Ray Tracing: State of the Field Report

Justin Springer

Abstract

Ray tracing is a rendering technique used to generate images. These images are of a significantly higher quality than images produced through rasterization techniques. In order to produce realistic images and dynamic scenes, ray tracing techniques must become the standard used in the computer graphics industry. Information was gathered on hardware improvements, acceleration structures, and expert opinions that suggests ray tracing is a viable alternative to rasterization, with the potential to surpass ray tracing in popularity and usage. With the continual advances in research on topics related to ray tracing, ray tracing techniques will become the standard rendering techniques used to produce high quality images and dynamic scenes.

Contents

List of Figures

1 Introduction

Computer graphics have evolved to the point where it is used in everyday life on a daily basis. Entertainment is the largest provider of these sources, especially video games and movies. There are more developments involving computer graphics being made in other fields as well, such as chemistry and biology. So far, video games have pushed the limit of what computer graphics can achieve and these advances are continuing today. With a general understanding of the algorithms behind computer graphics, we are able to see how the advances continue to progress. We are also able to theorize how long this progression will continue and what technological advancements are necessary for the next step in computer graphics to unfold. An analysis of ray tracing algorithms and a comparison to other computer graphics algorithms will unveil a leading trend that may demonstrate the next step in computer graphics. All this information will lead to the conclusion that ray tracing algorithms will continue to improve on the computer graphics of today.

2 Survey of Computer Graphics

The idea for ray tracing began when Arthur Appel began developing ways to automatically shade line drawings. The line drawings of the time left much up to the imagination. Solid lines and dashed lines were used to show the third dimension of an object, but it was impossible to know where the object was in relation to other objects. Appel believed shading could "specify the tone or color of a surface and the amount of light falling upon that surface from one or more light sources" [1]. He developed algorithms that began by projecting lines of sight from the observer to the object. He would determine if the lines pierced an object and the amount of light the object was receiving from the pierced side. He used this information to draw symbols on the line drawing, with larger symbols meaning less light. This idea was the beginning of ray tracing and led to the development of ray tracing algorithms in use today.

2.1 Quick Timeline

Computer Graphics started with Ivan Sutherland's Sketchpad in 1963. The Sketchpad "allowed interactive design on a vector graphics display monitor with a light pen input device [7]. Later on in the 60s Author Appel developed a few algorithms that are considered the pre-cursors to ray tracing. At this point in time, computer graphics amounted to the ability to draw lines and later circles. Beginning in the 70s, rendering and shading was discovered along with visibility algorithms that determine which pixels are visible to the viewer. A few years later, recursive ray tracing was developed by Turned Whitted and became popular. Old school arcade games like Pong were examples of computer graphics in this era. Research into computer graphics continued through the 80s with some specific research being done in ray tracing. Starting in the 90s, "OpenGL became the standard for graphics APIs and has continued to be a leader in this area [7]. The Nintendo 64 was released a few years later and demonstrated the power of computer graphics in the 90s. Doom and Quake also hit stores in the late 90s and here we see video games pressuring the upper bounds of computer graphics [7]. Today, graphics cards are becoming increasingly more powerful as well as computer processors, and this has led to an increase in the quality and efficiency of generating computer graphics.

2.2 Rasterization Algorithm

Although ray tracing began in the 60s, the algorithms were inefficient and required excessive computational power. This allowed rasterization algorithms to become popular, due to the efficiency and less-intensive computation costs of running the algorithm. Rasterization algorithms consist of looping over the objects in the scene and computing which pixels are covered by the object [5]. The pixels are colored to match the object and stored until they are displayed. Depth buffers are used to determine which object is closer to the screen and ensures the pixel color matches the closest object. The process of rasterization has become analogous to a pipeline. The scene is created using triangles, local coordinates are transformed into world coordinates, and lighting is applied. Afterwards, the scene is clipped to exclude objects outside of the viewing area and the pixels values are determined. The last step involves applying the colors and textures to the pixels. This pipeline was a standard in computer graphics and "lends itself to highly parallel hardware implementations" [5]. As technological advances increased the speed and efficiency of graphics processing units (GPUs), the graphics pipeline became virtualized onto the GPUs [5]. This allowed rasterization algorithms to become far superior to other rendering alternatives, such as ray tracing, due to the fast rendering times of the images and the low computational costs. Rasterization algorithms are the standard algorithm of choice when it comes to creating computer graphics quickly, efficiently, and of a decent quality.

