Computer Graphics Course Work 2
RayTracer

Implementation

Pinhole Camera

In order to implement the pinhole camera I followed the PBRT book's samples, refactoring and 'slimming' down the code where I could due to the slightly less generalised requirements.

This also involved implementing the Transform class as well in order to make dealing with transformations through out the ray-tracer considerably neater. This transformation class essentially acts as a wrapper for the Matrix44 class provided.

With only the pinhole camera implemented, the only image I was able to produce was a black square as can be seen in Figure 1.

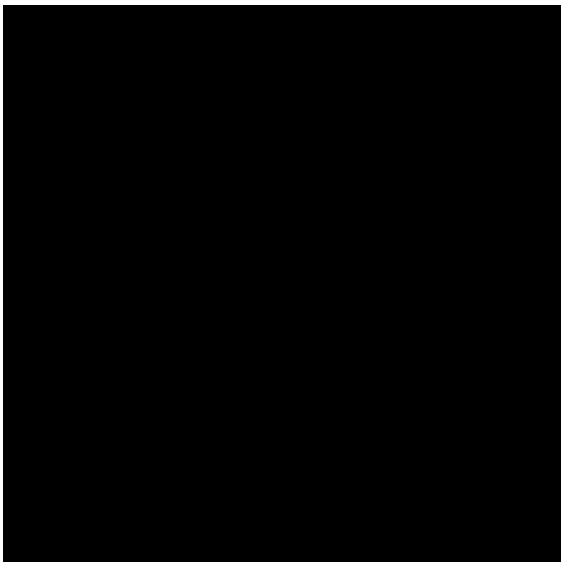

Figure 1 Pinhole Camera

Shape Intersections

For the sphere and triangle intersection tests I again used the algorithms detailed in the PBRT book as they were accurate and efficient, though missed out some important details such as how to find the normal vectors. I solved the issue with the normal: for the sphere by using the object space coordinate of the sphere hit as direction vector and normalising it as the sphere is centered at the origin in it's object space. For the triangles I simply chose two of the corners, found the vectors from the hit to them and used their (normalised) cross product.

For the plane instead of trying to find a plane specific algorithm I decided to trust that the planes would indeed always be planar and built a check into the constructor to ensure I had placed diagonal corners in specific variables and then used the four corners to build two triangles.

Intersection was then passed to the two triangles though the normal was found using cross product the vectors from one corner to two others instead of to the hit point.

The UV coordinates of the hit were found by finding the vector from the chosen 'top left' corner of the plane to the hit. This vector was then scalar projected onto the vectors from that same corner to the 'top right' and 'bottom left' corners and divided by the length of those basis vectors.

At this point I could run the ray tracer, hit shapes and get their colour, as such I was able to produce a flat shaded image with no textures as can be seen from Figure 2.

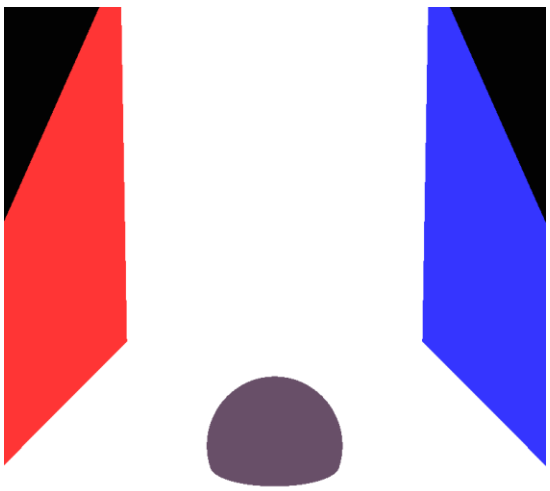

Figure 2 Shape-Intersection

Hit Structure

As a note, for the hit structure I decided to have it store a reference to the shape it hit in order to ease retrieving that shape's material.

