University of Nebraska at Omaha

Thesis-Equivalent Topic Proposal

Achieving Real-Time Raytracing with a GPU-constructed Bounding Volume Hierarchy

Alex Wissing

Department of Computer Science

Supervisory Committee:

Brian Ricks

Victor Winter

Dora Velcsov

Abstract

Raytracing has many benefits as technique for simulating light while rendering a scene, resulting in many effects like reflections, refractions, shadows, anti-aliasing, soft-shadows, ambient occlusion, and global illumination being achieved as a result of the techniques similarity to lights behavior rather than by deliberate addition as is done in non-raytraced real-time renderers. However, this technique comes at a great runtime cost. In order to make runtime raytracing feasible, it's necessary to optimize some of the more expensive components of a Monte-Carlo raytracer. The inclusion of a BVH (bounding volume hierarchy) reduces the runtime cost of hit detection, however, the cost of building the BVH and transferring it to the GPU (graphics processing unit) is expensive, due to the lack of parallelization available on the CPU compared to the GPU. To balance realistic rendering techniques with runtime performance, this paper aims to implement a method of building the BVH on the GPU [7]. This project aims to construct a GPU-based Real-time Raytracer which assembles and uses a Bounding Volume Hierarchy in a Vulkan Compute Shader. The image produced from the shader is then rendered to the screen.

1 Introduction

The field of computer graphics focuses on creating images with computers. It's involved in movies, video games, computer screens and digital art.

Real-time rendering is a field in computer graphics which focuses on producing and modifying images in real-time, that is, as a program is running.

Rendering is the process of generating an image from a model. It's used in films, TV shows, architectural programs, and games. These images, renderings, can be used to create entertainment, visualizations, and simulations. A common need is relatively accurate real-world effects appearing in the rendering. One of the most challenging effects to emulate is light. Computer graphics techniques involve many algorithms for reproducing the complex effects of light that color the

world. Many of these algorithms have long been too slow to be used in real-time applications. However, hardware improvements and faster algorithms have made some of these techniques feasible in real-time. Raytracing simulates the behavior of light by assume it behaviors like a geometric ray and working backwards; instead of sending light rays from a light-emitting surface which eventually arrive at a camera, rays are sent from the camera and eventually arrive at a light-emitting surface.

In conventional, real-time rendering, a number of methods are used to approximate the effects of light on a scene in order to pseudo-realistically display a digitized world on a computer screen. These methods are effective and fast but require individual implementation and consideration. Another technique, Raytracing, was previously not feasible in real-time due to it's demanding requirements on hardware and lack of fast algorithms in software. However, as hardware and software techniques has improved, real-time rendering applications, such as video games, have begun utilizing Raytracing to simplify their rendering pipelines, reduce game sizes, and capture complex effects of light.

Raytracing involves casting rays into a scene to simulate the linear motion of waves or particles. In the case of this paper, light. To approximate light's behavior in the scene, a Monte-Carlo summation is used to approximate The Rendering Equation [4]. Monte-Carlo refers to the process of approximating an integral result from mathematics by using a finite number of averaged samples.

Raytracing can produce photo-realistic images but the cost of simulating millions of rays bouncing around a scene, which is necessary to produce the image, it a very expensive process computationally. Creating a raytracer for use in real-time requires consideration of the bottle-necks associated with the algorithm. One of these spaces is its linear scalability in poly-count. In order to cast a ray into a scene, the algorithm involves determining when the ray intersects with triangles. This requires checking each triangle to find the closest intersection.

As light, often represented in raytracers by a geometric ray, travels through a scene, it potentially interacts with some surface. In this case, depending on the properties of the surface, the ray is bounced off the surface. These surfaces are often represented by primitive shapes, such as triangles and spheres.

When determining which primitive shape, if any, a ray hits, every primitive shape must be checked (or ruled out in some way). This is because we not only need to know if we hit something, but also what was the closest primitive hit. This creates a runtime cost per ray with depth of 1 of O(n), where n is the number of primitives. This cost is true regardless of if a hit is found while looping or if a ray hits nothing.

However, data structures have been used to reduce the complexity of finding the closest hit primitive. This is done by correlating the space a primitive occupies with it's location in a tree-like structure and only testing for intersections upon a small group of primitives. This is called a Bounding Volume Hierarchy [8].

This project aims to implement, in a real-time raytracer, an improvement to reduce linearity of this search utilizing a Bounding Volume Hierarchy. This tree-like structure enables faster intersection calculation by computing intersections only for a subset of the triangles in the scene. From a search perspective, this structure enables intersection checking to function similar to binary search. By moving down the tree on a particular path, large groups of triangles are avoided, resulting in speed improvements.

However, a BVH must be constructed (similar to sorting a list for use in binary search). In a real-time application where triangle primitives are being transformed each frame, the BVH must either be built each frame, or modified each frame after initial construction.

This project aims to utilize a BVH in a GPU-accelerated raytracer and avoid expensive construction costs through GPU parallelization. This real-time raytracer as the main rendering pipeline for a custom game engine, in the interest of further developing my own understanding of game engines, so some additional capability such as being able to move primitives, adding primitives, and removing primitives in real-time are also included.

To construct a BVH, in a simple way, a bounding volume is selected, often an axis-aligned bounding rectangular prism. The initial bounding volume is created such that it contains all the primitives in the scene. This volume is the root of the tree. Then the primitives are sub-divided into a number of some groups. Each groups becomes a child of the root node by constructing a bounding volume con-

taining all their respective primitives. This repeats to some degree of granularity (for instance, such that every leaf node contains a single primitive).

