more frequently than the BVH. Hence, the k-d tree saved more energy by using memory pooling to aggregate these small memory allocations. The k-d tree's memory allocation size and frequency depends on the numbers of triangles in the scene as well as their distribution. Hence, in **Figure 5**, unlike the BVH, the k-d tree's allocated memory and the energy consumed are not proportional to the triangle count. The distribution of triangles in the 1,368-triangle scene caused the k-d tree to make a large number of memory allocation calls and ultimately saved almost 40% of its build energy by using memory pooling. While this result suggests a potential for significant savings from memory pooling, more careful study is required to fully understand the energy implications of memory access patterns, allocations, and de-allocations.

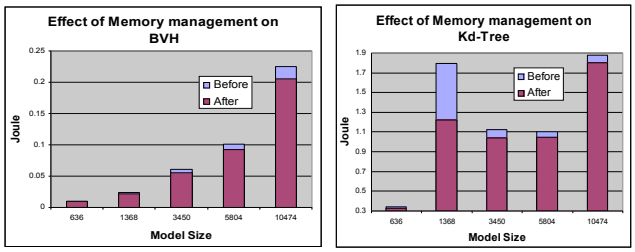


Figure 5: *Power Usage Reduction for k-d Tree and BVH*

2) *Triangle-Box Intersection to Optimize Uniform Grid Build-Energy Consumption*

The triangle-box (tri-box) intersection is computed during the UG build and is thus only relevant to the UG. A profiler showed that over 60% of the running time of a UG ray tracer was spent in the tri-box intersection function. We became interested in determining how the number of tri-box intersections increase as the model size increases and how this affects the energy per build.

We found a linear relationship with a slope of about 3 Joules of energy consumed for every 100k increase in the number of tri-box intersections. This implies that a uniform grid implementation that reduces or eliminates tri-box intersections will save energy in a linear fashion. We used a lazy build, that removes tri-box intersection tests, on two models, **Figure 2**b (11k triangles) and **Figure 2**c (48k triangles), which show an energy reduction of about 57% and 62% respectively. This is a significant energy savings that should be considered if the speed penalty incurred by rendering with this lazy build does not outweigh the energy saved by using it.

B. *Comparison of Acceleration Structure Energy Consumption for Static Rendering*

In this section, we compare the rendering energy efficiency of the UG, k-d tree, and BVH acceleration structures on both the CPU and GPU. Since we were interested in the energy consumed during the rendering of static scenes, in the following tests, acceleration structures are built once and only the rendering portion was repeated throughout the test duration. We consider screen sizes that we feel are representative of cell phones, personal digital assistants (PDAs), and laptops. Currently, cell phones with programmable GPUs have screen resolutions of 240x320. We can approximate expected results at this resolution based on our results of ray tracing at a resolution of 256x256. PDAs with programmable GPUs typically have a resolution of 480x640, which is just slightly larger than our test resolution of 512x512. Finally, we use our largest test resolution of 1024x1024 to approximate a laptop.

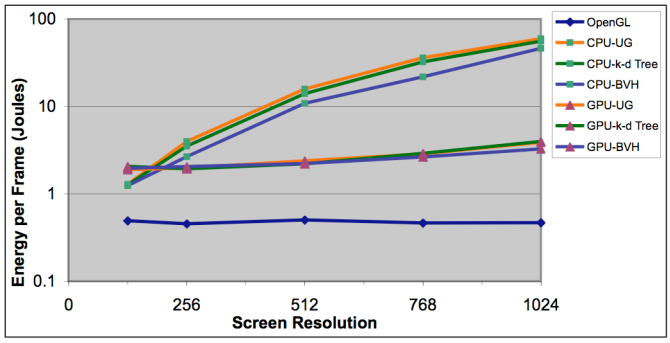


Figure 6: *Energy Cost per Frame of Rendering a Single Triangle at Various Resolutions*

