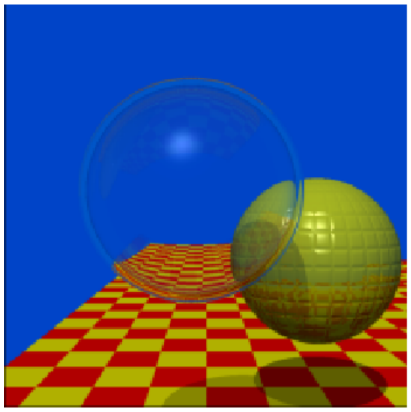
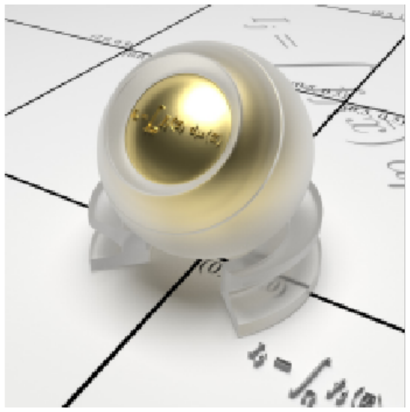
CPSC 453 Assignment 5
Ray Tracing (Mini Assignment)*

Fall 2021, 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 also implement ray-sphere intersection 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.

*Assignment specifications were taken from previous offerings of CPSC 453 by Dr. Sonny Chan and Dr. Faramarz Samavati. previous assignment write-up by John Hall. Please notify drlarsen@ucalgary.ca of typos or mistakes.

2 Due Date

December 9 at 11:59 PM

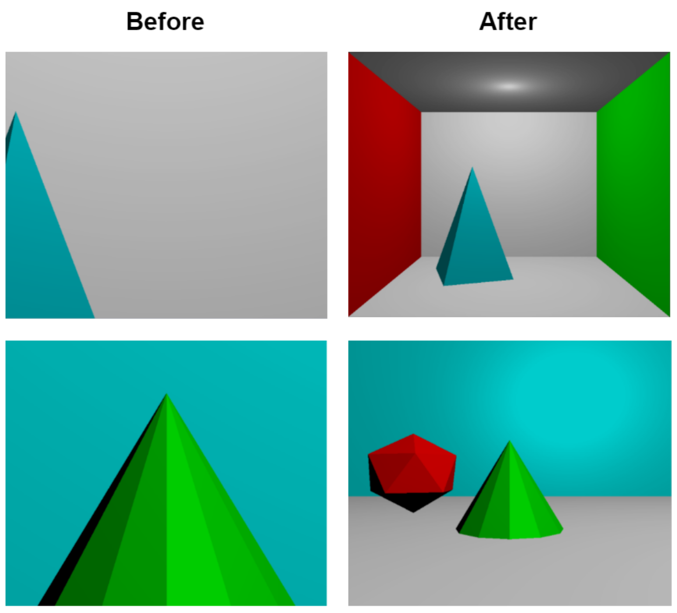
3 Programming Assignment

For this assignment, the boilerplate you will be given is the most extensive yet. A large portion of the ray tracing algorithm has already been implemented. Your job will be to implement the remaining four portions. Look for comments in the boilerplate code in order to determine where you will need to make edits. You should not have to make any changes to the boilerplate outside of the area that these comments appear. The README file included in the boilerplate will tell you where to look for these.

There are a total of four parts to this programming assignment, all of which add up to make a single application with two ray-traced scenes. This assignment is worth a total of 10 points, with the point distribution as indicated in each part, and an additional possibility of earning a maximum of 2 or 2.5 "bonus" points, by completing one of the two bonus designations.

