An Analysis Of GPU-based Interactive Raytracing

Ikrima Elhassan
The University of Texas at Austin
ikrima@gmail.com

Abstract

Raytracing is a simple and elegant solution to many realistic image synthesis problems such as global illumination. Recent hardware advances have made GPU based raytracing possible. This paper explores the design space as well as the characteristics and limitations of interactive raytracing on said platform.

We implement a GPU-based raytracer and compare it with a native as well as a modified CPU raytracer that mimics the GPU raytracer. We found that the GPU raytracer was almost always slower than the CPU raytracer, typically by a factor of 2x.

The primary reason for the poor performance is the high cost of per pixel branching. This is illustrated in a pathological test case that causes high ray divergence. Additionally, the constraints of the architecture mandate inefficient modifications to the kd-tree traversal routine.

1 Introduction

One of the primary goals of computer graphics is the ability to render, or synthesize, images that are indistinguishable from photographs. Unfortunately, this aim for photorealism is at odds with performance. Real-time and interactive applications will sacrifice visual fidelity for speed by using techniques such as Z-buffer rendering.

However, these current raster models restrict graphics applications to local illumination models, forcing the developer to fake inter-reflection among other effects through the use of pre-computed texture maps. More so, these workarounds generally break when dynamic, especially deformable, geometry is introduced to the scene. Transitioning to the next level of photorealism requires using more advanced rendering methods such as raytracing.

Raytracing provides an elegant solution to complex visual phenomenon such as global illumination or caustics. Current work has shown that raytracing can no longer be thought of as an inherently slow operation and that real time raytracing rates are achievable [Reshetov et all 2005; Wald et al 2001].

But, these current systems place restrictive conditions on the scene such as prohibiting deformable motion or effects beyond simple Whitted raytracing. This alone is not enough to make the switch from Z-Buffers to raytracing compelling. Instead, an interactive raytracer must be able to support:

- Dynamic scenes (including both rigid & deformable motion)
- Complex lighting effects such as soft shadows, glossy reflections, etc
- Geometrically complex scenes

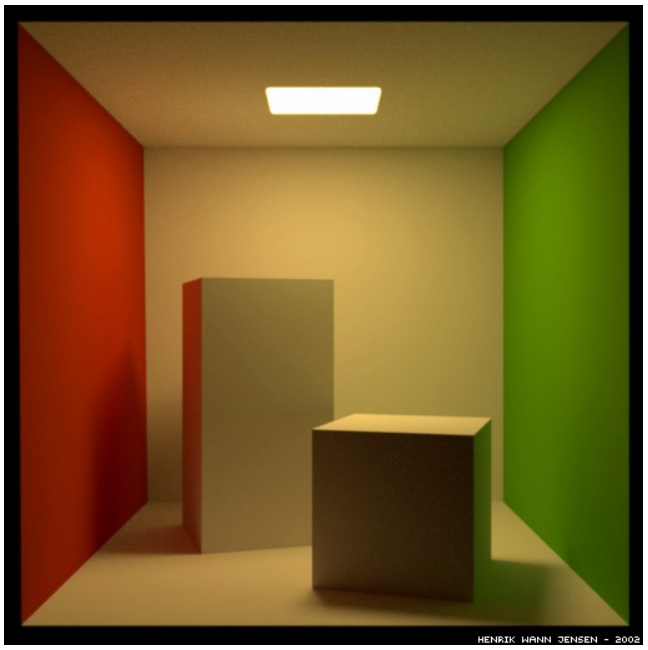


Figure 1. The indirect illumination of the side walls cast a red and green cast onto the cubes

Given our choice of rendering algorithm, the logical questions one must answer are:

- What hardware platform should we use?
- What components are necessary to make it fast?
- How do these components constrain our previously stated goals?

With our goals and evaluation framework in mind, the primary contributions of this paper are:

- To implement a GPU-based raytracer, GPURT
- To compare the capabilities and performance of GPURT against a similar CPU implementation and a naïve CPU raytracer
- To analyze the design constraints of GPU-based raytracing

The rest of the paper is organized as follows: First, we explain the basics of ray tracing (Section 2). Then, we detail the system design of a GPU based raytracer, GPURT (Section 3). We present performance comparisons between the GPURT and other CPU implementations (Section 4). Finally, we conclude with a discussion of future work (Section 5).

2 Background

In the following section, we provide a brief explanation of Graphics Processing Units for those unfamiliar with this hardware platform. Then, we outline the basic raytracing algorithm, its inherent challenges, and the solutions we utilize to overcome them. Additionally, we indicate the constraints of the hardware platform as well as the resulting algorithmic accommodations implemented.

