Assignment 1: Whitted Ray Tracing

EDAN30

March 23, 2011

This assignment will introduce you to some of the basic concepts of ray tracing. In order to pass the assignment you need to complete all tasks. Make sure you can explain your solutions in detail.

1 Introduction

Start out by opening your prTracer-project and making sure that it compiles and runs without any errors. This initial incarnation of the ray tracer is only capable of shooting eye rays and detecting whether they hit any spheres in a scene or not. After the program has finished running, you should find an image file *output.exr* in the root directory of the project. The image should be white for where eye rays hit any sphere, and black otherwise.

2 Scene setup

For this assignment you should use the "spheres" scene. It can be found in `main.cpp`.

3 Diffuse Reflection

Right now there is no light transport at all in the scene. It would definitly be more interesting if we would be able to apply diffuse shading to the spheres. For now we only consider *direct illumination* that originates from a point light source.

The radiance that reaches the viewer along the (negative) eye vector Θ can then be computed as;

$$L_{out}(x \rightarrow \Theta) = L_{direct}(x \leftarrow \Psi) f_r(x, \Psi \leftrightarrow \Theta) cos(N_x, \Psi) \quad (1)$$

In the prTracer, when a ray intersects a sphere, an `Intersection` object is returned. This object contains useful information about the ray-sphere intersection event. Amongst other things, you can retrieve the world position and normal of the hit point, as well as the material of the object.

Task 1: *Use this information to implement diffuse shading using direct illumination from a single point light source.*

Task 2: *Add more than one light source to your scene.* By simply summing the radiance contribution of equation 1 for each light source you will get the correct result.

$$L_{out}(x \rightarrow \Theta) = \sum_{lights} L_{direct}(x \leftarrow \Psi_i) f_r(x, \Psi_i \leftrightarrow \Theta) cos(N_x, \Psi_i)$$

4 Shadows

So far shadows are not considered. If there is an object between the hit point and the light source, there is no direct illumination, and thus the L_{direct}-factor should be set to 0. By sending a *shadow ray* from the hit point to the light source and checking for intersections, we can determine if the hit point is in shadow or not.

Task 3: *Implement shadow rays by shooting rays from the hit point towards the light sources.* Note that the intersection test only has to return a boolean, *true or false* answer, since we are not interested in any intersection information for shadow rays.

5 Reflection

The next step to achieve more realistic looking images is to add reflections. Real world materials are of course neither perfectly diffuse nor perfectly specular, but a combination of the two components can give fairly convincing polished materials.

Similar to shadow rays in the last excersise, a reflectance ray can be spawned at the point of intersection. This way, a ray originating from the eye can be traced recursively to account for alterations in the ray path caused by reflections. Now that we have reflection we get the following;

$$L_{out}(x \rightarrow \Theta) = (1-r) \sum_{lights} L_d(x \leftarrow \Psi_i) f_r(x, \Psi_i \leftrightarrow \Theta) cos(N_x, \Psi_i) + r L_s(x \leftarrow R)$$

This equation contains a new parameter, the *specular reflectance* (r). For this assignment it's enough to assume that r affects all wave lengths equally, and thus is a single coefficient $\in [0, 1]$. If $r = 0.0$ it means that a surface is not reflective and $r = 1.0$ means that it is a perfect mirror. Notice that we linearly interpolate the resulting color using r.

Task 4: *Implement specular reflection, where each intersection where $r > 0$ spawns a new reflectance ray.* Important note: It is possible for a ray to get "trapped" in an infinite series of reflections, so we introduce some stopping criteria. The easiest solution is simply terminating the ray tracing at a fixed recursion depth.

Task 5: *Add a few more spheres to the scene and play around with different r parameter values.*

6 Refraction

Another important feature of a ray tracer is the ability to handle transparency and refractions. Many real materials are more or less transparent (glass, plastic, liquids, etc). When light enters a transparent material, it is usually bent or *refracted*. How much is determined by the index of refraction of the material. By using Snell's law we can compute the refraction vector T (For more details on refraction see the lecture- and seminar notes.).

