Computer Graphics Assignment 1 - Ray Tracer

Andrew Kohler

Contents

Credits

Before beginning this report, I feel it is important to provide adequate attribution openly and plainly. The following sources were instrumental to helping me understand and implement various features, and have my utmost thanks:

- Marschner and Shirley’s *Fundamentals of Computer Graphics* (course textbook)
    - The textbook’s described ray tracing architecture was the basis of my project’s class structure, and I follow many of the same general ideas that they present for implementation of objects, materials, lights, shadows, and glazed surfaces.
- *Ray Tracing* and *Shading* slide decks (course materials)
    - The class slides were similarly instrumental in several parts of my implementation, helping me to understand many of the mathematical formulas necessary for this application.
- Scratchapixel 3.0’s guides on rendering triangles
    - When repeated attempts to implement various mathematical representations of triangle collisions failed, this series of guides explained the principles of the matter in a way I could clearly understand, and were the key to my third version of the implementation, which functions correctly; they were also highly informative as to how to implement backface culling. Although code snippets are present within the site, I refrained from using them directly, at most borrowing some naming conventions.
    - https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution.html
    - https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/single-vs-double-sided-triangle-backface-culling.html
- stb_image library
    - To export the sequence of images that form my short movie, I used the stb image library, specifically the stb_image_write command. The GitHub is linked, as well as a site which helped me understand how to set it up.

    - https://github.com/nothings/stb
    - https://solarianprogrammer.com/2019/06/10/c-programming-reading-writing-images-stb_image-libraries/
- Blender
    - I used the popular 3D modeling application Blender's video sequencer to make my images into a video clip.
- glm vector library
    - The glm vector library provided me with a helpful suite of vector-related types and functionality, greatly simplifying the process of performing the many necessary vector operations in a raytracer. A link to the GitHub page is provided, along with the video that helped me set it up.
    - https://github.com/g-truc/glm
    - https://www.youtube.com/watch?v=kuz8QC5-beQ
- Sample Report
    - The sample report provided in the documentation for this assignment proved very useful in establishing a general structure for my own report, and although I ended up partially discarding its implementation of constant multipliers affecting different material properties, it remains the place where the idea originated in my code.
- Julian Stennett
    - A fellow student whose impact on my project cannot be underestimated; he provided a clean VSCode project setup with GLFW and GLEW correctly configured, something I was unable to achieve in nearly 8 hours of trying. I would like to emphasize that there was no code in this project, only the base simpleTexture.cpp; regardless, if my folder structure appears similar to Julian's or other students', the reason is that they were a great help to many of us who struggled with setup.

Rays and Cameras

The foundation of my code is the Ray class. It contains two three-element vectors representing the origin point and direction of the ray respectively ("**origin**" and "**direction**" in code). The class contains a constructor, getter methods for each variable, and an evaluateRay() method, which when given a float t, returns a three-element vector representing the point **p**, found from solving the ray equation **p** = **o** + t**d**. Without a Ray, there would be nothing to cast.

Rays are generated by Cameras. The main function of a Camera object is to generate a ray for each point in the image plane based off of the current pixel coordinates of the double for loop used to iterate over each pixel of the image being generated. The Camera parent class contains variables representing the worldspace position of the Camera (**worldPos**, , a three-element vector commonly called **e** in class materials), the boundaries of the image plane (represented by floats l, r, b, and t for left, right, bottom, and top), and 3 three-element vectors (from here on, referred to as Vector3's) **u, v, and w**, representing the coordinate plane of the camera space. **w** is a vector facing the opposite direction that the camera faces, and **u** and **v** represent the rightward and upward directions of the camera space.Two other float variables, imgWidth and imgHeight, are present but redundant, holdovers from an old attempt at a feature implementation.

All cameras contain a constructor which takes **e** (worldPos),**d** (a vector representation of which way the camera is facing along the worldspace axes), **upDir** (a vector representation of which direction the camera should consider 'up'), the width and height of the camera as floats, and again the redundant imgWidth and imgHeight. All cameras also contain getter and setter methods for the position and direction of the Camera; the direction setter method recalculates **u, v,** and **w** to account for the worldspace-facing direction changing the orientation of the camera space.

Most importantly, Camera contains the virtual generateRay() method. generateRay() takes two floats, iPixel and jPixel, as inputs - these are the coordinates of the current pixel position being analyzed in the overall image in the double for loop which iterates over each pixel. It returns a Ray object representative of the corresponding ray which should be cast from the camera into the world. This method is overridden and implemented by the two child classes of Camera, OrthoCamera and PerspectiveCamera.

