CSE 167 (Fall 2022) Raytracer Final Project Guide

In this final project topic, you will write a ray tracer. Raytracers can produce some of the most impressive renderings with high quality shadows and reflections. This guide hints how you can modify the HW3 codebase into a ray tracer.

1 Expectation

The basic expectation is to have a raytracer that features Page-12–15 of the slides https://cseweb.ucsd.edu/~alchern/teaching/cse167_fa22/7-1RayTracing.pdf

Namely, the ray tracer can first of all reproduce the same result as your HW3 (basic diffuse + Blinn-Phong specular shading + ambient). And then, by adding a few modification for visibility of light to obtain shadows. Finally, replace the specular part by a recursive mirror reflection with a fixed recursion depth.

The bonus credit will be given if you also replace ambient and diffuse by global illumination technique (page 56 onward).

2 Turning HW3 framework to a ray tracing framework

First of all, you need to be familiar with the details presented in the slides. The following is just a suggestion how you can take steps to build the framework.

2.1 Create the “Image” class

Create a class “image” that contains the arrays of pixels where we store the colors. A possible set of class members for image would be

• int width;
• int height;
• std::vector<glm::vec3> pixels // RGB colors;

We can have a simple constructor that initializes the width and height, and a member function

• void Initialize();

that initializes the size of the array pixels to width times height (which would be the total number of pixels).

It would be convenient if we can draw the content of our image to the screen. To do so, we follow page 52 of the slides. Prepare a few more class members (can be set as private members)

• unsigned int fbo; // framebuffer object
• unsigned int tbo; // texture buffer object

In the Image::init() function, add

```
void Image::init(){
    glGenFramebuffers(1,&fbo);
    glGenTextures(1,&tbo);
}
```

Create another member function for the Image class called Image::draw() where we pour the data of the pixels into the texture buffer, and we attach the texture to the framebuffer.

```
void Image::draw(){
    glBindTexture(GL_TEXTURE_2D, tbo);
    glTexImage2D(GL_TEXTURE_2D,0,GL_RGB,width,height,
                 0,GL_RGB,GL_FLOAT, &pixel[0][0]);
    glBindFramebuffer(GL_READ_FRAMEBUFFER, fbo);
    glFramebufferTexture2D(GL_READ_FRAMEBUFFER,GL_COLOR_ATTACHMENT0,
                           GL_TEXTURE_2D, tbo, 0); // attach texture and the read frame
```
glBindFramebuffer(GL_WRITE_FRAMEBUFFER, 0); // if not already so
glBlitFramebuffer(0,0,width,height,0,0,width,height,GL_COLOR_BUFFER_BIT,
GL_NEAREST); // copy framebuffer from read to write
}

Now, you can test if the code work. Go to main.cpp. Create a global variable of type Image similar to how the scene is created.

static Image image(width,height);

In void initialize(void) function of main.cpp, call the initialization function for the image as well

image.init();

Now, you can add another keyboard trigger in the keyboard callback function where you can activate showing the image. You can set some dummy helper function that put some colors in image.pixels. After your image has pixel colors, the code that should show the result on your screen would be

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
image.draw();
glutSwapBuffers();

which can be located in the display callback function that would be activated whenever glutPostRedisplay() is called.

Once your image can be shown, you can try whether saving screenshot works as always.

2.2 Create the “Triangle” class

The class (or struct) “Triangle” will contain three point coordinates, three normal vectors, and a pointer to material. This should be short and a header is probably enough.

Triangle.h

#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <vector>
#include "Material.h"
#ifndef __TRIANGLE_H__
#define __TRIANGLE_H__
struct Triangle {
    std::vector<glm::vec3> P; // 3 positions
    std::vector<glm::vec3> N; // 3 normals
    Material* material = NULL;
};
#endif

2.3 Modify the “Geometry” class and its inherited classes

The “Geometry” class consists of the parameters for rasterization pipeline. Each geometry contains some vertex buffer, index buffer, number of vertices to draw, etc. Replace these parameters by simply a list of triangle. (To keep a working copy of the previous HW3 framework, let’s call the new class “RTGeometry” where “RT” stands for Ray Tracing.)

