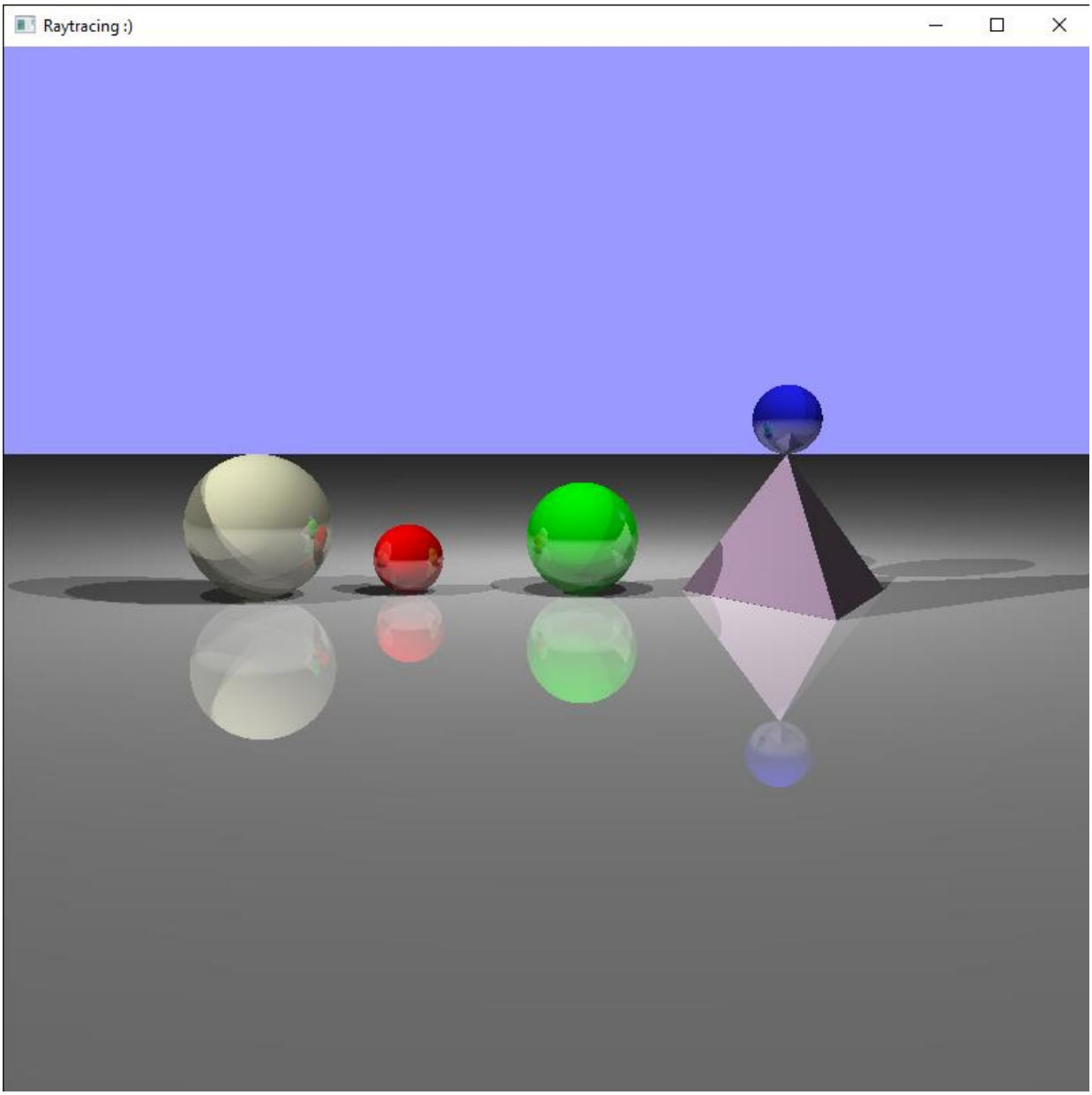
Ava McCormack
CAP4730
Project 1

Ray Tracer



Breakdown of this Report

The Code Overview will be explanations of my files and classes with details on functions.

Following this, I will include the values I initialized things in my program.