To traverse a BVH, a ray intersection algorithm for bounding volumes can be used. If the ray hits, then it might hit something contained within and thus attempts to hit the children of the volume. Once the ray hits a bounding volume that has no children, the ray attempts to hit any primitives contained within the volume. It can still miss at this point.

In a theoretical case, each time the ray traverses one level down the tree, it removes half of the remaining hit-able primitives from it's search. This is an incredible performance boost, but comes at the cost of needing to construct the BVH.

Another topic in this paper are Morton codes, also known as a Morton curve, Z-order curve, or Lebesgue curve. It allows for sorting any multi-dimensional, integral point into a 1-dimensional integral by interleaving the bits of the components of the multi-dimensional point. This also created an implicit and useful ordering of higher dimensional space that can be used in lower-dimensional data structures like arrays, sorting algorithms, hash-tables, and more.

This proposal will review past works in the field or raytracing and bounding volume hierarchies as well as list resources used to learn and study this topic. Next, a review of the development of the raytracer so far, then a detailed plan for the implementation of a BVH including a list a project specific success criteria which should be used to judge whether the implementation of the BVH is successful. Lastly, a conclusion discussing the proposal in summary.

2 Related Works

Many works have contributed to the space of raytracing.

Texts such as Shirley, Black and Hollasch [9] [10] [11] as well as Pharr, Jakob, and Humphreys [8] provided much of the information about raytracing, the processes involved, as well as optimizations and improvements. In these texts, the authors discuss the nature of a raytracer and detail how to create on that runs on the CPU. Some algorithms from these texts, like the triangle intersection algorithm

2 are from these texts. Pharr, Jakob, and Humphrey's text, Physically-Based Raytracing: From Theory to Implementation, creates an SAH-based HLBVH, which is similar to the goal of this project, however the SAH-based HLBVH isn't as quick to build as what this proposal will explore.

Meister et al. [6] and Garanzha and Loop [2] improve raytracing performance by organizing rays to reduce execution divergence and capitalize on locality and directional similarity between rays. This ray-reordering method packages information about groups of rays into packets and deploys them to the gpu to cast rays using packet information. This can improve performance in a number of ways compared to the method used in this paper: smaller work tasks without shared dependencies allow the GPU to run as many as possible without manual intervention, packets allow for rendering some regions more than others to make sure areas of high complexity are well rendered while areas of low complexity are quickly rendered, and this also allows for rending multiple rays per pixel at once. The use of Morton Codes in [2] inspired the application in BVH construction.

Much work has specifically gone into the implementation of acceleration structures like a BVH.

Bauszat, Eisemann, and Magnor [1], prioritize the BVH's memory footprint by reducing its per-node size to just 2 bits, which they find to be the smallest possible representation that doesn't produce empty space deadlocks. As is often the case, decreasing memory utilization increases runtime cost. Knowing this minimum does create an algorithmic target, much in the same way that knowing the minimal runtime for BVH construction does ($O(n \log_2 n)$).

Guo, Zhang, and Zhou [3] work on an improvement to the proposed algorithm in this paper, which is based on Pantaleoni and Luebke [7], by balancing between the topological performance gains with construction time. This approach constructs the BVH from the bottom-up using clusters. They first create the leaf node for all primitives by grouping together primitives based on their Surface Area Heuristic cost. The higher order tree connections are deduced similarly. This improves bottom-up construction time, which creates long chain of nodes by adding primitives on each level until no primitives remain.

Pantaleoni and Luebke [7] offer an improvement to Lauterbach et al. [5]

by using aspects of spacial locality inherent in Morton codes alongside a process of Compress-Sort-Decompress proposed for BVH construction by Garanzha and Loop [2]. These papers establish the Linear Bounding Volume Heirarchy (LBVH) and, its improvement, the Heirarchical Linear Bounding Volume Heirarchy (HLBVH), which will be the main topic of this proposal and are further detailed in Section 4.

3 Development Progress

The project has thus far been accomplished using C++, the Vulkan graphics API, the OpenGL Mathematics Library (GLM), and the Graphics Library FrameWork (GLFW). This section will highlight some of the current capabilities and features already developed as well as challenges along the way.

3.1 Encapsulating Vulkan API to classes

The Vulkan API is C-based and involves the use of POD (Plain-old Data) Objects to pass configuration information as well as some object primitives used to maintain memory references across the CPU and GPU. Many of these API calls are used frequently and were abstracted to classes to more easily and quickly develop the application.

Notable examples of this include the Buffer class, which maintains VkBuffer and VkDeviceMemory as well as some affiliated information such as memory flags used in the creation of the buffer and the number of elements and element size. Utility functions are also provided to allow mapping and unmapping memory to the GPU or flushing CPU-side buffer content to the GPU.

Another example is the Device class, which collects information about the machine, selects which device (GPU usually) to use and maintains a reference used in most calls to the Vulkan API. It also maintains the command pools, surface used in the window to display resulting images, and command queues.

3.2 Support for Triangles and Spheres

Most conventional renderers use exclusively triangles. This makes rendering spheres somewhat costly, as what can be described as a point and a radius becomes easily more than 20 triangles. This is due to the need to increase poly-count to create the appearance of curvature. Conventional renderers are highly optimized for rendering flat triangles, so objects are decomposed into triangles.

However, in a raytracer, the only expense based on the primitive is the intersection algorithm and the memory used. In both cases, spheres outperform triangles. However, not every model can be decomposed into spheres, so both triangles and spheres are supported.

