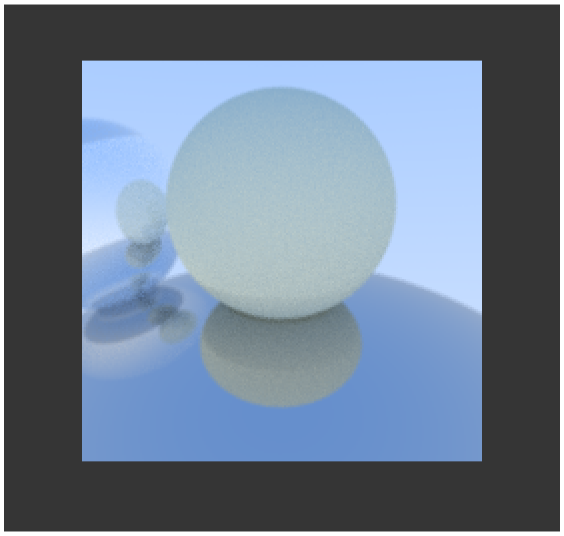
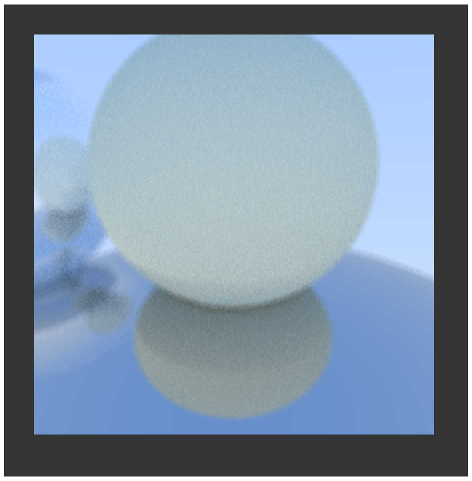
Lab 6 Report
Fan Wang
CSE 3541

Topic:
Implementing Ray Tracing Algorithm To Generate Realistic Image.

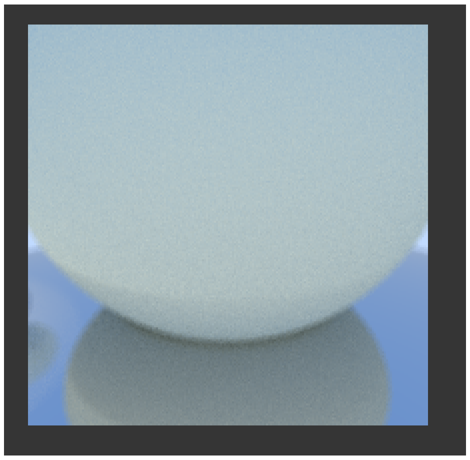
Basic Idea:
Ray tracing algorithm casts rays originating from the camera position to its view plane and traces the color of each pixel on that view plane. In order to create realistic images, the algorithm requires implementation of different textures and material of the object. By doing so, one can blend the color of each point hitted by a ray to create the final image. My old topic is about using openGL to implement ray tracing. However, it is really expensive especially considering the basic idea is to generate an ray tracing image first then use openGL to show that image. Besides, ray tracing is less feasible when it comes to real time rendering. Thus, I decided to change my topic to only generate ray tracing images.

Functionality:

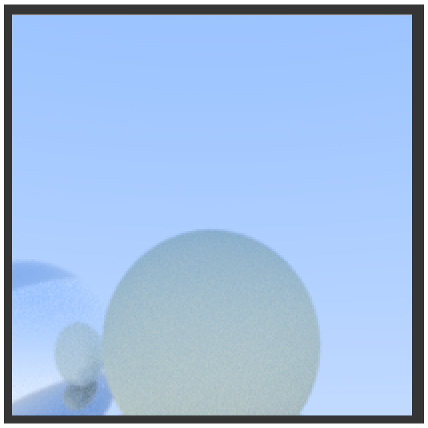
1. **Generate image of same object with different camera:**
   original:
   
   Camera move toward the object:

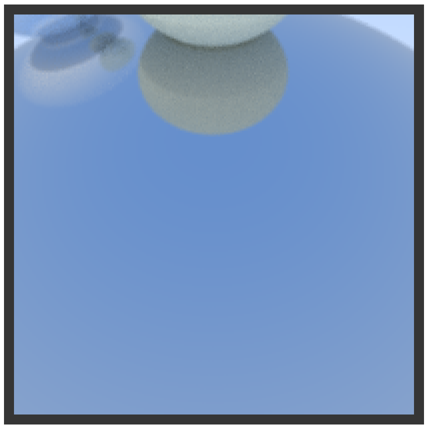


Change field of view(decrease):

