5-2014

Efficient Ray Tracing For Mobile Devices

Rafael De Melo Aroxa
University of Arkansas, Fayetteville

Efficient Ray Tracing For Mobile Devices

Efficient Ray Tracing For Mobile Devices

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science

by

Rafael de Melo Arôxa
Universidade Federal de Pernambuco
Bachelor of Science in Computer Science, 2011

May 2014
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

_____
Dr. John Gauch
Thesis Director

_____
Dr. Craig Thompson
Committee Member

_____
Dr. Tingxin Yan
Committee Member

ABSTRACT

The demand for mobile devices with higher graphics performance has increased substantially in the past few years. Most mobile applications that demand 3D graphics use commonly available frameworks, such as Unity and Unreal Engine. In mobile devices, these frameworks are built on top of OpenGL ES and use a graphics technique called rasterization, a simple concept that yields good performance without sacrificing graphic quality. However, rasterization cannot easily handle some physical phenomena of light (i.e. reflection and refraction). In order to support such effects, the graphics framework has to emulate them, thereby leading to suboptimal results in term of quality.

Other techniques, such as ray tracing, do not require such emulation to be implemented, as the aforementioned phenomena of light are inherently considered. In this work, we first design and implement a 3D renderer that uses ray tracing to generate high quality graphics for mobile devices. Our implementation yielded high quality results but at a high computational cost, which impacted performance. To alleviate this problem, we then developed special algorithms and data structures to substantially improve the performance of the rendering engine. In our experiments, we achieved frame rates that were 7 to 15 times faster than the brute force approach. Being able to render high quality graphics with good performance can potentially revolutionize the mobile gaming industry. To the best of our knowledge, this has never before been implemented in commonly available devices, such as smartphones and tablets.

ACKNOWLEDGMENTS

Thanks to my advisor, Dr. John Gauch, for all the support and feedback given, and for the valuable directions that led to successful completion of this work.

Thanks to Dr. Craig Thompson and Dr. Tingxin Yan for providing valuable feedback during the early stages of development of this work.

Thanks to my family, for raising me to be who I am, and for being such good inspirations.

Special thanks to Daniel, to whom I owe sincere gratitude for being the best friend one could ever have. For helping me keep my sanity through remote guitar jamming sessions.

Finally and most importantly, special thanks to my wife, who supported me throughout my entire Master's program. I have the most sincere admiration for who she is and for what she has always done for me. I am profoundly grateful for having her in my life.

TABLE OF CONTENTS

I. INTRODUCTION

A. DRIVING PROBLEM

The adoption of smartphones has increased dramatically over time. As of July 2013, the adoption rate among U.S. customers was 61%, which is over 10 percentage points higher than the same time the previous year. As the number of users and devices are increasing, the hardware embedded in them evolves significantly. For that reason, users are now expecting applications with better graphics quality, which is especially true when it comes to mobile games, since gamers are fanatical about graphics.

Most currently available frameworks for rendering graphics in mobile platforms use the rasterization technique, which yields good performance and quality. As it became commonly used over time, various efforts led to the hardware implementation of several portions of this technique, thereby yielding optimal performance. Additionally, most of the smart devices (phones and tablets) available today have specialized graphics hardware embedded into them. Performance varies considerably across devices, but support for graphics APIs (e.g. OpenGL ES [12]) is common among most of them.

Even though rasterization presents a good performance and graphical quality, it has some severe limitations. First, it does not natively take into consideration global illumination techniques and some light phenomena (e.g. reflection and refraction). Second, simple behaviors of the light in the environment, such as shadows, are not a natural outcome of rasterization. These effects often need to be emulated in order to improve image quality. Not surprisingly, this process delivers less than optimal quality when compared to other techniques.

Global illumination techniques are based on the following fact: light rays are emitted from light sources and propagate through the environment. While traveling, these rays are either

absorbed, reflected or refracted depending on the properties of the objects with which they collide. Rays that hit objects with rough surface properties are reflected non-uniformly. This phenomenon is known as diffuse reflection and is also considered in the rendering process. The mathematics behind the traversal of the ray in the environment is given by basic principles of optical physics.

In nature, the rays originate from a light source and are affected by the aforementioned phenomena. These rays eventually hit the eyes of the observer. A naive implementation of a renderer that works exactly like nature would be unfeasible because of the uncountable quantity of light rays emitted by each source. Techniques such as ray tracing use a different approach. In order to make this problem computationally feasible, it makes the rays traverse the environment in the opposite direction, i.e. from the observer to the objects. In this case, a ray is traced for each *pixel*, and the quantity of traced rays is bounded by the quantity of *pixels* present in the rendered image, i.e. its resolution.

In order to calculate the color of a given pixel, the given ray has to be tested for intersection against all objects in the scene. Not surprisingly, this approach does not scale very well as the number of objects increase. There are several data structures that can be used to accelerate the ray traversal. KD-Tree (*k*-Dimensional Tree), BVH (Bounding Volume Hierarchy) and BIH (Bounding Interval Hierarchy) are among the most used ones.

These data structures help alleviate the high computational cost of ray tracing. Enabling mobile platforms with such powerful techniques allows for developers to create more photorealistic applications, which in turn could yield better user experience on mobile games and graphics.

B. THESIS STATEMENT

This work proposes a rendering engine that is capable of generating photorealistic images efficiently in mobile devices. Initially, the renderer implemented in this work will only support Apple's iOS platform. Google Android and Windows Phone are left as future work. Many challenges rise from developing an efficient renderer for mobile devices, especially when hardware constraints of such devices are considered. In order to overcome the problems mentioned earlier, developers often have to find the sweet spot in a trade-off between quality and performance.

Ray tracing is the core rendering technique implemented in this work, and the nature of the algorithms and data structures involved in this process are complex. By using the device's CPU and GPU, the renderer is able to perform parallel computation and therefore improve performance of the ray tracer. This approach also enables the renderer to deliver high quality graphics without sacrificing other aspects of the application, such as user experience and interactivity.

Acceleration data structures are among the most used optimization techniques for renderers. This work will focus on using the BIH (Bounding Interval Hierarchy) to improve performance of the ray tracer. This data structure was chosen because its implementation is simple, and yet it yields remarkable results. The detailed analysis of the algorithms involved in the construction and traversal of the BIH data structure, as well as the various optimizations, is also considered in this work.

C. PREVIEW OF RESULTS

This work showed that it is possible to achieve efficient rendering in mobile devices. In fact, the implementation presented in this work, which is detailed throughout the rest of this document,

yielded performance improvements of up to 1495% over the non-accelerated implementation. We saw that the BIH can take great advantage of scenarios in which there is great quantity of empty spaces between the primitives. We also noticed that by adjusting the BIH parameters correctly, we can achieve a reduction of approximately 80% in memory consumption while still maintaining reasonable performance. Overall, the use of special techniques indicated that good results can be achieved in mobile devices by using ray tracing.

D. THESIS ORGANIZATION

Chapter 2 explains basic concepts about computer graphics and rendering techniques, as well as acceleration methods. Chapter 3 presents the implementation details about the framework and techniques used. This includes specifics about the architecture model used, as well as encapsulation and utilization of the framework by third parties. Chapter 4 discusses the results obtained for various run configurations with regards to rendering time, memory footprint, battery usage and other measurements. Finally, Chapter 5 offers recommendations for improvement and future work.

II. BACKGROUND

A. COMPUTER GRAPHICS

In Computer Science, Computer Graphics (CG) are the result of the use of specialized hardware and software techniques to process mathematical representations of images. One of the most common use of CG is to generate a visual representation of a mathematical model of a 2D/3D virtual world that is comprised of vertices, object primitives and their surface material properties. Another use of CG is in the creation of images whose abstract representation is given only by mathematical equations. These are known as fractal images, and are very common in landscape generation and fluid simulation applications. Several computer graphics techniques exist for the purpose of image generation and they usually trade off quality for performance. For the sake of simplicity, only Rasterization and Ray Tracing are considered in this work.

Rasterization *vs.* Ray Tracing

Rasterization is based on the idea of projecting 3D geometric objects onto a 2D projection plane. A simple algorithm that implements this technique is called *priority fill* [2], commonly known as Painter's Algorithm. In this algorithm, the objects in the scene are sorted and processed one by one based on their distance to the camera, from the farthest to the closest. The coordinates of the objects are then transformed and mapped onto the projection plane in order to obtain the rendered image. In this way, geometric objects such as lines, curves, and polygons are mapped onto pixels on the screen (Figure 2.1).

After the quick adoption of this technique in various types of applications, rasterization rapidly became the main technique for rendering. This happened due to the well-established hardware implementation of a graphic pipeline that runs straight from the video cards. Graphic

frameworks (e.g. OpenGL [11] and DirectX [8]) communicate with the hardware directly to execute the rendering process.

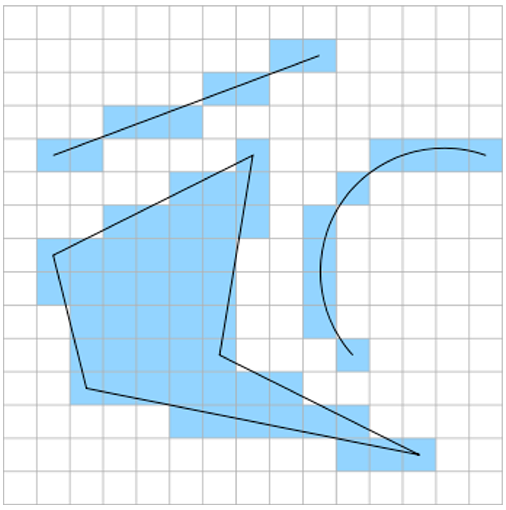


Figure 2.1: Result of rasterizing a line, a curve, and a polygon. Note the presence of jagged edges, which are inherent to this process.

Rasterization yields good image quality, with low computational cost, whereas ray tracing is capable of delivering much better results at a higher computational cost (Figure 2.2). This high cost involved in the ray tracing algorithm is due to the need of handling the intersection of the traced ray with all of the objects in the scene. More precisely, about 95% of the total time spent in the rendering process is related to ray-object intersection calculation [23]. For example, a given scene with 100 primitives being rendered at a resolution of 800x600 *pixels* will perform 48 million intersection tests in a single frame. Each of these tests executes a great quantity of operations with floating point numbers, especially multiplication and division, which are particularly expensive.

Fortunately, the intersection tests are completely independent for each *pixel*, which makes the algorithm ideal for parallelization. Because of this characteristic of the ray tracing algorithm,

relatively high *fps* (frames per second) rates can be achieved even by implementations that have not been thoroughly optimized.

Figure 2.2: Demonstration of *ray tracing* delivering more realistic images.

Fortunately, the intersection tests are completely independent for each *pixel*, which makes the algorithm ideal for parallelization. Because of this characteristic of the ray tracing algorithm, relatively high *fps* (frames per second) rates can be achieved even by implementations that have not been thoroughly optimized.

One of the reasons of ray tracing being computationally feasible in the first place is the assumption that the rays are traced from the camera, going through the projection plane, and hitting the objects in the scene. This is the contrary of the natural behavior of the light in the real world. The rays that originate in the camera are called primary rays. The process responsible for tracing one ray for each *pixel* in the projection plane is called ray casting [1] (Figure 2.3).

One may wonder what makes ray tracing different from ray casting, given that both techniques are very similar. In fact, ray tracing uses ray casting to calculate the result of the intersections for the primary rays. The key difference is that each time a ray hits an object, a secondary ray can be spawned [23]. These rays can be either shadow or indirect illumination rays, or it can be a result of the reflection or refraction of the primary ray.

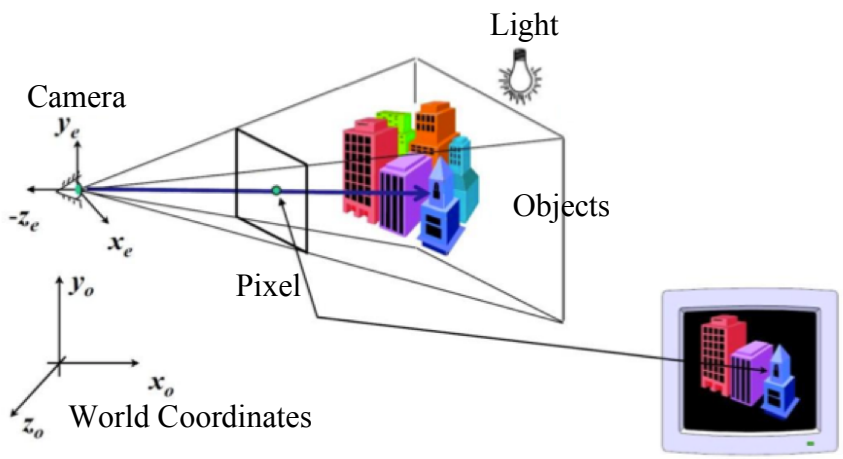


Figure 2.3: Visual representation of the ray casting technique. Note that the vector that originates in the camera focal point and points to the given *pixel* in the projection plane defines the direction of the ray.

Shadow rays are used to determine whether or not the intersection point is in a shadow spot. Indirect illumination rays are responsible for transmitting the color from one object to another. Reflected rays stem from the application of The Reflection Law, and Snell's Law [5] rules the refracted rays (Figure 2.4).

