Deanna Hambleton

Simulating a Disco Light

Motivation/Problem

For my final project, I implemented a simple simulation of a disco light. A disco light is a device that is composed of a solid cylinder that contains lights, on top of which is a plastic crystal that rotates, and through which the lights shine through. The refraction of these lights, along with the rotating of the crystal, causes a mesmerizing array of shapes and colors on the walls/ceiling, which makes them great for parties. It was actually at a party that I got the idea to simulate one; I saw one of them in action, and after watching it for a while, I was reminded of the ray tracing assignment I had done for this class, after which I decided it would be interesting to implement a simulation of this device. On an overall scope, though, the interaction of light with objects is extremely important in computer graphics, since it is the reason why we are able to see anything in the physical world. Indeed, it is so ingrained in our everyday experiences that we never think about it, yet it is something that comes to the forefront of our minds when we think about rendering anything. Thus, in order to accurately generate 3D scenes, knowing how light and objects interact is critical, and this project allowed me to experiment with one such interaction: the shining of a light through a rotating crystal. Overall, the main problem that I had to tackle was implementing a ray tracer that would account for rotations, and using that tracer to render the scene.

Figure 1: Disco light

Approach/Implementation

The general approach that I took is as follows: first, I extended my ray tracing code from assignment 5 to allow for continuous transformations of objects/light sources. Then, I rendered the scene. Here is where my project changed from my original intention; initially, I was going to render the scene with OpenGL, creating a program that would allow 3D viewing of the scene, but since my ray tracing code computes the colors for each pixel and returns an array of them, the only way to use OpenGL with this output was to individually draw each pixel. I initially attempted to do this, but it was extremely time-consuming to draw everything, so I decided to utilize saved images. Basically, every time the object is transformed, it is rendered, and then the resulting data is saved to a separate PNG file, creating a set of PNG files in the end that shows each of the

transforms. Then, I used the openFrameworks toolkit to create a program that loaded these images to the program in such a way that it animated them, creating the illusion of an animation.

Concerning the extensions to my ray tracer, I implemented three new classes: ContinuousTransform, Spotlight, and RotatingSpotlight. ContinuousTransform is a subclass of the existing Transform object that stores an extra matrix (the one describing the transformation to be applied continuously). It also takes advantage of a new Object3D method called updateAngle() (I defined it within Object3D (and left it blank) because I wanted to be able to call updateAngle() on groups, which would then call the appropriate updateAngle() method for each of its constituents). In short, when updateAngle() is called, it takes the ContinuousTransform's current rotation matrix, multiplies it by the continuous rotation matrix, and then stores the result as the new current rotation matrix.

After that, I implemented the Spotlight class. This is a Light subclass that contains an origin, a direction, and two angles that determine the light's coverage (see image below on what each one is for — the "outer cone" section defines the area where the light starts to fall off, creating a fade effect). After that, I created a Spotlight subclass called RotatingSpotlight that behaves very much like a ContinuousTransform, in that it contains two rotation matrices and uses them to compute the spotlight's current rotation matrix in an updateAngle() method.

Then, I implemented the program itself. I had decided to implement some user functionality: if the 'r' key is pressed, the rotation starts or stops; if the 'c' key is pressed, the light color changes; if the 's' key is pressed, the rotation's speed changes; and if the 'e' key is pressed, the rotation's direction changes. Using the provided keyPressed() method, I created an event listener that would respond and make changes depending on the key pressed.

The general execution of an openFrameworks program is as follows:

```
initialize window
while true:
        update()
        draw()
```

