CSC 305 – Assignment 1 – Ray Tracing – May 22/ 2017

Jonathan Healy – V00845624

For this assignment I designed a Ray Tracer with the help of two books, “Ray Tracing: The Next Week” and “Ray Tracing in One Weekend”, both written by Peter Shirley. This program uses C++ and implements object oriented programming to design a simple ray tracer with some nice effects. For the most part I relied on code as presented in this book although things often didn't seem to go together exactly as they should and took much longer than I initially expected. Understanding how everything worked was challenging but rewarding.

In object oriented programming, programs are designed by making them out of objects that interact with each other. This program is contained of many different classes and objects that fit together to implement simple ray tracing. Vec3.h lays the groundwork to be able to implement the mathematics needed for vector algebra. Texture.h for example defines different classes of textures that are used for objects like the class, constant_texture which is used to create diffuse objects. Sphere.h specifies how rays which are called by hitable.h interact with a sphere object. The most common data structure in this program is the hitable list which contains an array of objects specifying how many specific objects are in a scene.

The most significant algorithm in this program is bvh.h which is the bounding volume hierarchy used to improve efficiency and help the code to run faster. The idea is to make the ray object intersection which it the most time consuming part of a ray tracer, a logarithmic search function. It is a tree structure on a set of geometric objects that are wrapped in bounding volumes that form the leaf nodes of a tree. Children volumes do not have be examined during collision testing if their parent volumes are not intersected.

A ray tracer implements the transport of light to generate images by tracing that path of light through pixels in an image plane and simulating the effects as it encounters various objects. The light rays of a ray tracer originate at the cameras position, travel through each pixel and calculate various information such as colour and other material properties of an object. Rays bounce off objects by calculating that objects normal in space.

When a ray hits a surface it can calculate any one of three new types of rays: reflection, refraction and shadow. A reflection ray is traced in the mirror reflection direction. The closest object it interacts with is what will be seen it its reflection. Refraction rays are similar but may enter or exit and object. Shadow rays are traced towards each light. If an opaque object is found between the surface and the light, the surface is in shadow and the light does not illuminate it. This is a recursive ray tracing method and adds realism. The number of reflections a ray can take must be limited.

A Lambertian surface shows the same amount of brightness to a viewer regardless of the viewing angle. This defines an ideally diffuse surface. Light that reflects off a diffuse surface has its direction randomized. Light may also be absorbed and the more dark an object is the more light it has absorbed. The algorithm used in this program randomizes the direction of light rays that hit a diffuse or Lambertian object.

```
class lambertian : public material {
public:
```

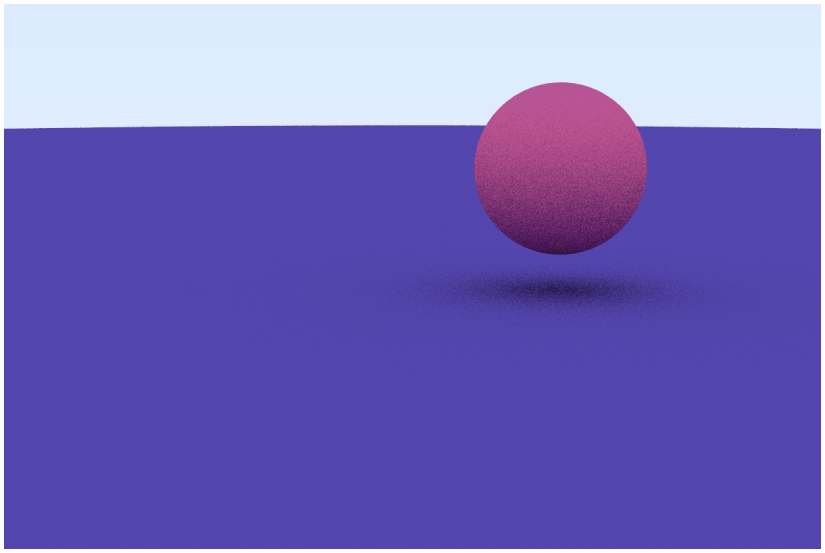
```
    lambertian(texture *a) : albedo(a) {}
    virtual bool scatter(const ray& r_in, const hit_record& rec, vec3& attenuation, ray&
scattered) const  {
        vec3 target = rec.p + rec.normal + random_in_unit_sphere();
        scattered = ray(rec.p, target-rec.p, r_in.time());
        attenuation = albedo->value(rec.u, rec.v, rec.p);
        return true;
    }

    texture *albedo;
};
```