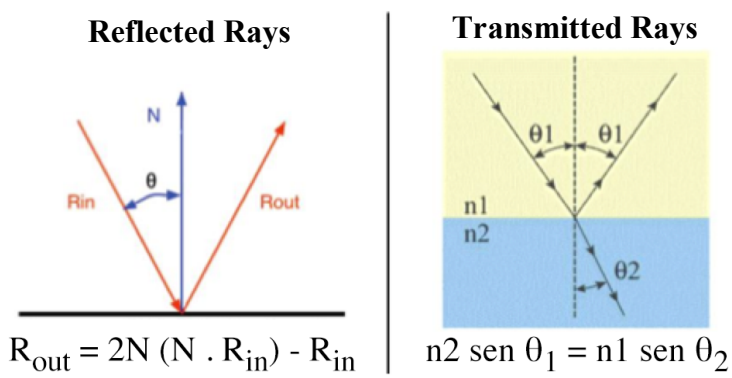


Figure 2.4: Visual representation of the calculations behind the creation of reflection and refraction rays.

One big advantage of using ray tracing in the rendering process is that, contrary to rasterization, the objects are not tessellated into triangles before being projected. This means that

ray tracing can be applied to rendering all types of objects, such as lines, curves, polygons, curved surfaces, etc. Parametric surfaces, such as spheres, cones, quadrics and Bézier surfaces), can be rendered at maximum quality using ray tracing. This ray tracing characteristic greatly affects the final quality of the rendered image.

B. ACCELERATION DATA STRUCTURES

It is rather common for real time implementations of ray tracing to use acceleration data structures. These structures are used in the ray tracer to intelligently detect which objects will be hit by said ray. BVH (Bounding Volume Hierarchy) [20], KD-Tree (*k*-Dimensional Tree) [14][16], and BIH (Bounding Interval Hierarchy) [18] are particular types of BST (binary search tree), which are the most commonly used data structures for this purpose. The use of these data structures in a ray tracer is comprised of two main steps: construction and traversal. In the construction, the 3D scene is analyzed and processed in order to create a data structure that represents the scene state in terms of vertex positions, transformations, object shapes, etc. In the traversal, the information present in the data structure is used to rapidly intersect rays with objects and improve the performance of the ray tracer.

The data structures aforementioned perform the ray traversal in a directed way, thereby avoiding unnecessary ray-object intersection tests. The BIH uses roughly the same principles used in the KD-Tree and BVH in the traversal, but it has a relatively simpler and quicker construction algorithm. This approach yields various advantages when it comes to its utilization in dynamic scenes, as it is necessary to reconstruct the data structure in each frame at interactive *fps* rates.

The different structures present in the literature take advantage of different aspects of the ray tracing process in order to improve its performance. The main aspects lie in the construction and

traversal steps. Algorithms that spend more time in the construction process, such as SAH (Surface Area Heuristic) [21], are able to deliver a better quality data structure. This happens because such technique aims to maximize the empty spaces in the scene and thus minimizes the ray traversal time. The construction of data structure that uses SAH for a relatively complex scene can easily require a great amount of time, which can range from minutes to days [19][21].

A given data structure constructed using SAH yields great performance in the traversal process, which is good for static scenes. However, it becomes unfeasible for dynamic scenes because the reconstruction of the structure at each frame would considerably lower the *fps* rate, thereby depriving the user of the sense of interactivity. Normally, data structures that provide the fastest traversal have their drawback in the construction process. The combined time of the construction and traversal of the data structure in addition to the required in the shading process is commonly known as *time to image* [22], and is equal to the rendering time for a given frame. The construction step affects *time to image* in an inverse proportion when compared to the traversal. In other words, the extra complexity involved in the construction pays off in the traversal, which becomes faster.

BVH

Bounding Volume Hierarchy [20] (BVH) is an object-partitioning tree that uses the concepts of AABB (Axis Aligned Bounding Box) to implement the construction and traversal algorithms. Each node in the tree contains the maximum and minimum coordinates of the AABB that involves it, as well as a reference to each child node. Even though this approach considerably increases memory footprint, it is indeed possible to guarantee that the primitives will be contained in the AABBs whose volume is just enough to contain them (Figure 2.5).

In general, it is possible to affirm that the volume of the AABB of the root node is always bigger than the union of all other AABBs of non-root nodes. This characteristic leads to maximization of empty spaces, which ultimately increases performance in the traversal. The empty space between the nodes makes the traverse algorithm return early.

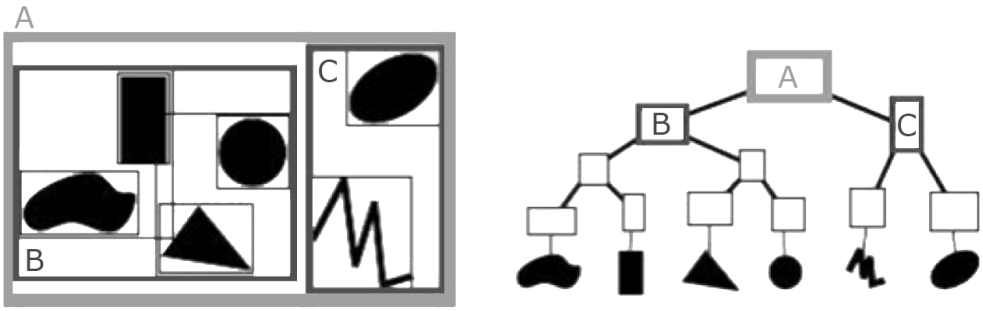


Figure 2.5: Visual representation of how nodes are organized in the BVH. AABBs and their positions on the left, and the constructed tree on the right.

For being an object-partitioning tree, BVH allows for overlapping between sibling nodes. Therefore, if a ray intersects the said overlapping area, the farthest node has to be stacked and later processed. In fact, the traversal algorithm cannot be finished until all nodes are unstacked and the closest intersection is found.

KD-Tree

KD-Trees (*k*-Dimensional Trees) are a special type of BSP Tree (Binary Space Partitioning Tree) that yields optimal performance for rendering static scenes with ray tracing. Contrarily to BVH, the KD-Tree only stores information about one AABB, which is the one in the root node. The construction algorithm can establish a cutting plane for each node by using said AABB. The primitives within that node would then be associated with the left or right child of said node.

In a KD-Tree, the sibling nodes do not overlap, as it is a space (rather than object) partitioning tree. This characteristic inherently makes the construction algorithm more complex, as it has to deal with specific cases in which the cutting plane intersects a primitive object. When

this situation is met, object-plane intersection tests are executed with the purpose of dividing the AABB that contains the object in two smaller AABBS. Each of these smaller boxes is located in one side of the cutting plane, and contains part of the object that was split in this process. This approach to constructing the data structure is efficient from the quality standpoint, but it however increases the memory footprint necessary to store the whole structure. Additionally, when dividing the AABB into smaller boxes, the reference to said object primitive is duplicated on both child nodes, thereby creating data redundancy.

Most KD-Tree implementation use heuristic techniques (e.g. SAH) for selecting the best cutting plane (Figure 2.6). In this case, split operations take place in levels 1, 2 and 3 of the tree, generating the red, green and blue cutting planes respectively. It is important to mention that the leaf nodes are in each side of the blue cutting planes. The use of SAH balances better the traversal cost of the entire tree across all nodes, thereby improving performance.

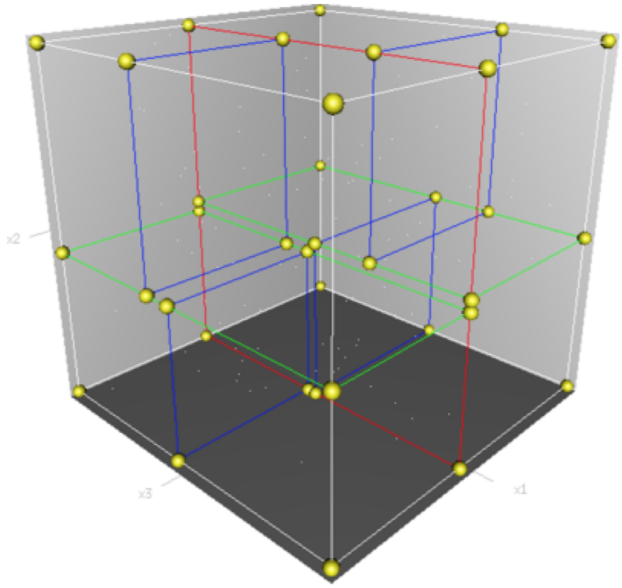


Figure 2.6: Visual representation of the main AABB for the KD-Tree and the hierarchical divisions in each level of the tree.

SAH works by measuring the traversal cost of a given node and of the primitive objects it includes and it takes into consideration several factors. These include the surface area of the AABB of the current node and the ones for each child node, as well as the number of primitive objects in each node. Other constant factors are also considered. For example, the traversal cost of a single node and the ray-object cost for a single intersection test. These help balance the depth of the tree as well as the number of primitive objects to which nodes can refer. The use of SAH results in higher quality trees, whose traversal performance is optimal. However, as the algorithm implemented in this heuristic is complex and costly, its use in dynamic scenes may become unfeasible.

BIH

Bounding Interval Hierarchy (BIH) aggregates characteristics of both BVH and KD-Tree in order to improve construction step performance. Namely, it uses the idea of maximizing the empty space between the nodes that is present in BVH, and it also uses the concept of cutting planes present in KD-Tree. For this reason, the BIH can be considered both a space and an object-partitioning tree, as it takes advantage of both approaches in order to improve performance and quality in the rendering process. Similarly to the BVH, the nodes can overlap in the BIH, thereby requiring a stack to be used in the traversal process.

BIH was created with the intent of minimizing *time to image*, thereby allowing the renderer to deliver high *fps* rates for both static and dynamic scenes. To reach this objective, the construction algorithm for the BIH is somewhat simplified, thereby greatly decreasing the amount of time spent in this step. Even with the simplified construction algorithm, the performance loss related to the traversal step is quite acceptable. In fact, the inherent properties of BIH, combined with a well-optimized implementation, leads to optimal total rendering time.

These characteristics are what make BIH a great candidate for use with completely dynamic scenes [18].

In this work, BIH will be used as the main acceleration data structure. This is due to its simplified approach to construction combined with its great *time to image* performance results. Given the great importance of data structures in the implementation of an efficient ray tracer, further details about BIH are explained in Section 3.2.

C. HARDWARE ACCELERATION

Until the introduction of multi-core processors, increases in performance always meant requiring higher *clock* rates. This statement remained accurate until the processors approached the physical limits for *clock* rates, as well as the price feasibility for customers. The introduction of multi-core processors in the early 2000s changed this scenario advantageously. The developers can now use various programming techniques to leverage a higher computing performance.

Another possibility is to use hardware implementation for parts of a given application, usually the portion that requires more processing power. In other words, the core algorithm is implemented directly into the board, and therefore yields better performance. In fact, it is rather unlikely to see software implementations outperform hardware versions of the same algorithm. Even though this approach yields good results, having some portion of one's application written in the hardware is not always an advantage. For instance, flexibility is reduced when this approach is used. Customizing parameters (e.g. memory allocation, load balancing) for the algorithm become somewhat more difficult and/or limited.

GPU Architecture

In the Computer Graphics context, hardware implementation for rasterization exists in ordinary graphic cards. After rasterization became common, various efforts were developed to optimize its performance. This effort played an important role in the demand for a hardware implementation of said technique.

Graphics cards often have a dedicated GPU (Graphics Processing Unit) that is responsible for processing all operations related to displaying content on the screen. Given the mathematical nature of graphics and rendering, these units are quite specialized and therefore have outstanding performance in performing mathematical calculations. In addition, the performance for executing linear batches of operations is remarkable. However, their performance with branching is far from optimal. In other words, statements such as *if* and *loops*, as well as function call are very expensive for GPUs to perform. In addition to that, recursion is also a complication for GPUs as it is not only costly, but some architectures do not even support it. Developers often have to rely on memory intensive, less abstract and sometimes hard to understand iterative implementations.

GPUs often have lower clock rates when compared to standard CPUs. In fact, it is rather common to see models of the latter run at clock rates of 3GHz and higher, whereas relatively powerful models of the former only go up to 1GHz. One may then ask why it is advantageous to use GPUs for increasing performance of an application. This question is somewhat pertinent and yet its answer is quite simple. Ordinary CPUs meant for end consumers typically have 2-3 cores, and high performance models have 12 cores. On the other end, recent GPUs often have more than 400 and can go up to programmable 2880 cores [4]. This high number of cores indeed allows the programs to run in a massively parallel fashion, with the use of parallel programming techniques.

In order to take full advantage of the GPU hardware, programs have to use a special SDK (Software Development Kit), such as OpenCL [10] and NVIDIA CUDA [9]. Developers have to implement their algorithm (the one that is to be parallelized) in a *kernel* function that is passed directly to the GPU for execution. This *kernel* is then executed in all cores at the same time in a SPMD (Single Program Multiple Data) scheme. This means that the cores execute the same code, but with different PC (Program Counter), and different cores process different data, hence the SPMD acronym. Executing cores may very well finish at different times when compared to other cores. These SDK intelligently regroup finished cores, and reallocate them in blocks that are dispatched to process another chunk of data.

