Ray Tracing Report

You run the ray tracer through a terminal giving it two arguments. The first argument is the json of the scene you would like to render. The second argument is the location and name of the .ppm image file output (you must include .ppm in the output name, e.g. output.ppm).

The ray tracer reads the json, firstly in the main class it reads

{“nbounces": int,

which is how many times the light ray will bounce. Then it calls the camera class, where it fills out the camera information.

"camera":{},

Then scene class has methods that fills out a scene object. It respectively fills background colour, a lights vector, and finally a shapes vector.

```
"scene":{
   "backgroundcolor":[ 0 to 1, 0 to 1 , 0 to 1 ],
   "lightsources":[] ,
   "shapes":[]
}}
```

I will cover the json format for the camera, light sources and the shapes when it comes up in the report. It is important to note that colour is represented by rbg where the value 0 to 1 corresponds to 0 to 255.

Basic Ray Tracing

Ray Casting

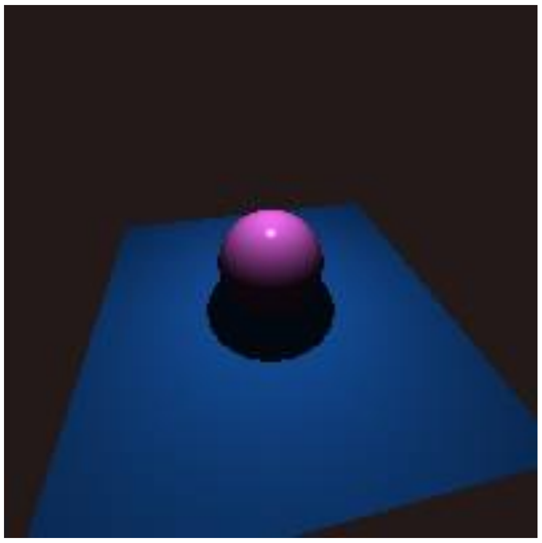

Figure 1. Simple Ray Trace - BallandPlane.json

The ray tracing occurs in the **RayTracer** class, called by the **render()** method. This method declares an output array of the program. For every pixel in the output it calls the camera method

getOriginDirection() which gives a vector for the origin of the ray and the direction. The ray is cast by the method **castRay()** which will return a pixel value for the outputted image.

Render() handles the different cases for thin lens and pin hole as well as the different samplings for thin lens. For thin lens there are more than one ray cast per pixel. It will average the colour of the pixel depending on how many samples are sent.

There exists a structure **Ray** in **RayHitStructs** which contains information about the ray cast. It contains its type **PRIMARY, SECONDARY, SHADOW,** as well as the origin and direction.

CastRay() when first called will cast a **PRIMARY** type ray. This first is set to the background colour in case that the ray hits nothing. The ray is traced with the **traceRay**() method. For **PRIMARY** and **SECONDARY** rays checks if the ray intersects with all the shapes and all the area lights. Return the shape or light that intersects and is the closest. It returns a **Hit** structure.

Hit structure is also declared in **RayHitStruct**, this structure contains a bool value if the ray hit any shape or area light, and a distance from the camera. It also contains information about the hit point. It has the actual hit point in the world space, surface Normal, the view Direction of camera from hit point, colour of the hit point, the location in shape or lights vector, as well as a Boolean if the shape has a texture as well as the texture colour.

CastRay() returns the background colour if the **PRIMARY** ray hits nothing or a vector of zeros if **SECONDARY** ray hits nothing. If the hits an area light it returns the colour of the area light. If an intersection was detected it, traces the rays that go from the hit point to all the lights with **traceRay()**. It checks for shape intersections, if there isn't any then there is clear line of sight from light to the hit point. It calculates the colour based on the material properties of Blinn-Phong and the light. Summing up for every light in the scene. If a shape is intersected that means for that specific light the object is in shadow, so it returns only the ambient colour of the object.

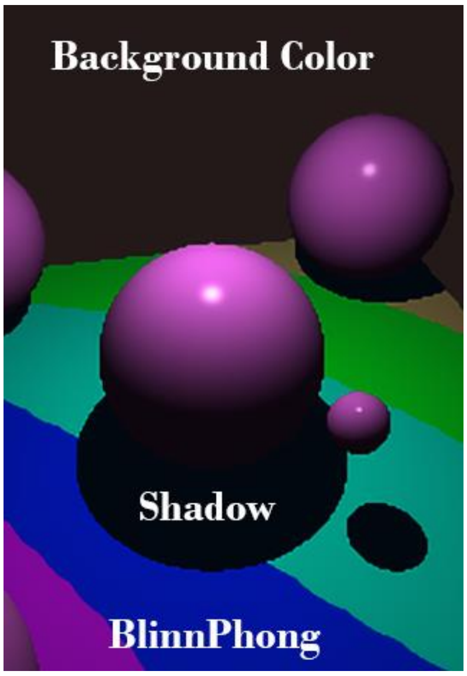

Figure 2. Hit point colour return

If the number of bounces is less than specified number of bounces then the reflection ray is calculated and **castRay()** is called again for the secondary ray. The returned colour is then summed to the parent **castRay()** call and that is return. This results in a rendered image.

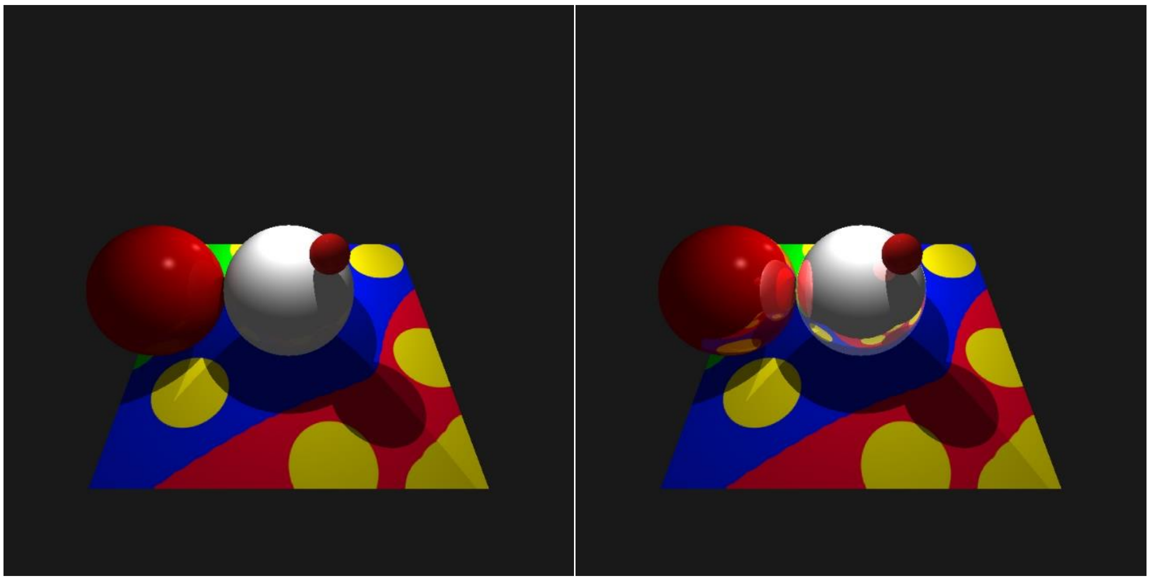

Figure 3. 3 recursive light bounces
smaller reflective constant on left, larger on right – lightbounce.json

For the rest of the experiments I used nbounce of 1 to speed up render time.

