University of Nebraska at Omaha

Thesis-Equivalent Project Defense

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 [16]. 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 [9]. 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 [18].

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 containing 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 defense 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 before implementing the BVH, then a detailed explanation of how the BVH was implemented and a results section describing the measured improvement over the non-BVH version. Lastly, a conclusion discussing the defense in summary.

2 Related Works

Many works have contributed to the space of raytracing.

Texts such as Shirley, Black and Hollasch [19] [20] [21] as well as Pharr, Jakob, and Humphreys [18] 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 project will utilize.

Meister et al. [15] and Garanzha and Loop [6] 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 [6] inspired the application in BVH construction.

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

Bauszat, Eisemann, and Magnor [2], 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 [7] work on an improvement to the proposed algorithm in this paper, which is based on Pantaleoni and Luebke [16], 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 [16] offer an improvement to Lauterbach et al. [13]

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 [6]. 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 project 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 [19] 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
vec3 oc = $R$.origin - $S$.center; /* $A - C$ */
float a = dot($R$.direction, $R$.direction); /* $b \cdot b$ */
float halfB = dot(oc, $R$.direction); /* $b \cdot (A - C)$ */
float c = dot(oc, oc)- (s.radius * s.radius); /* $(A - C) \cdot (A - C) - r^2$ */
float underRadical = (halfB * halfB) - (a * c);
if $underRadical < 0$ then
    /* if not 0 or positive, no roots */
    return $False$;
float radical = sqrt(underRadical);
float root = (-halfB - radical) / a;
if $root < Min$ or $root > Max$ then
    root = (-halfB + radical) / a;
    if $root < Min$ or $root > Max$ then
        return $False$; /* a hit would occur, but outside hit interval */
/* record hit information in $H$ */
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
vec3 u = $T$.v1 - $T$.v0;
vec3 v = $T$.v2 - $T$.v0; /* Compute triangle direction vectors u and v */
vec3 normal = cross(u, v);
vec3 nNormal = normalize(normal);
float D = dot(nNormal, $T$.v0);
vec3 w = normal / dot(normal, normal);
float denom = dot(nNormal, $R$.direction);
if $|denom| < 0.0001$ then
    return $False$;
float t = (D - dot(nNormal, $R$.origin)) / denom;
if $t < Min$ or $t > Max$ then
    return $False$;
vec3 intersectionPoint = $R$.origin + t * $R$.direction;
vec3 pointOnTrianglePlane = intersectionPoint - $T$.v0;