III. APPROACH

A. RAY TRACER

This section provides a detailed explanation about the implementation of our ray tracer. First, it shows how the ray tracer engine was designed from an architectural standpoint, and further details the relationship between the various entities in the system. Second, the intricacies of the implementation are exposed and explained, from the primitive set up to when the pixel colors are calculated and displayed in the screen. Finally, this section talks about OpenGL ES Shaders on mobile devices and how they were used to efficiently implement the actual rendering portion of the engine.

Architecture

In order to demonstrate the rendering engine in action, a mobile application was developed for the Apple iOS platform. The purpose of this approach is to encapsulate the functionalities of the rendering engine into an application that orchestrates the rendering process and also controls user interaction. This application was developed using the MVC (*Model-View-Controller*) architectural design. This design helps isolate responsibilities and keep the entities concise and reusable.

Figure 3.1 depicts the architecture of the application. Each MVC component is isolated in packages, and communicates with each other in two ways: direct (or active) association between entities (represented in solid lines), and indirect (or passive) association (represented in dashed lines). These types of associations differ in a sense that the former requires active communication between the entities (e.g. calling a function or method), but in the latter, one of the entities passively listens for changes in the other, and that change triggers an event.

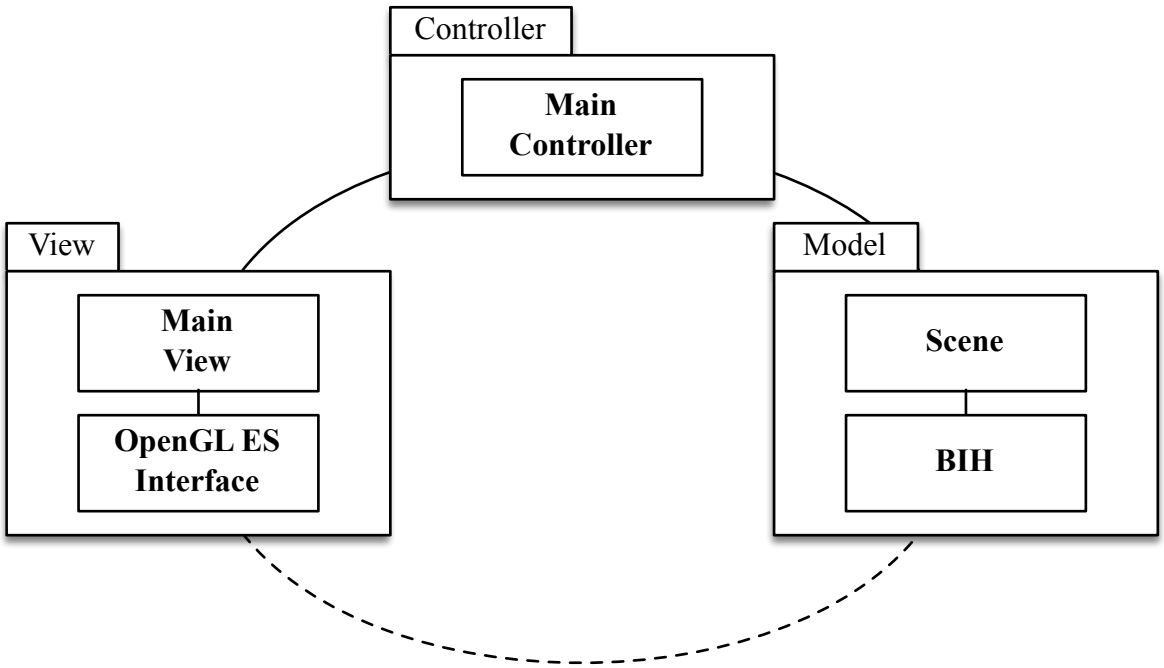


Figure 3.1: Architecture model of the ray tracer application.

In this context, each of the three packages has very distinct roles. First, the *Main View* is responsible for directly interacting with the user, and capturing the interaction events and communicating them to the *Main Controller*. In the context of rendering images, the View has a particularly important role: displaying content. This means that this component is responsible for communicating with the OpenGL ES Interface and drawing the rendered image on the screen. The *Main Controller* is responsible for receiving and handling user interaction events, but more importantly, it is also responsible for updating the Model according to these events. Finally, the *Model* is responsible for maintaining the application state throughout its lifetime. It stores and manages all the data related to the *Scene*, including camera, lights, primitives and materials, as well as the *BIH* and its configurations. To summarize, when a user slides his finger on the screen, that event is captured by *Main View* and forwarded to *Main Controller*, which processes the

event and updates the camera position in the Model. The *Main View* then receives an event saying that the *Model* has changed, which triggers a redraw operation.

Implementation

The actual implemented code of the application and the rendering engine was written in both Objective-C and C++ programming languages. Most applications written for Apple iOS platform are written completely in Objective-C. That is the easiest way to integrate with available frameworks for UI components, event handling and others, all of which were implemented using the same language. In this work, C++ was used to take advantage of some highly optimized implementations of commonly used data structures, and to provide support for advanced language features (e.g. *operator overloading*). More importantly, the BIH was implemented using this language in order to yield maximum performance.

Frameworks such as OpenGL ES were meant to be used with rasterization techniques for the majority of applications. It would be unfeasible to try to implement a ray tracer using the common rasterization approaches. This is because there is no simple way of calculating and tracing rays through the objects in the scene using only OpenGL ES. Normally, this framework has several requirements in order for the scene to be properly displayed on the screen. First, the developer has to set up a projection matrix, which will be responsible for transforming the primitives' vertices into projected coordinates. Second, the model view matrix has to be configured similarly to the projection one. This matrix is responsible for applying a constant transformation factor to all primitives' vertices. After the primitives are transformed by both matrices, the projected primitives will be ready for rasterization.

Using this principle, the implementation presented in this section overcomes the lack of support for ray tracing by using only two triangles as the OpenGL primitives. These triangles align

perfectly with the near plane of the *frustum*, and therefore fill the entirety of visible pixels in the projection plane. Figure 3.2 illustrates this approach, in which the two colored triangles (red and blue) are processed normally by OpenGL but the color for each pixel is calculated slightly differently.

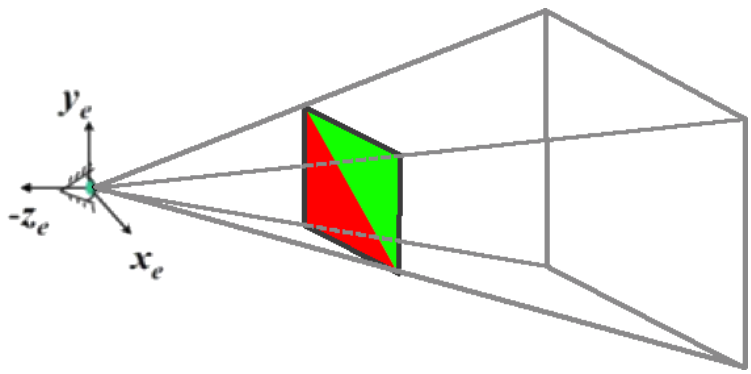


Figure 3.2: Visual representation of the two triangles located at the projection plane.

After both triangles were transformed and its coordinates are in the projection plane, now it is time for rasterizing them. However, instead of calculating the color of each pixel based on interpolating over the vertices' colors, or even using texture mapping, OpenGL ES allows one to use shaders for that purpose. By doing this, the developer can override the color calculation portion of OpenGL ES rasterization, and provide the renderer with a custom implementation of a rasterizer. In other words, OpenGL shifts to the developer the responsibility of calculating the final color for each pixel. The developer needs to write a special function that runs on the GPU and is called for every pixel in the screen. From inside this function, we have access to the coordinate of the pixel in the projection plane, as well as the position of the camera and other parameters needed for the ray tracer. Finally, after performing the ray tracing, this function must return the final color of the pixel being processed.

When the application is started, some important steps are executed before the rendering process starts. First, the OpenGL context is prepared for rendering, and the shaders are

instantiated and compiled. Second, it allocates and positions the primitives in the scene. Third, it creates the BIH and immediately triggers the construction process. Finally, it sets up a render loop that will remain active throughout the application lifecycle.

After this setup procedure is completed, the render loop is instantiated and scheduled to run 60 times per second. This loop can be divided into three main steps, as shown in Figure 3.3. First, the *Application* is responsible for clearing the screen buffer and setting up OpenGL parameters. After that, the *Application* processes the UI events and updates the Scene accordingly. In the second step, the *Scene* processes the updates from the *Application* and proceeds to reconstruct the BIH. This reconstruction process must make sure that the BIH will be able to represent the modified scene without drastic changes. More specifically, the arrays used to store the nodes and sorted objects need to have enough space to accommodate the updated tree. Finally, the *Renderer* uploads the BIH data to the GPU using the OpenGL ES Interface and triggers the draw operation that will ultimately execute the ray tracing shader code.

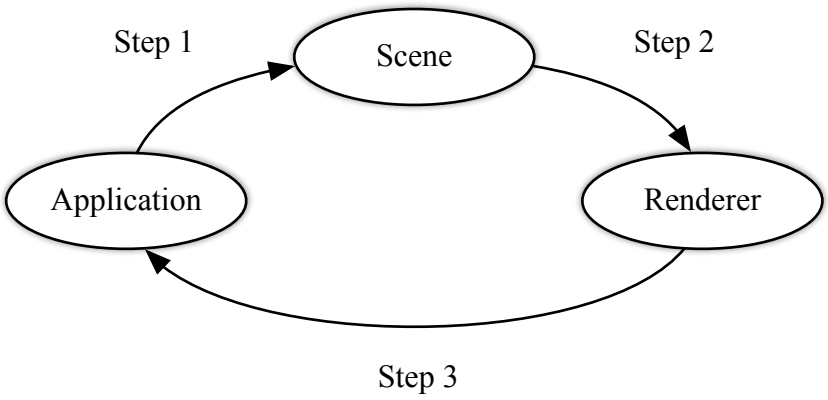


Figure 3.3: Flowchart representation of the main rendering loop.

Shaders

There are multiple versions of both OpenGL and GLSL (OpenGL Shading Language) [6] that are compatible with Apple iOS platform. However, implementing this work requires some

advanced features, such as bitwise operations and masking, which are only available in OpenGL 3.0 and GLSL 300. Since the BIH node structure is very compact and uses bit masking to store data in the higher order bits, being able to interpret this data inside of the shader is of crucial importance. In addition to that, GLSL 300 is the first version that provides full support for 32-bit full precision integer operations, which are used thoroughly in the BIH construction and traversal process.

This work takes advantage of the GPU present in mobile devices to perform high quality rendering using shaders. The individual shader calls for calculating pixel colors are added to the graphics pipeline present in the hardware. In this way, the GPU is able to efficiently reschedule worker threads and process all the pixels with higher performance. In order to take advantage of that, the entire ray traversal process was implemented inside of the shaders. In other words, the function call that OpenGL makes for calculating the pixel color runs the ray traversal algorithm and calculates the result based on the outcome of it.

This shader function is detailed in Listing 3.1. As it was mentioned earlier, the GPU calls a function in the implemented shader and this function has a very specific name and signature. Lines 1-11 represent the *main* function that is responsible for setting up and converting projected coordinates and starting the ray traversal process. Lines 12-37 represent the recursive function that performs the ray traversal and shading. It is important to mention that this code is highly abstracted and lacks specific implementation details. This was done on purpose in order to facilitate the understanding of the technique.

In the *main* function, Lines 2-3 are responsible for loading and transforming the projected coordinates. First, the shader loads the coordinates from a global variable named *gl_FragCoord*. After that, the shader applies the aspect ratio of the screen into the coordinates. In Lines 5-7, the

shader loads the information about the camera position that has been uploaded from the Application code into the GPU. Finally in Line 9, the *trace_ray* recursive function is called and the shader returns the result of it.

```
 1: vec3 main () {
 2:   vec2 coordinates = gl_FragCoord.xy;
 3:   vec2 uv = apply_aspect_ratio( coordinates );
 4:
 5:   Ray ray;
 6:   ray.origin = ...; // Load camera position
 7:   ray.direction = normalize( uv.x, uv.y, -1.0 );
 8:
 9:   return trace_ray( ray, 1 ); // Seed initial depth
10: }
11:
12: vec3 trace_ray ( Ray ray, int depth ) {
13:   Intersection hit = intersect_scene( ray );
14:
15:   if ( depth == MAX_RECURSION_DEPTH ) {
16:     return perform_shading( hit, ray );
17:   }
18:
19:   vec3 color = vec3 ( 0.0, 0.0, 0.0 );
20:
21:   if ( hit.type != GeometryTypeNone ) {
22:     color = perform_shading( hit, ray );
23:     Material material = hit.material;
24:
25:     if ( material.kr > 0.0 ) { // Object is reflective
26:       Ray reflected = calculate_reflected_ray( hit, ray );
27:       color += material.kr * trace_ray( reflected, depth + 1 );
28:     }
29:
30:     if ( material.kt > 0.0 ) { // Object is transmissive
31:       Ray transmitted = calculate_transmitted_ray( hit, ray );
32:       color += material.kt * trace_ray( transmitted, depth + 1 );
33:     }
34:   }
35:
36:   return color;
37: }
```

Listing 3.1: Shader functions implementation in GLSL.