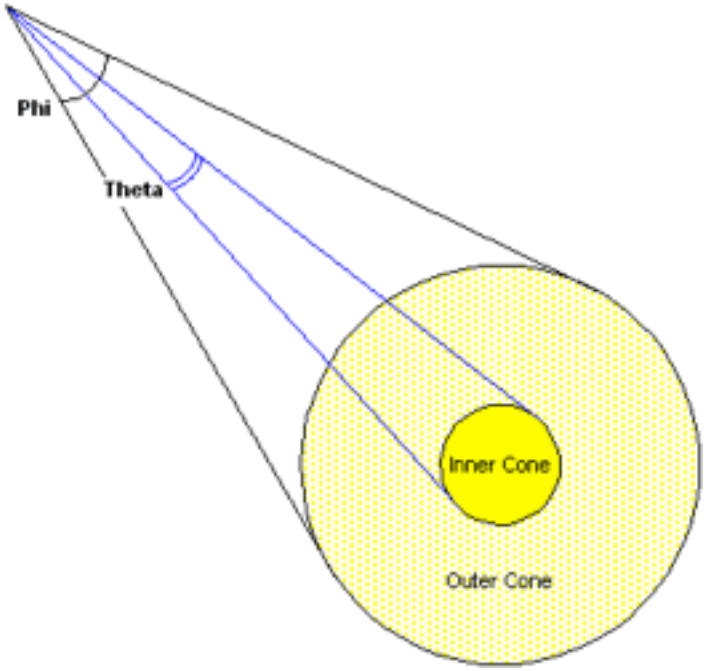

Figure 2: The angles in a spotlight

In my program, after initializing the window, I also initialized two variables: counter (which is used to determine the time step in the rotation, and to load the appropriate file) and colorCounter (which is used to determine which set of images to

use, from a set of two with different colors). Then, in update(), the counters are changed, as well as other factors (speed, direction), depending on events. After that, in draw(), I loaded the appropriate image file into an openFrameworks Image object, and used the object's draw function to draw it to the program. With all of this executed under a continuous loop, the effect is an animation made from images, with basic controls for manipulating it.

Results/Problems Encountered

Overall, the program worked as I had intended. The images below show a set of transformations, in both color schemes that I used.