3.3 Sphere Intersection Algorithm

This sphere intersection algorithm [9] takes a sphere and a ray suspected of intersecting as well as a hit interval (Min and Max) and an object to record useful intersection information if an intersection does occur.

A Sphere can be described with the following formula:

$$x^2 + y^2 + z^2 = r^2$$

if the center is at the origin, or as

$$(x - cx)^2 + (y - cy)^2 + (z - cz)^2 = r^2$$

if the center is at a point C (cx, cy, cz). Here, r describes it's radius and P (x, y, z) are a point on the surface on the sphere. This equation can be used to compute whether a point lays upon the sphere's surface. The same equation in a vectorized form is as follows:

$$(P - C) \cdot (P - C) = r^2$$

If the left side is less than r^2, P is inside the sphere. If greater than, P is outside the sphere. And if equal, P is on the sphere. If we presume an intersection occurs, then we can define P as a function of the ray:

$$P(t) = A + tb$$

Algorithm 1: sphereIntersect()

Input: Ray R, Sphere S, float Min; minimum distance along ray to be considered an intersection, Max; maximum distance along ray to be considered an intersection, H; struct containing data about an intersection if one does occur

Output: $True$ if an intersection occured, $False$ otherwise

```
1  vec3 oc = R.origin - S.center; /* A − C                                  */
2   float a = dot(R.direction, R.direction); /* b · b                      */
3   float halfB = dot(oc, R.direction); /* b · (A − C)                      */
4   float c = dot(oc, oc)- (s.radius * s.radius); /* (A − C) · (A − C) − r²  */
5  float underRadical = (halfB * halfB) - (a * c);
6  if underRadical < 0 then
       /* if not 0 or positive, no roots                                    */
7      return False;
8  float radical = sqrt(underRadical);
9  float root = (-halfB - radical) / a;
10 if root < Min or root > Max then
11     root = (-halfB + radical) / a;
12     if root < Min or root > Max then
13         return False; /* a hit would occur, but outside hit interval    */
   /* record hit information in H                                          */
14 return True;
```

where A is the origin of the ray, and b is the direction of the ray. Transforming the sphere equation before we get:

$$(A + tb - C) \cdot (A + tb - C) = r^2$$

Further simplifying, we get the polynomial:

$$(b \cdot b)t^2 + (2b \cdot (A - C))t + (A - C) \cdot (A - C) - r^2 = 0$$

Applying the quadratic formula, if no roots are found, no intersection occurs. If one root is found, the ray is tangential to the sphere and intersects. If two roots are found, the ray passes through the sphere and intersects.

3.4 Triangle Intersection Algorithm

Algorithm 2: triangleIntersect()

Input: Ray R, Triangle T, float Min; minimum distance along ray to be considered an intersection, Max; maximum distance along ray to be considered an intersection, H; struct containing data about an intersection if one does occur

Output: $True$ if an intersection occured, $False$ otherwise

```
1  vec3 u = T.v1 - T.v0;
2  vec3 v = T.v2 - T.v0; /* Compute triangle direction vectors u and v    */
3  vec3 normal = cross(u, v);
4  vec3 nNormal = normalize(normal);
5  float D = dot(nNormal, T.v0);
6  vec3 w = normal / dot(normal, normal);
7  float denom = dot(nNormal, R.direction);
8  if |denom| < 0.0001 then
9      return False;
10 float t = (D - dot(nNormal, R.origin)) / denom;
11 if t < Min or t > Max then
12     return False;
13 vec3 intersectionPoint = R.origin + t * R.direction;
14 vec3 pointOnTrianglePlane = intersectionPoint - T.v0;
15 float a = dot(w, cross(pointOnTrianglePlane, v));
16 float b = dot(w, cross(u, pointOnTrianglePlane));
17 if a < 0 or b < 0 or a + b > 1 then
18     return False;
   /* record hit information in H                                          */
19 return True;
```

This triangle intersection algorithm [10] takes a triangle and a ray suspected of intersecting as well as a hit interval (Min and Max) and an object to record useful intersection information if an intersection does occur.

This algorithm uses a secondary formulation of a triangle which has many advantages. A triangle can be defined as three vertices, but it can also be defined as a point and two direction vectors. The point Q and the 2 direction vectors u and v have many beneficial properties that are used here:

- vertices exist at Q, $Q + u$, and $Q + v$
- u and v span R^2
- u and v are formed by subtracting the Q vertex from the other two vertices. This means $Q + au + bv$ yields a point on the triangle if $a >= 0$, $b >= 0$, and $a + b < 1$

While useful, the render pipeline consumes triangles as a collection of 3 vertices, so u and v are formed during the intersection algorithm.

First, a plane is formed to see if an intersection occurs with the plane. A Plane is defined by:

$$Ax + By + Cz = D$$

it can also be defined by a normal vector and position vector:

$$n \cdot v = D$$

where n is a vector perpendicular to the plane and v is a position on the plane. With a point on the plane, as in Section 3.3, we can define this point as an intersection point:

$$n \cdot (A + tb) = D$$

where A is the ray origin, b is the ray's direction vector, and t is a multiplier on the direction vector used to determine if the intersection is within the hit interval. Rearranging to solve for t:

$$t = \frac{D - n \cdot A}{n \cdot b}$$

So we can compute t easily, but this is for a plane, not a triangle. If we hit the triangle, it is hit at $A+tb$, but first we need to know if the point $A+tb$ is actually in the triangle. Let's suppose we have a point H on the plane of a triangle composed of the point Q and direction vectors u and v. Because u and v span R^2, they can be used as basis vectors to uniquely identify any point on the plane of the triangle.