2.1 Graphics Processing Units (GPUs)

Modern day video cards contain Graphics Processing Units, GPUs. GPUs are a highly specialized, highly parallel SIMD architecture that employs a raster model to render images. These GPUs perform computations on geometry through the use of programmable shaders. The shaders can be thought of as subroutines that perform operations on blocks of data at a time. The shaders are written in a C like language that reflects the hardware's capability by placing restrictions such as no indexable variables, bitwise operations, etc.

Despite the limited programming model, developers can use the GPU in ways beyond its original intended use. Because of its architecture, GPUs provide a raw floating point capability that far exceeds the CPU. A top of the line video card from Nvidia, the GeForce 7800 GTX, can perform nearly 200 Gigaflops per second [Nvidia 2005]. Thus, certain classes of compute intensive applications may benefit greatly when mapped to the GPU.

2.2 Basic Raytracing

Raytracing is an image synthesis technique that simulates light interaction. By following light rays as they bounce throughout the scene, one can capture non-local illumination effects as depicted in Figure 1. The raytracer applies termination criteria to cap the number of light ray bounces; usually, the user sets the max number of ray bounces or either sets the minimum light contribution that a ray must have before it is traced.

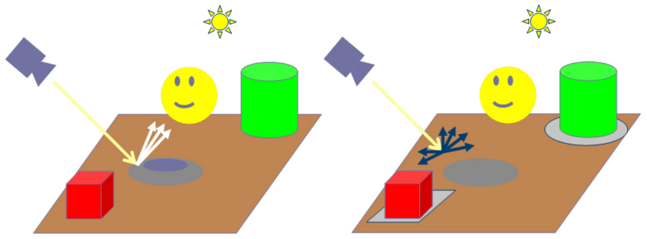


Figure 2. Light rays are cast from the camera and traced around the scene to determine the light arriving at that pixel.
Left: Secondary rays are cast to the light source to determine incoming direct light
Right: Secondary rays are cast to the scene to determine incoming indirect light

Thus, the primary operation is casting and intersecting rays with the scene. On a high level, raytracing operates as follows (depicted in Figure 2):

1. Cast a ray through current pixel from the camera position
2. Find the nearest intersection point along the ray
3. Determine incoming direct light at this point by casting rays to the light sources
4. Determine incoming indirect light by casting secondary rays into the scene
5. Perform shading to determine the color at the intersection point
6. Repeat from Step 2 for secondary rays, until termination criteria is met

To grasp the magnitude of this task, consider the following example: Suppose we desired to render a scene from a typical modern day video game, containing 1 million triangles [Unreal Technology 2006], at an image at a resolution of 1024x1024. For simplicity's sake, suppose we limit the number of max ray bounces to 1. Rendering this image requires tracing 3.1 million (1024 x 1024 x 3). Using the simple scheme described above, each ray requires 1 million triangle intersections, or 3.1 trillion intersections for the entire scene. Clearly, the naïve raytracing scheme is infeasible which is why much of the research in the field of raytracing centers on speeding up this intersection operation. One of the primary ways to do this is to employ an "acceleration structure" that allows for rapid intersection by intersecting rays with a small subset of the scene geometry. We describe such a structure next.

2.3 The KD-Tree Acceleration Structure

The acceleration structure implemented in GPURT is a k-dimensional tree. The kd-tree, a variant of the more general BSP tree, is a hierarchical spatial subdivision structure that accelerates ray-triangle intersections. Like a BSP tree, it does this by recursively splitting the world in half through the choice of an axis aligned splitting plane.

For non-leaf nodes, each child node is classified as above the split plane or below the split plane. For leaf nodes the classification is the same, but it is for the corresponding primitives stored in the leaf. Because of this hierarchy, the kd-tree has an expected computational complexity of O(log n), where n is the number of triangles in the scene. Research has shown that the fastest acceleration structure for raytracing is the kd-tree.[Havran 00] Moreover, Wald et al presents a method to trace packets of rays through a kd-tree, allowing for an order of magnitude increase in performance on SIMD architectures.

In the remaining section, we describe how to build and traverse a kd-tree.

2.3.1 Kd-tree Construction

To build a good kd-tree, we require the following properties:

1. Small node size
2. A tree that carves out as much empty space as possible (which leads to unbalanced trees)

Modern computers are memory bound instead of compute bound. Traversal of the kd-tree can often time occupy 60 percent of computation time. Thus, the first property—small node sizes—allows more kd-tree nodes to fit into a cache line. For instance, Pharr et al reports that merely cutting down the kd-tree node size from 16 bytes to 8 bytes increased the performance of their raytracer by 20 percent.

