Ray Tracer Assignment

Sagan Potenza

Objects and Scene

Building

My project uses no external libraries except for GLFW and GLEW, meaning it should be able to be compiled by the command given in the Makefile, though this was not tested due to my setup being on Windows compiling the binaries myself. As long as GLFW and GLEW are accessible via -glfw and -glew, it should work, but feel free to contact me if it's not building for some reason.

The RayTracer Class

Most functionality is handled by the *RayTracer* class, which defines classes that make up the geometry of the scene.

The *ColorPack* class of *RayTracer* stores the color information of a material, that being the ambient constant, diffuse constant, specular constant, phong exponent, and whether the object is reflective or not.

The *Object* abstract class of *RayTracer* is a generic object defining that all inherited objects will store a *ColorPack* and contain the pure virtual *intersection* and *getNormal* functions that will be used for ray tracing. Four classes implement the *Object* class, that being *Sphere*, *Plane*, *Triangle*, and *LightObj*. Each class stores the location information of the respective object and overrides *intersection* and *getNormal*. *LightObj* is a special case, functioning identically to the Sphere, except that it does not cast shadows and always has full ambient lighting. It is meant to be placed at the same place as any light sources and can be toggled on and off with a key bind to help visualize those light sources.

The *Light* class stores the location and intensity of the light, which will be used when rendering the scene. Intensity is a value from 0 to 1 indicating how bright the light source is. Usually, low intensities are used to avoid overexposure.

The *RayTracer* class also contains member variables defining information about the scene such as the ambient intensity, whether the camera is orthogonal or perspective, the image width/height, and projection distance of the camera (only used if camera is perspective). Two vectors store a list of every object in the scene and every light in the scene, respectively.

The *RayTracer* class contains the *produceImage* function which initiates the Ray Tracing Algorithm when provided with a camera position, LookAt vector, and Up vector. This function uses a separate findColor function to get the color of a ray and direction, allowing for recursion for the glazed surfaces.

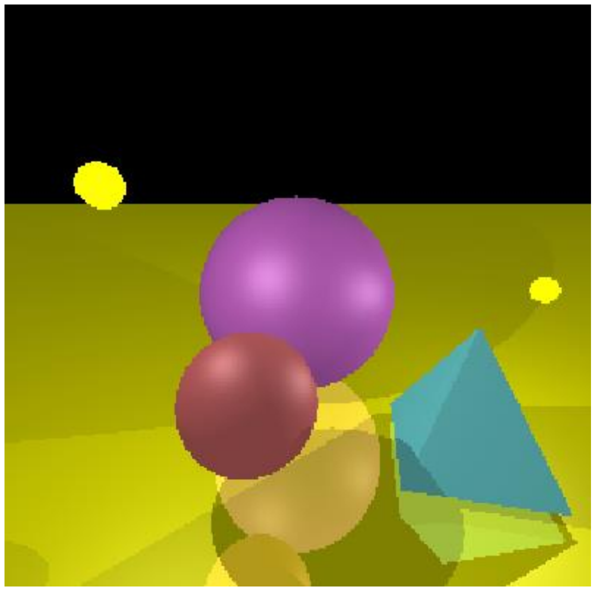
Key Binds

Several key binds can only be accessed in low-res mode, a mode which switches to a 64x64 resolution so that frames are rendered in real time. For more information, see the Bonus Advanced Feature Section.

Key	*Function*
Escape	Exits the Program.
O	Switches to Orthographic Mode.
P	Switches to Perspective Mode.
H	Switches Resolution to High Res Mode.
L	Switches Resolution to Low Res Mode.
W/S	Moves the camera forwards and back (only in low-res mode).
A/D	Moves the camera left and right (only in low-res mode).
Space/Left Ctrl	Moves the camera forwards and back (only in low-res mode).
Left/Right Arrow	Turns the camera left and right (only in low-res mode).
Up/Down Arrow	Turns the camera up and down (only in low-res mode).
Mouse	Mouse movement will turn the camera just like the arrow keys (only in low-res mode).
U	Turns off light visualization.
I	Turns on light visualization.
R	Starts recording, storing every low res rendered frame's lookAtVec and UpVec.
T	Stops recording, proceeding to render every stored frame in high res, writing each as a PPM image.