$$H = Q + au + bv$$

If we can solve for a and b, we can test if H exists on the triangle. Removing the points to just focus on the origin:

$$h = H - Q = au + bv$$

h has is used but cancels some things. To quickly show those cases first:

$$u \times h = u \times (au + bv) = a(u \times u) + b(u \times v)$$

$$u \times u = 0 => u \times h = b(u \times v)$$

Similarly:

$$v \times h = v \times (au + bv) = a(v \times u) + b(v \times v)$$

$$v \times v = 0 => v \times h = a(v \times u)$$

We can solve for a and b, as crossing u and v with h cancels out a and b respectively. The plane's normal n is defined by $u \times v$, so:

$$n \cdot (v \times h) = n \cdot a(v \times u)$$

$$n \cdot (u \times h) = n \cdot b(u \times v)$$

solving for a and b:

$$a = \frac{n \cdot (v \times h)}{n \cdot (v \times u)}$$

$$b = \frac{n \cdot (u \times h)}{n \cdot (u \times v)}$$

We need to swap the cross products of the a formula so that have a common denominator:

$$a = \frac{n \cdot (h \times v)}{n \cdot (u \times v)}$$

Now create a vector w defined as:

$$w = \frac{n}{n \cdot (u \times v)} = \frac{n}{n \cdot n}$$

Then

$$a = w \cdot (h \times v)$$

$$b = w \cdot (u \times h)$$

If $a + b > 1$ or if either a or b is < 0, H is not inside the triangle. Interestingly, the same algorithm can be used for a Quadrilateral, but instead of checking for $a > 0, b > 0, a + b < 1$, one can check for $0 < a < 1$ and $0 < b < 1$.

3.5 Gamma Correction

This project employed Gamma correction to distribute color into spaces humans have better ability to distinguish colors. Perceptions of brightness for humans allows more distinction in darker tones than in lighter ones. Linearly distributing the brightness of the image would mean nuance in brighter areas would be lost due to our perceptions. To account for this, gamma compression is used; specifically:

$$V_{\text{out}} = AV_{\text{in}}^{\gamma}$$

where $A = 1$, $\gamma = 1/2$, and V_{out} represents the resulting colors of each pixel in the image and V_{in} represents the colors of each pixel produced by the raytracer.

3.6 Game Engine Organization and Scene Tools

This raytracer allows for objects to be added, removed, and modified while rendering is occurring. The states of all objects in the scene are maintained on the CPU and transferred to the GPU every frame.

Scenes maintain a list of GameObjects which maintain their individual model, transform, and components list. GameObjects can be added, modified, and removed in real-time from scenes. GameObjects can load models from .obj files which are transformed into triangles in memory. A Material, containing information about how to render the object, is attached to each GameObject.

3.7 Render Pipeline

Three stages are used to generate the resulting image. The first applies model transforms to triangles and spheres from model space into world space.

$$A * M = W$$

where M is a 4x1 column vector representing a vertex in model space, W is a 4x1 column vector representing a vertex in world space, and A is a 4x4 matrix encoding transformations (translation, rotation, and scale) for a particular object

into a 4x4 matrix of the following form:

$$\begin{bmatrix} a & b & c & tx \\ d & e & f & ty \\ g & h & i & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here, tx, ty, and tz store translation information for the axes x, y, and z respectively. The constant 1 at index (3, 3) causes these translation values to be additively applied. Variables $a - i$ are a particular chosen rotation convention and encoded both rotation and scale. In this project, Tait-Bryan Angles with axis order Y, X, Z are used. Thus the variables are defined as follows:

$$c_1 = cos(rotation.y)\ s_1 = sin(rotation.y)$$

$$c_2 = cos(rotation.x)\ s_2 = sin(rotation.x)$$

$$c_3 = cos(rotation.z)\ s_3 = sin(rotation.z)$$

$$a = scale.x(c_1c_3 + s_1s_2s_3)$$

$$b = scale.y(c_3s_1s_2 - c_1s_3)$$

$$c = scale.z(c_2s_1)$$

$$d = scale.x(c_2s_3)$$

$$e = scale.y(c_2c_3)$$

$$f = scale.z(-s_2)$$

$$g = scale.x(c_1s_2s_3 - c_3s_1$$

$$h = scale.y(c_1c_3s_2 + s_1s_3)$$

$$i = scale.z(c_1c_2)$$

This matrix, A, is known as a 3D affine transformation via homogeneous coordinates. After transforming each vertex in this way, the triangles and spheres are ready for the raytracer. This stage identifies pixel locations, generates an initial ray based on camera location, and checks to see if the ray collides with anything

Algorithm 3: getRay()