Because current GPUs do not support bitwise operations, the GPU kd-tree requires a 4-d float for each node, laid out as follows:

KDTreeNode
NonLeaf Nodes
node.x = Flag. -1, 0, 1, 2, 3 means invalid node, split-axis = x, split-axis = y, split-axis = z,
leaf node
node.y = Split Position
node.z = Index of ChildAbove
node.w = Parent

Leaf Nodes
node.x = Flag
node.y = Number of primitives
node.z = Beginning index of Triangle List. The triangles are just stored as an array of vertices.
node.w = Parent

The kd-tree build process ensures that the index of the child node below the split is parentNodeAddr + 1, which eliminates the need to store the index of this child node. The kd-tree is stored as a 128-bit texture so that the GPU can operate on it.

The second property seems to go against intuition because in typical binary search trees, we desire a balanced tree to achieve optimal performance. However, the mistake lies in thinking of the kd-tree as a binary search tree. For raytracing, the kd-tree should carve out as much empty space as possible (even though this might create unbalanced trees) because we want to determine as quickly as possible whether a ray will intersect a triangle. Moreover, large empty nodes allow for early exits, further alleviating the traversal bottleneck (Figures 3 & 4).

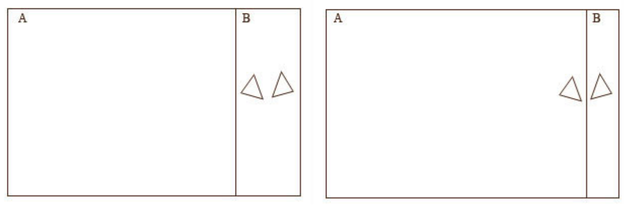


Figure 3. Left: A Bad Split Position Chosen to Make the KD-Tree More Balanced. Right: A More Suitable Split Position

To account for this, the kd-tree builder uses cost functions first introduced by MacDonald and Boothe [MacDo90] to create the acceleration structure:

Leaf Nodes:

$$\sum_{i=1}^{N} t_i(i)$$

where $t_i(i)$ *is the intersection cost for each primitive i in the current node*

Nonleaf Nodes:

$$t_t + p_A \sum_{i=1}^{N_a} t_i(a_i) + p_B \sum_{i=1}^{N_b} t_i(b_i)$$

where t_t *is the cost of traversing a node*
$t_i(a_i), t_i(b_i)$ *is the cost of intersecting primitive ai,bi in Node A, Node B respectively*
p_A, p_B *is the probability that the ray enters Node A, given that it enters the parent node*

Since the raytracer only accepts triangles, the formulas are reduced to *ti* * *N* and *tt* + (1 - *be*) * *ti* * (*pA* * *NA* + *pB* * *NB*), where *be* is a bonus (ranging from 0 to 1) for having one of the child nodes empty.

For a convex *volume A* contained in convex *volume B*, the conditional probability that a ray enters *A* given that it enters *B* is

$$p(A|B) = \frac{S_A}{S_B}$$

where S_A, S_B is the surface area of A and B, respectively [Santalo 76].

We choose the longest axis as the split axis. Then, we project the triangles onto the split axis and choose split positions that coincide with the beginning or ending of the triangle projections, as diagrammed in Figure 4. If a suitable split position is not found based on our cost functions, we cycle through the other axis and choose the axis that yields the lowest cost.

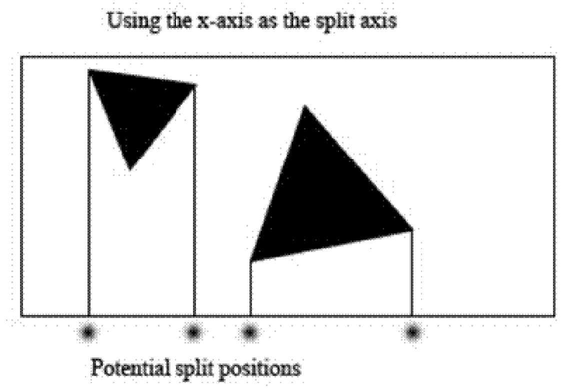


Figure 4. Choosing the split plane

2.3.2 KD-Tree Traversal

To traverse the kd-tree, we begin at the root node with a ray that has *tmin* & *tmax* that specifies the valid interval along this ray. We then determine the correct order of nodes that the ray will visit and whether or not both nodes need to be traversed. To do this, we determine which side of the splitting plane the ray origin lies on (Figure 5).

```
If(ray.orig[split_axis] < split_pos)
firstChild = childnodeBelow
secondChild = childnodeAbove
else
firstChild = childnodeAbove
secondChild = childnodeBelow
```

But we do not always need to visit both nodes. *firstChild* tells us that if both nodes need to be traversed, *firstChild* must be traversed before *secondChild*.