2.3 Ray Tracing Algorithm

Ray tracing algorithms were considered slow, but created high quality images. The algorithms were generally used to create static images and visual effects for films. The core concept of ray tracing is to project rays of light from the viewer towards the object. Each pixel on the screen will account for a single ray, and this is done to guarantee that only rays which hit the screen are traced. The rays are sent out until they collide with an object, and then new rays are sent from that point of collision if reflection or refraction has occurred. This algorithm is done recursively each time there is a collision with a reflective or transparent object. The rays will end when it collides with a non-reflective and non-transparent object. "Shadow rays" are also sent out upon an intersection with an object towards the light source to determine which objects are visible to the light source. The algorithm will combine the colors returned from each recursive call and that will be the final color displayed [6]. Ray tracing provides higher quality images than rasterization, but requires a higher computational cost. The sudden increase in popularity for ray tracing is due to the increase in computational power of modern day graphics cards. One example would be that ray tracing works well with parallel computing, since each ray can be calculated independently. As graphics cards continue to improve, ray tracing will become the standard in computer graphics.

3 Current State of the Field

Rasterization algorithms has had a significant impact on the world of computer graphics. Being a fast and efficient algorithm for generating images, rasterization quickly became the standard algorithm of use. In order to generate larger more complex images at a faster speed, the rasterization process has been streamlined onto graphics cards. This has ingrained the rasterization process into the hardware that is used daily. Due to this fact, the rasterization algorithm has been and will continue to be used in the foreseeable future.

3.1 OpenGL Standard

The standard today consists of OpenGL and rasterization with z-buffer algorithms, discussed in section 3.1.1, and various shading algorithms. OpenGL supports all of the operating systems currently available and most languages are taking advantage of OpenGL through bindings. OpenGL has continued to be the standard since its release in 1992 through continual updates. After the transfer of control of the OpenGL API standard to the Khronos Group, yearly releases after the first initial release became a pattern. The next version to follow this pattern will release in the summer of 2014. Most of the releases involve adding extensions into the core API of OpenGL, which helps tailor OpenGL to the needs of the users. These releases have allowed OpenGL to remain competitive while seeking input on which extensions to include into the core API of the next release.

3.1.1 Z-Buffer Algorithm

The Z-buffer algorithm determines which pixels in the scene are visible to the viewer. The algorithm uses a buffer or 2-d array to keep track of the foremost pixel that is in front of the viewer. As the objects in the scene are rasterized, the buffer is updated with a new color value if the new object's z value is greater than the buffer's current z value for the pixel coordinates. Once all the objects are rasterized, the buffer is traversed and the color values are output. Without the z-buffer algorithm, the objects could intersect and cause a mess of the image. This was caused by the objects being generated as the code states, which could be out of order with how the scene should have been.

4 Technical Analysis: Acceleration Structures

Computer graphics rendering is dominated by rasterization techniques. More recently, ray tracing has seen a resurgence in use due to the increase in hardware computing speed. One of the keys to the rendering techniques is the visibility algorithm. Ingo Wald explains that "to produce an image it is necessary to determine which surfaces are visible from the eye point, which surfaces are visible from the light(s) and hence not in shadow, and, if global illumination effects are being computed, which surfaces are visible from points on other surfaces" [8]. The use of ray tracing will produce the best image from a lighting standpoint.

4.1 Questions when using a Ray Tracer

Creating an image can be a lengthy and challenging task. Highly complex images that include a million triangles have become commonplace in the graphics world. If the scene is animated, the task becomes more difficult. Wald noted a few of these challenges in his state of the art paper on ray tracing animated scenes. The questions are important to the final result of the project and include:

1. "What kind of acceleration structure should be used?"

2. "How is the acceleration structure built or updated each frame?"

3. "What is the interface between the application and the ray-tracing engine?" [8]

These questions will be answered in the following sections.

4.2 Current Strategies

An acceleration structure is used to speed up the process of finding the object a ray has intersected. The purpose of the structure is to sort the objects in the scene. This can be accomplished by many structures, but they fall under two different categories:

- Spatial subdivision techniques

- Object hierarchies

Spatial subdivision techniques divide the scene into areas of equal object count. Object hierarchies divide the scene by primatives, meaning each object is partitioned an area from the scene. Wald describes these structures as such; "spatial subdivision techniques uniquely represent each point in space, but each primitive can be referenced from multiple cells; object hierarchy techniques reference each primitive exactly once, but each 3D point can be overlapped by anywhere from zero to several leaf nodes" [8].

There are advantages and disadvantages for each technique that revolve around traversing and updating the structure. Spatial subdivision structures are easy to traverse, but have the potential to revisit objects and waste time. Object hierarchies have the potential to overwrite intersections of an area due to sub-trees containing the same spatial location, and this will waste time as well. Updating an object hierarchy is easy, since the bounds around each primitive are contained completely within a single node, but the opposite is true for updating a spatial subdivision structure [8]. Each implementation of the two techniques fits a specific goal the designers possessed, and an examination of the different structures will show the necessary time to use the structure. The following sections will examine one spatial subdivision techniques, KD-Trees, and one object hierarchy, Bounding Volume Hierarchy.

4.3 KD-Trees

KD-trees are an implementation of the spatial subdivision technique discussed in the previous section. They are considered the best known acceleration structure for use with ray tracing. KD-trees will speed up the process of ray tracing by organizing the objects within the scene by the location of their points.

