Computer Graphics Coursework 2

s1969754

1 Demo

Figure 1 is an example of a render from the ray tracer described in this report (using the submitted demo.json file). It includes two TriMesh shapes and a sphere, as well as planar quads as a wall and floor. The sphere demonstrates a very reflective material that also has a green colour, whereas the pyramid demonstrates a very refractive, glass-like material. Texture mapping is also demonstrated on both the floor (planar quad) and the block (TriMesh).

The render uses an AreaLight with jittered sampling and the camera is a pinhole camera. The number of bounces is 5.

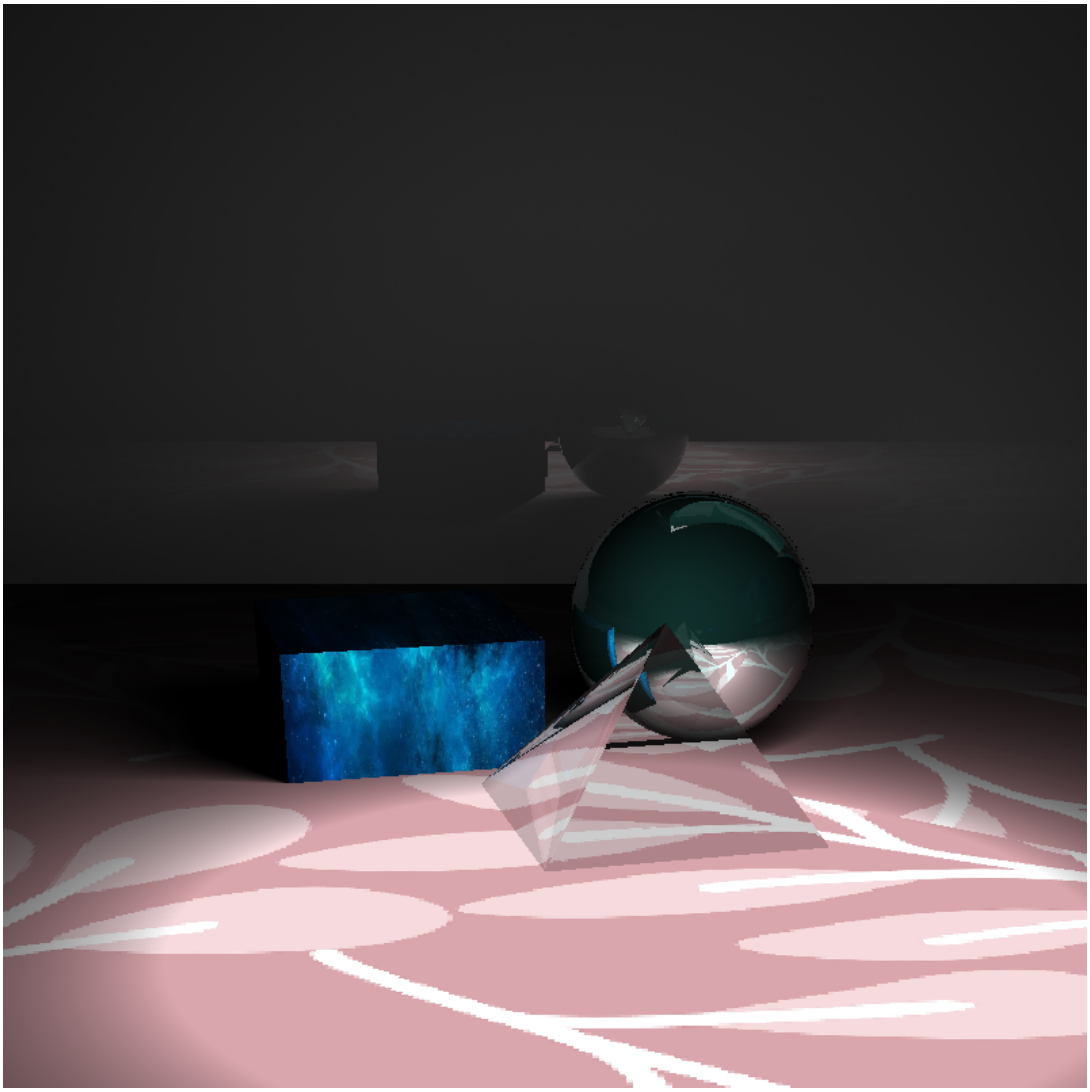

Figure 1: Example render from ray tracer using the demo.json file.