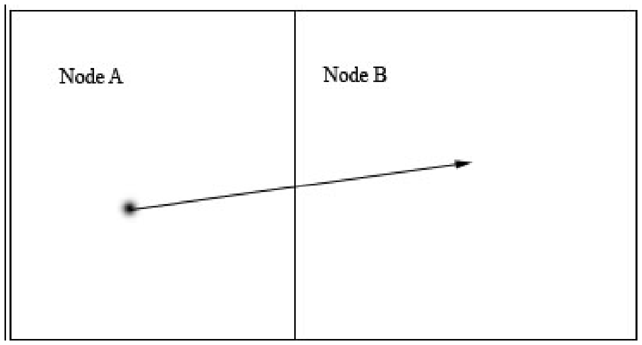


Figure 5. The Ray Origin Lies Below the Splitting Plane, So Ray Visits Node A Before Node B

To determine which nodes need to be traversed, we calculate the intersection along the ray (*tplane*) to the splitting plane and choose the nodes to traverse based on the following scenarios:

1. *tplane >= tmax of the ray or tplane < 0*

   If *tplane* >= *tmax*, then we know that the active interval of the ray is completely contained in the *firstChild* node. Thus, we do not need to traverse down the *secondChild* node and can directly proceed down the *firstChild* of this node.

   If *tplane < 0*, then we know the splitting plane (and thus the *secondChild* node) lies behind the ray, eliminating the need to traverse the *secondChild* node as well. Refer to the following figure.

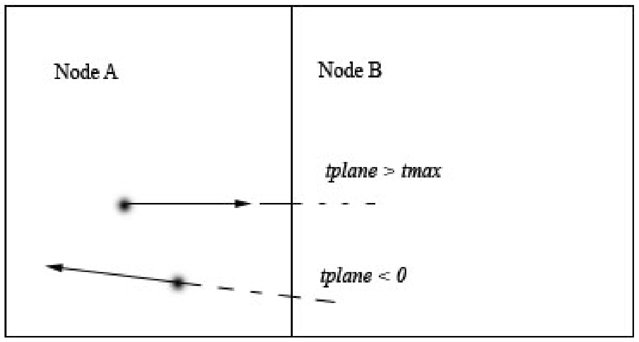


Figure 6. In these cases, we only need to traverse the left child

2. *tplane <= tmin and tplane >=0*

   Since *tplane >= 0*, we know the splitting plane lies in front of the ray, but since *tplane <= tmin*, we know that the active interval of this ray is completely contained within *secondChild*.

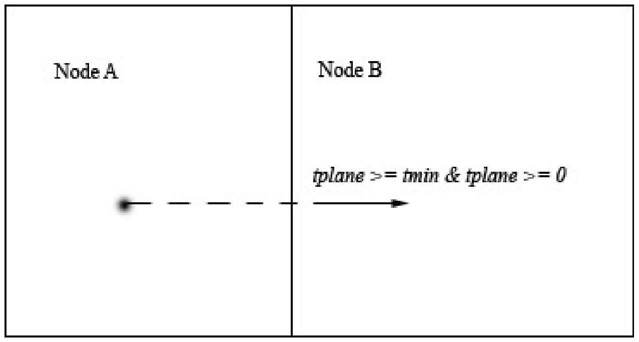


Figure 7. In this situation, we only need to traverse the right child

3. *All other cases*

   In this situation, we may have to traverse both nodes; if we do not find an intersection when traversing the firstChild portion of the kd-tree, we must return to the secondChild and look for an intersection there. On the CPU, we would keep a stack of secondChild nodes to traverse, but since current GPUs do not support this, we follow Foley et al's solution. They propose two solutions: kdrestart and kd-backtrack. To briefly summarize, once a ray reaches a leaf and needs to start traversing the nodes on the other side of the tree, the kd-restart algorithm retraces the ray in question with updated valid intervals from the top of the tree. The kd-backtrack algorithm backtracks through parents of nodes until it reaches a node where a secondChild had to be traversed. This modification causes rays to revisit unnecessary nodes in the kd-tree, incurring an extra cost for traversal. For further details, refer to the paper.

Once we reach a leaf node, we intersect all the triangles stored in that leaf with the ray in question. If we find an intersection within the active part of the ray, we can quit traversing the kd-tree because the kd-tree guarantees that the first intersection will always be the nearest. If no intersection is found, we either resume kd-traversal if we need to, or we report that no intersection is found. On the CPU, the termination criteria would be no remaining nodes on the stack. For our implementation, we use the criteria described by Foley et al.

2.3 The Problem of Dynamic Scenes

Dynamic scenes pose a significant problem for raytracing. Much of the literature concerning acceleration structures assumes static scenes, while those that consider dynamic scenes often ignore deformable motion. These acceleration structures often require an expensive preprocess creation phase. In the case of dynamic scenes, this cannot be thought of as "free" because the acceleration structure must be rebuilt every frame. We address this important issue in our design of GPURT described in the next section.

