Assignment 1: Raytracing

By

Duncan Stewart, Sai Ravi Teja Gangavarapu

UFID Duncan Stewart: 1644-7760

UFID Sai Ravi Teja Gangavarapu: 1050-4370

1 Scene

The scene we implemented contains that of three spheres, one tetrahedron, and one infinite plane. The three spheres are brought in by using our *Sphere* class, the tetrahedron is brought in by our *Triangle* class, and the plane is implemented by using the *Plane* class. The *Sphere* class, the *Triangle* class, and the *Plane* class all implement the *Obj* class, which then implements the struct class *HitResult.* The *Obj* class includes functions from the struct *HitResult, applyShading* and *hit_anything,* which detect if anything is hitting said object and how to apply the shading to the object the method is being used on.

We used the *camera.h* file to store the camera parameters, the light source and the objects that we previously talked about. With this file we would be able to create other objects to use the light source and camera with but however as previously mentioned we used the three spheres, the tetrahedron and the plane. The center and radius of the three spheres in the demo scene are given in the following table. The colors listed in the table below can be described more specifically using the RGB color vectors. We discuss these values in Section 3.

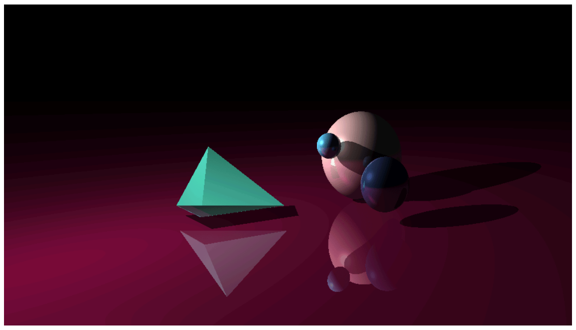
	radius	center	color
Sphere 1	0.8	(1.0, 0.0, -2.0)	green
Sphere 2	0.4	(2.0, 0.2, -0.5)	cyan
Sphere 3	0.2	(0.8, -0.3, -1.0)	dark blue

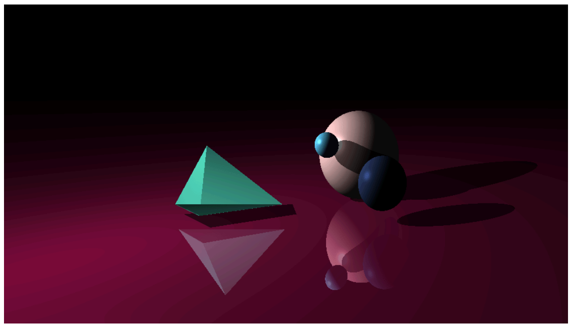
The vertices and the triangles we used for the demo scene are provided in the table below. The tetrahedron as seen in the demo scene is lime green.

Vertices	Triangles
v1 (-1, 0.6, -0.2)	Front (v1, v2, v3)
v2 (0, 0.6, -0.8)	Bottom (v1, v2, v4)
v3 (-2, 0.6, -0.8)	Left (v1, v3, v4)

v4 (-1, -0.35, -0.5)	Right (v2, v3, v4)

The infinite plane is located at (0, -1, 0), and it is colored magenta. Sphere and tetrahedron positions were chosen so that they are on top of the plane and reflect as needed. Below we have displayed the demo scene with the three spheres and tetrahedron. The details of the camera are specified in Section 2, and the details of the light source are specified in Section 3.





2 Camera

We created a *Perspective* and *Orthographic* camera that allows for the objects to be seen as how they are rather than from a parallel perspective and through a parallel perspective from the *Orthographic* camera. These cameras utilize the stored vectors within the *camera.h* file. We have certain parameters for the camera to follow such as the *cameraPosition, camera_up, focalLength,* and *look_at.* Each parameter is used in later vectors that would distinguish the camera's focal point and view plane. The *viewplane_height* is pre-established as a float of 6.0 and the *viewplane_width* comes from measurements dealing with the height of the view plane and the general height of the aspect ratio. The vectors *U, V,* and *W* are all established as the basic vectors for the camera with W starting as the *cameraposition* minus the *look_at* vector with vector *V* crossing *W*. The Vectors along the view plane are that of *viewplane_u* and *viewplane_v,* where *viewplane_u* deals with the width of the view plane and *viewplane_v* deals with the height. This method is helpful when getting a genuine look at what is being displayed rather than through an orthographic lens which would be more helpful with the design of objects. The *viewplane_upper_left* vector deals with finding the location of the uppermost left pixel at all times with the camera to know where the camera is looking at all times. The equation to get said pixel is given below:

*(cameraPosition - (W*focalLength)) - viewplane_u/2 - viewplane_v/2*

The last aspect the camera in our scene has is the vector of *initial_pixel* which utilizes the previous vectors we have shown such as *viewplane_upper_left, pixel_delta_v,* and *pixel_delta_u,* which will calculate the initial pixel the camera gives off and will be printed out to help establish where the camera is looking.

