Assignment 8: Photon Mapping

CS148 Fall 2019

Due Date: Up to Monday, 25 November 2019 by 7pm

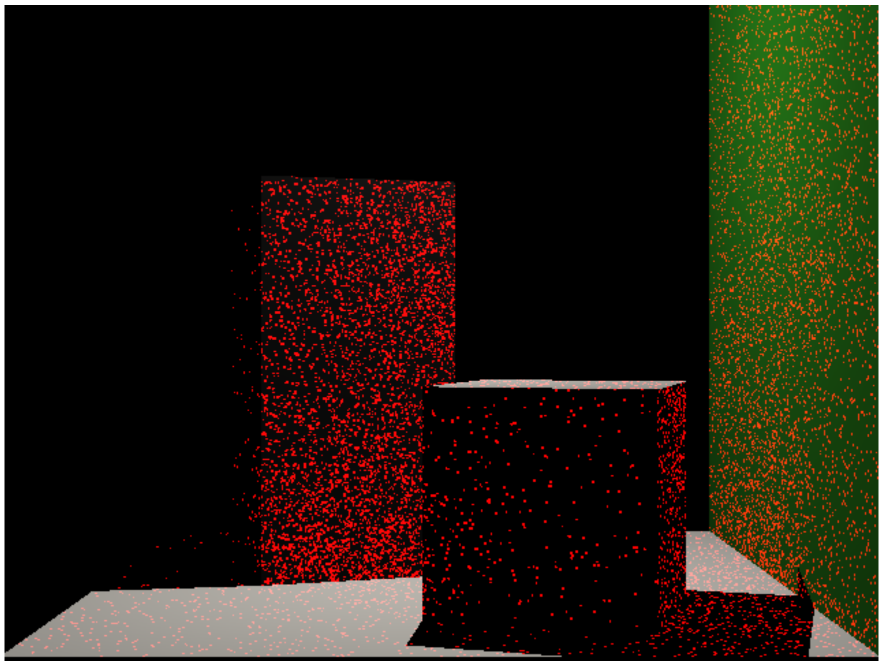

Figure 1: The instructor generated result for this assignment. Your results should be similar.

Introduction

This week we will explore the topic of photon mapping. Photon mapping is one method by which you can achieve global illumination effects such as diffuse interreflection and caustics. Currently, the ray tracer uses a "backward" ray tracing algorithm. This means that you shoot rays from the camera out into scene and compute the color of the sample wherever the ray hits. Photon mapping, on the other hand, shoots rays out from the lights and stores what we call "photons" around the scene. In this assignment, you will explore how to create and scatter photons from point lights into a purely diffuse scene. The diffuse scene you will be using is a stripped down version of the Cornell Box where the back wall, left wall, and ceiling have been removed. As always, if you find a bug, please report it on Piazza. Start early and ask questions if you have any.

Assignment

Photon Emission

As mentioned earlier, we will only generate photons for point lights. The function that will generate photons for a given light is 'GenerateRandomPhotonRay.' You can find the function definition for point lights in 'common/Scene/Lights/Point/PointLight.cpp.' For every photon that we will send out from the light, 'GenerateRandomPhotonRay' is called. This function takes in a reference to a 'Ray' object and sets the position and direction of the ray. You will find the functions '**Ray**::SetRayPosition' and '**Ray**::SetRayDirection' to be helpful.

For a point light, the origin of the ray is fairly obvious: it is just the position of the light. Note that the 'Light' class inherits from 'SceneObject' which contains a helpful '**SceneObject**::GetPosition' function that you can use.

The randomness for point light photon generation comes in the direction we choose. We will use the method recommended by Jensen in his SIGGRAPH 2000 course, "A Practical Guide to Global Illumination using Photon Maps."

The pseudo-code for generating the ray direction is as follows (taken from the above course):

```
do {
    x = random number between -1 and 1
    y = random number between -1 and 1
    z = random number between -1 and 1
} while ($x^2 + y^2 + z^2 > 1$)
```

You would then set the ray direction to be the vector with components x, y, and z. Don't forget to normalize the vector! You will find the rand function to be helpful. Note that this function generates a number (int) between 0 and RAND_MAX so do not forget to scale appropriately.

Photon Storage

Starting from this subsection, we will begin to flesh out the 'TracePhoton' function in 'common/Rendering/Renderer/Photon/PhotonMappingRenderer.cpp.' This function will be responsible for a number of tasks:

- Storing the photon when it hits a scene
- Making the decision on whether to scatter or absorb the photon

We will take care of the storage step first. As mentioned in lecture, there exists a maximum number of photon bounces to make sure that the photon map generation step terminates. It is by default set to 1000. The 'TracePhoton' function has a parameter named 'remainingBounces.' Once this value becomes **LESS THAN** 0, the 'TracePhoton' function should terminate.

If the 'TracePhoton' function is not terminated, you will want to trace the photon ray into the scene. Under the line that says:

```
state.currentIOR = currentIOR;
```

you will want to use the stored scene to trace the photon ray and test for intersections. You can access that functionality by using the 'storedScene' pointer in the RendererObject:

```
storedScene->Trace(photonRay, &state)
```

Note that 'Scene::Trace' returns a boolean as to whether or not there exists an intersection. If no intersection exists, then 'TracePhoton' can end. However, if there was an intersection then you will want to make a decision as to whether or not to store the photon and whether or not to scatter the photon. Generally, you will want to store the photons that have come directly from the light as well as those that have bounced around the scene in the photon map; however, in our case we will only store those that have bounced around the scene.