Point-light

For the point-light the sampleLight function was adjusted to take in a hit structure in order to have it handle shadow detection as well. This function would cast a ray from the hit point to the point light and if it found a valid hit (with a t value above a threshold to catch re-hitting the shape being shaded) then that point was in shadow and the 0 vector would be returned. If no valid hit was found then the point was not in shadow and the light power was returned.

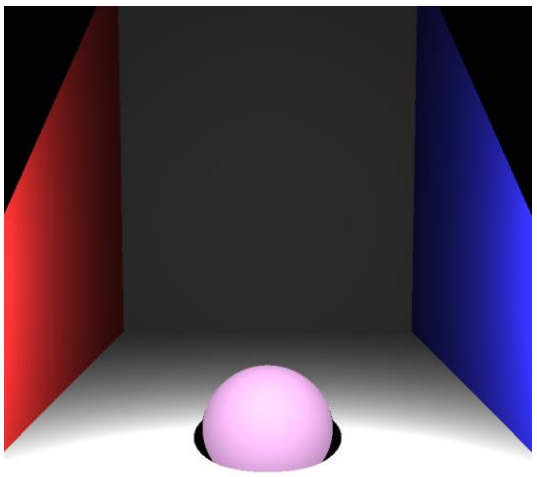

Figure 3 Point Light Shading

The light power was calculated using the theory I remembered from physics classes and basic problem solving. The power is spread evenly as the distance from the point light increases. We can imagine this as an invisible sphere centered at the light with a radius of the distance to the light from the hit point, all points on the surface of this sphere receive the same energy, and as such we can find the energy at any point by dividing the light's output by the surface area of that sphere.

I was now able to produce an image an image with basic shading as can be seen from Figure 3.

Blinn-Phong shading

In order to implement blinn-phong shading I returned to the lecture slides and implemented the equation I found there, though I ran into many issues with it and while researching them found a resource that explained blinn-phong using the half vector which worked for me.

Source: https://paroj.github.io/gltut/Illumination/Tut11%20BlinnPhong%20Model.html, Blinn-Phong reference

After having implemented these though I found some strange artifacting in my image, in order to fix these issues I had to use tone mapping. After a lot of research I found myself unable to find a method I was able to implement, as such I came up with two methods:

1. Find the larges component of any colour and scale every pixel down by that value to ensure the largest value is 1.
2. Clamp every value to be between 0 and 1.

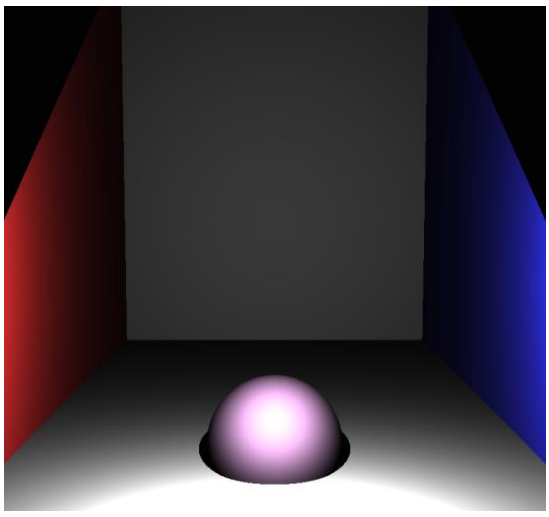

Figure 4 Blinn-Phong Shading

Method 1 worked well and preserved colour but resulted in a very dark image. Method 2 resulted in a well lit image, though bright areas shifted towards white.

I decided to use method 2 as I felt it produced a better image and more closely resembles the real physics of a camera, with the pixel colour 'buckets' filling up and throwing away any excess, resulting in bright areas shifting towards white.

The output for method 1 and two can be seen compared for the same scene in Figure 4

Reflections

Implementing reflections was relatively simple, after finding the blinn-phong component of the shading, a reflected ray was found and cast (again ensuring no immediate self-intersection) and the shading of that hit was found (recursively). To ensure this wouldn't happen ad-infinitem I added a parameter that counts how many bounces are left which would be decremented when making a recursive call, if this parameter was 0, no reflection ray would be cast or shaded.