Note: the user defaults to Orthographic High-Res mode with light visualization off.

Scene



Perspective View of Demo Scene Including Lights:

Camera: {175, 105, 14} LookAt: {-0.25, -0.25, -0.93} UpVec: {-0.06, 0.97, -0.25}

The demo scene is predefined with two spheres, 6 triangles (making up a tetrahedron), a plane, and two lights:

Name	*Point(s)/Size*
Sphere 1	{125, 50, -150} Radius = 50
Sphere 2	{128, 41, -64} Radius = 20
Triangle 1	{175, 5, -75} {200, 55, -100} {225, 5, -75}
Triangle 2	{225, 5, -75} {200, 55, -100} {225, 5, -125}
Triangle 3	{225, 5, -125} {200, 55, -100} {175, 5, -125}
Triangle 4	{175, 5, -125} {200, 55, -100} {175, 5, -75}
Triangle 5	{225, 5, -125} {175, 5, -125} {175, 5, -75}
Triangle 6	{225, 5, -75} {225, 5, -125} {175, 5, -75}
Plane	{0, 0, 0} {1, 0, 0} {0, 0, 1}
LightObj 1	{100, 100, -50} Radius = 5
LightObj 2	{231, 63, -127} Radius = 5

Light Location	*Intensity*
{100, 100, -50}	0.2
{231, 63, -127}	0.2

Constant	*Value*
Ambient Intensity	0.5
Distance Away Constant	0.1

Each of these objects have their own defined *ColorPack* giving their shading constants/color (the light objects share a pack and all triangles in the tetrahedron share the same pack):

Name	*Ambient Constant*	*Diffuse Constant*	*Specular Constant*	*Phong Exponent*	*Reflective?*
Sphere 1	{255, 128, 255}	{255, 128, 255}	{255, 255, 255}	16	No
Sphere 2	{255, 128, 128}	{255, 128, 128}	{255, 255, 255}	16	No
Tetrahedron	{128, 255, 255}	{128, 255, 255}	{255, 255, 255}	16	No
Plane	{255, 128, 128}	{255, 128, 128}	{255, 255, 255}	16	Yes
Light Obj	{255, 255, 0}	{255, 255, 0}	{255, 255, 255}	16	No

Note: Background Color is set as {0, 0, 0}

Ray Tracing Intersection

Intersection with geometry functions returned the lowest positive t found.

Sphere/Light Obj

Intersection with a ray from point p in direction d of a sphere or light object uses the formula $t = -d \bullet x \pm \sqrt{(d \bullet x)^2 - x \bullet x + R^2}$ where x is the vector from the center of the sphere to p and R is the radius of the sphere. This returned either one, two, or no solutions for t.

Plane

Intersection with a ray from point p in direction d of a plane uses the formula $t = ((a - p) \bullet n) / (d \bullet n)$ where a is a point on the plane and n is the normal vector of the plane. This returned either one or no solutions for t.

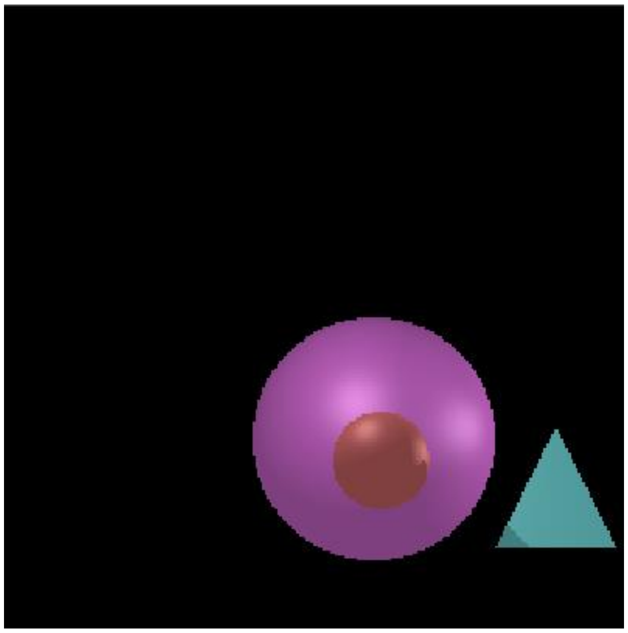
Triangle

Intersection with a ray from point p in direction d of a plane uses the above plane formula to find a t if it exists to check that it is on the plane dictated by the triangle. If so, it checks the edge vector of each side of the triangle in clockwise order, returning the t only if it is within the boundaries of all 3 edges (meaning the returned cross product is lower than or equal to zero).

