COSC 3P93 – Project Step 2

OZZIE GRAHAM - 6435275 and BRETT TERPSTRA - 6920201

1 TOPIC

Topic chosen was Triangle Mesh Rendering with depth support using Ray Tracing.

2 FOREWORD

Before we get started here with the write-up we'd like to say a few things about the code/building the code. You will find lots of TODO comments throughout the code and although with this submission we have met the minimum implementation requirements, Project Step 3 will be upgraded with all these missing features, which we will talk about here in the design section.

2.1 Building

CMake is the quintessential build system and is the closest thing C++ has to a standard. As a result, we used CMake to make the building process much easier, since I use Ninja locally to speed up the compile times, and makefiles for distribution. Any modern compiler should be able to build it, however, I have only tested this on Linux with GCC 12. To Build the project, please follow the instructions:

```
cd /your/path/to/the/sourceroot/
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ../
make -j 16
./Step_2
```

3 DESIGN AND LOGIC

3.1 Libraries

Currently, the code uses one library which is included in the include folder. STB Image write is a public domain single header single threaded image writing library that supports .png, .jpg, .bmp, and .hdr. We make use of all of these functions, but the code makes use of the PNG writing function by default. You can change the output format by providing *–output-format "png|jpg|bmp|hdr"* to the command line arguments.

3.2 Vectors

Starting with the most basic part of any ray-tracing algorithm, the vector! We make use of 4D vectors, storing a x, y, z, w, with variable precision. You can change the precision by changing the typedef in the vector class. We decided that 4D vectors are better because they align better with SIMD instructions like AVX and with GPU compute libraries like OpenCL. The vector class comes with two options. By default the engine will compile using standard C++ doubles, in pseudocode:

Authors' address: Ozzie Graham - 6435275; Brett Terpstra - 6920201.

Update date: 19 October 2022

```
class Vec4 {
    [....]
        double x, y, z, w;
    [....]
};
```

However, you can uncomment the define at the top of the vector class to switch to AVX2 256bit vectors. AVX stands for Advanced Vector Extension and they allow us to compress the four individual doubles into one large 256bit register. The CPU will then run batch run a single instruction (say multiplication) over two of these compressed vectors returning the vector result. The AVX version of the vector class would look something like:

```
class Vec4 {
    [....]
    // for conversion back to doubles which we can use with the ray
    tracing engine
    _union {
        struct {
            double _x, _y, _z, _w;
        };
        __m256d avxData;
    };
    [....]
};
```

This benefits us as we can speedup the sequential execution of our vector math by sending the CPU one instruction to run on the vector instead of the multiple instructions in the non-vectored Vec4 class suffers. Example:

```
// Using individual instructions
inline Vec4 operator*(const Vec4& left, const Vec4& right) {
    // 4 whole multiplication instructions!
    return {left.x() * right.x(), left.y() * right.y(), left.z() * right.
    z(), left.w() * right.w()};
}

// vs the vectored version:
inline Vec4 operator*(const Vec4& left, const Vec4& right) {
    // single instruction over multiple data! (SIMD)
    __m256d multiplied = _mm256_mul_pd(left.avxData, right.avxData);
    return Vec4::AVXToVector(multiplied);
}
```

3.3 Rays and the Ray Caster Algorithm

An image is provided to the ray caster class, which contain an width, height, and pixel color data. Rays then are cast multiple times per pixel and averaged to create the final color result. The color results are then written to the image and finally output to an image file.

The pseudocode for this is as follows:

ALGORITHM 1: Image Ray casting

```
Data: image
1 for x in image.Width do
2     for y in image.Height do
3         pixelColor;
4         for s in maxRaysPerPixel do
5             pixelColor += raycast(cameraProjectRayFromScreenToWorld(x, y), 0);
6         pixelColor /= maxRaysPerPixel;
7         image.setColor(x,y, pixelColor);
```

The cameraProjectRayFromScreenToWorld transforms the x and y coord of the image into a world position ray emanating from the camera. World coordinates are in left hand -Z forward. The raycast function takes a ray and a depth value as it is a recursive function returning the color vector. It uses the ray to check with the world if there is an object that the ray intersects with. If it does it asks the object if and how it wants to scatter the incoming ray. In pseudocode:

ALGORITHM 2: Ray casting function

```
Data: Ray, depth
Result: Color
1 if depth > maxRayDepth then
2     return black;
3 if world.rayInteresctsObject then
4     if intersectedObject.scatter then
5         return intersectedObject.diffuseColour * raycast(scatteredRay, depth+1);
6     return black;
7 return skyboxColor;
```

This is effectively all it takes to implement a basic ray tracer. We support triangle ray intersection via the Möller–Trumbore intersection algorithm. It is an algorithm that doesn't calculate the triangle plane, and as a result, is capable of fast intersection calculations. This cannot be made parallel and must be done sequentially. We plan on making our own triangle intersection algorithm, however, we have run out of time to derive one.

Additionally, the engine supports the dynamic loading of .obj type model files. Using a custom model loader we are capable of loading any blender obj output and rendering it in our scene. Obj loading can only be parallelized on a per-object level and since it is loaded before the ray tracing algorithm runs, its run time is irrelevant.

3.4 Limitations

There are plenty of performance limitations with our setup. On the sequential side, every single ray needs to be checked against every object in the scene. Every object may have say 50 triangles meaning in a 1920x1080p image with a ray per pixel value of 50 and a max depth of 50 and 3 objects with 50 triangles each, we'd have 5,184,000,000 rays total possible rays being checked against 150 triangles. A total of 777,600,000,000 triangle ray intersection checks, most of which will fail because the ray goes nowhere near the triangle! Further elaboration on a solution to this problem is in the next subsection. On the parallel side, splitting the image into N sub-quadrants where N is equal to the number of processors.

Running each sub-quadrant on its own processor could in theory provide an embarrassingly parallel speedup since each quadrant isn't dependent on the other.

3.5 Missing Features

There are a lot of missing features that were planned to be implemented (as extra features are not required for the project) and have yet to be implemented. Due to life circumstances and other class assignments we were unable to implement all of them. If this was all made due at the end of the semester then I would've had it completed. Currently, the list of missing features are:

A Graphical display of the image, with real-time updates on how far the CPU has rendered the image. Step 3 will have a display for each thread and where it is running. Along with a bunch of other stats that I might find useful. Plus a visualization of the BVH.

BVH - Bounding Volume Hierarchy. The code is fully in this submission. However, we were running into issues with infinite recursion when trying to add objects. The use of a BVH has the potential to solve the problem listed above since we can check if the ray will intersect using a cheap AABB check. At each level of the tree, we reduce the problem size by half, meaning it's possible to go from checking 15 triangles to maybe 2. Plus rays that are very obviously outside the bounding box will be rejected far before we reach the expensive triangle intersection algorithm.

AVX - Advanced Vector Extension. The code for AVX is currently implemented, however, it doesn't work. Outputs an incorrect image, unknown reason.

Various Algorithms - Wanted to implement various AABB, triangle intersection, and material algorithms to graph and compare their performance.

Lights - Wanted to add lighting to the scene, it may come in the future.

3.6 Extending to the Parallel

Although we have already discussed our plans to make the ray tracer parallel, we will reiterate. We can split the picture into segments each containing a specific and equal number of pixels. A slave thread will be created for each block of the image. In theory, the more blocks there are, the faster the program should run because there will be more threads. Each slave thread will perform the same set of instructions but over a different set of data, a sort of SIMD System. There will be a master thread that will then combine all the segments which will create the final image. We can split the image down to each pixel gets its own thread, but we are limited at from that point. We could make each ray have its own thread but that would come with a lot of overhead, even if the bounces are handled by the same processor.
Planning on implementing each segment of an image parallel rather than each thread creating its own image would produce more of an animation and would be too computer resources costly.