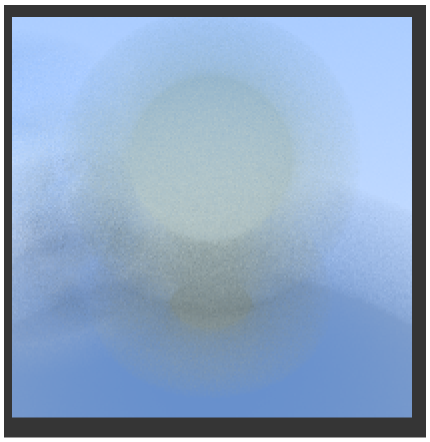


Rotate camera:

(Up)

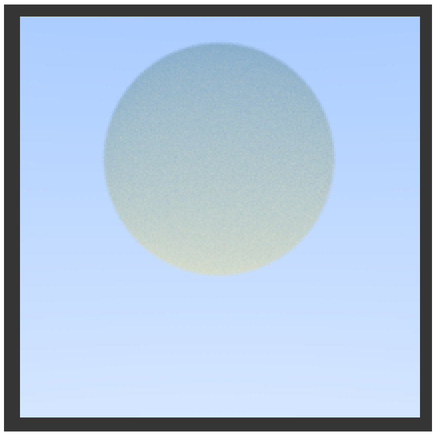

(Down)

Change aperture:

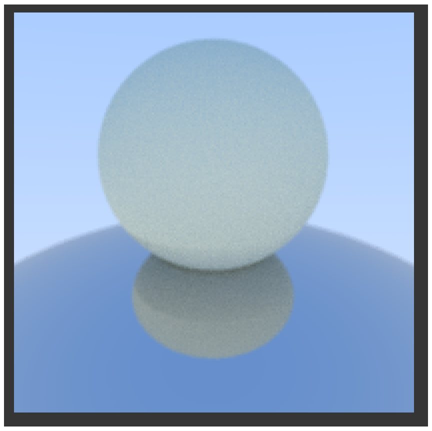


2. **Modify world space, show spheres with material.**

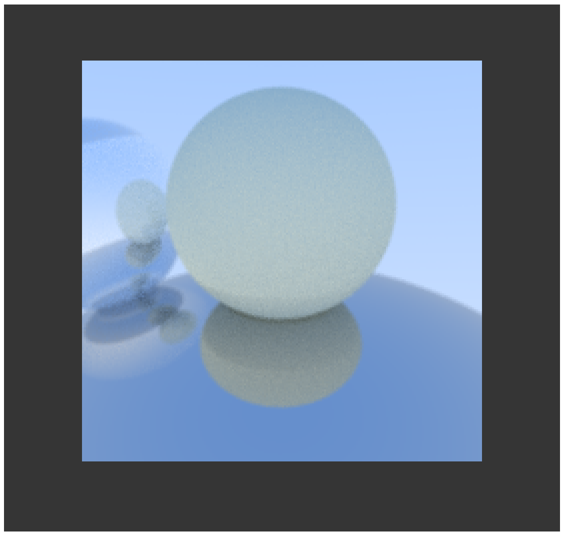
Lambertian



Metal(bottom)



Dielectric


(left)

Implementation:

Image.hpp (Followed tutorial to have understanding of using material and color blending. Left are implemented by me.)

