Assignment 1

CAP 4730

Arsh Panesar
UFID: 2047-3653

Ray Tracer

1 Introduction

This report contains my work on the first assignment where I implemented the Ray Tracing algorithm to display digital objects with realistic lighting. Ray Tracing is a technique that is used to render shapes such as spheres and triangles by generating rays and intersecting with the shapes in the scene. It is a powerful albeit slow algorithm to produce realistic looking scenes when combined with shading. This report will cover the math and code used in the Ray Tracer project.

2 Platform and Compiling Instructions

I have used the Windows operating system to build this project, alongwith the following technologies:

1. Libraries for OpenGL: GLFW and GLEW were used to run the sampleTexture.cpp program provided in the assignment. This allowed for an image to be used as output for the Ray Tracer.
2. MinGW: MinGW binaries were installed to run the project. The project uses the *Clang++* compiler along with the *lldb* linker.
3. Premake5: I used premake as a build system for my project since it was easier to build the project with multiple source files and headers.
4. Gmake: Premake acts as a makefile generator, and the project can be compiled by using the *gmake* package.
5. CImg: CImg is a C++ library to manipulate and save images on disk. This is used to create an animation with the Ray Tracer.
6. Visual Studio Code: This was used as an IDE to edit and run the project. The *tasks.json* and *launch.json* configuration files were used to automatically run premake5 and then using make to compile the project.

To compile the project, open the project folder in Visual Studio Code and press F5. This will run the entire compilation process and executes the binary generated. The *tasks.json* file contains two tasks: 1. Run premake5 with the gmake argument to generate a makefile for the project, and 2. Compile the Project using the make command. The *launch.json* project simply launches the binary generated after compilation. Premake5 and make are necessary to

compile the project. The C++ compiler and linker can be changed to use g++ and gdb linker instead, by editing these files: *premake.lua, tasks.json* and *launch.json*. All libraries and include files are provided with the project.

3 Utilities: Vector, Ray, Color and Image

3.1 Vector and Ray class

I started the project by building a few utility classes to make development and testing easier. The Vector class is used to represent a Vector in 3D space, and has operator overloads for vector arithmetic(addition, subtraction, multiplication by scalar, division by scalar). There are functions for computing the magnitude, dot product, cross product and normalization as well.

The Ray class is a simple representation of a ray in 3D space. As we are using parameterized equations, the ray class only contains an origin **e** in world coordinates and a normalized direction **d**. Rays are extensively used to test intersections with objects in the scene, and an intersection point **P** can be computed by finding a parameter **t**, which is the distance between the ray's origin and the intersection point:

$$P = e + d * t$$

3.2 Color and Image class

The Color class uses `unsigned char` to represent 8-bit RGB values. It comes with two functions that are used for the shading process: the first is `mix_light()` function which takes an intensity of light as a parameter. The computation involves multiplying each component **r, g, b** with a light intensity **I**, and then clamping their values between 0-255.

$$mix\_color(I) = (r * I, b * I, g * I)$$

The second function is `weighted_combine()`, which combines the caller's color **C** with another color $\mathbf{C_0}$ provided as a parameter, alongwith the weights $\mathbf{W_C}$ and $\mathbf{W_{C0}}$ of each color. The sum of the weights must be kept between 0 and 1.

$$weighted\_combine(C, C_0, W_C, W_{C0}) = C * W_C + C * W_{C0}$$

$$0 \geq W_C + W_{C0} \leq 1$$

An Image class is used to store the output image that is displayed with OpenGL as a texture. It uses a single array of size `w * h * 3`, where w and h are the width and height of the image, and 3 corresponds to the RGB values of each pixel.

4 The Scene

My scene contains four objects: 2 Spheres, a Triangle (acts as a plane), and a Tetrahedron. A camera is placed in the Perspective view style which looks at the tetrahedron, and each sphere is on the left and right of it.

The objects have specifications as given below. The color name is derived from Wikipedia's list of colors.