<u>Camera</u>

```
{"type":"pinhole",
 "width": int,
 "height": int,
 "fov": int,
 "location": [float, float, float],
 "look":[float, float, float]
}
```

The basic camera is a pinhole, the JSON must be in this format. The width and height describing the image size output, the fov is the field of view and the location and look are the world coordinates of the camera. Where the camera is and where it is looking at.

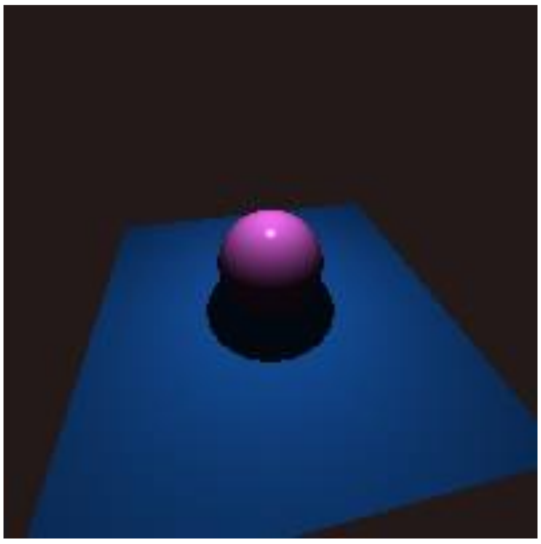

Figure 4. 90 FOV - BallandPlane.json

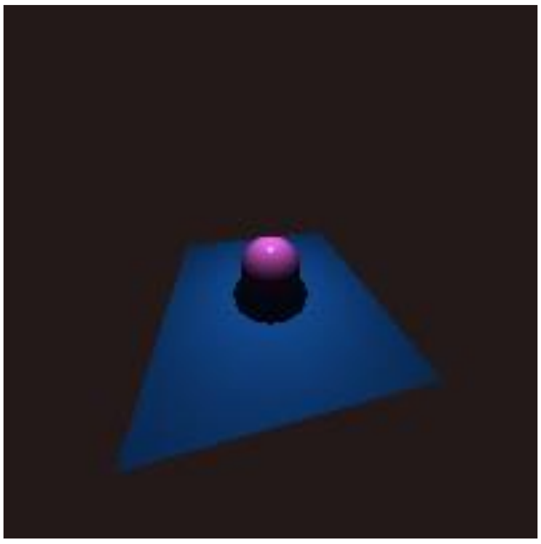

Figure 5. 120 FOV - BallandPlane.json

The way the pinhole works is that the image plane is behind the pinhole in the direction of the camera location from the look at point. The corresponding pixel is the origin of the ray and the direction is through the pinhole. The method **getOriginDirection()** converts the camera origin which is pixel location and -1 distance from pinhole, into the corresponding world location. Using the cameraToWorld matrix that is defined in the camera declaration, which depends on the camera coordinates. The new calculated ray can be used to ray cast now.

Point Light

```
{
"type":"pointlight",
"intensity": float,
"position": [float, float, float],
"color":[0 to 1, 0 to 1 , 0 to 1]
 }
```

The basic LightSource is a point light, represented by a position in world space, a colour and a float for intensity. This information is used in Blinn-Phong.

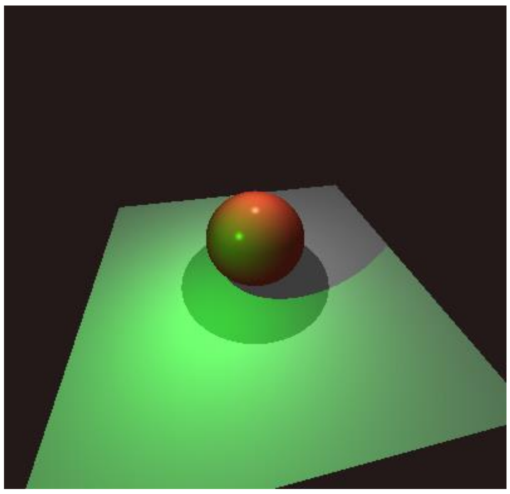
Figure 6. green, high intensity - pointlight1.json

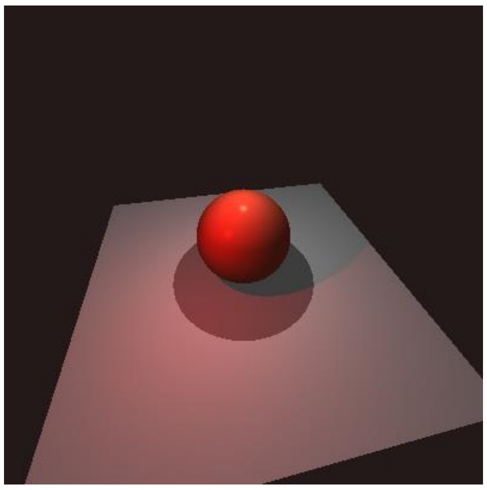
Figure 7. Red, lower intensity - pointlight2.json

Shapes

All shapes in the JSON are declared with a material property for Blinn-Phong. Which describes its materials interaction with light. All shapes have an **intersect()** method which determines if a ray vector intersects the shape and return the hit point. It is in that method where the Hit structure is filled out. The intersect method determines the distance of the intersection. Which I use to sum the ray origin with the ray direction that has been multiplied by the distance to find the hit point.

Sphere

```
{
    "type":"sphere",
    "center": [float, float, float],
    "radius":4,
    "material":{}
```

}

The centre describes its location in world space. The radius is how big the sphere is.

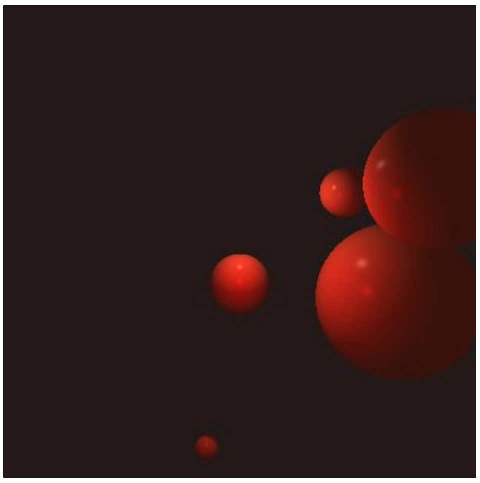
Figure 8. many spheres - sphere.json

To calculate whether the sphere is intersected by a ray, I used a quadratic solution to find the far and close intersection distance. Picking the closer intersection as the hit point distance.

Planar Quad

```
{
    "type":"plane",
    "points": [[float, float, float], [float, float, float], [float, float, float], [float, float, float]],
    "normal": "optional" -1,
    "material":{}
}
```

The quad is defined by 4 world space points and the respective normal. It is up to the user to determine the 4 points are on the same plane and the normal corresponds to the correct normal. The first point is the bottom left, the second is the top left, the third is the top right and the final point is the bottom right of the quad.

Intersection is determined by finding the point hit and then testing if the point is within the horizontal and vertical lines of the quad. Using the dot product on the surface normal and ray direction, we can determine which side of the plane is intersected. This allows us to set a bool to keep track, and if it is the back face it won't be cast in light.

