CSCI 4110 Second Assignment

Faking Global Illumination on the GPU

Due: November 1, 2019

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 three parts. The first two parts are relatively easy and the third part is more challenging.

Reflection and Refraction (4 marks)

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 laboratory five 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 laboratory four 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 laboratory five 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 Schlick approximation. Use this to combine the reflected and refracted light that is retrieved from the cube map. In addition, use a cube with our cube map to provide the background for the object. Note, you will need a separate shader program for the background cube. Your results should be similar to the one shown in figure 1.

Diffuse Reflection – Part One (4 marks)

We have seen that radiosity does a good job of diffuse reflections, but it involves computing form factors and dealing with large matrices. 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 extend that using some of the basic ideas from radiosity. 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 radiosity we collect all the light information for 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 the ones far from the normal. Therefore, we only need to sample directions close to the normal vector.

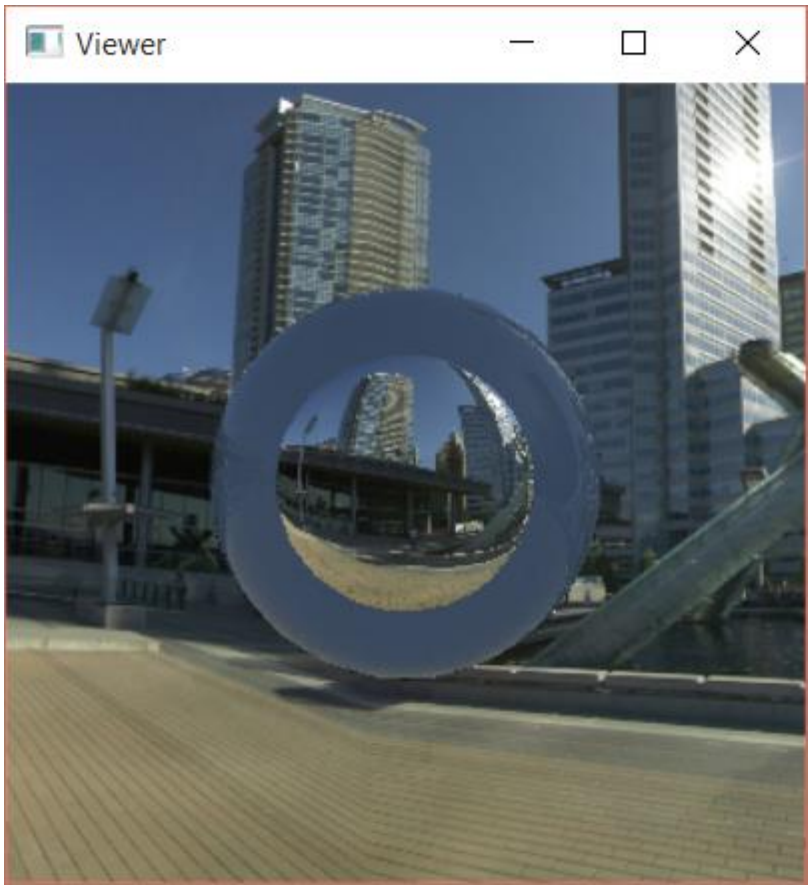

Figure 1 Reflection and refraction from cube map

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.

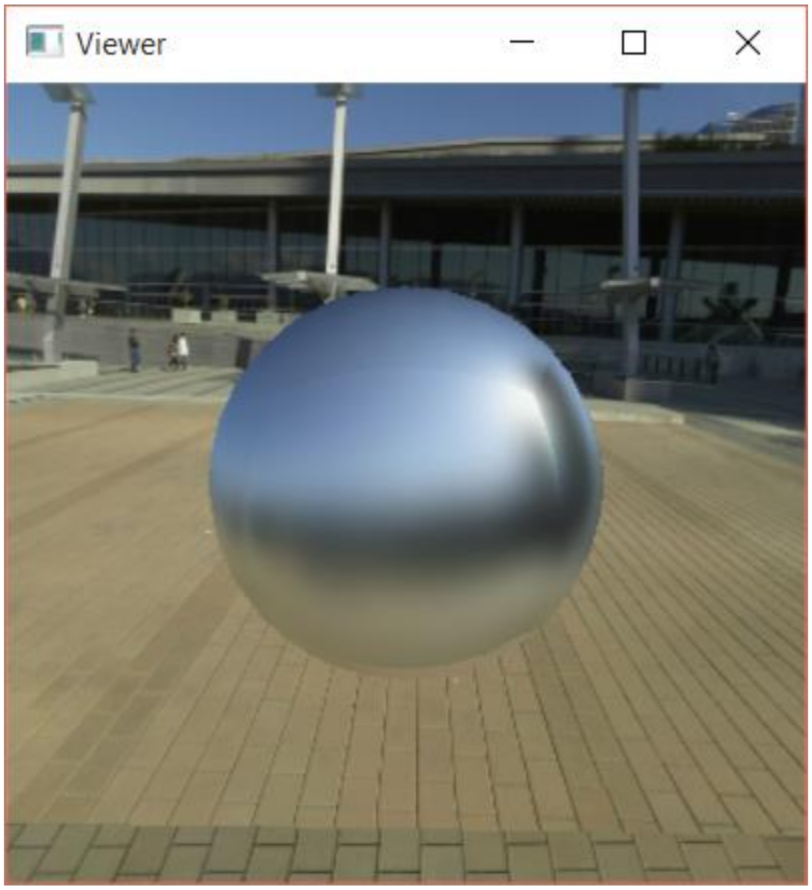

Figure 2 Sphere with simple diffuse irradiance map

Once we have the irradiance map we can load it into our program the same way that we loaded the environment map. Modify the program 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). You will also need to modify the fragment program that is applied to the sphere. Your result should look something like figure 2.

Diffuse Reflection – Part Two (2 marks)

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 produce 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. Unfortunately, GLSL doesn't have a random number generator, which complicates this approach (there are implementations on the web that could be used).

Instead we can use a uniform sampling strategy based on sampling directions close to the normal. Consider building a circle of vectors around the normal direction. This circle will have a radius (L), say around 0.2 and uniformly distributed angles (θ). We need two orthogonal vectors U and W that are both orthogonal to the normal. We can start by switching two of the components of the normal and taking the cross product with the normal, this can be U. We can get W by taking the cross product of U with the normal. We can construct a new vector V in the following way:

$$V = L\cos(\theta)\,U + L\sin(\theta)\,W$$

The sampling direction is given by N+V, which must be normalized before it can be used as a texture coordinate. Where N is the normal.

You can write a shader procedure to do the sampling and have the number of L and θ values as parameters to your shader. You can use this to determine a reasonable number of vectors to be used in the sampling.

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, some sample images produced by your programs and some analysis of the results from the third part of the assignment (the number of samples and how they should be distributed).