Input: vec2 xy; coordinates identifying the pixel

Output: Ray

```
1 vec3 rayOrigin = cameraPosition;
2 vec3 pixelSample = (pixel00Location + (xy.x * pixelDeltaU) + (xy.y *
   pixelDeltaV));
  /* The pixel00Location, pixelDeltaU, and pixelDeltaV variables are
     similar to Q, u, and v as described in the Triangle Intersection
     algorithm.  However, here, the vectors pixelDeltaU and pixelDeltaV
     are of length 1-pixel on the projected image surface in width and
     height respectively                                              */
3 vec3 rayDirection = pixelSample - rayOrigin;
4 return Ray(rayOrigin, normalize(rayDirection))
```

in the scene. If it doesn't collide with a triangle or sphere, it collides with the background (usually black (0, 0, 0, 1)). If it does collide with something besides the background, emitted light from the collided surface and the color of the collided surface are recorded, and the ray is scattered according to the properties of the surface it has collided with. Currently, only diffuse materials are handled by the scatter function. After getting a color for the ray, the color is added to the current color from the output image for that pixel, and then stored in that output image for that pixel. A pixel's color is a 4x1 vector (r, g, b, a) stored as 4 32-bit floating point numbers. When interpretted to deduce the pixel color in the final image later, these values are expected to be in the range $0 < c < 1$ where c is any component of the color vector. Using 32-bit floats instead of 8-bit integers allows for a larger range of colors to be expressed but also allows accumulation beyond the 0 to 1 range, which is useful considering the Monte-Carlo nature of the raytracer. This raytracer stage can produce an image with color values far in excess of this range due to repeated usage (ie, sending more than 1 rays per pixel) or due to initial brightness values being outside the range. This issue is corrected in the next and final stage. The raytracing stage can be run multiple times, each time firing a new ray for each pixel into the scene.

Algorithm 4: rayColor()

Input: Ray R

Output: vec3 Color

```
1  HitRecord rec;
2  vec3 color = vec3(0);
3  vec3 globalAttenuation = vec3(1);
4  vec3 unitDir = normalize(R.direction);
5  Ray curr = Ray(R.origin, unitDir);
6  for uint i = 0; i < MAXRAYTRACEDEPTH; i++ do
7      if !sceneHit(curr, rec) then
8          color = color + (BACKGROUNDCOLOR * globalAttentuation);
9          break;
10     vec3 attenuation;
11     vec3 emmitedColor = emmitted(rec);
12     color = color + (emmittedColor * globalAttenuation);
13     bool scattered = scatter(curr, rec, attenuation, curr);
       /* scatter() sets attenuation and curr to new values          */
14     globalAttenuation *= attenuation;
15     if !scattered then
16         break;
17 return color;
```

After generated a raytraced image, it needs to be rendered to the screen. This is done by placing a triangle to entirely take up vulkan's canonical view volume and mapping the raytraced-image's pixels to locations on the triangle. The vertex shader creates the triangle and the fragment shader returns image pixel values instead of any color associated with this triangle. [12]

The fragment shader does a few corrections as well. First, it divides the raytraced-image's pixel values by the total number of rays fired. This avoids the issue of going outside the interprettable range due to firing multiple rays. To handle the case of initial brightness values being too large, the colors are clamped into the required range. Also in this stage, gamma correction, as described in

section 3.5, is applied. In this stage, the alpha component for every color is set to 1, as the raytracer doesn't need to use the alpha channel for it's usual purpose and is free to use it in other ways.

3.8 Challenges So Far

The main challenge has been converting CPU raytracer code to work on the GPU. A number of factors become problems when moving to the GPU, especially random sources and synchronization.

Many randomization engines rely on a single-threaded environment, or at least involve a mutex, for generation. This poses a problem on the GPU and likely explains why few tools offer random generation tools on the GPU. For this project, a random seed was needed for each pixel as rays were cast repeatedly. To achieve this, each frame was given a seed, then the alpha channel of the resulting image stored a bias value created based on the seed after use. This bias value was used to modify the seed per pixel in further iterations, creating a seed for each pixel. This choice results in a poor source of randomness for the simulation, which is likely at fault for some of the minor discrepencies that can be seen when comparing to a CPU-based algorithm (see Section 3.9).

With a CPU-based raytracer, synchronization isn't a problem in many cases. But on the GPU, everything is run at once, making it difficult to write some types of functions. The ray color algorithm 4 uses additional variables, as the CPU-based version used recursion (specifically tail-recursion), which isn't allowed by GLSL. Another way synchronization becomes an issue is when parallelizing operations that have dependencies. By the nature of the Command Buffers used by Vulkan, tasks are started in an order but may complete in any order, unless synchronization tools, provided by Vulkan, are used to prevent starting of tasks before the completion of others.

3.9 Output Examples

Figure 1 and Figure 2 show a comparison of the output from a two raytracers, one running primarily on the CPU and another running the raytracing portion of

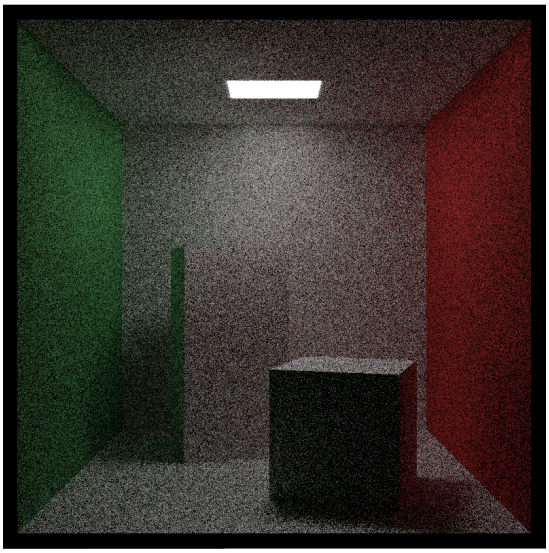

Figure 1: GPU Raytraced

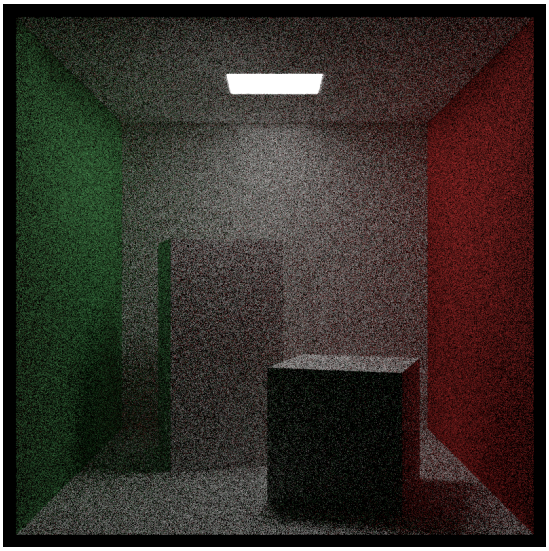

Figure 2: CPU Raytraced

the program on the GPU. Both 800x800 images were produced with 128 rays per pixel, with each ray being able to bounce up to 25 times.

