**Project One**



**Description:** Your ray tracing work will conclude with this project. The project will require you to implement shadow feelers to create shadows, trace reflection vectors to create mirror-like inter-object reflections, and add additional types of shapes to a scene. Finally you will do additional work to implement more advanced rendering techniques. Your starting point for this lab is your completed lab Five.

Your grade on this project will depend not only on successful implementation of additional aspects of ray tracing, but also on quality of the scene that is rendered by your project.

Your project must include the capability to enable and disable individual light sources using the ‘a,’ ‘d,’ ‘p,’ and ‘s’ keys as described in Lab 5.

**Quadric Surfaces**

Download the QuadricSurface [declaration](https://drive.google.com/file/d/1XtFEd6O9AnQ_Mjq9d7oYl3VpMVfpumU0/view?usp=sharing) and [definition](https://drive.google.com/file/d/1FTEQHWQtgflbJPgQr6-cH7Dy2TNBQ5Nk/view?usp=sharing) and add the class to your project.

Implement two specialized sub-classes of the QuadricSurface class. One should be the Ellipsoid. The other should be a limited version of an infinite shape. For instance, if you create a Cylinder or Hyperboloid class, the rendered object should have a specified radius or radii and a limited length. Cones, Elliptical Paraboloids, and Hyperboloids should go in only on direction and have a limited height. It must be possible to render them in any position within the scene.

In your sub-class findIntersect method, use the findIntersect of the QuadricSurface class to do the heavy lifting of intersection testing. In the sub-class findIntersect method, check the point of intersection to see if it is in the more limited shape you are creating. If it is not in the limited shape, cast a second ray going from the “non-intersection point to see if that ray hit the other side of the limited shape. If it does return this point of intersection for rendering. Otherwise, return a hit record that indicates that the ray did not intersect the object.

Both sides of the surface should be rendered with correct lighting. The normal for an intersection should always point away from the direction of the ray being traced.

**Shadows**

Implement shadows by checking shadow feelers for intersections with the surfaces in the scene. Cast a shadow feeler whenever a traced ray intersects an object. For directional light sources, an intersection with a surface anywhere along a shadow feeler that points in the direction of the light vector for the source should result in there being no contribution from that light source. For positional light sources, an intersection with a surface in between the origin of the shadow feeler and the position of the light source should result in there being no contribution from that light source.

To make shadow determinations, utilize the getLightVector and getLightDistance methods of the LightSource struct and its sub-classes. Shadow feelers should be appropriately generated and tested with in the RayTracer class.

**Inter-Object Reflections**

Add mirror-like inter object reflections by tracing reflection rays to the closest surface intersection for each view ray. Once generated, the reflection ray can be traced in exactly the same way as the viewing rays. The best way to accomplish this is by calling the traceRay method recursively and adding what it returns to the total color for the pixel. To do this, it is necessary to keep the recursion from being infinite. This can be accomplished by using the recursionLevel parameter that is passed to the traceRay method. This parameter can be decremented prior to each recursive call. The recursion would stop when the value is less than or equal to zero.

color RayTracer::traceRay(const Ray& ray, int recursionLevel){...

**Taking it Further**

Implement at least three of the following extensions for your ray tracing program. You are of course encouraged to implement more if you like.

Do at least **one** of the following:

**Reflect the “Sky”**

Reflection rays often do not intersect any surfaces in the scene. The illumination for these non-intersecting rays can simply be “no color” or if can be a scaled down default color. The later will cause shiny objects to appear to reflect the default color as though it is the sky. Modify your program to create this effect.

**Day and Night**

Modify the program so that pressing the ‘m’ and ‘n’ causes the scene to be rendered in day and night modes respectively. You can simply “dim” the lights or change the lighting entirely to create the two different effects.

**Multiple Views**

Modify the program so that pressing the ‘f,’ ‘g,’ and ‘h’ keys causes the scene to be rendered from three significantly different opposite viewpoints.

Do at least **one** of the following:

**Attenuation**

Use constant, linear, and quadratic attenuation constants to attenuate the contributions of shading based on their distance from the point of intersection.

Use attenuation to reduce the contribution of reflection vectors based on the distance to the intersection between the reflection ray and the object it hits.

To test this, you may want to position either a light source at a distance and render the scene with and without attenuation.

