Graphics Programming HW 5

Due: 5/16 11:59 PM

Prob 1. Basic Ray Tracing (150 Points)

Ray tracing is a rendering technique for generating an image by tracing the paths of light from image pixels and simulating the effects of their encounters with virtual objects. In this assignment, you will implement a basic ray tracer!

We will implement simple ray tracing using a fragment shader. Unfortunately OpenGL, what we have learned so far, is a rasterizer. It is not designed to do ray tracing. Accessing geometry information inside the shader is quite a challenging task. It is also hard to constructing acceleration data structure. Furthermore, because GLSL does not support recursive functions, it is difficult to implement recursive reflections and refractions. This is why ray tracing has been considered to be not ideal for real time graphics in the past few years. In order to implement the true ray tracing, you had to use CPU or GPGPU such as OpenCL or CUDA. However, using CPU takes so much time and you will waste a lot of time for debugging. Learning new APIs (OpenCL, CUDA or OptiX) can be also quite burdensome and we decided to stick to using OpenGL that you are familiar with by now. (Of course using OpenGL requires a lot of hacks and tricks, but we will step through basics together.)

We start from rendering a fullscreen quad. In the fragment shader, we generate rays and compute the ray-geometry intersection. Because recursive ray tracing is not possible within OpenGL, alternatively we propose an iterative way. The steps below follow the process within Optix quick start guide[1]. Although it is written for an Optix user, it is also a helpful reference when implementing a basic ray tracing step by step. You can also visit Scratchapixel. It is a useful resource to understand ray tracing. The detailed steps are provided below.

- First, generate rays in the fragment shader using `TexCoords` input. We assume a pinhole camera. Camera's orientation, fovY and screen ratio would be required.
- Fig. 1-(a) Implement intersection functions for primitive geometries (box, plane, sphere and triangle). Also implement a function to find the closest-hit surface. You have to store the information describing the hit point (distance, normal, material), so you can use it later. If the ray intersects with any geometry, it would be colored someway. Now, let's just set it as material's diffuse. If the ray doesn't hit anything, set the color to a specific value (miss color).
- Fig. 1-(b) Now let's add basic Phong shading. The coefficients for Phong shading is determined from the hit surface. Don't consider reflection, refraction or shadow in this step. Also assume point lights only.

[1]Full code is available in NVIDIA OptiX homepage. Go to homepage and download OptiX. (Version 6.5 is recommended.) You can find the code in SDK/optixTutorial folder.

- Fig. 1-(c) Let's now add shadows. Phong shading does not give us shadows. To determine the point is shadowed or not, we have to check whether the point is visible to the light source. To do so, we conduct additional ray tracing, starting from the hit point toward the light source. If the ray does not hit anything, then it means the light is visible. If ray hits something, then it means the current point is shadowed.

- Fig. 1-(d) Now let's add a perfect mirror reflection. Note that this should be done in a recursive way. However, because GLSL does not allow recursion, we have to do it in an iterative way. Change the recursion to a for-loop. Don't forget to set the maximum bounce number to prevent an infinite loop.

- Fig. 1-(e) Adding an environment map is quite simple. Just change the miss color (color of the ray that doesn't hit anything) to the texture value fetched from the environment map, or skybox.

- Fig. 1-(f) Add richness to reflection using the Fresnel effect. Use the Schlick's approximation.

From now, it is optional.

- Fig. 1-(g) Implement a refractive dielectric material such as glass. Here, we will consider refraction and reflection simultaneously. Some fraction of incident light refracts while other reflects. Thus, we have to trace rays in a tree structure. (Note that for perfect mirror reflection, ray doesn't make tree.) Although it can be done in an iterative way (using stack), we will use Monte-Carlo approximation. Sample multiple rays for a single pixel with small random jitter. Use a factor derived from the Fresnel effect to decide the probability for a ray to reflect or refract. You then have to average all rays to get the final result. Don't forget extinction because of the Beer-Lambert law. The shadow should be also changed. Find the trick to use in Chapter 11 in this document.

- Fig. 1-(h) We will now import an obj file and render a mesh. Assuming a mesh has been already triangulated, we can think a mesh as a set of triangles. Since we do not use previous VAO, VBO pipeline, we have to transfer triangle information to shader in a different way. You may pass triangles as uniform variables to the fragment shader, but there is a limit for the size of uniform variables. So you have to use uniform buffer object, texture buffer object, or shader storage buffer object. I used uniform buffer object, but of course you can use other methods. You can find explanation in LearnOpenGL Because multiple meshes would cause a lot of computational cost, assume there is only a single mesh. This problem is to let you realize that we cannot use shader for ray tracing when there exists a complex mesh. Also assume triangles in a mesh have only single material. Don't consider normal or texture uv. Just consider vertices and render them in flat shading. You can try other models, but make sure that the number of triangles for your model is under certain value! (From my experience, icosphere with only 80 triangles was burdensome.)

(100 Points) Followings are required features, and each corresponds to Fig. 1-(a) to (f). We just provide the outline. You can implement details at your own discretion.

(a) (35 points) Generate rays on a fragment shader and do ray tracing on a sphere, a box, a triangle and a plane.