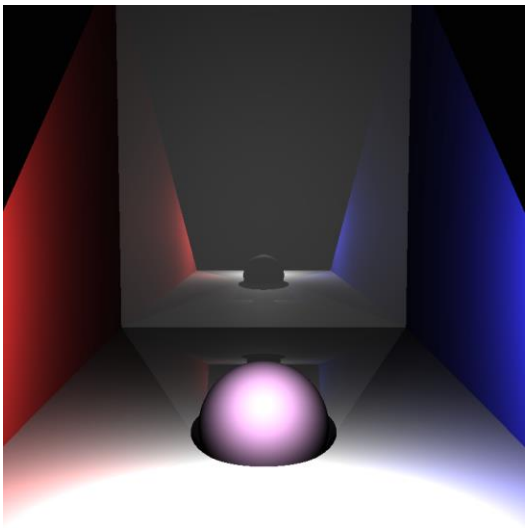

Figure 5 Reflections

The contribution of this was then scaled by the material's kr value and added to the shading.

Additionally, if the hit material had no kr value, then no reflection ray was cast.

The equation for finding the direction of the reflection ray was found online.

Source: https://math.stackexchange.com/questions/13261/how-to-get-a-reflection-vector, How to find reflected direction of vector using incoming direction and surface normal.

The result (with 3 bounces) can be seen in Figure 5.

Texture Mapping

To implement texture mapping, I would check if the material had a texture, if it did I would then use the hit UV coordinate to find the pixel from the texture to sample and used it's colour data instead of the diffuse colour for the blinn-phong shading.

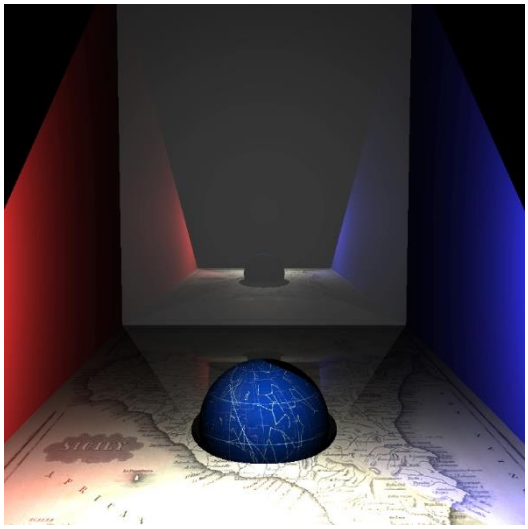

Figure 6 Texture Mapping

In order to read textures I used the stb_image library, this is a header file that lets you read and access colour data from many image formats.

Source: https://github.com/nothings/stb/blob/master/stb_image.h, header file library that allows you to read images and access their pixel data.

To speed up texture sampling, I decided to read all the pixel data into a pixel buffer of Vec3f when the material is constructed. This meant getting the pixel colour is simply a memory access and not a library call resulting in a memory access (and potentially file read).

You can see the result Below in Figure 6.

Refraction

In order to implement refraction, I would check if the material of the object hit had a refractive index. If so then I would use the normal, incoming direction and hit point to produce a refracted ray, and would intersect that only with the same object that was hit, performing this operation once again to get the refracted ray after it had exited to object and would then cast that ray back into the environment to get a colour sample.

For this I heavily used the source below to calculate the refracted vector.

Source: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel, breakdown of how to calculate refracted vectors.

