CS130 - LAB 2

Name: Tanish Arora **SID:** 862012234

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 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 the intersected object 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 # 42.

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 # 54.

3. Locate where the loop that iterates through all pixels are located.

a. The loop is located in Render_Pixel 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 # 50.

b. Cast_Ray function is *called* in Render_Pixel 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 object of type Hit.

a. Closest_Intersection function is *implemented* in render_world.cpp starting from line #__23__.

b. The output object hit should store the following information:
__dist >= small_t dist__.

c. Any intersection with distance <= small_t should be __ignored__.

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 return an hit object the closest intersection.

7. **Vectors.** Given that 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`	adds two vectors completely u + v, arithmetic operation
`vec3 p=u[0]*v`	scalar multiplication, x component of u vector with the “v” vector
`double k=dot(u,v)`	calls the dot product function and computes the dot product of u . v
vec3 c = cross(u,v);	Create a vec3 c that stores the cross product of u and v
u.magnitude();	returns magnitude (length) of u
`u.normalized()`	returns u/|u| (the unit vector in u's direction)
vec3 k = u * (dot(u,v)/u.magnitude_squared())	Create a vec3 k, such that $k = \frac{(u \cdot v)u}{\|u\|^2}$
`cout << u << endl;`	prints vector u (values separated with commas)

GETTING STARTED WITH THE RAY TRACER PROJECT

Compile: `scons`
Run test N (00-29): `./ray_tracer -i ./tests/N.txt`
Compare test N (00-29): `./ray_tracer -i ./tests/N.txt -s ./tests/N.png`
Run grading script: `./grading-script.py .`

Functions to implement for this lab:

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

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 the camera position
              // screen horizontal and vertical vectors etc.
```

object.h:
```
class Hit: // Stores the ray object intersection data such as the distance
           // from the endpoint to the intersection point with the object.
```

ray.h:
```
class Ray // stores ray parameters: end_point, direction
     vec3 Point(double t); // returns the point on the ray at distance t,
                           // i.e. (end_point + direction * t)
```

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

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

World position of a pixel (camera.cpp):

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:

The pseudo code is:

```
Set min_t to a large value (google std numeric_limits)
For each object* in objects:
    use object->Intersect to get the hit with the object
    If hit is the closest so far and larger than small_t
    (i.e. with smallest t, that is larger than small_t)
        then store the hit as the closest hit
return closest hit
```

Cast_Ray:

Get the closest hit with an object using *Closest_Intersection*
If there is an intersection:
   Set *color* using the object Shade_Surface function which calculates and returns the color of the ray/object intersection point.
   Shade_Surface *receives as parameters: ray, intersection point, normal at the intersection point and recursion_depth. You can get the intersection point using the* ***ray*** *object and the normal using the* ***object*** *pointer inside the* ***hit*** *object.*
Else (if there is no intersection)
   Use *background_shader* of the render_world class. The background shader is a flat_shader so you can use any 3d vector as parameters.

Credits: Muzaffer Akbay (Winter/17), Revision: Cassio (Fall/18)