Simple Path Tracer with BVH Acceleration Structure

Ludwig Boge, Jonatan Ebenholm
Linköping University

May 28, 2026

Contents

1 Introduction

For as long as computers have been able to render images, methods to render images both faster and better have been developed and studied. Whitted introduced his variant of raytracing in 1980 that gave nearly photorealistic results for transmissive and reflective surfaces. Whitted's results took at times nearly 2 hours to produce, but with modern technology those very same images would most likely be rendered in a millisecond. Today we are therefore able to take advantage of more expensive methods to produce more photorealistic images and still have it done in real-time. Path tracing is one such method, where many rays bounce multiple times in the scene, just like a light ray would in real life. Path tracing is an expensive method which even modern technology can struggle with if the algorithm is not optimised. In this project we will therefore try to create a path tracer, on the GPU, which we want to get as close to real-time rendering as possible.

1.1 Background

The path tracing algorithm achieves realistic lighting by solving the rendering equation as presented by Kajiya[7] in his 1986 paper. The rendering equation is given as an integral over incoming light directions towards a surface point to be shaded. The computer cannot solve the integral analytically and therefore has to approximate it. This can be done by using Monte Carlo integration which in practice calculates a number of random samples of the integral and averages them to achieve an approximation. The result of the approximation will converge to the analytical result of the integral as more and more samples are used. This numerical scheme will lie as foundation of our implementation and will be used to calculate bounced lighting, and direct illumination to achieve soft shadows.

1.2 Project Specifications

- To implement a path tracing algorithm in the fragment shader.
- Allow for diffuse, specular, glossy, and transmissive materials.
- Implement a bounding volume hierarchy to drastically increase performance.
- Create a UI, letting the user load and move objects in the scene.

2 Implementation

An overview of the different steps of implementing the project

2.1 Program Structure

The program was written in C++ and several libraries were used. GLFW was used to handle the application window and the OpenGL context, GLAD was used to create function pointers for the OpenGL functions and vector4Utils was used for all linear algebra math. In addition, DearImGui was used for implementing a basic UI but was not necessary for the application to run. Most of the program functionality was implemented in a *Application* class. This included window creation, rendering and buffer binding functions, and all other necessary functionality to create a 3D rendering application. Other classes were also created for handling different parts of the program, the major ones were the classes: *Camera*, *Shader*, *Scene*, *OBJLoader*, and *BVHTree*. The most important part of the program was the fragment shaders, that is where the path tracing algorithm was implemented.

2.2 Sending Geometry to the Shader

When an object has a lot of data, such as the Stanford bunny with 70000 triangles, a proper method needs to be used in sending over the data to the shader program as to be as efficient as possible. For this purpose we could use Uniform Buffer Objects[3] (UBO), which is a buffer object that stores uniform data. UBOs can store upwards of 16kB worth of data, which for our purposes really is not enough. There is however another type of object buffer which has been core to OpenGL since 4.3, namely the Shader Storage Object Buffer[2] (SSBO). The SSBO guarantees a storage of 128MB, but as the documentation explains, most implementations allows the SSBO to be as large as the GPUs memory limit itself. There are currently three more interesting SSBOs that we send to the GPU, namely the Primitive objects, the BVH nodes which will be explained further in 2.6, and the indices that the BVH-tree needs to find the primitives in the primitive SSBO. The struct for a primitive in our implementation is shown in 1. The variable names should make sense for a triangle, but for a parametrized sphere variables such as normals and edges, as well as vertices make little sense. We do still want to use the same struct for all objects. For this purpose we simply defined that for a sphere primitive *vertex1* is the center point of the sphere, and the x-value of *vertex2* is the radius of the sphere. This means that for the rendering of a sphere, there will be alot of empty data. This is still a nice pay off though since if we were to model a sphere using triangles we would instead send in a whole lot of triangles that make up a sphere, which would definitely cost us more memory anyway. Another example of a struct is the BVHNode shown in 2. It is worth showing since it shows a quirk of SSBOs which is that it packs the data in sets of 16 bytes at a time. Since a vec3 is 12 bytes and an int is 4 bytes, imagine what

would happen if the pad1 variable was not included at the end of the BVHNode, we'd end up with an 44 byte large struct, which means that when we send the data over the first 48 bytes of data is sent in to the first BVHNode, eliminating the x-value of the bBoxMin variable. This cascades and at some point all the data gets jumbled up and impossible to manage. To help remove this issue, you could do what we did and add padding to the struct.

```
struct Primitive{
    vec3 vertex1;
    int ID; // 0 == Triangle, 1 == Sphere
    vec3 vertex2;
    float smoothness; //Odds that the ray would bounce off of the surface.
    vec3 vertex3;
    int materialType;
    vec3 color;
    float ior;
    vec3 normal;
    float bounceOdds;
    vec3 edge1;
    vec3 edge2;
};
```

Figure 1: The Primitive struct.

```
struct BVHNode {
    vec3 bBoxMin;
    int leftChild;
    vec3 bBoxMax;
    int rightChild;
    int startTriangle;
    int triangleCount;
    int escapeIndex;
    int pad1;
};
```

Figure 2: The BVHNode struct.