RTGeometry.h

#include <vector>
#include "Triangle.h"
#ifndef __RTGEOMETRY_H__
#define __RTGEOMETRY_H__
struct RTGeometry {...}

#define __RTGEOMETRY_H__

class RTGeometry {
public:
    int count; // number of elements to draw
    std::vector<Triangle> elements; // list of triangles

    virtual void init();
    virtual void init(const char* s);
};
#endif

Modify the implementation of the init functions in RTCube and RTobj classes (modified from Cube and Obj), which are the inherit class of RTGeometry. Instead of filling in the vertex/index buffer data, fill in the list of triangles. Note that the coordinates used in these triangles would be assumed to be under the model coordinate system (which is similar to that the coordinates in the vertex buffer is relative to model coordinate system).

2.4 Modify the “Scene” class into “RTScene”

Scene contains the structure for scene graph. For the scene class, we will no longer need Scene::draw(). But we will still reuse the depth-first search algorithm in the draw function.

In “RTScene” (Ray tracing version of Scene) we need an additional member

• std::vector<Triangle> triangle_soup; // list of triangles in world or camera coordinate

and a member function

• void buildTriangleSoup();

that builds the triangle soup by traversing over the scene graph, apply the proper model matrix to a common coordinate system (world or camera), and assign the proper material for each triangle.

Since we name the modified classes by a different name (RT...), the code should still build and run like HW3.

2.5 Create the “Ray” and “Intersection” classes

A ray (page 19) contains members

• glm::vec3 p0; // basepoint
• glm::vec3 dir; // direction

and an intersection (page 29) contains

• glm::vec3 P; // position of the intersection
• glm::vec3 N; // surface normal
• glm::vec3 V; // direction to incoming ray
• Triangle* triangle; // pointer to geometric primitive (and material info)
• float dist; // distance to the source of ray

Basic ingredients are all set.
2.6 Build your Ray Tracer

Now, build the core functions for the ray tracer

```cpp
namespace RayTracer{
    void Raytrace(Camera cam, Scene scene, Image &image); //page 9
    Ray RayThruPixel(Camera cam, int i, int j, int width, int height); //page 10, 18
    Intersection Intersect(Ray ray, Triangle triangle); //page 30, 33
    Intersection Intersect(Ray ray, Scene scene); //page 11, 28, 31
    glm::vec3 FindColor(Intersection hit, int recursion_depth); //page 15
};
```

Each of the function is straightforward individually.

You can first test whether the ray-triangle intersection is working properly. If the ray casting works, you should be able to obtain the silhouette of the scene.

2.7 Reproduce HW3

Once the ray casting is working properly, you can implement \texttt{FindColor} like a fragment shader (without recursion yet).

At this point, you can demonstrate that the ray tracer produces the same image as your HW3. Note that the rendering could be significantly slower than HW3 since everything is computed in CPU not in GPU.

2.8 Add shadow and recursive mirror reflections

Finally, you can add shadows and recursive mirror reflection. These only amount to modifications of the \texttt{FindColor} function.

To add shadow for each light, shoot a ray towards the light and test if the ray intersect with any other triangle (not including the self triangle). If it hits other triangle that is between the point and the light, then the light is not visible.

To add mirror reflection effect, replace the specular term by the specular coefficient times the color of the hit of the mirror reflecting ray. In particular, \texttt{FindColor} would call \texttt{FindColor} within the function, which is a recursion. Therefore, we will keep track of the depth of the recursion. When the recursion depth reaches the max depth (say 6) then

3 Bonus: Fully Global Illumination

Here, we expand the ray tracer into a full global illumination rendering. In particular, not only that the specular reflection is computed through a recursive ray tracing, but also that the diffuse reflection is computed recursively.

3.1 Simulating photons’ paths from the camera

The idea is that we paint each pixel of the image by the total colored light (in RGB) contributed by all the photons that would travel from every light source to the perspective center through the pixel (see Figure 1).

How would a photon travel in the scene? Instead of simulating their physical trajectory from the light sources to the eye, we trace the trajectory backward from the pixel back to the lights. This latter strategy is more convenient if we only want to collect those rays that pass
The color of each pixel is the average of all colors of rays (paths of photons) through the pixel that connect the vantage point to the light source. A diffuse material scatters the ray at random direction, while a specular material mirror reflects the ray. A path of $n$ bounces randomly picks the diffuse mode or specular mode of reflecting at every bounce; at the last bounce, shade the final hit by the diffuse lighting model; this amounts to directly connecting the ray to every light sources (with visibility check). The ray color at the pixel (the color to be summed to the pixel) is the final hit color multiplied by the diffuse colors (with Lambert cosine law) and/or the specular colors along the journey.

through the pixel. It is also reminiscent to the recursive mirror reflection in the main part of your ray tracer.

