GPU-Based Parallel Stackless BVH Traversal for Animated Distributed Ray Tracing

Charles Lohr
UMBC

Abstract

This paper presents a series of accelerations and techniques useful for performing interactive GPU-based distributed animated ray tracing. By taking advantage of several techniques used to accelerate ray tracing along with some additional effects, one is able to achieve enough speed to perform distributed ray tracing animated scenes at interactive rates.

A framework is provided to load both limited Standard Procedural Database files [Haines 1987] or from standard OBJ files, these objects can be assembled into scenes that can be interactively explored. Use of transform nodes makes it possible to add additional features, including animation, without recomputing the entire bounding volume hierarchy.

Some acceleration structures used to achieve this result are stackless traversal of bounding volume hierarchies using spheres and **parallel traversal** which will be presented in this paper.

CR Categories: I.3.7 [Computing Methodologies]: Graphics—Three-Dimensional Graphics and Realism

Keywords: ray tracing, distributed ray tracing, drt, realtime, opengl, bvh

1 Introduction

Whitted-style ray tracing [Whitted 1980] has for many years been a powerful mechanism for producing high quality computer images. Ray tracing is a system of generating images that involves treating the output image pixels as a plane in front of location where the user wishes to place the "eye." For every pixel on the view plane, a ray is cast out from the eye, through that point on the view plane into the scene. All geometry is then intersected with that ray. For the closest piece of intersected geometry, new rays must be sent out to every light in the scene. If reflections are used, the intersecting ray must reflect off of the surface and check scene again. Addition of reflections causes the complexity of the ray tracer to grow as seen in the tr term of Equation 1. Where x and y are the width and height of the screen, n is the number of geometry elements, l is the number of lights and tr is the average number secondary rays from reflection or refraction.

$$time = x * y * n * l * tr \tag{1}$$

Ray tracing appears to be prohibitively expensive for the purpose of rendering images at interactive rates. It can offer effects that are either difficult or impossible to perform with rasterization. The addition of distributed ray tracing aids this by allowing even more effects that are difficult to achieve when using rasterization.

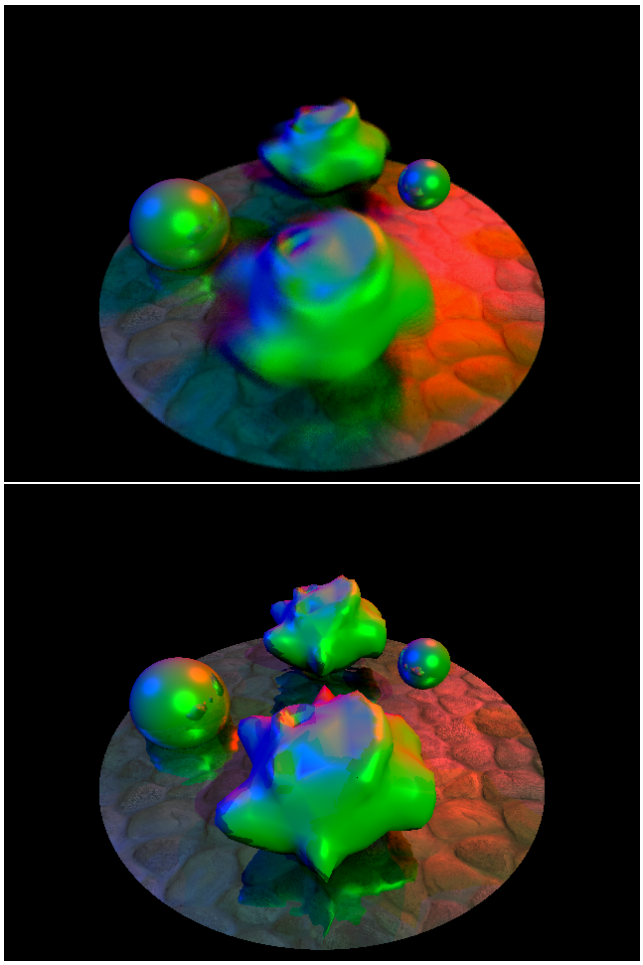


Figure 1: *The top image shows a final output of our results rendered at 640x480 with 16x DRT rays and 1767 pieces of geometry. The bottom image has no DRT.*