Sphere	Position	Radius	Color	RGB Value
Left Sphere	(-300, -425, 1200)	75	Barn Red	(124, 10, 2)
Right Sphere	(100, -400, 1350)	100	Midnight Blue	(25, 25, 112)

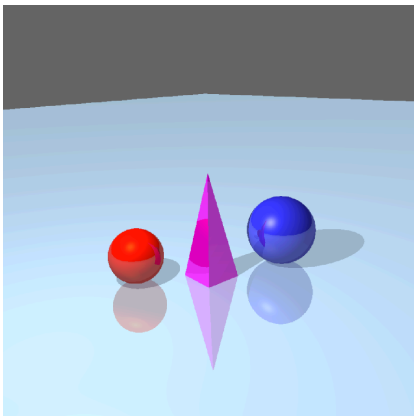
A triangle is added to simulate a plane below the objects. The color of the triangle is set to "Wild Blue Yonder", with RGB values (156, 181, 203).

Plane (Triangle)	Position
Point 1	(-10000, -500, -10000)
Point 2	(-10000, -500, 100000)
Point 3	(10000, -500, -10000)

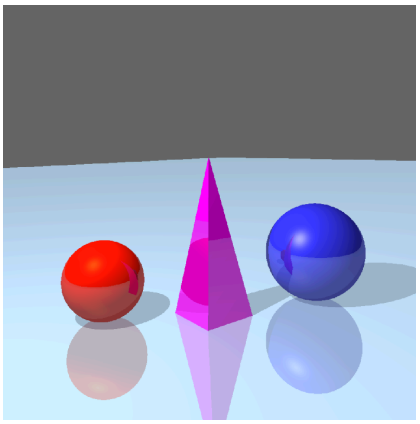
A tetrahedron is a complex shape built using 4 triangles - 3 side triangles and a base triangle. All triangles share the same color, which is "Patriarch", with RGB values (128, 0, 128). An additional position called the "top point" is also used to describe where all the side triangles meet i.e. it determines the height of the tetrahedron.

Tetrahedron	Point 1	Point 2	Point 3
Top Point	TP: (-100, -200, 1216)	-	-
Base Triangle	BP1: (0, -600, 1250)	BP2: (-100, -600, 1150)	BP3: (-200, -600, 1250)
Side Triangle 1	BP1	BP2	TP
Side Triangle 2	BP2	BP3	TP
Side Triangle 3	BP1	BP3	TP

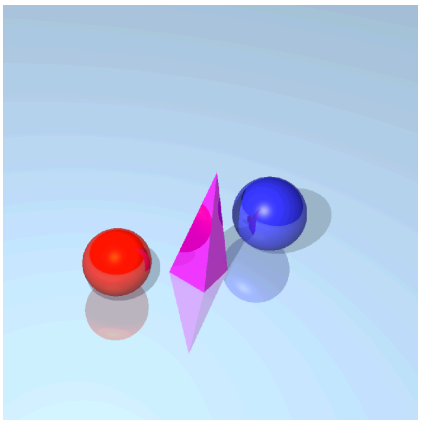
This setup generates the following images, the camera position **CP** and the look direction **CW** is provided below the image. The camera basis is set in a way that it always faces towards the tetrahedron with some offset **CO**.



CP: (0, 0, 200)
CO: (0, 100, 0)
CW: CP - TP + CO



CP: (0, -200, 500)
CO: (0, 100, 0)
CW: CP - TP + CO



CP: (-300, 200, 500)
CO: (0, 100, 0)
CW: CP - TP + CO

4.1 Shapes

All shapes in the project must first derive the *IShape* abstract class, which contains fields that are shared by all derived shapes such as `{position, color, lighting coefficients}`. It also contains a pure virtual function named `ray_intersects()` that every shape must implement. By deriving from the *IShape* class, I built the Sphere and Triangle classes and implemented the ray intersection function.

4.1.1 Sphere

A sphere can be defined using a center position **X**, and a radius **R**. For a ray defined by origin **e** and direction **d**, we can check if a ray intersects the sphere, and by finding a suitable distance **t** from the ray's origin to a point on the sphere, we can acquire the intersection point **P** as well [0].