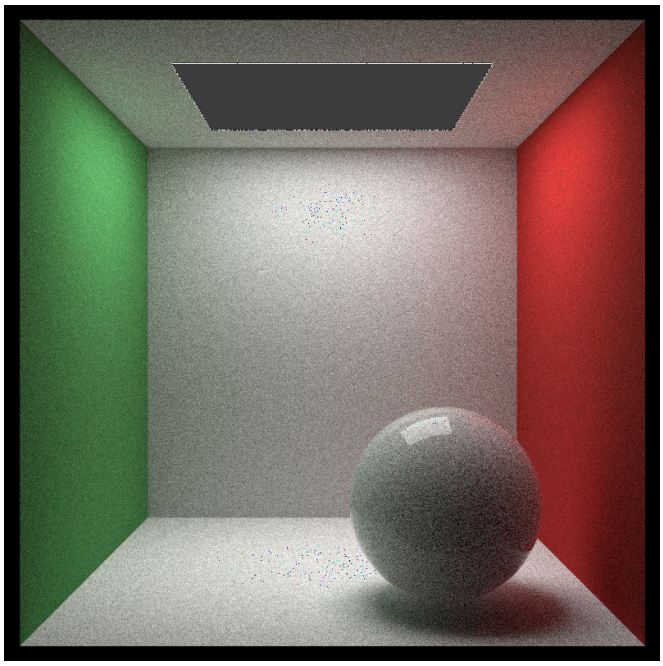
For a reflective or metal like surface rays are not randomly scattered and by calculating v, the ray, and n the surface normal we can use the reflect method to return the direction of the reflected ray to model a metal surface.

```
vec3 reflect(const vec3& v, const vec3& n) {
    return v - 2*dot(v,n)*n;
}
```

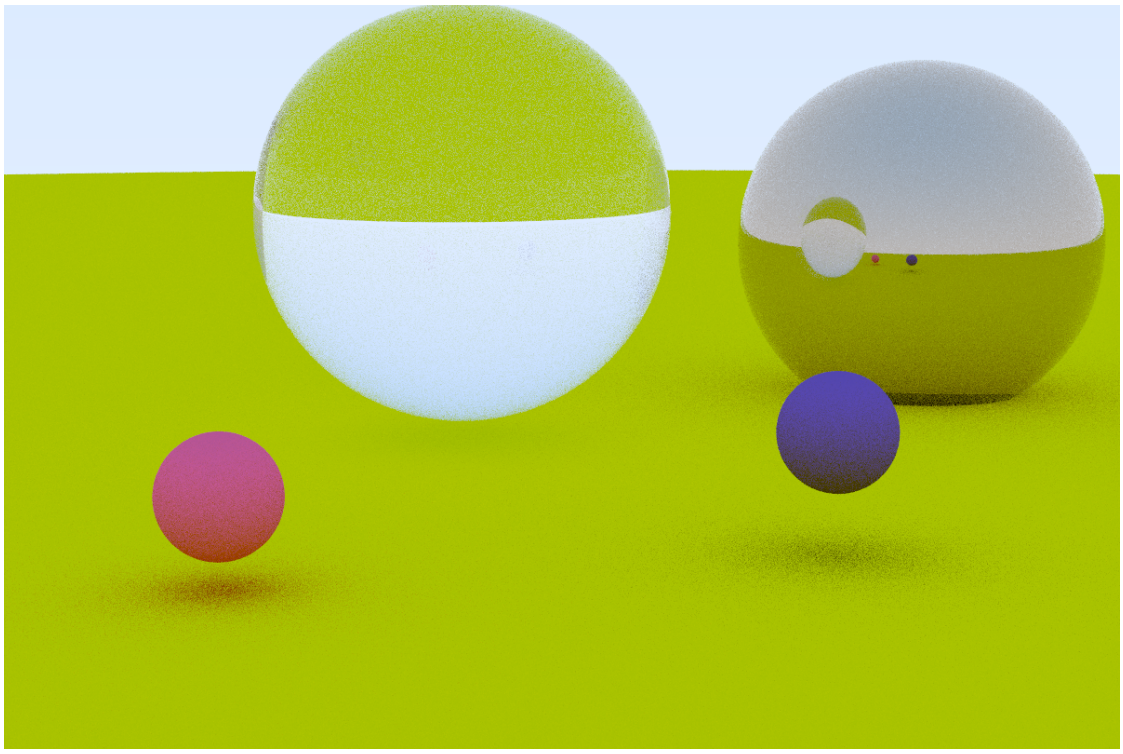
Clear materials like water or glass is called a dielectric surface. For refraction or a dielectric surface, when a light ray hits it is split into a reflected and a refracted ray. This program handles that by randomly choosing between reflection or refraction and only generating one scattered ray per interaction. Refraction is described by Snell's law: n1sintheta1 = n2sintheta2. N is a refractive index and air is usually 1, glass anywhere from 1.3 to 1.7 and diamond 2.4.