Ray tracing a single triangle: Looking at *energy cost per frame* for rendering a single triangle (**Figure 6**), the main factor that affects the energy consumption is the choice between the CPU and GPU. In fact, for such a simple model, this GPU vs. CPU choice impacts energy efficiency more than the choice of acceleration structure. The second most significant factor is screen size. At a resolution of 128x128 the CPU was more energy efficient than the GPU. However, at resolutions of 256x256 and above, the GPU becomes the more-efficient processing unit and is increasingly more energy efficient than the CPU as screen size increases. At the highest resolution of 1024x1024, the acceleration structures on the CPU consumed roughly 10 times the amount of energy as their GPU counterparts. The energy cost of ray tracing on the CPU at a resolution of 256x256 is comparable to that of the GPU at a resolution of 1024x1024. Although the resolution is 16 times the size, the efficiency of the GPU allows the energy cost to be similar, and renders more frames in the same amount of time.

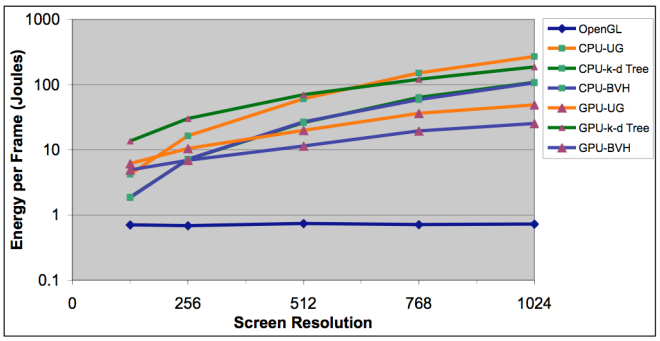


Figure 7: *Energy Cost per Frame of Rendering 48k Triangles*

Ray tracing 48k triangles: Figure 7 shows the energy cost per frame for ray tracing the model in **Figure 2c** (48k triangles). For this model size, the choice of CPU vs. GPU and screen size do not dominate as much as the single triangle case, and the choice of acceleration structures is now more important. On the CPU, the UG has fallen off from the BVH and k-d tree implementations, which now have almost identical energy costs. For rendering this model, the CPU-based BVH and k-d tree were the most efficient acceleration structures for resolutions less than 256x256, and the GPU-based BVH was the most efficient for larger resolutions. Apart from the k-d tree, there still exists a crossover where the GPU becomes more energy efficient than the CPU, however this crossover is less dramatic than before. Tests on larger models showed similar trends.

C. *Effects of Mobile Device Screen Size on Rendering Energy Efficiency*

In order to better target a particular mobile device, we now compare *each acceleration structure and processing unit* over the set of test scenes for a given resolution. Here, acceleration structures were built once and only the actual rendering was repeated throughout the test duration. Considering screen resolution, we aimed to identify an efficient acceleration structure depending on whether we are rendering on a cell phone (256x256), PDA (512x512), or laptop (1024x1024). While it is understood that the specific configuration of mobile devices may affect our results, measuring energy on a single device (laptop) while varying screen size establishes a common basis for direct comparisons at the *algorithmic* level. We now summarize our results. Due to space constraints, figures are only shown for our PDA results.

Figure 8 displays the *energy cost per frame* of rendering at a resolution of 512x512. There is an increasing trend in energy cost as the number of triangles increases, most evident for the CPU-based implementations, and on the GPU-UG. The energy consumed by the GPU-k-d tree differs drastically depending on the scene. The GPU-BVH, interestingly, has almost constant energy consumption, even as scene size increases. The BVH was more efficient than all structures for the larger scenes. Despite the ~900% increase in triangles between the 11k and 99k models, the GPU-BVH's energy consumption was almost constant.