In *trace_ray*, Line 13 executes the ray traversal through the BIH tree structure. A struct of type *Intersection* is returned, and it contains information about which primitive was hit, what type of primitive it is, and other information. Lines 15-17 represent the stopping condition for the recursive algorithm. Namely, if the recursion depth reached the value of *MAX_RECURSION_DEPTH*, no secondary rays are casted for the current intersection, and thus only the color of the currently intersected primitive is returned. Line 21-23 checks whether the intersection was valid of not and performs the shading in case a primitive was hit by the current ray. If the ray did intersect the primitive that has reflective and/or transmissive properties, the reflected and/or transmitted rays are calculated and a recursive call is performed each of the rays, as seen in Lines 25-33. After all recursive calls have returned, and the final color has been calculated, the result is returned to the *main* function, and the final color is attributed to the current pixel.

User Interaction

In order to demonstrate the interactivity and efficiency of our ray tracer, some extra features were implemented. These features relate to how the user interacts with the application and how he is able to modify the scene being rendered. First, the user is able to move the camera along the *xy* plane by touching and moving one finger on the screen. The second gesture uses two fingers and enables the user to move the camera along the *z* axis. By placing two fingers on the screen and spreading them apart, the camera will move forward. If the user moves both fingers in a pinch gesture, the camera will move backwards.

B. BOUNDING INTERVAL HIERARCHY (BIH)

Keeping in mind what was explained in Section 2.2.3, this section explains in more details the nuances of the BIH that was implemented in this work. More advanced concepts are also

explained, such as how the cutting planes are selected, how are the tree nodes structured and how the primitives are sorted. The construction and traversal algorithms are thoroughly detailed, including step-by-step explanation of the code related to the most important parts of each algorithm. Finally, the BIH is compared to other common data structures in order to evaluate the advantages and disadvantages of this data structure.

Cutting planes and empty spaces

In order to compensate for the lack of heuristics in the construction of this data structure, the BIH implements intervals in the structure of each internal node. These intervals represent the volume within the node in which primitive objects exist. Initially, the tentative cutting plane is always positioned half way through the widest dimension of the AABB that involves the node. The distribution of the primitive objects across each side of the cutting plane is given by the position of their centroids.

The BIH implements the binary search in the node structure based on said intervals, since each primitive is associated with only one side of the cutting plane. In case a primitive object intersects the tentative cutting plane, and its centroid is indeed in the same side of it, the tentative cutting plane is adjusted accordingly in case it intersects with a primitive object whose centroid is in the same side of said cutting plane. The reason for this adjustment is to include the intersected primitive object completely (Figure 3.4), based on the AABB that encloses the primitive.

One of the main characteristics of BIH is the way it handles empty spaces. In the construction process, the tentative cutting planes are defined and empty nodes are easily found. The use of two cutting planes, rather than just one, opens the possibility of having empty spaces between the left and right child of the same node (Figure 3.4d and Figure 3.5c). This

characteristic is particularly useful and improves performance when it comes to rendering scene whose primitive objects are heterogeneously distributed in the space.

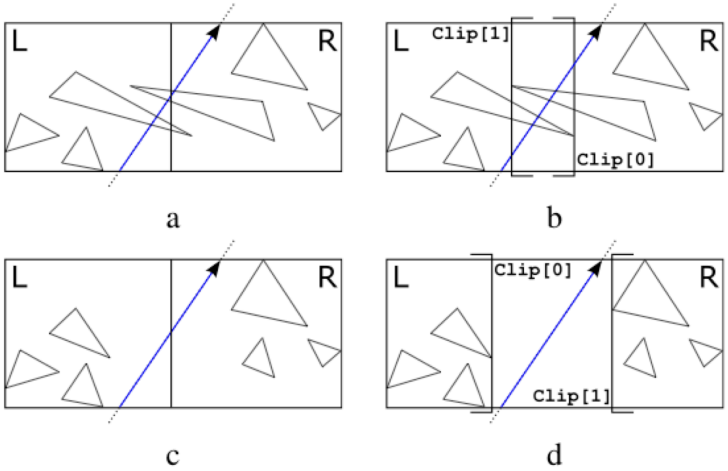


Figure 3.5: Representation of the tentative and final cutting planes. In (a), the planes were obtained by dividing the AABB in its widest dimension [18]. In (b), the empty nodes were not considered. Finally in (c), the cutting planes were adjusted to enclose the primitive objects.

Node structure

The internal nodes of the BIH consist of two cutting planes and a pointer to the right child. This pointer also stores information about the cutting planes. The fields in the node structure are intelligently reused to improve memory utilization.

In the construction process, the right child is always instantiated beside the left child in the node array. Therefore, there is no need to store the pointer to the right child. All nodes are aligned in structures with 12 *bytes*, according to Listing 3.2.

```
1: typedef struct {
2:    unsigned int axis_leftChild;   // 4 bytes
3:    unsigned int clipL_primOffset; // 4 bytes
4:    unsigned int clipR_primCount;  // 4 bytes
5: } BIH_Node;
```

Listing 3.2: BIH node structure

Line 2 declares the pointer to the left child and also defines the axis of the cutting planes if it is an internal node. The axis information is hidden in the 2 higher order bits of *axis_leftChild* in such a way that 00, 01 or 10 represent the cutting direction in either *x*, *y* or *z* respectively. If the value 11 is stored in said bits, the current node is a leaf. Line 3 and 4 declares the left and right cutting planes respectively, if it is an internal node. If it is a leaf, *clipL_primOffset* and *clipR_primCount* represent the offset in the array of primitives and the number of primitives in the current node.

Primitive sorting

The algorithm executed in the construction of BIH is in fact a sorting algorithm whose structure is identical to the one of a *quicksort*[18] and runs in O(n log$n$). This sorting algorithm affects the array of primitives while the BIH is constructed. The main objective of using this sorting algorithm while constructing the BIH is to make sure that the primitives are ordered both in space and in the array. In other words, given a node and a level in the tree, all primitives that are located to the left (in the space) of the pivot in the array of primitives are at indices smaller than the pivot's index.

This comparison of coordinates of the primitives always considers the chosen axis for the cutting planes. As mentioned before, this chosen axis is always the widest dimension of the AABB of the node. The value considered in this comparison is the center of the AABB that encloses the primitive being processed.

This approach of using the AABBs instead of the primitives' coordinates improves the performance of the construction in about 2 or 3 times [18]. Even though this approach may lead to less than optimal trees, the loss in performance for the traversal is almost inexistent.

Numeric precision

Space partitioning techniques require mathematical operations of intersection between the primitives and the cutting planes, as well as with the AABBs. It is well known that this type of operation opens possibilities for precision issues when using floating point numbers. In order for the algorithms implemented by the BIH to be executed in a stable manner, extra care needs to be taken by the programmer that leads to loss in performance [18].

The BIH construction algorithm uses only the AABBs coordinates over the canonical axes, and maximum and minimum operations. This approach makes the entire process more robust and stable, and it does not lose performance with extra multiplication and division calculations over floating point numbers.

Construction

This is the most important step in the implementation of BIH. Among the optimizations and new approaches proposed by this data structure, this is the one that contributes the most for the high rendering performance of the ray tracing. Implementation details of the algorithm, as well as the reason for such great relative performance, are further explained in this section.

Initially, minimum and maximum operations are used against the primitive coordinates in order to calculate the smallest AABB that contains it. Simultaneously, the AABB for the entire BIH is also calculated. This AABB is responsible for enclosing all primitives in the scene. It is also used in the early stage of the traversal in such a way that rays that do not intersect said AABB are prematurely discarded and never begin to visit the tree nodes. The outcome of this step will be posteriorly used in the sorting of the primitive array, as well as in the subdivision step during the optimization of the cutting planes.

In the beginning of the subdivision step, the cutting planes are calculated as being the mean of the minimum and maximum values of the AABB in the widest dimension. The left and right cutting planes, which will now be referred to as clipL and clipR in this work, are initialized with the minimum and maximum value respectively of the cutting axis. Each primitive in the current node is then iterated in the following manner:

- The center of the AABB that encloses the primitive is calculated and checked whether it is in the left or right side of the cutting plane.
- If the center of the AABB is in the left side and the maximum value of the AABB of the primitive is higher than clipL, the latter is updated with the value of the former.
- If the center of the AABB is in the right side, the current primitive is swapped with last primitive in the queue and this process is repeated. Similarly, the value of clipR is updated with the minimum value of the AABB of the primitive, in case it is lower than clipR.
- Simultaneously, the minimum AABB that encloses all primitives in this node is calculated, which will be referenced to as nodeNewAABB.

Following that process, the algorithm performs a check that is implemented by BVH. This additional step works as an optimization over the approach originally proposed by [18]. The BIH only specifies the cutting planes starting at the center of the AABB of the current node. This is not good from the performance standpoint because all primitives in said node may be concentrated in one side of the cutting plane, thereby configuring most of the node's volume as empty space. The objective of this optimization is to use the value of nodeNewAABB and verify the relation between the volume occupied by the primitives and the total volume of the node over the cutting axis. If the size of the nodeNewAABB, multiplied by a factor of 1.3 (defined

empirically), is less than the size of the AABB, the current node will be iterated again considering the value of nodeNewAABB. This fixes the problem of choosing the cutting planes in a less than optimal way, which would lead to performance loss in the traversal step. In this work, nodes that qualify for this optimization are called *Optimized Nodes*.

If all primitives are located to the left and the median for the cutting axis is the same as the previous nodes' subdivision, the recursion is finalized by creating a leaf node containing the primitives in the current node. Since all primitives are located in the left side the AABB of the current node is reduced in half, having its maximum bound to the value of the median of the cutting axis. The current node is then processed again in order to find better cutting planes. Similar process is performed in case all primitives are located in the right side of the median.

If primitives exist in both sides, meaning that the algorithm is being successful in dividing the objects, the left and right nodes are allocated. After that, the current node is updated with the reference to the child node and with the information about the cutting axis and planes, followed by the recursion over the left and right nodes.

As mentioned earlier, this algorithm works very similarly to a *quicksort*. The stop condition for the recursive construction of the tree is somewhat similar to the one in the *quicksort* algorithm, which is when the interval (over the primitive array) used in the current iteration is less than or equal to one. In the case of BIH, this is handled a little bit differently, because leaf nodes may contain more than one primitive, as explained earlier. In other words, the stop condition for the *quicksort* in the BIH happens when the number of primitives in the node is less than or equal to the maximum number of primitives per node, or when the current depth is equal to the maximum permitted.

Implementation Details

In the previous section, details were explained about how the construction step works. These details include how to best choose the cutting planes, how to deal with special cases and some optimizations. The in-depth construction process explanation, from the initialization step to the end of the recursion with the stopping conditions explained earlier, is divided in two parts: setup and recursion.

The first part of the construction is responsible for preparing and configuring the initial state of the BIH. Listing 3.3 shows the individual steps needed to accomplish this task. Lines 2-4 initialize the main variables that will be used throughout the construction process. The variables *nodes*, *primitive_aabbs* and *global_aabb* will hold the values for all nodes in the tree, the AABBs of each primitive and the global AABB for the BIH, respectively.

```
 1: void bih_setup( BIH* bih ) {
 2:   bih->nodes = new BIH_Node[];
 3:   bih->primitive_aabbs = new AABB[];
 4:   bih->global_aabb = INVALID;5
 5:
 6:   for (primitive[i] in all_primitives) {
 7:     AABB primitive_aabb = calculate_aabb( primitive[i] );
 8:     bih->primitive_aabbs[i] = primitive_aabb;
 9:     bih->global_aabb = aabb_union( bih->global_aabb,
          primitive_aabb );
10:   }
11:
12:   bih_subdivide( bih, 0, all_primitives.count - 1,
          bih->global_aabb );
13: }
```

Listing 3.3: Algorithm that represents the BIH setup.

Line 4 initializes the global AABB as invalid, meaning that its minimum coordinates are set to +*infinity*, whereas its maximum coordinates are set to -*infinity*. In this way, the global AABB adjusts itself when processing the first primitive, and valid coordinates will lie between -*infinity*

and +*infinity*. Lines 6-10 loop through all primitives calculating their individual AABBs, as well as computing the global AABB as the union of all individual AABBs. Finally, line 12 contains the recursive call for the BIH construction.

The second part is the recursion, which represents the core of the construction algorithm. Listing 3.4 contains the logic necessary to process a group of nodes and recurse, to deal with special cases and to implement optimizations over the original algorithm.

Lines 2-5 determine the two stopping conditions for the recursive algorithm. First, the number of primitives in the current node has to be less than or equal to *MAX_PRIMS*. Second, the depth of the current subtree cannot exceed *MAX_TREE_DEPTH*. If either condition is not met, the algorithm creates a leaf node and finishes the recursion.

Lines 14-25 are responsible for sorting the primitives in each side of *split*. If the primitive is on the right, it is swapped with the last primitive in the interval and the test is repeated. This process is similar to what happens in a standard *quicksort* algorithm. In the end of each iteration, the variables *nodeL* and *nodeR* are adjusted to enclose all primitives in each side.

Lines 27-30 implements an optimization based on the BVH over the original algorithm proposed by [18]. The algorithm calculates the relation between the volume occupied by primitives and the total volume of the node and reiterates the current node in case the primitives are not homogeneously distributed.