2 Basic raytracer

2.1 Components

First, three basic components were implemented: a pinhole camera, a point light source and a sphere. Using these components a basic ray tracing algorithm was implemented, including ray-sphere intersection. Then, planar quads and ray-plane intersections were implemented in order to see any shadows thrown by the sphere. Blinn-Phong shading was added, as well as reflections and refractions.

Triangles and TriMeshes were then implemented, as well as ray intersection with these shapes. The TriMesh shapes are defined using a collection of Triangle shapes which are created in the TriMesh constructor. To check for an intersection with a Trimesh shape, the algorithm checks for intersections with all of the triangles that it consists of and returns the one closest to the camera (if there is a hit).

2.2 Ray tracing

The basic flow of the ray tracing is as follows. First, the scene and cameras are parsed in the main class, and the `render()` function from the RayTracer class is called. The `render()` function iterates through each pixel in the output image, and shoots a ray from the camera through that pixel into the scene. A single ray is traced using the `trace()` method in RayTracer.

The `trace()` method iterates over the shapes in the scene and returns an instance of the Hit struct that is closest to the camera (or a Hit with the field `hasHit` as `NO_HIT` if it is a miss). Once a hit is found, the method iterates over all light sources and computes the shadow at that point. After that, the reflectivity and refraction are computed by recursion on `trace()` function. The `trace()` method is called recursively as many times as specifed by `nbounce` in the input json, until the ray no longer hits any shape, or until there is no reflection or refraction. Lastly, `trace()` applies a texture to the shape if applicable and the final colour at the pixel is returned.

Once the image has been rendered, the `tonemap()` method is called on the result in order to transform the rgb values between [0,1] into the range [0,255].

3 Textures

Once all shapes were implemented, texture mapping was implemented for all of them. The texture information is stored in the Material of the shape. A texture is mapped to the shape if there is a path specified in the input json file, otherwise the parsing will assume that no texture was intended (even if there are other texture parameters such as tWidth).

The texture mapping of TriMeshes is particularly interesting. Either each face of the trimesh can be mapped with the entire texture or the mesh must be flattened so that a section of the texture can be used for a particular face. In my implementation, the second method is used, though the flattening method is simple. Essentially, the trimesh is unravelled into a long strip as seen in Figure 2. The example shows how a TriMesh with 6 faces (or five, in which case the last triangle in the texture doesn't show on the shape) is unravelled to fit a texture. The faces on the texture are in the same order as they are in the input json.

Texture mapping for planar quads (floor) and TriMeshes (cuboid) can be seen in 1, and texture mapping for Triangles (right side wall) and Spheres can be seen in Figure 3.

4 Bounding Volume Hierarchy (BVH)

Bounding Volume Hierarchy was implemented to speed up the rendering. The BVH tree is implemented using a top-down method and the bounding volume is a bounding box.

Building of the BVH tree is initialised in the `createScene()` method in the `Scene` class. The method creates the scene as described in section 2, and then uses the `shapes` vector to build a BVH tree. Once the BVH tree is built, the `shapes` vector is cleared and only the root of the BVH tree is added back to the vector.

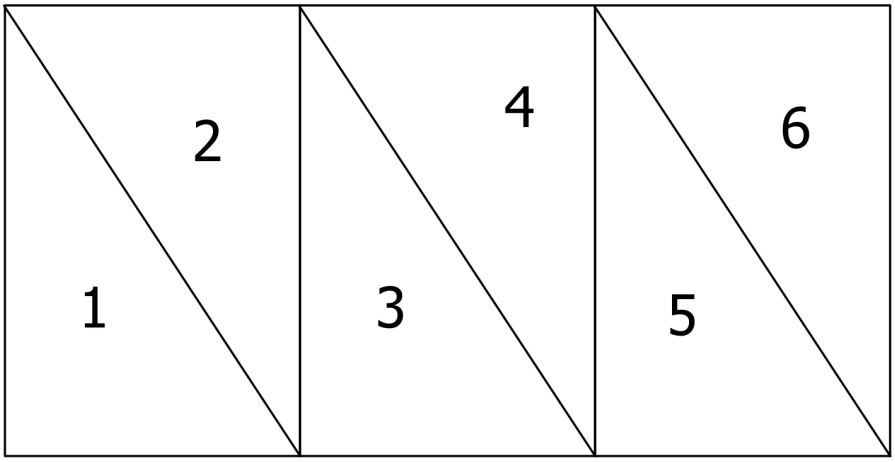

Figure 2: Texture mapping of a trimesh with 6 (or 5) faces.