4.3.1 Description and Analysis

In order to use a KD-tree structure, memory must be allocated to construct the tree and the root node. In order to create an efficient KD-tree, we must know what the surface area heuristic (SAH) is to determine where to create a split in the scene. The formula to determine the SAH is shown in Section 4.3.2. Kun Zhou then describes the process as follows:

1. Calculate the SAH cost to perform a split at each splitting position in the scene;
2. Pick the split position with the lowest SAH cost and perform the split;
3. Distribute the triangles to each of the split sides such that each side is balanced. [10]

This process is continued for each new node until a certain threshold is met. Figure 1 shows an example of the completed scene partitions as well as the constructed KD-tree.

Now that the KD-tree is constructed, the structure can be put to use within the ray tracing algorithm. The below algorithms taken from [2] shows the pseudocode for a KD-tree traversal method. The method determines which objects a ray will hit as it travels through the scene. The ray begins at the root node and travels down the tree using a stack to keep track of unvisited nodes. The ray will continue to traverse the tree until it reaches a leaf node. Afterwards the ray will terminate and the process will begin again with a new ray.

```
kd-search( tree, ray )
(global-tmin, global-tmax) = intersect (
    tree.bounds, ray )
search-node( tree.root, ray, global-tmin,
    global-tmax )


search-node( nod, ray, tmin, tmax )
if( node.is-leaf )
search-leaf( node, ray, tmin, tmax )
else
search-split( node, ray, tmin, tmax )


search-split( split, ray, tmin, tmax )
a = split.axis
thit = ( split.value - ray.origin[a] ) /
    ray.direction[a]
(first, second) = order( ray.direction[a],
    split.left, split.right )


if( thit >= tmax or thit < 0 )
search-node( first, ray, tmin, tmax )
```

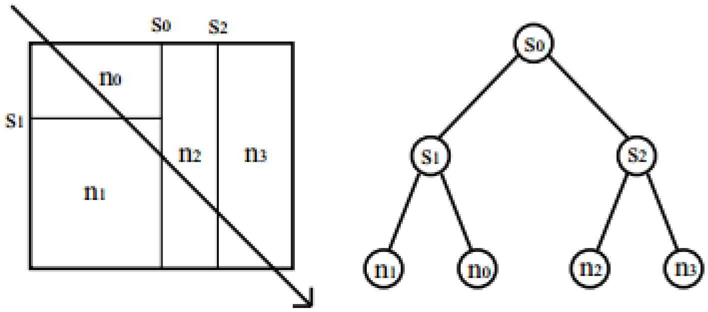

Figure 1: “Left: A two-dimensional kd-tree. Internal nodes are labeled next to their split planes and leaf nodes are labeled inside their volume. Right: A graph representation of the same kd-tree” taken from [2].

```
else if( thit <= tmin )
search-node( second, ray, tmin, tmax )
else
stack.push( second, thit, tmax )
search-node( first, ray, tmin, thit )

search-leaf( leaf, ray, tmin, tmax )
// search for a hit in this leaf
if( found-hit and hit.t < tmax )
succeed( hit )
else
continue-search( leaf, ray, tmin, tmax )

continue-search( leaf, ray, tmin, tmax )
if( stack.is-empty )
fail()
else
(n, tmin, tmax) = stack.pop()
search-node( n, ray, tmin, tmax )
```

4.3.2 Surface Area Heuristic

$$C_V(p) = K_T + K_I\left(\frac{SA(V_L)}{SA(V)}T_L + \frac{SA(V_R)}{SA(V)}T_R\right)$$

$$C_{NS} = K_I T$$

The SAH is used to determine an efficient location to split up a scene for use with kd-trees. In the above formulas, taken from [4], $C_V(p)$ is the split cost at the node p and C_{NS} is the cost of not splitting at node p. If $C_V(p)$ is less than C_{NS}, the node is split into two more nodes, otherwise the node is a leaf node and not split. K_T is the cost of traversing the node, K_I is the cost of the triangle intersection, $SA(V_L)$, $SA(V_R)$, $SA(V)$ are the surface area of the left, right, and current node, and T_L, T_R, T are the number of triangles in the left, right, and current node. This formula is used for each splitting location to determine the best location for the next split.

4.3.3 Issues and Open Research

KD-trees are fast when it comes to traversing the structure, especially with static scenes. The issue for KD-trees is the costly updates required when using a KD-tree to construct a dynamic scene. The KD-tree will have to be made from scratch for each new frame. This is due to the way the objects are sorted. Adjusting a partition line that separates the objects may result in unseen changes to other objects. There are three ideas that may solve the problem with KD-trees. One idea is to build the KD-tree fast enough through parallelization and other methods such that the structure can keep pace with the scene. Another idea is to transform the rays instead of the geometry. Finally, the KD-tree can be lazily rebuilt such that only the sub-trees that are traversed by rays are rebuilt [8]. There are many more ideas to be explored with KD-trees to further improve the speed and efficiency of the structure.