Lines 32-38 deal with two special cases: when all primitives in the node are either in the left or in the side of the splitting plane. If either case is true, said plane is recalculated and the node is iterated again. Namely, the *min* or *max* of the AABB of the current node is updated with the value of *split*. After the AABB is updated, the algorithm is now more likely to find a more

appropriate cutting plane. If this check were not executed, the probability of having empty nodes, and therefore unnecessary node traversal, would be considerably higher.

```
 1: void bih_subdivide( BIH* bih, int left, int right, AABB nodeBox ) {
 2:   if ( right - left + 1 <= MAX_PRIMS || current_depth == MAX_TREE_DEPTH ) {
 3:     create_leaf_node( bih, left, right );
 4:     return;
 5:   }
 6:
 7:   float right_original = right;
 8:   clipL = +INFINITY;
 9:   clipR = -INFINITY;
10:   float nodeL = +INFINITY;
11:   float nodeR = -INFINITY;
12:   float split = calculate_median_of_largest_axis( nodeBox );
13:
14:   for (primitive[i] in all_primitives in range [left, right]) {
15:     if (bih->primitive_aabbs[i].center < split) {
16:       clipL = min( clipL, bih->primitive_aabbs[i].max );
17:     } else {
18:       swap(primitive[i], primitive[right]);
19:       right = right - 1;
20:       clipR = max( clipR, bih->primitive_aabbs[i].min );
21:     }
22:
23:     nodeL = min( nodeL, bih->primitive_aabbs[i].min );
24:     nodeR = max( nodeR, bih->primitive_aabbs[i].max );
25:   }
26:
27:   if ( (nodeR - nodeL) * 1.3 < calculate_largest_axis( nodeBox ) ) {
28:     update_aabb_bounds( nodeBox, nodeL, nodeR );
29:     // Iterate current node again
30:   }
31:
32:   if ( all_primitives_to_the_left_of( split ) ) {
33:     nodeBox.max = split;
34:     // Iterate current node again
35:   } else if ( all_pritimitives_to_the_right_of( split ) ) {
36:     nodeBox.min = split;
37:     // Iterate current node again
38:   } else {
39:     create_left_and_right_nodes( bih, left, right );
40:     AABB aabb_left = update_box_with_clipping_plane( nodeBox, clipL );
41:     AABB aabb_right = update_box_with_clipping_plane( nodeBox, clipR );
42:
43:     // Invoke recursive calls
44:     bih_subdivide( left, right, aabb_left );
45:     bih_subdivide( right + 1, rightOriginal, aabb_right );
46:   }
47: }
```

Listing 3.4: Algorithm that represents the BIH recursion.

Lines 39-45 deal with the standard case: when there are enough primitives in both left and right sides. If that is the case, the current AABB is divided in half based on the value of *split*. The algorithm then calculates the values of *left* and *right* for the recursive calls that are fired for the left and right nodes. The process is then repeated as if each of the child nodes was the root of a new BIH tree.

Traversal

A critical characteristic of the BIH traversal process that affects performance positively is the fact that it is possible to tell the primitive that is closer to the camera, depending on the ray direction. This happens because the construction works as a spatial sorting algorithm. This characteristic makes the traversal of BIH almost identical to the one in the KD-Tree.

Given BIH's approach to using cutting planes that define valid intervals within the data structure, it is possible that a ray can traverse it without visiting any leaf node. This situation happens when the ray goes through the space between the left and right cutting planes, which contains no primitive objects. Since the possibility of intersecting no primitives exist, the ray goes out of the structure without visiting any other node. This approach is good from the performance standpoint because the traversal algorithm inherently ignores empty spaces.

As mentioned before, the volumes that represent the nodes in the tree may overlap, thereby implying that the nodes need to be stacked for further processing until an intersection is found. As soon as one is found, the branches of the BIH that represent primitives that are farther from the camera than the current intersection can be pruned in order to avoid unnecessary node visits.

The traversal process begins after the ray for the current *pixel* is calculated and sent to the BIH. Among the parameters passed to the BIH traversal are *t_out* and *prim_index*. The former represents the parametric distance between the ray origin and the intersection point inside of the

structure. The latter identifies the primitive that was intersect by pointing to its index in the primitive array. In case no intersection is found, *t_out* is set to *infinity*. These parameters are further used to calculate the final color of the *pixel*. This process is commonly known as *shading*.

Starting the traversal process, the ray is tested against the AABB that encloses the entire BIH structure. The outcome of this test is two floating point numbers that represent valid interval for values of *t_out*: *t_min* and *t_max*. These values are the parametric distances of the intersection points in the AABB's surface with regards to the ray origin and direction, as seen in Figure 3.6.

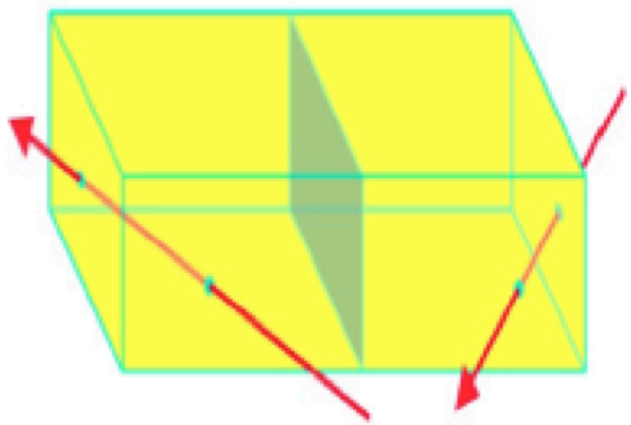


Figure 3.6: Visual representation of the intersection test of the ray with an AABB. Red arrows represent individual rays that hit the AABB in two different points.

If the ray was originated inside of the AABB, which means that the camera was located inside of the BIH, the *t_min* receives a negative value. In case this happens, *t_min* receives the value 0.0 instead. After that, the parametric distance of the ray intersection with both cutting planes of the current node is calculated. If these values lie outside of [*t_min*, *t_max*], it means that the ray traversed the empty space between the nodes, in which no primitives exist, thereby finalizing the traversal with not valid intersection.

If the ray intersects only one of the child nodes, and it is not a leaf node, the traversal algorithm immediately starts to process said node, as there is no need to stack the other node.

However, if both nodes are intersected, the node that is farthest from the ray origin is stacked and the algorithm starts processing the node closer to the camera.

The last two steps that are part of the node traversal are repeated until a leaf node is found. Once this condition is satisfied, said leaf node is processed. For each primitive present in the leaf node, the values for *t_out* and *prim_index* are calculated and stored. The final value for these parameters will be the ones for the primitive whose result for *t_out* is the smallest, which means that it is the primitive closest to the camera. After calculating the results for the current node, the next node that is on the top of the stack is unstacked and the process is repeated.

Implementation Details

In the previous section, the ray traversal process was explained, including the stopping conditions, node stacking procedures and primitive intersection. Listing 3.5 shows the details of the traversal algorithm, from parameters configuration and allocation, to calculating the final intersection factor and returning the primitive that was intersected.

Lines 2-11 execute the initial set up before the traversal. First, all variables are initialized to a state in which no intersections were found. Second, the ray is tested against the global AABB for the BIH tree. This operation returns a *boolean* flag that represents if the AABB was intersected, and also returns two floating pointing values: *t_min* and *t_max*. These values represent the valid range for the scalar parameter *t_out*. If no intersection was found, the algorithm stops promptly and returns the value for *t_out* and *prim_index*. Finally, if an intersection was otherwise found, the parameter *t_min* is adjusted according to whether or not the ray originated from inside the box (i.e. *t_min* < 0.0). In addition, the stack is created and the root node added to it.

```
 1: void ray_traversal( in Ray r, out float t_out, out uint prim_index ) {
 2:   t_out = INFINITY;
 3:   prim_index = -1;
 4:   float t_min = -INFINITY, t_max = +INFINITY;
 5:
 6:   if ( ! intersect_aabb( ray, bih->global_aabb, out t_min, out t_max) )
 7:     return;
 8:   t_min = MAX( 0.0, t_min );
 9:
10:   BIH_Node[] stack = new BIH_Node[MAX_BIH_DEPTH];
11:   stack.push( bih->root );
12:
13:   do {
14:     BIH_Node current = stack.pop();
15:
16:     while ( ! current.is_leaf() ) {
17:       float t_clipL = intersect_plane( ray, current.clipL_primOffset );
18:       float t_clipR = intersect_plane( ray, current.clipR_primCount );
19:
20:       if ( t_clipL < t_min && t_clipR > t_max ) {
21:         break;
22:       } else if ( t_clipL < t_min ) {
23:         current = current.left_node();
24:         continue;
25:       } else if ( t_clipR > t_max ) {
26:         current = current.right_node();
27:         continue;
28:       } else {
29:         stack.push( current.right_node() );
30:         current = current.left_node();
31:       }
32:     }
33:
34:     if ( current.is_leaf() ) {
35:       int offset = current.clipL_primOffset;
36:       int count = current.clipR_primCount;
37:
38:       for ( int i = offset; i < offset + count; i++ ) {
39:         float t_prim = INFINITY;
40:         if ( intersects_primitive( ray, t_prim, all_primitives[i] ) &&
                 t_prim < t_out && t_prim <= t_max ) {
41:           t_out = t_prim;
42:           prim_index = i;
43:         }
44:       }
45:     }
46:   } while ( ! stack.empty() );
47: }
```

Listing 3.5: BIH Ray Traversal

The main loop then starts by unstacking the top item and processing it. Lines 16-32 are responsible for navigating the tree structure searching for a leaf node. If the current node is a leaf, the *while* loop stops and the leaf is processed. Otherwise, the internal node is processed in three steps. First, the scalar parameters for both clipping planes are calculated and stored in *t_clipL* and *t_clipR*. Second, in order to tell how the ray intersects the node, these values are compared with the values of *t_min* and *t_max* based in two conditions: 1. *t_clipL* < *t_min*; and 2. *t_clipR* > *t_max*.

If conditions 1 and 2 are both met, it means the ray traverses the void space between the nodes and therefore the current loop should stop. If only condition 1 is met, the ray only intersects with the right node, and therefore the algorithm should promptly jump to said node and process it. Similarly, if only condition 2 is met, the algorithm jumps and processes the left node. Finally, if neither condition is met (i.e. both nodes were intersected by the ray), the right node is pushed into the stack and the algorithm jumps to the left node and processes it.

Once a leaf node is found, the algorithm tests for primitive intersection in Lines 34-45. As mentioned in Section 3.2.2, the values for *clipL_primOffset* and *clipR_primCount* in the current node respectively represent the offset and count in the array of primitives. All primitives in this range are then tested against intersection with the given ray. If an intersection is found and it is closer to the ray origin than the current value of *t_out*, the value of this output parameter (along with *prim_index*) is updated to represent the current intersection. This process is repeated until the closest intersection is found for the given leaf node.

The main loop repeats until the stack is empty, at which point the algorithm should return the output parameters. It is important to notice that this stopping condition not necessarily means that the algorithm found a valid intersection. In other words, intersecting with the global AABB

does not mean that it intersected an actual primitive. This is perfectly reasonable, as there is plenty of void space inside of the volume of the BIH tree.

Comparison with Other Data Structures

This section provides an analysis of advantages and disadvantages of BIH in comparison to BVH and KD-Tree. This analysis considers four main aspects that are related to the use of such data structures in a ray tracing system.

BIH *vs.* BVH

- *Node structure*: In this aspect, BIH has a big advantage due to its simpler approach to structuring the nodes within the tree. Contrarily to BVH, the BIH does not store an AABB for each node, which means a savings of 24 *bytes* per node (x, y and z coordinates for the maxi- mum and minimum points of the AABB). In addition to that, the BIH is able to intelligently reuse the fields in the structure depending on whether it is an internal or leaf node.
- *Memory utilization*: Given the requirement of storing one AABB per node in the BVH, this structure requires a large amount of memory in order to store the entire structure. The BIH, on the other hand, has an advantage in this aspect, given its node structure simplicity.
- *Construction*: The algorithm responsible for constructing the BIH is in fact a *quicksort*. As the algorithm builds the structure, it sorts the primitives in the space according to the cutting planes. This approach makes things slightly more complex from an algorithm standpoint. On the other hand, the exhaustive use of AABBs during the construction of the BVH allows for a more optimized data structure that maximizes the empty spaces.

- *Traversal*: This process is more efficient in the BIH due to the primitive sorting. This happens because the primitives are sorted and the algorithm can easily tell which primitive is closer to the camera.