Image.hpp implements class Image. This class is equivalent to a ray tracer. It includes settings for the world space(add sphere to the world), configuration of the image(width, height, resolution). It is also responsible for detecting rays hitting the sphere and generating image color in .ppm format.

```
#ifndef IMAGE_HPP
#define IMAGE_HPP
#define SAMPLE_TIME 100

#include "Camera.hpp"
#include "Sphere.hpp"
#include <iostream>
#include <fstream>
#include <vector>
```

```
class Image{
   private:
       int imageWidth;
       int imageHeight;
       Camera camera;

   public:
       std::vector<Sphere *> sphereList;

   public:
       Image(int w, int h, Camera c){
           imageWidth = w;
           imageHeight = h;
           camera = c;
       }

       bool hitAny(std::vector<Sphere*> hArray, int hSize, Ray & r, float
tMin, float tMax, HitRecord & rec)
       {
           bool isHitAny = false;
           HitRecord tempRecord;
           for(int i = 0; i < hSize; i++)
           {
               if(hArray[i]->hit(r, tMin, tMax, tempRecord))
               {
                   isHitAny = true;
                   tMax = tempRecord.t;
                   rec = tempRecord;
               }
           }
           return isHitAny;
       }

       glm::vec3 printSky(Ray & r)
       {
           glm::vec3 normalDir = glm::normalize(r.getDirection());
           float t = 0.5f * (normalDir.y + 1.0f);
           return (1.0f-t) * glm::vec3(1.0f, 1.0f, 1.0f) + t *
glm::vec3(0.5f, 0.7f, 1.0f);
```

```cpp
        }

        glm::vec3 rayColor(Ray & r, std::vector<Sphere*> hArray, int
hArraySize, HitRecord & record, int depth)
        {
            glm::vec3 color;

            float tMIN = 0.001f;
            float tMAX = 100.0f;
            if(depth <= 0)
            {
                return glm::vec3(0, 0, 0);
            }

            if(hitAny(hArray, hArraySize, r, tMIN, tMAX, record))
            {
                Ray childRay;
                glm::vec3 changed;
                if(record.matP -> sendNewRay(r, record, changed, childRay))
                {
                    color = changed * rayColor(childRay, hArray,
hArraySize, record, depth - 1);
                }else
                {
                    color = glm::vec3(0.0f, 0.0f, 0.0f);
                }
            }else{
                color = printSky(r);
            }

            return color;
        }

        void castColor(glm::vec3 const color)
        {
            int x = color.x * 255.99;
            int y = color.y * 255.99;
            int z = color.z * 255.99;
            std::cout << x << " " << y << " " << z << "\n";
        }
        void addLambertianObjectToScene(glm::vec3 color, glm::vec3 center,
float radius)
        {
            Material *m = new Lambertian(color);
            Sphere *s = new Sphere(center, radius, m);
            sphereList.push_back(s);
        }

        void addMetalObjectToScene(glm::vec3 color, glm::vec3 center, float
radius)
        {
            Material *m = new Metal(color);
            Sphere *s = new Sphere(center, radius, m);
            sphereList.push_back(s);
        }

        void addDielectricObjectToScene(float ir, glm::vec3 center, float
radius)
        {
            Material *m = new Dielectrics(ir);
            Sphere *s = new Sphere(center, radius, m);
            sphereList.push_back(s);
        }

        void printImage()
        {
            std::cout << "P3\n" << imageWidth << " " << imageHeight <<
"\n255\n";
            int depth = 10;
            for(int j = imageHeight - 1; j >= 0; j--)
            {
                for(int i = 0; i < imageWidth; i++)
                {
                    HitRecord record;
                    glm::vec3 color(0.0f, 0.0f, 0.0f);
                    for(int k = 0; k < SAMPLE_TIME; k++)
                    {
                        float u = (float(i) + drand48())/ (imageWidth - 1);
```

```
                       float v = (float(j) + drand48())/ (imageHeight -
1);
                       Ray ray = camera.getRay(u, v);
                       color += rayColor(ray, sphereList,
sphereList.size(), record, depth);
                   }
                   color.x /= SAMPLE_TIME;
                   color.y /= SAMPLE_TIME;
                   color.z /= SAMPLE_TIME;
                   castColor(color);
               }
           }
       }

};

#endif
```