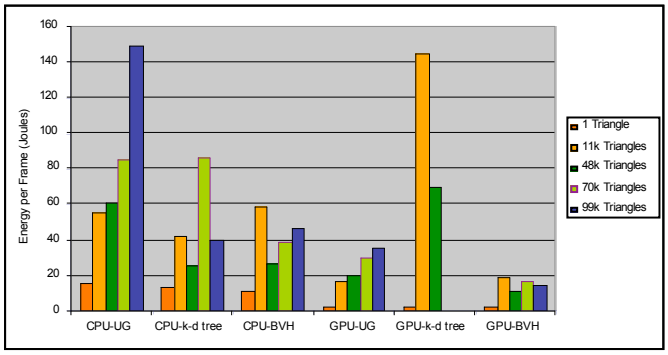


Figure 8: *Comparing Energy Cost per Frame of Rendering at a Resolution of 512x512*

Cell phone and laptop resolutions showed similar trends as the PDA. In general, the energy consumption of the UG was strongly correlated with the triangle count. The energy consumption of the k-d tree and BVH were additionally affected by the distribution, density and location of triangles in the scene. Finally, for larger meshes, the GPU BVH was the most energy efficient.

D. *Dynamic Scenes*

Dynamic scenes have objects whose positions change from frame to frame, requiring the acceleration structure to be either updated or completely rebuilt. For such scenes, the energy saved by fast rendering must be balanced with the energy overhead associated with rebuilding the acceleration structure as the objects move. Thus, we are interested in the *combined energy consumption of both the build and rendering stages* of ray tracing.

For our dynamic scene tests, we used two models: the 11k model (**Figure 2b**) and the 48k model (**Figure 2c**). Rather than animate the triangles, as in a real dynamic scene, we simply forced a fresh rebuild between consecutive frames in order to isolate the energy required for the rebuild from the energy required to animate the scene. We tested three acceleration structures in two variations each, for a total of six, in order to gauge the impact of acceleration-structure-specific optimizations. The six structures tested on both the CPU and GPU were the UG with tri-box intersection test (UG-A), UG without tri-box intersection test (UG-N), k-d tree with SMS (KD-M), k-d tree with the SAH (KD-S), BVH with a rebuild every frame (BV-F), and BVH that updates every node, but never rebuilds (BV-U). BV-U is similar to the previous approach to simulate BVH updates; each node is updated between frames based on a given probability.

Figure 9 and **Figure 10** are results for our dynamic scene experiments on the 11k and 48k models. We performed our tests at 256x256 and 768x768 resolutions. The KD-M GPU results are again missing. As expected, the results for the CPU (C) are simply the sum of the build and render energies for each acceleration structure on the CPU. The GPU (G) results, on the other hand, incur some extra overhead, since the CPU scene data has to be converted to a GPU-friendly format and transferred to GPU texture memory. Hence, the results for the GPU include the build energy on the CPU, the data conversion and transfer energy from the CPU to the GPU, and the energy required to ray trace the scene on the GPU.