CPU Raytracer	GPU Raytracer
621.186 seconds	1.287 seconds

Table 1: Runtime Comparison of CPU and GPU raytracers rendering a single frame of the Cornell Box

The images are produced in vastly different times with comparable quality. Comparing the time to generate a frame as the number of rays per pixel changes yields the graph in Figure 3. This chart shows approximately a 500 times increase in performance, though notably, the image in Figure 1 does appear slightly more noisy. This is likely a result of the poor randomization engine mentioned in Section 3.8.

4 Proposed Approach

To improve upon this raytracer, several things can be improved: A BRDF (Bi-directional Reflectance Distribution Function) can be used to reduce the amount of depth (how many times rays can bounce without hitting a light source or the

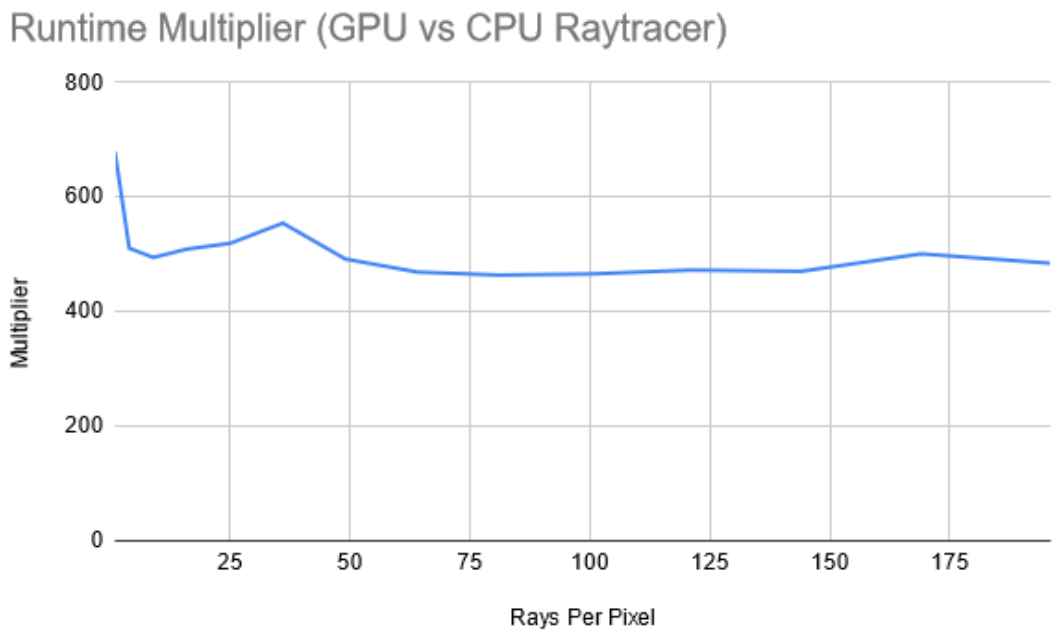

Figure 3: Number of Multiples faster the GPU renders the Cornell Box compared to the CPU with varying rays per pixel

background) and the number of rays needed per pixel, a swapchain can be used to avoid waiting for each frame and all associated operations to be finished (this would allow one frame to be ray tracing while another clears the previous result and another loads in engine primitives), rays can be dispatched in a more orderly way to correlate with the space they may interact with, and a de-noising algorithm can be applied after the image has been rendered to smooth out the spotty coloring allowing for a reduced number of rays to be cast per pixel.

For this project, the improvement that will be worked on will be the introduction of a BVH (Bounding Volume Hierarchy).

For this proposal, a HLBVH (Hierarchical Linear Bounding Volume Hierarchy) will be implemented as described in work by Pantaleoni and Luebke [7] which extends and improves upon work by Lauterbach et al. [5] as well as Garanzha and Loop [2]. The approach propose modifying the construction problem into a sorting problem, by first performing dimensionality reduction using Morton Codes. Then sorting the primitives by these codes in two ways: a top-level primitive sorting step and a bottom-level sorting step. This top-level step utilizes the nature of Morton codes to associated groups of primitives with neighboring primitives. Finally, a

process is used to deduce hierarchy of the BVH from the sorted Morton codes.

For the first step, each primitive is contained within an axis-aligned bounding box (AABB). The entire scene is also enclosed into an AABB. The N primitive-containing AABB in 3D space is reduced to a Morton code by interleaving a quantized version (quantized by a $2^k \times 2^k \times 2^k$ lattice, which produce a $3k$-bit Morton code) of its Barycenter's 3 coordinates together.