**The light path for a concrete number of bounces**

Let us first consider paths with a known number of bounces $n$. To simulate/sample one of these paths, first shoot a random ray from the eye through the pixel. Unlike the main part of your project where the ray pass the center of the pixel, here we take the ray through a uniformly distributed random point in the pixel. This is illustrated in Figure 1.

Next, as it intersects with a surface, we toss a coin (50%-50%) to decide whether we want the ray to reflect like a diffuse reflection or a specular reflection. If we go about the diffuse mode, then we shoot the next ray at a random direction (described more concretely below). If we go about the specular reflection, then we shoot the next ray along the mirror reflection direction.

If we have reached the $n$th bounce, then we terminate the random process. The color of this last hit is computed using the direct diffuse lighting like for local illumination:

$$\text{Color}(hit_{\text{last}}) = \sum_{\ell \in \text{Lights}} \left( \frac{L_\ell D}{\text{attenuation}_\ell} \right) \max(\langle l_\ell, n \rangle, 0)$$

where $D$ is the diffuse color, and $L_\ell$ is the light color of the $\ell$-th light. Effectively, the last bounce forces the light ray to connect to each visible light source, contributing the color at this last bounce with a Lambert cosine weight.
Effectively, the color of this photon path through the pixel (i.e., beginning of the path) is the last color multiplied by the material diffuse/specular color at all the intermediate bounces:

\[ \text{Color}(\text{RayAtPixel}) = W_1 W_2 \cdots W_{n-1} \text{Color(hit\textsubscript{last})}, \]

where the weight \( W_i \) is the diffuse or the specular color of the material at the \( i \)-th bounce depending on whether that bounce is of a diffuse mode or a specular mode.

**Emissive Surface**

If the light is bouncing off from a surface with emission, then we want to add that emission color \( E_i \) to the path color. In a recursive formulation,

\[ \text{Color}(\text{RayAtPixel}) = \text{Color}(\text{RayBetweenHit}_0 \text{AndHit}_1) \]

\[ \text{Color}(\text{RayBetweenHit}_{i-1} \text{AndHit}_i) = E_i + W_i \cdot \text{Color}(\text{RayBetweenHit}_{i} \text{AndHit}_{i+1}), \quad i = 1, 2, \ldots, n-1 \]

\[ \text{Color}(\text{RayBetweenHit}_{n-1} \text{AndHit}_n) = E_n + \text{Color(hit\textsubscript{last})}. \]

Note that we don’t add ambient color like in the shading model for HW2 and the main part of HW4. Our full global illumination with recursive lighting replaces the ambient lighting.

**Diffuse bounce**

Suppose we are bouncing the ray using the diffuse mode; that is, we want to shoot the next ray at a random direction. What should the distribution of the random direction be?

One direct approach is to sample this next ray direction uniformly on the hemisphere (above the surface), like Figure 2 (left). Let \( \mathbf{d} \) be this new random ray direction. If one does so, the weight at the bounce is

\[ W_i = D_i \langle \mathbf{d}, \mathbf{n} \rangle, \quad \mathbf{d} \text{ is sampled uniformly on the hemisphere.} \]

where the inner product is the Lambert cosine term.

Another approach is to sample the next ray direction \( \mathbf{d} \) with a distribution on the hemisphere that is already weighted by the Lambert cosine factor, shown in Figure 2 right. That is, there is a higher probability to shoot a ray near the normal direction than the directions nearly tangent to the surface. In that case, the weight is just the diffuse color

\[ W_i = D_i, \quad \mathbf{d} \text{ is sampled according to the cosine-weighted distribution.} \]

The latter approach will let the final image converge a bit faster than the former uniform sampling.

These hemisphere samplings are obtained by the following mapping formulas. Let \( s, t \) be two independent uniform random numbers in the interval \([0, 1] \). Then a uniform hemisphere sampling of \( \mathbf{d} \) is given by

\[ u = 2\pi s, \quad v = \sqrt{1-t^2}, \quad \mathbf{d} = \begin{bmatrix} v \cos(u) \\ t \\ v \sin(u) \end{bmatrix}. \]
Figure 2 Left: Uniform hemisphere. Right: Cosine-weighted hemisphere.

A cosine weighted hemisphere sampling of \( \mathbf{d} \) is given by

\[
\mathbf{d} = \begin{bmatrix}
\frac{v \cos(u)}{\sqrt{t}} \\
\frac{v \sin(u)}{\sqrt{t}}
\end{bmatrix}.
\]

(9)

Remember to rotate the frame so that the hemisphere is pointing in the normal direction.