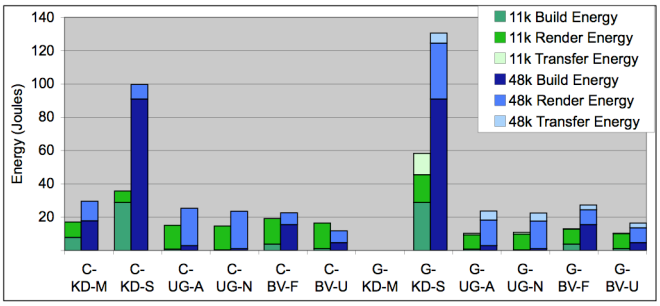


Figure 9: *Energy Plots for Dynamic 11k and 48k Models at 256x256 Resolution*

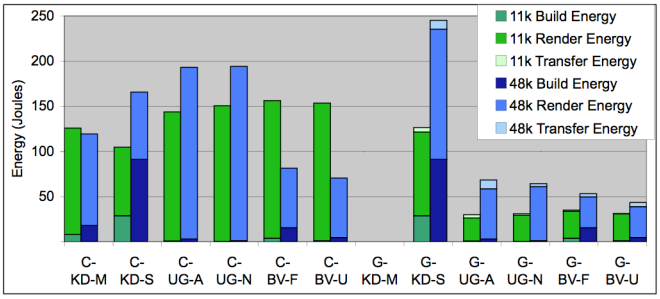


Figure 10: *Energy Plots for Dynamic 11k and 48k Models at 768x768 Resolution*

Comparing **Figure 9** and **Figure 10**, we find that for dynamic scenes, except for the k-d tree, the build energy required for all acceleration structures is much less than the rendering energy. Hence, dynamic scenes suggest a balance between energy saved from a fast rendering stage versus the energy required to build the acceleration structure. This suggests that as the amount of motion in scenes increases, the k-d tree should be avoided due to its high rebuild energy costs. The BVH performs best overall on both the CPU and GPU.

VI. Conclusion

CPU Vs GPU: State-of-the-art GPUs generally consume more power (energy per second) than comparable CPUs. Our measurements showed that the GPU generally consumed less energy per frame, especially for larger scenes. For *static* scenes (build once, render many times), the k-d tree was marginally the most energy-efficient acceleration structure on the CPU and the BVH was the most energy-efficient structure for rendering on the GPU. However, considering *dynamic* scenes (frequent rebuilds), the BVH was the most energy-efficient structure, especially on the GPU. The energy required to convert CPU data to GPU-friendly format and transfer from CPU memory to GPU texture memory is small (insignificant) compared to the energy required to build the acceleration structure or render the scene.

Scene Complexity: As scene complexity increased, the energy spent to build acceleration structures grew almost linearly and was correlated with running time. However, the energy consumption of *rendering* was not linearly related to scene complexity, but depended on the acceleration structure selected and the distribution of scene triangles.

Using memory pooling to reduce the frequency of memory allocation can especially benefit acceleration structures such as the k-d tree, which allocate many small chunks of memory during their build process. For the uniform grid, as scene complexity grows, using a lazy build to reduce the number of tri-box intersections should be considered. For instance, Wald *et al.* [34] used a lazy build to create a uniform grid and used mailboxing to avoid testing repeated triangles, hence avoiding penalties associated with the lazy build.

Screen size: For small screen sizes (<256x256), the CPU is more energy efficient for most data structures especially when rendering static scenes. For large screen sizes, the GPU is more energy efficient. This result suggests that from an energy perspective, and considering only the ray-tracing algorithm, larger mobile devices such as laptops would benefit more from GPUs than small devices such as cell phones. However, we note that in practice, the actual energy consumption of a given GPU depends on many factors and is unique to each device. Specifically, the energy consumption of a GPU on a handheld device may be proportionally smaller than the energy consumption of a GPU on a laptop, and overall may be the more energy-efficient ray tracing processor.

Scene Object Motion: Object motion influenced how frequently the acceleration structure had to be updated or rebuilt. Except for the k-d tree, the build energy of all acceleration structures was much smaller than rendering energy. For highly dynamic scenes that require frequent rebuilds, the k-d tree is a bad choice from an energy perspective especially on the GPU. Dynamic scenes require a balance between energy saved from a fast rendering stage versus the energy needed to build the acceleration structure.