2.3 Tracing Rays

The path tracing algorithm implemented in this project contained multiple steps. The outline was to shoot multiple rays through each pixel in the camera plane, check for ray-surface intersection with the scene, and determine whether and how to bounce the ray at the surface into the rest of the scene. A ray was represented by three variables in our implementation, *direction*, *startPoint* and *endPoint*.

2.3.1 Camera Rays

The first step was to shoot many rays through each pixel on the camera plane. This was done by first calculating the dimensions of the camera plane based on the given field of view and aspect ratio, then using that and the current fragment position to calculate the position u,v in the camera plane that the camera ray was to go through[1]. The direction of the ray was then decided by multiplying the u-coordinate with the right camera vector and multiplying the v-coordinate with the up vector in order to take into account where the camera is looking. This code was implemented in the function *generateCameraRay*. As mentioned, the implementation works by sending multiple rays through each pixel. To do this, a random offset between -0.5 and 0.5 was added to the fragment position when calculating the position in the camera plane of each ray, the average of the accumulated light from each ray was then calculated and assigned to be the fragment color.

2.3.2 Surface intersection tests

To know if a ray hits an object in the scene every surface in the scene has to have a ray intersection test performed on it. The amount of tests can be drastically reduced by the use of a BVH-tree, which is explained in 2.6. The tests look very different depending on if the object in question is a triangle or a sphere.

Triangle intersection test For triangle intersection tests we use the Möller-Trumbore algorithm. The Möller-Trumbore algorithm takes the edges of the triangle using its' vertices

$$\begin{aligned} \mathbf{e_1} &= \mathbf{v_1} - \mathbf{v_0} \\ \mathbf{e_2} &= \mathbf{v_2} - \mathbf{v_0} \end{aligned} \tag{1}$$

, where $\mathbf{e_1}$ and $\mathbf{e_2}$ are the calculated edges and $\mathbf{v_0}$, $\mathbf{v_1}$ and $\mathbf{v_2}$ are its' vertices. The algorithm essentially parametrises these values using barycentric coordinates u and v which

gives the point of intersection given the equation

$$\mathbf{T}(u,v) = (1-u-v)\mathbf{v_0} + u\mathbf{v_1} + v\mathbf{v_2}. \quad (2)$$

Combining this with the equation for defining the endpoint of a ray

$$x(t) = \mathbf{p_s} + t\mathbf{d} \quad (3)$$

, where $\mathbf{x}(\mathbf{t})$ is a point along the ray, $\mathbf{p_s}$ is the starting point on the ray, t is the distance the ray has traversed and $\mathbf{d}$ is the ray direction, we can derive the equation

$$\mathbf{T} = \mathbf{ue_1} + \mathbf{ve_2} - t\mathbf{d}. \quad (4)$$

An intersection has occurred when the equation is fulfilled with the requirements that

$$0 \leq u, v \leq 1 \quad and \quad u + v \leq 1. \quad (5)$$

If an intersection has occurred then the distance of the ray can be found with the equation given from equation 4

$$t = \frac{\mathbf{Q} \cdot \mathbf{e_2}}{\mathbf{P} \cdot \mathbf{e_1}} \quad (6)$$

where $\mathbf{Q}$ is the cross product between $\mathbf{T}$ and $\mathbf{e_1}$, and $\mathbf{P}$ is the cross product between $\mathbf{d}$ and $\mathbf{e_2}$.