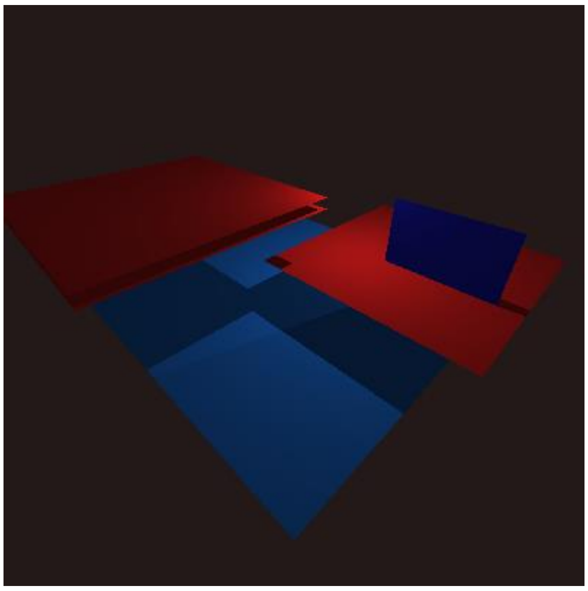
Figure 9. many quad planes - plane.json

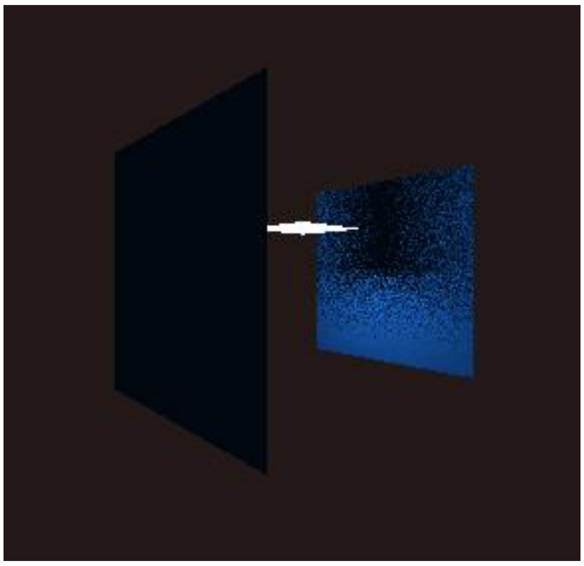
Figure 10. Plane face correct lighting – backfaceplane.json

Triangle

```
{
 "type":"triangle",
 "points": [[float, float, float], [float, float, float], [float, float, float]],
  "normal": “optional” -1,
      "material":{}
 }
```

Triangles are like planar quads. Except defined with 3 points instead of 4. The intersection is determined by finding the distance of the triangle plane and then the hit point. It then tests the hit point to see if it is inside all 3 edges. The normal input is optional and will switch the normal if the generated normal is incorrect facing.

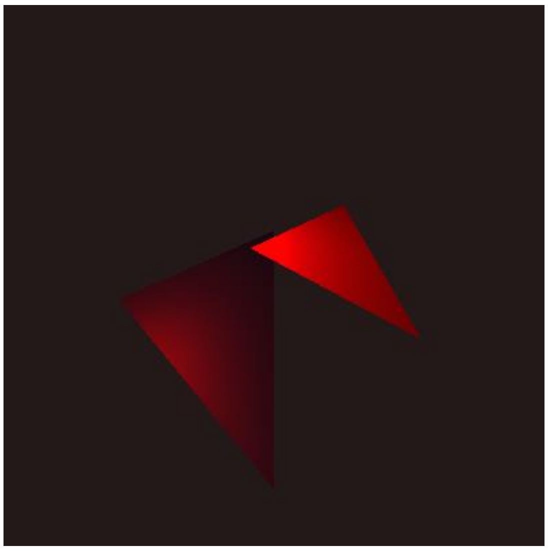
Figure 10. “many” triangles - triangle.json

Triangle Mesh

```
{
    "type":"trimesh",
    "size": int,
    "points": [
            [float, float, float]…
                ],
    "material":{}
}
```

Trimesh is declared with a size which should match how many points are in the points array. The minimum number of points must be 3 for at least for 1 triangle. The trimesh is constructed by creating triangles from the points. The first 3 points (0,1,2) are the first triangle. Points (1,2,3) become the second etc… So the number of triangles in the trimesh is the number of points - 2.

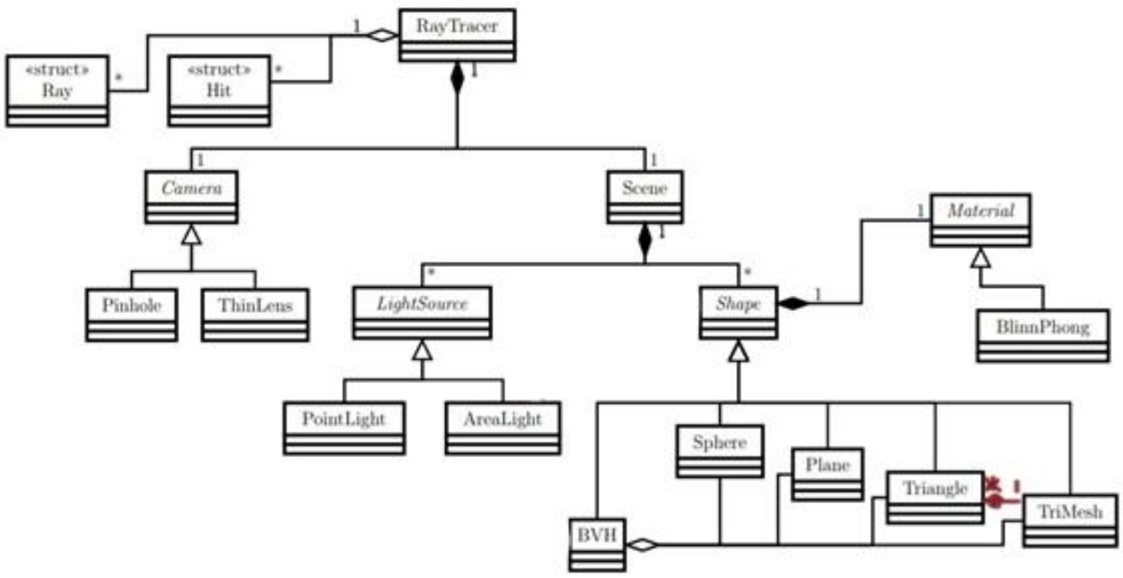

Figure 11. UML update for TRIMesh

To test intersection, the trimesh tests intersection with all its faces(triangles).

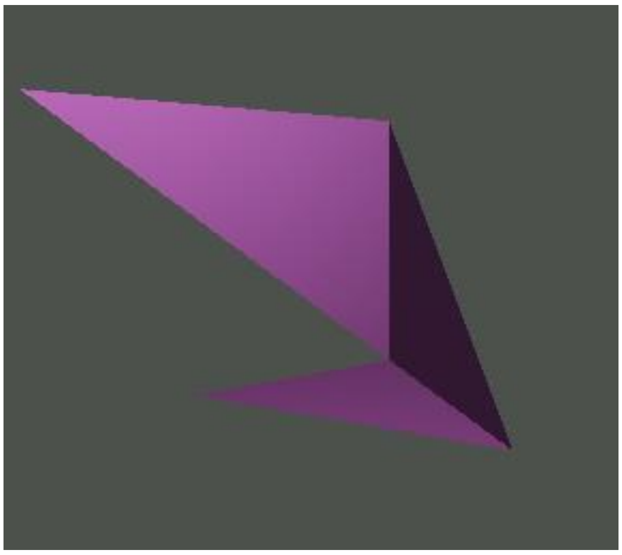
Figure 12. Trimesh with 3 triangles – TriMesh.json

Blinn-Phong

Each shape has a material property which is used for blinn-phong.

```
"material":{
        "ks": 0 to 1,
        "kd": 0 to 1,
        "specularexponent": int,
        "diffusecolor":[0 to 1, 0 to 1, 0 to 1]
}
```

Back in the ray tracer castRay, when we calculate hit point colour, we use blinn phong. If the object is visible to the light we call the methods in Blinn Phong to get ambient, diffuse and specular colour using the corresponding vectors, material properties and light colour for the algorithm. If the hit point is not visible to the light, we return the diffuse colour multiplied by a low value to get a corresponding ambient colour.