In addition to the lectures, Sulaiman and Bade[1] was consulted in implementing the BVHs.

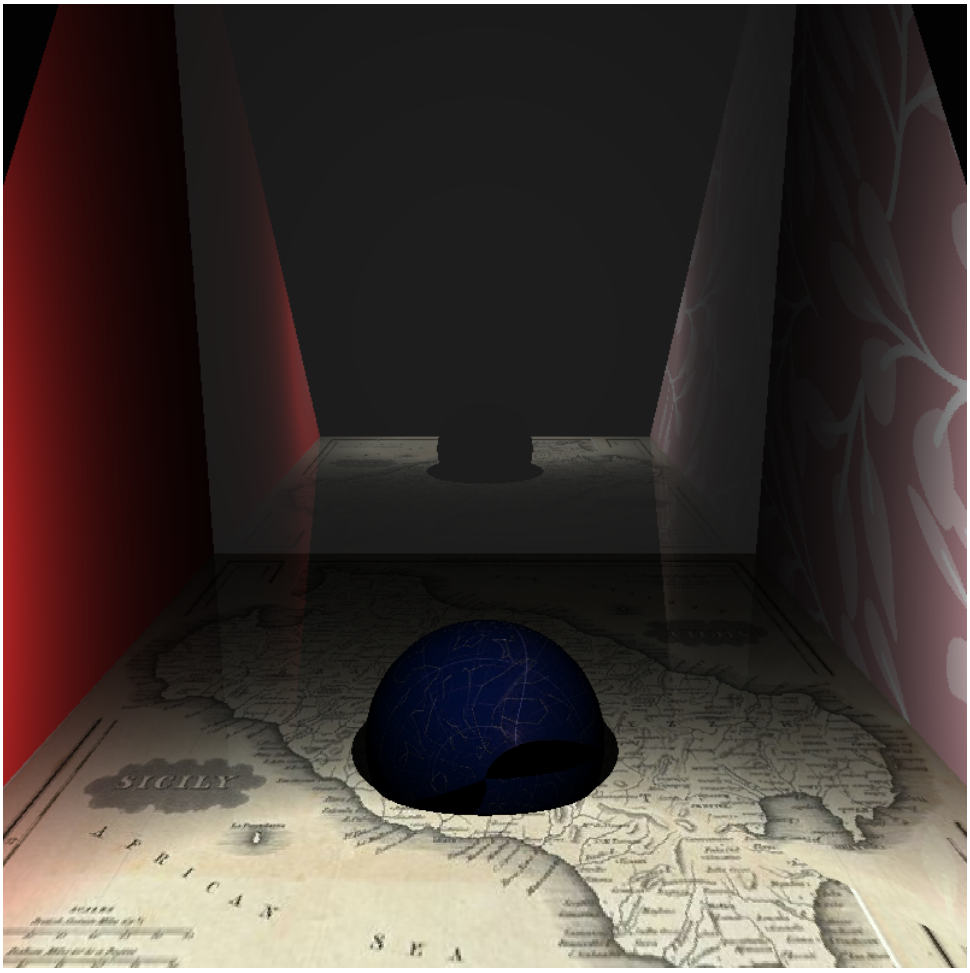

Figure 3: Render of the example.json file provided in the coursework material, with a texture added to one of the triangles instead of the blue colour.

4.1 Tree creation

A BVH tree is created by calling the constructor of the BVH class. There are three parameters for the class: a vector of shapes, the maximum amount of shapes in a leaf node, and an integer that specifies the axis to split on. In this implementation, the axes to split on are the base vectors of the scene space (i.e. the x, y and z-axes, where the input parameter is 0,1, or 2 respectively).

Each shape has a centroid that can be retrieved using `getObjectCenter()`. The centroid is simply the centre of the shape's bounding box. The median of all centroids in the vector of shapes is calculated (along the axis specified in the parameters). Using the median the shapes in the input vector are then split into a left and right subtree (the subtrees are also BVHs). Given that the input shapes are split in two by the median value, the left subtree can be at most one node larger than the right subtree at any point in the BVH tree.

The left and right subtrees are initialised recursively by calling the BVH constructor again. The split-axis is changed each recursion by calculating `(currentAxis + 1) % 3`. Further subtrees are no longer created when the amount of shapes in the input vector is less than or equal to the amount specified in the parameter. In the leaf nodes, all shapes are stored in a `shapes` vector.

