Minecraft++

Goal:

Our goal was to create Minecraft with ray tracing on the GPU to add features like color bleeding, reflections, and realistic looking soft shadows to make a more realistic lighting model. We attempted to make the performance of our program real time, but the photon mapping requires a lot of time to get good textures, although the program runs pretty fast if we have a low resolution for our photon map, but you can see obvious artifacts of photon mapping .

Theory:

A brief synopsis of the theory of our algorithm goes as follows: We raytrace the image using the traditional ray tracing algorithm we discussed in class using the two following formulas:

Phong Shading Model

$$I_{phong} = k_e + k_a I_a + \sum_j \left[I_{l_j} \left[k_d (\mathbf{N} \cdot \mathbf{L}_j)_+ + k_s (\mathbf{V} \cdot \mathbf{R}_j)_+^{n_s} \right] \min\left\{ 1, \frac{1}{a_0 + a_1 d_j + a_2 d_j^2} \right\} \right]$$

Raytracing Model

$$I_{pixel} = I_{phong} + k_r I_{reflect} + k_t I_{transmit}$$

Then we use photon mapping to approximate the rendering equation over a certain radius. Our algorithm uses the fact that we can approximate irradiance and substitute it into the rendering equation to approximate the light at a certain point using a sample of photons using the following equations:

$$E_i = \sum_{i=0}^{n} \frac{\Phi_i}{A} \qquad L_i(\vec{\omega}') = \frac{dE_i(\vec{\omega}')}{\cos\theta_i d\vec{\omega}'} \qquad L_r(x, \vec{\omega}') = \int_\Omega f_r(x, \vec{\omega}', \vec{\omega}) dE_i(\vec{\omega}')$$

Although we don’t implement these exact formulas in our code, we do use these concepts to create an even further approximation of the rendering equation with photon maps for the sake of speed. You could call it an approximation of an approximation. From here we use UV coordinates to map the color from the raytraced texel to the pixel on the screen. One we read from the texture the process is finished, and we get a pretty good semi real time approximation for global illumination.

Implementation:

We start by using two compute shaders to simulate bidirectional photon mapping. The first compute shader marks spots hit by photons up to N bounces shot directly from the light source. This creates a texture where we can look up which exact points have been marked by photons. From here, we pass the work to another compute shader which handles photon scattering. To do this more efficiently, what we do is for each point, shoot rays in random direction[2] (using a random function we found on stack overflow) until we hit a point marked by a photon. When we hit a photon, we calculate the color using the attenuation of distance, and diffuse light contributions of all objects that rays hit in the traced path. One resource that we used to help understand how to set up compute shaders was a tutorial[1] that we found on GitHub. This tutorial basically taught us how to set up ray tracing on the GPU and how to use compute shaders, and gave us the idea to store the colors into a texture, which we can then render normally on the GPU. The last compute shader that we used did the ray tracing part. What this shader essentially does is shoots a ray from the eye to all pixels on the screen, looks up the contribution of photons to the light at that point, and then applies the Phong shading model given the incoming light intensity of the point that the ray hits. In order to get the performance to be close to real time we had to add some hacks into our code to make it a lot faster. One of the hacks we implemented was shooting the photons bidirectionally so we could have more control over how many rays we would distribute over each point in space since doing recursion on the GPU is not possible. Another hack that we implemented to be able to shoot less photons and decrease the resolution of our photon map texture, was we interpolated color values between two points on the texture so that we could have less of the artifacts of photon mapping and it would look more smooth even if we did not have enough samples. One known problem that we have is that we sometimes have over exposure if we do not tweak our parameter values in our photon map resolution and in the distance attenuation amount in our color interpolation.