*viewplane_upper_left + (pixel_delta_u + pixel_delta_v) * 0.5*

2.1 Orthographic Camera

If the bool of *orthogonal* is set to true than all of the ray directions will be set to W*-1. Whereas the ray origin is set up through the equation shown below:

focalLength = 0

*rayOrigin = initial_pixel + (pixel_delta_u * x) + (pixel_delta_v * y)*

2.2 Perspective Camera

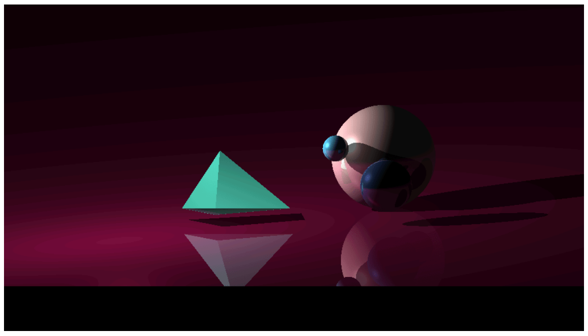
When orthogonal is set to false, perspective camera is in use which uses the *cameraPosition* variable as the *rayOrigin.* Through using previous variables such as *pixel_delta_u* and *pixel_delta_v,* we get a new variable labeled as *viewplane_pixel_loc* which eventually gets us the equation for the *rayDirection* variable as seen below:

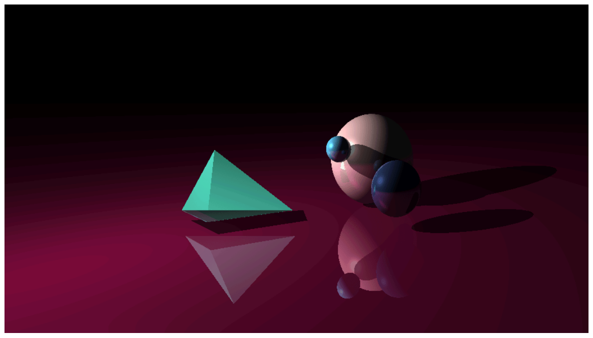
rayDirection = (viewplane_pixel_loc - cameraPosition).unit_vector()

2.3 Demo Scene

When the scene is created, *orthogonal* is set to false. In doing so, the camera that is first in use when the demo scene is created is that of the perspective camera. By pressing the "o" key, the user can toggle the bool to be set to true to go into the orthogonal camera. The default variables of each camera are given below. We have also given a view from both cameras within the demo scene.

	focalLength	*cameraPosition*	*camera_up*	*look_at*	*Width*
orthographic	0.0	(4.0, -1.0, 5.0)	(0.0, -1.0, 0.0)	(0.0, 0.0, -1.0)	400
perspective	(cameraPosition - look_at).length()	(4.0, -1.0, 5.0)	(0.0, -1.0, 0.0)	(0.0, 0.0, -1.0)	400


orthogonal


perspective

3 Shading

We have a specific file dedicated to shading called *shader.h,* where it will calculate the desired shaders we need for this assignment.The specific color of each object is given in the *main.cpp* file as the variable *C.* Additionally, the file holds coefficients for the different types of shading used such as *ambientCoefficient (ka), diffuseCoefficient (kd),* and *specularCoefficient (ks)*.

The shader file also holds the shininess coefficient that would be used alongside the other coefficients, *shininess (p).* Lastly, back in the *objects.h* file we have a bool denoting if the object will be glazed or not. In the table below, I label all of the necessary properties for shading:

	c	*ks*	*kd*	k_s	*p*	*glazed*
Sphere 1	(183, 145, 145)	0.6	12	9	65	false
Sphere 2	(30, 40, 70)	0.6	12	9	65	false
Sphere 3	(70, 150, 184)	0.6	12	9	65	false
Tetrahedron	(255, 255, 255)	0.6	12	9	65	false
Plane	(255, 255, 255)	0.6	12	9	65	true

The values of the colors are given in RGB values from 0 to 255, however in code it is represented as decimals such as 184 being 1.84.