Here, in Figure 7, is the result

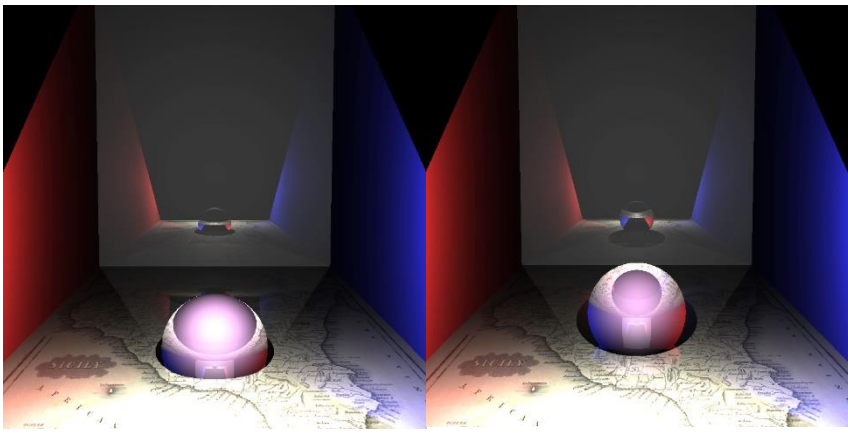

Figure 7 Two Examples of Refraction

Bounding Volume Hierarchy

For the BVH I again used the PBRT book as a reference for the algorithm and a good approach for it. I also decided to use the SAH (surface area heuristic) described in the book as the partitioning method.

I also implemented the book's Bounds3f class due to its guaranteed correctness and efficiency for checking bound intersection.

Once the partitions had been found though and the hierarchy created I decided to simply use the tree (rather than flatten it as the book suggests). This decision was made due to the time constraints of the coursework.

This meant that I had to come up with testing intersections in the BVH myself. My method was to, for each node of the hierarchy, if internal: check if the node's bounds are intersected, if so recursively call intersect on it's children, then take the resulting Hit structures and return the one that is valid, if both valid, then the one with the lowest t value. If the node is a leaf then run through its list of shapes checking for intersection and again return the valid hit with the lowest t value or an invalid hit if none were valid.

For the basic scene using a BVH had no noticeable impact, this makes sense due to the extra overhead from calls and the low number of object intersections. However once triangle meshes were added, using a BVH became essentially mandatory, this is expanded upon in the triangle meshes section.

There was no visible change in produced image and so there is no figure.

Triangle Meshes

To start with I found a library called hapPly which was able to read and return ply model information, including it's vertexes, vertex normal, vertex texture coordinates and the vertexes that correspond with each face.

Source: https://github.com/nmwsharp/happly, header file library for reading .Ply files

Once I had all the model data in I needed to make some small changes to the Triangle shape in order to use it effectively. First I added normal and UV coordinate variables for each vertex, and added a name variable for differentiating pure triangles from mesh faces and ensured these would be passed in when the triangles are created. Then I needed to implement Triangle texture mapping, and returning the correct normal, for both of then I simply return the weighted sum of each vertexes normal or UV, weighted using the barycentric coordinate of the hit point in the triangle.

Initially my models looked strange, with missing faces, this was because each face I was getting held 4 vertexes, expecting me to create 2 triangles, while I was only expecting 3 vertexes, so I added a check and this was fixed.