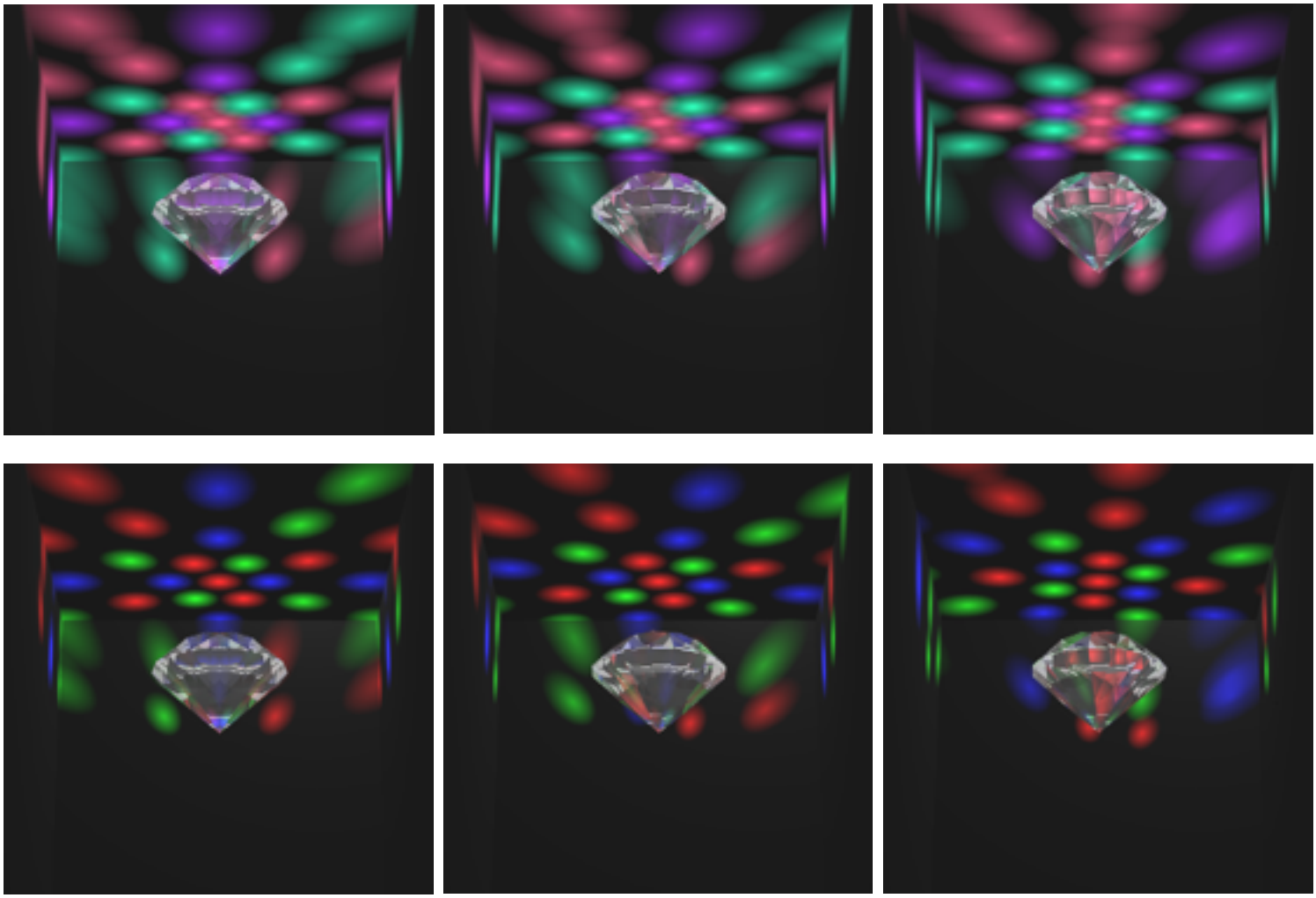

Figure 3: Some resulting images

There is also a video on Youtube in which I show how the animation works and how the user input can change it (link: https://www.youtube.com/watch?v=yrVrvUWZA-Q&feature=youtu.be). The program that performed the animation itself also worked as intended — animations were smooth and appropriate inputs from the user changed the scene as desired. However, one thing I forgot to account for was implementing the

refracting of light beams, so the results of the simulation were the same in the cases where the diamond was there and when it was not.