4.4 Bounding Volume Hierarchy

Bounding volume hierarchies (BVH) were less used when KD-trees became the standard. Being an object hierarchy structure and having slower render times than KD-trees, BVHs fell out of favor. With the renewed interest in using ray tracing for dynamic scenes, the advantages of BVHs became evident. BVHs are particularly adept at constructing dynamic and deformable scenes.

4.4.1 Description and Analysis

BVHs are "trees that store a closed bounding volume at each node. In addition, each internal node has references to child nodes, and each leaf node also stores a list of geometric primitives" [9]. The bounding volume shape is typically a box that is aligned on the axes, otherwise known as an axis-aligned bounding box (AABB). Figure 2 shows an example of a BVH for four triangles that splits the triangles into two pairs.

```
function partitionSweep(Set s)
bestCost = T_{tri}*|S| {cost of making a leaf}
bestAxis = -1, bestEvent = -1
for axis = 1 to 3 do
sort S using centroid of boxes in current axis
{sweep from left}
set S1 = Empty, S2 = S
for i = 1 to |S| do
S[i].leftArea = Area(S1) {with Area(Empty) =
    \infty}
move triangle i from S2 to S1
end for
{sweep from right}
S1 = S, S2 = Empty
for i = |S| to 1 do
S[i].rightArea = Area(S2)
{evalutate SAH cost}
thisCost = SAH(|S1|, S[i].leftArea, |S2|,
    s[i].rightArea)
move Triangle i from S1 to S2
if thisCost < bestCost then
bestCost = thisCost
bestEvent = i
bestAxis = axis
end if
end for
end for
if bestAxis = -1 then {found no partition
    better than leaf}
return make leaf
else
sort S in axis 'bestAxis'
S1 = S[0..bestEvent); S2 = S[bestEvent..|S|)
return make inner node with axis
'bestAxis'
end if
```

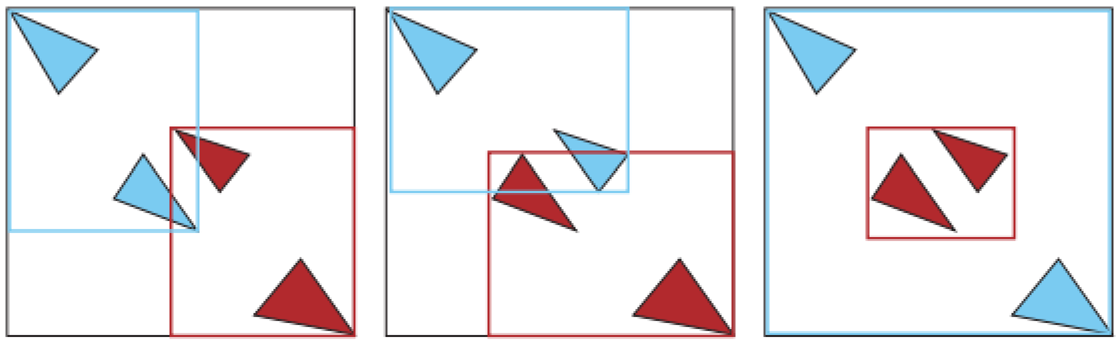

Figure 2: Examples of a BVH taken from [9]

```
end
```

The algorithm above, taken from [8] is used today to create the most efficient BVH structure. The algorithm uses the SAH function from section 4.3.2 to determine what split location will be the most cost efficient for the BVH structure. The algorithm assumes the scene is comprised of triangles, but can be altered to work with any primitive. The end result of the algorithm is a BVH tree structure with a balanced amount of primitives throughout the branches of the tree.

With construction of the BVH structure complete, traversal is the next step. Traversal in a BVH tree is straightforward. When a ray hits one of the nodes in the BVH tree, the children nodes are tested for overlapping bounds. If the child node overlaps with the parent node, the ray traversal continues with the child node. This is done recursively until the last node a ray traverses is a leaf node [9].

4.4.2 Issues and Open Research

There are a few issues to be aware of when working with BVHs. The nodes have the potential to overlap the same are, and this can lead to some issues with ray intersection. The other issue to consider is the enclosure of the primitives by the BVH. BVHs will not be as tightly bound around a primitive as a KD-tree would. This can lead to more primitive intersections that need to be accounted for. Research is continuing to improve the efficiency and speed of BVHs. Wald believes a "set of animation benchmarks" would allow better comparisons between the different approaches of BVHs [9]. The benchmark would be a better guide to determine what techniques work and which ones need more fine-tuning.