Lastly, the Theory and Formulae will go into details of what formulas I used, and more specifics of certain functions (like intersect).

note: The video also demonstrates functionality that I added besides the animations. The animations start at the 37 second mark

Ava McCormack
CAP4730 Ray Tracer
Project 1

Links

Code Overview

Initializations

Theory and Formulae

References

Fun picture progression

(not very comprehensive since I kept forgetting to take pictures)
(also they are mostly phone pictures because I didn't plan to do this)

Ava McCormack
CAP4730
Project 1

Ray Tracer

Code Overview

Classes: `Camera` -> `OrthoCam` , `HitResult`, `Material`, `Ray`, `Scene`, `Shapes` -> `Plane`, `Triangle`, `Sphere`

Main cpp: I have organized main into several functions

```
// the default int main() of course
framebuffer_size_callback // window resizing

key_callback // processes all user input
createScene // initializes important variables in scene
generateCams // will generate and save cameras that move around the scene
generateCams2 // a second animation. Better for perspective mode
rayTrace // handles raytracing loop
initOpenGLstuff // will initialize all glew, glfw, and opengl things
```

int main: inside int main, I initialize everything by calling `initOpenGLstuff` and `createScene`. Afterwards the render loop will begin, in which `rayTrace` and `scene.nextCam` will be called.

`framebuffer_size_callback`, and `initOpenGLstuff`, are both code from the given SimpleTexture.cpp [0] file. I simply put all glew, glfw, and OpenGL initialization into a function in order to clean up main.

Inside `createScene`, I have initialized 4 spheres, 1 tetrahedron, 1 plane, 2 initial cameras, and 3 lights. Each object is then saved to its respective variable in my Scene class.

Both `generateCams` functions will initialize cameras to be saved to the `Scene` Class. These cameras will then iteratively be set as the current camera and the viewpoint will change allowing movement around the scene. (`scene.nextCam` is what is called to change to the next camera by iterating through a camera vector stored in the `Scene` class) Obviously, with very high resolution (or slow hardware), the camera changes will be quite slow; therefore I have also built in a way to change resolution.

`key_callback` is where all keyboard inputs will be handled.

- Pressing key “1” will change perspectives.
- Pressing key “2” will start an animation by running `generateCams`.
- Pressing key “3” will start animation 2 by running `generateCams2`.
-
- Pressing the up arrow will increase resolution by 100 pixels in both width and height.
    - Meaning if we start with 800x800 and press the up arrow, we will now have 900x900.

- Similarly, pressing the down arrow will then decrease resolution by 100 pixels in both width and height.
- Pressing “4” , “5” , or “6” will flip the switch for lights 1-3 respectively (key 4 controls light 1, etc)
    - This enables users to turn on or off the 3 lights to see the effects
- Pressing “8” will decrease the number of reflective bounces (lowest is 1)
- Pressing “9” will increase the number of reflective bounces (highest is 5)

`rayTrace` is the function called from inside the GLFW window loop that will handle the raytracing loop. Looping over the width and height, we will then generate a ray by calling the camera class function, `generateRay`. This generated ray will then be passed to the Scene function, `traceRay`, to see if it intersects with any of our objects (shapes). If there was a hit, another Scene function, `shadeRay`, is called to find the shading of that pixel. That shading is then saved to an array. If the ray does not intersect with any object (shape), a default color is assigned. After this width X height loop is done, the pixel colors array is passed to `glTexImage2D` and `glGenerateMipmap` in order to be displayed to the screen.

Scene:

My `Scene` Class holds many relevant variables including: cameras, objects, and lights. This class also has several key functions including `traceRay`, `shadeRay`, `switchPerspective`, and many getters and setters. This class will also keep track of the current camera.

When shapes are initialized in main, they are also saved to a vector in scene class along with lights, and cameras. `switchPerspective` is the function that will swap between the perspective and orthographic camera when “1” is pressed on the keyboard. `nextCam` is the function that will iterate over the vector of camera objects which are created in main after “2” is pressed.

The 2 important raytracing functions are located in this class: `traceRay` and `shadeRay`.

`traceRay` is the function that will determine the closest object hit by the passed in ray. This function utilizes the `HitResult` class. A `HitResult` object is created and then a loop over all of the objects (shapes) (stored in a vector in this, `Scene`, class) is started. Each object (shape) has its own intersect class that is called. The resulting `HitResult` object is compared to the one we created before to give us the closest hit result by the end of this loop.

