CSCI 3090 Assignment Three
Faking Global Illumination on the GPU
Due: December 01, 2025

Introduction

Global illumination techniques produce high-quality images, but are slow. We would like to do some of the global illumination effects in real time, but in order to do this, we will need to make some simplifying assumptions and use some tricks. In this assignment, you will explore some of the ways in which we can fake global illumination effects. This assignment has two parts. The first two parts are relatively easy, and the second part is more challenging. This assignment is based on example 12, and you are free to use that code in the assignment. You should be able to do everything in the shader code, take advantage of the fact that the C++ program has parameters for the shader file names.

Reflection and Refraction (50%)

We have seen that ray tracing can accurately model reflections and refractions, but in order to do this, it must have access to the entire model when computing the colour of a single pixel. With only the information at a single pixel in a fragment shader, we cannot duplicate these effects exactly. In example 12, we saw how an environment map can be used to simulate reflection under certain conditions:

- The objects that are reflected are distant and can be represented by a cube map.
- There are no other objects occluding the view of the cube map.

It turns out that we can do the same thing with refraction. In example 12, we used the reflect function to compute the reflection vector, which was then used as the texture coordinates. There is also a refract function that has one extra parameter, the index of refraction. We can use the value produced by this function as texture coordinates for the cube map. This does not produce an accurate view through a transparent object, since we are performing a single refraction and not a refraction upon entering and leaving the object. In an interactive application, this is usually good enough. Make this change to the shader from example 12 and observe the result.

Most transparent objects both reflect and refract light depending upon the angle between the surface normal and the viewing direction. In our discussion of classical ray tracing, we said that the fraction of reflected light can be approximated by the Schlick approximation. Use this to combine the reflected and refracted light that is retrieved from the cube map. Your results should be similar to the one shown in Figure 1.

Diffuse Reflection – Irradiance Maps

In class, we mentioned that radiosity produces accurate ambient and diffuse reflections. Unfortunately, the algorithms are complicated and computationally expensive. Again, we would like to have something like the quality of radiosity, but only using the information available at a single pixel. We can produce a reasonable approximation by reviewing how diffuse reflection is computed in the Phong model and extending that using some of the basic sampling. If we recall from the Phong model, the diffuse reflection is proportional to the cosine of the angle between the surface normal and the vector to the light source. Therefore, we get the maximum diffuse reflection when the normal is pointing at the light source, and the amount decreases as we move away from the normal. In reality, light comes in all directions from the hemisphere above the point on the object. This light intensity is summed, but it is weighted by the cosine of the angle between the normal vector and the light direction. Thus, contributions for directions close to the normal are more important than those far from the normal. To do this correctly, we would need

to integrate over the hemisphere, but that is extremely expensive. Instead, we can sample the light intensity in many different directions as an approximation to the integral.

Figure 1 Reflection and refraction from cube map

How do we put this into practice and make it efficient? We've already seen that environment maps are an efficient way of doing specular reflections and refraction, so it is tempting to use the same technique for diffuse reflection. But we can't use the environment map the way it currently is for our purpose; it would give too sharp a reflection. Instead, we want a blurred version of the environment map, and we want to use the normal vector for texture lookup. When we blur an image, we average the pixels in a neighbourhood. If we do this with an environment map, we are averaging the light intensity over a range of angles. This is what we want to do, and the resulting cube map is called an irradiance map.

The computation of this form of irradiance map starts with the environment map. Environment maps are high resolution since we want to capture all the details of the surrounding environment. On the other hand, diffuse reflection changes slowly, so we don't need this level of detail. The images in our environment map are 2048 x 2048, so the first step is to reduce the size of the images to something on the order of 512 x 512. Once this has been done, a blur filter is applied to the image. This process is repeated for all 6 images in the cube map. You can use your favourite image manipulation program for this; I used Gimp.

Figure 2 Sphere with simple diffuse irradiance map

Once we have the irradiance map, we can load it into our program the same way that we loaded the environment map. The program can be modified so it loads in both maps, applies the environment map to the cube and the irradiance map to the sphere (or whatever other object you are using). The result looks like Figure 2.

Diffuse Reflection – Part Two (50%)

There is a problem with the technique from the previous section. We can easily see the seams between the images in the cube map. This is the result of doing the blurring on the individual images in the cube map. The blur doesn't extend across the edges of the images, so we can see where the edges are. You can see some of these edges in Figure 2 if you look carefully. There are several ways that we could solve this problem; one of the obvious ones is to write a program that properly blurs across the edges. We won't take this approach; instead, we will do the averaging in the fragment shader. In this case, we will sample the environment map in the general direction of the normal to compute the diffuse reflection value.

The correct way of computing the diffuse reflection is to integrate over the hemisphere centered on the normal vector. This is far too expensive for a fragment shader, so instead, we will sample the environment map over a range of angles and average the results. The trick is to come up with a sampling scheme that is reasonably efficient and produces a good result. One approach that comes to mind is to randomly generate vectors and use them to sample the environment. We need to make sure that they are sampling the hemisphere above the surface point, which we can do by taking the dot product of the vector with the normal and keeping it if the result is positive. This approach will uniformly sample the hemisphere, but we know that directions closer to the normal contribute more to the result; thus, we need to use a weighted average. Unfortunately, GLSL doesn't have a random number generator, which complicates this approach. I've provided you with a random number generator and the code for initializing it in the main() function of

your fragment shader.
Use this random number generator to randomly generate sampling directions, making sure that the dot product of each random vector with the normal is positive. Weight the result of the texture lookup with the dot product. Also, sum the dot products, and for the final value, divide the weighted sum by the sum of the weights. You will need approximately 500 samples to get a reasonable image.

Assignment Report

The assignment report should include all the source code for your programs and anything else that is required to build your solution. There should also be a written report that describes what you have done, and some sample images produced by your programs. In the second part, how many samples did you need to use to produce a reasonable image? Please also submit any recorded videos. Produce a zip file that contains your report, your program code, shaders, and anything else required to build your program. Name the zip file as "assign3_lastname_firstname". Submit the zip file using Canvas.

Late Submission

A late penalty of 10% per day late will be applied, in the absence of a medical note, to a maximum of 3 days late. (After 3 days, it will not be accepted.)