To check if a ray intersects the sphere, we need to find a value **D**, and check if it is greater than or equal to 0. If it is equal to 0, it grazes the sphere. If it is negative, then there is no intersection.

$$eX = e - X$$
$$D = (d.eX)^2 - d.d(eX.eX - R^2)$$

We need to find where the intersection points are located. When **D** > 0, then there will be two intersection points - these are present on front and back of the sphere. We can use **D** to find the two intersection points by finding two values of **t** named $\mathbf{t_0}$ and $\mathbf{t_1}$. To store the front intersection point in $\mathbf{t_0}$, we must find the minimum and maximum of both variables, and use the one which is > 0. If both are negative, there is no intersection.

$$t_0 = \frac{-d.eX + \sqrt{D}}{d.d}$$
$$t_1 = \frac{-d.eX - \sqrt{D}}{d.d}$$

$$t_{min} = min(t_0, t_1)$$
$$t_{max} = max(t_0, t_1)$$

4.1.2 Triangle

A triangle can be specified with three points **a**, **b** and **c**. Given a normal $\mathbf{N_t}$ and a ray, we can check if the ray intersects the triangle or not [1]. We first check if the ray is parallel to the triangle by computing the dot product of the $\mathbf{N_t}$ and ray's direction **d**. If it is negative, there is no intersection.

We then check if the triangle is behind the ray and does not need to be rendered:

$$t = \frac{N_t.e + (-N_t.a)}{N_t.d}$$

If t >= 0, the triangle has to be rendered. We can now find an intersection point **P** that lies on the plane of the triangle. We can check if the point is inside the triangle by computing the cross product between an edge of the triangle and the vector between the intersection point and the intersection point, and then finding the dot product of this cross product and the normal.

$$C_0 = N_t.(\ (b-a) \times (P-a)\ )$$
$$C_1 = N_t.(\ (c-b) \times (P-b)\ )$$
$$C_2 = N_t.(\ (a-c) \times (P-c)\ )$$

If any value $\mathbf{C_0}$, $\mathbf{C_1}$ or $\mathbf{C_2}$ is negative, then **P** lies outside the triangle. Otherwise, the point must be rendered and we can use **t**, **e** and **d** to find the intersection point.

4.1.3 Tetrahedron

A tetrahedron is a complex shape as it is built out of four triangles - three side triangles and a base triangle. As explained before, it makes use of a vector called *top point* that is used to set the height of the tetrahedron. This is a point which all side triangles share. When it has to be rendered, each triangle is checked for an intersection separately.

4.2 Camera

The Camera class is designed to represent a viewing plane in the scene. It consists of the following fields:

- `center_position: Vector3` is used to define the center of the viewing rectangle.
- `image_size: Vector3` is used to define the width and height of the image.
- `window_size: Vector3` is used to define the width and height of the camera's viewing rectangle
- `center_basis_u, v and w: Vector3` are used to define the camera basis.
- `viewpoint_distance: float` is used to define the focal length of the camera for perspective viewing.

4.2.1 Computing Camera Basis (lookAt)

The camera basis is calculated by first acquiring a **w** vector that will be opposite to the viewing direction. We can get **w** by subtracting the target position $\mathbf{P_T}$ that we want to look at, from the camera's position $\mathbf{P_c}$. Then, we can calculate the **u** vector by producing a cross product of **w** and a vector **t** that is not collinear with w. Lastly, the **v** vector can be computed using the cross product of **w** and **u** [0].

$$w = P_T - P_c$$
$$u = t \times w$$
$$v = w \times u$$

The order of cross product has to be maintained as it is assumed that we are using a right hand coordinate system.

4.2.2 Centering the Pixel Coordinate

Before a ray is produced, we must first convert the image's pixel location (**i**, **j**) to the center of the pixel on the camera's viewing plane (**x**, **y**). Given the image's width and height ($\mathbf{I_W}$, $\mathbf{I_H}$) and the viewing plane's width and height ($\mathbf{V_W}$, $\mathbf{V_H}$) , we can get **x** and **y** by performing the given calculation:

$$x = \frac{i + 0.5}{I_W} * V_W - \frac{V_W}{2}$$
$$y = \frac{j + 0.5}{I_H} * V_H - \frac{V_H}{2}$$

In this project, the viewing plane's dimension and the image's dimension are kept the same. The values used in the project are 1024 x 1024.