Algorithm 5: quantize()

Input: float x, float sMin; scene min for this axis, float sMax; scene max for this axis, uint k; number of bits to use

Output: uint quantizedCoordinate

```
1  uint output = 0;
2  float min = sMin;
3  float max = sMax;
4  for ; k != 0; k -= 1 do
5      float mid = (max - min) / 2.0;
6      if x > mid then
7          output |= 0b1 « (k - 1);
8          min = mid;
9      else
10         max = mid;
11 return output;
```

After quantization of each axis of a barycenter using Algorithm 5, the bits can be shifted to interleave all three values into a single int.

Then, the Morton codes are run-length encoded based on their higher level $3m$-bits. This is because every group of 3 bits in a Morton code identifies, in 3D space, an Octant. The next 3 bits identifies an Octant within the higher level Octant. m is a parameter to this process and must be smaller than k. The unique, encoded run values (the highest $3m$-bits of unique Morton codes) and their indices are then used in a $3m$-bit radix sort with the run values as keys and the indices as values. After sorting, the lengths of sorted consecutive runs are used in an exclusive scan algorithm to compute the offsets in the unsorted run values. These

offsets are used to decode indices of run values, which are decodes once more to yield Morton codes sorted by their upper $3m$ bits.

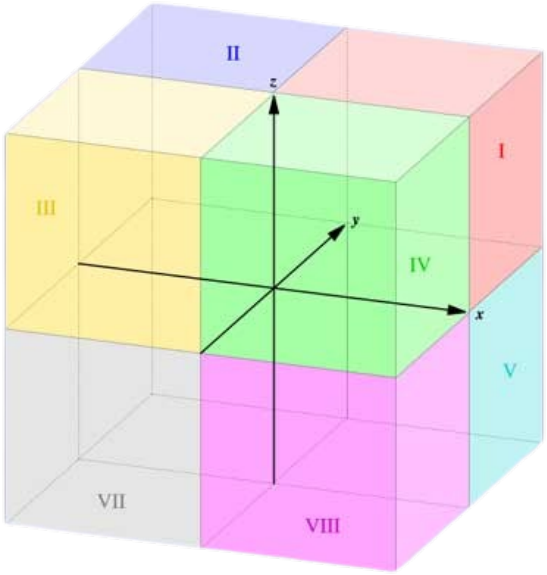

Figure 4: Octant

Figure 4 shows how the $3m$-bit groups can be used to identify locations. The location of the primitive is described with successive levels of specificity as each group of 3-bits further specify it's region, creating a natural, tree-like structure. Sorting happens to, then, group them into Octants in a sequence to easily parallelize the tree-construction process.

To sort the remaining aspects of the Morton codes, an odd-even sort is used as it is fast in highly parallel environments.

Lastly, to form the BVH, the sorted Morton codes are used. The algorithm looks to p-bits of the Morton codes, and attempts to form all the nodes of a level given the $(p-1)$-bit-level was previously computed. The binary split occurs based on the first element in the segment of sorted Morton codes with at bit position $3k-p$ to have a 1. All nodes before this within the segment are in the left and all nodes after this within the segment are in the right. These left and right groups are then the segments for the next level. Special cases, such as when a segment's Morton codes contain only 0s or 1s at bit position p are marked as they can be simplified later.

The resulting tree produced can then be flattened into an array, depth-first, so it may be easily traversed later during raytracing.

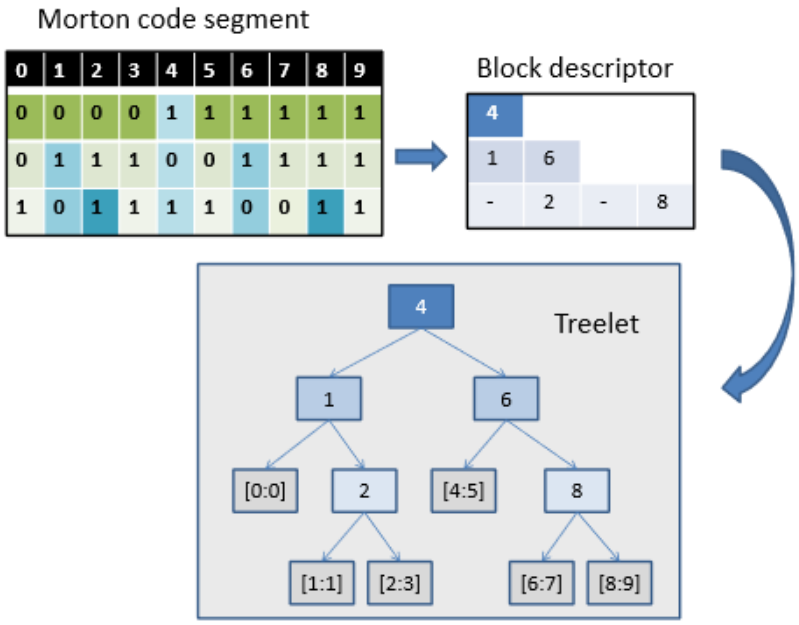

Figure 5: sub-tree emission process

For traversal, a stack and a few values can be used to keep track of where to look next. This algorithm is described in pseudo-code in Algorithm 6. The closest hit still needs to be found, so an exhaustive search must be done, but if a node isn't hit during traversal, then any primitives within itself or its children can safely be ignored. When a node is hit, if it is a leaf node, then it's primitives can be checked for hits. Any hits found decrease the max range of the search, further eliminating future nodes as the search continues. If it is not a leaf node, then one child is added to the stack while the other is searched. This continues until the stack is empty, at which point, the closest hit, the primitive involved, and any other information have been collected and are ready to be used to continue the raytracing process.