float a = dot(w, cross(pointOnTrianglePlane, v));
float b = dot(w, cross(u, pointOnTrianglePlane));
if $a < 0$ or $b < 0$ or $a + b > 1$ then
    return $False$;
/* record hit information in $H$ */
return $True$;
```

This triangle intersection algorithm [20] 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
HitRecord rec;
vec3 color = vec3(0);
vec3 globalAttenuation = vec3(1);
vec3 unitDir = normalize($R$.direction);
Ray curr = Ray($R$.origin, unitDir);
for uint i = 0; i < MAXRAYTRACEDEPTH; i++ do
    if !sceneHit(curr, rec) then
        color = color + (BACKGROUNDCOLOR * globalAttentuation);
        break;
    vec3 attenuation;
    vec3 emmitedColor = emmitted(rec);
    color = color + (emmittedColor * globalAttenuation);
    bool scattered = scatter(curr, rec, attenuation, curr);
    /* scatter() sets attenuation and curr to new values */
    globalAttenuation *= attenuation;
    if !scattered then
        break;
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. [24]

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

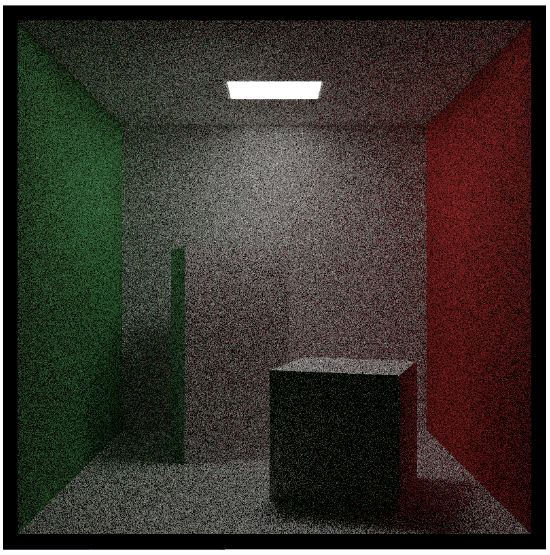
Figure 1: GPU Raytraced

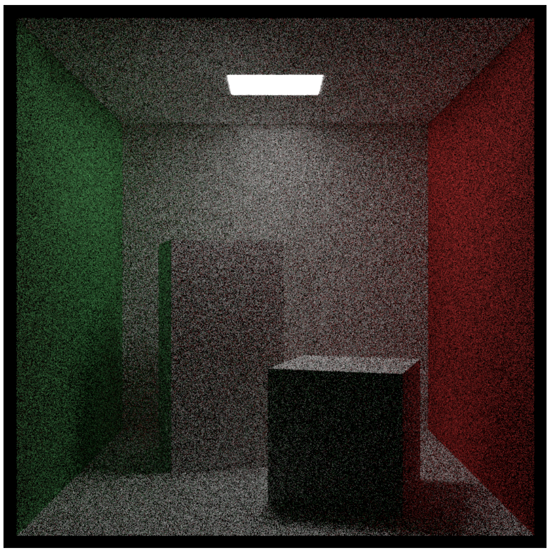
Figure 2: CPU Raytraced

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 BVH Implementation

To implement this BVH on the GPU, a sequence of GLSL shaders were used. The pipeline worked as follows:

1. Transform primitives from model space to world space

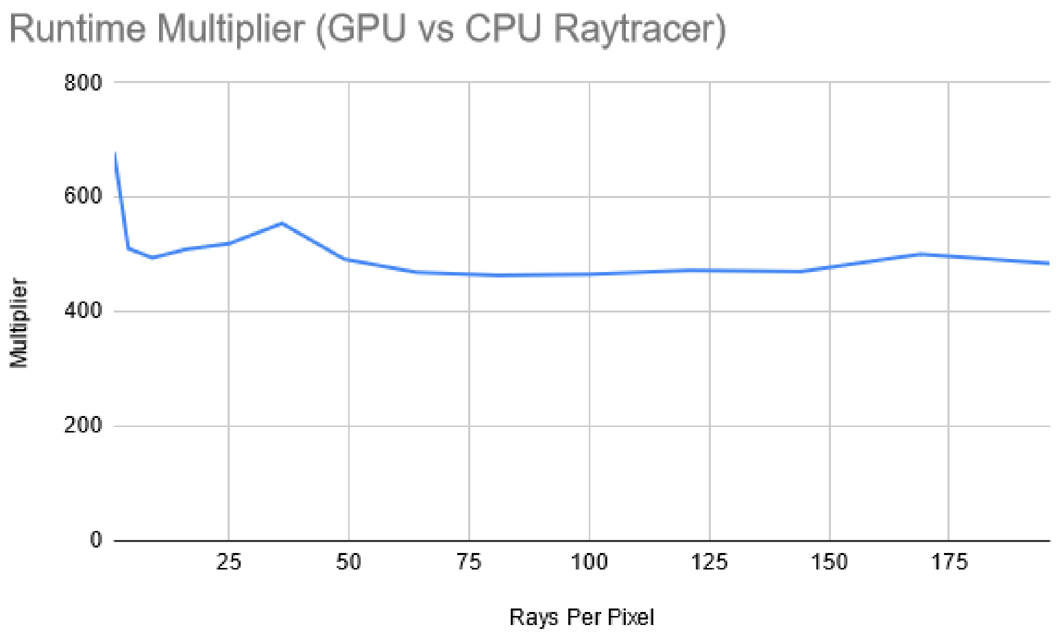

Figure 3: Number of Multiples faster the GPU renders the Cornell Box compared to the CPU with varying rays per pixel

2. Find an enclosing Axis-aligned Bounding Box (AABB) for all primitive centers
3. Quantize and convert centers into Morton Codes
4. Sort the Morton Codes
5. Construct the structure of the tree using the Morton Codes
6. Build AABBs for each tree node
7. Raytrace with BVH

All but the first step was developed as part of this project.

4.1 Enclosing AABB

Finding an Axis-aligned bounding box that contains all centers is needed as the morton codes, which provide the ordering for primitives in a 3D space, as based on locations within an encapsulating space. This functionality is effectively a multi-threaded reduction function.

All primitives (triangles and spheres), a count of primitives, a shader storage buffer object with space for the enclosing AABB output, an a shared int used for atomic operations are provided to the shader. It functions but first having a single thread set initial values for the bounding box and setting the atomic int to 0. Then 256 threads across 4 to 8 Streaming Processors each find a primitive, get it's center, and update individually held $localMin$ and $localMax$ values. To find an element, a thread uses an invocation id, 0-255 in this case. Progressing through the list of primitives just involves incrementing the 0-255 value by 256 (the size of the workgroup).

Algorithm 5: Example of Traversing a large list with fewer threads