Camera.hpp (Followed tutorial to add effects like aperture and field of view. Left are implemented by me)
Camera.hpp is mainly responsible for the behavior of the camera. It controls things like image position, uvw coordinate of the eye space and space that can be seen by camera. It also casts rays from the camera to each pixel. Besides, camera.hpp also implements the effect named defocus blur.

```
#ifndef CAMERA_HPP
#define CAMERA_HPP

#include "Ray.hpp"
#include <math.h>

glm::vec3 randomUnitDisk()
{
   return glm::normalize(glm::vec3((drand48() * 2.0f - 1.0f), (drand48() *
2.0f - 1.0f), 0));
}
class Camera
{
   private:
       glm::vec3 position;
       glm::vec3 horizontal;
       glm::vec3 vertical;
```

```
        glm::vec3 lowerLeftCornor;
        glm::vec3 u, v, w;
        float vfovd;
        float lensRadius;


    public:
        Camera()
        : position{0.0f, 0.0f, 0.0f},
          horizontal{0.0f, 0.0f, 0.0f},
          vertical{0.0f, 0.0f, 0.0f},
          lowerLeftCornor{0.0f, 0.0f, 0.0f}
        {}

        Camera(glm::vec3 p, glm::vec3 h, glm::vec3 ve, float vd, glm::vec3
lookAt, glm::vec3 vUp, float aspectRatio, float aperture, float focusDist)
        :position(p),
        horizontal(h),
        vertical(ve),
        vfovd(vd)
        {
            // degree to radian
            float vfovr = ( vd * M_PI ) / 180.0f;
            float ratio = tan(vfovr / 2);
            horizontal = ratio * horizontal;
            vertical = ratio * vertical;

            // rotate camera;
            w = glm::normalize(position - lookAt);
            u = glm::normalize(glm::cross(vUp, w));
            v = glm::normalize(glm::cross(w, u));
            horizontal = u * horizontal * focusDist;
            vertical = v * vertical * focusDist;

            lowerLeftCornor = position - 0.5f * horizontal - 0.5f *
vertical - focusDist * w;
            lensRadius = aperture / 2.0f;
        }

        Ray getRay(float s, float t)
        {
            glm::vec3 rd = lensRadius * randomUnitDisk();
            glm::vec3 offset = u * rd.x + v * rd.y;

            return Ray(position + offset, lowerLeftCornor + s * horizontal
+ t * vertical - position - offset);
        }

};
```

Ray.hpp (Watch tutorial to get basic idea then Implemented by me)
Ray.hpp implements a single ray. Ray can be used to requery any point on that ray given t.

```
#ifndef RAY_HPP
#define RAY_HPP

#include <glm/glm.hpp>

class Ray
{
   private:
       glm::vec3 origin;
       glm::vec3 direction;
   public:
       Ray()
       : origin{0.0f, 0.0f, 0.0f},
         direction{0.0f, 0.0f, 0.0f}
       {}

       Ray(glm::vec3 o, glm::vec3 d)
       : origin(o),
         direction(d)
       {}

       glm::vec3 getOrigin()
       {
           return origin;
       }

       glm::vec3 getDirection()
```

```
        {
            return direction;
        }


        glm::vec3 at(float tP)
        {
            return origin + tP * direction;
        }



};

#endif
```

HitStruct.hpp(Followed tutorial)
HitStruct.hpp includes one struct that stores all information required or can be derived from calculating whether a ray hit some object.

```
#ifndef HITSTRUCT_HPP
#define HITSTRUCT_HPP

#include "Ray.hpp"

class Material;

struct HitRecord
{
   public:
        float t;
        glm::vec3 hitPoint;
        glm::vec3 normal;
        Material * matP;
};


#endif
```

Sphere.hpp(Watch tutorial to get basic idea then Implemented by me)
Sphere.hpp includes information about each single sphere inside the world. The information it has includes the center and the radius. It has one method to detect whether a ray has hit this specific object.

```
#ifndef SPHERE_HPP
#define SPHERE_HPP

#include <math.h>
#include "Ray.hpp"
#include "Material.hpp"
#include <iostream>
class Sphere
{
   glm::vec3 center;
   float radius;
   Material * matP;

   public:
       Sphere(){}
       Sphere(glm::vec3 c, float r, Material * m) : center(c), radius(r),
matP(m) {}
       bool hit(Ray & r, float tMin, float tMax, HitRecord & rec)
       {
           glm::vec3 oc = r.getOrigin() - center;
           float a = glm::dot(r.getDirection(), r.getDirection());
           float b = 2 * glm::dot(r.getDirection(), oc);
           float c = glm::dot(oc, oc) - radius * radius;
           float d = b * b - 4 * a * c;
           if(d < 0.0f)
           {
               return false;
           }
           else{
               float t = (-b - sqrt(d)) / (2 * a);
               if(t < tMin || t > tMax)
               {
                   return false;
               } else {
               rec.t = t;
               rec.hitPoint = r.at(t);
               rec.normal = glm::normalize(rec.hitPoint - center);
               rec.matP = matP;
               return true;
           }
   }


}
};