With the innovation of distributed ray tracing (or DRT) [Cook et al. 1984], even more effects are able to be included with only a minor, constant increase per ray. This enables effects such as motion blur, anti-aliasing, depth of field, soft shadows and specular reflections. This raises the complexity, as seen in Equation 2. $Rays$ is the number of rays added. With a minimum number of additional rays (16-64), one obtains all of the DRT effects at once instead of needing to increase the number of additional rays for each effect.

$$t = x * y * n * l * tr * rays \tag{2}$$

This is done is by choosing a fixed number of additional rays that will be required ($Rays$) and then proceeding to shoot them out into

the scene, applying randomness at each level. For instance, a ray may start out at a random point on the view plane, shoot through the focal point for that pixel and into the scene. When it hits a surface, it shoots out a shadow ray randomly at the volume represented by the light that is being targeted.

Ordinarily, generation of images can take extremely long periods of time to render when utilizing regular ray tracing, or DRT. Many techniques, such as packeted scene traversal, are able to reduce the x and y quantities in the traditional ray tracing. Most other techniques attempt to attack n, reducing the total number of geometry intersections necessary to check.

People have attempted to leverage specialty programmable GPU hardware for some time in order to accelerate ray tracing [Purcell et al. 2002]. Since ray tracing is often referred to as embarrassingly parallel, and programmable GPU hardware is also regarded as highly parallel, people have found this a convenient match.

We have been unable to find any papers that have implemented the combination of these techniques on the GPU that can be applied toward distributed ray tracing. The approach is straightforward. However, interesting effects were found because of the hardware used in GPUs. Traditional packetized traversal was not used in lieu of a technique that is described in this paper called **parallel traversal**. Instead of treating a series of rays as a packet or conical frustum, we treat each ray individually, then traverse the BVH on an either-or basis. If any of the rays that we are testing in this pass hit a volume, we traverse into that volume.

Because of relatively similar paths that rays shot out from the same pixel on the screen take, it is likely there will be a high level of coherence between all secondary rays related to that packet. This can yield a performance increase approaching that of the total number of parallel rays shot out in the traversal step as can be seen in Figure 2 and Figure 4.

Unlike many of the other ray tracing papers cited, we have decided to include techniques that are not trivial with rasterization. Much of the current research in this field of accelerated ray tracing involves ray tracing scenes that could have been rendered utilizing conventional rasterization.

Our discussion of the techniques will discuss both a CUDA and OpenGL Shading Language (GLSL) implementation. We will cover the acceleration structures as well as their performance on different systems. The final implementation is done in GLSL for a variety of reasons including compiler and performance issues with CUDA. All of our tests are run on NVIDIA hardware, including a GeForce 8400G M, GeForce 9800GTX+ and a GeForce 9800.

We can see that acceptable speedups are used in order to minimize the cost of the DRT (as seen in Figure 1). Figure 1 (Top) was rendered in 2.74 seconds on a GeForce 9800GTX+, 3.57s on a GeForce 9800GT and 75.32s on a GeForce 8400GM. The bottom image shows the same scene rendered with no DRT in .26 seconds on a GeForce 9800GT.

The images used throughout this paper include an 882-polygon "baddie" OBJ models, a texture- and bump-mapped ground surface, loose spheres, and the SPD "balls".

2 Previous Work

Attempts to speed up ray tracing have been in the works for quite some time. Attempts have been made with packeted traversal, as seen in [Günther et al. 2007] and [Boulos et al. 2007]. Packeted traversal, as stated above attacks the x and y components of Equation 1. It does this by binding all common rays together into frustums or conical frustums. By doing this, traversal of the structure needs only be done once per group of rays. Previous research finds optimal groups to be of size 4x4 or 8x8 [Wald et al. 2006].