4.2.3 Orthographic Projection

For producing an orthographic projection, a ray must start from the pixel coordinate (**x, y**) and have a direction **-w**.

$$e = x * u + y * v + P_c$$
$$d = -w$$

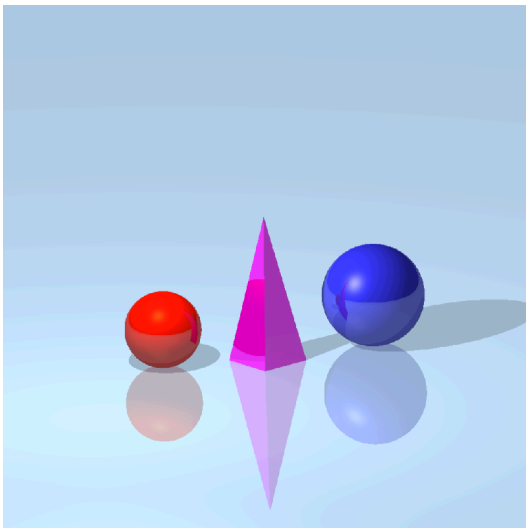
Orthographic Projection produces a ray from each pixel and keeps the direction of the ray fixed. This generates a parallel projection of 3D space onto a 2D plane.

4.2.4 Perspective Projection

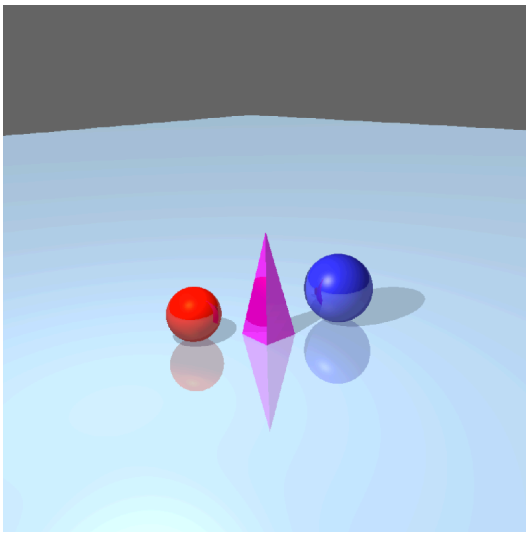
For producing a perspective projection, a ray must start from the center of the camera with some distance called the focal length **f**. The direction of this ray will be towards each pixel coordinate (**x, y**).

$$e = P_c$$
$$d = x * u + y * v - w * f$$

The results of each kind of projection are given below.



ORTHOGRAPHIC
CP: (0, 0, 0)



PERSPECTIVE
CP: (0, 0, 0)

A user can switch between orthographic and perspective modes using the "***Enter***" key. Since the program is single-threaded, the screen may freeze during the switch. The `viewpoint_distance` is set to 1000 units, as a large distance is required to make a smaller Field-Of-View (FOV) angle. A large FOV produces distortion in perspective projection [2].

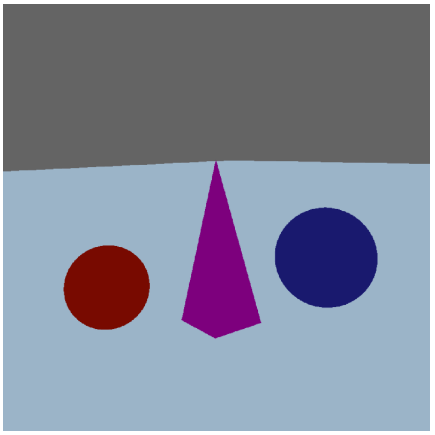
4.3 Shading

Shading is an important aspect of computer graphics, and without it, realistic scenes cannot be generated. Shading allows us to produce lighting effects in the world, and this gives a significant improvement in the scene. This project employs diffuse, specular and ambient lighting effects. Along with this, shadows and glaze effects are also added.

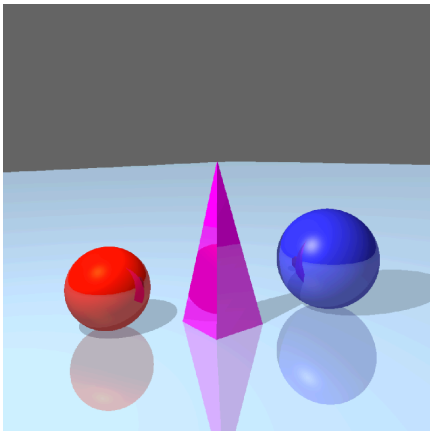
4.3.1 Light Source

A Point Light source is implemented in the project. The *Light* class has only a position $\mathbf{P_L}$ and an intensity **I**. The lighting effects are all produced through a single function in the *Illumination* class called `compute_shading()`. To produce good lighting effects, some information about the surface being shaded is required. This information is stored in the *IShape* class' `LastHitRecordField` field, which contains the ray position **e** and direction **d**, the intersection point $\mathbf{P_I}$, and the surface normal **N**.