3.1.1 Path summation

In the earlier discussion about sampling paths (and their colors) with a fixed number of bounces, we can compute the color contributed by all paths by \( n \) bounces. For each pixel \((i,j)\), we take a large number \( N \) of random paths and take the average of their resulting colors

\[
\text{TotalColorFromPhotonsWith} \ n \ \text{Bounces}_{ij} = \frac{1}{N} \sum_{k=1}^{N} \text{Color(RayAtPixel}_{ij,n,k} \).
\]

(10)

The final pixel color is the sum of the color from all path lengths:

\[
\text{PixelColor}_{ij} = \text{TotalColorFromPhotonsWithOneBounce}_{ij} + \text{TotalColorFromPhotonsWithTwoBounces}_{ij} + \text{TotalColorFromPhotonsWithThreeBounces}_{ij} + \cdots
\]

(11)

In practice, you may truncate this infinite series so that you only compute up to a certain number of bounces (like a maximal recursion depth). In your implementation, you may use this direct truncation method, or the Russian Roulette method.

3.1.2 Russian Roulette

Instead of truncating the path by setting an artificial choice of maximal recursion depth, the Russian Roulette method is a practical method that sums all the terms in the infinite series (11).
First of all, introduce a parameter $0 < \lambda < 1$. Now modify the expression tautologically at first

$$
\text{PixelColor}_{ij} = \frac{(1 - \lambda)\lambda}{(1 - \lambda)\lambda} \text{TotalColorFromPhotonsWithOneBounce}_{ij} 
+ \frac{(1 - \lambda)^2\lambda}{(1 - \lambda)^2\lambda} \text{TotalColorFromPhotonsWithTwoBounces}_{ij} 
+ \frac{(1 - \lambda)^3\lambda}{(1 - \lambda)^3\lambda} \text{TotalColorFromPhotonsWithThreeBounces}_{ij} 
+ \cdots
$$

(12)

Then, instead of picking a fixed path length $n$ and sampling paths with that length, we will let the photon run indefinitely. But, at every bounce, we toss a coin with probability $\lambda$ to decide whether we want to terminate the path tracing. In that way, we have a probability $(1 - \lambda)^{n-1}\lambda$ to obtain a path of $n$ bounces (to survive $(n - 1)$ bounce and to be killed at the $n$th bounce).

By looking at each term of (12), we learn that we should re-weight the resulting color from an $n$-bounce path by a factor of $\frac{1}{(1 - \lambda)^{n-1}\lambda}$. By doing so, the expectation value of the resulting color will unfold into the formula (12), which is the same as what we want (11).

1. ColorSum$_{ij} = 0$;
2. for $(i, j) \in \text{Image}$, $k = 1, 2, \ldots, N$ do
3. Color$_{ij,k} = (0, 0, 0)$; TotalWeight$_{ij,k} = (1, 1, 1)$; Factor = 1;
4. $n_k = 1$; \hfill \triangleright \text{Path length}
5. Ray$_{ij,k} = \text{RandomRayThruPixel}_{ij}()$;
6. while True do
7. hit = Intersect(Ray$_{ij,k}$, Scene);
8. if hit = $\emptyset$ then break;
9. terminate = random $\begin{cases} 1 & \text{probability} = \lambda \\ 0 & \text{probability} = (1 - \lambda) \end{cases}$
10. if terminate then
11. Factor * = $\lambda$;
12. Color$_{ij,k} += \text{TotalWeight}_{ij,k}(E_{hit} + \text{FinalDiffuseLighting}_{hit})$;
13. break;
14. else
15. $n_k += 1$;
16. Factor * = $(1 - \lambda)$;
17. Color$_{ij,k} += \text{TotalWeight}_{ij,k}E_{hit}$;
18. TotalWeight$_{ij,k} *= W_{hit}$; \hfill \triangleright \text{Specular/Diffuse decision}
19. Ray$_{ij,k} =$ bounce into next ray; \hfill \triangleright \text{Specular/Diffuse decision}
20. end if
21. end while
22. ColorSum$_{ij} += \frac{1}{\text{Factor}} \text{Color}_{ij,k}$;
23. PixelColor$_{ij} = \frac{1}{N} \text{ColorSum}_{ij}$.
24. end for
3.2 Specification

This extra credit is open-ended. The minimal requirement is to implement the global illumination, and design your own artistic scene (for example a Cornell box (search this keyword online and you will see what that is)). The objects and their materials should be placed to best display the nontrivial results from your global illumination.