You can see the stages of correctly implementing these in Figure 8

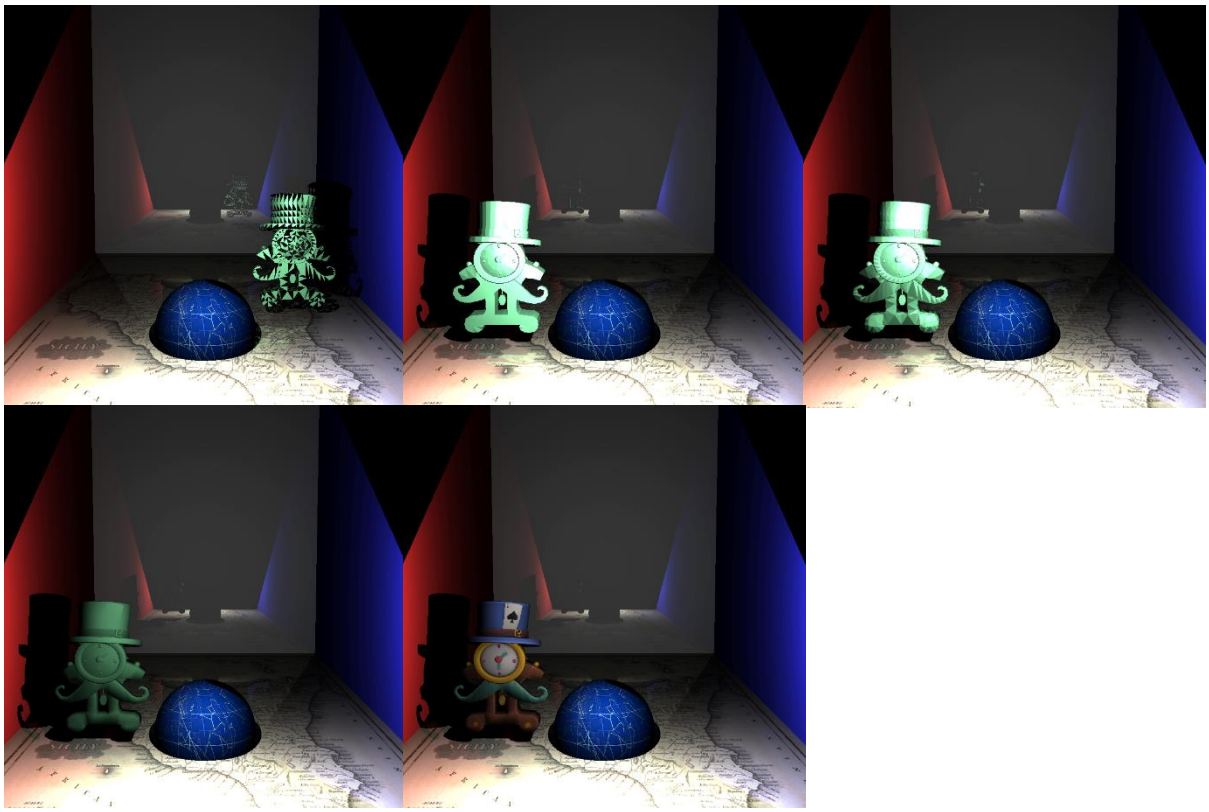

Figure 8 Trimesh development: Incorrect face parsing; correct face parsing; averaging vertex normals; using barycentric coordinates; texture mapping. Source: https://www.cgtrader.com/free-3d-models/furniture/cabinet/stylized-magicians-clock, model

For basic intersection I could have run through all the triangles in the mesh and returned the valid hit with the lowest t value (or no hit) similarly to how I checked for plane intersection. However this resulted in unusable performance, as such I decided to use a BVH inside the triangle meshes as well.

Once all the faces had been created, a BVH was created using them and all intersection testing was handled by them.

Distributed Raytracing

Area-Light

For the area light I decided to make some assumptions to make my life simpler. As allowing any shape would have required a way to accurately spread rays across the surface of the shape, I decided to assume all area lights were planes. This meant I could find basis vectors for the plane and use them to generate points on the area light and therefore more easily cast rays to specific parts of the light.

For the sampleLight function I used the same power function as the point light as integrating the power over the entire area would have taken too long to implement. However this time I cast multiple rays over the area, counting all the rays that made it to the area light un obstructed, then divided that value by the number of samples, this gave me a rough estimate of the proportion of the area that was obstructed from view. This proportion (from 0 to 1) was then used to weight the power received.

Implementing jittering was similar though for each sample I would generate it's lowest value and would generate a random addition to that base point, limiting its values to ensure the generated point would stay within it's section of the area.

Figure 9 holds some examples.