3 GPURT

3.1 Why GPUs?

While CPUs provide a much more general purpose programmability, modern day GPUs are supporting more and more general programmable shaders with each new generation. Although the clock rate of a GPU is much lower than that of a CPU, GPUs provide a much higher raw floating point capability.

Additionally, a dual compute model can be utilized to overcome the significant hurdle of raytracing dynamic scenes. The CPU can build the kd-tree for *frame n+1* while the GPU traverses/renders *frame n*. Using this setup, the application can support fully dynamic scenes because the application rebuilds the acceleration structure every frame. Since kd-tree traversal is inherently parallel, it is easier to map to the GPU, whereas kd-tree construction is by nature a serial process better suited to the CPU.

3.2 System Design

Instead of using many small shaders (a generate rays shader, a kd-down shader, an intersect shader, etc) to perform raytracing as suggested by Purcell et al, GPURT raytraces scenes using 4 passes by employing branching and looping instructions. The motivation behind this is to avoid load-balancing between many small shaders and the overhead of rendering intermediate results to various textures.

To determine whether GPU-based raytracing is a fruitful path, we chose to initially implement a basic set of functionality before attempting a more ambitious feature complete raytracer. GPURT currently supports:

- Diffuse and specular shading
- Texture mapping
- Interpolated vertex normals
- One-bounce reflection
- Shadows

Input data, such as the kd-tree or the scene list, is stored in textures, which are blocks of data. To perform raytracing, the application renders a screen-aligned quad that is operated on by the various raytracing passes, as shown in Figure 8.

The black boxes represent the different passes. Red boxes and lines represent 32-bit floating point output data representing intersection information. Blue lines signify where this data is used as an input. Finally, the green boxes and lines represent 8-bit output color data.

3.2.1 RTPrim Pass

The *RTPrim* pass generates primary rays based on the current camera position and current pixel position. Then, it traces the generated ray using the kd-tree, and outputs intersection information containing the intersected triangle id, the barycentric coordinates of the intersection, and the t value of the intersection.

If an intersection does occur, the z-value of the current pixel is updated to 0.0; otherwise, a 1.0f is written instead. This is done to enable early z-culling of all the inactive pixels in the remaining passes.

3.2.2 ShadePrim Pass

We then shade primary rays using the intersection information from the previous pass. Shadow rays are traced for active pixels. If the current pixel is not in shadow, it is shaded using lighting equation employed by OpenGL and Direct3D.

3.2.3 RTRefl & ShadeRefl Passes

These passes are analogues to RTPrim & ShadePrim. The only notable difference is that the blend_add raster operation is set so that the output of ShadeRefl is accumulated into the frame buffer.

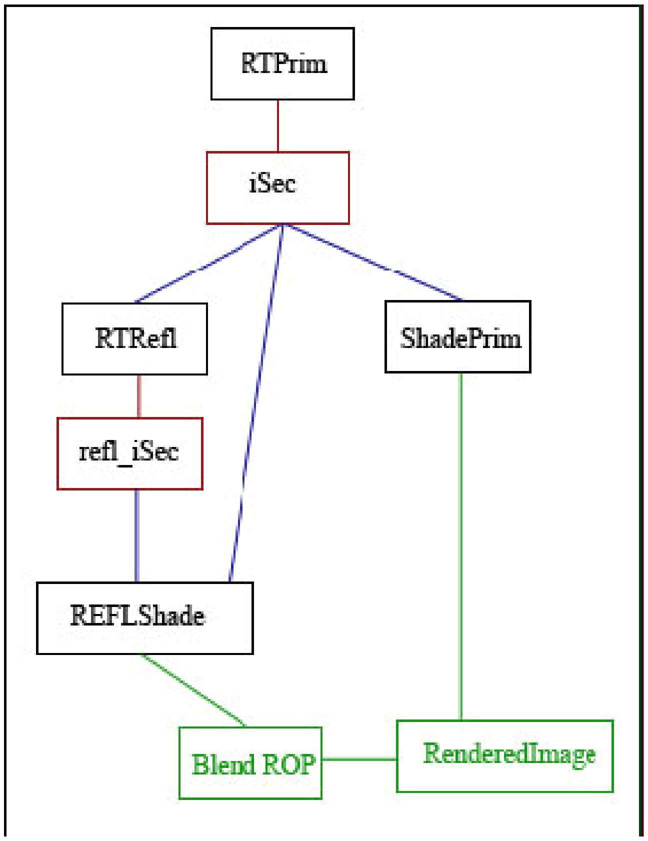
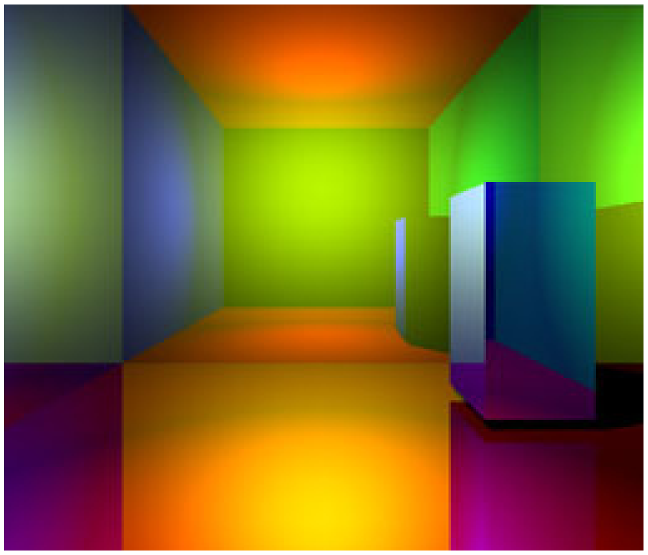