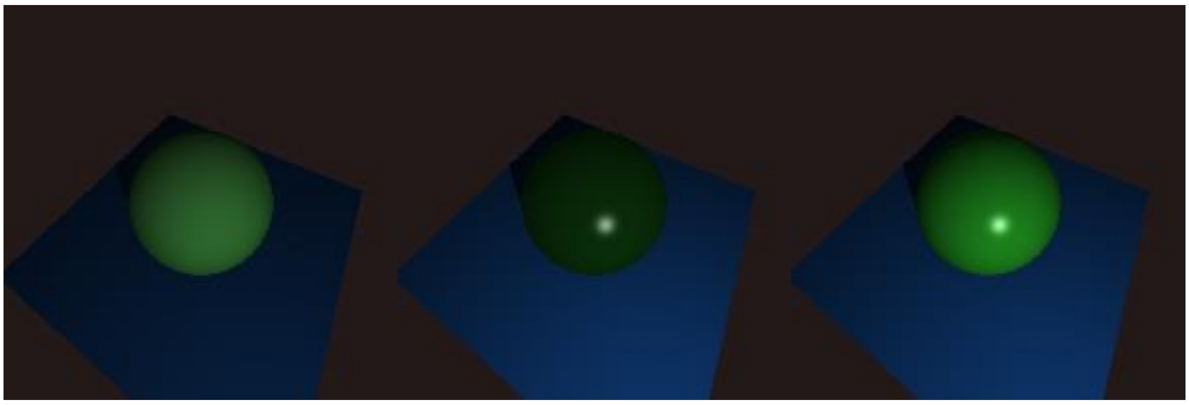
Low specular high diffuse high specular low diffuse high specular high diffuse

Figure 13. varying specular and diffuse constants – blinnphong.json

Texture Mapping

For texturing, you include **“texture": path to texture location,** In the JSON shape object. Then in **CastRay** when it is calculating **BlinnPhong** the texture colour is passed as a parameter instead of using the diffuse colour of the material.

The texture file format can be .png, .jpg and probably other formats but the other formats are not tested. This is because I am using OpenCV to open and read the file texture. I then convert the colour from 255-0 to 1-0 in the RayTracer class. To match the raytracer format. The texture is kept in an array of the texture size in the shapes class. For the program to be built you will need to install OpenCV and link it to the project, otherwise the project cannot be built. Or there will be problems atleast with texturing and PPMWriter.cpp I used the following tutorial [10]

You may also need to modify the texture location in the json if you run the attached JSON files. Some test textures are included in the JSON folder.

Sphere

Taking the hit point. I use that to calculate the surface normal. Where I apply the algorithm, which I got from online [4], to find the u (0-1) and v(0-1). I then map the u and the v to texture coordinates. where 0 is the bottom and 1 is the texture height and similarly for horizontal. This gives us the texture point. Which we work out the corresponding point in the texture array to get the correct colour. If you look at figure 13, you can see how a texture would be mapped to a sphere.

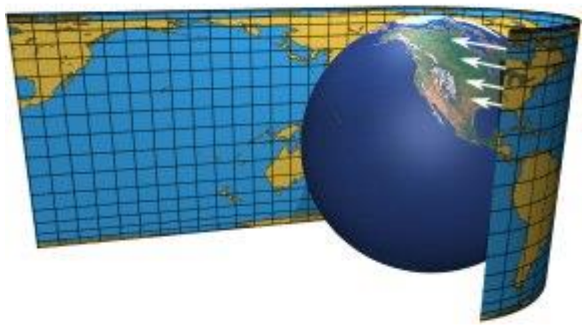

Figure 14. Texture mapping

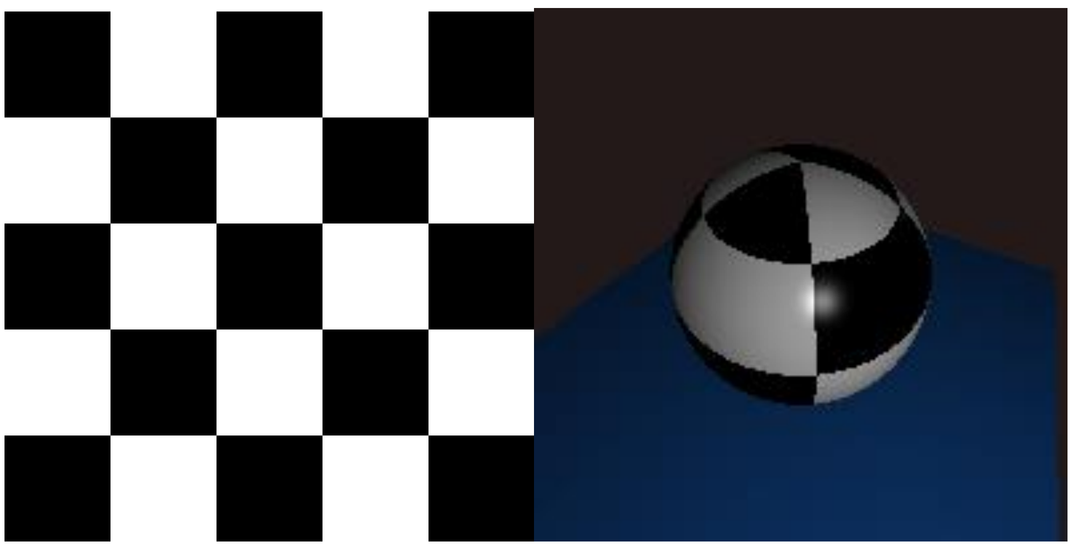

Figure 15. texture on sphere – spheretexture.json

Planar Quad

By taking the vertical vector of the plane (from point 0 to 1) projecting the hit point onto it and divide by the vertical height of the plane, we can work out how high up the image the hit point is from 0 to 1. We can do a similar thing for horizontal (from point 0 to 3) to find out the horizontal location from 0 to 1.

We then map that coordinate to the texture space, where 0 is the bottom and 1 is the texture height and similarly for horizontal. This gives us the texture point. Which we work out the corresponding point in the texture array to get the correct colour.