**“Capped” Objects**

The cones, cylinders, etc. that implemented above do not have “ends.” That is, the cones have are more like a container for a snow cone and the cylinders are more like sleeves with no ends. Create a capped cylinder, cone, etc. by not only checking for intersection the base object but also checking for intersection with a disk(s) on the end(s) of the object.

Checking for the disk intersection is similar to the Convex polygon above only easier. You just have to see if you have hit the plane in which the circle lies and then see if the distance between the point of intersection and the center of the circle is less than the radius of the circle.

**Simple Polygon Surfaces**

Simple polygons are described by three or more vertices in same plane, are convex, and have no sides which cross one another. The front face of the polygon is normally defined as the face on which the vertices appear in counter clockwise order. The surface normal for the polygon should point out of the front face.

Implement a subclass of the Plane class that represents a simple polygon. Name the class something like SimplePolygon or ConvexPolygon. DO NOT NAME THE CLASS POLYGON, THIS WILL CAUSE A NAME COLLISION WITH SOMETHING IN ONE OF THE LIBRARIES WE ARE USING. The constructor among other things should have an input parameter that is a vector of dvec3 objects which specify the vertex locations for the polygon in counter-clockwise order.

The sub-class should use the findIntersect method of Plane class to determine if the plane in which the polygon lies is intersected by the ray. It can then use the method described in the notes to determine if the intersection is inside the polygon described by the vertices passed to the constructor. You should be able to account for intersections with both the front and back side of the polygon. Both sides of the polygon should be rendered with correct lighting. Thus, you must make sure that the surface normal placed in the hit record is pointing in the correct direction.

Do at least **one** of the following:

**Texture Mapping**

Read color information out of an image to determine the ambient and diffuse colors of an object.



Basic texture mapping uses a function to determine the diffuse color at the point of intersection with a surface. To make it work, you not only have to calculate XYZ coordinates that describe the position of the intersection, you also need to calculate UV coordinates using a planar, cylindrical, or spherical mapping function. When performing lighting calculations, you use the UV coordinates to pull a texel color out of an image. This color is used in the ambient and diffuse lighting calculations.

You can download a new version of the material struct that supports texture mapping [here](https://drive.google.com/file/d/1Yl5EYAejx66nKLZY1Oj6vtaPtrrIq7kK/view?usp=sharing).

You can download a simple class for loading images and returning colors based on UV coordinates [here](https://drive.google.com/file/d/1P-P3_4F6nMDGU9sqwVf2vB4T6PcBD7KO/view?usp=sharing). Note this class requires images to be in a ppm format which is relatively rare. The [GIMP (GNU IMAGE MANIPULATION PROGRAM)](https://www.gimp.org/) is able to load photos in just about any format and export them in the PPM format.

You can download mapping functions to calculate UV coordinates [here](https://drive.google.com/file/d/1VOdE0O1C80GyJIw0Yv0Gtf4okSaPxJxg/view?usp=sharing). Use the Object coordinates of the point of intersection instead of the World coordinates when calling the mapping functions.

hitRecord.uv = calcSphericalTextCoord(hitRecord.interceptPoint - center, 1, 1, Y\_AXIS);

**Transparency and Refraction**

Create transparent object(s) that simulate refraction.



Transparent objects allow refracted light to pass through them. Refraction causes light rays to bend based on differences in the speed of light in the different mediums in which the light is passing through. Refraction rays can be generated in much the same way as reflection rays by calling the glm::refract function. Besides the incidence vector and the surface normal, the refract function requires the ratio of the index of refraction (IOR) of the material being entered to the IOR of the material that the light is leaving.

dvec3 refraction = glm::refract(ray.direct, closestHit.surfaceNormal, etat / etai);

Transparent objects both reflect and refract light. Fresnel equations can be used to determine the ratio of reflected light to refracted light. Fresnel equations are implemented in the function below.

/\*\*

\* @fn static double fresnel(const dvec3& i, const dvec3& n, const double& etai, const double& etat)

\*

\* @brief Compute Fresnel equation

\*

\* @param i is the incident view direction.

\* @param n is the normal at the intersection point.

\* @param etai is the refractive index of the material the light is leaving.

\* @param etat is the refractive index of the material the light is entering.