Similar to the reflection term, we add the refraction term to our light transmittance model as follows;

$$L_{out}(x \rightarrow \Theta) =$$
$$(1 - r - t) \sum_{lights} L_d(x \leftarrow \Psi_i) f_r(x, \Psi_i \leftrightarrow \Theta) cos(N_x, \Psi_i) +$$
$$r L_s(x \leftarrow R) + t L_t(x \leftarrow T)$$

Just like for reflection, a refraction ray can be traced by (recursively) spawning a new ray at the hit point of a refractive surface where $t > 0$. Like before, we interpolate between the direct illumination, reflection and refraction components, so it should hold that $r + t \leq 1$.

Task 6: *Implement refraction in your ray tracer.*

Task 7: *Try different combinations of the r and t parameters.*

7 Super Sampling

All of the images produced so far appear very jagged when examined at close up. This is because we are only tracing a single ray through each pixel. To get a smoother result, it is necessary to use some kind of super sampling.

Task 8: *Implement stratified grid sampling to produce anti-aliased images.* Note that the output color should be the *average* of the color returned by the samples. Unfortunately, performance scales linearly with the number of samples you use. You can try to use 3×3 samples per pixel, and then zoom in on the silhouette of a sphere to see the improved result. It may also help to lower the recursion depth cutoff for reflection/refraction rays (3 or so should be enough).

8 Ray-Triangle Intersection Testing

Only being able to trace spheres will not make for very interesting scenes. Triangles are more appropriate building blocks for complex models, since they are the simplest primitive that can be used to define a surface in space.

Complex objects can consist of meshes with millions of triangles (or more!).

Hidden in the "spheres" scene is a plane consisting of two triangles. You cannot see it yet, because your ray tracer doesn't know how to test rays and triangles for intersection. One way to perform ray-triangle intersection testing is to calculate the intersection point of the ray and the triangle plane, and then check if the point is inside the triangle using barycentric coordinates.

Task 9: *Implement ray-triangle intersection testing (make sure to include a boolean test for shadow rays).* You will find a detailed description of ray-triangle intersection in the seminar slides.

9 Blinn-Phong Shading (Optional)

Instead of just having diffuse surfaces, a more commonly used BRDF is the Blinn-Phong shading model. We go back and look at equation 1 again;

$$L_{out}(x \rightarrow \Theta) = L_{direct}(x \leftarrow \Psi) f_r(x, \Psi \leftrightarrow \Theta) cos(N_x, \Psi)$$

where f_r is the BRDF. Since only diffuse surfaces were considered so far, the BRDF is just a constant; $f_r(x, \Psi \leftrightarrow \Theta) = k_d$. What this means is that the light is not concentrated in any particular direction, but outgoing light is equally distributed in all directions. Adding Blinn-Phong shading, however, gives us an additional glossy specular reflection component.

Add a new Blinn-Phong material to your ray tracer. Note that this is an **optional excersice** that you should only attempt if you are finished with the rest of the assignments.

By adding a Blinn-Phong specular highlight for direct illumination to the prTracer, we actually end up with two types of specular reflection – the Blinn-Phong one (glossy) and the Whitted one (perfectly specular). This is of course not really physically correct, but nevertheless, the results can look quite pleasing.

10 Cliff Hanger

Finally, you will discover just how slow ray tracing can be if not implemented properly. At the end of the `buildSpheres` function in `main.cpp` you will find a few lines of code that are commented out. *Uncomment these lines.* A triangle mesh that approximates a sphere is now present in the scene. The mesh consists of a mere few hundred triangles, yet the rendering has become tremendously slow. *Why is that?*

11 Conclusion

In this assignment you have implemented the core parts of a very simple Whitted ray tracer. Starting with a dull scene without shadows, reflections and refractions, all of these effects were added relatively easily. Supersampling was implemented to reduce aliasing and making the images smoother. Ray-Triangle intersection testing was implemented, enabling more complex scenes. Finally, a tessellated sphere was added, which heavily impacted rendering performance. This performance hit will be addressed in the next assignment.