```

Material.hpp(Followed tutorial)
Material is a virtual base class. It includes information of how each child ray should go after its parent hit the material. In this file, I have implemented 3 materials, they are Lambertian, Metal and Dielectric.

```
#ifndef MATERIAL_HPP
#define MATERIAL_HPP

#include "HitStruct.hpp"
#include <math.h>
glm::vec3 reflect(glm::vec3 & v, glm::vec3 & normal)
{
   return v - 2 * glm::dot(v, normal) * normal;
}

glm::vec3 refract(glm::vec3 & in, glm::vec3 & normal, float etaRatio)
{
   glm::vec3 outV = etaRatio * (in + normal * (glm::dot(-1.0f * in,
normal) / (glm::length(in) * glm::length(normal))));
   glm::vec3 outH = -1 * sqrt(1 - (glm::length(outV) * glm::length(outV)))
* normal;
   return outV + outH;
}

float schlickReflectance(float cos, float rid)
{
   float r0 = (1-rid) / (1+rid);
   r0 = r0 * r0;
   return r0 + (1-r0) * pow((1-cos), 5);
}

class Material
{
   public:
```

```
   virtual bool sendNewRay(Ray & r, HitRecord & hr, glm::vec3 & changed,
Ray & newRay) const = 0;
};

class Lambertian : public Material
{
   private:
   glm::vec3 baseColor;


   public:
   Lambertian(glm::vec3 color) : baseColor(color) {}
   virtual bool sendNewRay(Ray & r, HitRecord & hr, glm::vec3 & changed,
Ray & newRay) const
   {
       glm::vec3 tempNewRay = hr.normal +
glm::normalize(glm::vec3((drand48() * 2.0f - 1.0f), (drand48() * 2.0f -
1.0f), (drand48() * 2.0f - 1.0f)));

       // avoid the new ray point into the object
       float threshold = 1e-7;

       if((tempNewRay.x < threshold) && (tempNewRay.y < threshold) &&
(tempNewRay.z < threshold))
       {
           tempNewRay = hr.normal;
       }

       newRay = Ray(hr.hitPoint, tempNewRay);

       changed = baseColor;

       return true;
   }

};

class Metal : public Material
{
   private:
```

```
   glm::vec3 baseColor;
   float fuzz = 0.0f;

   public:
   Metal(glm::vec3 color) : baseColor(color) {}

   virtual bool sendNewRay(Ray & r, HitRecord & hr, glm::vec3 & changed,
Ray & newRay) const
   {
       glm::vec3 reflectedDirection =
reflect(glm::normalize(r.getDirection()), hr.normal) + fuzz *
glm::normalize(glm::vec3((drand48() * 2.0f - 1.0f), (drand48() * 2.0f -
1.0f), (drand48() * 2.0f - 1.0f)));

       newRay = Ray(hr.hitPoint, reflectedDirection);

       changed = baseColor;

       return (glm::dot(newRay.getDirection(), hr.normal) > 0);
   }

};

class Dielectrics: public Material
{
   private:
   float ir;

   public:
   Dielectrics(double indexOfRefraction) : ir(indexOfRefraction) {}

   virtual bool sendNewRay(Ray & r, HitRecord & hr, glm::vec3 & changed,
Ray & newRay) const
   {
       changed = glm::vec3(1.0f, 1.0f, 1.0f);
       float refractRatio;
       if(glm::dot(r.getDirection(), hr.normal) < 0)
       {
           refractRatio = (1.0f / ir);
       }else
```

```
        {
            refractRatio = ir;
        }

        glm::vec3 rayDirectNormalized = glm::normalize(r.getDirection());


        float cos = dot(-rayDirectNormalized, glm::normalize(hr.normal)) >
1.0f ? 1.0f :  dot(-rayDirectNormalized, glm::normalize(hr.normal));
        float sin = sqrt(1 - cos * cos);

        bool noRefract = (refractRatio * sin) > 1.0f;
        glm::vec3 dir;

        if(noRefract || schlickReflectance(cos, refractRatio) > drand48())
        {
            dir = reflect(rayDirectNormalized, hr.normal);
        }
        else
        {
            dir = refract(rayDirectNormalized, hr.normal, refractRatio);
        }

        newRay = Ray(hr.hitPoint, dir);
        return true;
    }

};

#endif
```

Video
https://youtu.be/qGk1eeaVbaw

Reference

https://www.reddit.com/r/GraphicsProgramming/comments/pudchw/how_do_we_write_a_ray_tracer_in_opengl/

https://github.com/RayTracing/raytracing.github.io

https://raytracing.github.io/books/RayTracingInOneWeekend.html