`shadeRay` is the function that will determine the color of a pixel. An object (shape) stores a `Material` class object that stores information about the ambient, diffuse, and specular colors and intensities. This class will determine the color by summing ambient, diffuse, specular, and glazed (mirror reflections). I have also included shadows in this raytracer.

Light:

The `Light` class is a small class containing a light's position, status (on or off), and the function to negate the current status.

Camera:

My `Camera` class holds information including position, direction, up, right, etc. Position and Direction are both retrievable using getter functions. I have designed `Camera` so that perspective geometry is the default Camera class, and an orthographic geometry is a subclass of `Camera` called `OrthoCam`. `OrthoCam` uses the exact same functions except for the `generateRay` function where a different formula is needed.

As a comparison:

```
// perspective
    Ray ray (this->pos, glm::normalize(this->right * u + this->up * v + this->dir
* -800.0f));

// orthographic
    Ray ray (this->pos + (u * this->right) + (v * this->up), this->dir);
```

Ray:

As you can see, a `Ray` class is used to encapsulate all information about rays. This class is rather compact and only has 2 private member variables: origin and direction (that are retrievable using getter functions). This class also has a function called evaluate that returns the value of a ray once its t value is found:

```
return origin + t*dir;
```

HitResult

`HitResult` is another small class that contains information regarding a ray and surface intersection. There is a Boolean value that denotes if the ray and surface intersected. If a ray and surface intersect, the hitPt (place on the surface that the ray intersected with), normal, t, object's `Material` are saved.

Shapes:

I have created a parent class called `Surface` that has 3 children classes: `Plane`, `Triangle`, and `Sphere`. The `Surface` class holds a `Material` object variable which can be defined for each subclass (shape). Each subclass stores relevant information about that shape and a specialized intersect function that returns a `HitResult` object.

The following is what each class holds to describe its respective shape:

`Plane`: position, normal

`Triangle`: vertices, normal

`Sphere`: origin, radius

Material:

`Material` class holds information that determines the shading of a shape: ambient, diffuse, specular colors and intensities and a Phong (p) value.

INITIALIZATIONS

(all values were initialized as floats)

`Shapes`

(4 triangles made up tetrahedron)

PLANE	Position	Normal	RGB
	-400, 0, 5000	0, 1, 0	200, 200, 200
SPHERE	Radius	Origin	RGB
	25	175, 125.1, -2	20, 20, 255
	25	-100, 25.1, 20	255, 0, 0
	40	25, 40.1, 20	0, 255, 0
	50	-200, 50.1, 45	255, 255, 204
[V]ertex 0	[V]ertex 1	[V]ertex 2	[V]ertex 3
175, 100.2, 0	250, 0.2, 0	100, 0.2, 0	175, 0.2, 150
TRIANGLE	Vertices	Normal	RGB
	V0, V1, V2	*	255, 204, 255
	V0, V1, V3	*	255, 204, 255
	V0, V2, V3	*	255, 204, 255
	V1, V2, V3	*	255, 204, 255

*Triangle normal is found by taking cross product of 2 vertices.

`Cameras (first 2)`

	Position	LookAt	Up
Orthographic	0, 2, 30	0, 0, -1	0, 1, 0

Perspecive	0, 100, 800	0, 0, 0	0, 1, 0

From these, I was able to solve for the camera basis: {u, v, w}

```
this->dir = glm::normalize(pos - lookAt); // w
this->right = glm::normalize(glm::cross(up, this->dir)); // u - right
this->up = glm::normalize(glm::cross(this->dir, this->right)); // v - up
```

`Lights`

	Position	Status
1	-250, 200, 150	True
2	0, 250, 150	True
3	100, 250, 200	True

THEORY AND FORMULAE

RAYTRACING

Inside `traceRay`, we get the functionality of determining whether a ray has intersected with any object (shape) from our vector. Internally, this function iterates through the `Shape` objects, then calls the `Shape`-specific intersect function. After receiving the results, it calculates the closest t, or the `Shape` that was intersected first.

INTERSECT FORMULAE: SPHERE

For the sphere intersect formula, I used the textbook given formula [1].