Initially (when the photon gets shot out from the light), the 'path' variable (which has type: 'std::vector⟨char⟩') has one element in it, 'L'. Since we do not want to store photons that come directly from the light, you will not want to insert a photon into the photon map when the 'path' variable has a size of 1. Otherwise, you will want to create and store a new 'Photon' object in the photon map. The 'Photon' struct has three properties: position, intensity, and toLightRay. For the purposes of this assignment, setting 'position' is sufficient but for completeness you should fill out all three variables. The 'intensity' variable should be set to 'lightIntensity' and the 'toLightRay' variable is a ray that points in the OPPOSITE direction of the photon ray. You can compute the intersection point using the information stored in the 'IntersectionState' object, assuming an intersection happened:

```
const glm::vec3 intersectionPoint =
    state.intersectionRay.GetRayPosition(state.intersectionT);
```

Insert the photon into the photon map and that is it for this step! Note that we already currently assuming a purely diffuse scene; in a more general-case photon mapper, you will also want to make sure that you are hitting an object that has a diffuse component before storing the photon.

Photon Scattering/Absorption

Next, we will implement a technique called 'Russian roulette' to determine whether or not a photon gets scattered or absorbed. This should be implemented in the 'TracePhoton' function. Once again, we will use the method described by Jensen in his SIGGRAPH course (link in the previous section). Given that your object has some diffuse response d, the probability of reflection is:

$$P_r = \max(d_r, d_g, d_b) \tag{1}$$

where d_r is the red component of the diffuse response, d_g is the green component, etc. Note that this is the diffuse response, not the diffuse color after the scene's lighting is considered. In terms of the Blinn-Phong material, this is simply the 'diffuse color.' You will find the function '**Material**::GetBaseDiffuseReflection' to be helpful. To access the material that you hit you can use the following code:

```
const MeshObject* hitMeshObject = state.intersectedPrimitive->GetParentMeshObject();
const Material* hitMaterial = hitMeshObject->GetMaterial();
```

Now generate another random number in the range $[0, 1)$. If it is less than P_r, then you will want to scatter the photon, and if it is greater, you will do nothing. Finally, you will have to implement diffuse reflection. Unlike specular reflection which will do a perfect reflection across the normal every time, diffuse reflection will shoot a ray out randomly into the hemisphere above the intersection point. To accomplish this, you will need to do two things:

1. Perform a generic hemisphere sample
2. Transform the ray into world space.

Hemisphere Sampling

The easiest way to sample a hemisphere is to sample a disk and project it into the space of the hemisphere. Assuming that you have two random numbers in the range $[0, 1)$ called u_1 and u_2, you can sample a disk using the following equations:

$$r = \sqrt{u_1} \tag{2}$$
$$\theta = 2\pi u_2 \tag{3}$$

Then to get the x, y, and z components of the ray:

$$x = r\cos\theta \tag{4}$$
$$y = r\sin\theta \tag{5}$$
$$z = \sqrt{1-u_1} \tag{6}$$

Don't forget to normalize your ray direction!

Hemisphere Ray Transformation

The above equations sample the hemisphere above a point but they do so in tangent space (you may remember this from normal mapping from the OpenGL part of the class). In normal mapping, we wanted the tangent space to be dependent on the UV's to get consistent results across the mesh, but we have no such restrction here so we can just compute two vectors orthogonal to the normal. So assuming you have a normal n (you can compute this using 'state.ComputeNormal()'), you can compute the tangent, t, and bitangent b vectors as follows:

$$t = n \times (1, 0, 0) \tag{7}$$
$$b = n \times t \tag{8}$$

where $(1, 0, 0)$ is a vector of unit-length in the x-direction and $\times$ is the cross-product operator. However, be careful! If n is close to parallel with $(1, 0, 0)$ you will get poor results. In that case, just choose a different vector like $(0, 1, 0)$. Hint: An easy way to determine if two unit length vectors are parallel is if their dot product is close to 1. Now construct a 3x3 matrix where the columns of the matrix are the tangent, bitangent, and normal vectors (in that order) and multiply it by the vector you got from the previous step to get your diffuse reflection ray direction in world space! Now trace the diffuse reflection ray into the scene and determine whether you want to store the photon again!

Final Implementation Notes

- The 'TracePhoton' function takes in a bunch of parameters, you only have to worry about the 'photonRay', 'path', and 'remainingBounces' parameters. The other parameters you can leave unchanged.
- 'photonRay' changes to become the diffuse reflection ray after you perform a diffuse reflection step the next time you call 'TracePhoton'.
- The contents of 'path' do not really matter, just make sure you grow the size of 'path' so you can differentiate between photons that came directly from the light versus those that came indirectly.
- 'remainingBounces' should be decremented by 1 every time you call 'TracePhoton.'
- Yes, you should be making recursive calls to 'TracePhoton.'

Grading

You will be graded as follows:

- 4 – Image has dots that look reasonable.
- 2 or 3 – Image has dots but has dots that look incorrect.
- 1 – Show up to the grading session and discuss what you tried (no dots).
- 0 – Do not show up to the grading session. :(