Many other techniques such as k-D trees for space division [Horn et al. 2007], sphere, or box based bounding volume hierarchies (BVHs)[Christen 2005], [Günther et al. 2007], and coherent grid traversal [Wald et al. 2006] have been proposed to attack the n term of the equation. While both k-D trees and BVHs are common, [Christen 2005] shows that both are viable options when doing ray tracing. Because of the simplicity of generation that BVHs offer, we have chosen to use sphere-based BVHs.

Because of the limitations of GPUs in their limited register space and difficulties in programming a stacks, stackless traversal [Popov et al. 2007], using a system similar to geometry images [Xianfeng et al. 2002] is a very powerful mechanism. They can be seen more generally in [Carr et al. 2006]. Geometry images can eliminate the need for a stack with no added time-complexity overhead and a minimal addition of space necessary to store the traversal information in the structure itself.

Unlike many of the other papers that implement geometry images, we take the approach of including texture pointers, which is found in some of the later papers. This enables us to have abnormally patterned data structures, including incomplete trees and trees with several leaves at any given level.

Recently, bump mapping [Blinn 1978] has been applied to rasterization rendering. This technique was applied originally in ray tracing and has continued in full force in that area. We will utilize it in our ray tracing project. The application of bump mapping is easy on any of the surfaces within the scene.

3 General Setup

Because of the nature of GPUs and systems already in place for texture fetching, we have chosen to store the scene and light data inside of textures similar to the geometry images listed above. Because of this, the CPU-side algorithm is uniform for both CUDA and GLSL. Because of the natural texture caching mechanisms, we have chosen to make our texture elements 2-high in the y direction and somewhat spatially coherent in the x direction. In all of our tests, the texture itself was square, 512x512.

The system directly uses an OpenGL output buffer to enable the user to interactively work with the scene. This also provides a convenient mechanism for displaying an output image: the screen itself.

The basic ray caster, with no reflections or lights, was written in both CUDA and GLSL. A comparison can be seen in Figure 2.

When operating in CUDA, we operate on an 8x8 block, since it seems to be sufficiently large for all systems. Also, a 16x16 block size does not appear to run on GeForce 8 series cards; and 4x4 block sizes incur a performance hit on all tested systems. Many common functions like cross products and dot products that can be found in NVIDIA's set of functions [NVIDIA 1993-2007] are used in the code. We found that CUDA consistently produced code that ran more slowly than the GLSL implementations. We also found that CUDA often produced incorrect output. We did, however, find that even when there was a major issue with the program, it did not cause the entire computer to crash.

The GLSL implementation uses native intrinsics such as $cross$, dot, etc. Overall, the code ends up being cleaner because the nature of ray tracing more closely matches what fragment shaders are intended to be used for. We are able to consistently get a 10-35%

Test	CUDA	GLSL
256x256 coherent	50	72
256x256 incoherent	45	64
512x512 coherent	33	51
512x512 incoherent	24	45

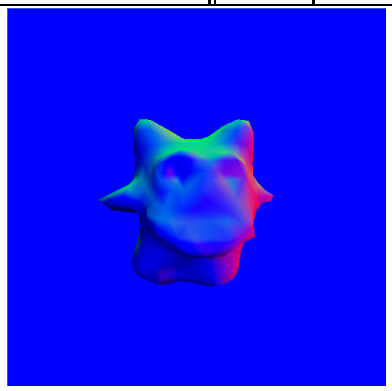


Figure 2: *This is the performance test on a single 882 polygon 'baddie' numbers are in frames per second. It should be noted that the second set of readings are with a slightly modified output image.*

performance increase in our GLSL implementation as compared to the CUDA one.

The difference between the coherent and incoherent test runs is whether the rays of adjacent pixels go in remotely similar directions. Typically in ray tracing, primary rays are highly spatially coherent, traversing through many of the same portions of the BVH as their neighbors. However, on secondary rays, it is very likely the rays may traverse through different parts of the tree. It can be seen here that when considering the rendering process as a whole, using our stackless method means the GPU is not hindered by poor spatial locality. Furthermore, when applying our parallel traversal, it does incur a fairly serious performance hit. Overall, we continued down the path of using parallel traversal because of its beneficial effects.