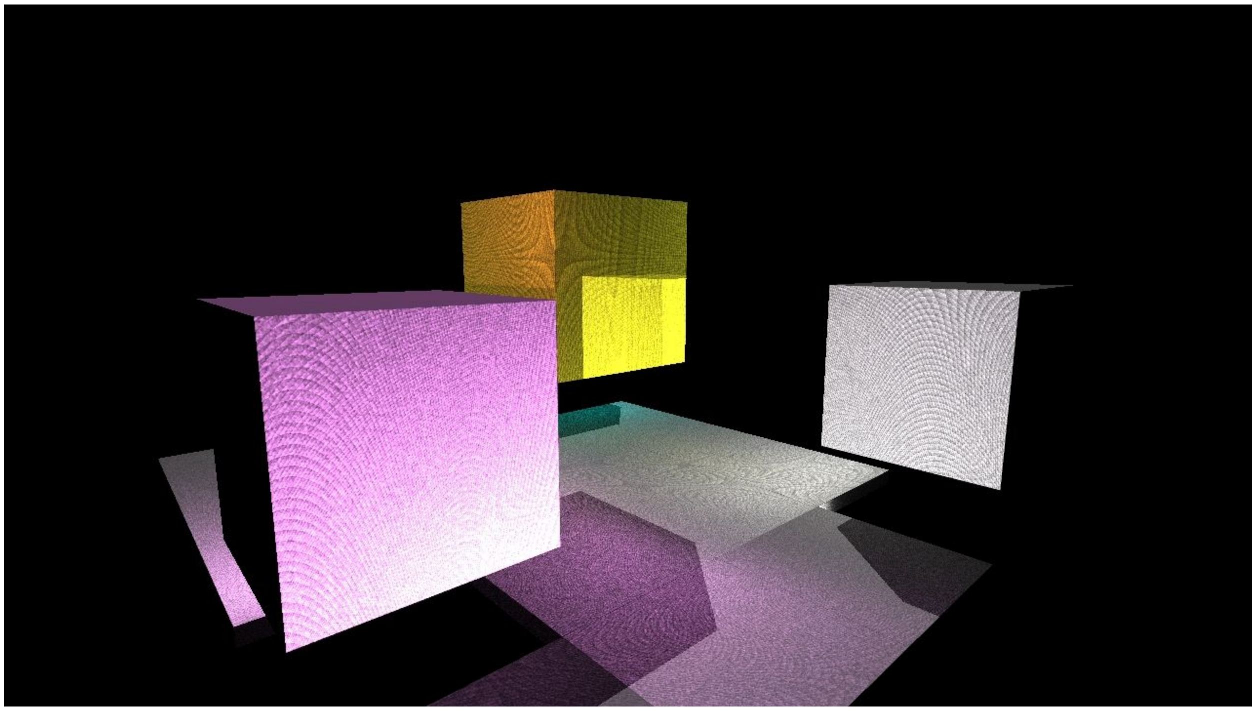
To-do:

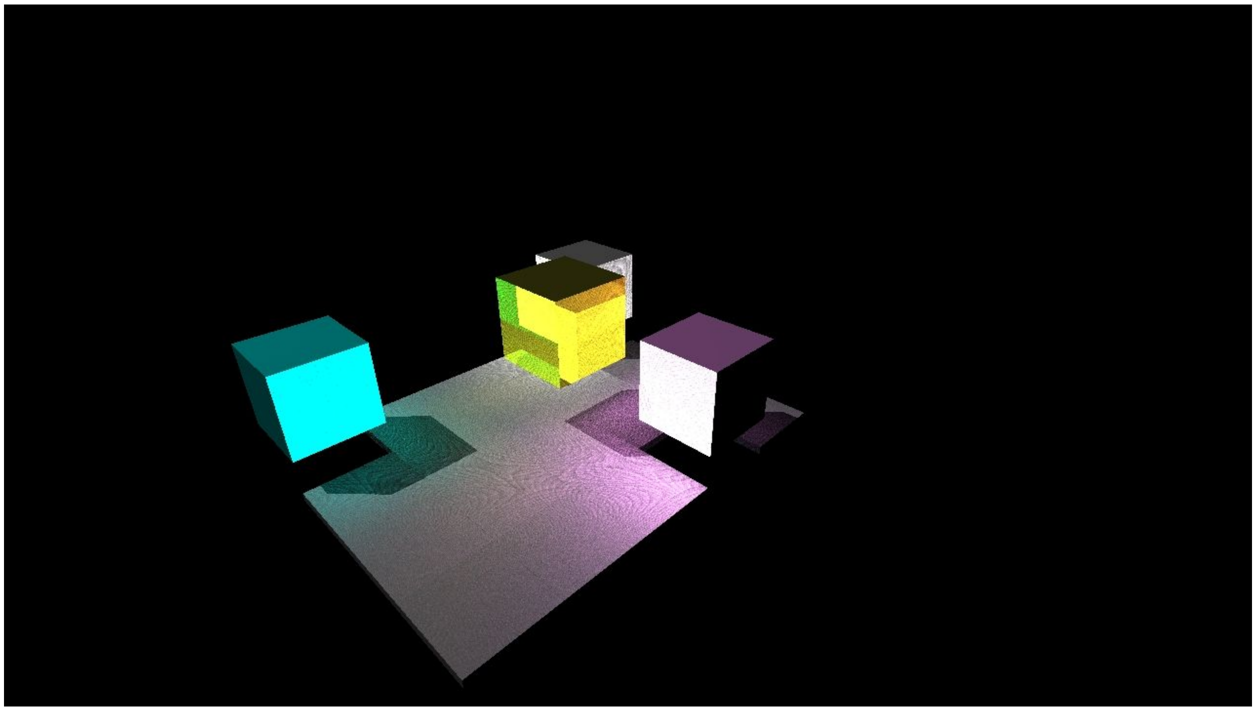
There are some features that we are in the middle of implementing that we have not finished quite yet, but might have done before the presentation. Right now we have a skeleton of a BVH on the GPU to accelerate the process significantly. We also have not cleaned the code up.