Shading isn't entirely done just through the material or the color of the object but also through the lighting that is set throughout the demo scene. For this project, we implemented a sunlight object in our *objects.h* file. The properties of sunlight are the *intensity* and the *position.* The intensity and direction of the light source in the demo scene are given in the following table:

	intensity	*position*
sunlight	12	(-2, -2, 1)

To calculate the shading required in all the objects, we used our function *calculateShading* which is provided with the intersection point, the object's color, and light intensity. So once it calculates all the shader intensities with the equations shown in sections 3.1-3.3, the function will then return the following equation back to *main.cpp* to change the objects' surfaces:

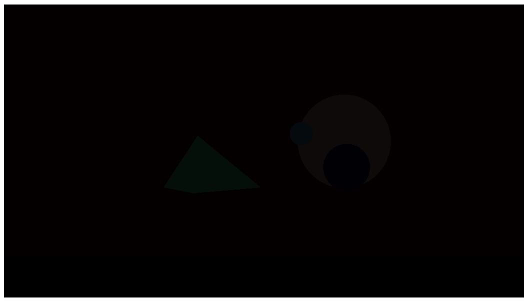
ambientIntensity + diffuseIntensity + specularIntensity

3.1 Ambient

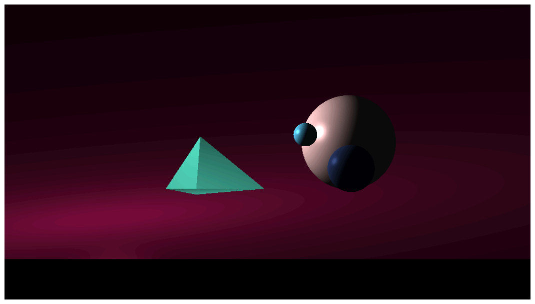
The ambient intensity is calculated simply by:

*ambientIntensity = ambientCoefficient * lightIntensity*

By just bringing the coefficient set earlier with the now established light intensity from the sun, we get the ambient intensity. Which will then be added with the other intensities to get our shading for the demo scene. Below, we have three images showcasing ambient shading, with the first image showing what it would look like with just the ambient shading and lastly what the scene looks like with them all enabled.



ambient only



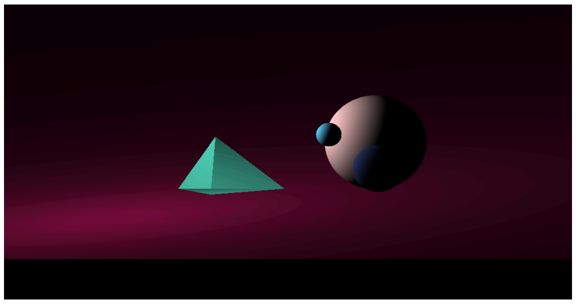
diffuse, specular, and ambient

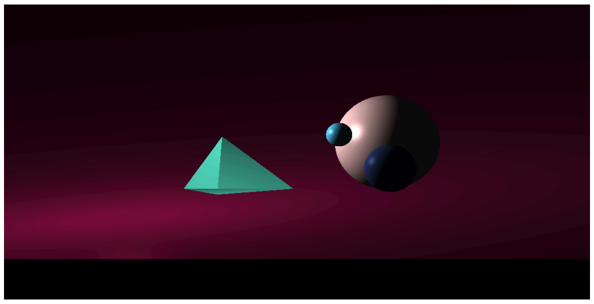
3.2 Diffuse

The next type of shading has a bit more of a complicated way to get the intensity needed for the scene at hand. The equation is provided below:

*diffuseIntensity = diffuseCoefficient * lightIntensity * std::max(0.0, normal.dot(VL))*

In the images below, we showcase how the scene would like with and without the use of diffuse shading. In the first image we have what it would look like with only diffuse active. In the last image we showcase what the scene would look like with all three shaders attached to the scene.


