CS130 - LAB 2 - Introduction to Ray Tracing (Assignment 1) Section:____

Pre-Lab - Part 1: Ray Sphere Intersection

Consider a ray with endpoint e and (unitary) direction u. Consider a sphere with center c and radius r. The equations for the ray and the sphere are:

- **ray: R(t) = e + ut**
- **sphere: S(x) = (x - c) • (x - c) = r^2**

Any point r on the ray can be found using the real value t >= 0.
Any point x that satisfies the sphere equation is on the sphere.

Follow the steps below and calculate t's such that the point in the ray intersects the plane.

<u>Step 1:</u> Combine the ray and sphere equations to find t that solves both equations by expanding S(R(t)) = 0.

<u>Step 2:</u> Rearrange the terms and group the terms with t^2, t and, write the equation at the form

$$at^2 + bt + c = 0$$

then find a, b and c.

Hint 1) group terms that does not have t, before distributing the dot product.
Hint 2) $u \cdot u = 1$***.***

$a =$

$b =$

$c =$

<u>Step 3:</u> The discriminant, $\triangle$, of the quadratic equation can be found by $\triangle = b^2 - 4ac$
Write $\triangle$ in terms of sphere and ray parameters by substituting a,b and c.

$\triangle$=

Name:_____ **SID:** _____

Step 4 : Finding the t's for the intersections

Case 1: if $\Delta < 0$: then there are no intersections

Case 2: If $\Delta == 0$: there is a single intersection (tangential intersection): $t = -b/(2ac)$

Case 3: if $\Delta > 0$: then there are two solutions: t_1 and t_2:

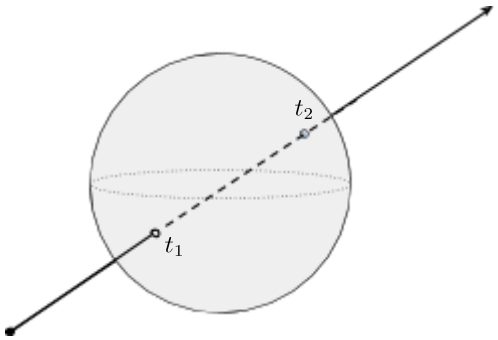
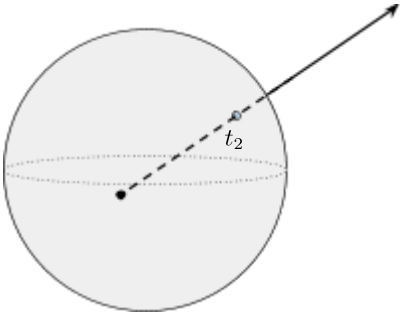
$$t_1 = \frac{-b - \sqrt{\Delta}}{2ac} \qquad t_2 = \frac{-b + \sqrt{\Delta}}{2ac}$$

Write t_1 and t_2 in terms of sphere and ray parameters by substituting a, b, c and Δ.

$t_1 =$

$t_2 =$

Step 5 : For case 3, there are two sub-cases:

If $t_1 > 0$: there are two intersections t_1 and t_2	**If $t_1 < 0$: there is one intersection : t_2**
	
End point is outside	**End point is inside**

You do not need to do anything for this step.

Pre-Lab Part 2: Familiarizing with the skeleton code

2.A. Vectors

src/vec.h includes the definition of struct vec, that stores the data and provide the functionality for vectors (for math).
There are also some typedefs for the most common vector types:
`vec3`**: a 3D vector of doubles**
`vec2`**: a 2D vector of doubles**
`ivec2`**: a 2D vector of integers (int), all the operations will be done with integer precision.**
e.g.

[1.2 2 -0.9] can be created by: `vec3(1.2, 2, -0.9);`
[1.2 -0.9] can be created by `vec3(1.2, -0.9);`
[1 5] can be created by either `vec2(1,5);` **or** `ivec2(1,5);`

Q1. Given u and v are vec3 objects storing 3D vectors, fill the missing cells in the table below with code or it's explanation.

Code	Description
`u[0]=5` `v[2]=6`	Sets u_x, x component of u ,to 5 Sets v_z, z component of v, to 6
`vec3 m = u+v`	
`Vec3 p=u[0]*v`	
`double k=dot(u,v)`	
	Create a vec3 c that stores the cross product of u and v
	returns magnitude (length) of u.
`u.normalized()`	*returns* u/|u| (the unit vector in u's direction)
	Create a vec3 k, such that $k = \frac{(u \cdot v)u}{\|u\|^2}$
`cout << u << endl;`	prints vector u (values separated with commas)

2.B. Ray Tracing Algorithm and related classes/functions

The general algorithm for ray tracing is as follows:

```
for each pixel (i,j):
      Compute the 'world position' of the pixel
      Create a ray from the camera position to the world position of the pixel
      Cast the ray and evaluate the color of the pixel:
            Find the closest object, obj, that intersects with the ray.
            Get the pixel color by using Shader_Surface function of a shader.
                  If there are no objects, use background_shader
                  else, use obj's material_shader
```

Please find the appropriate files in the skeleton code and fill the blanks below.

1. **World_Position** function in Camera class, returns the world position of a given pixel, ivec2 pixel_index.
   a. World_Position function is *implemented* in camera.cpp starting from line # _____.