Incorrect code was found generated by both the GLSL and CUDA compilers that normally resulted from indirect memory addressing or complex looping structures. For this reason, both the GLSL and CUDA implementations are written in a more awkward way, which is necessary for removing the need for indirect addressing by using `#defines` for many loops.

4 Stackless Traversal

Geometry images are the key component of our stackless traversal. They enable us to traverse the entire scene's BVH without need for recursion or a stack. They require a BVH to be assembled which can take $\Omega(n \lg n)$ to $O(n^2)$ time to compute with relatively large constant factors. This should only be done once for complex geometry. The BVH structure is typically non-binary, with nodes typically having more than two children. By "throwing back" pathological spheres into the parent, one can get a more efficient tree.

The trees start out on the CPU as arbitrary width trees and are in a typical tree structure with pointers. They must be converted into a texture by baking the stack into it. This is a two-step process. The CPU-based tree must be traversed in order to find out which is the next element in an in-order traversal. Each node is then notified what the next node in the in-order traversal is, and stores that node's position in its *next* texture pointer. Once this is done, a compression step can be performed, where each element that would end in a terminated state terminates immediately.

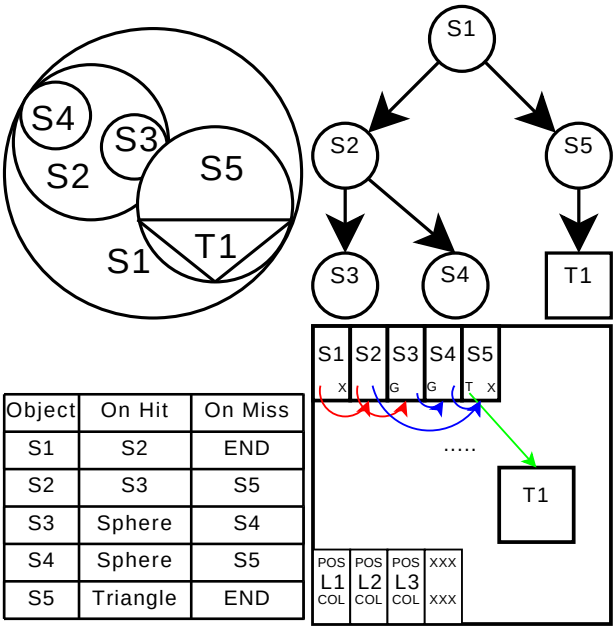


Figure 3: *The left image shows how lights and geometry are stored within our texture. The right image shows the equivalent tree of geometry. Ss are spheres, T is triangle information, X indicates end-of-traversal, G indicates sphere geometry.*

Because of the nature of GPUs and systems already in place for texture fetching, the geometry and light data is stored inside of textures, similar to the geometry textures listed above. A detailed diagram of the memory layout can be seen in Figure 3. Because of the layout, it made the CPU-side algorithm uniform for both the CUDA and the GLSL implementations. The hardware texture caching mechanism lets us take advantage of the minor spatial locality in the geometry image on both implementations.

The pointers are two floating point values that point to the next place on the texture $[0\,..\,1)$. There are integer offsets to determine what type of node will be hit (sphere, triangle, transform, or end).

The traversal of the algorithm is very simple. The algorithm starts at the upper left hand point in the texture and hops either in or out depending on whether the bounding sphere is intersected by the ray or not.