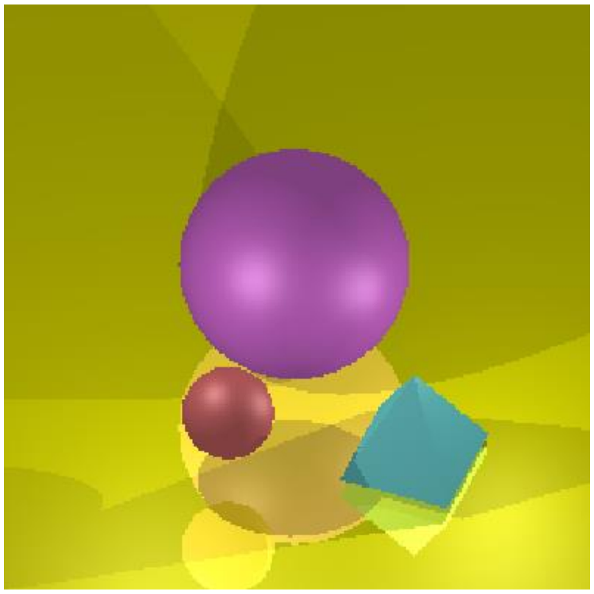
View Type

A basis of w = -lookAtVec, u = upVec x w, v = w x u was used with all being normalized.

Orthographic



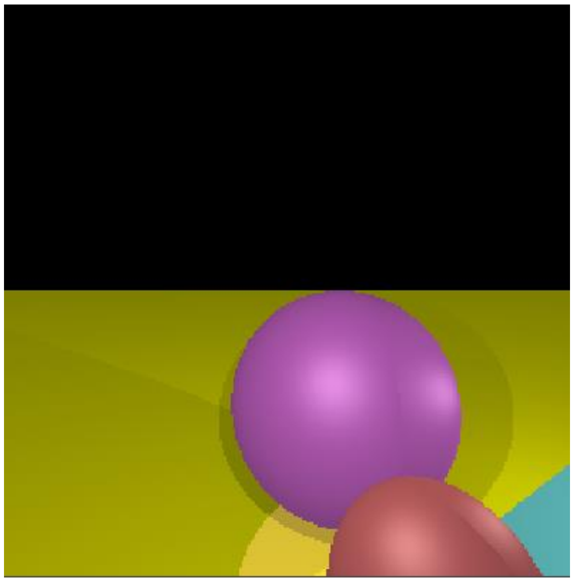
Orthographic View of Default Vectors (plane missing because orthographic camera is perpendicular to it)



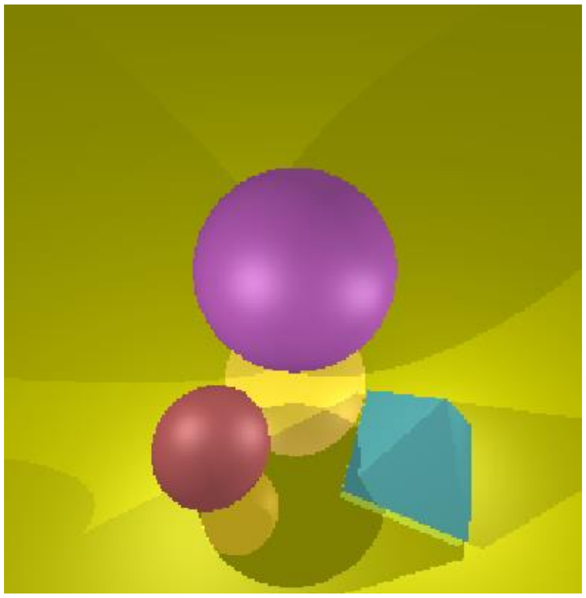
Orthographic View of Camera: {172, 159, -27} LookAt: {-0.25, -0.70, -0.67} UpVec: {-0.25, 0.71, -0.65}

The orthographic camera generates view rays all in the same LookAt direction (-w) at differing origins according to the formula: p = camera + *u* * u + *v* * v where camera is the origin of the camera, and *u/v* are scalars indicating where in the uvw coordinate system the current pixel is.