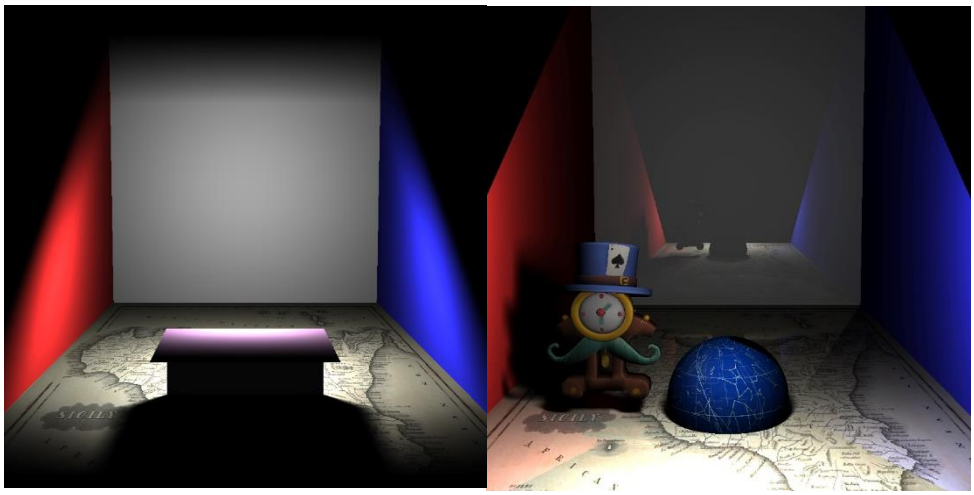

Figure 9 Area Light examples

Thin-Lens Camera

I used the PBRT book to implement the Thin-Lens method of generating rays, which differed very little from the pinhole implementation.

To produce a pixel value, I would generate random points on the lens for the ray to pass through casting rays for each of those, then averaging the resulting pixel values.

These points were generated by complete random in $[0, 1]^2$ or jittered in $[0, 1]^2$ and then passed through a function that mapped them to the until circle.

Source: http://psgraphics.blogspot.com/2015/04/sampling-disk-revisited.html, source for $[0,1]^2$ -> until circle mapping function.

Figure 10 holds some examples

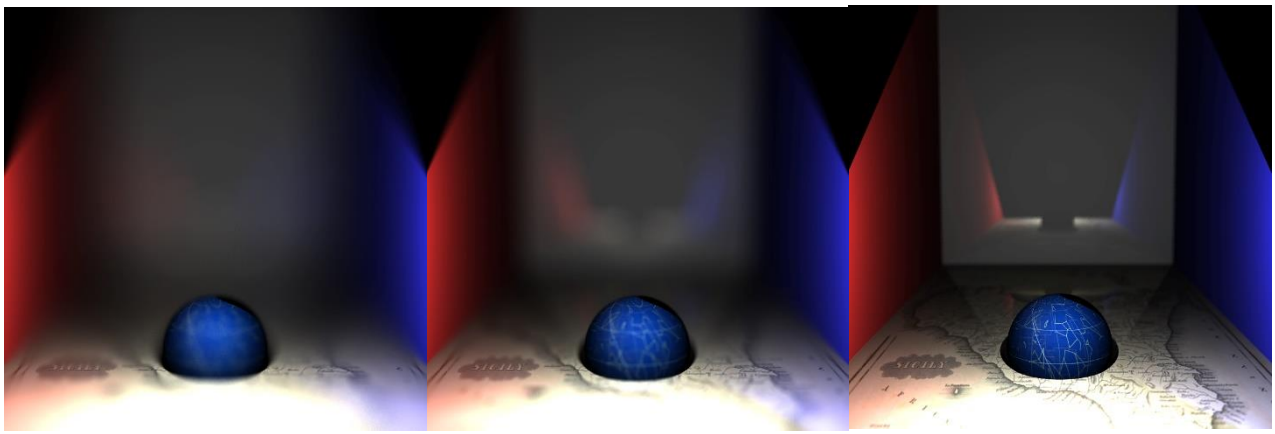

Figure 10 Thin Lens; large lens radius; medium lens radius; small lens radius;

Creative Sample

Please find below in Figure 11, my own scenes:

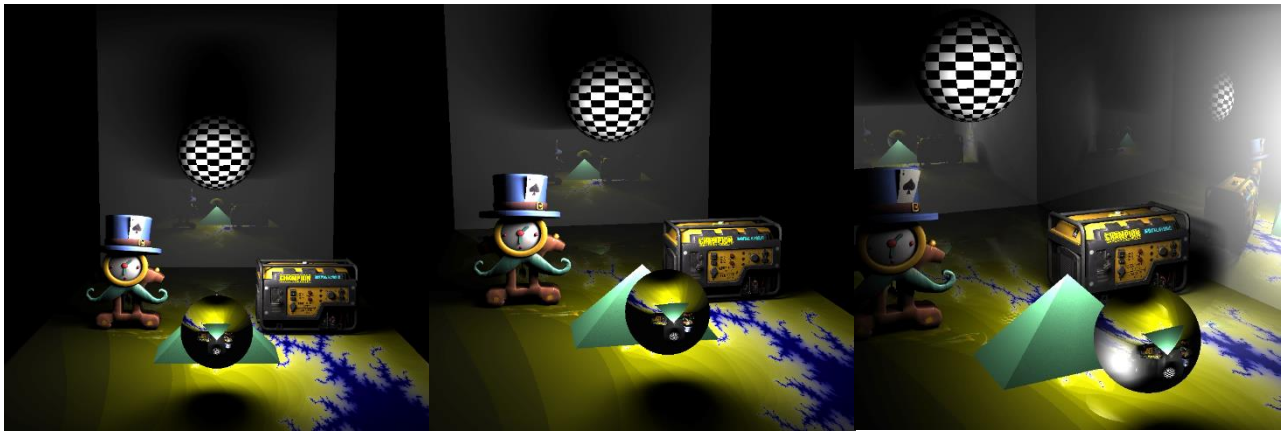

Figure 11 Custom Scenes

I used GIMP to produce the checkerboard and fractal textures and found the generator ply file online.

Source: https://www.cgtrader.com/free-3d-models/industrial/tool/lp-generator-realistic-3d-model-ready-for-animation-and-game, Generator ply model

Assessment

BVH

Unfortunately due to the time constraints of this coursework I was unable to set up experiments to evaluate the speed up gained from using BVHs. However I did find that without using BVH for Triangle meshes, my renders would never complete. Whereas when I did use them, they completed in around 30 seconds if no distributed rendering techniques were used, and around a minute when they were (at 100 samples).

Please note that I never used both distributed rendering techniques at the same time, as this resulted in extremely long rendering times, e.g. both area-light and thin-lens using 100 samples would mean at worst a pixel could require 30300 ray casts (if every ray hit a reflective surface, that bounced 3 times).

