Raytracing Warmup Report

Alexander Kim

Implementation

Libraries

I made use of three libraries to build this basic ray tracer. The first two, GLFW and GLEW, were required by the project. I also chose to utilize GLM to simplify vector math.

Sphere Struct

The sphere simply consists of a 3D coordinate for the center, a radius, and an RGB value for color.

Camera Struct

The camera has the following members:

- 3D vectors to specify origin, look at direction, up direction, right direction, and back direction
- Values for the distance to the near and far plane
- A vector indicating the 3D location of the pixel in the top left of the screen
- Two values indicating the pixel width and height in world space

The camera is initialized with an origin, view direction, up direction, x resolution, y resolution, near plane distance, far plane distance, and in world view width and height. I chose to let the user specify the size of an in world camera so changing the resolution doesn't impact what the camera sees. This was done by setting the pixel width and height variables equal to the view width and height divided by the number of pixels in the x and y resolutions respectively. The u and w vectors were calculated based on the instructions given in class. Lastly, I included a helper function that takes x and y pixel values in screen space coordinates and returns the in world ray origin for that pixel. This is super useful later on.

Ray Sphere Intersection

The ray sphere intersection function takes

- A 3D vector representing the origin of the ray
- A 3D vector representing the ray direction
- A sphere object

We will call the ray given as input the pixel ray. Using the geometric perspective discussed in class, I calculated whether or not the distance from a ray perpendicular to the pixel ray to the origin of the sphere was larger than the radius. I also checked if the ray from the pixel to the sphere was pointing the opposite direction of the camera. Both of these conditions disqualify the

ray from ever intersecting the sphere. If it was possible for the ray to intersect the sphere, I calculated the discriminant using the equation discussed in class to get a value for t. I took the smaller of the t values to ensure the function returned the closest point of intersection. The function returned -1 if there was no intersection.

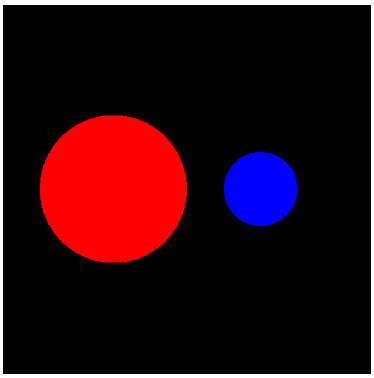
Pixel Color Assignment

I used the given for loops to check every pixel in the screen resolution. For each pixel, I used the helper function discussed earlier to get its world space coordinates. The direction was equal to the camera direction since I implemented an orthogonal camera. Then, I checked every sphere in the scene to see if it intersected with the given pixel ray. I took the smallest result to ensure the nearest intersection, and used the corresponding sphere's color as that pixel's color. If no intersection occurred, I set the pixel color to black.

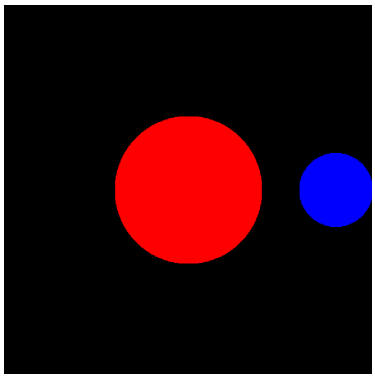
Results

Spheres with Changing Camera Position

Camera facing along X direction with Z direction as up vector
Spheres: center {5, 2, 0} radius 2 and center {5, -2, 0} radius 1



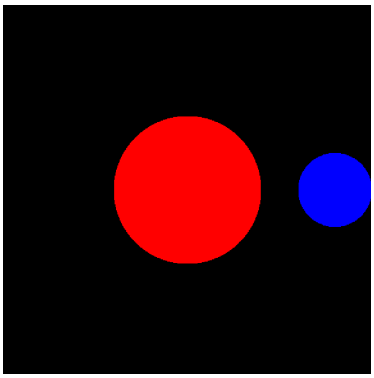
Camera at {0, 0, 0}



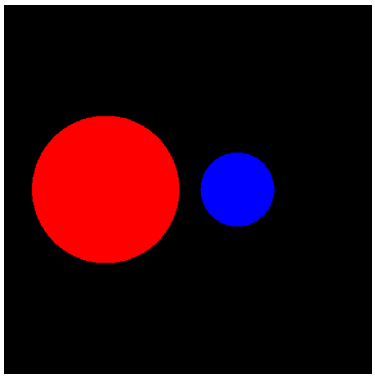
Camera at {0, 2, 0}

Spheres with Changing Camera Angle

Camera at {0, 2, 0} with Z direction as up vector
Spheres: center {5, 2, 0} radius 2 and center {5, -2, 0} radius 1



Camera Look Vector {0, 0, 0}



Camera Look Vector {0, -0.5, 0}

Occlusion Test

Camera at {0, 0, 0} with Z direction as up vector
Spheres: center {5, 2, 0} radius 3, center {4, 1, 0} radius 2, and center {3, 0, 0} radius 1
Illustrations made in Desmos to help with perspective