FINDNEXTINTERSECTION : $I = (V, E) \rightarrow C$

```
1  ptr ← (0., 0., 0.)
2  do
3      do
4          sphere ← GeometryImage(ptr.xy)
5          jmp ← GeometryImage(ptr.xy + Offset_y)
6          ptr.z ← DETERMINANT(sphere)
7          ptr.xy ← (ptr.z > 0.)?jmp.xy : jmp.zw
8      while ptr.x > −0
9      if ptr.x < T_val break
10     PROCESSGEOMETRY
11     ptr.xy ← jmp.zw
12 while true
13 PERFORMGLOBALGEOMETRYTESTS
14 return parameters
```

The geometry image cannot use any texture filtering, lest the scene graph become corrupted due to the filtering modifying our pointers. It should be noted that in some cases it is necessary to offset

Test	FPS
512x512 (single)	51
512x512 (2xAA) coherent	44
512x512 (2xAA) partially-coherent	26
512x512 (2xAA) incoherent	14
256x256 (single)	72
256x256 (2xAA) coherent	66
256x256 (2xAA) partially-coherent	54
256x256 (2xAA) incoherent	36

Figure 4: *This shows how coherence effects our parallel traversal algorithm. All frame rates are on a GeForce9800GT with the same image as shown in Figure 2*

the pointer when looking up into the texture to prevent accidentally going into the wrong data element. If one operates on the very edge of a cell, it is possible to look up into its neighbor. This offsets can be baked directly into the table. For all texture pointers, (0.5/Width of Geometry Image, 0.5/Height of Geometry Image) may be added as a valid offset.

In PROCESSGEOMETRY one may use integer values between T_{val} and 0 in order to discern different primitives. Since the determinant is already calculated, it is possible to do only a minimal amount of computation to get the actual depth to intersection of any spheres or other objects that could use that data. Other primitives could include transforms, triangles, spheres, cylinders, etc. This project only includes triangles, spheres and transforms as primitives.

PERFORMGLOBALGEOMETRYTESTS handles any pieces of geometry that one may wish to leave outside of the tree and hard code into the tracer itself. An example of this would be the round ground plane that can be seen in Figure 1 and Figure 6. It can be handled in any way and should only modify the $parameters$ return value if it is closer than the current minimum depth.

By writing the code in this manner, an entire group of shading units will traverse through the spheres until all have hit the end of the innermost loop. Then, all processors at once will execute PROCESSGEOMETRY. This helps prevent issues with poor granularity on the GPU regardless of how fragmented the traversal is. Also, by writing the loops as do-while-break, they can take advantage of the native methods by which GPU executes loops and branch statements under shader model 3.0 and 4.0.

5 Parallel Traversal

One of the major innovations in this project is the use of **parallel traversal** which closely resembles packeted traversal. Instead of transforming all rays into a packet or other structure that can contain all rays, we can simply ray cast all rays that we are checking against at the same BVH structure. These rays can be spawned out of a single pixel which cuts down on the n term in Equation 2.

When two rays intersect all of the same elements in the structure, the only time that is lost is in the time that is needed to perform an additional determinant test followed by a MAX operation on the two determinants. One can see from Figure 4 that in cases where there is virtually no coherence, much more time is wasted performing this optimization in comparison to the single-ray method. However, most scenes tested have a high level of coherence and thus enjoy a speedup even in detailed scenes (Figure 6).

In tracing three or more rays per pixel, a minimal speedup was noticed over two. When moving beyond three rays per pixel, issues were encountered that produced an extreme slowdown. In future generations of video cards, these concerns may not exist.

Two copies of all parameters, including eye location, ray direction, normal of hit surface, and distance to surface are required in order to perform parallel traversal. Once both rays have their information stored, it is then possible to simply mix the two rays' output color for the end result.

6 Built-in Transforms

The layout of the overall program enables mounting a large sum of geometry to specific, easily controllable nodes. A scene graph can be generated out of this. Utility functions have been written to take select nodes and tree-ify them. It is possible to have large portions of the scene graph manually controlled as well as other parts automatically controlled with a highly compressed hierarchy. It is prohibitively expensive to pack every polygon in the scene, however, it is not difficult to pack a few manually controlled root level objects.

In a typical scene, a series of nodes can be loaded using regular models from OBJ or a subset of NFF files. These can be high polygon. These objects can be attached to a large root node through transform nodes. The transform nodes can be moved, rotated and scaled however needed on a per-frame basis and re-inserted into the geometry texture without recomputing the structures for the transform node's children. Children of transform nodes will take on all of the modifications that the parent has.