Area-Light

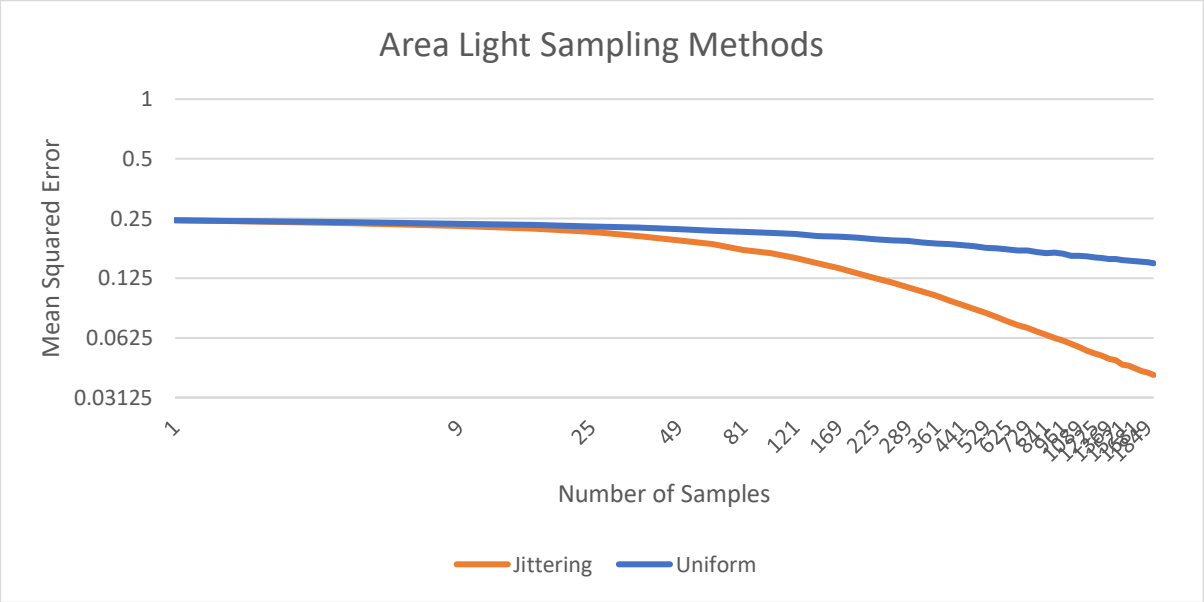

Figure 12 Graph of Area Light RGB mean squared error of sampling methods at different sampling numbers

Here we can see that both jittering and uniform random sampling start off with very similar errors, however we can clearly tell that the Jittering method begins to converge considerably faster than the uniform method does.

Visually, the jittering sampling method also looks less ‘grainy’ faster.

Thin-Lens Camera

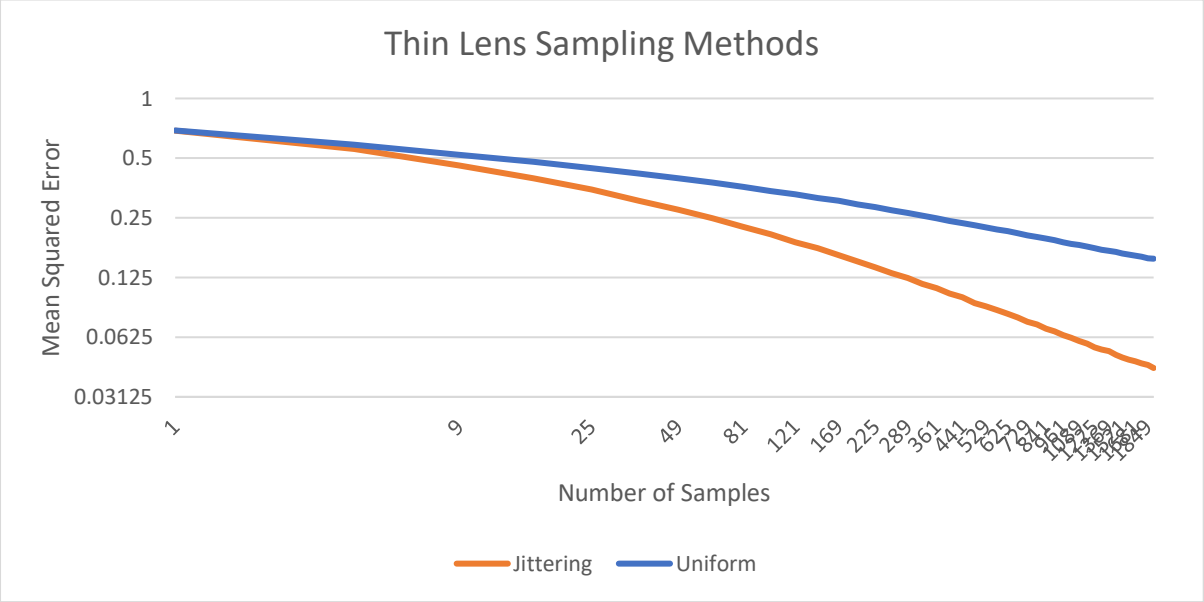

Figure 13 Graph of Area Light RGB mean squared error of sampling methods at different sampling numbers

Again we can see that both jittering and uniform random sampling start off with very similar errors, though these errors start off larger than for the area light, and we can clearly tell that the Jittering method begins to converge considerably faster than the uniform method does.

Again, visually, the jittering sampling method also looks less ‘grainy’ faster.