BEGG Rain-Alexandra (TDA1.1)

Computer Graphics & Image Processing Lab

Ray Tracer

1.1 Intersection calculation

- Ray-Sphere OK
- Ray-Plane OK
- Ray Tracer test OK

1.2 Coloring and Shading

- Shade function OK

1.3 Shadows

- Shadows OK

2.1 Beckmann distribution and Fresnel term

- RDM_Beckmann and RDM_Fresnel OK

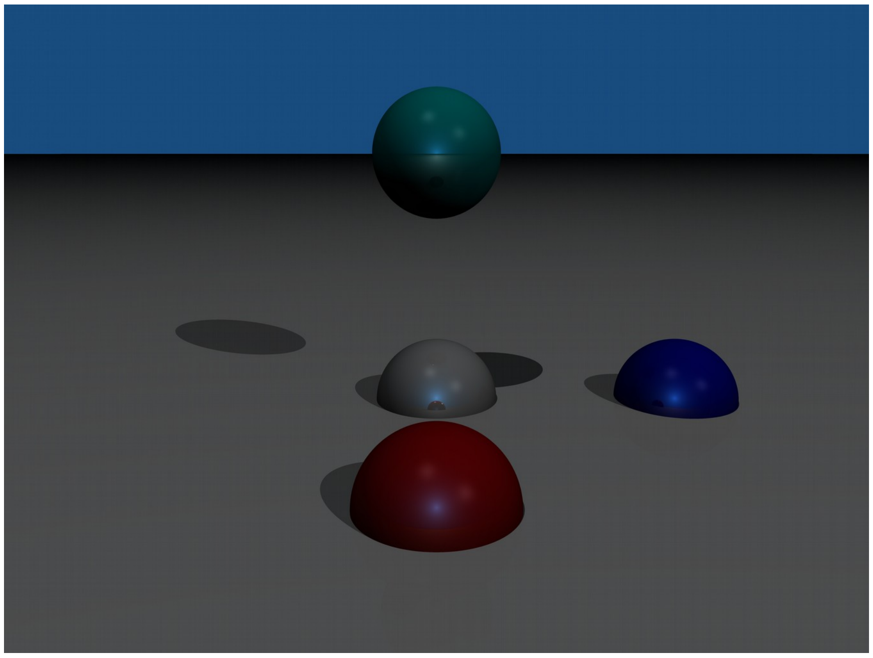
2.2 BSDF implementation

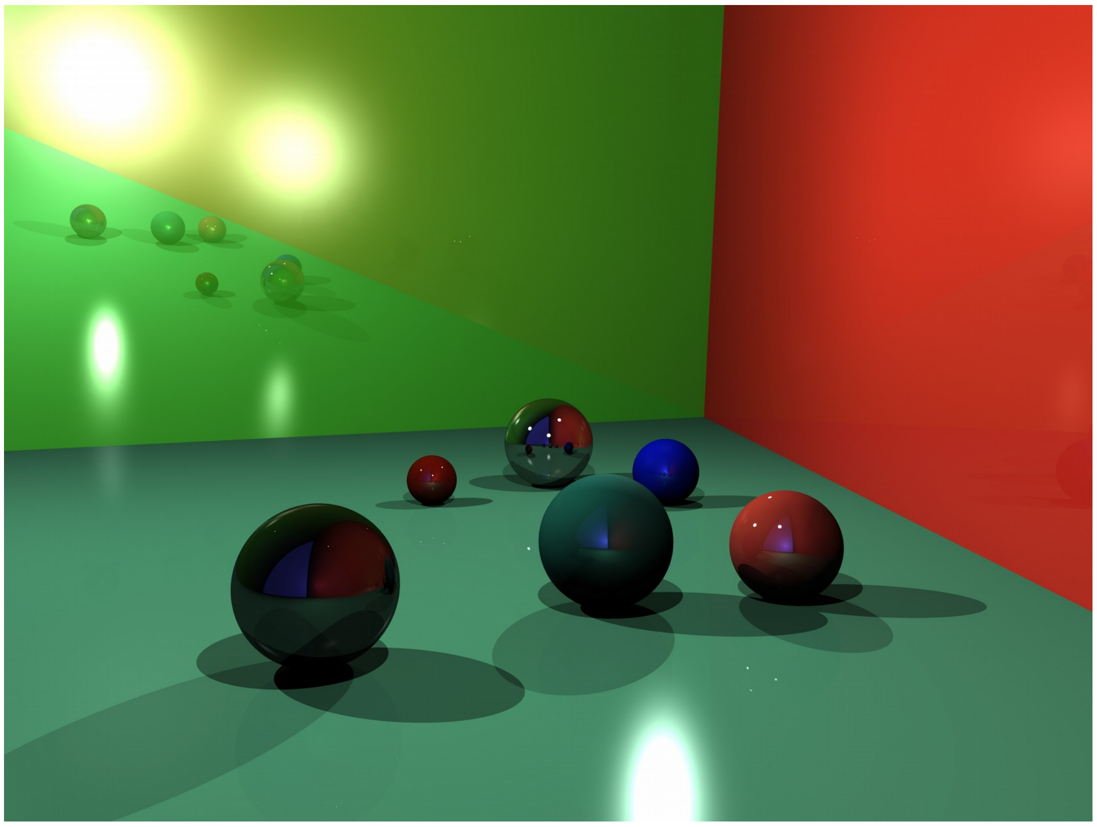
- RDM_G1 OK
- RDM_Smith OK
- RDM_bsdf OK
- shade OK