Sphere intersection test The intersection test for the sphere is very different since it is parametrized, and has no normal predefined. The equation of a sphere is given by

$$(\mathbf{x_c} - \mathbf{C})^2 = \mathbf{r}^2 \quad (7)$$

, where $\mathbf{x_c}$ is any point on the surface of the sphere, $\mathbf{C}$ is the center of the sphere, and $\mathbf{r}$ is the radius vector from the center of the sphere to the surface. The ray equation was given by equation 3. By inserting the ray equation into the surface point on the sphere $\mathbf{x_c}$ we get

$$\mathbf{r}^2 = (\mathbf{x_r} - \mathbf{C})^2 = ((\mathbf{p_s} - \mathbf{C}) + \mathbf{D}t)^2 \quad (8)$$

Substituting the coefficient of the expanded equation given in 8 gives a solution to the distance t with the equation

$$t = \frac{-c_2 \pm \sqrt{c_2^2 - 4c_1c_3}}{2c_1}. \quad (9)$$

The project was optimized using the fact that the values within the square root give away in what manner the sphere was hit. From 9 we get that

$$arg = c_2^2 - 4c_1c_3 \quad (10)$$

if $arg < 0$ then no intersections have occurred and no further calculations need be computed. If $arg = 0$ then the ray has touched the sphere, only hitting it once. If $arg > 0$ then the ray has intersected the sphere twice. Finding the hit point with the lowest distance value is of course the closest, and therefore also pointing against the ray direction in the first point in which the sphere was hit.

2.3.3 Bouncing the Ray

Once the ray has hit a surface, the ray should either be terminated or bounced. The ray termination condition that was implemented was a simple check that makes sure the ray does not bounce more times than a *maxBounces* variable. How the ray should be bounced is determined by the material and the material's BSDF, the Bidirectional scattering distribution function. The function *raytrace(Ray ray)* was implemented to bounce the argument ray iteratively using a for-loop. Before the loop, the variables *importance* and *accumulatedColor* were defined to be 3D vectors, *importance* with a value of 1.0 in each component and *accumulatedColor* with 0.0 in each component. The *importance* variable keeps track of the ray contribution as the ray bounces off multiple surfaces and the *accumulatedColor* is used to accumulate color for the final pixel. The outline of the loop was to check for intersection between the ray and a surface, then update the *importance* by multiplying it with the surface color. At this stage, the direct illumination on the surface is multiplied by the *importance* value and added to the *accumulatedColor* variable. Lastly, the new ray direction from the intersection point is calculated and set. Once the loop has finished, the *accumulatedColor* variable is returned and averaged with the other rays for that pixel to produce the final pixel color.

2.4 Materials

In order to simulate realistic images, it is necessary to accurately model how light interacts with different surfaces. This is achieved through the use of Bidirectional Scattering Distribution Functions (BSDFs). A BSDF describes the relationship between incoming and outgoing light directions at a surface point and decides how light is either reflected, refracted, or absorbed by a material. It generalizes both the Bidirectional Reflectance Distribution Function (BRDF), which handles opaque materials, and the Bidirectional Transmittance Distribution Function (BTDF), which handles transparent materials. In the following section, three different BSDFs are covered.

2.4.1 Lambertian Reflector

The most basic material that was implemented first was a perfectly matte material. The reflection model is called the Lambertian Model and makes the assumption that light is reflected off the surface equally in all directions. This is as stated by Pharr et al. (2017)[5] not physically plausible but makes for a good approximation of for example matte paint. The model was implemented in code by generating a point on an infinitesimal hemisphere with its center being the point on the surface to be shaded. The axes of the local coordinate system were then calculated using the incoming ray direction and the normal of the surface. The axes were set to be the normal, tangent and bitangent of the surface. These vectors were then multiplied by the coordinate on the hemisphere to acquire the outgoing ray direction.

2.4.2 Specular Material

A specular material is one that reflects light perfectly in the mirrored direction. A simple specular material for creating mirror surface was implemented by reflecting the ray direction around the normal. This material was later combined with the Lambertian reflector to achieve a semi-specular material. This was done by assigning a *smoothness* value to the material, if a random number was below the *smoothness* value, then the ray was reflected, otherwise it was bounced in a random direction using the same function as for the lambertian reflector.

2.4.3 Transmissive Material

The final material that was implemented was a transmissive material, in our case glass. When a ray hits a transmissive object, it can either reflect or refract into the surface. Which of which is decided based on the Fresnel effect. The effect is as follows, the amount of reflected light depends on the viewing direction, meaning that a surface will appear more reflective when viewed at a angle close to parallel with the surface. The Fresnel factor is computationally heavy to calculate which is why Schlick's approximation is used instead. In the implementation, the Fresnel factor is approximated and a random number between 0 and 1 is generated. If the number is below the Fresnel factor, the ray gets reflected, otherwise it gets refracted. Once a ray has been refracted into a object, it can either be refracted out of the object or be reflected based on Fresnel. However, if the angle between the ray and the surface normal is below a critical angle, total internal reflection will occur and the ray will keep reflecting inside of the object.