5 Future Trends

Ray tracing has become an important technique for displaying computer graphics. It was not always as useful as it is today. There have been many technological discoveries and improvements that have made ray tracing a reality. Ray tracing has become viable as CPUs and GPUs have become more powerful. This increase in computational power has overcome the main problem with ray tracing: high computational costs. The technological advances will continue to improve computing power, thus changing the field of computer graphics. This change will involve rendering of dynamic scenes, and the acceleration structures that accompany ray tracing algorithms. The acceleration structures are a topic of much research today. There are a few common structures that have seen changes to improve build speed, traversal speed, and intersection detection. These improvements have increased the viability of using ray tracing to render dynamic scenes. Lastly, ray tracing programs are lacking when compared to the rasterization program, OpenGL. As ray tracing becomes widely used, a reliable ray tracing program or application programming interface will need to be developed to strengthen core values of ray tracing.

5.1 Rasterization and CPU/GPU Improvements

Computer graphics is a young field when compared to other areas of research. This field has been dominated by the rasterization algorithm used to display graphics on a screen. Rasterization has many advantages that make it the ideal choice for computer graphics. The algorithm is fast and efficient, while still maintaining quality graphical output. This has kept rasterization as the main algorithm in use today. Ray tracing has been around for a long time as well, but had some flaws that deemed the algorithm inefficient. Ray tracing required a lot of computational power to create the high quality images it output. It was unable to be used for dynamic scenes, and it was used mainly for off-line rendering.

CPUs and GPUs of today have become vastly more powerful than their older counterparts. With this increase in computing power, ray tracing has become a more viable option to be used in everyday computer graphics. This change has occurred over the last decade for GPUs specifically, but improvements to CPUs have occurred continuously over time. Forrest noted these changes over a decade ago when he stated graphics cards will have "support for rendering techniques such as ray tracing, shadows, path tracing, and photon mapping" [3]. GPUs will become the main source of the computational power needed to run a ray tracing algorithm. Specialized GPUs have been made to work efficiently with rasterization by converting the rasterization pipeline into simple instructions preloaded onto the graphics card. This idea has been implemented today, but will see more widespread usage in the coming years as specialized GPUs are made to work with the ray tracing algorithm.

5.2 Acceleration Structure Trends

There are a few main acceleration structure techniques in use as of today. The main techniques are kd-trees and Bounding Volume Hierarchies. A third technique is the Grid-based technique that is similar to a kd-tree, except the algorithm uses consistent split lines to separate the objects in the scene. The Grid-based technique is not used as heavily as the other techniques, and this will continue to be normal. Kd-trees and BVHs are stronger acceleration structures than a Grid-based technique. A combination of kd-trees and BVHs are going to be common in the coming years. This combination would separate the objects in larger groups using BVHs and then separate the objects even further with the kd-tree. The kd-tree would be contained within the BVH, to put it simply. This would allow the acceleration structure to be updated easily with the outer BVH, since the strength of the BVH is in the easily update-able nature of the structure. The inner kd-tree would allow the structure to be easily navigated, as is the strength of the kd-tree. The combination of the two structures is the logical next step in the process of finding the fastest, most efficient acceleration structure.

An skd-tree is an example of this future trend. Discussed briefly in Wald's State of the Art report on ray tracing animated scenes, an skd-tree is a structure similar to a kd-tree, but "behaves like a bounding volume hierarchy" [8]. The structure can be updated relatively easily, but the traversal can be slower in some cases. Some experiments have shown similar performance speed to a BVH, but it has not been fully investigated yet [8]. Acceleration structures that combine the known structures, or aspects of different structures, will continue to improve the performance of ray tracing algorithms. This will lend itself to the improvement of ray tracing dynamic scenes as well.

5.3 Ray Tracing Programs

Rasterization has seen such widespread usage due to the OpenGL application programming interface. OpenGL simplified the way computer graphics were made and has continued to be the top API used for rasterization. OpenGL does not support ray tracing, but another API was created similar to OpenGL for the purpose of supporting ray tracing exclusively, titled OpenRT. OpenRT was modeled after OpenGL, since OpenGL was very successful in the way it accomplished using rasterization. OpenRT allowed users to create graphics applications that utilized the power of ray tracing. OpenRT was not open source, therefore it was not widely in use. A new ray tracing API would be useful for the further development of ray tracing techniques, and also to introduce new users to ray tracing in a user-friendly manner. With OpenGL being relevant for rasterization, it would be unsurprising if a ray tracing API would be released within the next five years.

6 Demonstration

In order to demonstrate some of the differences between rasterization and ray tracing, I generated some images using both algorithms with the program, Blender. These images will clarify what is possible with each rendering algorithm. I will also compare the build speed and file size of the images, in order to show the differences in computational power needed to run each algorithm. Finally, I will discuss some of the issues I had while working on the demonstration.