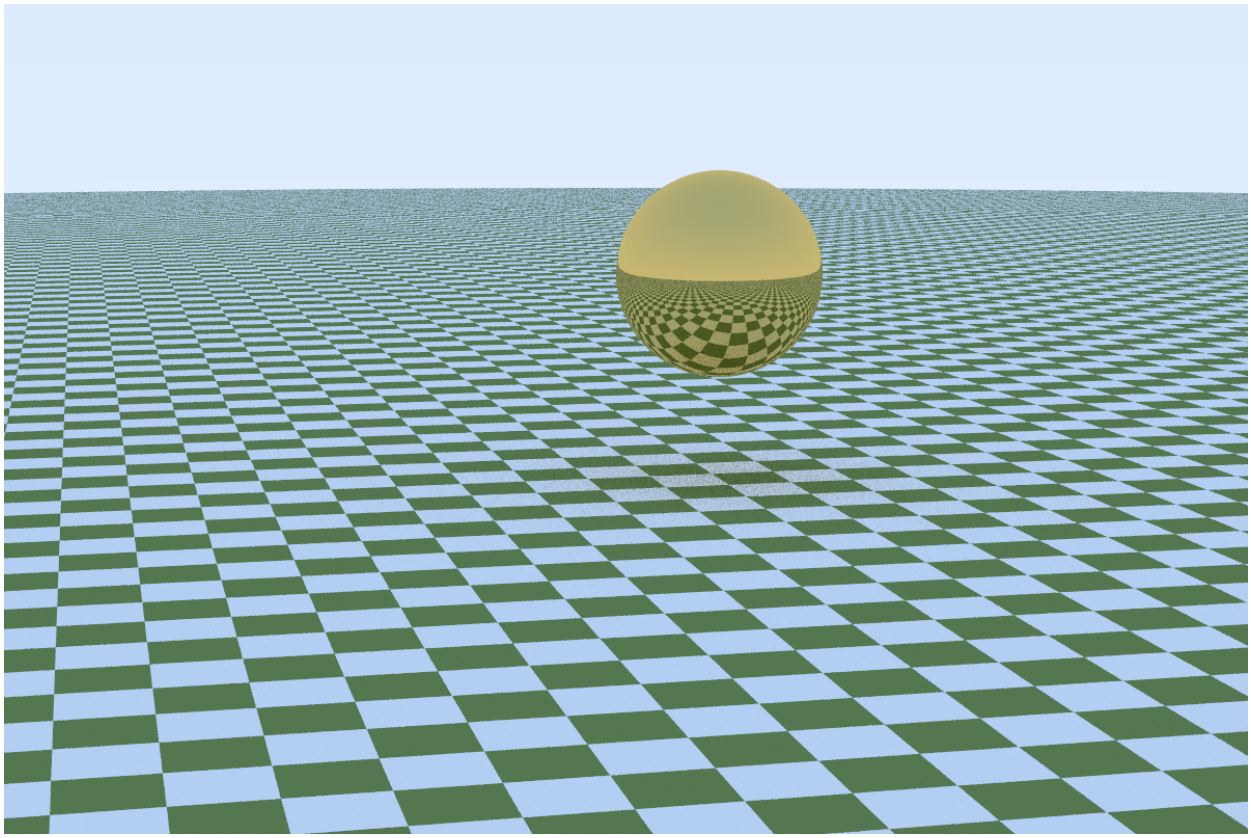
This is a sphere with a Lambertian surface. Notice the shadow.