First, I found the discriminant:

$$discriminant = \big((p-c)\cdot d\big)^2 - (d\cdot d) * \big[\big((p-c)\cdot(p-c)\big) - r^2\big]$$

Where: p = ray origin

D = ray direction

C = sphere origin

R = sphere radius

If discriminant == 0, ray hit something.

if discriminant < 0, ray does not hit something

if discriminant > 0, we find t by completing the quadratic equation:

$$t = \frac{(-((p-c)\cdot d) - \sqrt{discriminant})}{(d \cdot d)}$$

$$t2 = \frac{(-((p-c)\cdot d) + \sqrt{discriminant})}{(d \cdot d)}$$

We then check, whichever t is the smallest or largest among the two, and within the range tmin , tmax is where our ray intersects an object(shape).

Whenever we have a hit, the hitResult object is then updated with:

Boolean: hit = true

t value: t = t or t2

the point where the ray and object intersect: hitPt = ray.evaluate(t)

normal: normalize(hitPt – sphere origin)

INTERSECT FORMULAE: TRIANGLE

For the triangle intersect formula, I used the textbook given formula [1] to find β, γ , t, and M

These are given by (images straight from textbook) :

$$\mathbf{A} = \begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix},$$

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} j \\ k \\ l \end{bmatrix},$$

Cramer's rule gives us

$$\beta = \frac{j(ei - hf) + k(gf - di) + l(dh - eg)}{M},$$

$$\gamma = \frac{i(ak - jb) + h(jc - al) + g(bl - kc)}{M},$$

$$t = -\frac{f(ak - jb) + e(jc - al) + d(bl - kc)}{M},$$

where

$$M = a(ei - hf) + b(gf - di) + c(dh - eg).$$

We then check, if t is in the range tmin and tmax AND gamma between (0, 1) AND beta between (0, 1-gamma) , then we have a hit.

Whenever we have a hit, the hitResult object is then updated with:

Boolean: hit = true

t value: t = t

the point where the ray and object intersect: hitPt = ray.evaluate(t)

normal: normalize((v[1] – v[0]) X (v[2] – v[0])) a normalized cross product

INTERSECT FORMULAE: PLANE

I used this stackOverflow answer : **https://stackoverflow.com/a/23976134** [2]

First I found the denominator:

$$denom = this \rightarrow normal \cdot d$$

If absolute value of denom is greater than min, we find t.

$$t = \frac{(planePos - p) \cdot planeNormal}{denom}$$ reminder: p = ray origin

If t is within the range (tmin, tmax), then we have a hit.

Whenever we have a hit, the hitResult object is then updated with:

Boolean: hit = true

t value: t = t

the point where the ray and object intersect: hitPt = ray.evaluate(t)

normal: plane's normal

LIGHTING-SHADING

After `traceRay`, we are back inside the `rayTrace` function in main. If the ray intersected with an object (shape), we will call `shadeRay` to determine the color of the pixel. The following are the formulas used. All shading formulas were found in the book [1] and lecture slides [3]

L = {0.0f, 0.0f, 0.0f}

```
AMBIENT: L += mat.ambientColor * mat.ambientI;
DIFFUSE: L += mat.diffuseI * mat.diffuseColor * max(0.0f, glm::dot(lightDir,
hit.normal));

SPECULAR: L += mat.specularI * mat.specularColor * max(0.0f,
glm::pow(glm::dot(vh, hit.normal), mat.p));
```

Where vh =

```
glm::vec3 vh = (light.pos + hit.hitPt)/glm::length(light.pos + hit.hitPt);
```

```
MIRROR: L += reflectHit.material.specularI * this->shadeRay(reflectRay, tmin,
tmax, limit);
```

To keep it simple, ambient and diffuse are the same for an object (shapes). Specular is also the same but multiplied by 3. All intensities are the same throughout all objects (shapes).

Ambient intensity	0.2
Diffuse intensity	0.4
Specular intensity	0.4
P	1.0

ANIMATION

Ava McCormack
CAP4730 Ray Tracer
Project 1

For my animation, I used the `generateCams` paired with `scene.nextCam` to automatically generate an animation and then iterate through the cameras used to acquire different positions around the scene. There are 2 premade animations made. One is a simple 360 around the scene and is suited for either camera modes (perspective/orthographic). While the other is also a 360 it steps forward in the middle to zoom in on one of the spheres. The second animation (activated by key “3”) is best suited for perspective mode and will look strange in orthographic.