```
#define WORKGROUPSIZE 256;
layout(local_size_x = WORKGROUPSIZE) in;
/* ...                                                    */
vec3 localMin;
vec3 localMax;
for uint elemIndex = gl_localInvocationID; elemIndex < primitiveCount;
  elemIndex += WORKGROUPSIZE do
    vec3 center;
    if elemIndex < ubo.numTriangles then
        center = getTriangleCenter(triangles[elemIndex]);
    else
        center = spheres[elemIndex - ubo.numTriangles].center.xyz;
    localMin = min(center, localMin);
    localMax = max(center, localMax);
```

With 256 $localMin$ and $localMax$ values aggregated, they must be combined. They are first combined by subgroup using $subgroupMin()$ and $subgroupMax()$, which collects all the values for a variable in a subgroup and performs an operation on them, returning to result to all threads within the subgroup. This reduces the values left to combine from 256 to 8 or 4 (depending on video card vendor, 8 for NVIDIA (32 threads per streaming process), 4 for AMD (64 threads per streaming process)).

Finally, a single thread from each subgroup waits on the atomic int until the subgroup's ID is the same as the atomic int. If it is, it performs the min and max operation using it's $localMin$ and $localMax$ values. It then increments the atomic int.

There are many other ways to implement reduction-like operations on GPUs. Another which was considered involved storing aggregation values into shared arrays, and making a single subgroup handle aggregation. This technique will be demonstrated with $localIndexOffsets$ and $globalIndexOffsets$ in Section 4.3.

4.2 Generate Morton Codes

Morton Codes are critical to this project as they provide an ordering to elements in 3D space and correlate to Octant coordinates for each set of 3 bits (this project uses 30). This creates a useful method for grouping primitives by their locality, which is important when considering BVH construction.

For spheres, the centers are simple, but triangles offer many types of centers than can be used. In Pantaleoni and Luebke [16], a barycenter is used for triangle centers. In this project, the centroid of the triangle is used (a simple average of the vertices).

To create a morton code from a primitive center, we must:

1. quantize the center into a vector of integers

2. formulate integers into a single integer such that their bits are interleaved

Algorithm 6: quantizeForMorton()

```
Input: vec3 coord
Output: uvec3
1 vec3 locationWithin = coord - enclosingAABB.min;
2 vec3 span = enclosingAABB.max - enclosingAABB.min;
3 vec3 offset = coord / span;
  /* 0.0 to 1.0 scale for each dimension */
4 return uvec3(offset * (1 « MORTON_BITS));
```

To quantize the center, the enclosing AABB is used. We can create a floating-point value between 0 and 1 to represent where the primitive exists within the enclosing AABB. 0 means the center is very close to the minimum of the AABB, and 1 means the center is very close to the maximum of the AABB. This floating point value can then be scaled based on the number of bits being used to generate the Morton codes. More bits results in more fine grained location information.

Algorithm 7: separateBitsBy3()

```
Input: uint val
Output: uint
1 val = (val | (val « 16)) & 50331903;
/* moves bit 9 and 8 to last byte.  masks off the rest */
2 val = (val | (val « 8)) & 50393103;
/* moves bits 4-7 to upper portion of second byte.  masks off the rest
   */
3 val = (val | (val « 4)) & 51130563;
/* moves bits 6, 7, 2, and 3 the to the next nibble.  masks off the
   rest.  this leaves 5 pairs of adjacent bits */
4 val = (val | (val « 2)) & 153391689;
/* separates each pair by moving the first of the two forward 2 spaces.
   Masks off the rest.  Due to earlier spacing, all bits now have to 0's
   between them and the next bit. */
5 return val;
```

With the center quantized, interleaving the bits remains. In this project, bit shifts and masks were used to separate the bits such that 2 "holes" or empty values existed between each. This allows for one set of bits to be interleaved with the others by simple shifts.

With the Morton codes created, the respective primitive index and primitive type (triangle or sphere) is also sorted to correlate the Morton codes after they get sorted in the next section.

Algorithm 8: mortonCode3D()

```
Input: uvec3 quantizedCoord
Output: uint
1 return ( separateBitsBy3(quantizedCoord.x) « 2 |
    separateBitsBy3(quantizedCoord.y) « 1 |
    separateBitsBy3(quantizedCoord.x) );
```

4.3 Sort Morton Codes

There are many publications about sorting algorithms on the GPU. Due to SIMD environments' ability to essentially perform some O(n) tasks in constant time but not with the same write guarentees of a single threaded environment, different algorithms are the fastest on the GPU. Noted in both GPU Gems 2 [17] as well as Pantaleoni and Luebke [16], odds-evens sort is an easy-to-understand example of how the SIMD environment affects sorting algorithm comparison. Archipov, et al. [1] explores GPU sorts in a survey and finds bucketization sorts, particularly Radix Sort, to be particularly effective, alongside many other characteristics. Others, such as Maltenberger, et al. [14] have even focused on multi-GPU sorting algorithms, sorting 1,000,000,000 elements in 45ms. Sorting on the GPU is a well documented and developed space. The implementation in this paper is uncomparably simple compared to the papers mentioned above, but operates at good enough speeds for the task at hand (see, Section 5).

For this paper, LSB Radix sort was implemented (4 iterations w/ 8 bits each iteration) using a single workgroup. A workgroup of size 256 and two shader storage buffer objects are used. To sort, several steps must be completed:

1. enumerate the number of elements that fall within each bin
2. figure out at what index each bin starts in a resulting array
3. traverse all primitives and relocate them correctly while maintaining order from previous iterations

For the first step, an array of integers, one per bin, is initialized. Each element is examined and their bin collected using a bitmask. The count at the bin identified, which acts as an index, is used to increment the count atomically.

The second step makes used of some more complex subgroup operations, particularly exclusive scan. As an algorithm, exclusive scan takes an array of summable elements, and returns a progressive sum of elements such that the resulting index is the sum of all previous indices. This is especially useful in finding the index where a given bin should begin in the result of LSB radix sort. However, parallelizing this requires a reduction similar to Section 4.1.

Algorithm 9: Subgroup Reduction And Subgroup Exclusive Scan