VII. Future Work

Our current study is a first step in understanding the energy-efficiency of ray tracing. Many follow-up directions are possible. We would like to validate our results on actual cell phones, PDAs, and diverse mobile devices. A more in-depth comparison between the energy consumption of ray tracing and raster graphics (such as OpenGL) would be insightful. It would also be interesting to investigate the energy efficiency of popular real-time rendering techniques such as precomputation, lookups, and texture substitutions, since these techniques significantly increase memory accesses which have been noted to be energy hungry. We could investigate the energy-efficiency of reflections, refractions, more-sophisticated lighting, complex materials and other elements of photorealisic ray tracing. The energy consumption of subsurface scattering, shadow algorithms, photon mapping, and other physical phenomena can all be characterized. Another interesting direction is the development of models for predicting energy usage and ultimately for adaptive energy management algorithms. To make our dynamic scene results more realistic, animated scenes such as the BART scenes [35] can be used to trigger rebuilding of acceleration structures. The recently released shader model 4.0, particularly the geometry shader, will affect some of our implementations and hence energy consumption. We would also like to evaluate the energy efficiency of the recently proposed Bounding Instance Hierarchy [36]. Finally, more rigorous tests on the energy implications of memory access patterns and a lazy uniform grid build would be interesting.

Ray packets: Using ray packets, [37], we can now traverse several rays in parallel on the CPU using SIMD instructions, and still retain the robust logic control available on the CPU. Wald's implementation achieves four frames per second (FPS) for a static scene with 43k triangles, and two FPS for a dynamic case of the same model. In 2006, Wald *et al.* published a coherent grid traversal technique, which was able to achieve 29 FPS with pure ray casting, and seven FPS with full ray tracing effects on an 11k animated model [34]. We would like to characterize the energy implications of casting multiple rays per pixel, using ray packets.

References

[1] Havran, V., Prikryl, J., and Purgathofer, W. 2000. Statistical Comparison of Ray-Shooting Efficiency Schemes. Tech. Report TR-186-2-00-14. Inst. of Comp. Graphics, Vienna Univ. of Tech.