6.1 Image Comparisons

In Figure 3, a sphere was created and rendered with rasterization. The image appears flat and the shadows do not look the greatest. Compared to Figure 4, this image was rendered using ray tracing and has

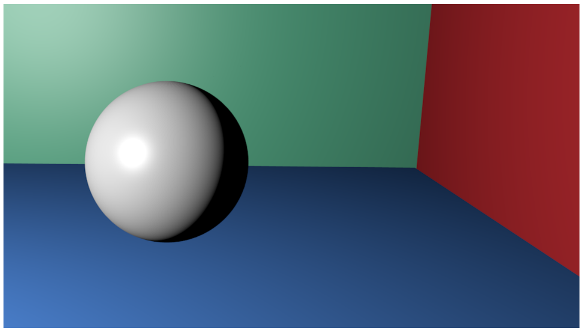

Figure 3: A sphere in front of three different colored planes rendered using rasterization.

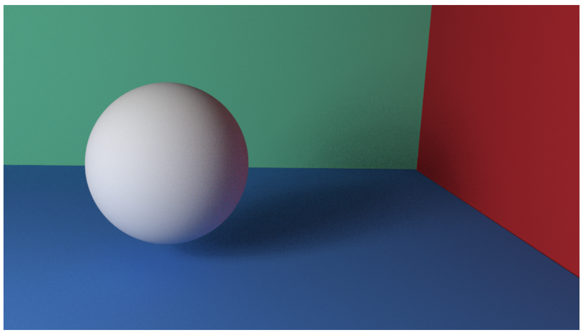

Figure 4: A sphere in front of three different colored planes rendered using ray tracing.

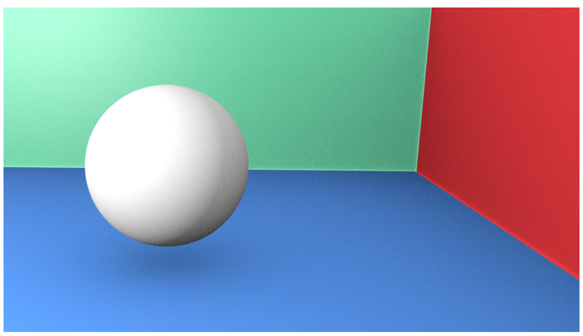

Figure 5: A sphere in front of three different colored planes rendered using rasterization with ambient occlusion.

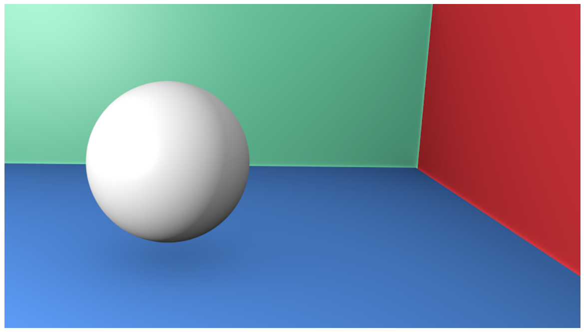

Figure 6: A sphere in front of three different colored planes rendered using rasterization with approximate ambient occlusion.

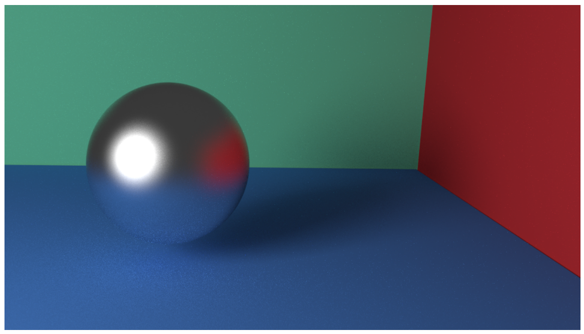

Figure 7: A glossy sphere in front of three different colored planes rendered using ray tracing.

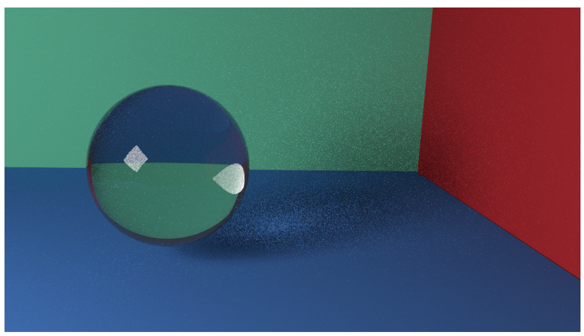

Figure 8: A glass sphere in front of three different colored planes rendered using ray tracing.

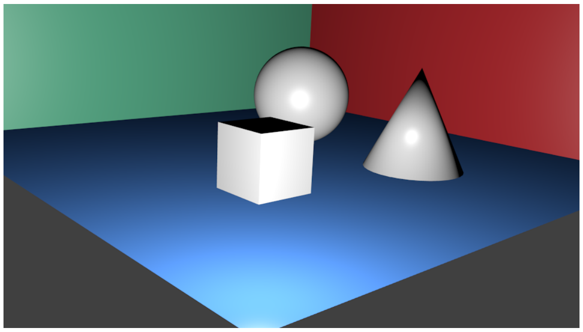

Figure 9: Three shapes in front of three different colored planes rendered with rasterization.

more depth and better shadows. There are some techniques used to improve rasterization, and one such technique is ambient occlusion. Figure 5 uses ambient occlusion and Figure 6 uses ambient occlusion approximation. Ambient occlusion takes into consideration ambient light in the environment. This technique brightens the shadowed areas in the image, but it does not generate shadows that are shown in Figure 4. Figure 7 demonstrates the reflective rays and Figure 8 demonstrates the refractive rays that are possible with ray tracing. I was unable to generate the same kind of image with rasterization. A texture could be used to make the image similar, but the shadows would be missing, as well as the green, blue, and red colors on the sphere caused by refracting and reflecting rays. I believe the image in Figure 4 is the highest quality image, due to the shadows behind the sphere and the slight coloration of the sphere from the light reflected off of the walls.