```
1 if $gl\_LocalInvocationID.x < BINS$ then
      /* histogram contains the number of elements that need to go in each
         bin                                                              */
2     uint histogramCount = histogram[gl_LocalInvocationID.x];
3     uint sum = subgroupAdd(histogramCount);
4     uint prefixSum = subgroupExclusiveAdd(histogramCount);
5     localIndexOffsets[gl_LocalInvocationID.x] = prefixSum;
      /* subgroupElect() returns true for only 1 thread within the
         subgroup.  All others return false                               */
6     if $subgroupElect()$ then
7         subgroupReductions[gl_SubgroupID] = sum;
```

GLSL has a function *subgroupExclusiveAdd* which creates the input array for exclusive scan using the values inside sequential threads of a single streaming processor (32 Nvidia, 64 AMD). This means thread 0 gets back 0, thread 1 gets back thread 0's count, thread 2 gets back the sum of thread 0's count and thread 1's count, etc.

After doing subgroup level reduction, results from the sum and exclusive scan are stored in two arrays. A single subgroup then handles going through each bin and collecting the subgroup reductions into the true sums, yielding the index offsets for each bin in the soon to be sorted array.

Lastly, to transfer each element over to the resulting array correctly, an order must be created to identify where within the bin an element should be written. To solve this problem, an array of arrays of unsigned integers. The outer array, provides a set of integers to store bit flags to each bin. The inner arrays store

Algorithm 10: Subgroup Reduction And Subgroup Exclusive Scan

```
1 if subGroupID == 0 then
2     uint indexOffset = 0;
3     for uint i = gl_SubgroupInvocationID; i < BINS; i += SUBGROUP_SIZE do
4         globalIndexOffsets[i] = indexOffset + localIndexOffsets[i];
5         indexOffset += subgroupReductions[i / SUBGROUP_SIZE];
```

enough integers to contain a boolean value (1 bit) for each thread in the work group. Morton Codes (and primitive index information) are written to their sorted location in batches of 256 (the work group size).

If a Morton code should be put into a particular bin, then the bin is used to identify the index of out outer array and it's work group invocation number is used to identify and set the bit in the inner array. After all 256 have done so, the number of bitCount of the respective inner array until the thread's respective boolean identifies the proper index for it to write. The last element to write in a bin adds the number of elements just written to the bin to the index offsets identifying bin start locations, allowing for another group of 256 to be written.

This process represents 1 iteration of radix sort, which operates, in this implementation, on 8 bits at a time. After 4 iterations, all Morton codes are sorted.

4.4 Tree Construction

A common system for tree construction involved building the tree during a frame, and updating it in the subsequent frames until a heuristic evaluates that the tree isn't worth the time to modify and is reconstructed. Bittner, et al. [3] explores optimizations in the space of maintaining trees, however, in this project, the implementation will follow the ideas of Karras [11] which follow the works of Lauterbach et al. [13] and Pantaleoni and Luebke [16] in reconstructing the tree every frame.

There are many aspects to BVH construction. From the research reviewed, it appears that three in particular, memory footprint, construction speed, and

Algorithm 11: Bin Indexing Flags And Finding Inter-Bin Offset