\*

\* @returns kr is the percentage of light reflected As a consequence of the conservation of

\* energy, transmittance is given by: kt = 1 - kr

\*

\*/

static double fresnel(const dvec3& i, const dvec3& n, const double& etai, const double& etat)

{

// Percentage of light that is reflected

// Percentage of light that is refracted is equal to 1-kr

double kr;

// Calculate the dot product of the incidence vector and the normal

// vector of the surface the the light is entering

double cosi = glm::clamp(-1.0, 1.0, glm::dot(i, n));

// Compute the sine of the angle of refraction using Snell's law

double sint = etai / etat \* sqrt(glm::max(0.0, 1.0 - cosi \* cosi));

// Check if angle of incidence exceeds critical angle

if (sint >= 1.0 ) {

kr = 1.0; // Total internal reflection

}

else {

// Calculate the percentage of light that will be reflected

double cost = sqrt(glm::max(0.0, 1.0 - sint \* sint));

cosi = fabs(cosi);

// S polarized light (parallel)

double Rs = ((etat \* cosi) - (etai \* cost)) / ((etai \* cost) + (etat \* cosi));

// P polarized light (perpendicular)

double Rp = ((etai \* cosi) - (etat \* cost)) / ((etat \* cost) + (etai \* cosi));

kr = (Rs \* Rs + Rp \* Rp) / 2.0;

}

return kr;

} // end fresnel

To make transparency work, you will have to keep track of when the ray is entering an a surface. This can be done using a property that the HitRecord already has. To determine if the ray is entering or leaving a surface, you can compare the surface normal to the ray direction whenever a “hit” occurs.

/\*\* @briefIndicates whether if the ray is entering an enclosed object or leaving it. \*/

RAY\_STATUS rayStatus = ENTERING;

You will also have to add an IOR to the material properties of the surfaces.

/\*\* @brief index of refraction for the material \*/

double dielecticRefractionIndex = 1.0;

/\*\* @brief True if is dielectric and BOTH transmits and reflects light,

false if not \*/

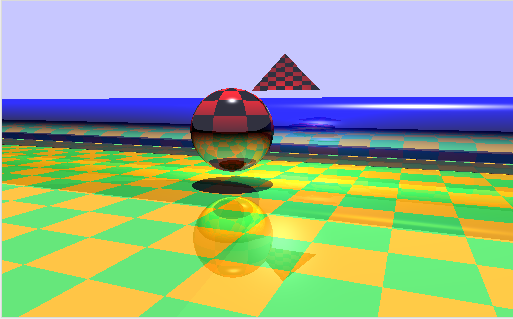
bool isDielectric = false;

You might want to just code up the plane as though the light was always leaving since it will not be possible for the ray to then hit the “backside” of the plane.

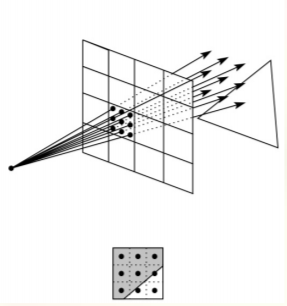
To simplify, when the light ray is entering an object, you can assume the IOR for the material the ray is leaving is 1.0. Likewise, when the ray is leaving a surface you can make the IOR of the material being entered 1.0.

**Antialiasing**

Modify the program so that pressing the ‘z’ key toggles on and off antialiasing.



Because of the discrete nature of raster image representation, rendered images will include aliasing artifacts. These artifacts create a jagged or stair stepped in objects that should appear smooth. In ray tracing applications, anti-aliasing can be performed by simply tracing multiple rays per pixel. The simplest approach is to subdivide each pixel into the grid, cast one rays per grid square, and then set the rendered pixel intensity to the average of color associated with the rays.



If you implement this functionality, don’t go crazy with the number of rays. Remember, we have no hardware acceleration. Nine rays per-pixel as depicted above should have an impact. Also realize that there are much more complex stochastic methods for generating the rays.

**Create a Scene**

Create a scene composed of the spheres, planes, polygons (if you implemented them), and the quadric surfaces you created. Create a full-screen capture of the best result you achieve when rendering your scene.

**Turn it in**

Submission instructions for this project are similar to the lab instructions with the exception of the screen capture.

1. When submitting your project include the screen capture of the scene that is created by your project.
2. Copy the folder containing your solution to the desktop.
3. Change the name of the folder to CSE287ProjectOne followed by your unique identifier. For instance “CSE287ProjectOneBachmaer.”
4. Open the solution. Make sure it still runs.
5. Clean the solution by clicking on clean.bat. (The will delete all the intermediate temporary files that are part of your project and reduce the size of your submission.)
6. Zip up the solution folder using the standard windows compression tool. (No 7zips, rars, etc.)
7. Submit your zip archive of the solution through canvas.