diffuse only

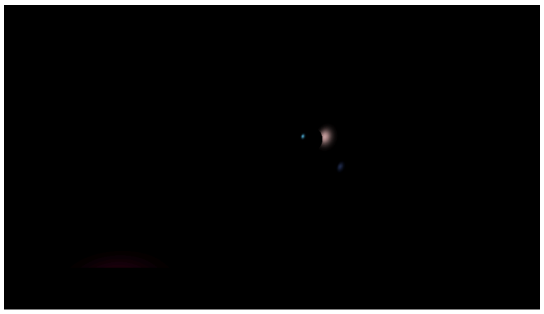

diffuse, specular, and ambient

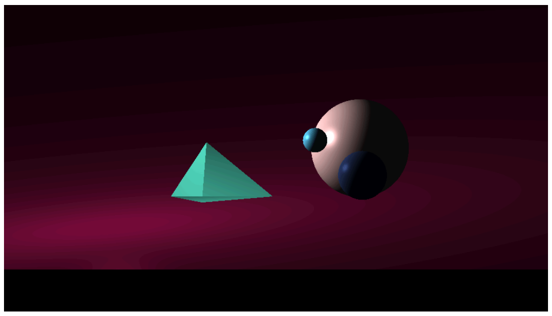
3.3 Specular

For specular shading, we would need the same light intensity variable we have used in previous shaders. However, with this shader we now need to use the variable *shininess.* This will help with most of the reflective colors from the sunlight and other objects. The equation below will be how we get the intensity required for our scene:

*specularIntensity = specularCoefficient * lightIntensity * std::pow(std::max(0.0, normal.dot(VH)), shininess)*

We have provided an additional three photos to showcase the effectiveness of specular shading with the leftmost image first showing what the scene would look like with just specular active. The last image is showing off what the scene looks like with all three types of shaders active.


specular only


diffuse, specular, and ambient

4 Glazed Surfaces

Certain materials will reflect back the image of the objects around them almost giving the same effect as that of a mirror, this is called glaze. In our project, the function *applyGlaze* is in the *objects.h* where the object's surfaces will apply the glaze on top of them in addition to the previous shading and colors on the objects. There are two variables that are immediately needed with glazing, the intersection point and the direction of the reflection. We provided this with the equations below:

intersectionPoint = ray.pointAt(t)

*reflected_dir = ray.direction - normal * (ray.direction.dot(normal)) * 2*

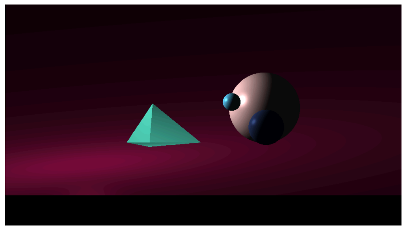
The intersection point needs the ray that is pointing at the given location for that object, whereas the reflected direction needs that same rays direction and the normal vector with the dot product to achieve this reflection. Once it is done, the reflected ray object will use the reflected direction and the intersection point to create the new reflected ray that will be needed to achieve the glaze effect.Once this has been achieved we will use the color of the object that is closest to the current object we are working on with the equation below:

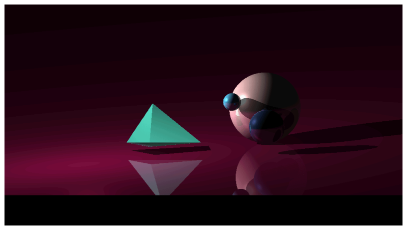
Color returnColor = closest_object->objColor();

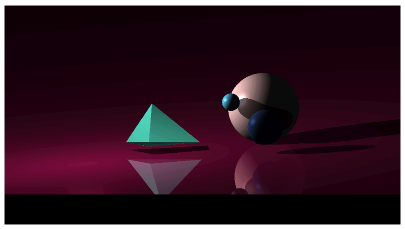
Lastly with this new returned color that we have achieved with the previous equation we will use one of our functions *hit_anything_for_shadows* to check with the reflected ray we got earlier. If there are shadows with this new ray then we will have to get a new glazed color for the objects that we are setting it on for. We get this new glazed color with the provided equation below:

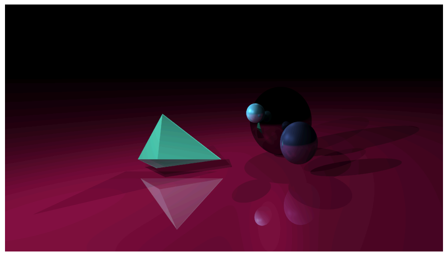
*glazeColor = (castRay(reflected_ray, *this, lightsource.position, true) * 0.4).clamp(0, 255)*