Each shape (including BVHs) also has `getMaxBound()` and `getMinBound()` methods. These methods specify the bounding box around the shape. Bounding boxes are aligned with the base axes and enclose the entire shape. The bounds of a BVH object are specified using the minimum and maximum bounds of the shapes in *both* of its subtrees.

The amount of shapes in the leaves is specified to be 1 in the `createScene()` method when creating the root, given the relatively small amount of shapes in the scene.

An improvement on this BVH structure could be to improve how `TriMesh` shapes are handled. In this implementation the entire object is considered as a whole, i.e. the individual triangles cannot be further split. Splitting a `TriMesh` object further could speed up the ray tracing.

4.2 Traversal

In the ray tracing algorithm, BVHs are treated like any other shape. When tracing a ray, the `intersect()` method is used to check for an intersection. In BVHs `intersect()` first calls the helper method `intersectBVH()`, which checks whether the ray hits the BVH bounding box. There are three possible results:

1. No hit
2. Hit, and this BVH *is not* a leaf
3. Hit, and this BVH *is* a leaf

In case 1, the `intersect()` method is immediately returned as a miss. In case 2, `intersect()` is called recursively on both subtrees and if both subtrees return a hit the closest of these hits is returned. In case 3, all shapes in the BVH are iterated over and `intersect()` is called on each of the shapes to find the closest shape (exactly like without BVHs), and the result from that is returned. Hence the results from the shapes are propagated back through the tree.

5 Performance

The performance of the ray tracer was measured on 5 different scenes of different complexity. Each scene was rendered 10 times, and the average was taken of this number. Scenes 1, 2, and 3 use point lights, whereas 3,4, and 5 use area lights. All scenes use pinhole cameras.

The performance without BVH was measured by commenting out the code regarding BVH in the Scene class. Without this code, no tree is built and all shapes are in the root, i.e. in the vector of shapes (with BVH the only shape in this vector is a BVH).

Table 1 depicts the recorded performance.

	BVH	No BVH
Scene 1	2.78	2.19
Scene 2	5.67	4.3
Scene 3	23.43	33.81
Scene 4	133.72	173.32
Scene 5	380.11	581.87
Scene 6	477.3	691.26

Table 1

The results above indicate, however, that BVH increases the speed of rendering on renderings with area lights. Furthermore, Scene 3 had 5 more elements compared to Scenes 1 and 2, which may indicate that BVH provides more benefit as the amount of elements grows. BVH being more useful on a larger amount of elements is also indicated by the fact that scenes with area lights are sped up with BVH (due to multiple sampling the amount of iterations through the shapes is larger than with a point light). With fewer elements, the overhead of building the tree is not necessarily worth it, as seen by scenes 1 and 2, where the BVH performs slower.

A more throughout benchmarking of the ray tracer would require many more scenes to be generated and rendered. The scenes are quite tedious to build, and it is difficult to make scenes that are comparable to each other, as number of elements, sizes, locations and so on all can affect the performance significantly.

6 Distributed raytracing

6.1 AreaLight

The arealight implemented in this ray tracer is always a planar quad, and four corners need to be specified in the json (similarly to Plane shapes).

The AreaLight makes use of Monte Carlo integration to calculate the shadow for each pixel. Each time a ray hits a shape, x amount of shadow-rays are sent from the hit point to a light source. The ray returns 0 if the shadow-ray hits an object on the way and 1 if it does not. The result is then averaged over all samples to get the total shadow thrown by that light.

Both jittered and random sampling were implemented for AreaLights. In random sampling, the amount is directly specified in the input json (for example 100 samples), and uses a random generator to sample the AreaLight. In jittered sampling, however, the amount of samples specified in the input json is the square root of the total samples (e.g. 10 in the json means 100 samples). This allows easy generation of a grid over the AreaLight, from which the jittered samples are then generated.

In the case of a point light, the attenuation of light is calculated in terms of the distance to the light. With area lights, the attenuation is calculated using the distance to the *nearest* point to the plane.

Figure 5 shows the difference between a random AreaLight, a jittered AreaLight, and a PointLight on the same scene. The AreaLights are the same size, and the PointLight is located where the midpoint of the AreaLight is.