6.2 Build Speed And File Size

The next few images were generated and the time it took to generate was recorded. The images were

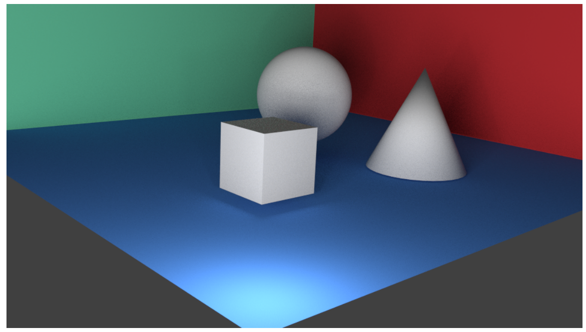

Figure 10: Three shapes in front of three different colored planes rendered with ray tracing.

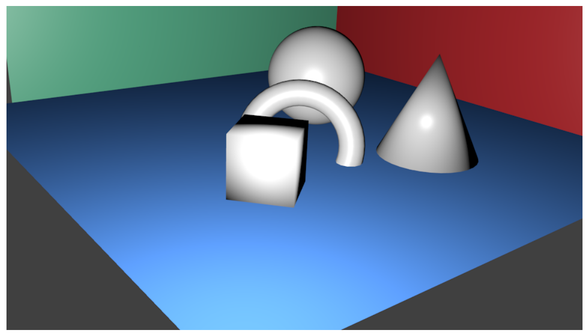

Figure 11: Four shapes in front of three different colored planes rendered with rasterization.

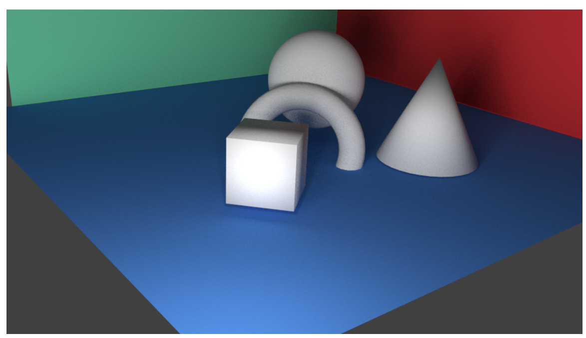

Figure 12: Four shapes in front of three different colored planes rendered with ray tracing.

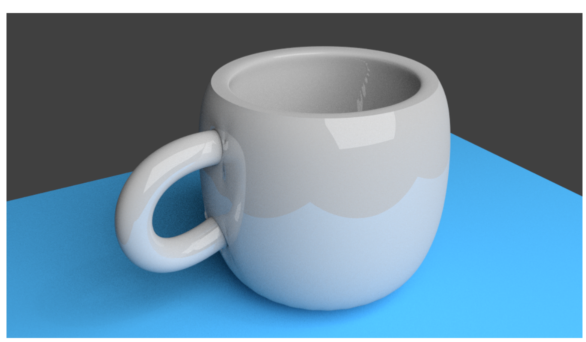

Figure 13: A ceramic cup.

saved as a .png file. Figure 9 was generated in 0.39 seconds and the file size is 185 KB. Figure 10 was generated in 1:06 minutes and the file size is 639 KB. The rasterization algorithm was 169 times faster than the ray tracing algorithm, and the file size was 3.5 times smaller. With the addition of one more object as shown in Figure 11 and Figure 12, the rasterization render speed increased to 0.74 seconds with a file size of 227 KB and the ray tracing render speed increased to 1:09 minutes with a file size of 685 KB. This translates into about a 90% increase in build time for the rasterization algorithm and about a 5% increase in build time for the ray tracing algorithm. There was about a 23% increase in file size for the rasterization algorithm and about a 7% increase in file size for the ray tracing algorithm. Figure 13 took 1:43 minutes to render using ray tracing and had a file size of 555 KB. The file size was smaller than the other ray traced images, but the build time was longer. This would be caused by a more complex image with less color variance throughout. From what was shown by the image statistics, it can be concluded that ray traced images take longer to render and have a larger file size. This follows the ideas presented above stating ray tracing takes a longer time to execute, but produces a higher quality image.