```
const uint indexFlagBin = gl_LocalInvocationID.x / BITS;
const uint indexFlagBit = 1 « (gl_LocalInvocationID.x % BITS);
const uint indexFlagBitMask = indexFlagBit - 1;
for uint blockID = 0; blockID < primitiveCount; blockID += WORKGROUP_SIZE do
    barrier();
    const uint ID = blockID + workGroupInvoID; MortonPrimitive elem;
     uint binID = 0; uint binOffset = 0;
    if ID < primitiveCount then
        elem = GET_INPUT_ELEMENT(ID, iteration);
        binID = uint(elem.code » shift) & bitMask;
        binOffset = globalIndexOffsets[binID];
        atomicAdd(binIndexingFlags[binID].locationInfo[indexFlagBin],
         indexFlagBit);
    barrier();
    if ID < primitiveCount then
        uint interBinOffset = 0; uint count = 0;
        for uint i = 0; i < WORKGROUP_SIZE / BITS; i++ do
            const uint bits = binIndexingFlags[binID].locationInfo[i];
            const uint fullCount = bitCount(bits);
            const uint partialCount = bitCount(bits & indexFlagBitMask);
            interBinOffset += (i < indexFlagBin) ? fullCount : 0U;
            interBinOffset += (i == indexFlagBin) ? partialCount : 0U;
            count += fullCount;
            /* count is used to identify which thread is the last in a bin
               */
        /* binOffset + interBinOffset is the index to write to          */
```

traversal speed, are the most important factors to consider when implementing a BVH. BVH construction can result in a highly optimal tree which can be traversed incredibly fast, but often this construction becomes an expense all it's own in real-time settings. A recent thesis by Athos van Kralingen [12], found that the optimal BVH construction, as of now, is scene dependent. Surface Area Heuristic (SAH) is often used as a gold standard for comparing BVH traversal speeds. Several heuristics reviewed by Kralingen including SAH, the Scene-Interior Ray Origin Metric [5], Preferred Ray Distributions [8], Ray Distribution Heuristic [4], and Occlusion Surface Area Heuristic (OSAH) [22] were evaluated. Preferred Ray Distributions, the Ray Distribution Heuristic, and OSAH were found to construct the fastest BVH depending on the scene in Kralingen's research. SAH has been used for real-time settings [23], but is generally considered to have a slow construction speed [16] [13]. Unfortunately many of those methods are far from feasible in a real-time setting, making them difficult to use in this project. SAH alone can easily take 400ms to construct a BVH in large scenes [16]. If the target framerate of just 30 fps, that only leaves 33.3ms per frame. So quicker construction methods with good-not-great traversal speeds need to be examined.

Now that the Morton codes have been sorted, The BVH structure can be created but it's split into two parts. The first is described here and creates the nodes and sets all the pointers such that each primitive is contained within a node and exists only once. The nodes are stored in an array and arranged such that $n - 1$ internal nodes are followed by n leaf nodes, which don't contain child nodes but instead primitive pointers. The total number of nodes in the BVH data structure is $n(n - 1)$. The root node is the 0th node in the array. Nodes contain an $AABB$, a $leftChild$ and $rightChild$, a *primitiveIndex* and a *primitiveType*. Leaf nodes use the *primitiveIndex* and *primitiveType* while internal nodes use the $AABB$, $leftChild$, and *rightChild*. $leftChild$ and *rightChild* are indices to the array containing the nodes.

The first step involves constructing all of the leaf nodes. Each primitive has an AABB constructed. The node index is $n - 1 + primitiveIndex$. The AABB is set, the $leftChild$ and $rightChild$ are set to 0 (representing an invalid index), and the primitive index and type are transferred. AABBs of internal nodes are constructed in a different shader described in Section 4.5. This is because we can construct all of the internal nodes at once, but doing so makes getting accurate AABB information impossible, as child nodes may not yet be available in memory.

To construct all of the internal nodes at once, a pair of useful algorithms are used. The first, Function 13, determines creates a range across the Morton code array. This range represents the set of primitives that will be a descendant by this node. The second, Function 14, determines where, within that range, to split it in two. The $leftChild$ will handle one portion while the $rightChild$ will handle the rest.

With this information, the nodes can be formed. If the split identified is the first element in the range, the $leftChild$ is only handling one primitive, so the $leftChild$ is a leaf node and the node index is $n - 1 + \mathit{split}$. Similarly, if the split identified is one less than the last element, then the $rightChild$ only has one primitive to handle, so the $rightChild$ is a leaf node and the node index is $n-1+\mathit{split}+1$. Otherwise the index to the next internal node is split and $\mathit{split}+1$ respectively.

Algorithm 12: countLeadingZeroesFromDifference()

```
Input: int i, int j
Output: int
1 if either index is out of bounds of primitive array then
2     return -1;
3 uint code1 = mortonPrimitives[i].code;
4 uint code2 = mortonPrimitives[j].code;
5 if code1 == code2 then
6     return 32 + 31 - findMSB(i ⊕ j);
7 return 31 - findMSB(code1 ⊕ code2);
```

Before taking a closer look at Function 13 and Function 14, Function 12 is used in them and must be explained. This function finds how many zeroes exist before the first 1 in the difference between two Morton codes. This is useful as the more zeroes, the more that the start of the Morton codes are identical, meaning the closer the two objects are in space. For duplicate Morton codes, the indices in the Morton code array are used. The result from that case is increased by 32, representing the Morton codes were identical while still giving unique results for a sequence of duplicate Morton codes.

Algorithm 13: determineRange()