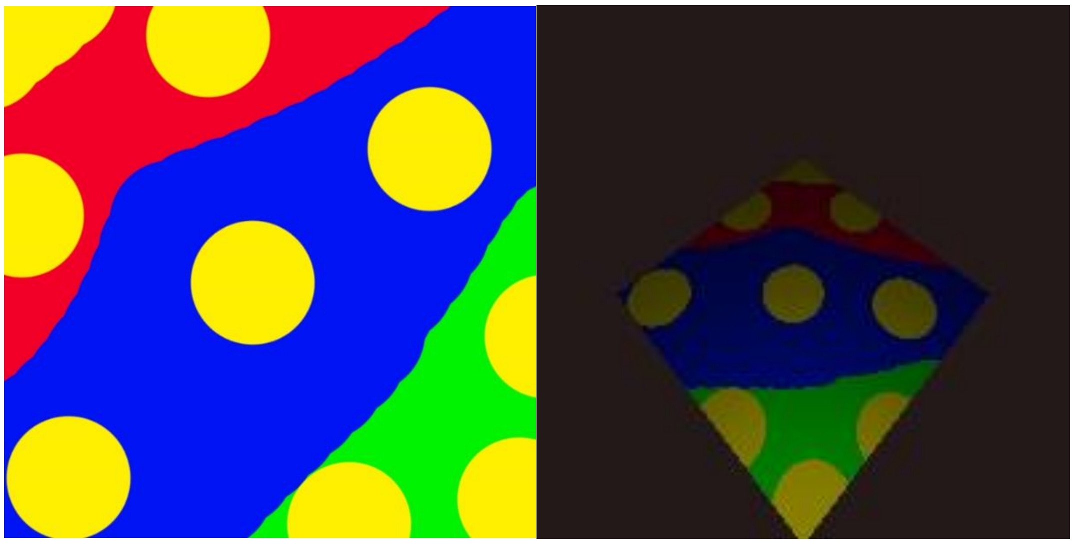
Figure 16. texture on plane – planetexture.json

Triangle

Triangle is identical to plane. There is room to optimise the texture storage but for now I treat texture retrieval identically to planar quad.

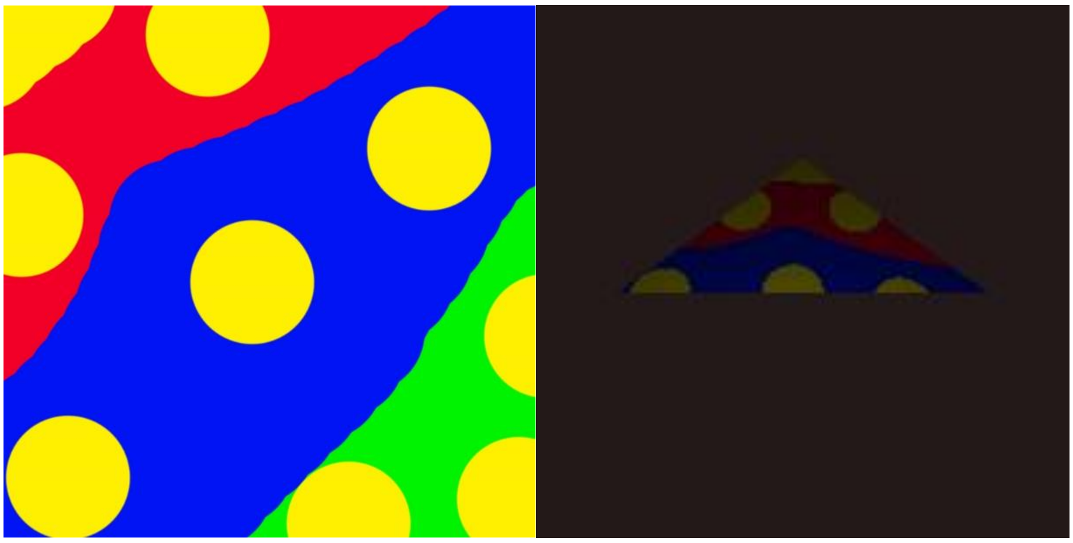

Figure 17. texture on triangle – triangletexture.json

Triangle Mesh

I haven't implemented triangle mesh texturing

BVH

I built a large majority of BVH but I never got around to integrating it. To prevent the BVH from interfering with the raytracer it has all been commented out. But if you were to read through the code you can follow my train of thought.

I added a **BBox** class to the **BVH.h** and this new class acts as the bounding box. It maintains two vec3f positions. The first is the bottom left and the second is the top right of the bounding box. It also has a method to return the center of the bounding box. It also contains a method to work out if a ray intersected the bounding box. And returns the distance to the closest point of intersection, else it returns -1.

BVH is created passing all shapes, how many and one axis. It will then proceed to develop a tree of the shapes. If only one or two shapes are given to the BVH it sets the nodes of its children to those shapes. Else it proceeds to create a bounding box by passing all points from the shapes and testing if they expand the bounding box in **BBox.ExtendBox().** For a sphere the method **sphere.GetPoint()** , finds the largest and smallest x, y and z values. It then passes those back for increasing the bounding box.

It finds the center point of the large BBox, and then using **qsplit** it orders the list of shapes into the order left of axis and right of axis and returns the point in the shapes array of the last left shape. It then creates 2 children BVH, one for all left shapes and the other for all right recursively until only 1 or 2 shapes are left in the list.

For intersection it tests if the BVH is intersected using the **BBox.Intersect()** then if it is, it tests if the children are intersected. And this will recursively search until it comes to the actual shape of intersection.

I believe most of the code is there, but it hasn't been tested and debugged, thus I commented it out.

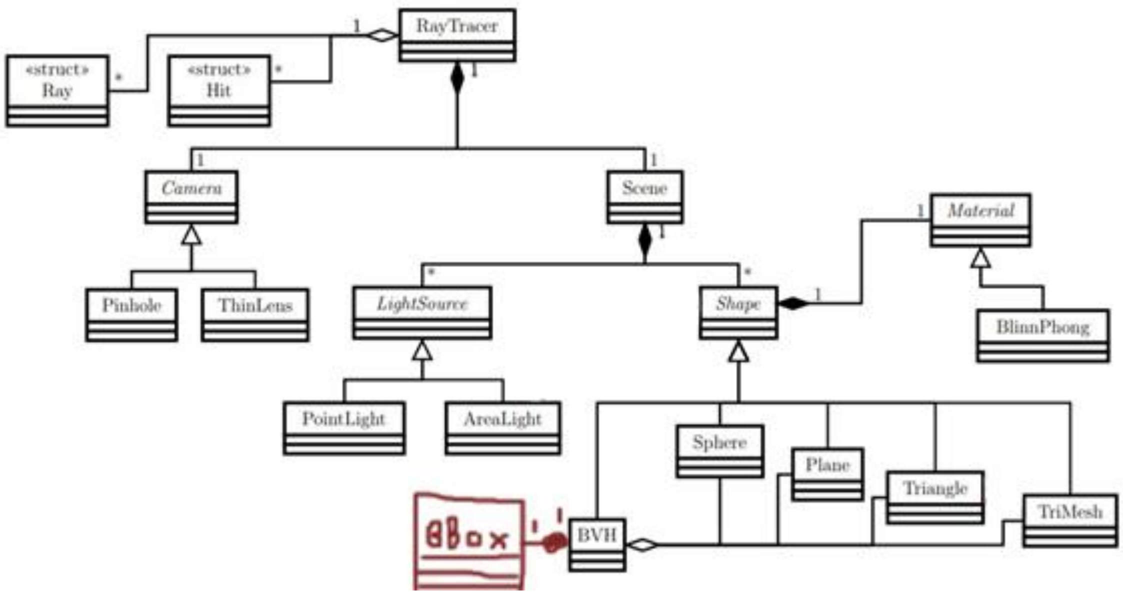

Figure 18. BBox UML

Distributed Ray Tracing

Thin lens

To declare a thin lens you define the camera JSON as such,

"camera":{
"type":"thinlens",
"width": int,
"height": int,
"fov":float,
"aperture":float,
"focal":float,
"location": [float, float, float],
"look":[float, float, float],
"sampling":"random" or “jittered”,
"nsamplesx":int,
"nsamplesy":int
}

The new elements are the aperture, focal, sampling and nsamples x and y all kept in the thin lens object. A thin lens is very similar to a pinhole except that the direction ray goes through a point on the lens which is a disk of diameter of the aperture. The lens is located at the camera location. The focal is the distance of the focus. A higher number means a further away object will be in focus.

