Computing Science CMPT 361 | **Fall 2019**
Instructor: Tom Shermer | **Simon Fraser University**

Assignment #3

Due date: November 27th at 11:59 pm.

Ray Tracing

You will write a basic ray tracer. The entire assignment can be completed in Euclidean space; there is no need for homogeneous coordinates. There are 25 points available.

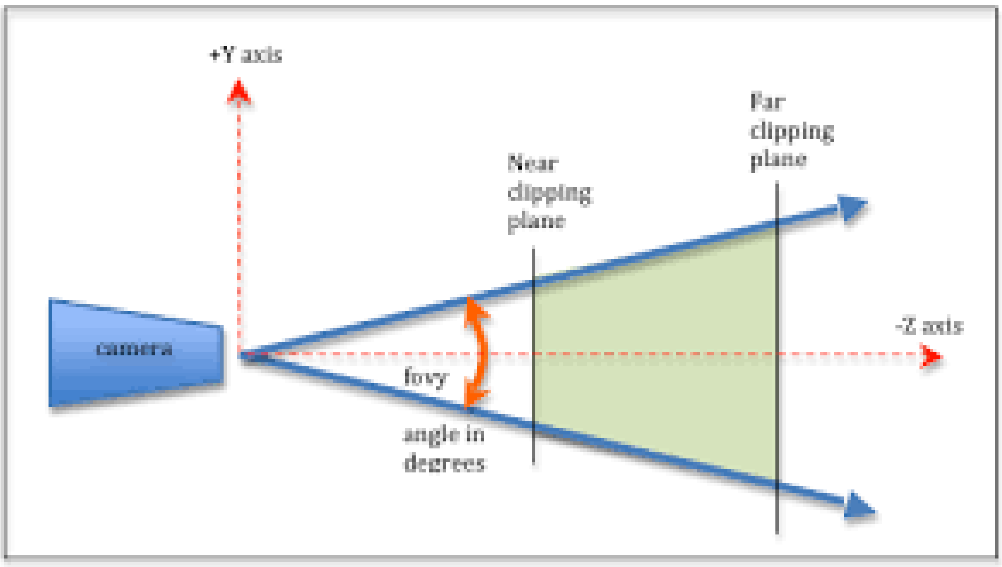
(a) [1 point] Window setup

Use OpenGL to set up a window with a 400 x 400 pixel drawing window. Use a symbolic constant rather than 400 so that you can change resolution by changing the constant.

(b) [2 points] Viewing

The eyepoint is the origin, and the view is down the negative z-axis with the y-axis pointing up. The coordinate system is right-handed.

Have a constant or a procedure call that defines the Field of View in the Y-direction (fovy), in degrees. The field of view is the angle between the top of the view volume and the bottom of the view volume. It is illustrated in the following image, but note that we will have no near or far clipping planes. Note that since we have a square window (400 x 400) the field of view in the x-direction (fovx) is equal to fovy.



(image from learnwebgl.brown37.net)

Assume that the virtual screen is on the plane at $z = -1$.

Design an object called **Ray** to hold a ray, which consists of a start point p_0 and a vector v and represents the ray $p(t) = p_0 + tv$ for $t \geq 0$.

In a routine named **render,** write a loop nest that will generate the initial rays for each of the pixels. These initial rays start at the origin and have a vector v = (v_x, v_y, -1). Your job here is to figure out the correct v_x's and v_y's given fovy. At this point, the body of the loop nest should just create a ray object.

(c) [2 points] Primitives

You will have two types of objects that you will be making images of: spheres and planes.

Define an object for each of these primitives, e.g. a **Sphere** and a **Plane**. Design ways of creating these objects so that they are put on a globally-accessible list (array, vector, etc.) of objects. You may either keep one such list or two (which would be one for Spheres and one for Planes).

When creating an object, specify its geometry and its material/surface properties. For example, you might have:

```
new Sphere(1.0, 2.0, -3.0, 1.25, 0.1, 0.8, 0.8, 0.9, 0.9, 0.9, 32, 0.5, 0.0, 1.5)
```

which is quite hard to figure out. It's better if you have objects that group the parameters, such as a **Point** and a **Color**:

```
new Sphere(Point(1.0, 2.0, -3.0), 1.25, Color(0.1, 0.8, 0.8),
                                        Color(0.9, 0.9, 0.9), 32,
                                        0.5, 0.0, 1.5)
```

This is still a long parameter list, but it is meant to represent:

Center of sphere: (1.0, 2.0, -3.0)

radius of sphere: 1.25

k_d: Color(0.1, 0.8, 0.8)

k_s: Color(0.9, 0.9, 0.9)

q: 32

k_r: 0.5

k_t: 0.0 (no refraction)

index of refraction: 1.5

One can perhaps simplify the parameters farther by defining an object Material consisting of all the material properties.

```
Material* aqua = new Material(Color(0.1, 0.8,0.8), Color(0.9, 0.9, 0.9), 32, 0.5, 0.0, 1.5)

…

new Sphere(Point(1.0, 2.0, -3.0), 1.25, aqua);
```

The object Plane should be similar, with the first four parameters denoting A, B, C, and D of the plane equation Ax + By + Cz + D = 0:

```
new Plane(0.0, 1.0,  0.0,  1.0 , aqua);
```

This is the plane y = -1 given the 'aqua' material properties.