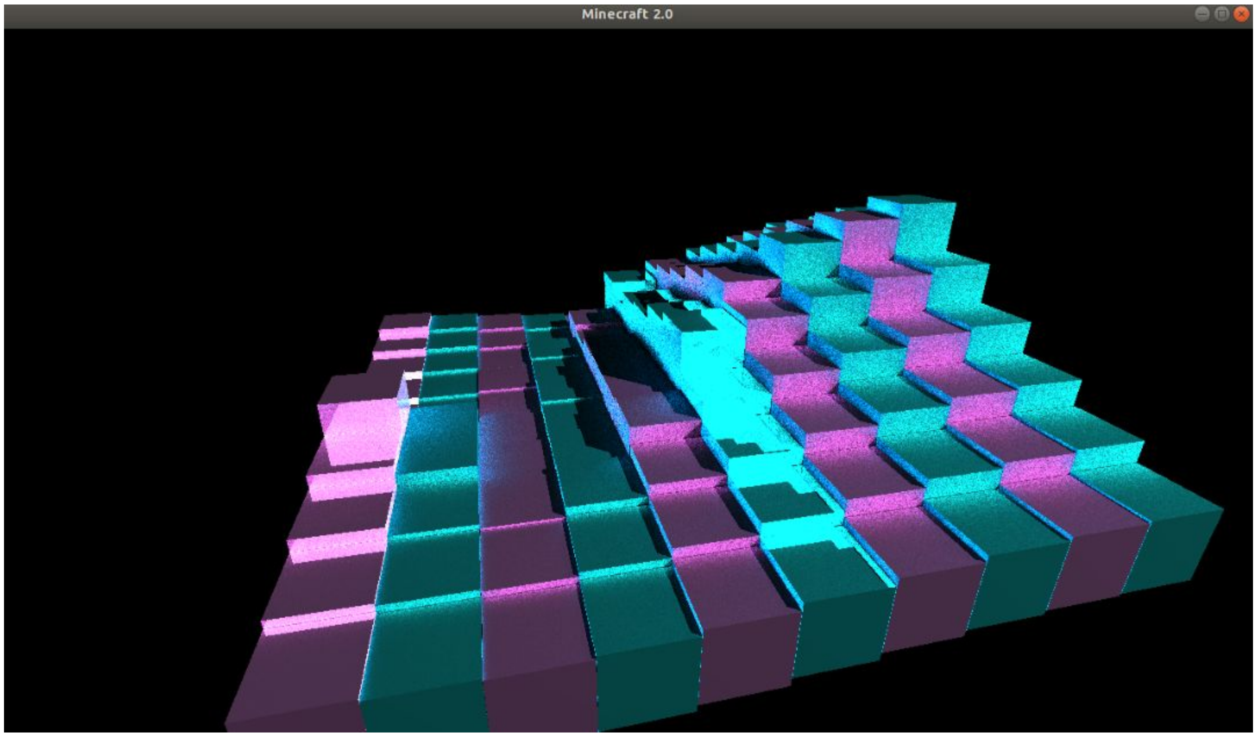
Resources:

[1][**https://github.com/LWJGL/lwjgl3-wiki/wiki/2.6.1.-Ray-tracing-with-OpenGL-Compute-Shaders-%28Part-I%29**](https://github.com/LWJGL/lwjgl3-wiki/wiki/2.6.1.-Ray-tracing-with-OpenGL-Compute-Shaders-%28Part-I%29)
[2] [**https://stackoverflow.com/questions/4200224/random-noise-functions-for-glsl**](https://stackoverflow.com/questions/4200224/random-noise-functions-for-glsl)

Screenshots:






Minecraft 2.0