3.1 Part I: Perspective Ray Generation (2.5 points)

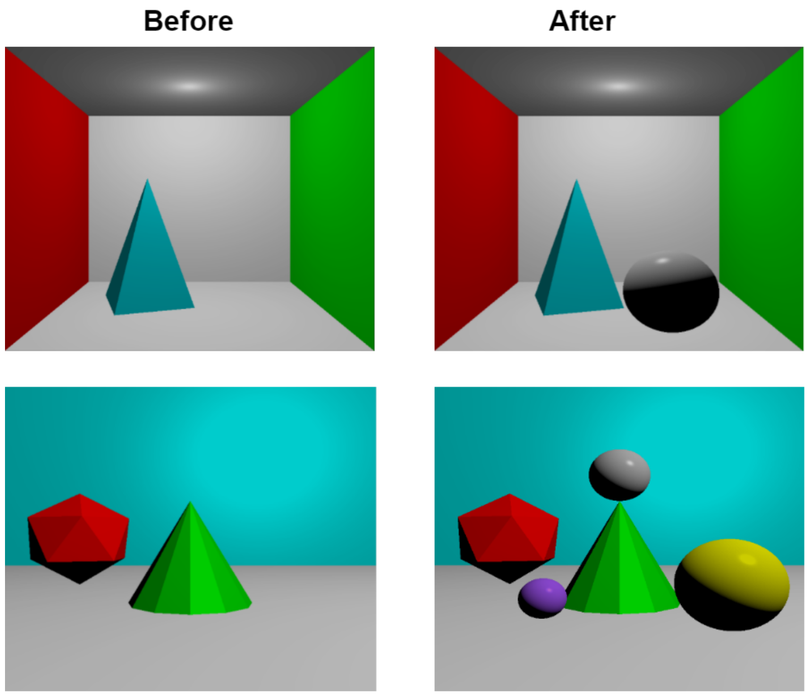
In the boilerplate, rays are currently being constructed to simulate an orthographic camera. That is, all rays have the same directional component (i.e. are parallel), and they all originate from the centres of their respective pixels. You will modify this to implement ray generation for a pinhole camera, 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 definitely look very similar at least!

3.2 Part II: Ray-Sphere Intersection (2.5 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. That is, you will need to implement ray-sphere intersection so that the algorithm knows when a ray has hit a sphere in the scene as opposed to an object behind it. Another before and after image is provided below.

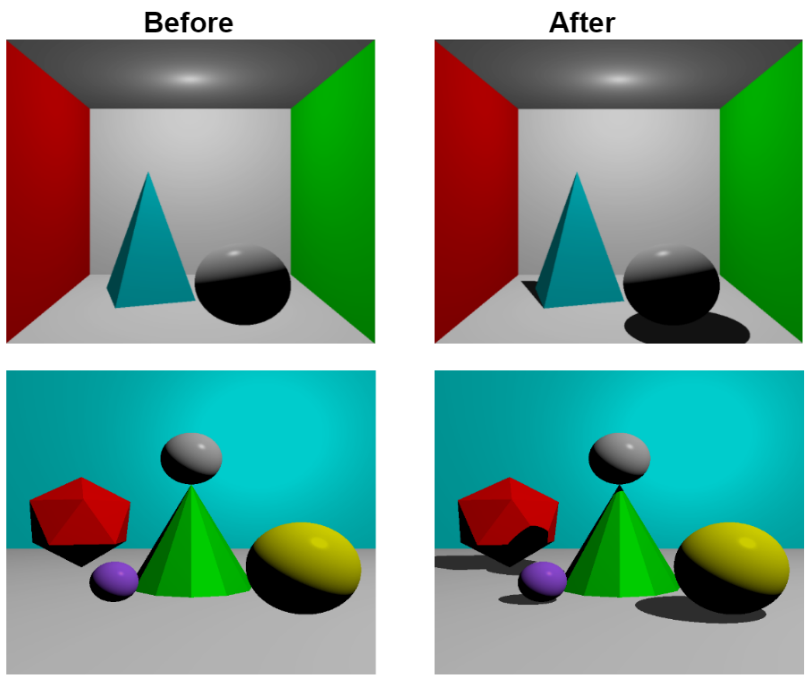


3.3 Part III: Shadows (2.5 points)

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.