2.5 Accumulation

Accumulating samples over time is a necessity if any noise-free image wants to be rendered. This is because a decent rendering without noise may require thousands of samples or rays per pixel which would take too long to run in the fragment shader. Instead, a smaller number of samples are calculated per frame in the fragment shader. The color of the frame is then blended with the previous frames using a rolling average, which can be seen in equation 11, C_n being the final color

displayed to the user and n being the number of frames rendered. This is accomplished by ping-ponging between framebuffers.

$$C_n = \frac{C_{n-1} + n \cdot accumulatedColor}{n+1} \quad (11)$$

The name ping-ponging comes from how the method works by having two framebuffers with textures binded to them, and then having those textures swap after each frame has been rendered. One texture holds the already accumulated color, while the other one gets drawn to with the new and rendered image. The latter gets the C_{n-1} from the other texture. After the frame has been displayed, the textures swap place so that the recently rendered image becomes the one to hold the color C_{n-1} for the next frame, and so it repeats.

2.6 Bounding Volume Hierarchy

A bounding volume hierarchy, which will be referred to as a BVH from now on, is a data structure which follows a classic tree pattern. The purpose of this data structure is to fulfils a purpose that most tree structures do, namely, it allows for a search time complexity of $O(log(N))$ as opposed to $O(N)$. Using a tree structure for finding the primitives in the scene then helps reduce the intersection test. Instead of iterating through every single triangle we traverse the tree till we find a suitable triangle, or none is found, drastically reducing the time to render each frame. The gist of the BVH-tree can be seen in 3. The example is in 2D but can and is easily applied in 3D. The primitives, which in the image is two triangles and a circle, get an axis-aligned bounding box which we will refer to as an *AABB* from now on. These bounding AABBs are seen in the (a) section of 3. Then one step up the tree you have a node, which is a combination of the AABBs of the nodes underneath it. This keeps on going until we have a root node at the top which encompasses the whole scene. The full tree for the scene in (a) is seen in (b). It is worth noting three things: firstly, the primitive bounding boxes are not actually apart of the tree structure, the blue nodes in the image instead point directly at the primitives indices, more on this in 2.6.2. The second point worth mentioning is that the BVH tree in our project is built with a top-to-bottom approach, starting with the root and then traversing down, which is more standard. Lastly, there is is not only BVH nodes and the primitive vector that is necessary to traverse the tree, there is also a vector of triangle indices. This is necessary since we wanted the vector of primitives to be constant and instead of partitioning the primitives we instead partition the indices. More on this in 2.6.2.

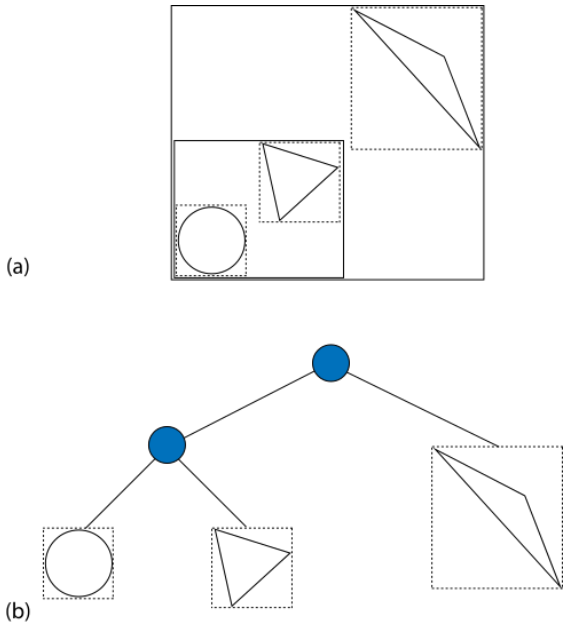

Figure 3: BVH-tree structure.[5]