In my scene, the light source is positioned around the blue sphere with an offset of *(-500, 500, -500)*. The scene without shading looks unrealistic and two-dimensional.



WITHOUT SHADING
CP: (0, -200, 500)

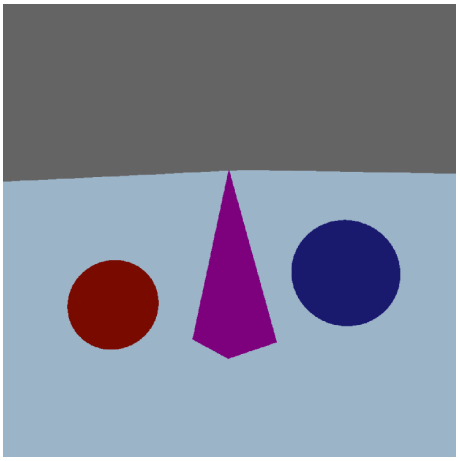


WITH SHADING
CP: (0, -200, 500)

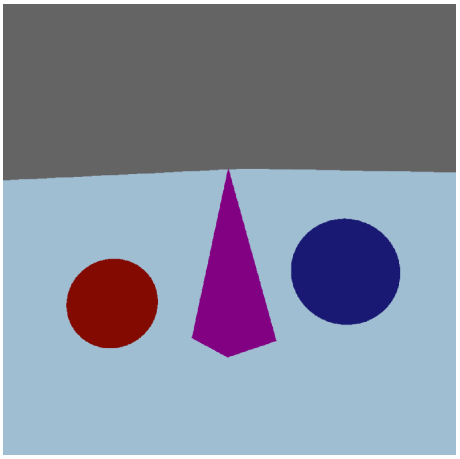
4.3.2 Ambient Shading

To simulate a realistic lighted environment, we add some extra light to every pixel in the scene. This is called ambient intensity $\mathbf{I_a}$, which is multiplied by the ambient coefficient of an object $\mathbf{k_a}$.

$$L_a = k_a * I_a$$



WITHOUT SHADING
CP: (0, -200, 500)



WITH AMBIENT SHADING
CP: (0, -200, 500)

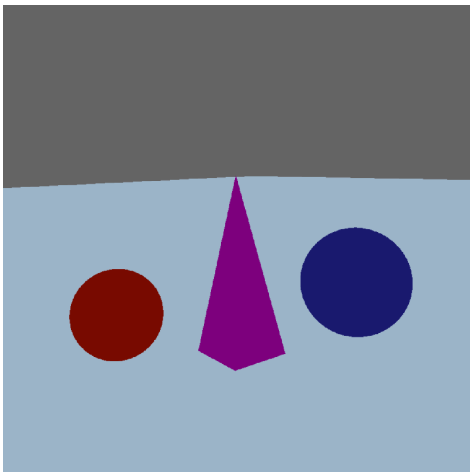
The difference in the images above is hard to see, but there if closely seen, the objects on the right image are brighter than on the left image.

4.3.3 Diffuse Shading

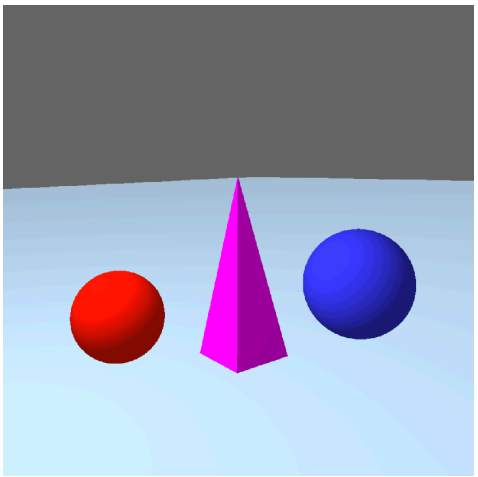
To add diffuse shading contribution $\mathbf{L_d}$ to the scene, the following computation must be done:

$$v_L = P_L - P_I$$
$$L_d = k_d * I * max(0, N.v_L)$$

The value $\mathbf{k_d}$ represents the diffuse coefficient, which controls how sensitive the material is to diffuse shading.