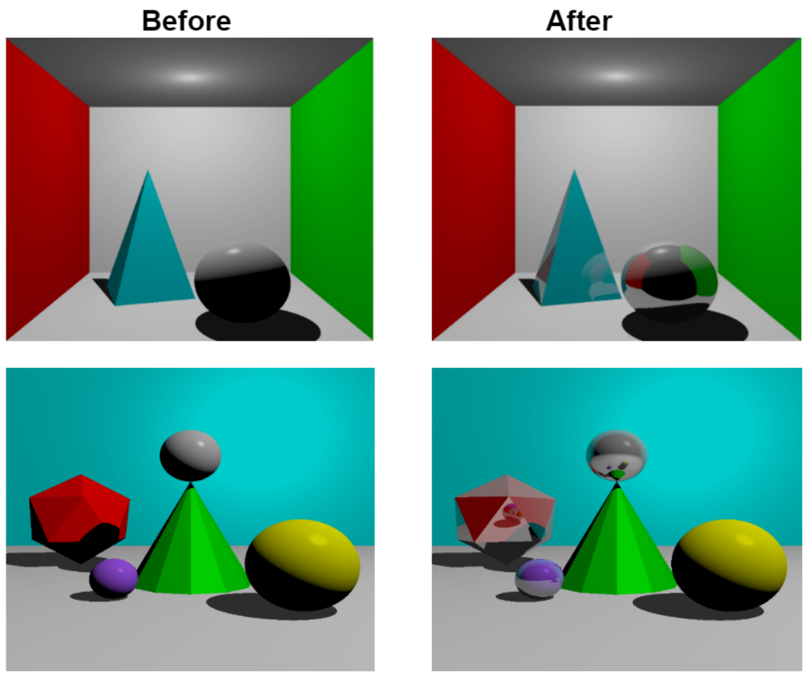
As before, you can see a before and after image for what your scene should approximately look like once you are finished with implementing shadows.



3.4 Part IV: Reflections (2.5 points)

Finally, you will implement one of the *other* signature components of ray tracing: reflections! Reflections are computed by reflecting the view or eye ray about the surface normal and recursively tracing the reflected ray within your program. 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.



3.5 Bonus Part I: Ray-Cylinder Intersection (2 points)

Note: *The 2 points awarded to this bonus cannot be combined with the 2.5 points from the other for a total of 4.5 bonus points. A maximum of 2 or 2.5 bonus points, from completing one of the bonuses in full, will be awarded.*

In a similar way to how you implemented ray-sphere intersection, also incorporate ray-cylinder intersection into your assignment. Note that you cannot simply form cylinders out of multiple triangles; you must introduce, and test intersections with, "perfect", smooth cylinders. You will also need to make sure that reflections and shading still work properly with your cylinders, i.e. that the correct normals are calculated.

Once you have completed this, demonstrate that it works by instantiating some cylinders into your scenes. To demonstrate that your solution is working perfectly, ensure that the following criteria are met:

- At least one of the cylinders must be reflective, and it must be positioned in such a way that this is visually clear, i.e. the other objects in the scene are showing up in this reflection. This is to verify that reflections are working properly.
- At least one of the cylinders must have *no* reflective component. This is to test that the shading still works correctly.
- At least one of the cylinders (possibly one of the above two) must be "tilted" with respect to the scene, i.e. it cannot be standing perfectly upright.

In total, the above means that you must have at least two cylinders across your scenes. While we don't deduct marks for the following, please try not to block too much of the other objects in the scene with your cylinders, so that the other parts of the assignment are still easy to validate without us having to modify the code to remove the cylinders.

3.6 Bonus Part II: Refraction (2.5 Points)

Note: *The 2.5 points awarded to this bonus cannot be combined with the 2 points from the other for a total of 4.5 bonus points. A maximum of 2 or 2.5 bonus points, from completing one of the bonuses in full, will be awarded.*

Implement refraction into your ray tracing algorithm, and demonstrate that it works by including a refractive sphere somewhere in front of other objects in your scenes. It should be visually apparent that the refraction is working correctly without having to dig through the code. Include the ability to toggle this sphere on and off, so that the original scene without the sphere is also visible.

4 Previous Year's Assignment (Reference Only)

Upon receiving feedback from some students, we have decided to share with you what a version of this assignment might look to in a normal year, had deadlines not been shifted and more time was available. You do **NOT** need to read this if you are not interested. It is only for students who are curious and who want to challenge themselves for the educational value, particularly if they plan to do more computer graphics in the future.

A previous year's version of this assignment can be found at:
https://pages.cpsc.ucalgary.ca/ sonny.chan/cpsc453/resources/handouts/CPSC453-Assignment4.pdf
Note that it is very similar to this current assignment, except that far less boilerplate is given, the bonuses are different, and you are responsible for creating your own additional scene. Learning how to create a ray tracer "from scratch" as this previous year's assignment requires is something that you can attempt on your own time if you are interested in this.

5 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 3.2+ 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/`.