The **getOriginDirection()** works differently to pinhole. Like pinhole it first gets the origin and direction through the camera location. Then it works out the point hit on the focus plane. This is the origin of the ray plus the direction times the focal distance. This point hit becomes the direction vector. The origin vector becomes a random point on the lens disk.

While this isn't an exact model of how thin lens work, by calculating ray refraction it functionally acts the same. We can also more easily define the focal range as we give it as a constant.

By changing the focus plane further from the camera, objects close become more blurred. By decreasing focal range further objects become clearer and closer object become more distorted.

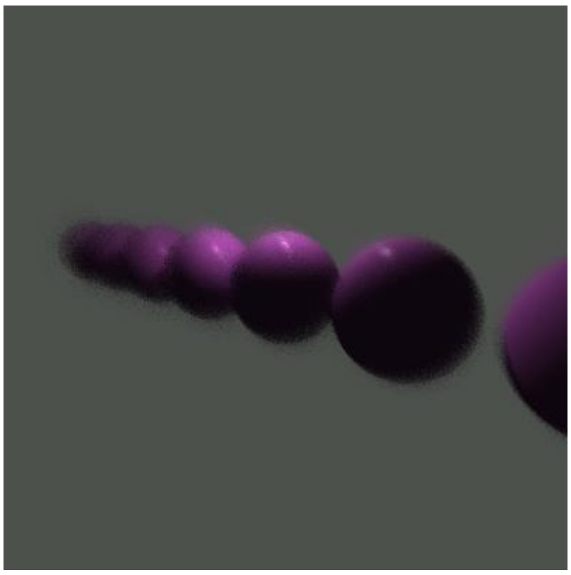
Figure 19. Focal range 10 – DoF.json

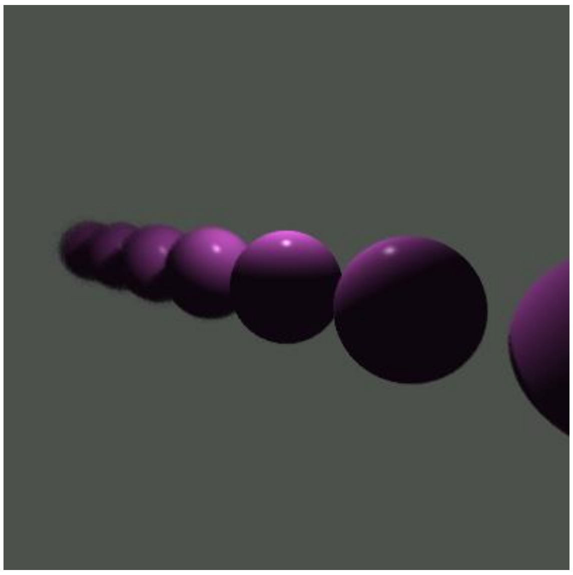
Figure 20. Focal range 20 – DoF.json

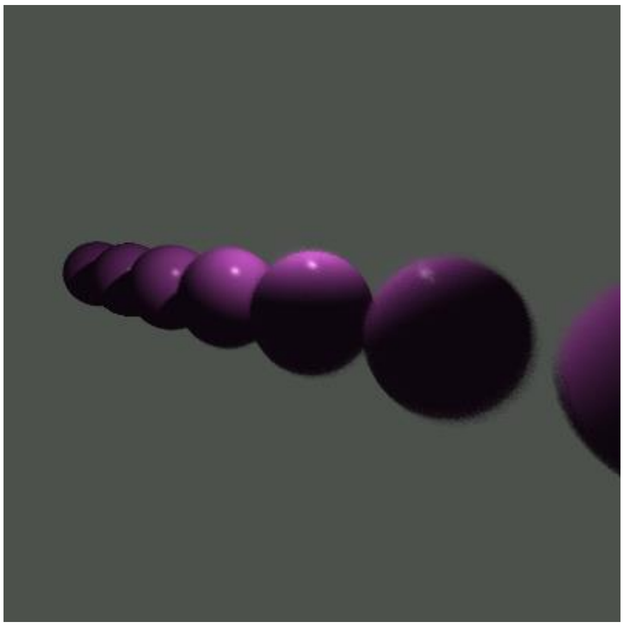
Figure 21. Focal range 40– DoF.json

By increasing the aperture, the distortion/blur increases more rapidly the further you are from the focus plane. This is due to the ray direction variation becoming larger as a result of a larger aperture.

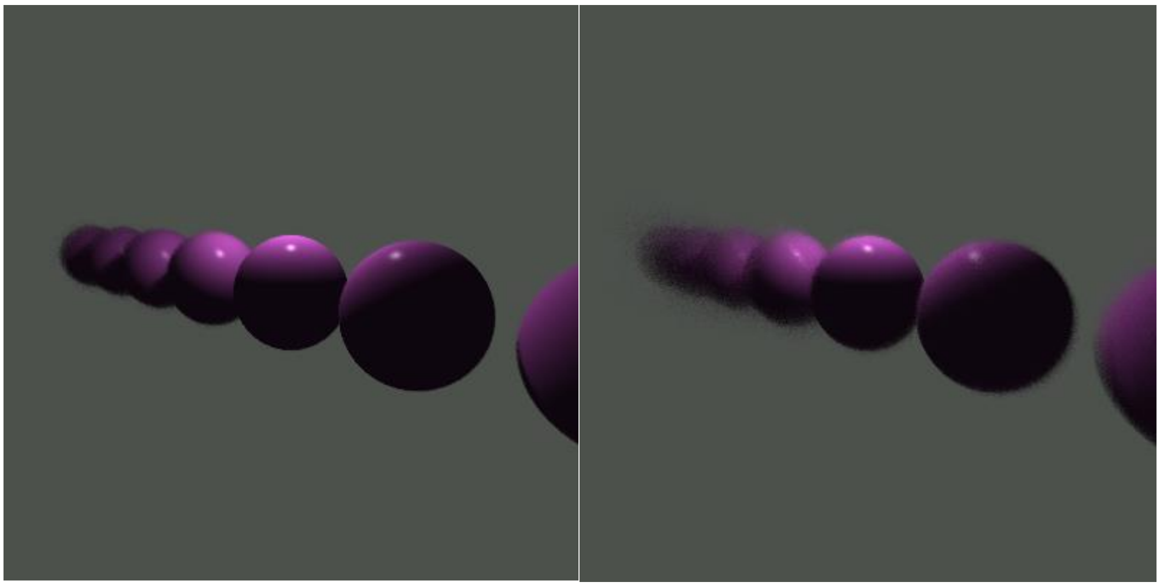
Figure 22. Focus 20 with aperture is 2 on the left and 8 on the right – DoF.json

The sampling methods change how the samples are determined, random just randomly planes an origin on the plane. Jittered, jitters the ray origin in a uniform manner every 360 / number of x samples degrees and along the radius every radius/number of y samples.

The greater the sampling the better looking the depth of field effect becomes.