BIH *vs.* KD-Tree

- *Node Structure*: In this aspect, the KD-Tree is highly advantageous. A well-optimized implementation of a KD-Tree only needs 8 *bytes* per node, whereas in the BIH at least 12 *bytes* are required. On the other hand, the BIH has the advantage of specifying two cutting planes per node, allowing for empty spaces between sibling nodes, which do not happen in the KD-Tree.
- *Memory utilization*: The BIH has a big advantage in this aspect. First of all, the KD-Tree needs to deal with the cases in which the cutting plane intersects a primitive, requiring the minimum AABBs to be recalculated and stored for each side. In addition to that, primitives that are intersected by cutting planes need to be referenced by both left and right child nodes. Another bad aspect of the KD-Tree is that empty nodes always need to be allocated in this structure, increasing the memory usage.
- *Construction*: This is clearly the best aspect of BIH when compared to the KD-Tree. There is no need to recalculate AABBs when a cutting plane intersects a primitive. The fact that the BIH does not use heuristics when creating the data structure makes the entire process much faster.
- *Traversal*: The traversal step is almost identical in both structure, but the BIH has a slight advantage when it comes to empty nodes. As mentioned before, the KD-Tree requires that empty nodes are stored, and therefore visited during the traversal, which does not happen in the BIH. On the other hand, the use of SAH in the KD-Tree construction lead

to highly optimized trees with quite small traversal times, thereby resulting in a better performance when compared to the BIH.

IV. RESULTS

In this section we describe the platform we used to evaluate our ray tracing algorithm and the experiments we ran to determine how our software performed in different configurations.

A. EVALUATION PLATFORM

Our software was implemented and evaluated using an iPhone 5s Model A1453 with 1GB of RAM (Random Access Memory) and 32GB of storage. The processor in this device is the A7 chip, and it works as both CPU and GPU. It is the first processor for mobile devices to support 64-bit architecture, and it has presented huge graphics performance improvement when compared to the processor in the iPhone 5. This processor features a dual-core CPU that runs at 1.3GHz, and a quad-core GPU that is able to deliver 76.8 GFLOPS (billions of floating point operations per second).

The iPhone 5s features a 4 inch LED-backlit display with maximum resolution of 640x1136 pixels. The pixel density in this display is approximately 326 *ppi* (pixels per inch). For the sake of performance, the actual resolution used in the results was 640x1088. The reason for using this configuration is because OpenGL ES has less than optimal performance when dealing with screen resolutions whose width and height components are not multiples of 32. In order to avoid hurting the performance, the height of the OpenGL ES viewport was set to 1088.

B. GRAPHICAL MODELS

For the sake of simplicity, this work only considers spheres as a valid type of primitive. This is because they are simple to implement and yet are powerful enough to illustrate the high quality image artifacts that stem from rendering reflected and transmitted rays. In addition to that, it performs outstandingly well when demonstrating how ray tracing deals with smooth surfaces. In

fact, spheres can do this without a huge impact in the memory footprint, as opposed to using triangles, which requires the surface to be represented by thousands of small primitives.

In this work, three different scenarios were considered for capturing the results. In all scenarios, all primitives lie in a cube whose center is located at the origin (0.0, 0.0, 0.0) and whose coordinate values are in the [-20.0, 20.0] range. In order to improve visualization of reflection and refraction effects, a plane exists in all scenarios and is located below the cube that encloses all primitives.

This plane, for all scenarios, is centered at (0.0, -20.0, 0.0) and its normal vector is (0.0, 1.0, 0.0). Some portions of this chapter refer to and compare with the brute force approach. In the ray tracing context, this approach means that neither acceleration data structures nor other techniques are used as a form of optimization. Furthermore, every ray traced in the scene will be tested for intersection with every primitive present in the scene, regardless of its position in the space. In this way, a color for a given pixel can only be determined after all primitives were tested and the closest intersection was found. For the sake of simplicity, only two materials types were considered in this section.

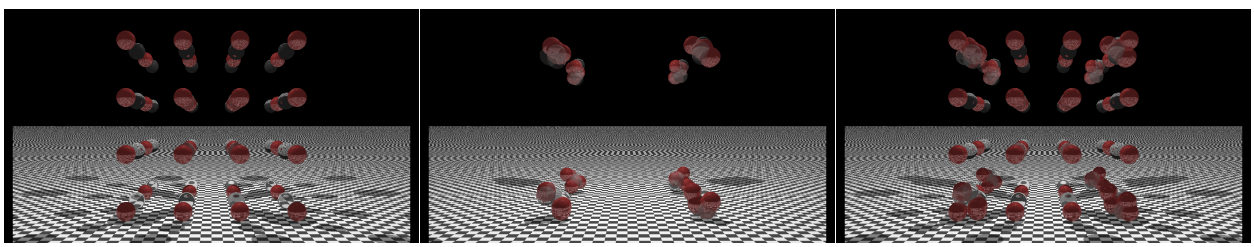
Table 4.1 shows the details about each material. The first material has a predominant red color and 30% of reflectiveness. The second material on the other hand is predominantly transparent, which means that most of its color comes from transmitted rays that hit other objects. In other words, the 70% refractiveness in the second material easily overcomes the other 30% of white color that is spread across the ambient, diffuse and specular components.

Figure 4.1 depicts each of the 3 scenarios. In Scenario 1, all primitives are distributed with equal spacing across the width, height and depth of the cube. In Scenario 2, the primitives are divided in 8 smaller cubes and each is located at one corner of the cube. Each group of primitives

is then positioned randomly in said smaller cube. Finally in Scenario 3, the primitives are distributed across the space using both previous approaches. Half of the primitives are clustered in the corners, and the other half is evenly distributed across the volume of the cube.

		Red	Transparent
Color (*rgb*)		(0.9, 0.3, 0.3)	(1.0, 1.0, 1.0)
Ambient		0.30	0.10
Diffuse		0.30	0.10
Reflective		0.30	0.00
Specular	Component	0.10	0.10
	Power	40.00	50.00
Transmissive	Component	0.00	0.70
	Refraction Index	0.00	0.95

Table 4.1: Constant values for materials used in the shading portion of the rendering.



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3

Figure 4.1: Scenarios used for capturing results.

The position of the camera plays an important role in the capture of the results. In fact, it is crucial that the camera is positioned in such a way that all primitives are visible in the screen, and yet they are as close as possible to the observer's eye. Since the cube that encloses the primitives is centered at the origin, the camera needed to be translated a certain distance away from the origin in order to visualize all primitives. For this reason, the camera is centered at (0.0, 0.0, 100) in all scenarios.

The same requirement does not exist when it comes to positioning the light correctly. For the sake of simplicity, this work considers only one light source that is located at (0.0, 40, 0.0). It is a point light source whose color is 100% white.

C. VISUAL QUALITY

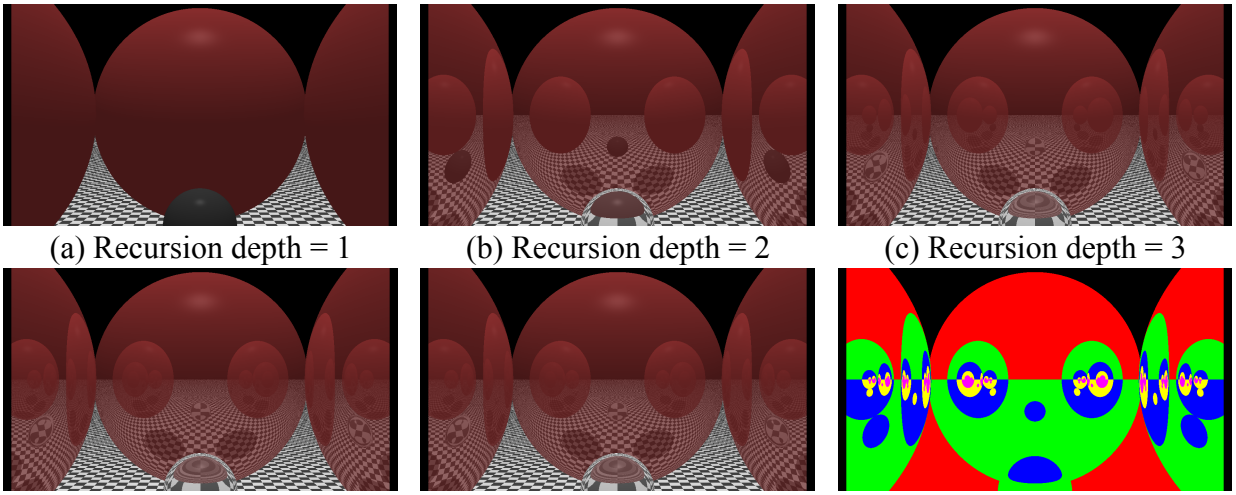
Taking into account the concepts presented in Section 2.1.1, this section illustrates the impact in the visual quality that stems from the use of ray tracing. It was mentioned earlier in this work that the implementation of reflection and refraction alone makes the final result somewhat more realistic when compared to rasterization.

In nature, the light rays bounce indefinitely until they are completely absorbed by the object surfaces. Not surprisingly, the number of ray bounces ray tracer has to be limited for processing time feasibility reasons. In this work, the maximum number of ray bounces is referred to as recursion depth.

The particular scenario used in this section is comprised of 4 primitives only. The idea here is to show the visual impact of choosing the recursion depth in the ray tracer. Figure 4.2 illustrates the result of using different configurations for recursion depth. It is important to mention that transmitted rays also count towards the recursive calls. For this reason, the only transparent sphere in Figure 4.2a is shaded in dark gray color. The better quality in images with higher recursion depth can easily be noticed. One should be able to easily notice the increase in image quality and photorealism of the images as the recursion depth increases.

In order to better visualize the impact of maximum recursion depth, Figure 4.2f presents the same scene using a recursion depth of 5 but with color-coding based on depth. The black portion of the image represents areas that did not hit any primitive. The rest of the image is color coded to represent the recursion depth necessary to render the given pixel. Namely, red, greed, blue,

yellow and magenta are used to represent recursion depths of 1, 2, 3, 4 and 5, respectively. Each of these colors represents the exact recursion depth needed to render a given pixel.



(a) Recursion depth = 1 (b) Recursion depth = 2 (c) Recursion depth = 3

(d) Recursion depth = 4 (e) Recursion depth = 5 (f) Depth color coding

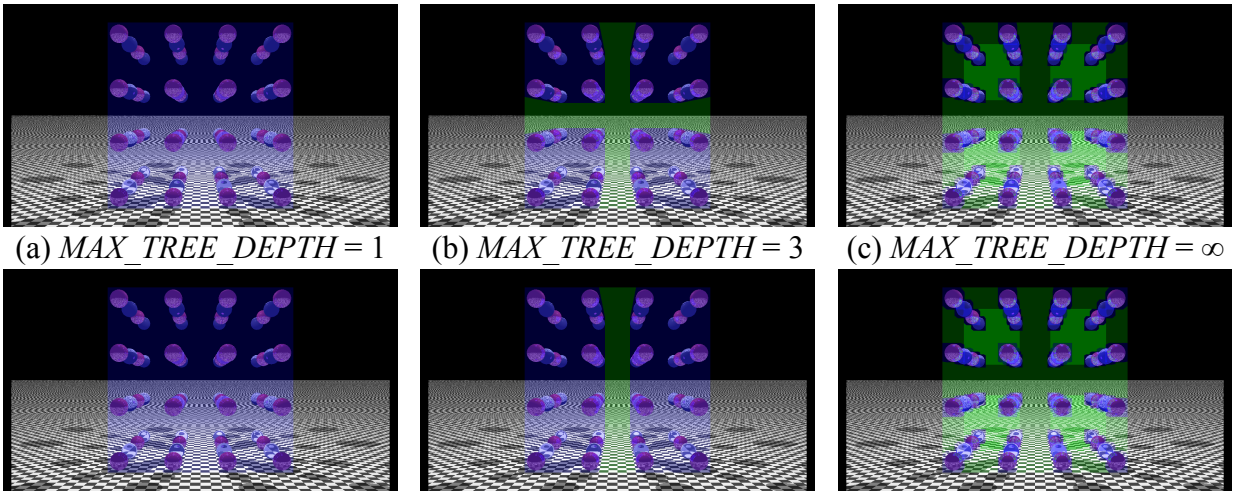
Figure 4.2: Renderer output for different recursion depth configurations.

D. BIH PARAMETERS

The BIH construction has several input parameters that can be tuned for better performance, and can greatly increase performance. In this section, two of the most relevant parameters are considered: *MAX_TREE_DEPTH* and *MAX_PRIMS*. These two parameters play a rather important role in defining the performance of the tree in the traversal process. In practice, they define how well balanced the tree will be, which directly affects performance.

In order to better visualize the different construction results for the BIH tree, the traversal algorithm was slightly modified. This change allows the algorithm to gather information about the traversal and encode it into the final pixel color. More specifically, if the ray passes through the empty space that may be present between sibling nodes, the green component in the final color is incremented by 0.2 Similarly, if the ray traverses through an internal node or leaf, the blue component of the color is incremented by 0.2.

Figure 4.3 shows the result of using different configurations for the maximum tree depth and maximum primitives per node in Scenario 1 with 64 primitives. The first three images had *MAX_PRIMS* equal to *infinity*, whereas the last three had *MAX_TREE_DEPTH* equal to the same value. The area shaded in blue represent that the traversal algorithm visited a leaf one, whereas the green areas mean that the ray traversed through the empty space between sibling nodes. In Figure 4.3b, four leaf nodes were generated with 16 primitives each. Similarly, the value of *MAX_PRIMS* in Figure 4.3e implicitly determined that the tree would be only two levels deep and each leaf node would have 32 primitives.



(a) *MAX_TREE_DEPTH* = 1 (b) *MAX_TREE_DEPTH* = 3 (c) *MAX_TREE_DEPTH* = ∞

(d) *MAX_PRIMS* = ∞ (e) *MAX_PRIMS* = 32 (f) *MAX_PRIMS* = 1

Figure 4.3: Illustration of the different results for various configurations of *MAX_TREE_DEPTH* and *MAX_PRIMS*.