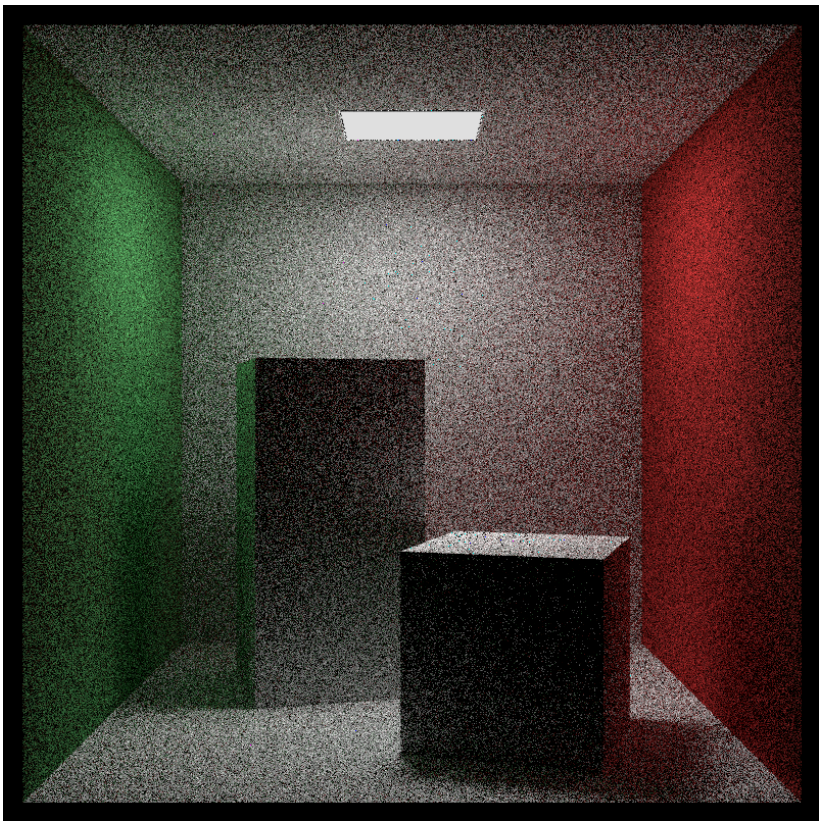
This is the Cornell box.