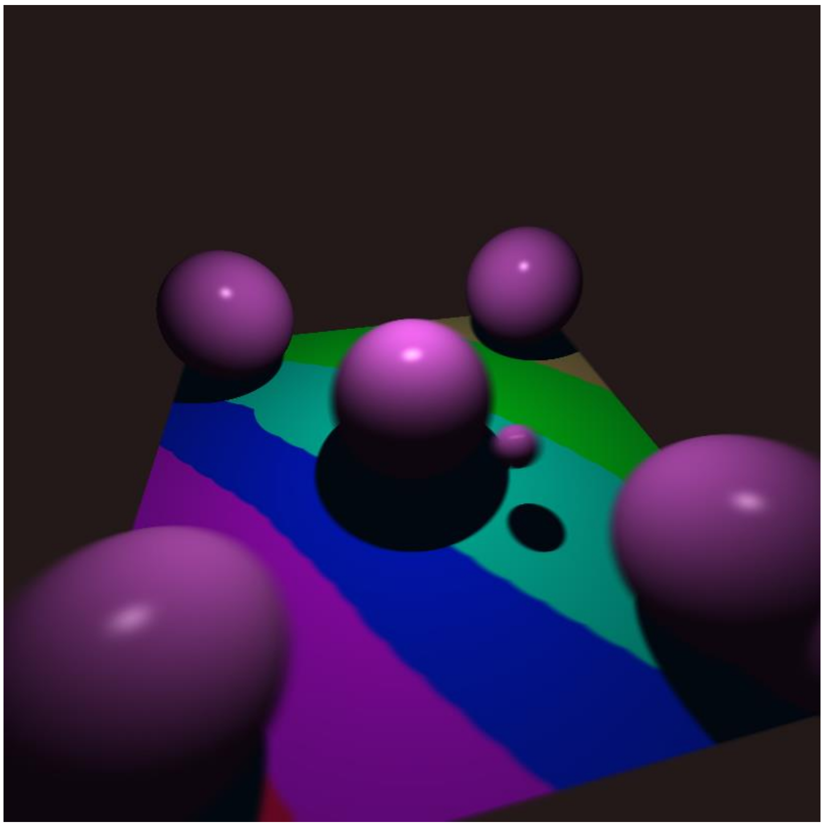
Figure 23. High sampling rate – testareajitter.json

Area light

For area lights, the Light object has a Shape as a class variable. This means that the UML looks like

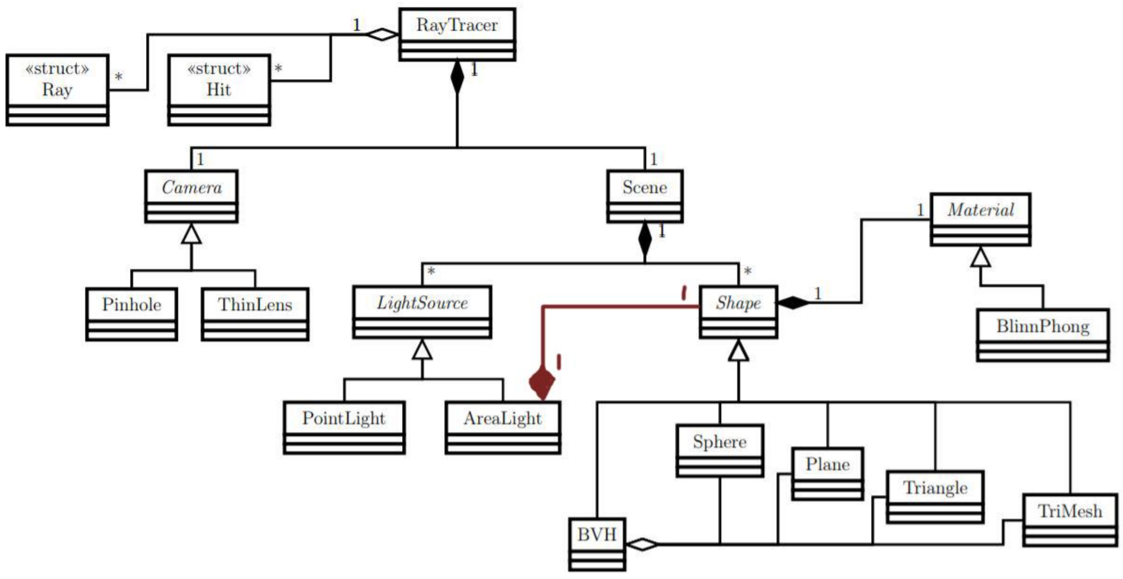

Figure 24. update UML

With the update in red. And the JSON for the area light is

```
{
      "type":"arealight",
      "intensity": float,
      "position": [float, float, float],
      "color":[0-1, 0-1, 0-1],
      "sampling":"random" or “jittered”,
      "samples": int,
      "shape : {}
}
         }
      }
```

The new items are sampling, samples and a shape. Where the shape is expecting a shape like before. I have only implemented area lights for planes. All other shapes will have weird behaviour or cause crashing.

When dealing with the area light, in **castRay()** instead of sending a ray to the light position. We send a ray to a position that is either random on the plane or jittered, using the plane methods **getRandomPosition** and **getJitteredPostition**. Random is just a random point. Jittered breaks the plane into a grid and randomly picks a point within each of the grid squares. Therefor the grid size depends on sample size. When entering “samples” if the sampling random it directly correlates to how many samples are taken of the area light. If it is jittered, then it takes the samples to the power of 2 total samples.

When calculating shadows and BlinnPhong for a hit point to an area light. It casts the corresponding samples to the area light. Then it averages for that light and hit point. It still sums over all colours returned from all lights for each hit point.

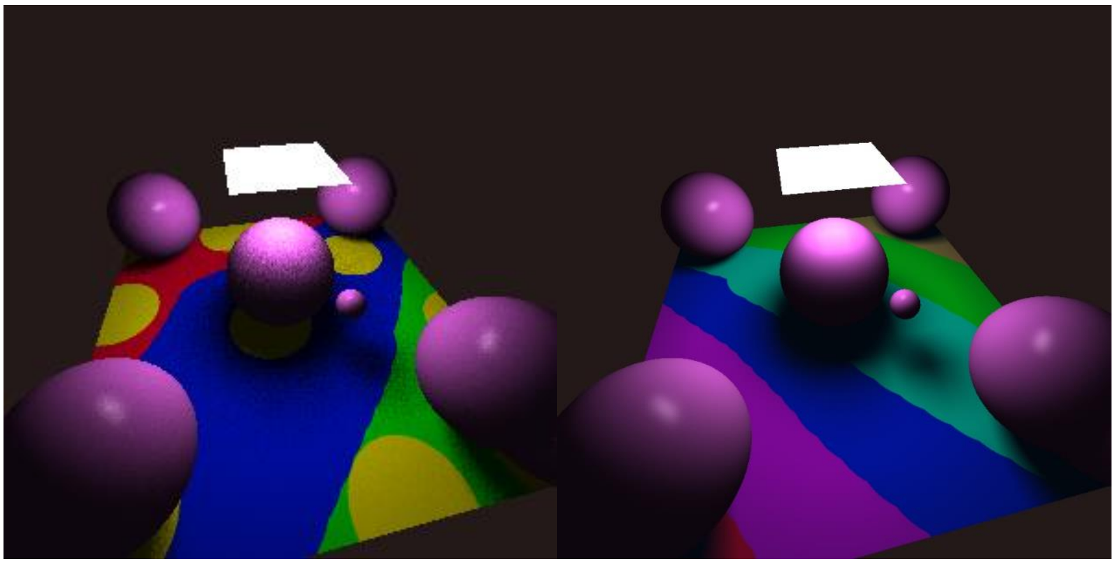
Figure 25. area light with different sample rates, 30 and 5000 – testareajitter.json

The greater the sample rate, the clearer and softer the shadows become.