2. **Cell_Center** function in Camera class, returns the screen position of a given pixel, ivec2 pixel_index.
   a. Cell_Center function is *implemented* in camera.h starting from line # _____.

3. Locate where the loop that iterates through all pixels are located.
   a. The loop is located in _____ function in render_world.cpp

4. **Cast_Ray** function in render_world.cpp returns the color of the pixel using the shader of the closest object it intersects with. Find the function in render_world.cpp and fill below.
   a. Cast_Ray function is *implemented* in render_world.cpp starting from line # _____.
   b. Cast_Ray function is *called* _____ function in render_world.cpp.

5. **Closest_Intersection** function will be used in Cast_Ray function to find the closest object that intersects with the ray and (if any) provide it's intersection information (in a Hit object, hit).
Find the function in render_world.cpp, read its comments and fill below.
   a. Closest_Intersection function is *implemented* in render_world.cpp starting from line #_____.
   b. The input/output parameter,hit , should store _____.
   c. Any intersection with t <= small_t should be _____, where t is the distance to intersection.

6. **Intersection** function is a function of the Object class (object.h) which is a base class for scene objects such as plane and sphere. This function is overloaded by these classes and should return true if the object intersects with the ray, and fill the hits input/output parameter with all the intersections. See the Intersection functions in Sphere and Plane classes.

LAB 2 - Introduction to Ray Tracing (Assignment 1)

How to compile & run:

Compile: `make`
Run test N (00-37) with opengl display: `./ray_tracing_preview -i ./tests/N.txt`
Compare test N (00-37) with ground truth: `./ray_tracing -i ./tests/N.txt -s ./tests/N.png`
Run grading script: `./grading-script.py ./tests`

Functions to implement for this lab:

- *camera.cpp:* ***World_Position***
- *render_world.cpp:* ***Render_Pixel*** (only ray construction)
- *sphere.cpp:* ***Intersection***: returns all the intersections of the ray and the sphere.
- *render_world.cpp:* ***Closest_Intersection***
- *render_world.cpp:* ***Cast_Ray***

NOTE: *the code mostly uses "*double*" for floating numbers*

Important Classes:

```
render_world.h/cpp:
class Render_World: //Stores the rendering parameters such as
       std::vector<Object*> objects   //list of objects in the scene
       std::vector<Light*> lights;    //list of lights in the scene
       Camera camera;    //the camera object (see below)

camera.h/cpp:
class Camera: // Stores the camera parameters, such as
              // camera position, screen horizontal and vertical vectors etc.

hit.h:
class Hit: //Stores the ray object intersection information such as
       double t; //t parameter of the ray for intersection point (also the distance to the
endpoint)
       bool is_exiting; // stores if the ray is exiting or entering the object that it hit


ray.h:
class Ray // stores ray parameters: end_point, direction
       vec3 Point(double t); // stores if the ray is exiting or entering the object that it hit

sphere.h/cpp:
class Sphere // Stores sphere parameters (center, radius)

plane.h/cpp:
class Plane // Stores plane parameters (x0, normal)
```

Tips:

> ***Note:*** *The tips are only for the minimum requirement of the lab. You might need to change the code for the latter steps of the assignment.*

World position of a pixel (camera.cpp):

As you remember from the last week, the world position of a pixel can be calculated by the following formula:

$$F_p + u\ C_x + v\ C_y$$

- u: horizontal_vector, v: vertical_vector, and F_p: film_position (bottom left corner of the screen)
- C: of type vec2, can be obtained by `Cell_Center(pixel_index) //see camera.h`

Constructing the ray (Render_Pixel function):

end_point : camera position (from camera class)

direction: a unit vector from the camera position to the world position of the pixel.

vec3 class has *normalized()* function that returns the normalized vector;

e.g. `(v1-v2).normalized()`

Closest_Intersection:

Read the comments in the code for description. The pseudo code is given below

```
Set min_t to a large value (google std numeric_limits)
For each object in objects:
        Create an empty list of hits
        use object.Intersect to fill the hits
        For each h in list of hits:
                If h is closest so far (i.e. with smallest t, that is larger than small_t)
                        Set the object as closest_object, set hit to h and update min_t
return closest_object
```

Intersection *in sphere.cpp***:**

Use the equations in Pre-Lab Part 1 to check if there is an intersection

If so, compute t1 and t2 and, create Hit objects hit1 (if t1>0) , and hit2 and append them to hits vector

Cast_Ray:

Get the closest object, obj, intersecting with the ray using Closest_Intersection function

if there is an intersection:

To set the "*color*" variable, call obj's material shader's Shade_Surface function that calculates and returns the color of the ray/object intersection point.

Shade_Surface function requires (ray,intersect,normal,recursion_depth) for parameters.

For flat shading (tests 00-04) you can use a dummy vec3, or any available vec3, for intersect and normal.

E.g.

```
vec3 dummy;
color=obj->material_shader->Shade_Surface(ray,dummy,dummy,1,false);
```

This code, with dummy vectors, would not work for any tests after 04.txt

else:

Use *background_shader* of render_world class. The background shader is a flat_shader so you can use the dummy vector as above.

```
vec3 dummy;
color=background_shader->Shade_Surface(ray,dummy,dummy,1,false);
```

Note: flat_shading class is already fully implemented.