WITHOUT SHADING
CP: (0, -200, 500)



WITH DIFFUSE AND AMBIENT SHADING
CP: (0, -200, 500)

4.3.4 Specular Shading (Phong Model)

To add specular shading contribution $\mathbf{L_s}$ to the scene, we perform the following calculations:

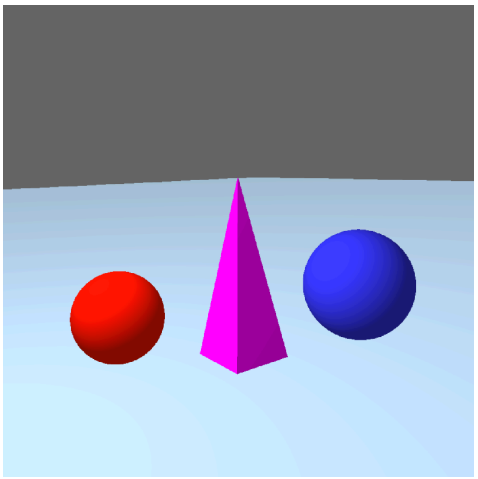
$$v_r = 2N(N.v_L) - v_L$$
$$v_e = e - P$$
$$L_s = k_s * I * max(0, v_r.v_e)^n$$

The value $\mathbf{k_s}$ is the specular coefficient, and **n** is the fall off distance or the phong exponent. We have to keep the value of **n** as powers of 2. In this project, the value used is *16*.

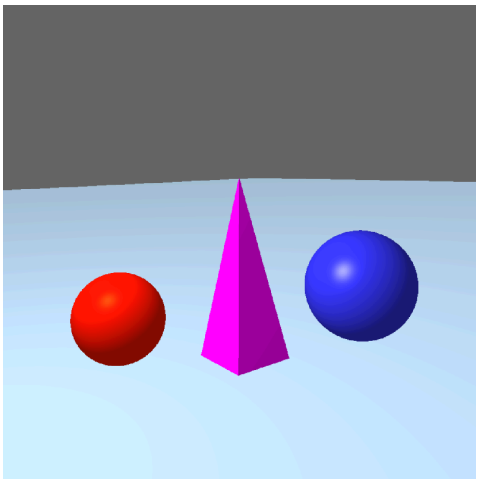
4.3.5 Phong Shading

Phong Reflection Model refers to the summation of light contribution from ambient, diffuse and specular shading. Through this summation, we get a final intensity **L** that is mixed with the color of the pixel using the `mix_light()` function.

$$L = L_a + L_d + L_s$$
$$C = \ mix\_light(C, L)$$



WITH DIFFUSE AND AMBIENT SHADING
CP: (0, -200, 500)



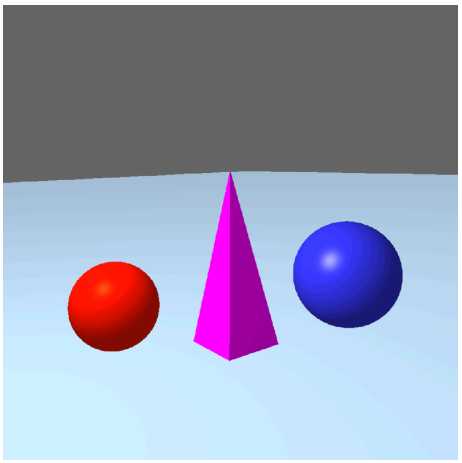
WITH PHONG SHADING
CP: (0, -200, 500)

4.3.5 Shadows

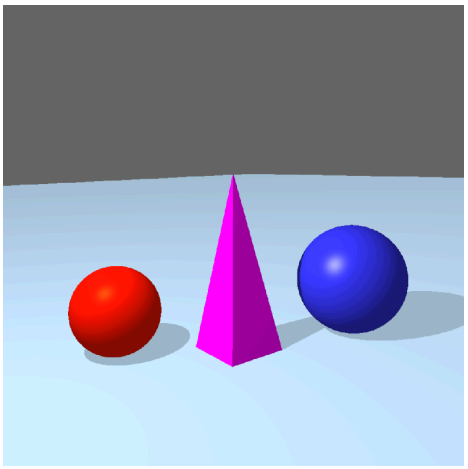
I have added shadows for all objects in the scene. To check if a point **P**, is in shadow, we have to generate a shadow ray from **P** to the light source position $\mathbf{P_L}$, and check if this ray

intersects any object. If there is an intersection, the point is in shadow [0]. The shadow ray can be computed as follows:

$$e = P$$
$$d = P_L - P$$



WITHOUT SHADOWS
CP: (0, -200, 500)