Figure 8. Raytracing Passes

4 Results and Analysis



To measure the impact of the GPU hardware architecture on raytracing, we implemented and collected timings for CPURT, a simple CPU raytracer, as well as GPURT. Additionally, previously stated GPU constraints mandated modifications to the kd-tree traversal algorithm. To decouple the role of the hardware from the algorithmic modifications in our performance analysis, we measure performance timings for a third CPU based raytracer, CPU Kd-restart. This raytracer is similar to CPURT but it utilizes the modified kd-tree traversal algorithm. Tests were performed on AMD 64 2.21 GHz, 1 GB RAM, nforce4, Geforce7800 GTX, on a 512x512 image.

4.1 Varying Scene Size

We first measure GPURT's performance with respect to scene size to measure the effect of the GPU architecture on kd-tree traversal. The scene consists of an open box with one wall removed and a procedural ball, tessellated at a resolution of 1K triangles, 4K triangles, and 9K triangles.

As expected, the properties of the kd-tree causes the performance of the raytracers to decrease logarithmically with respect to the number of triangles; yet, the CPU raytracers' performance far exceeds that of GPURT. Unfortunately, CPURT's performance far exceeds GPURT's.

The performance of CPU Kd-restart only falls behind CPURT by 15%, revealing the cost incurred by the less efficient kd-traversal algorithm. However, because GPURT is an order of magnitude behind CPURT, it follows that the GPU hardware platform introduced additional inefficiencies. The next series of measurements are designed to reveal the source of the inefficiencies.

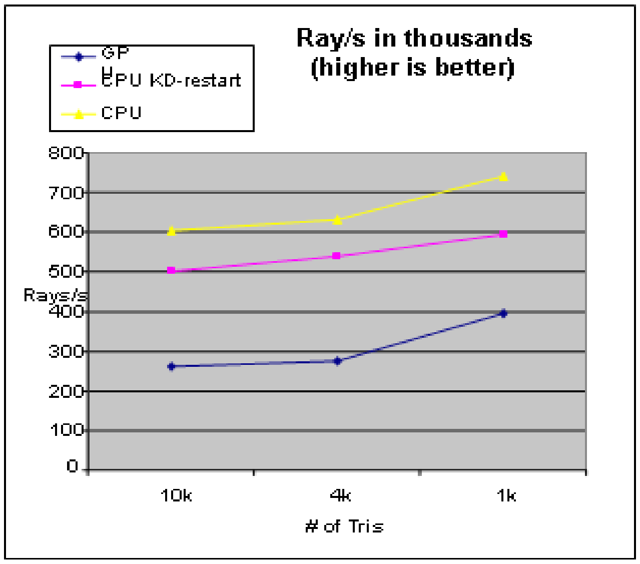


Figure 9. Performance decreasing logarithmically with respect to scene size

4.2 Highly occluded scenes

To accurately gauge the impact of the acceleration structure in culling away large portions of the scene, we render an image of a scene consisting of 25 rooms each containing a box, resulting in an overall scene size of 10K triangles (Figure 10). However, the camera is oriented such that only one of the rooms is viewable to the camera.

Surprisingly, GPURT performs 2x faster on this scene compared to the similarly sized scene in Section 4.1 (Figure 11). Moreover, GPURT performs on par with the other raytracers and only 13% slower than CPURT. One plausible explanation for this behavior is that the impact of divergent neighboring rays severely impedes performance. To verify this, we constructed a pathological case described in Section 4.3.