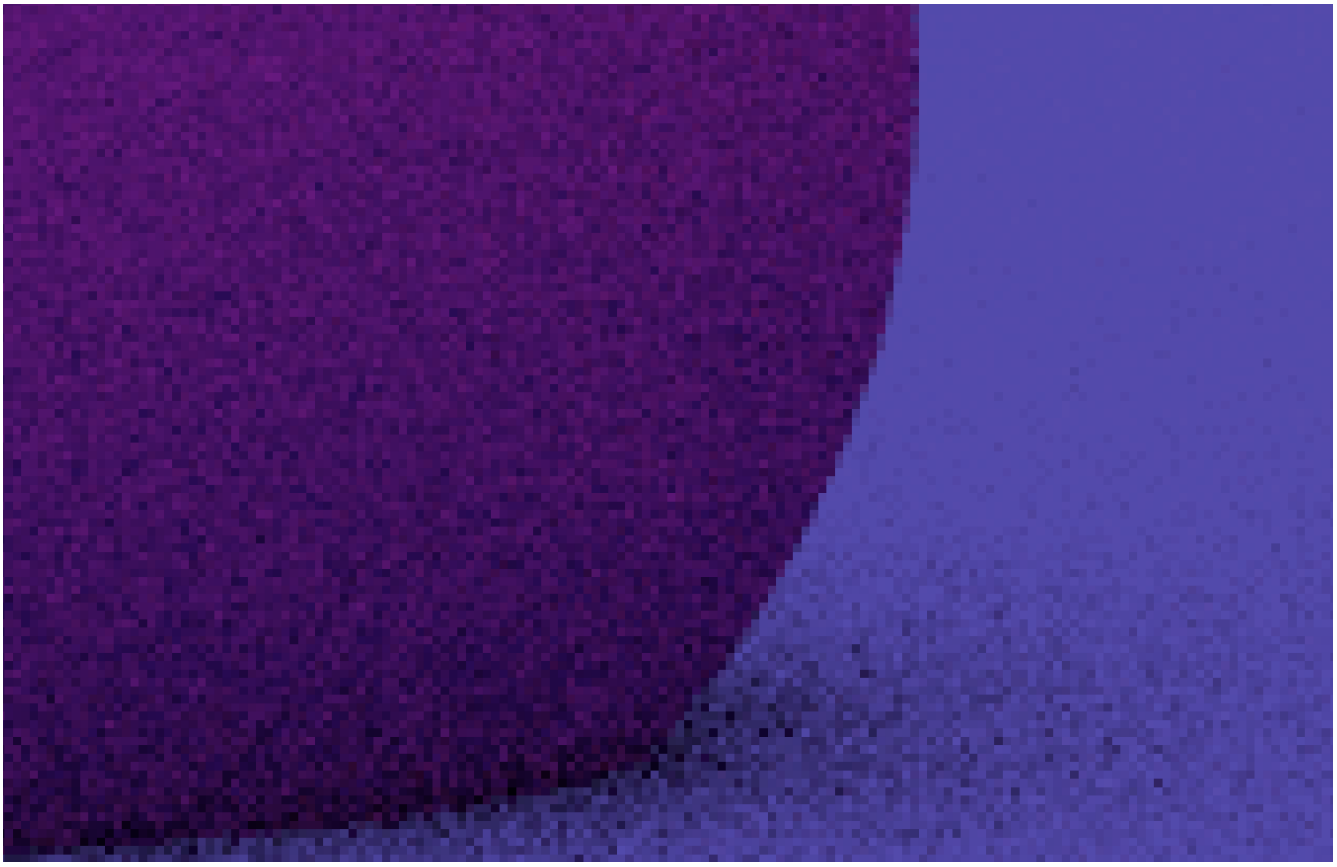
This image demonstrates a reflective and a refractive sphere. The two smaller spheres are both Lambertian added for effect.



A checkerboard texture was added to this plane.



Area light added to a Cornell box with boxes in the scene for effect.



Anti aliasing. This is accomplished using a random number generator. For each pixel numerous samples are taken. Rays are sent through each sample and the colours are averaged to produce anti aliasing which blurs the jagged pixelated look there would be without it.

```
for (int s=0; s < ns; s++) {
            float u = float(i + drand48()) / float(nx);
            float v = float(j + drand48()) / float(ny);
            ray r = cam.get_ray(u, v);
            vec3 p = r.point_at_parameter(2.0);
            col += color(r, world,0);
        }
```

This loop repeats ns number of times where ns represents the number of samples that are being taken per lixel.

Compiling:

The file rayTracingX handles the Cornell box while rayTracingW handles all the other cases.

G++ main.cc to compile
./a.out > ball.ppm to execute and save image to file