Once we have done this we now have returned our new glazed color and if in the case that there are no shadows with the reflected ray we just return color (0, 0, 0). In our demo scene the glazed surface is the plane to reflect the objects we've provided, however, below we have rendered what it would look like with no glaze with the image on the far left, with just all objects glazed in the middle, and lastly, having just the plane glazed.


no glaze

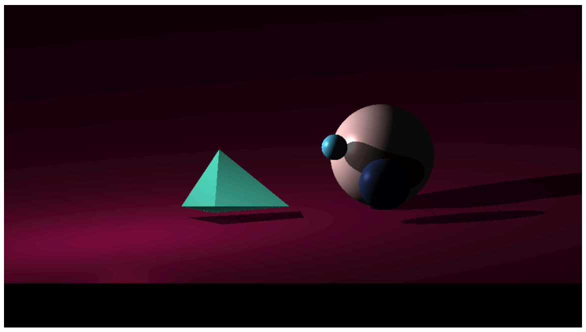

all glazed

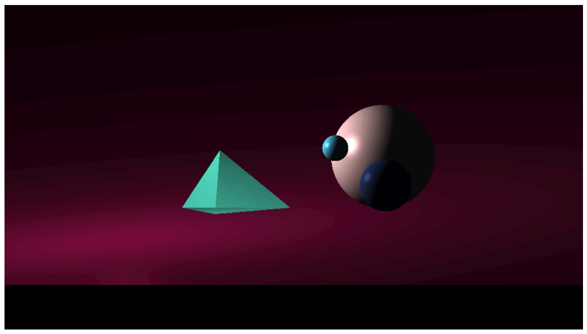

glazed plane


all glazed with a no color sphere

5 Shadows

We added shadows to our project through multiple parts of our file such as *object.h.* The shadows first start out with the *hit_anthing_for_shadows* method that is used alongside the variable *reverse_lightray* which has been made previously by the intersection point of the different objects we have in our scene. Once we have this variable it is sent over to the method and checked to see if the objects do in fact have anything hitting them, in this particular case the plane. Once it figures this out it will send back a *boolean* variable which will be used to determine once and for all if the method is true or false. If returned true, the color will then be changed as a new shaded color and give off shadows from underneath the objects present in the scene.


with shadows


without shadows

6 Movies

We created two movies for the project to showcase the demo scene we have created alongside some of the additional aspects that were added specifically for the movies such as extra lights. The first movie is a static camera to show a simple animation where the second movie is to make unique camera movements and extra lights to showcase the demo scene. We have also included each movie as a MP4 file in the zip file.

6.1 Movie 1 (clickable link)

The first movie we created was a much simpler movie that just showcased moving objects and lights while the camera stayed still to showcase all of the aspects we have previously talked about in this report. One such thing is the movement cycle that the circle is moving on and that can be provided by the two equations below:

*sphere_x = sphere_motion_radius * cos(angle);*
*sphere_z = sphere_motion_radius * sin(angle);*

Overall, this first movie was the basis for the second one which was more intricate as the camera began to move around causing for the shadows and lights to be showcased in a different light then the previous.

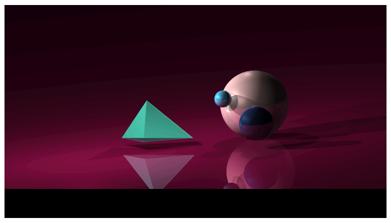
6.2 Movie 2 (clickable link)

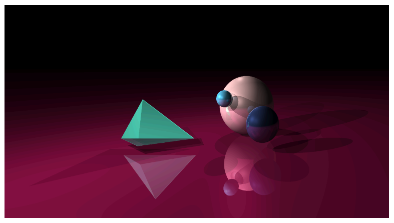
In the second movie ("movie2.mp4"), we moved the camera through the objects in a circular motion while some of the objects were also moving, causing this trippy movement. The circular motion of the camera is created by this equation:

angle += 0.015

Some of the advanced features that we added specifically for this movie was that of multiple lights, one that was moving during the animation with an intensity of 8 and two more lights in a constant position with an intensity of 4 for both. One additional feature was that of shadows, as we previously talked about we did add shadows to the base scene however with this animation the shadows were shown on multiple surfaces and not just the plane to give an effect of the surrounding area being more like a box then just a flat plane.

To showcase the additional lights that were shown throughout the movie, we provided more pictures to showcase these elements below:





no lights

multiple lights

multiple lights