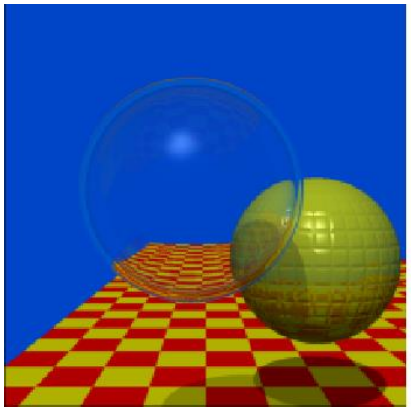
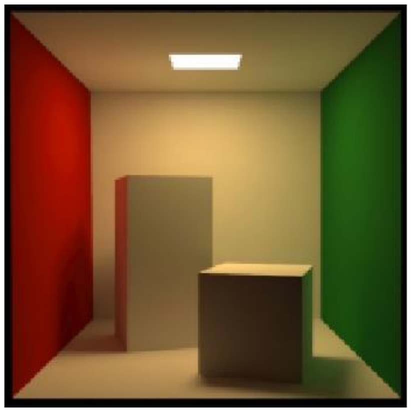
CPSC 453 Assignment 5
Ray Tracing (Bonus Assignment)[1]

Fall 2024, University of Calgary

1 Overview and Objectives

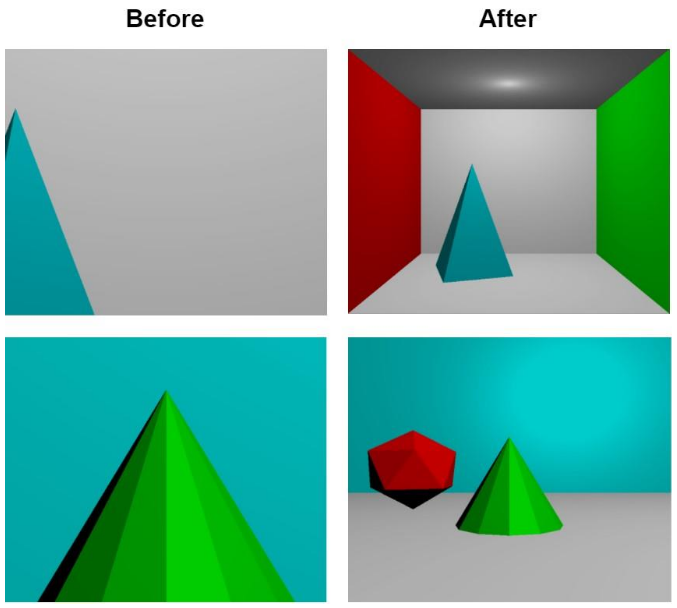
The purpose of this final, mini assignment in CPSC 453 is to provide you the opportunity to gain an introduction to the power of ray tracing. Ray tracing is a classic computer graphics technique used to synthesize perspective images of virtual three-dimensional scenes. The core algorithm, although a change from what you have seen so far with rasterized graphics, is also beautifully simple in relation to the level of realism it can produce.

In ray tracing, one shoots out rays from a camera and models their hits, reflections, etc. within the scene. Given the ray generation for an orthographic camera, you will basic ray generation to simulate the optics of a pinhole camera. You will implement ray-object intersection tests to learn how the algorithm decides which objects are visible from which pixels. Finally, you will implement reflections and shadows, two graphical effects that showcase ray tracing's power, as they are much more straightforward to implement in ray tracing than in rasterized graphics.

[1] Certain assignment specifications were taken from previous offerings of CPSC 453 by Dr. Sonny Chan and a previous assignment write-up by John Hall. Please notify Jeffrey.layton@ucalgary.ca of typos or mistakes.

2.1 Part I: Perspective Ray Generation (2 points)

In the boilerplate, rays are currently being constructed to simulate an orthographic camera. That is, all rays have the same directional component (i.e. are parallel), and they all originate from the centres of their respective pixels. You will modify this to implement ray generation for a pinhole camera, where, in contrast, no two rays will have the same direction, but all will originate from the same location, the "pinhole". You should set this at $(0,0,0)$. Rays should still be generated for every pixel of the image. You can see a before and after image for each of the two scenes below.



Note that in the above image, and all following ones, your rendered output does *NOT* need to match the pictured output *exactly*! Small choices such as the chosen field of view may affect the exact output. However, your output should look very similar to the provided images.