When traversing into a transform node, all components (Direction, Eye, Normal...) are modified by the transform's matrix. When traversing out of a transform node, all the parameters are modified by the transform's inverse matrix. It is important to see that the traversing-out stage cannot be overlooked. If it were, the normal and other properties will still be in the object's local space because of the stackless traversal. Because of corner cases, one must traverse into a transform node even if it is behind the eye.

The use of these transforms makes it possible for us to provide an animated scene or add motion blur to static scenes with a minimal amount of CPU overhead each frame.

Due to difficulties coding this feature, it was not implemented in the CUDA code base.

7 Distributed Ray Tracing

Once we have a complete working ray tracer that can maximize the number of rays per time on any given scene, augmenting it to add distributed ray tracing is trivial. A random texture is used to generate at least partially random numbers to perturb the ray for every effect. We have found that by updating the noise every frame with new random data generated on the CPU, we can trace another set of rays. This allows us to lose temporal coherence and enables our eye to blur together the scenes, thus getting additional *fake rays* for the distribution and improving image quality when the frame rate is increased.

Examples of each of these effects running at interactive rates can be seen with our tracer in Figure 5.

7.1 Depth of Field

Depth of field can be achieved by simply treating the image plane as being a fixed point in front of the camera. Then, the rays that are being shot out into the scene can be rotationally perturbed around these points in order to blur points closer or further from the focal point. It is important that the rays still start near the location of the original eye, so that we do not end up tracing from in front of the

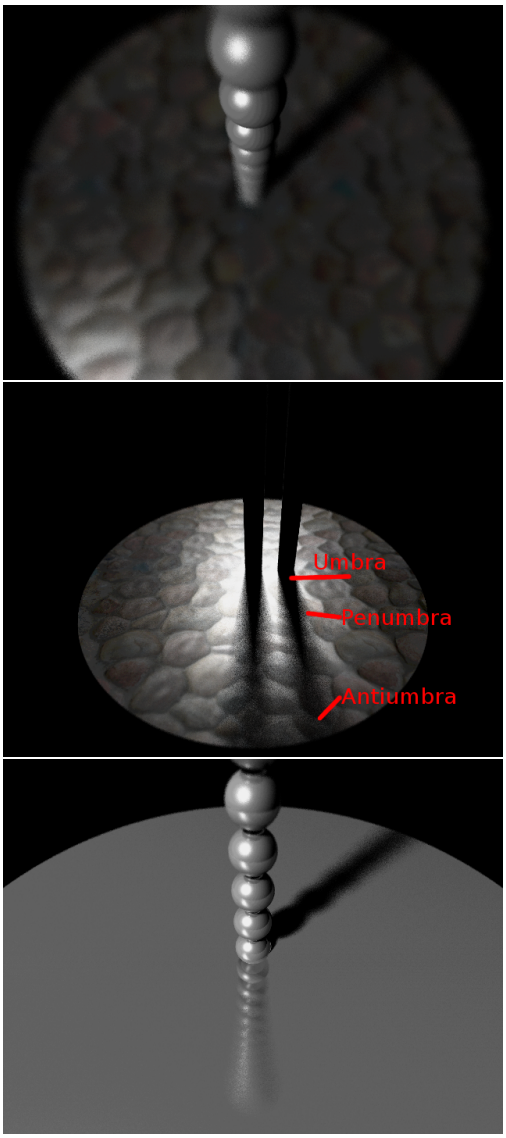


Figure 5: *Top: Depth of Field, Center: Soft shadows (labeled), Bottom: Specular Reflections*

geometry that we need to render. We are currently perturbing by a random xyz value that is not uniform over what a realistic space for distributed ray tracing would be. We have found this to produce acceptable results.

The perturbation of the source of the ray in depth of field needs to be done only once for every ray emitted. It is not necessary for secondary bounces.