2.6.1 BVH Node

As a reminder, our current implementation of a BVH Node is seen in figure 4. The *bBoxMin* and *bBoxMax* variables are the positions of the corners of the AABB. The *leftChild* and *rightChild* variables are the indices of the child nodes of the current node. The *startTriangle* variable is the index of the triangle, or in our case primitive, that the node is pointing to. The *triangleCount* variable is the amount of

primitives that the node has under it, which could be zero, or if it is a leaf it should have 1-inf, but a good BVH tree should not have many more than two. The *escapeIndex* is a way to optimise traversing the tree in the fragment shader, which is explained in 2.6.3. Lastly, the *pad1* variable is just to make sure that the each node is packed with a size divisible by sixteen, which was explained in 2.2. It is worth noting that if the BVH tree was build to be used on the CPU side of the code, it would be easier and more intuitive to use pointers for each node instead of indices for the children indices and escape index, but pointers do not exist on the GPU side, so indices have to be used instead.

```
struct BVHNode {
    vec3 bBoxMin;
    int leftChild;
    vec3 bBoxMax;
    int rightChild;
    int startTriangle;
    int triangleCount;
    int escapeIndex;

    int pad1;
};
```

Figure 4: BVH Node struct.

2.6.2 Building the tree

As mentioned in 2.6 the tree is built with a top-down approach. As such, we chose to use a recursive function called *buildRecursive* which has the input arguments be the start triangle index, the amount of primitives the current node has under it, and the current depth of the node in the tree structure. The output is the index return index of the node.

First we calculate the full bounds of the current node using all the primitives that it holds, then we create the node and set its' bounds. The node index is then decided using the amount of nodes in our nodes vector, then the node is pushed back to the end of said vector. Then we check a base case: if the amount of primitives is less than or equal to a pre decided amount of primitives, we make this node a leaf. This is done by applying the triangle count and and start triangle to this node, then we say that it has the left and right child index of -1. These indices can later be used to find if a node is a leaf or not during the tree-traversal. Lastly, we set the escape index to be current node index + 1. This works because the way the program is set up, we always create the left nodes first in a recursive manner, meaning the tree will create the bottom most branch before creating even a single right branch. There are then three cases to think about:

Case 1; left leaf : Let us imagine the leaf at the bottom left, the next node that will be created after it will be the right leaf next to it, or the right node of a parent to the current left leaf, which would be the next node index. So for this case, just adding one on top of the current index is satisfactory.

Case 2; right leaf : The right leaf is similar to the left, in that the next node that is created is bound to be a right node to a parent. So again we can just add a one to the current index.

Case 3; non-leaf : If we are not at a leaf, we will not have a base case occur. For this reason, we will have already added all the left and right children. So an escape index of the current node should be the current size of the nodes vector, which at this point will be larger that the current index, and because of how indices work in c++ vectors we actually do not need to add anything onto it.
But to order the BVH tree nodes correctly, we want to create a proper heuristic so that the depth can remain short and consistent.

The surface area heuristic is a heuristic that tries to keep as much empty space as possible between the AABBs, to rule out as many primitives as possible. Another way of understanding is to think of the heuristic as a way to calculate the odds of hitting an AABB, and then use the configuration that makes the odds the lowest[4]. This heuristic does not account for the camera being inside of the object, where the heuristic suddenly does very poorly. In such cases, the normal way of simply splitting the largest axis can be utilized. This heuristic would normally be very expensive to compute, having a time complexity of $O(N^3)$ for every level of splitting up the tree. The reason for this is that we have to account for every centroid of the primitives contra every axis. This is very expensive, but if we utilize what is commonly referred to as bins, we can get a good estimate for a perfect SAH tree. A bin system would essentially divide up the axis into a set number of uniform planes, and then test the heuristic relative to these planes. The idea is visualized in 5. The more bins we add, the more the heuristic will converge to the same result as not having the bin system in place. After finding the best solution for the SAH, we partition the triangle indices by if the centroid of the primitive is less in the split axis or more than the split axis.

After that all that is left to do is create the children of the node and set the escape index.

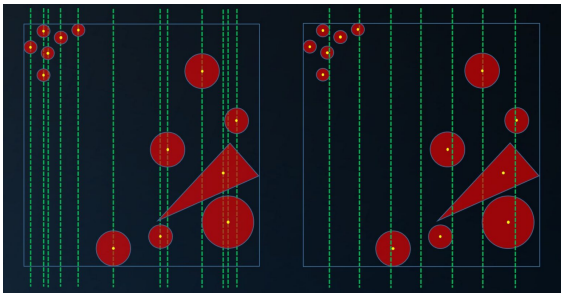

Figure 5: Left: split plane candidates through the primitive centroids. Right: uniformly spaced split plane candidates.[6]

2.6.3 Traversing the tree

To traverse the tree we start at node index zero. The function then loops until the node index does not equal -1 or until the nodeIndex is outside of the scope of the tree, which will happen for nodes that have no proper escape index. There is then an intersection test with the AABB of the current BVHNode, and if it hit another check for if it has a triangleCount is done. If it does then the primitives are tested and if one or more of them are found that has a proper intersection on them then the closest one is saved and the BVH tree continues on with it search until the index is out of bounds.