[2] THRANE, N. and SIMONSEN, L.O. 2005. *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*. Master's Thesis, U. of Aarhus, Denmark.
[3] STARNER, T. 2003. Powerful Change Part 1: Batteries and Possible Alternatives for the Mobile Market, *IEEE Pervasive Computing*, 2, 4, 86-88.
[4] INTEL. 2002. PC Energy-Efficiency Trends and Technologies. Retrieved Jan. 2007 from: http://cache-www.intel.com/cd/00/00/10/27/102727_ar024103.pdf.
[5] FLINN, J. and SATYANARAYANAN, M. 1999. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. *Proc. 2nd IEEE WMCSA*, pp 2-10.
[6] BANERJEE, K. and AGU, E. 2005. PowerSpy: Fine-Grained Software Energy Profiling for Mobile Devices. *Proc. of IEEE WirelessCom 2005*, 2, 1136-1141.
[7] BARR, K. and ASANOVIC, K. 2003. Energy Aware Lossless Data Compression. *Proc ACM MobiSys 2003*.
[8] TSCHEBLOCKOV, T. 2004. Power Consumption of Contemporary Graphics Accelerators. Retrieved Jan. 2006 from: http://www.xbitlabs.com/articles/video/display/ati-powercons.html.
[9] KRAVETS, R. and KRISHNAN, P. 1998. Power Management Techniques for Mobile Communication, *Proc. ACM Mobicom 1998*, 157-168.
[10] FEENEY, L. and NILSSON, M. 2001. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment, *Proc. IEEE InfoCom 2001*, 1548-1557.
[11] ZENG, H., ELLIS, C., LEBECK, A., and VAHDAT, A. 2002. ECOSystem: Managing Energy as a First-Class Operating System
[12] HENSLEY, J., SINGH, M., and LASTRA, A. 2005. A fast, energy-efficient z-comparator. Proc Graphics Hardware Graphics Hardware.
[13] SOHN, J., WOO, R., and YOO, H. 2004. A programmable vertex shader with fixed-point SIMD datapath for low power wireless applications. *Proc of Graphics Hardware*.
[14] EUH, J., CHITTAMURU, J., and BURLESON, W. 2005. Power-Aware 3D Computer Graphics Rendering. J. VLSI Signal Process. Syst. 39, 1-2
[15] KAMEYAMA, M., KATO, Y., FUJIMOTO, H., NEGISHI, H., KODAMA, Y., INOUE, Y., and KAWAI, H. 2003. 3D graphics LSI core for mobile phone "Z3D". Proc Graphics Hardware, 60-67.
[16] TACK, N. LAFRUIT, G. CATTHOOR, F. LAUWEREINS, R. A content quality driven energy management system for mobile 3D graphics, Proc. IEEE Workshop Signal Processing Systems Design and Implementation.
[17] CARR, N. A., HALL, J.D., and HART, J.C. 2002. The Ray Engine. *Proc. Graphics Hardware 2002*, 37-46.
[18] PURCELL, T, BUCK, I., MARK, W, and HANRAHAN, P. 2002. Ray Tracing on Programmable Graphics Hardware. *ACM Trans. Graphics*, 21, 3, 703-712.
[19] KARLSSON, F. and LJUNGSTEDT, C.J. 2004. *Ray Tracing Fully Implemented on Programmable Graphics Hardware*. Master's Thesis, Chalmers U. of Technology.
[20] CHRISTEN, M. 2005. *Ray Tracing on GPU*. Diploma Thesis, University of Applied Sciences, Basel.
[21] FOLEY, T. and SUGERMAN, J. 2005. KD-Tree Acceleration Structures for a GPU Raytracer. *Proc. SIGGRAPH/Eurographics Graphics Hardware,* 15-22.
[22] WOO, A. 1990. Fast Ray-Box Intersection. *Graphics Gems*, Academic Press Professional, 395-396.
[23] MOLLER, T. and TRUMBORE, B. 1997. Fast, Minimum Storage Ray/Triangle Intersection. *J. on Graphic Tools*, 2, 1, 21-28.
[24] BIKKER, J. 2005. Raytracing: Theory & Implementation Part 4, Spatial Subdivisions. June 10, 2005. Retrieved Jan. 9, 2007, from: http://www.devmaster.net/articles/raytracing_series/part4.php.
[25] HAINES, E. 1999. Quicker Grid Generation via Memory Allocation, *Ray Tracing News*, 12, 1.
[26] AKENINE-MÖLLER, T. 2001. Fast 3D triangle-box overlap testing. *J. of Graphic Tools*, 6, 1, 29-33.
[27] AMANATIDES, J. and WOO, A. 1987. A Fast Voxel Traversal Algorithm for Ray Tracing. *Proc. EuroGraphics 87*, 3-10.
[28] HAVRAN, V. and BITTNER, J. 2002. On Improving KD-Trees for Ray Shooting. *Proc.WSCG 2002*. 209-216.
[29] PHARR, M. and HUMPHREYS, G. 2004. *Physically Based Rendering*, Morgan Kaufmann.
[30] MACDONALD, J.D. and BOOTH, K.S. 1990. Heuristics for ray tracing using space subdivision. *The Visual Comp.*, 6, 3, 153-166.
[31] KAY, T. and KAJIYA, J. 1986. Ray Tracing Complex Scenes. *Proc. ACM SIGGRAPH*, pp. 269-278.
[32] LAUTERBACH, C., YOON, S.-E., TUFT, D. and MANOCHA, D. 2006. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs, *Proc. IEEE Symp. on Inter. Ray Tracing 2006*, 39-46.
[33] ACPI. Advanced Configuration & Power Interface, Retrieved Oct. 2006 from: http://www.acpi.info/
[34] WALD, I., IZE, T., KENSLER, A., KNOLL, A., and PARKER, S.G. 2006. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Trans.Graphics*, 25, 3, 485-493.
[35] LEXT, J., ASSARSSON, U., and MOLLER, T. 2001. BART: A Benchmark for Animated Ray Tracing. *IEEE Computer Graphics and Applications*, 21, 2, 22-31.
[36] WACHTER C AND KELLER A, Instant Ray Tracing: The Bounding Interval Hierarchy, in Proc. EGSR 2006.
[37] WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD Thesis, Saarland Univ., Germany.