2.3 Reflections

- trace_ray 2.0 OK

I ran into a problem while generating reflections which led to the following results:





The reflections seemed to be “too strong”. So materials that weren't meant to reflect did so.

To fix this, I modulated the color of a reflection by the specular color of the corresponding light. To do this the final color calculation had to be changed (`F = RDM_Fresnel(LdotH, 1, intersection.mat→IOR` and `cr = trace_ray(scene, &newRay, tree)`):

~~`color += (F * cr);`~~

```
color += (F * cr * intersection.mat→specularColor);
```

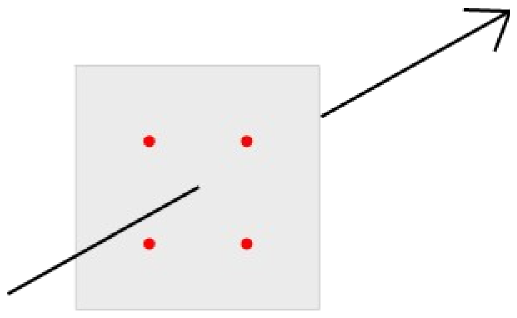
3.1. Anti aliasing

- Anti aliasing OK.

In the `renderImage` function, each pixel at the coordinates (i, j) in an image has a ray passed through it. Each pixel has a dimension of 1, so instead of passing one ray through each pixel, five were passed through by adding or subtracting 0.25 from the (i, j) coordinates when initializing a ray. For example

```
vec3 ray_dir3
= scene->cam.center + ray_delta_x + ray_delta_y + float(i - 0.25)*dx +
float(j + 0.25)*dy;
```

Here is a diagram illustrating the idea described above. The black line is the original ray which each pixel is intersected by, and the red points are the intersection points of the four extra rays:



The final color is the average of the colors at each of the five intersection points.

3.2 Kd-tree

I was not able to complete the Kd-tree by the deadline but an implementation of the tree root's bbox was attempted.

A minimum point is initialized at very large coordinates and a maximum point is initialized at very small coordinates. Each object of the scene is evaluated. If the object is a sphere (in my implementation only spheres are evaluated), its minimum coordinates are calculated by taking the coordinates of the sphere's center and then subtracting the length of its radius from these center coordinates (the same is done for its maximum coordinates but the radius' length is added instead). For each object, the bbox's min and max are updated.

3.3 Other objects

Triangle OK

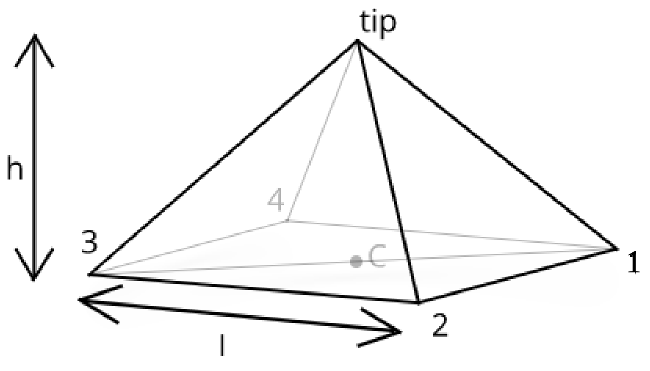
Triangle data type added to the *Geometry* data type. A triangle has three points: a, b, and c as its parameters.

Implemented by using the Möller-Trumbore algorithm. I had problems calculating the intersection normal, which made a triangle black even if it was colored, blocked by nothing, and facing light. I had to test if the triangle's normal had the same direction as the ray (if `dot(ray→dir , triNorm) ≥ 0`). In this case, the intersection's normal will be opposed to that of the triangle.