Perspective



Perspective View of Default Vectors



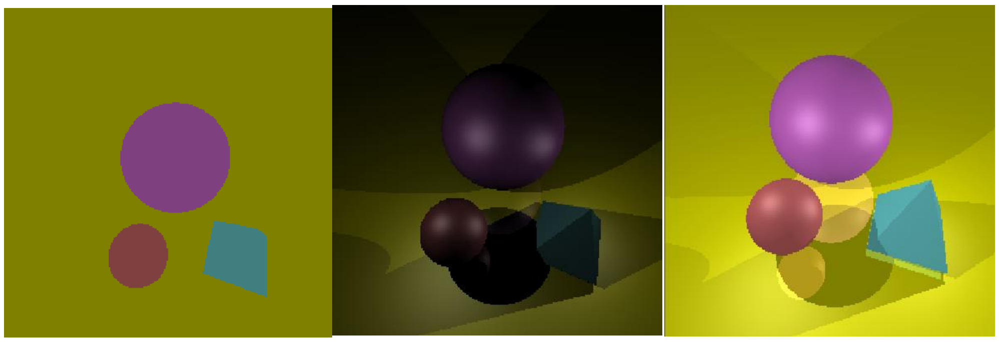
Perspective View of Camera: {172, 159, -27} LookAt: {-0.25, -0.70, -0.67} UpVec: {-0.25, 0.71, -0.65}

The perspective camera generates view rays all from the same camera origin, but with differing directions according to the formula: d = -w * projectionDistance + *u* * u + *v* * v where projectionDistance is a constant. A projectionDistance of imgSize * 16 / 9 was used for all shots.

Shading

Shading took place by adding the lighting from each different method (ambient, diffuse, specular, reflected) and then clamping each RGB value to 1.0f. This produced RGB array was multiplied by 255 outside the findColor function to give the color value for each pixel.

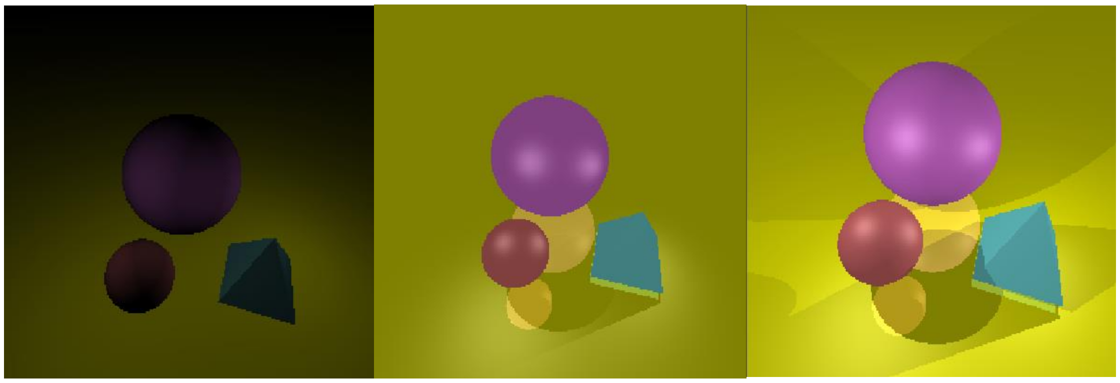
Ambient



Only Ambient vs All Except Ambient vs All Lighting

Ambient light was added to the totalLight only once (even if multiple lights were present) given by the formula totalLight += ambientIntensity * ambientColorConstant. Ambient Intensity was defined as a constant per scene, while ambientColorConstant is the ambient constant of the object given in the *ColorPack*. For lightObj, if visualized, these were simply given *full* ambient lighting, and no other shading. Otherwise, if not visualized these objects were treated as if they did not exist.