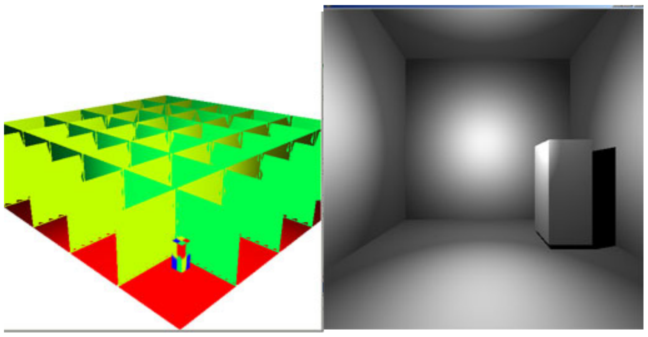


Figure 10. Left: An DirectX render depicting the entire scene. Right: The rendered viewport of the scene.

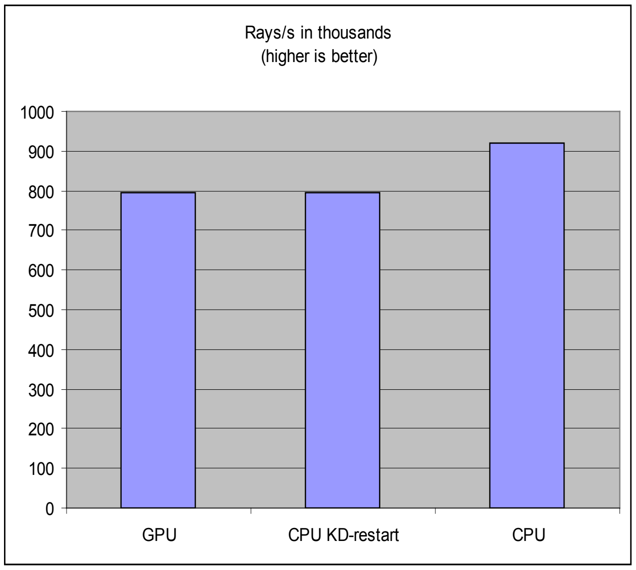


Figure 11. The GPU performs very well as long as the number of triangles on screen is small.

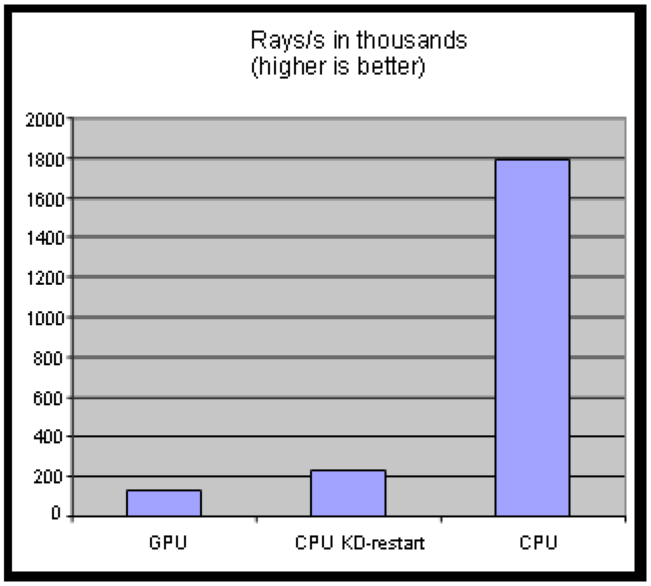
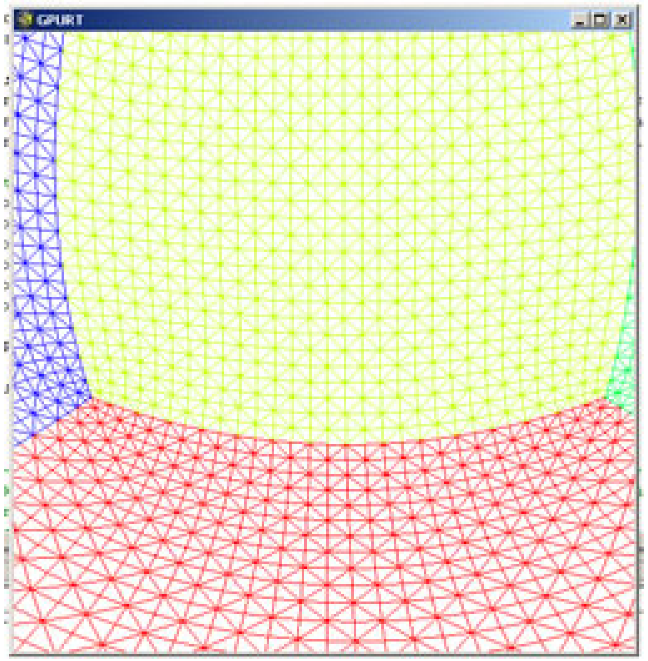