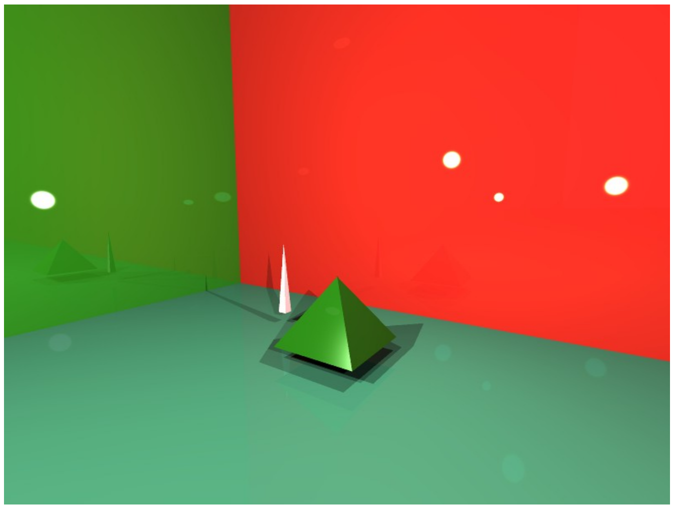
Pyramid OK (Scene 6)

Pyramid data type added to the *Geometry* data type. I chose to have a height *h*, a length *l* (the length of its square base), a center *C* (this is where the pyramid will be placed in the scene), and an array of 6 triangles, 2 for the base of the pyramid and 4 for the others (see diagram)

To initialize a pyramid the `initTriangle` function is called for the six triangles that form it. The three points passed as arguments in this function are calculated based on the height, length, and center of the pyramid. For example, one of the triangles of the base will have points 1, 2, and 3 as parameters and one of the triangles that are not at the base may have points *tip,* 1, and 4 as parameters (see diagram)



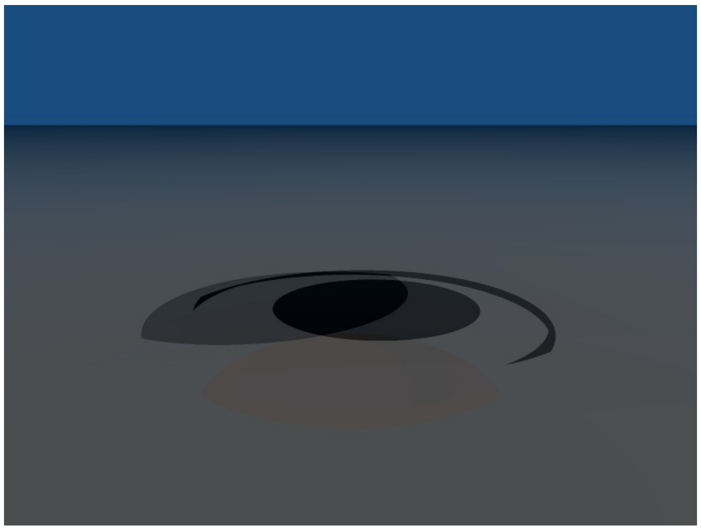
Problem: with this specification, the triangle cannot be oriented on an axis. So its base is always orthogonal to the x and z axises:



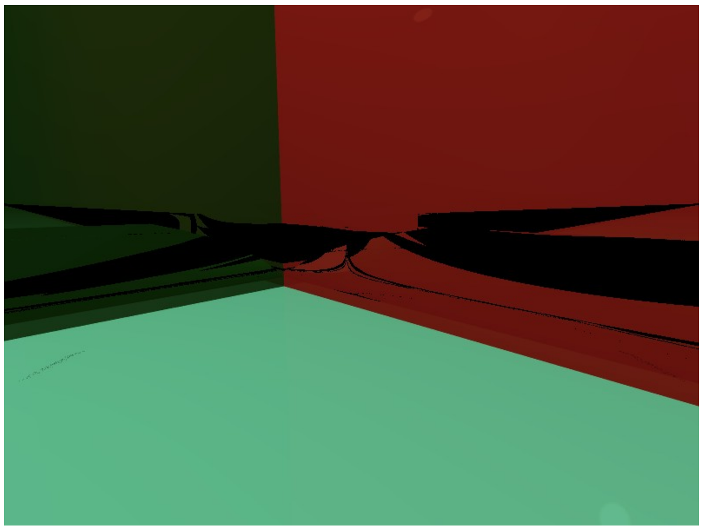
Cylinder and Cone Attempt

There are many possible representations for the cylinder and cone. I chose to have a center, a height, and a radius

Fail → There were definitely other implementation errors but I am sure that the intersection normals were one of the causes of these sorts of results:



CYLINDER



CONE

Mesh OK (Scene 7)

Mesh data type added to *Geometry* data type. The parameters are a number of vertices, a number of triangles, an array of vertices, and an array of triangles.

From a `.obj` file, the vertices are put into the array (at the 'v' lines, sometimes they must be scaled down depending on the mesh object) as well as the faces (at the 'f' lines, triangles are initialized and put in the array of triangles of the *mesh* data type). So in `intersectMesh`, `intersectTriangle` is called for each triangle of the object.

Only low poly meshes were used for speed purposes, but image generation is quite slow (for example, the deer mesh has 1508 triangles). It takes between 30 to 50 minutes for an image to be generated with anti aliasing and about 15 minutes without. The kd-tree would have been very useful at this point.