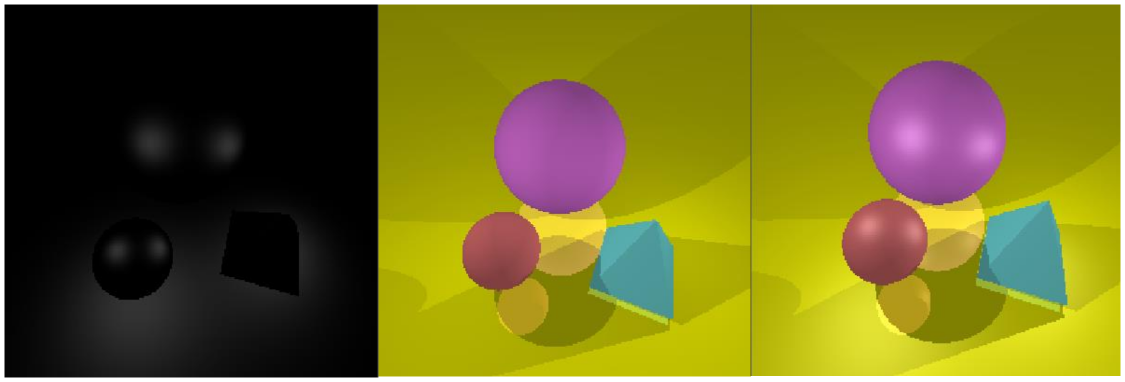
Diffuse



Only Diffuse vs All Except Diffuse vs All Lighting

Diffuse light was added to totalLight for every light source available, given by the formula totalLight += lightIntensity * max(0, n • l) * diffuseColorConstant, where lightIntensity is the intensity of the current light, n is the normal vector of the surface, l is the vector from the surface to the current light source, and diffuseColorConstant is the diffuse constant of the object given in the *ColorPack*.

Specular



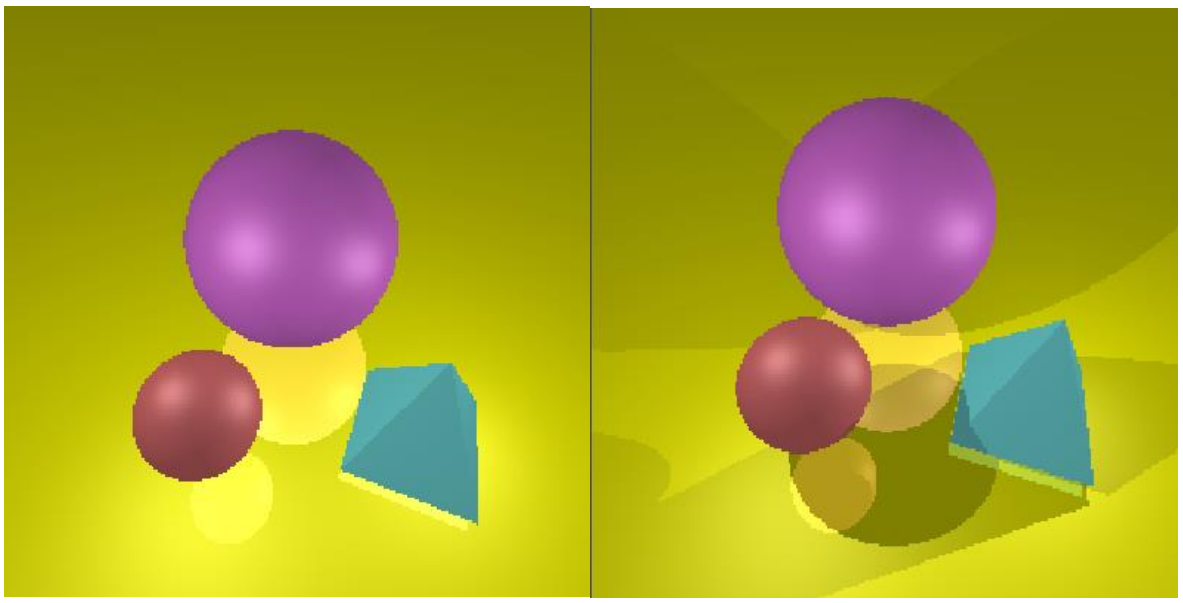
Only Specular vs All Except Specular vs All Lighting