The purpose of the escape index is to remove a dependancy of a stack, which is the normal way to traverse a BVH-tree. There is however no stack in GLSL and while there are solutions such as creating a static array that we save each index to, we figured we would use an escape index approach since it saves on memory, and it also helps us traverse the tree a bit faster.

3 Results and Conclusions

Results, conclusions and possible improvements of the project

3.1 Results

The final application produced implemented a path tracing algorithm for rendering triangles and implicit spheres. A diffuse material, specular material, and transmissive material were implemented. The option to load models of the Wavefront file format was implemented and along with that a BVH acceleration structure to handle larger models.

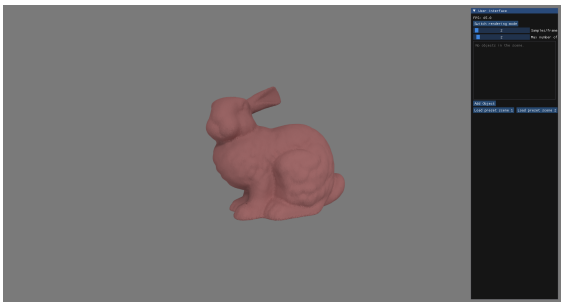

Figure 6: Bunny with 70k triangles rendered at 1920x1080 resolution. As long as the camera is not to close to the bunny the framerate is around 50-150 per second on a Nvidia Geforce GTX 1070.

Figure 7: Bunny with 70k triangles rendered at 1920x1080 resolution.

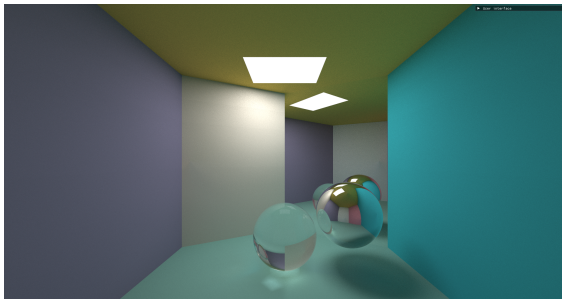

Figure 8: A room scene with mirrors and a glass sphere rendered at 1920x1080 resolution. Around 30-40fps on Nvidia Geforce GTX 1070.

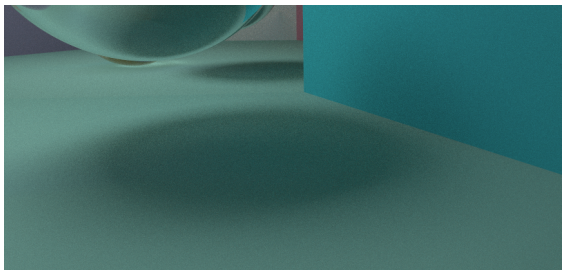

Figure 9: An example of bounce light in shadow.

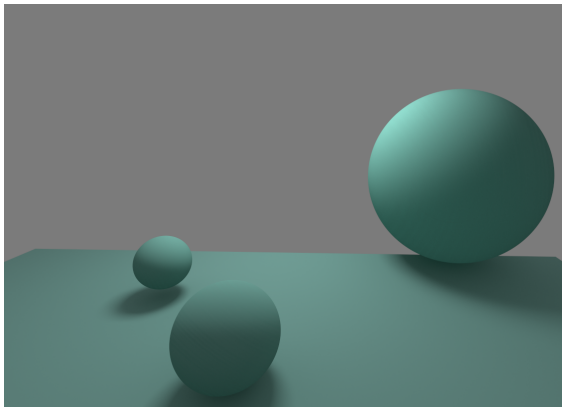

Figure 10: An example of diffuse spheres.

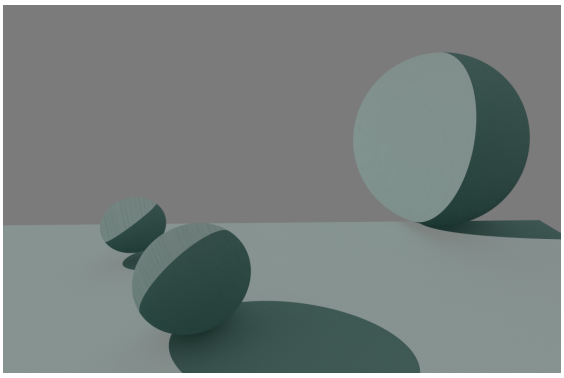

Figure 11: An example of point lights.