```
Input: int id
Output: int lower, int upper
1  const int deltaL = countLeadingZeroesFromDifference(id, id - 1);
2  const int deltaR = countLeadingZeroesFromDifference(id, id + 1);
3  const int dir = (deltaR >= deltaL) ? 1 : -1;
4  const int deltaMin = min(deltaL, deltaR);
   /* pick the one most different                                  */
5  int iMax = 2;
   /* upper bound                                                   */
6  for ; countLeadingZeroesFromDifference(id, id + iMax * dir) > deltaMin;
     do
7      iMax «= 1;
8  int i = 0;
   /* back the other way to get actual value                        */
9  for int t = iMax » 1; t > 0; t »= 1 do
10     if countLeadingZeroesFromDifference(id, id + (i + t) * dir) >
        deltaMin then
11         i += t;
12 int endId = id + i * dir;
13 lower = min(id, endId);
14 upper = max(id, endId);
```

Function 13 takes the id of the current thread for the global workgroup and outputs a *lower* and *upper*, representing the start and end of the range. To set *lower* and *upper*, Function 12 is used on the current Morton code and it's neighbors. For the range, the current id will be used for one of either *lower* or *upper*. The goal now is to find the other end of the range. Specifically, the range is such that all codes within have an equivalent or greater number of zeroes from Function 12. This range represents all the Morton codes in the scene that the same prefix of bits. To do this, the upper bound is first over estimated and then corrected. To find the upper bound, an increasing range of codes are evaluated until they become too different. Then bit shifting is used to find the exact index where before the difference becomes too large. Using bit shifts causes the algorithm to make jumps as it searches for the right index, effectively performing binary search.

Algorithm 14: findSplit()

```
Input: int first, int last
Output: int
1  int commonPrefix = countLeadingZeroesFromDifference(first, last);
2  int split = first;
3  int stride = last - first;
4  do
5      stride = stride + 1 » 1;
6      int newSplit = split + stride;
7      if newSplit < last then
8          int splitPrefix = countLeadingZeroesFromDifference(first,
            newSplit);
9          if splitPrefix > commonPrefix then
10             split = newSplit;
11 while stride > 1;
12 return split;
```

Function 14 takes the resulting range from Function 13 and outputs an index within that range to represent where the split should occur. The goal is to find a balanced point to split the range based bit prefix difference. If the range is 75% of very close together primitive (really big prefix) towards the start, then the split will occur after them. If they're towards the end, then before. This results in primitives that are further away from other primitives appearing closer to the top of the tree. If a ray is going to hit a dense patch of primitives, then a larger number of internal nodes are needed to deal with that (as this is a binary tree). To achieve this, binary search is used once more. A *stride* is defined as the distance from the end to the start of the range and the initial *split* value is set to the start of the range. *stride* is divided in half and a proposed *newSplit* is created. If that split is within the range and has a more similar prefix than the start and end of the range, it is adopted as the new split location. Equality does not update the split location.

With these functions, the internal nodes can be created all at once. However, their AABBs cannot. This is because the AABB of an internal node is determined by the minimums and maximums of their children's AABBs. In creating all internal nodes at once, there is no guarantee that a child node has set their AABB yet. To solve this, another shader is used, see Section 4.5. That shader does need some information to build the AABBs bottom-up, so those structures are set during the internal node construction process. They essentially create the information necessary to go from the bottom to the top of the BVH (as nodes lack a *parent* variable as a member, but it also contains a synchronization primitive that will be used later.

4.5 Building AABBs for Tree Nodes

The BVH that has been constructed still lacks properly defined AABBs for each node. Individual nodes lack parent members, making bottom-up traversal difficult. However, in Section 4.4, some information was stored in an array to allow bottom-up traversal. These structs contain a *parent* member and a *visitationCount* and are stored at the index of each child node.

The AABB of a given node is defined by the minimums and maximums of it's child nodes. In other words, the parent AABB must contain all child AABBs. To achieve this, 1 thread starts at each leaf node (which contain primitives and have defined AABBs). They then use the struct info to find the parent and increment, atomically, the visitation count. If this thread was first to arrive, the other child node may not be fully computed yet, so this thread dies, as it won't be needed anymore. When the second thread arrives, both children now have properly defined AABBs, so they are combined to create the parent AABB. This continues upward till 1 last thread arrives at the root for it's second visitation. After setting the AABB, it returns and the BVH is complete.

Algorithm 15: buildAABBsForInternalNodes()

```
const int primitiveCount = int(numTriangles + numSpheres);
const int leafOffset = primitiveCount - 1;
if gl_GlobalInvocationID.x >= primitiveCount then
    return;
uint nodeId = constructionInfo[leafOffset +
  gl_GlobalInvocationID.x].parent;
for ;true; do
    int visitations = atomicAdd(constructionInfo[nodeId].visitationCount,
     1);
    if visitations < 1 then
        return;
    HLBVHNode node = nodes[nodeId];
    HLBVHNode left = nodes[node.leftIndex];
    HLBVHNode right = nodes[node.rightIndex];
    node.aabb = combineAABB(left.aabb, right.aabb);
    nodes[nodeId] = node;
    if nodeId == 0 then
        return;
    nodeId = constructionInfo[nodeId].parent;
```

4.6 Raytracing with a BVH

For traversal, a stack and a few values can be used to keep track of where to look next. This algorithm is described in Function 16. 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.

Algorithm 16: hitBVH()