7.2 Soft Shadows

Soft shadows can be calculated in the lighting stage by perturbing the location of the light around where the light's center is. The surface then shoots a single ray off at where it perceives the light to be. If the light is in shadow for that ray, then that ray does not contribute a lighting component. If it is in view of the light, then both specular and diffuse components are added. This is an acceptable way to simulate soft shadows since any points within the umbra of other objects will have no shadow rays able to reach the light at all. Objects in the penumbra have a stochastic possibility of hitting the light source based on how much of the area of the light is visible by the light's position modified by the random distribution. A side effect of this is any points that are in the antumbra of the light source will continue to have the appropriate amount of light on them because of the mechanism of simulation.

7.3 Specular Reflections

Specular reflections are used to simulate surfaces that are not completely reflective. This helps to add to the realism of the scene because most objects we observe in life that are shiny are not perfectly shiny. We can perturb the normal once the normal has hit the surface; this causes all secondary rays to be perturbed by the random amount and to simulate a more realistic, imperfect surface.

It is advised to keep this separate from the diffuse/specular lighting terms. These terms help provide a large degree of approximation that would be extremely costly if all that was being done was to see if the surface point reflected directly into the light.

8 Conclusion

We have shown that by utilizing distributed ray tracing with BVHs, we have been able to accelerate our system to provide near-interactive highly realistic environments utilizing readily available consumer hardware. While it does not appear to be useful for video games at the moment, it is conceivable that at some point in the near future, more geometry and pixels could be added to techniques similar to this to help produce highly realistic realtime environments.

The use of parallel traversal aids in systems with high coherence which can be helpful when attempting DRT. By making use of several of these systems together, we are able to reduce the $rays$ and n terms in Equation 2.

In doing so, we have been able to make a ray tracer that performs well when attempting to perform DRT and does not have the same asymptotic properties as the above equations in practice. We can see from Figure 6 that our performance does not decrease linearly with n. This is ideal for a ray tracer that is intended to be scalable.

9 Future Work

It would be useful to be able to examine different forms of BVHs in order to see if AABB's or other such shapes would be beneficial to the algorithm's traversal. Since most of this paper is agnostic to the bounding algorithm used, it should be possible with minimal effort to analyze other algorithms such as k-D trees, taking advantage of benefits like early-intersection.

More real-world experiments with parallel traversal will be needed to determine in what situations it is most, and least effective. This could be done by running a single-ray-per-pixel tracer twice as many times as a parallel-tracing one and comparing the two times.

It would also be interesting to investigate different values for the width factors of bounding trees in order to optimize for the width-factor of the bounding trees. This could affect the running time on different hardware with different input parameters and data sets.

Modifying the system in order to enable a non-binary tree structure increased frame rate. It is conceivable that the choice of how to split the scene up could have a high degree of dependence on the hardware on which the system is operating.

Using a GPU-based random function, such as MD5 [Curtis 2009], may be useful for accelerating the run-time performance by remov-

ing the need for a random texture and thus enabling the cache to be used for more important data such as the tree data.

Acknowledgments

I'd like to thank Dr. Marc Olano for his assistance and guidance in this project. I'd like to thank the UMBC MC2 group for the use of a GeForce 9800GT and Chris Putsche for the use of a GeForce 9800GTX+. Will Murnane helped with LaTeXand provided input on the original algorithm. David Chapman helped find references for various papers. David Chapman, Will Murnane and Mary Lohr helped edit the final paper.

References

BLINN, J. F. 1978. Simulation of wrinkled surfaces. *Computer Graphics 12*, 3 (August), 286–292.

BOULOS, S., EDWARDS, D., J DYLAN LACEWELL, J. K., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2007. Packet-based whitted and distribution ray tracing. *Proceedings of Graphics Interface 2007.*.

CARR, N. A., HOBEROCK, J., CRANE, K., AND HART, J. C. 2006. Fast gpu ray tracing of dynamic meshes using geometry images. *Graphics Interface*, 203–209.

CHRISTEN, M., 2005. Ray tracing on the gpu, January.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *SIGGRAPH*, 137–145.

CURTIS, A. 2009. *Real-time Soft Shadows on the GPU via Monte Carlo Sampling*. Master's thesis, University of Maryland, Baltimore County.