4 EXAMPLE OUTPUT

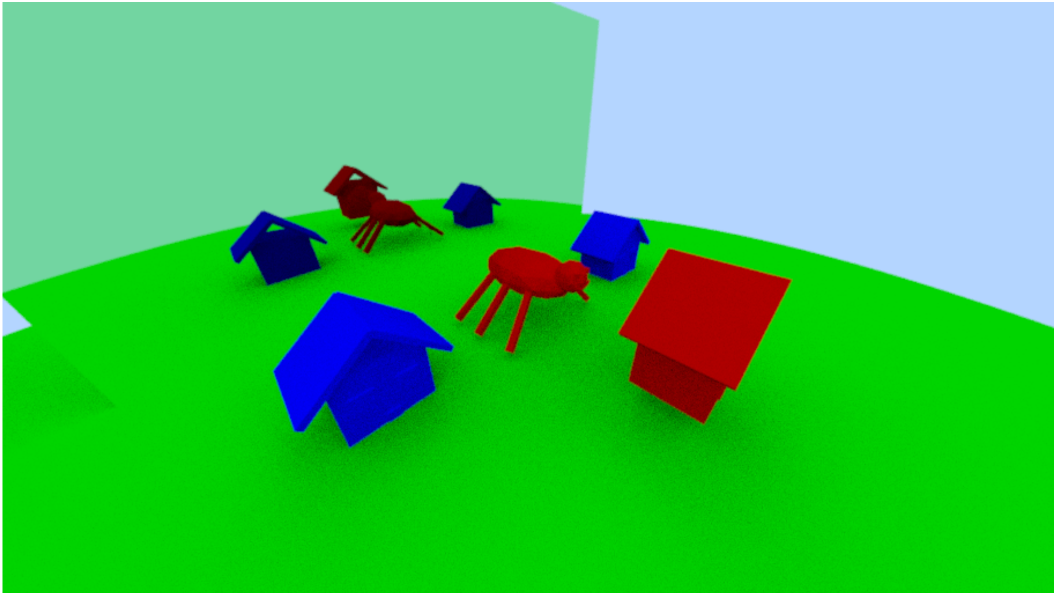

Fig. 1. Rendered output of the code provided in this submission. You can see the reflective green mirror in the background, with a red spider and his three homes. Ambient Occlusion occurs naturally as a result of the diffuse material scattering using full unit spheres.

5 PERFORMANCE ANALYSIS

The larger the image, the more the ray bounces or the higher the max depth, the more time it'll take to complete the render.

5.1 Vectors

There are two simple methods that we can use to generate unit vectors, which are mostly used in diffuse scattering. The first is to randomly generate vectors with x,y,z values in the range [0, 1], rejecting them if the length is greater than 1. This is guaranteed to generate a random vector inside the unit sphere.

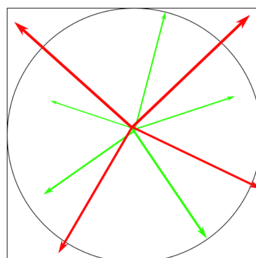

Fig. 2. Where red arrows are rejected and green arrows are accepted and normalized.

The second method is to generate the random vector but instead of checking if it fits within the unit sphere we just normalize it so it does. This is about 18% faster than using

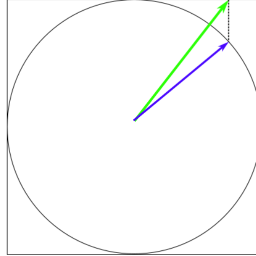

Fig. 3. We normalize the first vector we generate (green) resulting in the blue arrow.

the rejection method, however, it produces results that are slightly less realistic. This is due to the rays occurring more frequently near the corners of the square, resulting in fewer rays in the cardinal directions.

5.2 Intersection Algorithms

The algorithms we use to check intersections are slightly better than a crude plane intersection algorithm, however there are more complex algorithms that we might implement in the future.

5.3 Bottlenecks

The potential bottleneck is having to check every single triangle everywhere which would have to check the entire scene. This could be upwards of millions of rays which would cause a performance decrease and the program wouldn't be as efficient. We have already touched on this with the BVH.

REFERENCES

[1] Wikipedia contributors. Möller–Trumbore intersection algorithm, Wikipedia, The Free Encyclopedia, 23 Jul. 2022.
https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm
, Accessed October 19, 2022