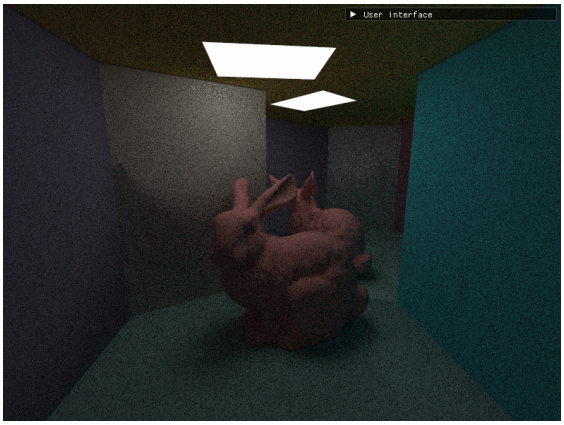

Figure 12: The room scene with the 70K bunny in it.

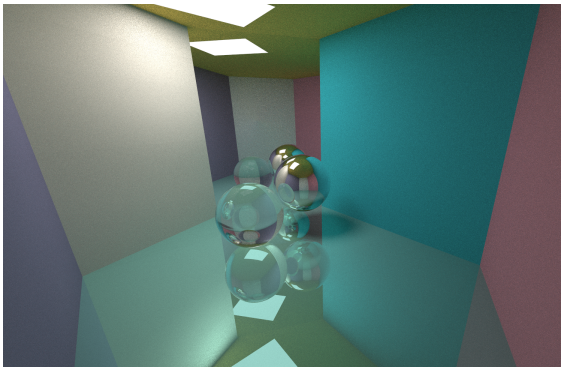

Figure 13: An example of a semi-specular floor.

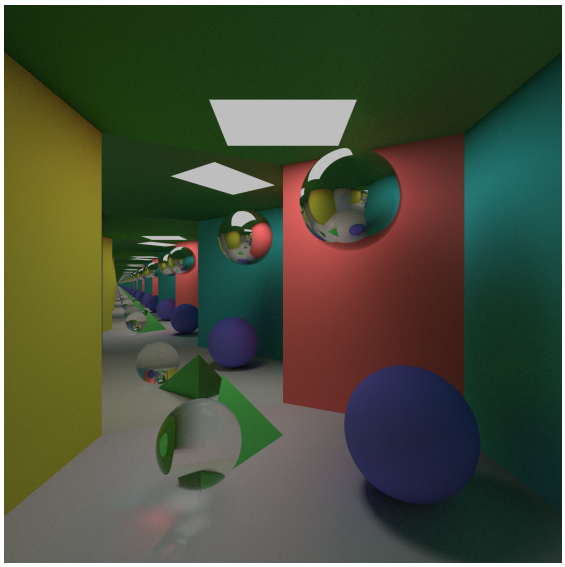

Figure 14: Result from previous CPU path tracer project made during the course TNCG15 as comparison. The image took around 1.5-2 hours to render, all tough we did not document the exact time.

3.2 Discussion

This project has put a lot of perspective on the up sides and down sides of programming on the GPU as opposed to the CPU, since it is similar to our previous path tracing project. The upside is of course that rendering an image, even if it is just a single one per second, is incredibly useful for debugging purposes. This is something that might be overlooked since normally we would have a lot more debugging tools available when writing CPU code, but the rendering time on CPU bottle necks the time investment since just observing the results and seeing if there is something that is not working correctly is the best way to debug a path tracer.

With that being said, once realizing that something is not working as intended, debugging the issue in GLSL is much worse. One easy way to test if things work as intended on CPU is simply by console logging a specific part of the code. This is not possible in GLSL and makes testing very difficult. Pretty much the only way to get direct feedback is to set the output colour of a pixel to the value

of the value that we want to debug, which if the value is used in the middle of a function might mean that the functions output has to be changed just to test that particular value. So creating the path tracer on GPU has upsides and downsides, but if you are comfortable debugging GLSL code by pretty much just looking at it and extrapolating where the issue comes from with the help of the render, then GPU is definitely the way to go.