WITH SHADOWS
CP: (0, -200, 500)

4.3.6 Glazed Surfaces

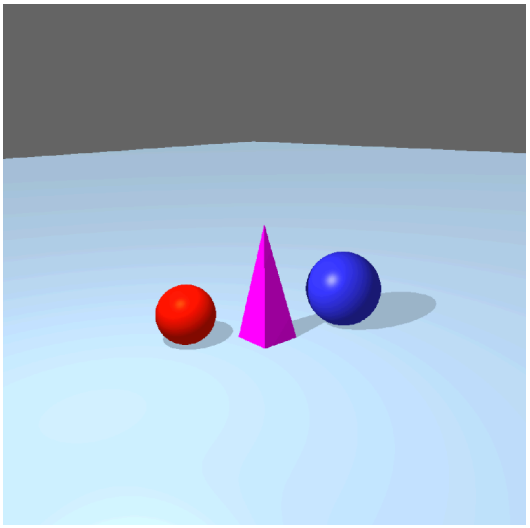
To add a glaze to any object, we have to generate another ray from a point **P**, which has the same direction as the direction of the reflection vector **r** of the ray projected from the camera with origin **e** and direction **d**. We can calculate this new ray with origin $\mathbf{e_r}$ and direction $\mathbf{d_r}$ by performing the following calculations:

$$e_r = P$$
$$r = d - 2N(d.N)$$
$$d_r = r$$

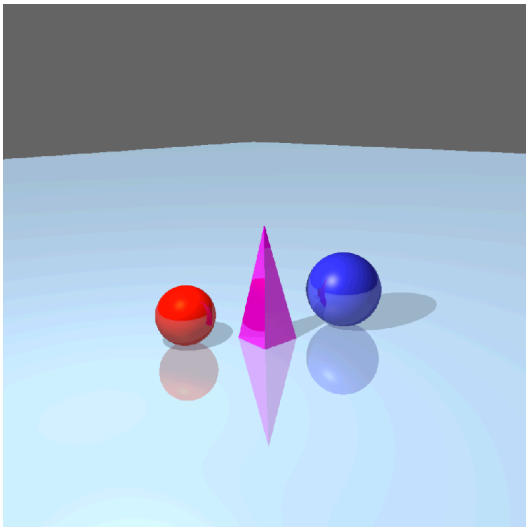
If this ray intersects any object in the scene, we combine the color $\mathbf{C_P}$ at point **P** and the color at the intersected object $\mathbf{C_I}$ using the `weighted_combine()` function. The weight values $\mathbf{W_P}$ and $\mathbf{W_I}$ I have used in my project are fixed for all objects, and they're done in such a way that the major contribution comes from the original object.

$$C_P = weighted\_combine(C_P, C_I, W_P : 0.75, W_I : 0.25)$$

Glaze is turned on for all objects by default.



WITHOUT GLAZE
CP: (0, 0, 0)



WITH GLAZE
CP: (0, 0, 0)

4.3.7 Coefficients of Objects

The lighting coefficients of the objects in the scene are given in the table below. The tetrahedron values are shared by all the triangles in it.

Object	k_a	k_d	k_s
Red Sphere	1.0	1.0	4.0
Blue Sphere	1.0	1.0	3.0
Triangle (Plane)	1.0	0.2	0.2
Tetrahedron	1.0	1.0	2.0

5 Animation

A small animation was created with the scene. The animation involves the camera backing up some distance and the two spheres revolving around the tetrahedron.