GÜNTHER, J., POPOV, S., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Realtime ray tracing on the gpu with bvh-based packet traversal. *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing.*.

HAINES, E. 1987. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications 7*, 11 (November), 3–5.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree gpu raytracing. *Presented at I3D*.

NVIDIA. 1993-2007. vector_math.h. http://stdcuda.googlecode.com/svn/trunk/stdcuda/vector_math.h.

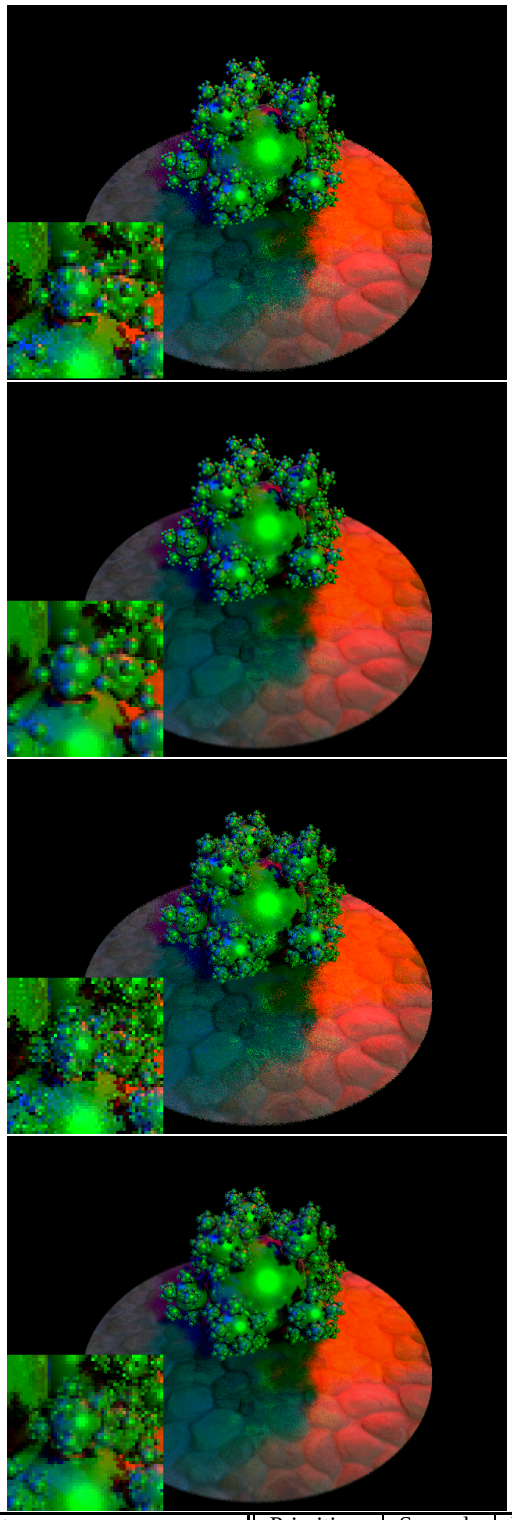
POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (Sept.), 415–424. (Proceedings of Eurographics).

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable hardware. *ACM Transactions on Graphics. 21*, 3, 703–712.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. *SIGGRAPH*, 485–493.

WHITTED, T. 1980. An improved illumination model for shaded display. *ACM Transactions on Computing Machinery 23*, 6 (June), 343–349.

XIANFENG, G., J., G. S., AND HUGHES, H. 2002. Geometry images. *article ACM Transactions Graphics 21*, 3, 355–361.



Test	Primitives	Seconds	Pos
Balls(s 3) (2xAA) Parallel	821	.35	1
Balls(s 3) (16xAA) Parallel	812	2.11	2
Balls(s 3) (16xAA) Naive	821	2.52	
Balls(s 4) (2xAA) Parallel	7382	1.1	3
Balls(s 4) (16xAA) Parallel	7382	8.28	4
Balls(s 4) (16xAA) Naive	7382	9.93	

Figure 6: *Demonstration of performance between varying levels of geometry. All timings are taken on a GeForce 9800GT.*