I have also included a 3rd animation just for submission (and not to be activated by a key press since it is very slow).

All animations are edited to be faster in the video

References

[0] simpleTexture.cpp – OpenGL initialization

[1] “Fundamentals of Computer Graphics'', Steve Marschner and Peter Shirley – intersect and shading formulas

[2] **https://stackoverflow.com/a/23976134** - plane formula

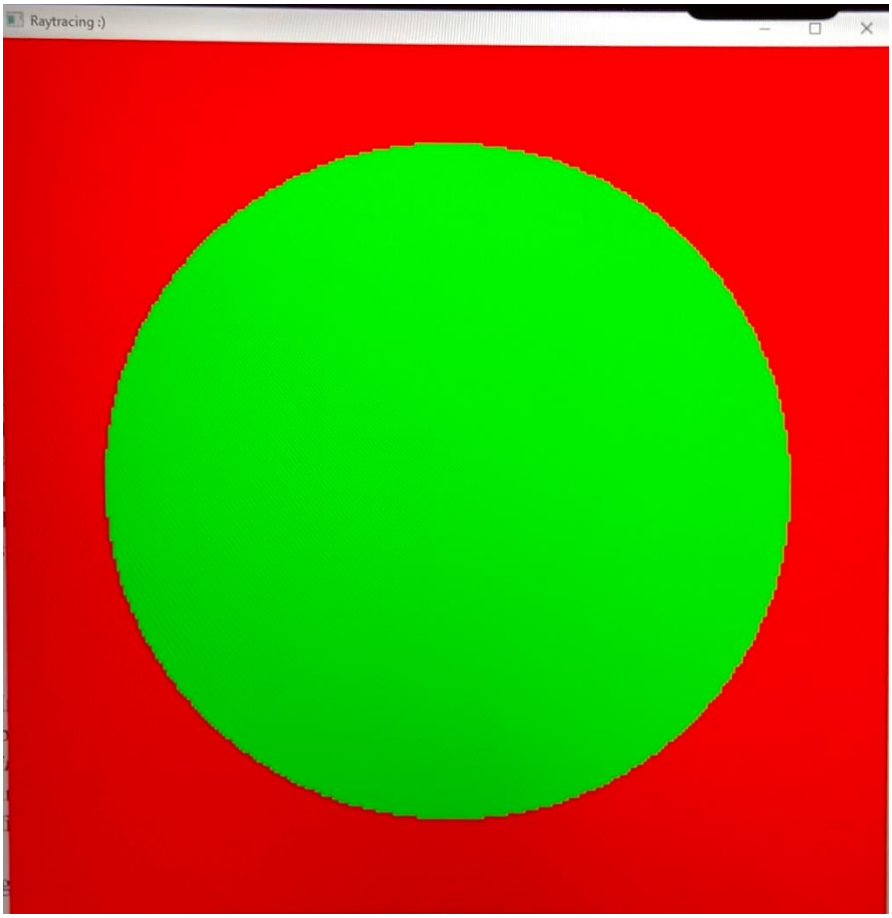
[3] https://ufl.instructure.com/courses/505309 lecture slides - shading formulas

Ava McCormack
CAP4730
Project 1

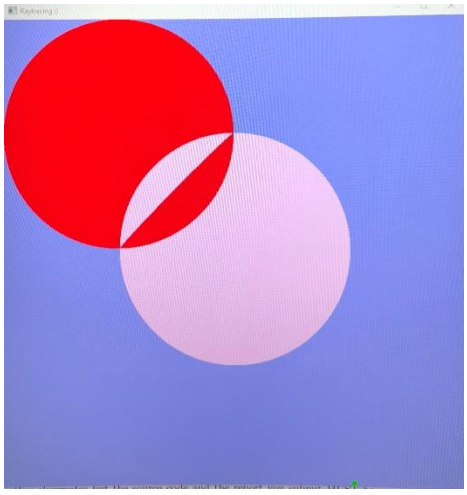
Ray Tracer

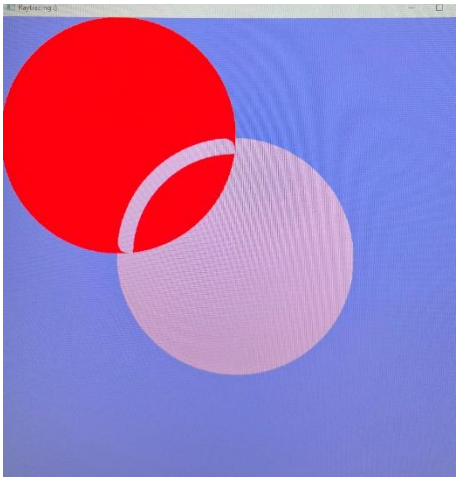
Pictures Just for Fun

The beginning …. This circle was actually made completely wrong ! it was supposed to be a red circle and green background