To compare random sampling against jittered grid sampling for area graphs. I first ran the scene with 5000 sample rates. Then for random area scattering, I ran it with 50 sample rates and incremented it by 50 sample rates up to 2000. I then logged both the sample rate and the MSE to get the log log graph. The code to find the MSE is a for loop that is commented out in the **main.cpp**.

For jittered scattering I did the same thing, except for the sampling rate I started at 1, and then 2^2 and then 3^2 etc.. up to 45^2, which is just over 2000 samples. I recorded the results in **jitterarealight.txt** and **randomarealight.txt**, for jittered you would need to square the sampling rate to get the graph that I got below. (Also both values were logged, because it is a log log graph).

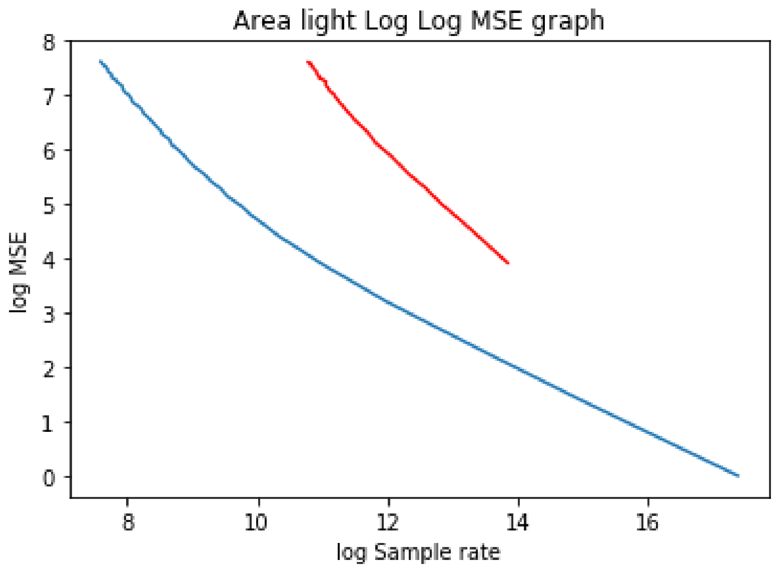

Figure 26. Blue is jittered grid sampling, red is random sampling.

From the graph we can clearly see that jittered sampling converges quicker than random sampling.

To find the MSE results for thin lens was identical method. Except I input the sampling rate for sampleX and sampleY as 1, 2, 3.. etc up to 45. This results in incrementing sampling rates up to 2025. The Sampling rate of the comparison was 71*71 which is a bit over 5000. I didn't have time to complete for jittered sampling rate.

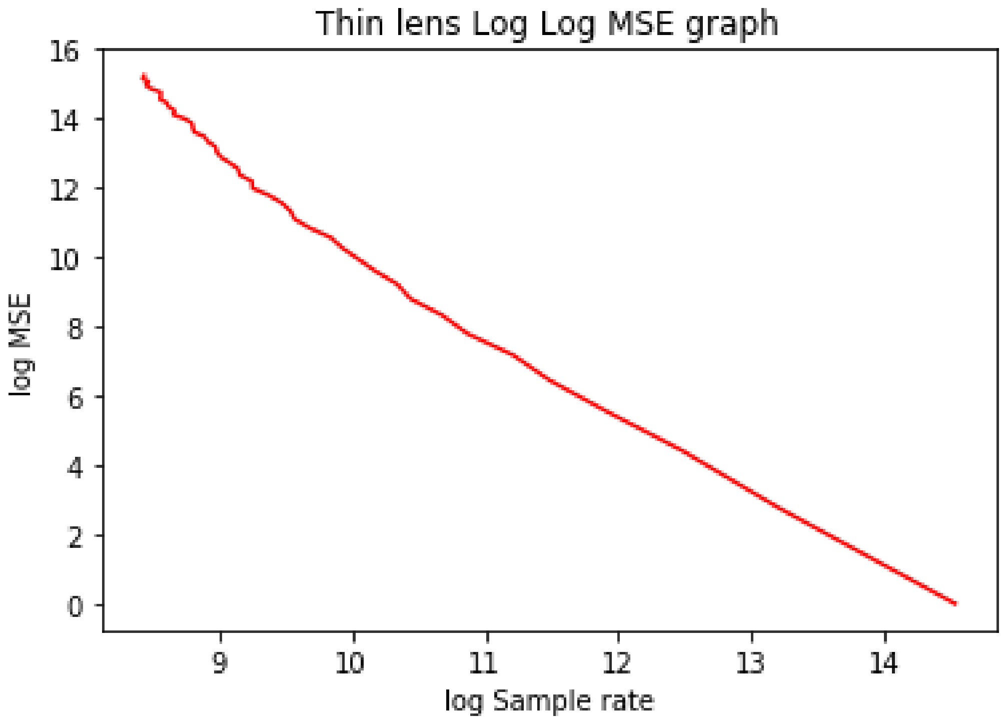

Figure 26. red is random sampling.

Bibliography

[1] Matt Pharr, Wenzel Jakob, and Greg Humphreys (2004-2019). *Transformations*. [online] Physically Based Rendering: From Theory To Implementation. Available at: http://www.pbr-book.org/3ed-2018/Geometry_and_Transformations/Transformations.html [Accessed 4 Nov. 2019].

[2] Whitted, Turner, "An Improved Illumination Model for Shaded Display," Communications of the ACM, Vol. 23, No. 6, June 1980, pp. 343-349.

[3] McKesson, J. (2012). Blinn-Phong Model. [online] Learning Modern 3D Graphics Programming. Available at: https://paroj.github.io/gltut/Illumination/Tut11%20BlinnPhong%20Model.html [Accessed 4 Nov. 2019].

[4] Game Development Stack Exchange. (2016). How to get UV coordinates for sphere. [online] Available at: https://gamedev.stackexchange.com/questions/114412/how-to-get-uv-coordinates-for-sphere-cylindrical-projection [Accessed 4 Nov. 2019].

[5] Scratchapixel (2014). Ray Tracing: Rendering a Triangle (Ray-Triangle Intersection: Geometric Solution). [online] Scratchapixel.com. Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution [Accessed 4 Nov. 2019].

[6] The blog at the bottom of the sea. (2018). Pathtraced Depth of Field & Bokeh. [online] Available at: https://blog.demofox.org/2018/07/04/pathtraced-depth-of-field-bokeh/ [Accessed 4 Nov. 2019].

[7] Scratchapixel.com. (2014). A Minimal Ray-Tracer: Rendering Simple Shapes (Sphere, Cube, Disk, Plane, etc.) (Ray-Sphere Intersection). [online] Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection [Accessed 4 Nov. 2019].

[8] Boulos, S. (n.d.). Notes on efficient ray tracing Bounding volume hierarchies. [online] Available at: https://graphics.stanford.edu/~boulos/papers/efficient_notes.pdf [Accessed 6 Nov. 2019].

[9] Scratchapixel.com. (2014). A Minimal Ray-Tracer: Rendering Simple Shapes (Sphere, Cube, Disk, Plane, etc.) (Ray-Box Intersection). [online] Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection [Accessed 11 Nov. 2019].

[10] Shailendra's Homepage. (2018). Shailendra Paliwal. [online] Available at: http://shailendra.me/tutorial/add-opencv-to-visual-studio-c++-project/ [Accessed 11 Nov. 2019].