5.1 Setup

The red sphere and blue sphere are positioned at (-500, -425, 1200) and (300, -400, 1350) respectively. The camera is positioned at (0, 300, 200), looking at the tetrahedron. During the animation, the plane and the tetrahedron remain static. Only three objects are moved in the scene - the red sphere, the blue sphere and the camera. A smooth transition between the positions of objects in a sequence of frames is done through the use of linear interpolation, or

the `lerp()` function. Given two values **a** and **b**, we can find a linear point **x** between **a** and **b**, given a weight **t**, such that when **t** = 0, then **x** = **a** and when **t** = 1, then **x** = **b**.

$$x = lerp(a, b, t) = a + (b - a) * t$$

5.2 Movement

The camera simply moves backwards (zoom out), specified by an offset of (0, 200, -100). This is done by calculating the start position (current position before the animation starts) and the end position, and then interpolating between the values. The RayTracer stores two important fields related to animation - `fps` (which determines the number of frames rendered per second) and `duration_sec` (which determines how long the animation will be in second). Through this, we can find the total number of frames **F** that need to be rendered, as well as the interpolation weight **t**, for every frame **i**.

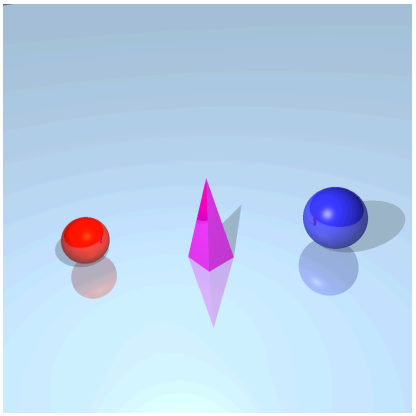
$$F = fps \times duration\_sec$$
$$t_i = \frac{i}{F}$$

Both spheres revolve around the tetrahedron, with each sphere going in opposite directions. As we want the spheres to go in a circular direction, we have to find a vector **v** from the tetrahedron to the circle (keeping the heights the same). Then, we can produce a cross product of **v** and a vector **t**, such that the cross product produces a vector **m** that is in the desired direction. Given this information and the fact that the movement will only happen in the XZ Plane, **t** can simply be a vector from the base of the tetrahedron to the top point of the tetrahedron. This will result in a vector **m** perpendicular to both **v** and **t**, which lies on the XZ plane. So, given a sphere's position $\mathbf{C_S}$ and the tetrahedron's top point $\mathbf{C_T}$, we can perform the given calculations:

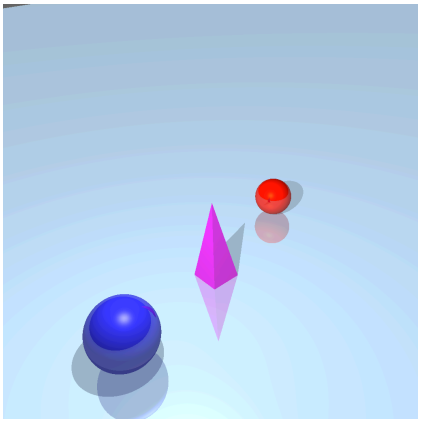
$$C_{S1} = Vec3(C_S.x, 0, C_S.z)$$
$$C_{T1} = Vec3(C_T.x, 0, C_T.z)$$
$$v = C_{S1} - C_{T1}$$
$$t = C_T - C_{T1}$$
$$m = v \times t$$

5.3 Results

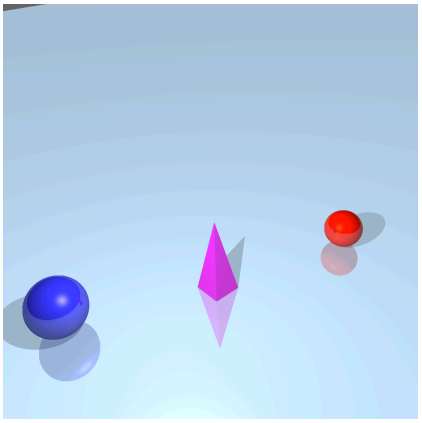
There are 2 videos provided for the same animation in the *animations* folder of the project, named *anim.mp4* and *anim_low_quality.mp4*. The original version has the highest quality but may not run on some media players. I was able to run the video on ***VLC Media Player***. Thus, a lower quality version is provided for quick viewing as well. The following results are produced with the animation.



Frame 0



Frame 150



Frame 300

6 Sources

[0] “Fundamentals of Computer Graphics, 5th Edition”, Steve Marschner, Peter Shirley.

[1] “Ray-Tracing: Rendering a Triangle”, https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution.html

[2] “Ray-Tracing: Generating Camera Rays”, https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays.html