Adding some more circles and getting some fun clipping

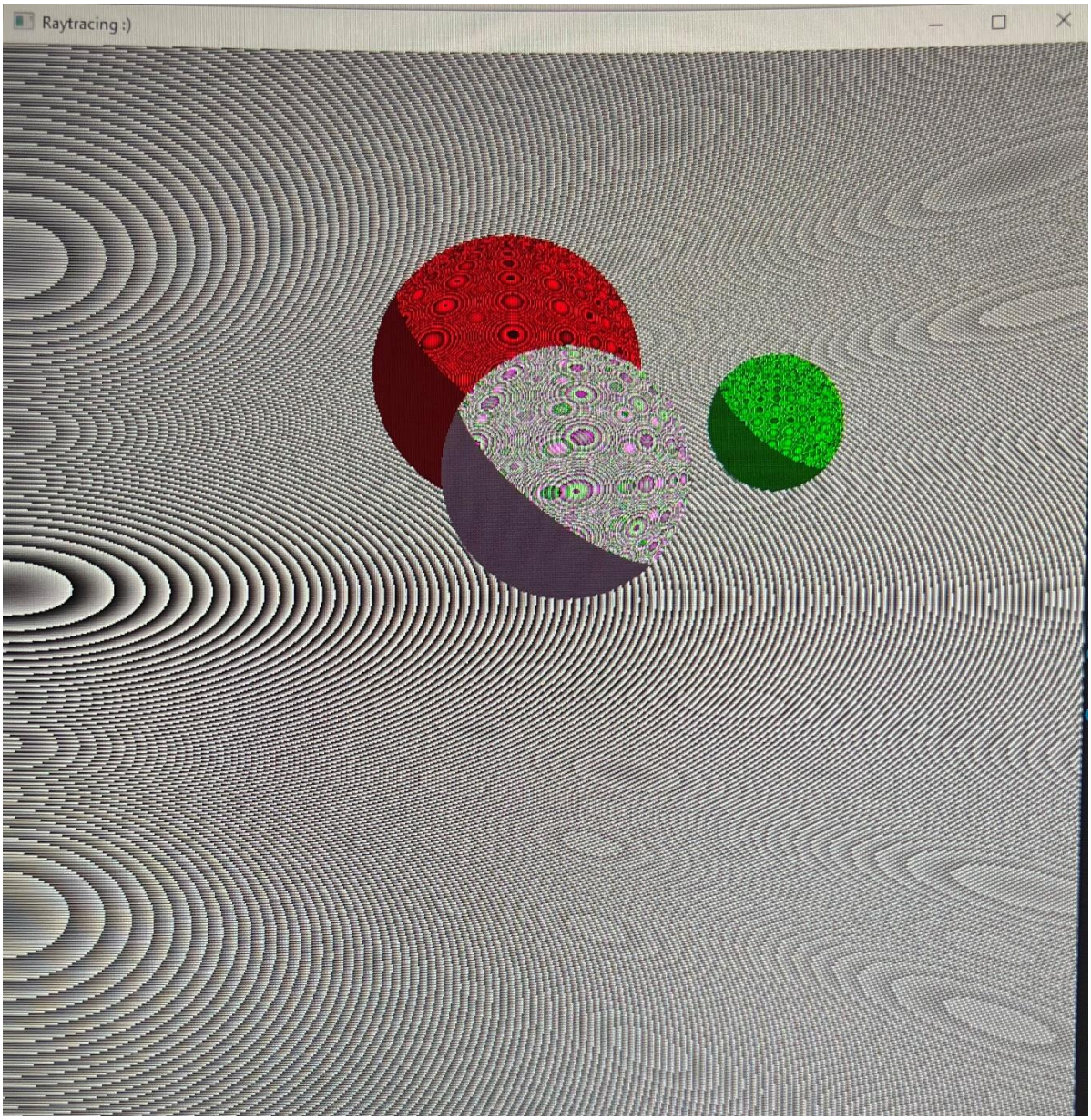




Ava McCormack
CAP4730
Project 1

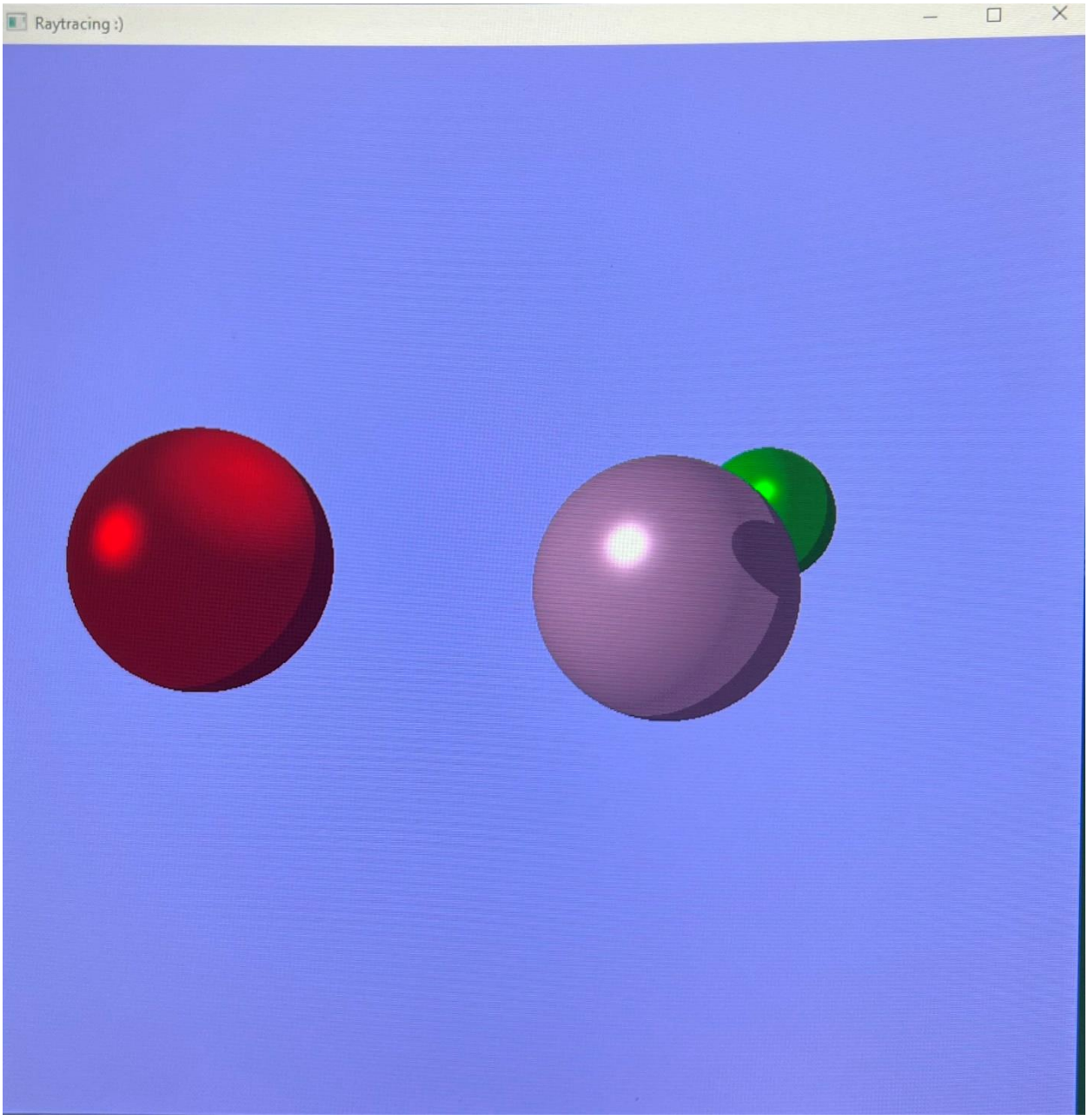