Now, even though we had an easy way to debug our results by just rendering the scene, we still managed to get a lot of things wrong before presenting our work during the scheduled presentation. There were namely two large issues that we did not realise was issues at the time, and another one in which we were aware of but not sure why it occurred. The first one that we were unaware of was the fact that spheres had the wrong normal calculated. The way we calculate the normal of a sphere is to simply take the endpoint of the ray, where it intersected the sphere, and subtract the center of the sphere from it. Applying the right direction of this vector and normalizing it gives the correct normal. What we accidentally did was to instead take the rays start position, and then do all the other steps. This made it so pretty much any collision that the sphere had would also have the wrong normal applied to it. This would have been extremely obvious to us if we were testing the mirror sphere, but we will get to that in a bit.

The second issue that we were unaware of is that the bounce lighting was entirely broken. The examiner did question it during the presentation, since the results did not include them, but to our mind at the time we had it implemented. Now we were in a way both right, since it was implemented, but one single line made the bounce light malfunction. This line was introduced after we had already tested bounce lighting and we simply did not consider that things might not work as intended. After considering the feedback from the presentation and looking back at earlier results, we figured out the bug and fixed the issue.

The last big glaring issue we had was one which would have revealed the first aforementioned issue, namely that our spheres could not be mirrors. As seen in the CPU path tracer in figure 14 this is not intended, since the way the implementation is meant to work the spheres should be able to be any material we give it. The issue was not with the normals, and it was not with any of the sphere logic at all. The fix was simply to offset the end of a ray, meaning the point of contact on the sphere, by a small amount away from the sphere surface. This made the spheres work normally. Now, why this was confusing is that for any other material on a sphere, there was no issue not re-intersecting with the same sphere immediately after contact, but for the mirror it clearly happened so often it always crashed the program.

3.3 Possible Improvements

Interpolation of normals: Something that is achieved very easily in rasterized graphics is interpolated normals. This is a feature that was not added due to time constraint and the effect of having one normal per triangle is flat shading. This can be seen clearly in figure 7. It is of less importance when rendering a model with a diffuse material, but as soon as reflections are introduced, it becomes more noticeable. Rendering the bunny with a semi-specular material results in the surface looking more like that of a diamond than of plastic.

Glossy material: One of the project specifications was to implement a glossy material. The closest to a glossy material in the implementation is the semi-specular material which blends between perfect reflection and diffuse reflection based on a *smoothness* variable and can be seen on the floor in figure 13. It was realised late into the project that this was not sufficient to produce glossy objects such

as tomatoes, which have blurry highlights. It is only sufficient to produce for example plastic surfaces. Implementing a glossy material would mean blending between diffuse reflection and a reflection that is spread around the perfect reflection angle in a lobe.

BVH-tree thoughts: While the BVH-trees improved the frame rate of the renders by over 100 frames in some cases, they are still not optimal. First issue is that the SAH is not meant to be used when the ray is starting inside of the AABBs. To counteract this, we could have the walls of the room not be apart of the tree at all, and keep separate objects to separate BVH trees and iterate through the objects. This way if we did have camera inside the room the BVH tree for say the rabbit inside that room would be balanced neatly and we would probably gain a lot on the frame rate.

Another issue that we believe is due to the BVH implementation can be seen on the 70k bunny when moving closer to the ears. What happens is that some triangles seem to be lost and do not show up in the render. It is not a matter of the model being broken since in raster mode the model looks just fine. So this might be an issue of the logic in either traversing the BVH tree, or the creation of it, but we have yet to find the specific error causing this issue.

3.4 Conclusion

The major conclusions that can be drawn from the project is that path tracing on the GPU is very fast and can be made to work in real-time. However, a clear downside is that it is very difficult to debug and develop a path tracer in the fragment shader alone, compared to on the CPU. The BVH-tree made a big difference to performance when rendering large meshes with thousands of triangles but some bugs still remain in the implementation. The materials implemented were quite basic BSDFs and the implementation of glossy objects such as tomatoes would require further work.

References

[1] Generating Camera Rays with Ray-Tracing — scratchapixel.com. https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays.html. [Accessed 24-05-2025].

[2] Shader Storage Buffer Object. https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object.

[3] Uniform Buffer Object. https://www.khronos.org/opengl/wiki/Uniform_Buffer_Object.

[4] Erin Catto, Blizzard Entertainment. Dynamic Bounding Volume Hierarchies. https://box2d.org/files/ErinCatto_DynamicBVH_Full.pdf.

[5] Matt Pharr, Wenzel Jakob and Greg Humphreys. Physically Based Rendering: From Theory To Implementation. https://pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies.

[6] JBIKKER. How to build a BVH - part3: Quick Builds. https://jacco.ompf2.com/2022/04/21/how-to-build-a-bvh-part-3-quick-builds/.

[7] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.