One may have noticed that Figure 4.3a and Figure 4.3d, as well as Figure 4.3c and Figure 4.3f, are quite similar. In fact, they should be identical because both configurations generate exactly the same BIH tree structure.

In the first case, the tree is comprised of only the root node, which therefore contains all the primitives. This particular situation happens to be less efficient than the brute force approach in terms of performance. The reason for that is existence of a small constant overhead of using the

BIH that is related to the global AABB intersection test for every ray, which does not happen in the brute force approach.

In the second case, the tree has a leaf node for each primitive in the structure. The empty space between the nodes is also maximized in this case, which considerably increases performance. The brighter shades of green mean that the ray went through the empty space in multiple nodes that were stacked for later processing.

E. BIH STATISTICS

The sole use of acceleration data structures in a ray tracer does not mean that one should get optimal performance in every scenario. In fact, if the parameters for the data structure are not carefully chosen, one might even experience performance that is worse than if compared to not using the data structure. Not surprisingly, this statement is also true for the BIH.

In this section, each of the three scenarios that were explained earlier in this chapter was submitted to multiple tests using different construction configuration parameters. In all tests, there were 4096 spheres with radii equal to 1.0. The first test performed for all scenarios have the parameters *MAX_TREE_DEPTH* and *MAX_PRIMS* set to infinity and one, respectively. The combination of parameters for the first test is responsible for generating the most complex and memory intensive trees for each scenario. In other words, this test generates trees that are deep and have a leaf node for each primitive in the scene.

The results of this experiment are expressed in Table 4.2. For each scenario, the first and second configurations use values for *MAX_PRIMS* of 1 and 8, respectively. The rows contain statistics about number of nodes, primitives per node, tree depth, number of primitives in leaf nodes, number of optimized nodes, and build time.

		Scenario 1		Scenario 2		Scenario 3	
MAX_PRIMS		1	8	1	8	1	8
Nodes	Internal	4,095	511	4,861	816	4,316	774
	Leaves	4,096	512	4,107	812	3,780	763
Primitives per Node	Min	1.00	8.00	0.00	0.00	0.00	1.00
	Avg	1.00	8.00	1.00	5.04	1.08	5.37
	Max	1.00	8.00	1.00	8.00	2.00	8.00
Tree Depth	Min	13.00	10.00	5.00	5.00	10.00	9.00
	Avg	13.00	10.00	15.30	12.09	14.96	11.94
	Max	13.00	10.00	19.00	16.00	21.00	16.00
Leaves with N primities	N = 0	0.00%	0.00%	0.27%	1.11%	0.11%	0.13%
	N = 1	100.00%	0.00%	99.73%	5.05%	91.43%	2.49%
	N = 2	0.00%	0.00%	0.00%	8.87%	8.47%	6.03%
	N = 3	0.00%	0.00%	0.00%	10.10%	0.00%	12.71%
	N = 4	0.00%	0.00%	0.00%	12.93%	0.00%	17.04%
	N > 4	0.00%	100.00%	0.00%	61.95%	0.00%	61.60%
Optimized Nodes		0	0	755	5	537	12
Build Time (miliseconds)		15.18	6.06	20.31	9.57	24.05	8.07

Table 4.2: BIH statistics for all three scenarios with different configurations for *MAX_PRIMS*.

All scenarios had the total number of nodes in the tree reduced in the second configuration. In fact, Scenario 1 had the biggest reduction with 87.51%, followed by Scenario 2 with 81.85%, and finally Scenario 3 with 81.02%. Since the tree is perfectly balanced in Scenario 1, the number of primitives per leaf doubles each time the maximum tree depth is reduced by 1. Due to the same reason, both configurations in Scenario 1 have a characteristic in common: the fact that all leaf nodes always have the same number of primitives.

Scenario 2 presented the highest number of optimized nodes in the first configuration. This is because the primitives are distributed sparsely across the BIH volume, which implies in the existence of a great quantity of void space inside of the structure. This situation ultimately allows for the construction algorithm to build the structure more intelligently.

This section also captured results about the build time of each of the configurations. These measurements were captured based on a five round execution average. Scenario 1 with

MAX_PRIMS set to 8 had the shortest build time, whereas Scenario 3 with same parameter set to 1 had the largest. Although the former presented the absolute smallest build time, Scenario 3 showed the best relative performance improvement. This is because of the hybrid nature of the primitive distribution in the space. With a higher *MAX_PRIMS* value, the primitives that are clustered in the corners can effectively be combined into fewer leaf nodes. This ultimately makes the tree shallower and thereby decreases the build time.

Scenario 1 is particularly disadvantageous for the BIH. This is because the primitives are evenly distributed in the space and the construction algorithm cannot take advantage of empty spaces efficiently. This is the reason why this scenario has no Optimized Nodes in both configurations. In addition to that, given the high quantity of primitives in such a small volume, the possibility of node overlap is higher, therefore minimizing even more the void spaces. On the other hand, Scenario 2 is perhaps the most advantageous for the BIH. A high number of Optimized Nodes indeed shows that there is plenty of empty spaces in the structure.

F. BIH PERFORMANCE

An optimal configuration for the maximum number of primitives per leaf node can drastically affect the performance of the BIH. This is because the tree becomes shallower as this number increases, thereby decreasing the traversal time and a faster performance. However, the maximum number of primitives per node has to be chosen carefully. This is because when a leaf node is responsible for more than one primitive, the traversal algorithm needs to test every primitive in said node for intersection. In fact, this process can only stop after all primitives in the node were tested, and the closest intersection is found, if any.

Having said that, all three scenarios were submitted to performance tests. All tests in this section used spheres whose radii equals 1.0 and different settings for number of primitives in the

scene and maximum number of primitives per leaf node. The purpose of these tests is to measure the performance of the ray traversal process. In order to make the test simpler and avoid adding bias to the results, only primary rays were considered. In other words, the maximum recursion depth for the traced rays is equal to one, which disables the reflected and transmitted rays.

Figure 4.4 shows the results of this performance test. Each series in the charts represent a different configuration used for the ray traversal. The first series is named *BF* and represents the use of brute force approach. This approach does not use acceleration data structures or techniques. The other series represent BIH with different configurations for *MAX_PRIMS*. This number is clearly expressed in the name of the series. The *x* and *y* axes respectively represent the number of primitives used and the frame rate (in frames per second). Results present *BIH 8* with 64 primitives in Scenario 1 and *BF* with 256 primitives in all Scenarios as configurations with best and worst performance, respectively. In fact, the former achieved frame rates up to 60 *fps*, while the latter only delivered 2 *fps*.

The *trade-off* between the number of primitives in leaf nodes and the traversal performance is also clearly visible in the charts. For BIH configurations with *MAX_PRIMS* ranging from 1 to 16, the performance starts smaller, increases and hits the sweet spot, and finally decays and converges to the performance of BF approach. In this context, *BIH 8* is indeed the configuration that yielded best performance in 7 out of 9 configurations. Interestingly enough, the BIH configurations for Scenario 2 with 128 primitives did not seem to find the sweet spot. In fact, the performance only improved as the value for *MAX_PRIMS* increased. Although the performance decay for this scenario is not visible in the chart, great possibility exists that *BIH 16* actually represents the sweet spot for this combination of scenario and number of primitives.

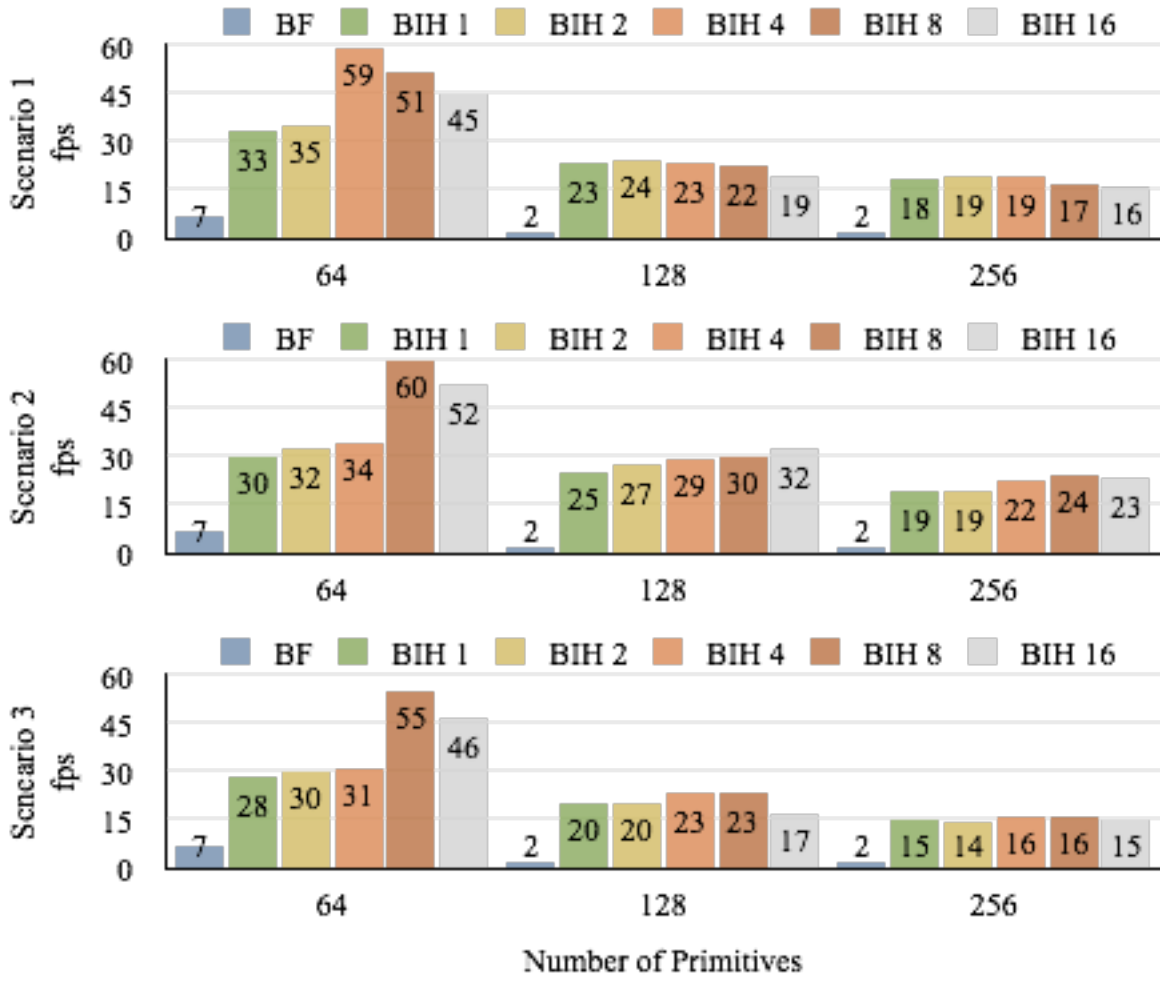


Figure 4.4: Performance results expressed in *fps* (frames per second) as a function of the scenario, number of primitives, and the configuration used.

One may question why the results of using the BF approach did not vary from one scenario to another. This question is perfectly reasonable, and its answer is quite simple: the BF approach does not care about how the primitives are distributed in the space. In fact, all spheres could be centered at the exact same point in the space, and still the performance would be similar to what was captured in this section. This is because the algorithm will always test all primitives for intersection for each pixel. Therefore, as long as the number of primitive remains the same, the results will remain somewhat stable.

Based on the results presented in this section, one can easily infer that Scenario 2 is indeed the most advantageous for the BIH. In this scenario, using *BIH 16* with 128 primitives yielded the highest improvement rate, whereas using *BIH 8* with 64 primitives yielded the lowest increase in performance. Respectively, the former and the latter delivered frame rates up to 1495% and 859% faster than the brute force approach. Overall, the average improvement rate for this scenario was 1212%.

On the other hand, the Scenario 3 was less than advantageous for the BIH because of the hybrid distribution of the primitives across the space. The BIH trees generated for this scenario were not well balanced, and the standard deviation of the number of primitives per leaf node was quite high. For these reasons, the BIH did not take the most advantage of this scenario, and thus did not yield optimal performance. The maximum performance increase in this scenario was 1074%, whereas the minimum was 787%. In addition to that, the average improvement rate in Scenario 2 was higher the maximum improvement seen in Scenario 3.

Due to hardware limitations and rounding inaccuracies some values in the charts may be prone to misinterpretation. First, the graphics hardware of the iPhone 5s is limited to displaying only 60 frames per second, regardless of how optimized the application is. In this way, possibility exists that the configuration that uses *BIH 8* with 64 primitives in Scenario 2 actually yielded frame rates higher than 60. Considering that Scenario 2 is actually the one that is the most advantageous for the BIH, and said configuration maximizes the empty spaces in an optimal way, this possibility is indeed great. Second, the results for the *BF* series in all scenarios are equal for number of primitives equal to 128 and 256. For visualization purposes, the decimal places were omitted in the figure, thereby causing the results to be rounded to the nearest integer.

The results for the *BF* series with 128 and 256 primitives are respectively 2.14 and 1.87 frames per second.