4.1 Implementation Plan

To implement this, portions of this algorithm will be created as independent shaders, tested, then integrated in a test environment. After confirming correct behavior, a duplicate of the current raytracing shader will be created and modified to traverse this BVH rather than loop through all primitives to find hit targets.

Compute shaders for constructing Axis-Aligned Bounding Boxes, converting

Algorithm 6: BVHIntersect()

Input: Ray r, HitRecord rec

Output: bool; true if hit, false otherwise

```
/* Presume an tree-embedded array of BVH nodes nodes exists          */
1  uint stack[100];
2  uint toVisitOffset = 0;
3  uint currentNodeIndex = 0;
4  bool hit = false;
5  float min = 0.001;
6  float max = FLOAT_MAX;
7  for ; true; do
8      Node node = nodes[currentNodeIndex];
9      if AABB_ray_intersect(node.bounds, ray) then
10         if node.hasPrimitives then
11             for int i = 0; i < node.numPrimitives; i++ do
12                 if Intersect(primitives[node.primitiveOffset + i], r, rec,
                    min, max) then
13                     hit = true; /* rec and max are updated if hits occur  */
14             if toVisitOffset == 0 then
15                 break;
16             currentNodeIndex = stack[–toVisitOffset];
17         else
18             stack[toVisitOffset++] = currentNodeIndex + 1;
19             currentNodeIndex = node.secondChildOffset;
20     else
21         if toVisitOffset == 0 then
22             break;
23         currentNodeIndex = stack[–toVisitOffset];
24 return hit;
```

Planned Date of Accomplishment	Task
October 20	Morton Code transformation implemented in shader and tested
October 30	Top-level sorting step implemented in shader and tested
November 5	Bottom-level sorting step implemented in shader and tested
November 12	BVH construction on the GPU complete in shaders and tested
November 20	Raytracer traversing BVH and measure performance characteristics
December 1	Present Defense document to Committee
December 4-12	Project Defense to Committee

Table 2: Schedule for Project

Axis-Aligned Bounding Box Barycenter to Morton Codes, sorting Morton Codes, emitting the tree hierarchy and flattening the tree hierarchy will be created. These will be individually tested and then integrated into the render pipeline.

To evaluate success, two factors should be used:

1. Tests validating the construction of a BVH as described in a basic scene containing a few primitives

2. Improved runtimes for high-poly scenes compared to the current raytracer

5 Conclusion

This proposal describes the current work that's been done to create a GPU-accelerated raytracer and proposes an improvement by creating a BVH to traverse for each frame, constructed on the GPU. The improvement should exponentially decrease the runtime cost of raytracing, particularly in scenes with a large amount of input geometry. Parallelizing the construction of the BVH on the GPU should minimize the construction time and only have negative runtime implications for very trivial scenes.

References

[1] Pablo Bauszat, Martin Eisemann, and Marcus Magnor. "The Minimal Bounding Volume Hierarchy". In: *Proceedings of the Vision, Modeling, and Visualization Workshop*. Jan. 2010, pp. 227–234. DOI: 10.2312/PE/VMV/VMV10/227-234.

[2] K. Garanzha and C. Loop. "Fast ray sorting and breadth-first packet traversal for GPU ray tracing". In: *Computer Graphics Forum* 29 (2 2010), pp. 289–298. DOI: 10.1111/j.1467-8659.2009.01598.x.

[3] Xiaozi Guo, Juan Zhang, and Mingquan Zhou. "Fast Parallel Bounding Volume Hierarchy Construction". In: *2020 Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC)*. 2020, pp. 121–124. DOI: 10.1109/IPEC49694.2020.9115145.

[4] J.T. Kajiya. "The Rendering Equation". In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (1986). DOI: 10.1145/15922.15902.

[5] C. Lauterbach et al. "Fast BVH Construction on GPUs". In: *Computer Graphics Forum* 28.2 (2009), pp. 375–384. DOI: 10.1111/j.1467-8659.2009.01377.x. URL: https://wwwx.cs.unc.edu/~geom/papers/documents/articles/2009/lauterbach09.pdf.

[6] Daniel Meister et al. "On Ray Reordering Techniques for Faster GPU Ray Tracing". In: *Symposium on Interactive 3D Graphics and Games*. I3D '20. San Francisco, CA, USA: Association for Computing Machinery, 2020. ISBN: 9781450375894. DOI: 10.1145/3384382.3384534. URL: https://doi.org/10.1145/3384382.3384534.

[7] J. Pantaleoni and D. Luebke. "HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry". In: *Proceedings of the Conference on High Performance Graphics*. HPG '10. Saarbrucken, Germany: Eurographics Association, 2010, pp. 87–95. DOI: 10.5555/1921479.1921493.

[8] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering, fourth edition: From Theory to Implementation*. The MIT Press, 2023.

[9] Peter Shirley, Trevor D Black, and Steve Hollasch. *Ray Tracing in One Weekend*. 4.0.1. 2024.

[10] Peter Shirley, Trevor D Black, and Steve Hollasch. *Ray Tracing: The Next Week*. 4.0.1. 2024.

[11] Peter Shirley, Trevor D Black, and Steve Hollasch. *Ray Tracing: The Rest of Your Life*. 4.0.1. 2024.

[12] Sascha Willems. *Vulkan tutorial on rendering a fullscreen quad without buffers*. 2016. URL: `https://www.saschawillems.de/blog/2016/08/13/vulkan-tutorial-on-rendering-a-fullscreen-quad-without-buffers/` (visited on 06/12/2023).