(d) [1 point] Lights

The lights you will be using are simple local point light sources with no falloff.

Define an object **Light** and a way of creating them so that they are put on a globally-accessible list (array, vector, etc.) of lights. Each light should have a location and a color intensity:

```
new Light(Point(-100.0, -100.0, -20.0), Color(2.5, 2.5, 2.0))
```

Note that the "Color" here is color intensity and can contain values greater than 1.0; normal colors (k_d and k_s) have channels that range between 0.0 and 1.0.

(e) [1 point] Scene setup and command-line argument

Your program must be capable of displaying one of four different scenes depending on a command-line argument (which will be a number, 1 to 4). Each scene should set up lights, primitives, camera (fovy), and also set a depth limit for the ray tree, an ambient light intensity, and a background light intensity. You should have one subroutine for each scene to display. Here's **roughly** what a scene subroutine should look like, if you use the Material object:

```
void scene2() {
        Material* aqua = new Material( … );
        Material* greenGlass = new Material(…);
        Material* chrome = new Material(…);

        fovy(40);
        rayTreeDepth(4);

        ambient(new Color(…);
        background(new Color(…);
        new Light(…);
        new Light(…);

        new Sphere(…);
        new Sphere(…);
        new Plane(…);

        render();
}
```

Don't slavishly copy that; you may have other syntax for creating your objects, or differently-named subroutines, etc.

(f) [3 points] Ray-primitive intersections

As member functions of the primitive objects (Plane and Sphere), implement a routine which will intersect a ray with the object. It should take the ray as a parameter and return the smallest positive t value at which the ray intersects the primitive. If there is no such t value, it returns a negative number. (This is an overloading of the return value but in this case speed is important so we'll forgive ourselves.) These computations have been covered in lecture.

(g) [7 points] Ray tracing

Implement a function **trace** that takes a ray and a depth of tracing as parameters, and returns a color intensity.

If the depth of tracing is 0, then **trace** should return the background color intensity.

If the depth of tracing is greater than 0, then **trace** should intersect the ray with all primitives in the scene and determine which one has the smallest positive **t** where it intersects. This is the object the ray hits. If the ray hits no object, then trace should return the background color intensity.

If the ray hits an object, then let **p(t)** be the point where it hits. At this point, apply the lighting formula

$$I = I_a k_d + I_r k_r + I_t k_t + \sum_{i=1}^{m} S_i I_i (k_d N \cdot L_i + k_s (R_i \cdot V)^q)$$

I_r and I_t are obtained by recursively tracing the reflected and refracted rays, respectively. Be sure to decrement the depth-of-tracing parameter by 1 when you make the recursive call. S_I should be 0 or 1 and is obtained by a ray <u>**cast**</u> from p(t) in the direction of light source i. The view vector V is the reverse direction of the incoming ray that hit the surface at p(t). You will need to find the normal vector to your primitive at p(t), and also L_i and R_i. You'll also have to find the directions of the reflected and refracted rays; the refracted one is more difficult but that is a problem I leave for you to solve. Do not generate reflected or refracted rays if k_r or k_t is zero.

Modify your **render** subroutine so that it traces the rays that it generates. When it gets a color intensity back from **trace**, it should first convert that color so that all channels lie between 0.0 and 1.0 (because the intensity you get may be out of range). You choose how to do this. Then it should write that color to the appropriate pixel on the OpenGL display. See the programs in section 23-1 of your text if you are unsure how to do this…it's done with code like:

```
glColor3f(r, ,g, b);
glBegin(GL_POINTS);
   glVertex2f(x, y);
glEnd();
```

(h) [8 points; 2 points each] Four scenes

Each scene should contain at least three spheres and one plane.

Scene 1 should convincingly demonstrate that you have basic Phong lighting controls implemented. I want to see diffuse and specular effects. Be sure to include surfaces with different values of q, the specular exponent. Be sure the image clearly shows an effect from the changing q.

Scene 2 should convincingly demonstrate that you have reflection and shadowing working.

Scene 3 should convincingly demonstrate that you have refraction working.

Scene 4 is your choice. Design a nice scene and impress us with what your ray-tracer will do.

Extras

No extra credit is available, but you can implement extra functionality if you would like. For instance, have a function to call from a scene function that will set the resolution of the render (rather than 400 x 400). Or you could implement extra primitives, like polygons, quadrics, or CSG. Or you could implement light attenuation. Or texture-map the plane like a checkerboard. On the structural side, you could implement bounding volume hierarchies. These are just some ideas in case you'd like to expand this project.

Note that you can include these extras in your scenes, particularly in scene 4.

<u>Submission</u>:

- All source code,
- a ***Makefile*** to make the executable called ***rayT*** (the make command should be simply "make"),
- four images called S1.jpg, S2.jpg, S3.jpg, S4.jpg that show the images that result from running your program with arguments 1, 2, 3, and 4, respectively,
- and a ***README*** file that documents any steps not completed, additional features, and any extra instructions for your TA.