Specular lighting was added to the totalLight for every light source available, given by the formula totalLight += lightIntensity * max(0, (n • h)^p) * specularColorConstant, where lightIntensity is the intensity of the current light, n is the normal vector of the surface, h is the vector halfway between the vector from the surface to the light and the surface to the camera, p is the phong exponent of the object from the *ColorPack*, and specularColorConstant is the specular constant of the object given in the *ColorPack*.

Other Ray Tracing

Other features that used ray tracing were included, that being shadows and reflectivity. For both, the ray traced from the object had a small constant called the distanceAwayConstant added to it to avoid hitting its own surface.

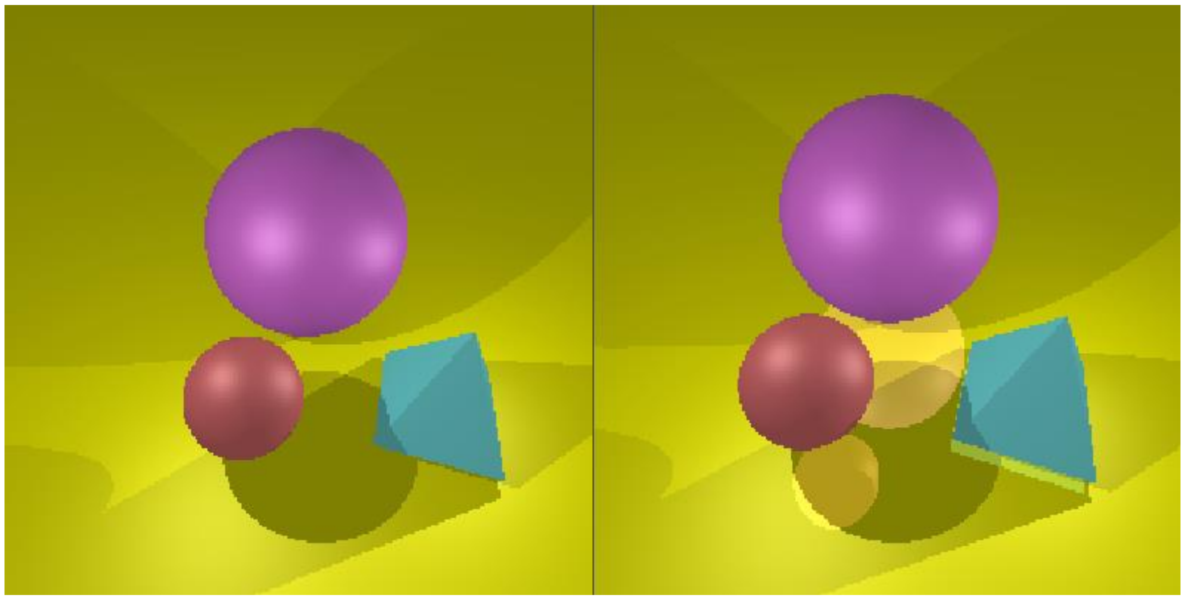
Shadows



Without vs With Shadows

Ray Tracing was repeated, using the ray from the point on the object, to the light source. If an object was intersected that was in between the object and the light source, the diffuse and specular light was not added to the totalLight for that light source. This results in only reflected and ambient light showing up in shadows.

Reflectivity



Without vs With Glazed Reflective Plane

For glazed surfaces, the *findColor* function is called recursively, this time with a ray from the point on the object, in the direction of r = d - 2 * (d • n) * n. A recursion limit of one was used to avoid unlimited recursion. The produced color was added to the totalLight for every light source available, given by the formula totalLight += lightIntensity * findColor() * specularColorConstant, where lightIntensity is the intensity of the current light, and specularColorConstant is the specular constant of the object given in the *ColorPack*. findColor() and specularColorConstant are both vectors, so the multiplication between them was handled via a * b = {a[1] * b[1], a[2] * b[2], a[3] * b[3]}.