6.3 Issues With Demonstration

Throughout the time dedicated to learning the Blender program and the intricacies involved in creating the different images, I was unable to create a dynamic or animated scene. The amount of work necessary for a decent quality scene with the ability to move through the scene or an animated walk-through of the scene was higher than I anticipated. With the addition of moving objects, the difficulty increased. Therefore, I was unable to complete the dynamic scene portion of the demonstration I was hoping to show.

7 Conclusion

Computer graphics have advanced rapidly through the course of fifty years. What started out as simple dots and lines on raster screens, quickly became the highly advanced and beautiful looking graphics we know today. Rasterization algorithms are and will continue to be the standard rendering algorithm in use, but with continual improvements to ray tracing algorithms, the standard will change. Rasterization is ingrained into every part of the rendering pipeline. If ray tracing is to replace rasterization, this same idea will need to be used with ray tracing. A rendering API similar to OpenGL that uses only ray tracing will need to be developed as well. Dynamic scene rendering using ray tracing is already a reality, and with the continual research of acceleration structures, will be faster and more efficient in the following years. The demonstration I presented also shows the advantages and disadvantages of using ray tracing. The images will take longer to render, but the high quality images are worth the extra time needed to render. Ray tracing will become the standard rendering algorithm in use, if a ray tracing API is developed, acceleration structures continue to improve, and specific hardware caters to the ray tracing process.

A Previous Work

Throughout my college career I was able to learn how to program using Java. This basic knowledge of the language continued to expand and I was able to learn the basics of other languages. This allowed me to learn quickly how to use new programs at a basic level. With continual study and practice, I am able to develop a knowledge of a language or program that I am able to work with. There are still many things I do not know and that I would need to know, but I am able to function with the knowledge I have. This translates into the working knowledge I developed in order to use the program Blender. I switched over to Blender later into the project than I should have, but it was a better program to use instead of the first program I planned on using, POV Ray-Tracer. Due to switching late to the Blender program, I wasn't able to learn more complicated concepts needed to work with the program.

I took the computer graphics course at St. John's University while I worked on this paper. I was able to learn about rasterization and apply that knowledge in programs I made for the class. This allowed me to develop a better understanding of rasterization that I could then use to compare to ray tracing. It also helped me learn about the inner workings of rasterization and the graphics pipeline. The pipeline is a necessary topic to understand, in order to realize why rasterization is so dominant in the computer graphics world. The computer graphics course was also interesting and an overall great experience that complimented my research topic for the semester.

During my college career, I was able to take a few course that focused on a semester long project. These courses taught me how to make continual progress. I learned how to break up larger parts into more manageable smaller tasks. This became helpful when trying to conquer multiple papers, presentations, and projects. The skills learned from the semester long projects were one of the most beneficial to learn for future job prospects as well as this research project. The ability to break apart larger projects into smaller tasks will be invaluable for future endeavors.

References

[1] Arthur Appel. Some techniques for shading machine renderings of solids, 1968.

[2] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer, 2005.

[3] A. R. Forrest. Future trends in computer graphics: How much is enough? *Journal of Computer Science and Technology*, 18(5):531–537, 2003. ISI Document Delivery No.: 725NA Times Cited: 0 Cited Reference Count: 8 Forrest, AR 0 Science china press Beijing.

[4] Keith Lantz. Kd tree construction using the surface area heuristic, stack-based traversal, and the hyperplane separation theorem, 2013.

[5] Steven G. Parker, Heiko Friedrich, David Luebke, Keith Morley, James Bigler, Jared

Hoberock, David McAllister, Austin Robison, Andreas Dietrich, Greg Humphreys, Morgan McGuire, and Martin Stich. Gpu ray tracing. *Commun. ACM*, 56(5):93–101, 2013.

[6] Paul Rademacher. Ray tracing: Graphics for the masses.

[7] William Shoaff. A short history of computer graphics, 2000.

[8] I. Wald, W. R. Mark, J. Gunther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. *Computer Graphics Forum*, 28(6):1691–1722, 2009. ISI Document Delivery No.: 495OE Times Cited: 11 Cited Reference Count: 112 Wald, Ingo Mark, William R. Guenther, Johannes Boulos, Solomon Ize, Thiago Hunt, Warren Parker, Steven G. Shirley, Peter National Science Foundation [0541009, 0306151, 0546236]; U.S. Department of Energy [W-7405-ENG-48LA¿-13111-PR¿]; Intel Corporation The writing of this survey has been supported by the National Science Foundation (awards 0541009, 0306151 and CAREER award 0546236), by the U.S. Department of Energy through the Center for the Simulation of Accidental Fires and Explosions (grant W-7405-ENG-48LA ¿-13111-PR ¿), and by research grants from Intel Corporation. The authors would particularly like to thank Jim Hurley at Intel, who has strongly supported academic ray tracing research over the past several years. 12 Wiley-blackwell publishing, inc Malden.

[9] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6, 2007.

[10] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware, 2008.