```
Input: Ray r, HitRecord rec, float min, float max
Output: bool; true if hit, false otherwise
uint stack[128];
uint toVisitOffset = 0;
uint currentNodeIndex = 0;
bool hit = false;
float closest = max; for ; true; do
    Node node = nodes[currentNodeIndex];
    if AABBhitCheck(node.aabb, ray) then
        if node.isLeafNode then
            if node.primitiveType == SPHERE_PRIMITIVE then
                /* perform sphere intersection and update closest and hit,
                   if hit                                              */
            else
                /* perform triangle intersection and update closest and
                   hit, if hit                                         */
            if toVisitOffset == 0 then
                break;
            currentNodeIndex = stack[–toVisitOffset];
        else
            stack[toVisitOffset++] = currentNodeIndex + 1;
            currentNodeIndex = node.secondChildOffset;
    else
        if toVisitOffset == 0 then
            break;
        currentNodeIndex = stack[–toVisitOffset];
return hit;
```

4.7 Challenges

There were many challenges over the course of this project. Perhaps one of the most prevalent was my lack of experience in highly-parallelized environments and lack of familiarity with the tools available. This project has helped alleviate those issues, hopefully so that future projects will go more smoothly, but learning about the in-shader synchronization tools such as $barrier()$, $memoryBarrier()$, and subgroup reduction tools such as $subgroupMin$ resulting in some re-writes in the implementation process.

A notable case of this was in finding the enclosing AABB. Initially, AABBs were constructed on all primitives, then transferred back to the CPU, where the reduction was done, then flushed to a uniform buffer and returned to the GPU. After learning of the reduction capabilities available, this was changed.

The initial plan involved constructing AABBs for all primitives first, but this became somewhat pointless due to the BVH construction. Rather than carry an additional shader filled with additional pointer redirection via index offsets, the implementation was able to avoid constructing AABBs until the Tree construction, where AABBs are stored in leaf nodes.

A notable bug was one where regions of triangles failed to render; not entire triangles, but triangular sub regions of triangles. This was caused by an inaccurate AABB construction algorithm for triangles. An initial implementation used the first vertex of the triangle and a created point $v0 + (v1 - v0) + (v2 - v0)$. This can form an AABB, but it formed them entirely too large, and in some cases, malformed as to cut off portions of the triangle. It was resolved by using the minimums and maximums of each coordinate.

Another notable bug involved putting too many uses upon a variable. When evaluating all primitives, $gl\_GlobalInvocationID$ comes an great tool for indexing into the array of primitives in a highly parallelized way. However, primitives are actually stored in 2 shaders storage buffers, one for triangles and one for spheres. This led to many cases where the index would need to be redefined. If the index was below the number of triangles, then it identified a triangle. But if it was above that but below the total number of primitives, then it identified a sphere via $index - numberOfTriangles$. When generating Morton primitives, this index was over used to identify both the primitive index and where the Morton primitive just created should be stored. This resulted in incoherent memory interactions, as two or more threads would try to write data to the same places. This was easily resolved.

5 Results

The benefit of a BVH is purely in frame generation time and has no visual impact. This makes a comparative methodology easy as timing is all that's needed. In particular, the overall frame generation time, the raytracing time, and the BVH build time are of interest. In these, the introduction of a BVH have been profoundly beneficial.

Scene	Poly-count	CPU	GPU	GPU + BVH
Cornell Box	37	621.186	1.287	1.865
Blender Suzanne	507	N/A	2.234	0.4769
Stanford Bunny	69,458	N/A	>60[1]	12.132

Table 2: Runtime Comparison of Raytracers across scenes (seconds, 32 rays per pixel, depth 16)

A binary tree BVH takes the average ray-scene hit detection from evaluating all primitives, n, to evaluating as few as $log2(n)$.

For extremely simple scenes (<200 primitives), the inclusion of a BVH can be detrimental, as scene with the cornell box example. This is due to increases in execution divergence (evaluating a for loop in parallel vs traversing a tree) and the need to interact with internal nodes. However, poly counts for video

[1] >60 means a single frame failed to be rendered in a minute

games have been easily above 1,000 and often in the millions for some time. Some technologies like Unreal Engine's Nanite are able to manage billions of triangles (though rendering significantly less) [10].

Scene	Poly-count	BVH construction time
Cornell Box	37	2.625 milliseconds
Blender Suzanne	507	2.754 milliseconds
Stanford Bunny	69,458	16.387 milliseconds

Table 3: BVH construction time comparison across scenes