Animation Demo

An animation was created using the built-in animation creator in the program (see next section) which created 202 1024x1024 PPM Images. These PPM images were stitched together with FFMPEG (*ffmpeg -framerate 14 -i rayTrace%d.ppm -c:v libx264 -crf 25 -vf "scale=1024:1024,format=yuv420p" -movflags +faststart rayTrace.mp4*). This demo can be viewed at https://www.youtube.com/watch?v=MKGo4FtkftY.

Bonus Advanced Feature: Real Time Viewer/Animation Creator

Overview

Although this project features shadows, an advanced feature not asked for in the rubric, I wanted to create a useful and creative advanced feature that would both help me create demos and line up shots for this report. This feature is the real time viewer and animation creator.

High Res/Low Res

Upon hitting L, a low-res mode will be entered. This mode features 64x64 resolution, but due to this low resolution it will be able to render frames in real time. When you are in this mode, you can control the camera with WASD, Space and Left CTRL, and the mouse (see key binds section). This means you can move the camera to the desired space using the low-resolution image as a guide, and then hit H, to reenter the high-res mode to render the scene. This system makes it easy to view a scene, setup shorts for demos, check for errors, etc.

Transform Matrix

The camera movement of this system is handled by keeping track of the camera position, adding/subtracting x, y, or z depending on the key hit. The rotation, however, is backed by a transformation matrix.

The last mouse position is tracked, and the difference in x and y is used to detect movements. Whenever the mouse is moved (or arrow keys are hit), yaw (for x) or pitch (for y) is added/subtracted. Every frame, this yaw and pitch is put into the transformVector function of *RayTracer*, which will left multiply the lookAtVec and upVec by a yaw transformation matrix and a pitch transformation matrix to get the proper vectors the camera should be rendering.

For the following matrices, a is roll, b is pitch, and c is yaw. A roll matrix is included, and roll is tracked via the Q/E keys in the code, but this feature is not implemented as it resulted in unintuitive movement and is always set to zero in this function. I have included its matrix just for completion's sake.

Roll Matrix

cos(a)	sin(a)	0
-sin(a)	cos(a)	0
0	0	1

Pitch Matrix

1	0	0
0	cos(b)	sin(b)
0	-sin(b)	cos(b)

Yaw Matrix

cos(c)	0	-sin(c)
0	1	0
sin(c)	0	cos(c)

Animation Creator

When in low-res mode, the user can hit R, which will start recording. When recording, all camera views are recorded and stored as movements. During this time, the user can move around the camera in low-res mode with the mouse and keyboard in real time, animating their desired scene. When they are finished, they can hit T which will stop recording, and re-render the movements, but this time in high resolution. These produced high-resolution frames of the animation are saved in the project directory as ppm files named rayTraceX.ppm where X is the frame number. As seen in the animation demo, ffmpeg can be used to easily stich these produced high resolution frames together, producing an animation of the movements you did via mouse and keyboard in real time in the low-res mode.

Demo of Real Time Viewer/Animation Creator

A demo of this feature can be found at https://www.youtube.com/watch?v=6YN4cONh3FM.

Sources

This project used no outside libraries except for the GLFW and GLEW libraries required. No code was copied or repurposed from any sources, except for the initial main file provided by the professor. Several sources were referenced for technical information:

- Equations and Fundamental Concepts were from the Lecture slides and Textbook
- Main file takes portions of code from the main file provided by the professor for GLFW/GLEW rendering of 2D array
- I referenced https://en.wikipedia.org/wiki/Transformation_matrix when creating my own transformation matrices for my coordinate system
- PPM was recommended to me by https://stackoverflow.com/questions/36288421/c-create-png-bitmap-from-array-of-numbers and I got the proper format for writing PPM files from https://netpbm.sourceforge.net/doc/ppm.html
- I used the FFMPEG command from https://stackoverflow.com/questions/68651839/how-to-convert-a-lump-set-of-ppm-files-to-a-single-mp4-video-using-ffmpeg