OrthoCamera

OrthoCamera is my implementation of an orthographic camera. It generates rays which all have the same direction, but different origins. The direction of all rays is **-w**; the origin is found by the equation

Origin = worldPos + uScalar * **u** + **vScalar** * v

where uScalar and vScalar are floats representing the necessary transformations for the ray to originate within the camera space. A ray with this origin and direction is what generateRay() returns.

PerspectiveCamera

PerspectiveCamera is my implementation of a perspective camera. It has an additional variable, a float focalLength, representing the distance between **worldPos** and the image plane. The origin of all rays is **worldPos** when using PerspectiveCamera, but their directions are determined by

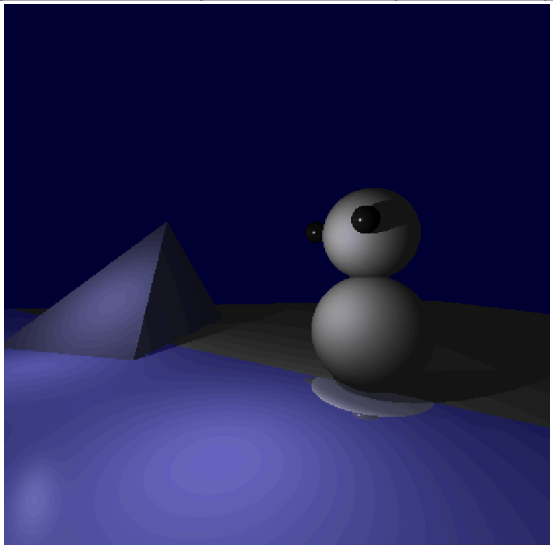
Direction = normalize(-focalLength * **w** + uScalar * **u** + vScalar * **v**)

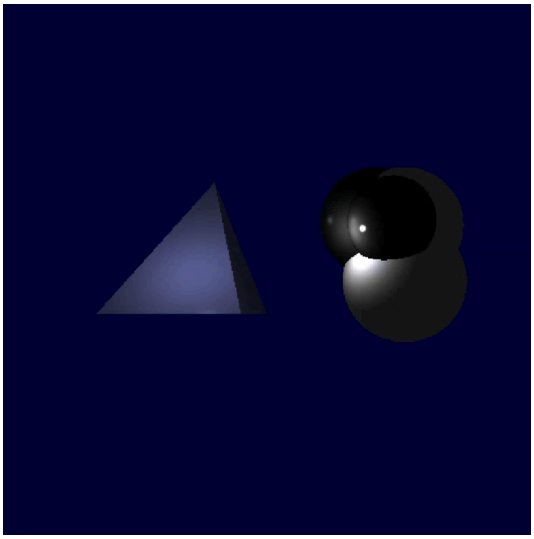
where uScalar and vScalar are the same as above. A ray with this origin and direction is what generateRay() returns.

There is one of each type of Camera in my scene; PerspectiveCamera is enabled by default, but a user can toggle to an orthogonal view and back by tapping the "p" key (an input inspired by the sample report; "p" for perspective just makes sense). The constructor data for each camera is provided below, as well as images of my scene from each view (perspective on the left, orthographic on the right).

	worldPos	Direction	Up vector	Width	Height	Focal length
Orthographic	(3.0, -3.0,	(-1.0, 0.0,	(0,1,0)	64	64	N/A

	-6.0)	.2)				
Perspective	(2.5, 1.0, -1.0)	(-1.0, 0.0, .2)	(0,1,0)	512	512	.5





Objects and the Scene

To explain the setup of my scene, I will first go through each of the different types of Object that it is composed of.

Object (Parent)

- The Object class is the basis of all physical objects. It contains a default constructor, a constructor for making an object with a unique material (to be covered later), and the most important method, the virtual objHit(). This method takes in a Ray and a lower and upper bound for where to search along the ray, t_0 and t_1, for a collision point.
    - It returns an instance of a HitReport, a special class that functions as a container for collision data to be passed and used as needed. A HitReport contains t, the distance from the origin to the point on the object collided with along the ray, n, the surface normal vector of the object at the point collided with, and m, the material of the object. If a collision is not made along a Ray, t will be set to infinity; all checks which use HitReport use this condition to confirm the contents of the report.
    - Each of the following classes is a child of Object, and overrides objHit() to perform calculations specific to its type of ray-object intersection.

Sphere

- The Sphere class contains a float value “radius” and a Vector3 **center** denoting the radius of the sphere and origin point respectively. It contains a constructor which facilitates its initialization with these parameters.