For those scenes with a higher poly-count (>200 primitives), a BVH comes almost essential to scale with the number of triangles. This is show cased in the case of the Stanford Bunny in Table 2. As poly counts increase, runtime approximately logarithmically increase with a BVH. But without, it linearly increase.

Comparing build times in Table 3, for the raytracing time gained, these build times are great. There's likely a bottle neck in the pipeline of shaders causing the BVH construction time to increase in the case of the Stanford Bunny. While the shaders generating the morton codes, constructing the BVH, and construct BVH bounding boxes are setup to use as big a work group as they need, the radix sort and enclosing AABB shaders only operate with a single work group.

6 Conclusion

In implementing a BVH on the GPU, the GPU based raytracer was vastly improved quantitatively and algorithmically such that it yields superior frame generation times than previously. However, it's still a far cry from being real-time viable. Fortunately, many aspects of the raytracer, including the current BVH, can be improved to likely yield better performance. These include more accurate techniques for estimating the rendering equation, such as adding a BRDF and doing importance or multiple importance sampling, all of which will require few rays to be fired to get improved color quality. Others include better randomization techniques and better ray generation and pipeline management. Currently 1 ray is fired from every pixel at a time and the shader doesn't begin the next round until all have finished their path. This can lead to varying degrees of GPU utilization. In that same vein, the pipeline currently doesn't use a swapchain to it's full extent, which can also yield performance gains. There are many improvements still to be made to bring this project closer to real-time application.

References

[1] Dmitri I. Arkhipov et al. *Sorting with GPUs: A Survey*. 2017. arXiv: `1709.02520 [cs.DC]`. URL: `https://arxiv.org/abs/1709.02520`.

[2] 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`.

[3] Jiří Bittner, Michal Hapala, and Vlastimil Havran. "Fast Insertion-Based Optimization of Bounding Volume Hierarchies". In: *Computer Graphics Forum* 32 (1 2013), pp. 85–100. DOI: `10.1111/cgf.12000`.

[4] Jiří Bittner and Vlastimil Havran. "RDH: ray distribution heuristics for construction of spatial data structures". In: *Proceedings of the 25th Spring Conference on Computer Graphics*. SCCG '09. Budmerice, Slovakia: Association for Computing Machinery, 2009, pp. 51–58. ISBN: 9781450307697. DOI: `10.1145/1980462.1980475`. URL: `https://doi.org/10.1145/1980462.1980475`.

[5] Bartosz Fabianowski, Colin Fowler, and John Dingliana. "A Cost Metric for Scene-Interior Ray Origins". In: *Eurographics 2009 - Short Papers*. Ed. by P. Alliez and M. Magnor. The Eurographics Association, 2009. DOI: `10.2312/egs.20091046`.

[6] 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`.

[7] 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.

[8] Vlastimil Havran and Jiří Bittner. “Rectilinear BSP Trees For Preferred Ray Sets”. In: 2001. URL: https://api.semanticscholar.org/CorpusID:18251601.

[9] 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.

[10] Brian Karis and Jerome Platteaux. *Unreal Engine 5 Revealed! | Next-Gen Real-Time Demo Running on PlayStation 5*. Unreal. 2020. URL: https://www.youtube.com/watch?v=qC5KtatMcUw.

[11] Tero Karras. “Maximizing parallelism in the construction of BVHs, octrees, and k-d trees”. In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG’12. Paris, France: Eurographics Association, 2012, pp. 33–37. ISBN: 9783905674415.

[12] Athos van Kralingen. “Assessing Alternatives to the Surface Area Heuristic for Bounding Volume Hierarchy Construction”. MA thesis. Utrecht University, July 2023. DOI: 20.500.12932/46026.

[13] 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.

[14] Tobias Maltenberger et al. “Evaluating Multi-GPU Sorting with Modern Interconnects”. In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD ’22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 1795–1809. ISBN: 9781450392495. DOI: 10.1145/3514221.3517842. URL: https://doi.org/10.1145/3514221.3517842.

[15] 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.

[16] 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.

[17] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005. ISBN: 0321335597. DOI: 10.5555/1062395.

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

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

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

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

[22] Marek Vinkler, Vlastimil Havran, and Jiří Sochor. "Visibility driven BVH build up algorithm for ray tracing". In: *Computers Graphics* 36.4 (2012). Applications of Geometry Processing, pp. 283–296. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2012.02.013. URL: https://www.sciencedirect.com/science/article/pii/S0097849312000362.

[23] Ingo Wald. "On fast Construction of SAH-based Bounding Volume Hierarchies". In: *2007 IEEE Symposium on Interactive Ray Tracing*. 2007, pp. 33–40. DOI: 10.1109/RT.2007.4342588.

[24] 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).