Figure 12. The measured performance of the raytracers

4.3 Diverging rays



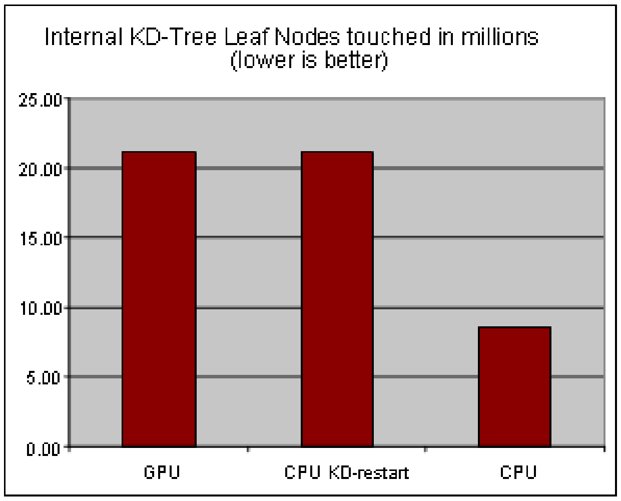


Figure 13. The number of accessed internal kd-tree nodes. The modified traversal algorithm accesses nearly 4x the number of required nodes

Increasing the number of onscreen triangles increases the probability that neighboring rays will intersect different triangles; this in turn causes nearby rays to access different nodes of the kd-tree. To measure the impact of ray divergence on performance, we render a scene consisting of 10K viewable triangles. The onscreen area of the triangles is made to be small to maximize the probability of neighboring rays intersecting different triangles.

The modified Kd-traversal algorithm performs poorly when rays diverge (Figure 13); the modified traversal routine performs 5x the work required during traversal causing CPU Kd-restart to perform ~8x slower than CPURT (Figure 14). Moreover, the GPU architecture further exacerbates this problem as seen by its performance that is 2x slower than its CPU counterpart. Thus, ray divergence combined with the modified kd-traversal routine is main factor in the performance inefficiency of GPURT.

5 Conclusion and Discussion

Mapping raytracing to the GPU poses several significant obstacles. First, development time is considerably longer due to a lack of a debugger and long compilation times. For instance, 1-2 days of development on the CPU raytracer would consume 1-2 weeks when developing for the GPU raytracer. Second, the lack of bitwise operations mandates larger kd-tree nodes. Third, a very low loop counter and max number of instructions executed lead to a very crippling cap on scene size.

Additionally, hardware constraints such as maximum number of instructions executed introduced a severe limit on the scene size. The kd-traversal loop persistently reached this ceiling with scenes consisting of 2k triangles. Furthermore, the cost/latency of pixel branching combined with frequent divergent branches outweighed any benefit gained from avoiding load balancing several small shaders.

Because of these obstacles, efficient GPU based interactive raytracing is still far away. Traversing acceleration structures on the GPU remain a costly procedure. The cost of using longer shaders and branching instructions is still too high on modern GPUs. To better support raytracing, GPUs must include more efficient branching in pixel shaders, a larger program counter/loop counter, indexable registers, and better development tools (compiler/debugger).

References

Foley, T., and Sugerman, J. "KD-Tree Acceleration Structures for a GPU Raytracer." Graphics Hardware 2005.

Luis A. Santalo. Integral Geometry and Geometric Probability. Encyclopedia of Mathematics and its Applications. Addison-Wesley, 1976.

MacDonald, J., and Booth, K. "Heuristics for ray tracing using space subdivision." The Visual Computer, Vol. 6, No. 3, pp. 153–166, 1990.

Nvidia. "NVIDIA® GeForce™ 7800 Receives Widespread Game Developer Endorsement." Press Release. 22 June 2005. < http://www.nvidia.com/object/IO_23416.html>

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

Purcell, T., Buck, I., William, M., and Hanrahan, P. "Ray Tracing on Programmable Graphics Hardware." ACM Transactions on Graphics. 21 (3), pp. 703-712, 2002. (Proceedings of ACM SIGGRAPH 2002).

Reshetov, A., Soupikov, A., and Hurley, J. "Multilevel ray tracing algorithm." SIGGRAPH '05: Proceedings of the 32nd annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA.

Unreal Technology. "Unreal Engine 3." May 2006. <http://www.unrealtechnology.com/html/technology/ue30.shtml>

Wald, I., Benthin, C., Wagner, M. and Slusallek, P. "Interactive Rendering with Coherent Ray Tracing." Eurographics Conference Proceedings, 2001.