Triangle

- The Triangle class is defined by 3 points which make up the Triangle. Triangles have the option to have backface culling turned on or off, preventing rendering if their normal is the same direction as the ray.

Tetrahedron

- The Tetrahedron class contains 4 sets of 3 dimensional coordinates, with **baseCoord1-3** representing the coordinates of the base of the shape, and **topCoord** representing the coordinate of the 'point' of the tetrahedron, typically its top. It also contains 4 triangles; its constructor takes 4 sets of coordinates, and then creates the 4 triangles, which are checked one by one for ray collisions.

Scene

- The Scene class uses objHit to iterate over a provided list of Objects, using a given ray to check if it intersects any of them, and if so, how to apply lighting to them.

Materials and Lighting

Each Object stores an ObjMaterial within it. This material contains a variety of variables determining the various material properties of the object, which are listed and explained below.

- diffuseColor (Vector3)
    - The color of an object (RGB 0-255) which is multiplied by the irradiance of the light in a diffuse light calculation.
- specularColor (Vector3)
    - The intensity (a tuning value, 0-?) of the specular white dot effect applied to a surface under a light.
- ambientColor (Vector3)
    - The color of an object (RGB 0-255) which is multiplied by the intensity (a tuning value, 0-?) in an ambient light calculation.
- ambientCoeff (float)
    - An arbitrary value which exists as a control knob for strengthening or weakening an ambient light source.
- reflectiveness (Vector3)
    - The intensity (a tuning value, 0-?) of how much color from reflections is applied to a given surface.
- glazed (bool)
    - Whether or not a surface should be glazed
    - The 'ice' in the final render and scene is a glazed surface; you can see it reflect the snowman. For additional examples of glazed surfaces, see Appendix.
- phongExp (int)
    - The Phong exponent applied for specular calculations.

Additional float values diffuseCoeff and specularCoeff exist from a previous attempt at a different kind of implementation, and are currently redundant.

I use 2 types of lights in my scene, both derived from a parent Light class: PointLight and AmbientLight.

PointLight

- PointLight carries out calculations for specular and diffuse lighting of a given object; its illuminateRay method returns the color value generated by applying diffuse and specular properties to the given object based on its material.,

AmbientLight

- AmbientLight returns a color value equal to ambientColor * ambientCoeff * intensity (a property of the light).

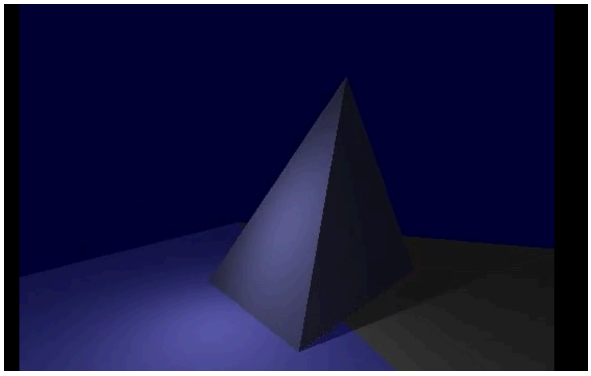
Additional Feature - Shadows

Shadows can be seen throughout the perspective render predominately used in this report; See **Shadows1** in the Appendix for an additional example of my shadows.

"Snowbound Land" - Short Movie

I created a 2-second clip running at 24 fps which consists of my perspective camera moving through my scene, rotating around to get a full view of it, and a slight comedy element of the snowman "turning" to face the camera. In reality, this effect is achieved by moving the two spheres representing its eyes while the back of the camera is turned.

Some images from the movie are included below. The 24 fps movie MP4 is included in my submission, as well as a 12 fps version which is slower and easier to process / observe detail within.



Appendix

- A variety of progress screenshots can be found in my final submission. They are described here, denoted by their file names.
- **FirstSpec** is an image I took when I first made specular surfaces work. My equation was slightly incorrect, but still clearly effective.
- **Persp** is an image I took when I completed my PerspectiveCamera. While its center was, at the time, the bottom left of the screen, it clearly illustrates the distortion induced by the viewpoint.
- **PlaneTrouble** was taken when I was having difficulty correctly orienting the normals of triangles. **SuccessfulTri** was taken after my rework of my Triangle class to ultimately resolve this issue.
    - **Tetra** was taken at an earlier phase of the project, when I had just completed the initial Tetrahedron.

- **Shadows1** is an image from when I got shadows working showcasing different shadows being cast from different lights.
- **Shine** is an image taken from when I got glazed surfaces working, showing off multiple reflections in a sphere.