G. MEMORY FOOTPRINT

Choosing the optimal value for the maximum number of primitives per leaf node directly affects the memory footprint of the application. This is because, as it was explained in previous sections, this parameter influences how the BIH tree will be generated. Namely, the tree becomes shallower or deeper as the number of primitives per leaf node is bigger or smaller, respectively.

In this section, all 3 scenarios were evaluated similarly to what has been done in Section 4.6, but with some key differences. First, the number of primitives used changed to 1024, 2048 and 4096 primitives. Second, the radii of the spheres were minimized to 0.001 because the high number of primitives in such small space could lead to overlaps and incorrectness in the tree structure.

This experiment's measures include everything that involves the use of the BIH in the ray tracer. First, it accounts for the nodes allocated, both internal and leaf nodes. Second, the *quicksort* portion of the construction algorithm does not actually the actual primitives array, but a shadow copy instead. This copy only contains integer pointers to the primitive indices in the original array. Finally, it also considers the space needed for storing the global AABB.

Results of this experiment are in expressed in Figure 4.5. Each series represent a different configuration for the *MAX_PRIMS* parameter of the BIH. The number of primitives used and the memory consumed (in bytes) are respectively expressed in the horizontal and vertical axes. The *BIH 1* in Scenario 2 with 4096 was the configuration that had the highest memory usage with little over 150 kB, whereas *BIH 16* in Scenario 3 with 1024 primitives had the last usage with slightly less than 6.2 kB.

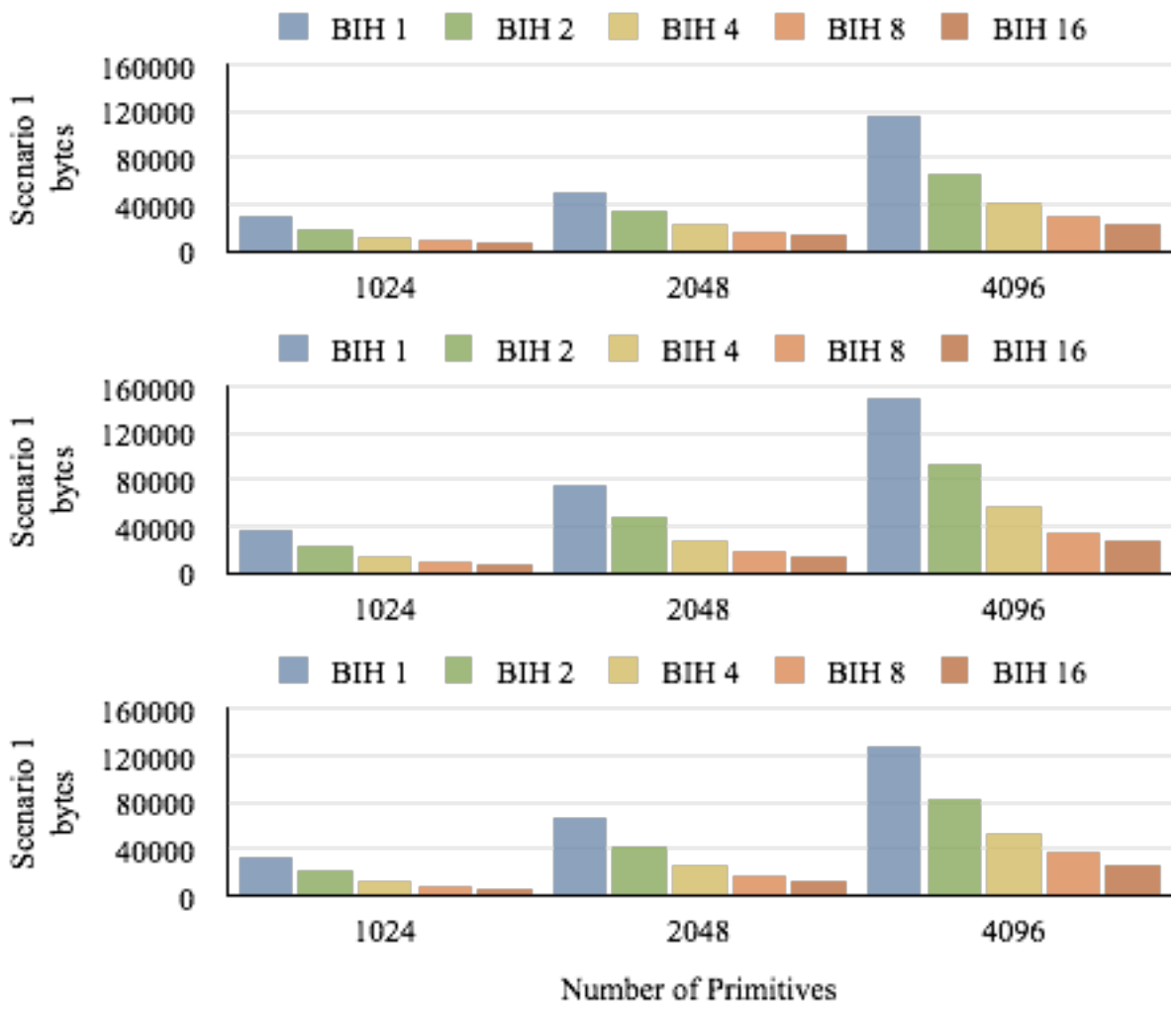


Figure 4.5: Memory consumption results, expressed in *bytes*, for combinations of scenario, number of primitives, and *MAX_PRIMS*.

Another interesting way of measuring of how well the BIH performs is to analyze the relative reduction in overall memory consumption by comparing it to *BIH 1* in the same Scenario / Number of Primitives configuration. It is not a surprise that the configurations with higher values for *MAX_PRIMS* will yield less memory consumption. In fact, in all scenarios and configurations, *BIH 16* had the highest relative reduction in memory usage. In that context, among all configurations that used *BIH 16*, the ones that presented the highest and lowest relative reduction in memory consumption were respectively Scenario 2 with 1024 primitives and Scenario 1 with

2048 primitives. The former showed 82.5% reduction in memory usage, whereas the latter presented a rate of 73.99%.

It is important to note that the scenarios that had the biggest and smallest overall memory consumption were Scenario 2 and Scenario 1 respectively. The reason behind this result is related to how the primitives are distributed in the space. For instance, each split operation in the internal nodes divide the primitives in exactly half for the Scenario 1. However, given the randomness factor that exists in Scenario 2, split operations are not guaranteed to do the same.

V. CONCLUSIONS

The goal of this research was to design, implement and evaluate a ray tracing system for mobile devices. In this section, we review our research accomplishments and describe several directions for future work.

A. RESEARCH ACCOMPLISHMENTS

We implemented two ray tracing algorithms and evaluated them based on rendering performance, memory footprint and other aspects. The first algorithm uses brute force to calculate the *pixel* colors and, not surprisingly, had the lowest performance. The second algorithm, however, uses the BIH as acceleration data structure, and thus was able to deliver significantly better results, yielding frame rates up to 14 times faster when compared to the brute force approach. Overall, using the BIH proved to be very efficient when rendering images for mobile devices.

We have explained that the BIH can dramatically increase memory consumption. However, due to its simplified approach to the node structure and AABB storage, and to its highly configurable construction parameters, the BIH proved to be able optimize memory footprint without hurting performance. In fact, by adjusting the maximum number of primitives per node, we found that it is possible to reduce memory consumption by up to 82.5% while still maintaining close to optimal performance.

Finally, we verified that the use of sophisticated heuristics in the construction process is not really necessary for obtaining good performance. This is because the global simplified heuristic used in the BIH was able to divide the space efficiently, thereby increasing construction performance. Furthermore, when it comes to rendering dynamic scenes, this approach helps minimize build time, which ultimately makes it a great option for rendering these types of scenes.

B. FUTURE WORK

Several important improvements were left as future work. Our system was implemented and evaluated on an Apple iOS platform. It would be interesting to see how our approach can be adapted for other platforms, such as Google Android and Windows phone. This will dramatically change the number of potential users of applications that use what has been implemented in this work. Second, our system used spheres as the only graphical primitives. In order to better represent most 3D models, other graphical primitive types need to be supported, such as triangles meshes, parametric surfaces, algebraic entities, and others. Third, other physical phenomena of the light must be considered, such as caustics and diffuse inter-reflection. This will considerably improve the visual quality of the rendered images.

One other important feature that greatly affects performance needs to be implemented: the use of texture samplers. This improvement would allow the data that contains the BIH tree, the objects, positions and other information to be transferred to the GPU more efficiently. In addition to that, this technique also allows for caching that is available across shader threads that run in parallel.

Even though the renderer already provides good performance, the BIH is a relatively new data structure, and more work should be done to understand subtle performance parameters. This is especially true when it comes to discovering how to optimize even more the construction and traversal algorithms.

VI. BIBLIOGRAPHY

[1] A. Appel. "Some Techniques for Shading Machine Renderings of Solids". In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: ACM, 1968, pp. 37–45. DOI: 10.1145/1468075.1468082. URL: http://doi.acm.org/10.1145/1468075.1468082.

[2] J. Foley et al. *Computer Graphics: Principles and Practice (2Nd Ed.)* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 0-201-12110-7.

[3] H. Fuchs, Z. Kedem, and B. Naylor. "On Visible Surface Generation by a Priori Tree Structures". In: *SIGGRAPH Comput. Graph.* 14.3 (July 1980), pp. 124–133. ISSN: 0097-8930. DOI: 10.1145/965105.807481. URL: http://doi.acm.org/10.1145/ 965105.807481.

[4] *GeForce GTX 780 Ti.* URL: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780-ti/specifications.

[5] A. Glassner, ed. *An Introduction to Ray Tracing*. London, UK: Academic Press Ltd., 1989. ISBN: 0-12-286160-4.

[6] *GLSL*. URL: http://www.opengl.org/documentation/glsl/.

[7] J. Goldsmith and J. Salmon. "Automatic Creation of Object Hierarchies for Ray Tracing". In: *IEEE Computer Graphics and Applications* 7.5 (May 1987).

[8] *Microsoft DirectX SDK*. URL: http://msdn.microsoft.com/en-us/directx/default.aspx.

[9] *Nvidia CUDA*. URL: http://www.nvidia.com/object/cuda_home_new.html.

[10] *OpenCL*. URL: http://www.khronos.org/opencl/.

[11] *OpenGL*. URL: http://www.opengl.org/.

[12] *OpenGL ES*. URL: http://www.khronos.org/opengles/.

[13] B. Phong. "Illumination for Computer Generated Pictures". In: *Commun.* ACM 18.6 (June 1975), pp. 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839. URL: http://doi.acm.org/10.1145/360825.360839.

[14] S. Popov et al. "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing". In: *Computer Graphics Forum*. Vol. 26. 3. Wiley Online Library. 2007, 415424.

[15] A. dos Santos et al. "kD-Tree Traversal Implementations for Ray Tracing on Massive Multiprocessors: A Comparative Study". In: *Proceedings of the 2009 21st International Symposium on Computer Architecture and High Performance Computing*. SBAC-PAD '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 41–48. ISBN: 978-0-7695-3857-0. DOI: 10.1109/SBAC-PAD.2009.25. URL: http://dx.doi.org/10.1109/SBAC-PAD.2009.25.

[16] M. Shevtsov, A. Soupikov, and A. Kapustin. “Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes”. In: *Computer Graphics Forum* 26.3 (2007), pp. 395–404.

[17] J. Teixeira et al. “Improving Ray Tracing Anti-aliasing Performance Through Image Gradient Analysis”. In: *Proceedings of the 2010 11th Symposium on Computing Systems*. WSCAD- SCC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 144–151. ISBN: 978- 0-7695-4274-4. DOI: 10.1109/WSCAD-SCC.2010.18. URL: http://dx.doi.org/10.1109/WSCAD-SCC.2010.18.

[18] C. Wächter and A. Keller. “Instant Ray Tracing: The Bounding Interval Hierarchy”. In: *Proceedings of the 17th Eurographics Conference on Rendering Techniques*. EGSR’06. Nicosia, Cyprus: Eurographics Association, 2006, pp. 139–149. ISBN: 3-905673-35-5. DOI: 10.2312/EGWR/EGSR06/139-149. URL: http://dx.doi.org/10.2312/EGWR/EGSR06/139-149.

[19] I. Wald. “Realtime ray tracing and interactive global illumination”. In: (2004).

[20] I. Wald, S. Boulos, and P. Shirley. “Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies”. In: *ACM Trans. Graph*. 26.1 (Jan. 2007). ISSN: 0730-0301. DOI: 10.1145/1189762.1206075. URL: http://doi.acm.org/10.1145/ 1189762.1206075.

[21] I. Wald and V. Havran. “On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)”. In: *Proceedings Of The 2006 IEEE Symposium On Interactive Ray Tracing*. 2006, pp. 61–70.

[22] A. Watt. *3D Computer Graphics*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993. ISBN: 0201631865.

[23] T. Whitted. “An Improved Illumination Model for Shaded Display”. In: *SIGGRAPH Comput. Graph*. 13.2 (Aug. 1979), pp. 14–. ISSN: 0097-8930. DOI: 10.1145/965103. 807419. URL: http://doi.acm.org/10.1145/965103.807419.