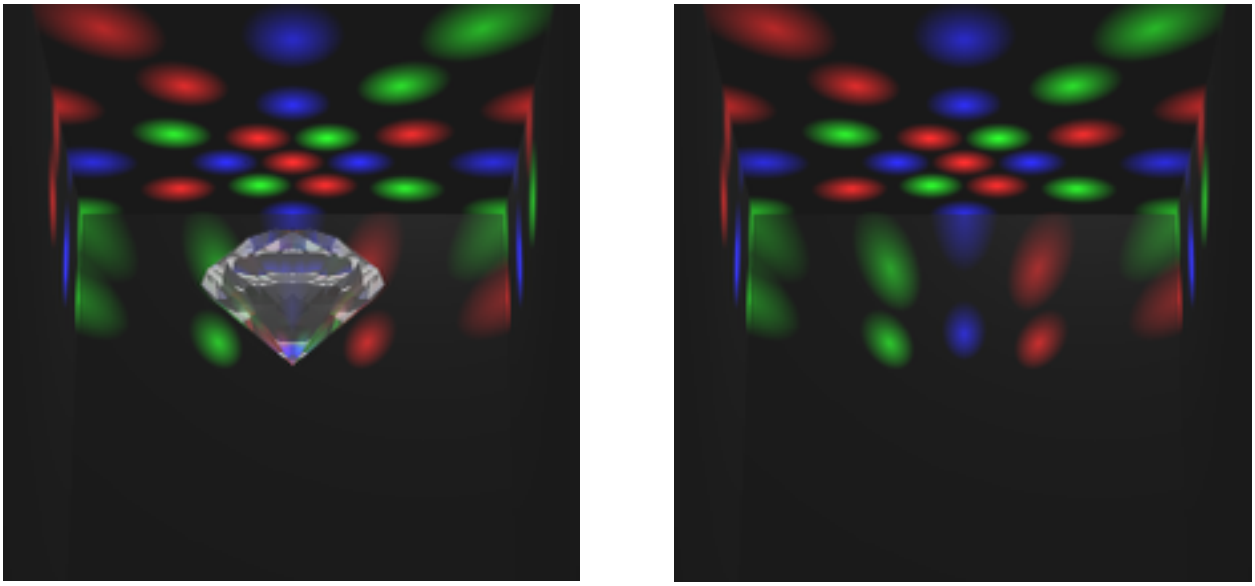

Figure 4: The lights with and without a crystal

While looking for a solution for this problem, I came across the concept of photon mapping (initially developed by Henrik Wann Jensen), the basic idea of which is to compute a set of "photons" from the light sources, and to use those photons to determine the color of points in the scene. Due to time constraints, I decided to implement a brute-force version of it; I also defined a photon to be a light source, with an origin, a direction, and angles. Basically,

```
for each light source:
        create a photon containing the light source's origin, direction, and
                angles, add it to an array (a photon "map")
        generate ray from light source
        if the ray intersects anything:
                trace ray
```

Concerning the ray tracing algorithm used in this scenario,

```
check if refraction occurs
if so:
        create photon containing refracted ray's origin and direction, and
                the angles passed into the function
        add it to photon map
        trace ray
```

The basic idea of this is that each light source is ray-traced, and at each intersection point, a photon (a cone of light) is created and stored in a photon map. Then, after this photon map is created, my rendering method would use it when ray tracing. When getting a point's illumination from a light source, it would check if that point falls within any of the cones, add up the contribution from each one, and return that. This value is then used to compute the pixel color.

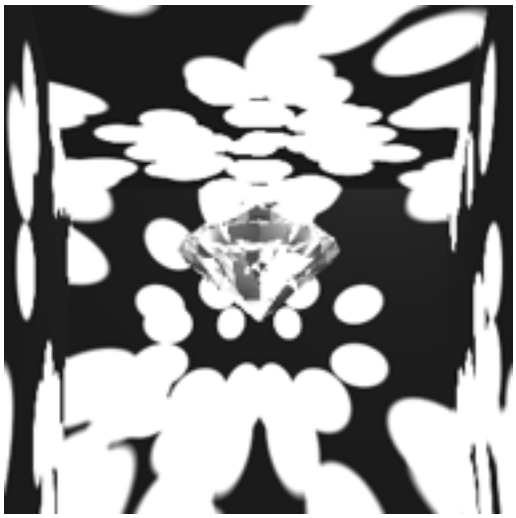

Figure 5: Attempt at ray tracing with photon mapping

Due to it being brute-force, it was extremely slow (~5 - 10 minutes to make one 300 x 300 image). Not only that, but a (most likely) incorrect implementation caused the image produced (see image to the left) to be inaccurate, at least in terms of color; I did manage to achieve the effect of refracting light, though, which was an improvement. However, due to time constraints, I wasn't able to fix what was wrong, so there's no proper implementation of light refraction as of now.

Conclusion/Future Improvements

Overall, the main behavior that I was trying to implement (the refraction of light through a rotating object) didn't go as planned. In the future, however, I can add some improvements to better implement this. For one, I can create a better way of utilizing photons and photon maps. Instead of storing cones, I can just find out all of the points that the light sources intersect with, compute the point's overall color from the interacting light sources, and store this in a photon map (which, to make things way more efficient, will be a k-d tree). Then,

```
for each light source:
        generate ray
        check if there's an intersection
        if so:
                create Photon that stores light's color and the intersection point OR
                        update the photon already at that point if there is one
                trace ray (create more photons at later intersection points in the
                        same way)
```

Then, when ray tracing is being performed on the scene, at each intersection point, check for the point's nearest Photon neighbors, and interpolate the point's color based on those.

Another improvement I'd implement would be the use of OpenGL in this project. Since this current code isn't very compatible with OpenGL, I can use shaders to create a new way of implementing ray tracers, one that OpenGL can easily make use of. Being able to animate the crystal in 3D (initially one of my main goals of this project) is the

driving force behind this improvement, as well as allowing for a user to be able to interact with the scene in a variety of new ways. Color, rotation direction, and rotation speed would still be changeable, but there would also be new ways of manipulating the scene — the camera angle and position can be changed, as well as the position of the actual objects in the scene (made possible with mouse clicks). This would allow greater exploration of how the light interacts with the objects in a scene, as well as much more user flexibility in how they interact with the scene themselves. Overall, once these two improvements are implemented successfully, then my original main goal for this project will be complete, and my simulation of a disco light will be much more authentic.