Lab 1: Whitted Ray Tracing

EDAN35

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. You only need to make changes in the `traceRay()` and `main()` functions in `main.cpp`, the `intersect()` function in `swTriangle.cpp`, and the `getReflectedRay()` and `getRefractedRay()` functions in `swIntersection.cpp` to complete the lab.

1 Introduction

Start out by cloning swTracer at:

`https://github.com/LUGGPublic/Lab1-RayTracing`

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 `out.png` in the directory where the compiled executable resides. On Windows this could be

`Lab1-RayTracing/out/build/windows-default/Debug/out.png`

The image should be white for where eye rays hit a sphere, and black otherwise.

2 Diffuse Reflection

Currently, there is no lighting calculation in the scene. To be able to see the spheres that have been intersected, we can add diffuse shading. For now, we only consider *direct illumination* that originates from a point light source.

When a ray intersects a sphere, a `sw::Intersection` object is returned. This object contains useful information about the ray-sphere intersection event. Including the world position and normal at the hit point, as well as the material of the object.

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

3 Triangles

Supporting only spheres in a ray tracer quickly becomes very limiting. To represent more interesting shapes, we can add support for triangles, which allows us to render any triangulated geometry.

Just like `sw::Sphere` inherits from the abstract class `sw::Primitive` and implements an `intersect()` function, we have `sw::Triangle` inheriting from `sw::Primitive`, but currently missing its `intersect()` function.

Task 2 *Implement ray-triangle intersection.* Don't forget to uncomment the lines that add the triangles for the floor in the scene.

Use the ray-sphere intersection function as guidance for how the intersection object should be constructed once an intersection is found. Once ray-triangle intersection is working, uncomment the rest of the box.

4 Shadows

An important visual cue in images are shadows. If there is an object between the hit point and the light source, then there is no direct illumination, and the `directColor` 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 source.*

Task 4 *In `getShadowRay()` where the shadow ray is created, investigate what happens if a small epsilon is not used for* t_0.

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 exercise, a reflectance ray can be spawned at the point of intersection. Each material has a parameter, *reflectivity* or r. Combine the reflected colour with the direct illumination colour using linear interpolation based on the reflectivity.

Task 5 *Implement specular reflection, where each intersection where $r > 0$ spawns a new reflectance ray.* Don't forget to uncomment the lines that add two reflective spheres to the scene. 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 6 *Add a few more spheres to the scene and play around with different reflectivity 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. For more details on refraction, see the lecture notes.

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

$$L = (1 - r - t)directColor + rL_s + tL_t$$

where r is reflectivity, t is transparency, L_s is the light returned from the reflected ray, and L_t is the light returned from the refracted ray.

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 7 *Implement refraction in your ray tracer.* Don't forget to uncomment the lines that add two refractive spheres to the scene. Those spheres will have both a reflective and a refractive component.

Task 8 *Try different combinations of the r and t parameters.*

7 Supersampling

If you look closely at the pixels generated in the images so far, you will see that the edges appear jagged. This is because we only trace one ray through each pixel. To get a smoother result we need to increase the number of samples using a technique called supersampling.

Task 9 *Implement stratified grid sampling to produce anti-aliased images.* Note that the output colour should be the *average* of the colour 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 Ways forward

Your Whitted Ray Tracer is now done for this lab, but there are many ways you can expand upon it, either in your free time or as part of the course project. Here are some examples:

- Load more complex scenes by using a model loader like `tinyobjloader`, `https://github.com/tinyobjloader/tinyobjloader`.
- Accelerate the ray intersection testing by introducing a spatial data structure like a BVH
- Parallelize the ray tracing by using multiple CPU cores, using threading or OpenMP
- Parallelize the ray tracing by implementing it on the GPU