Ray Tracer

Moving on to lighting… um



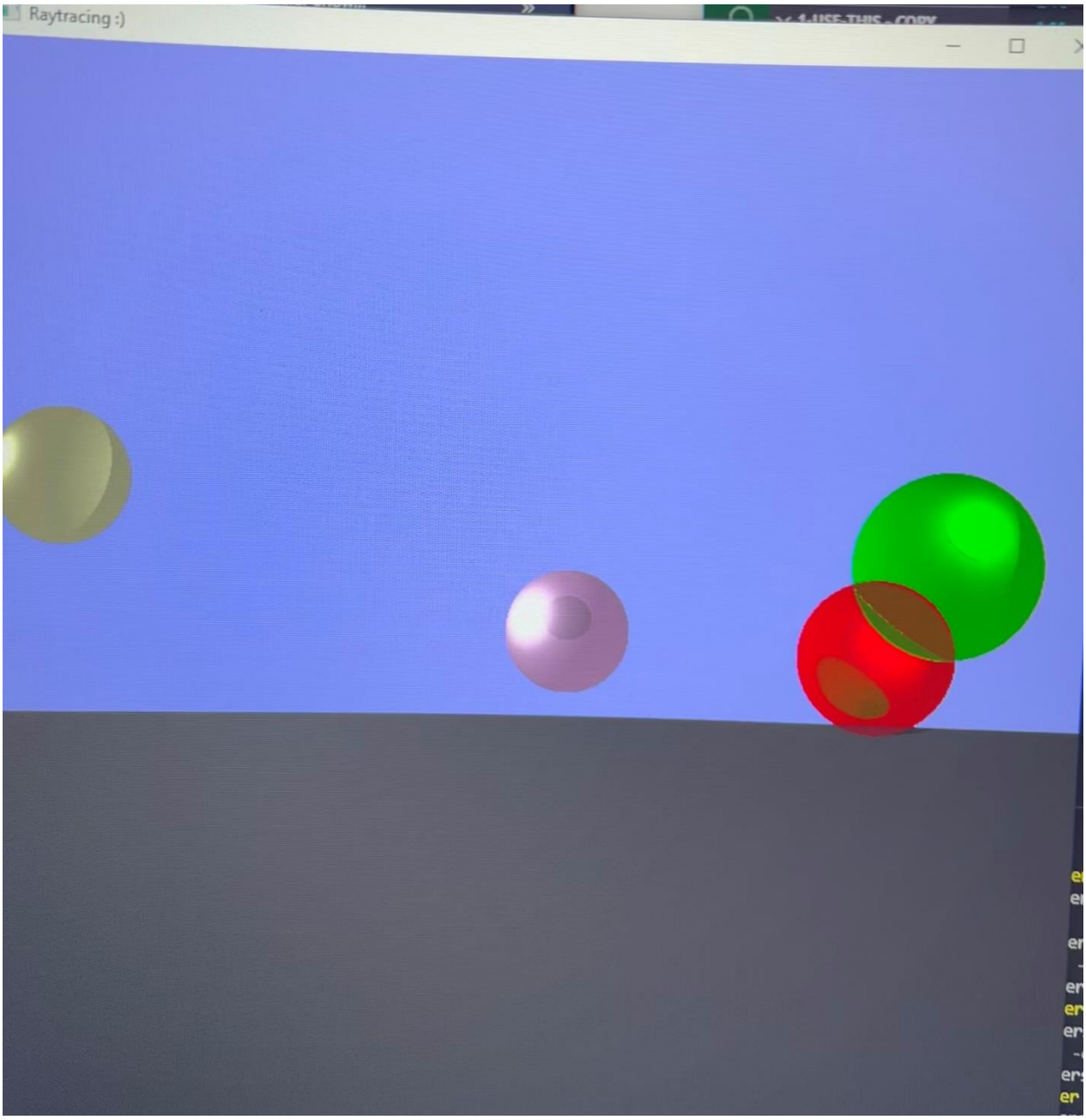
Ava McCormack
CAP4730 Ray Tracer
Project 1

Oh wait we're good… they look so good !



Ava McCormack
CAP4730 Ray Tracer
Project 1

it got worse before it got better



Ava McCormack
CAP4730
Project 1

Ray Tracer

It took awhile but we finally got here