(b) (20 points) Implement simple Phong illumination (ambient, diffuse and specular).

(c) (20 points) Implement a shadow using shadow rays.

(d) (15 points) Implement recursive reflection.

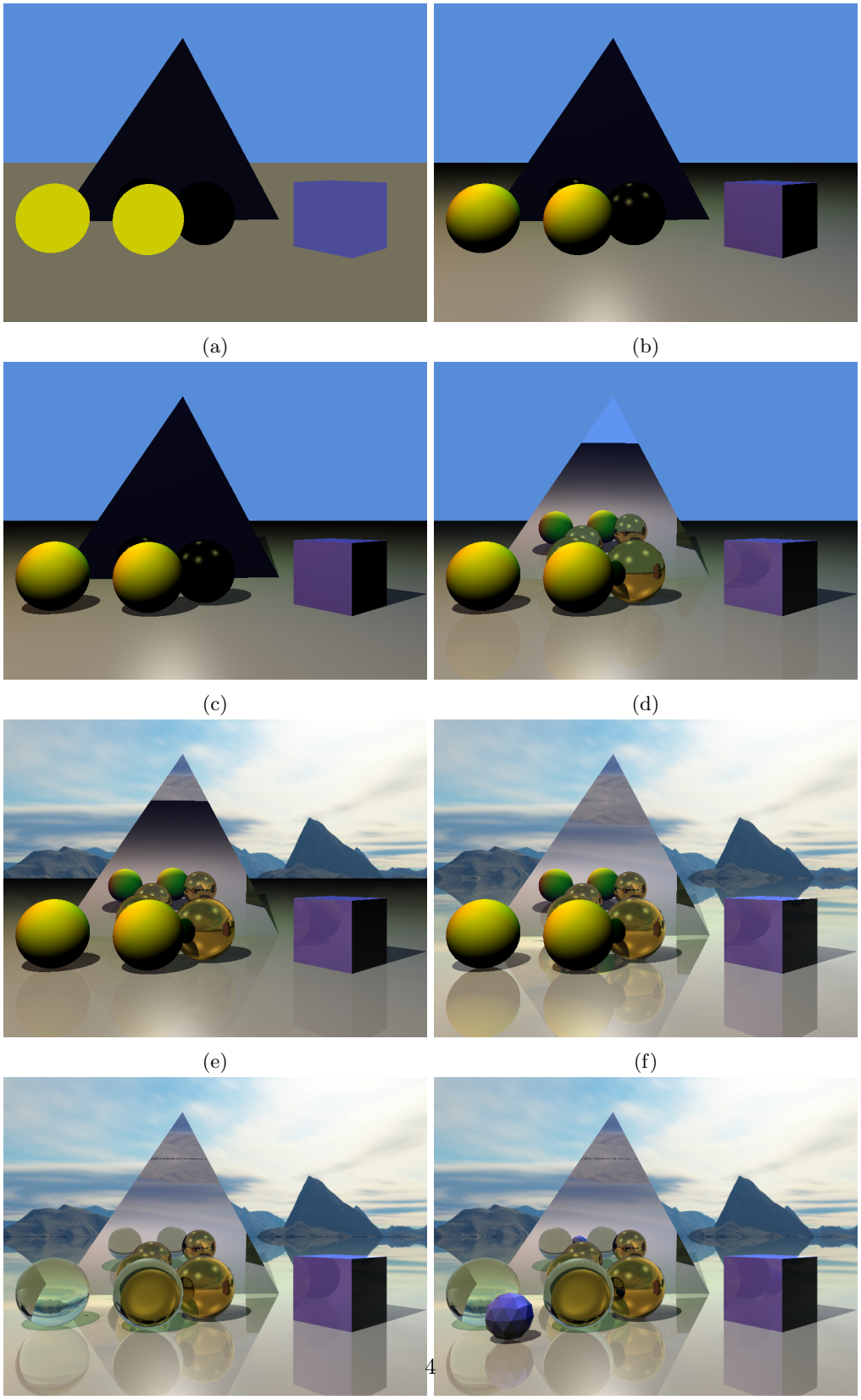
(e) (5 points) Implement an environment map lighting.

(f) (5 points) Add Fresnel effect to the reflection.

(20 Points) Followings are additional features. First two of them are already mentioned in detail. Implementing other features is also okay, but please implement something that is as meaningful and technically challenging as the example. You can get 10 extra points for each item, total maximum 20 points.

- Do recursive refraction and reflection simultaneously (dielectric material like glass). Shadow of the refractive object also have to be changed.
- Ray-trace a complex mesh.
- Implement more advanced ray tracing algorithms/techniques such as path tracing that uses BRDF importance sampling.
- Implement a soft shadow with an area light. In this case, you have to implement a light sampling.
- Implement accelerating data structures such as BVH (bounding volume hierarchy).
- Implement texture mapping.
- Other features at your discretion.

(30 Points) You also have to submit a short report (1-2 page) describing what you've implemented and some screenshots. A function for taking a screenshot is already given in the skeleton code. Just use it. You need FreeImage library.



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 1: Steps for implementing ray tracing.