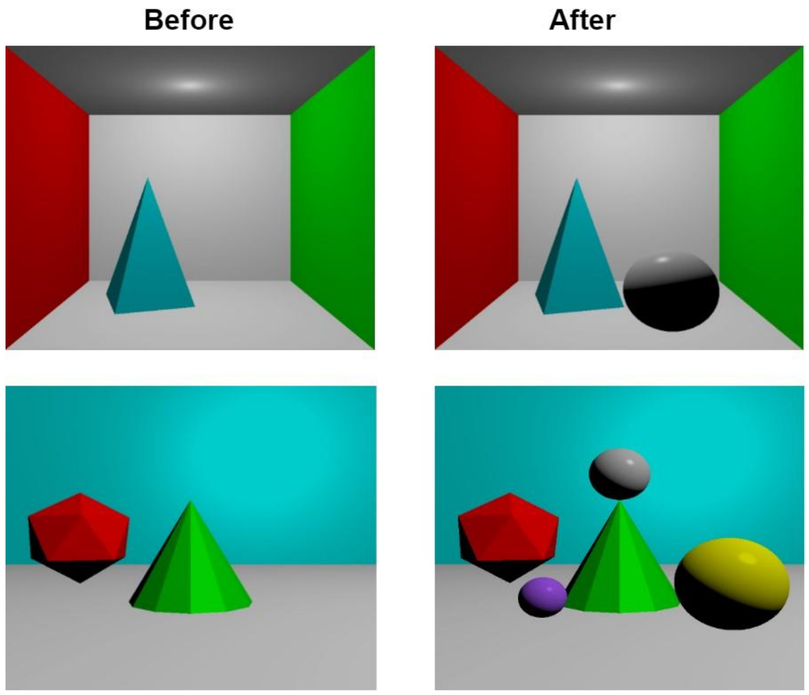
2.2 Part II: Ray-Object Intersection (4 points)

In the boilerplate, ray-triangle and ray-plane intersection have already been implemented, and they are responsible for the planes and triangles currently appearing on-screen. Now, you need to do the same for spheres and cylinders. In order to receive the full marks, you should implement:

1. Ray-Sphere intersection. Use the intersection algorithm to define at least one cylinder in each scene. Place your spheres so that they don't intersect with the other objects in the scene. (two points)
2. Ray-Cylinder intersection. Use the algorithm to define at least one cylinder in each scene. Rotate one of the cylinders so that it is slightly tilted. You do not need to add top and bottom caps to your cylinders. Your cylinders should have a limited height which should be accounted for in the algorithm. (two points)

Both the spheres and cylinders must be implemented as "perfect" objects, meaning

Example before and after images are provided below:

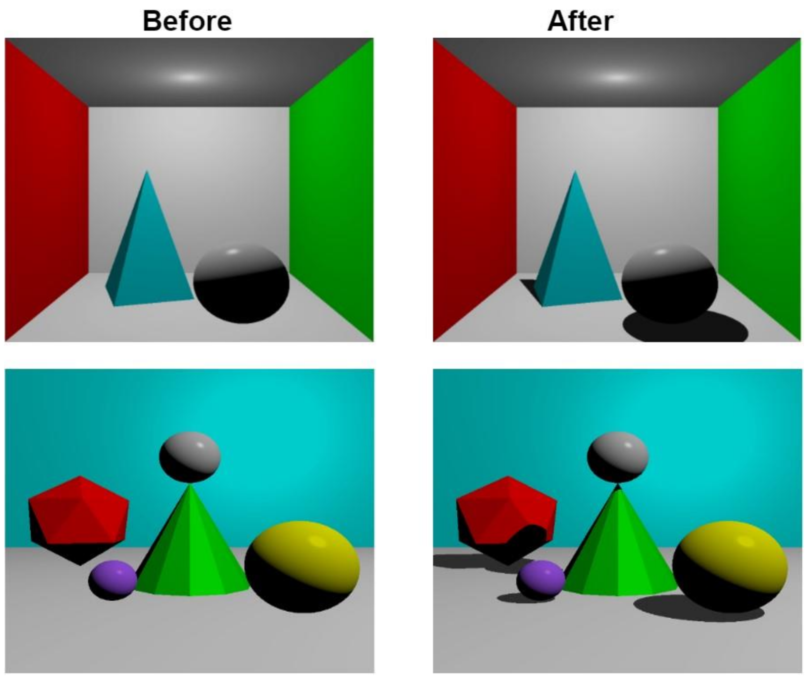


2.3 Part III: Shadows (2 points)

Next, you will implement one of the signature components of ray tracing: shadows! Shadows can be rendered by tracing a ray from your intersection/shading point to light sources in your scene. If the shadow ray is blocked by another object, then your surface will not receive the diffuse and specular contributions from that light.

When tracing shadow rays, you may need to be careful with self-occlusions, or rays intersecting the same object you are shading. Think carefully about when you may need to test for occlusion against the object you are shading, and when you may not.