Although I was not able to do the full sampler comparison, we can visually see some visual differences. Jittered shadows are more uniform, although it is less noticeable on higher sample counts (the pictures have a sample count of 400, which is high enough that there are no visible "faults" caused by skewed random sampling). Furthermore, the randomly sampled shadows are grainier.

Random sampling is overall a little faster than jittered, however, the difference is very small. Both, however, grow linearly with the number of samples. This can be seen in figure **??**, where 4 sample sizes have been averaged over 5 render runtimes per sample size (on the scene in demo.json).

In the implementation of the area light (and in the attempts for thin lens), the PBRT book[2] was the main resource consulted.

6.2 Thin Lens

An attempt was made at implementing a thin lens camera, but it was unsuccessful. In the render function, the thin lens camera has an additional loop that iterates over the amount of samples and averages the result. The `initRay()` function that initialises each ray, has not been successfully implemented, which is why thin lens has not been implemented.

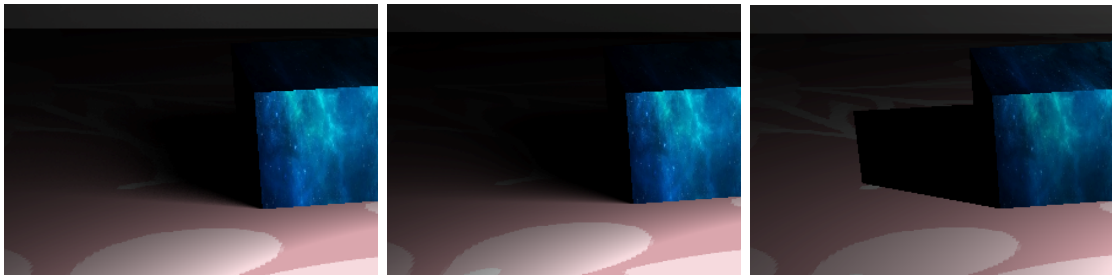

Figure 4: Comparison of shadows produced by different lighting. In order: random, jittered, point light.

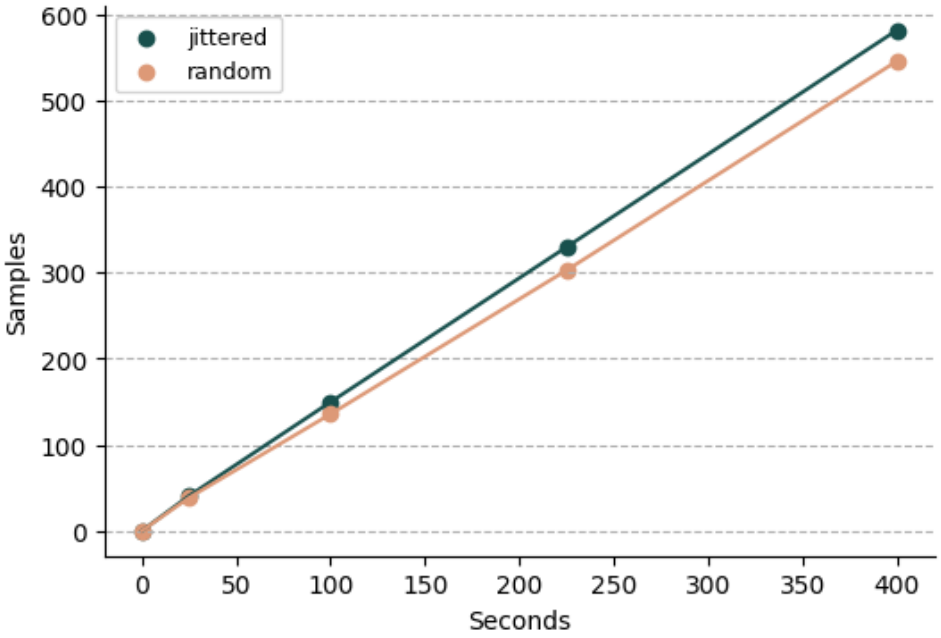

Figure 5: The speed of random and jittered sampling on AreaLights.

References

[1] Hamzah Sulaiman, Abdullah Bade Bounding Volume Hierarchies for Collision Detection, Computer Graphics. *InTech*. 2012. ISBN: 978-953-51-0455-1. `http://www.intechopen.com/books/computer-graphics/bounding-volume-hierarchies-for-collision-detection`

[2] Matt Pharr, Wenzel Jakob and Greg Humphreys Physically based rendering: From theory to implementation. *Morgan Kaufmann*. 2016. `https://www.pbr-book.org/`