As before, you can see a before and after image for what your scene should approximately look like once you are finished with implementing shadows.



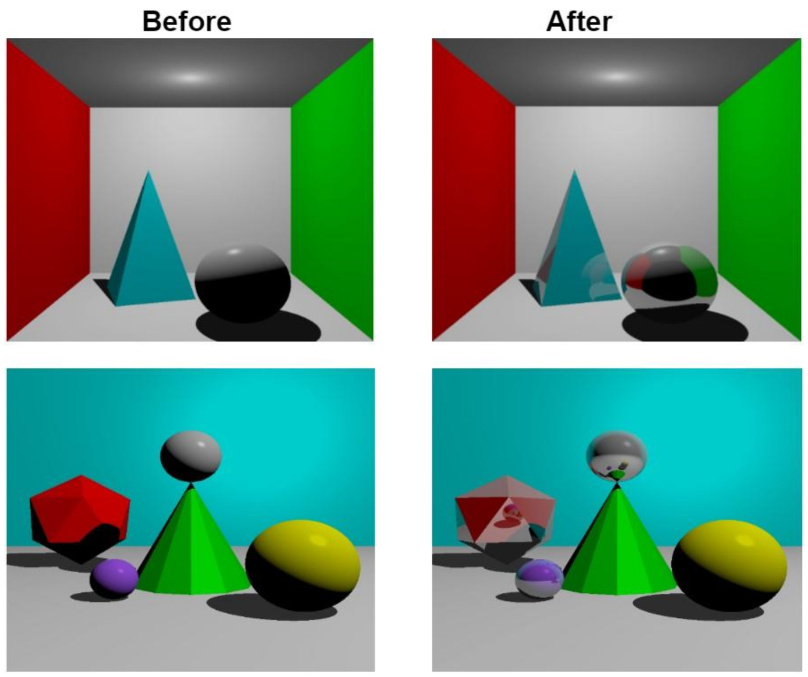
2.4 Part IV: Reflections and refractions (4 points)

Finally, you will implement one of the *other* signature components of ray tracing: reflections and refractions! Reflections are computed by reflecting the view or eye ray about the surface normal and recursively tracing the reflected ray within your program. Refractions are computed using Snell's law. Surfaces are often not 100% reflective, and you can achieve a partial reflection by mixing the colour obtained from the reflected ray with the shaded colour of the object itself. Because these are technically pure specular reflections, it is often most visually correct to keep the amount of ray-tracing reflection consistent with the object's specular shading parameters (e.g. modulating the reflected colour by the material's specular colour). Set reflective properties of the objects in the two provided test scenes so that your rendered images look similar to the ones in the below image. Note that no object is purely reflective, and that a lot of the objects, such as the blue pyramid, are only slightly reflective. Pay careful attention to these sample images in order to replicate something similar!

Note that some view rays may be reflected many times, or at worst, lead to an infinite recursion! Typically, a ray tracer will limit the number of "bounces" a ray makes before ending that trace. It depends on the scene, but a recursion depth limit of 10 is very reasonable.

In total you can receive four points from completion of this section:

1. 1.5 points: reflection implementation
2. 1.5 points: refraction implementation
3. 1 point: accurate reproduction of the given test images.



3 Submission

We encourage you to learn the course material by discussing concepts with your peers or studying other sources of information. However, all work you submit for this assignment must be your own, or explicitly provided to you for this assignment. Submitting source code you did not author yourself is plagiarism! If you wish to use other template or support code for this assignment, please obtain permission from the instructors first. Cite any sources of code you used to a large extent for inspiration, but did not copy, in completing this assignment.

Please upload your source file(s) to the appropriate drop box on the course Desire2Learn site. Include a "readme" text file that briefly explains the keyboard controls for operating your program, the platform and compiler (OS and version) you built your submission on, and specific instructions for compiling your program if needed. In general, the onus is on you to ensure that your submission runs on your TA's grading environment for your platform! It is recommended that you submit a test assignment to ensure it works on the environment used by your TA for grading. Your TAs are happy to work with you to ensure that your submissions run in their environment before the due date of the assignment. Broken submissions may be returned for repair and may not be accepted if the problem is severe. Ensure that you upload any supporting files (e.g. makefiles, project files, shaders, data files) needed to compile and run your program. Your program must also conform to the OpenGL 3.3+ Core Profile, meaning that you should not be using any functions deprecated in the OpenGL API, to receive credit for this part of the assignment. We highly recommend using the official OpenGL 4 reference pages as your definitive guide, located at: https://www. opengl.org/sdk/docs/man/.