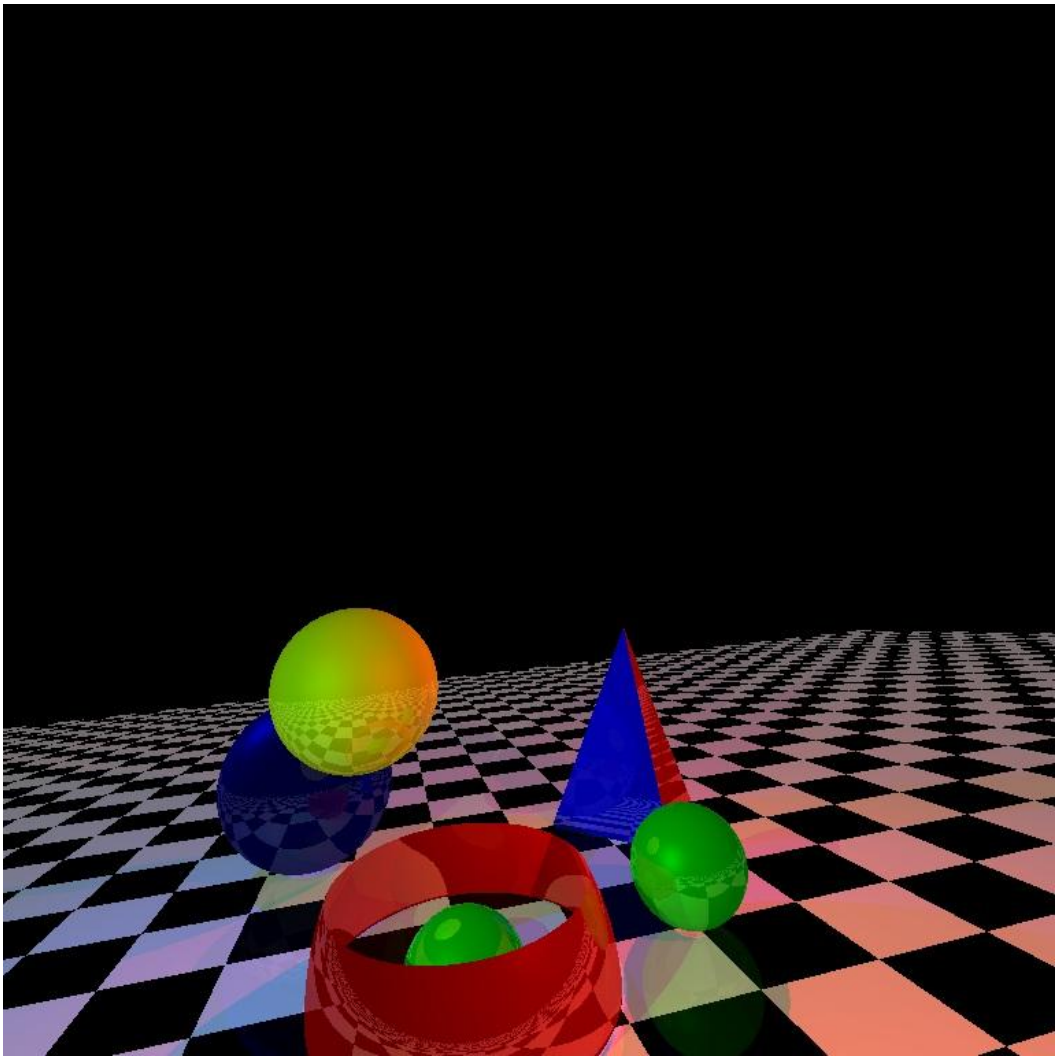
CSE410 (January 2021) - Assignment 3 : Ray Tracing

Table of Contents

In this assignment, you have to generate realistic images for a few geometric shapes using ray tracing with appropriate illumination technique.

Prerequisites

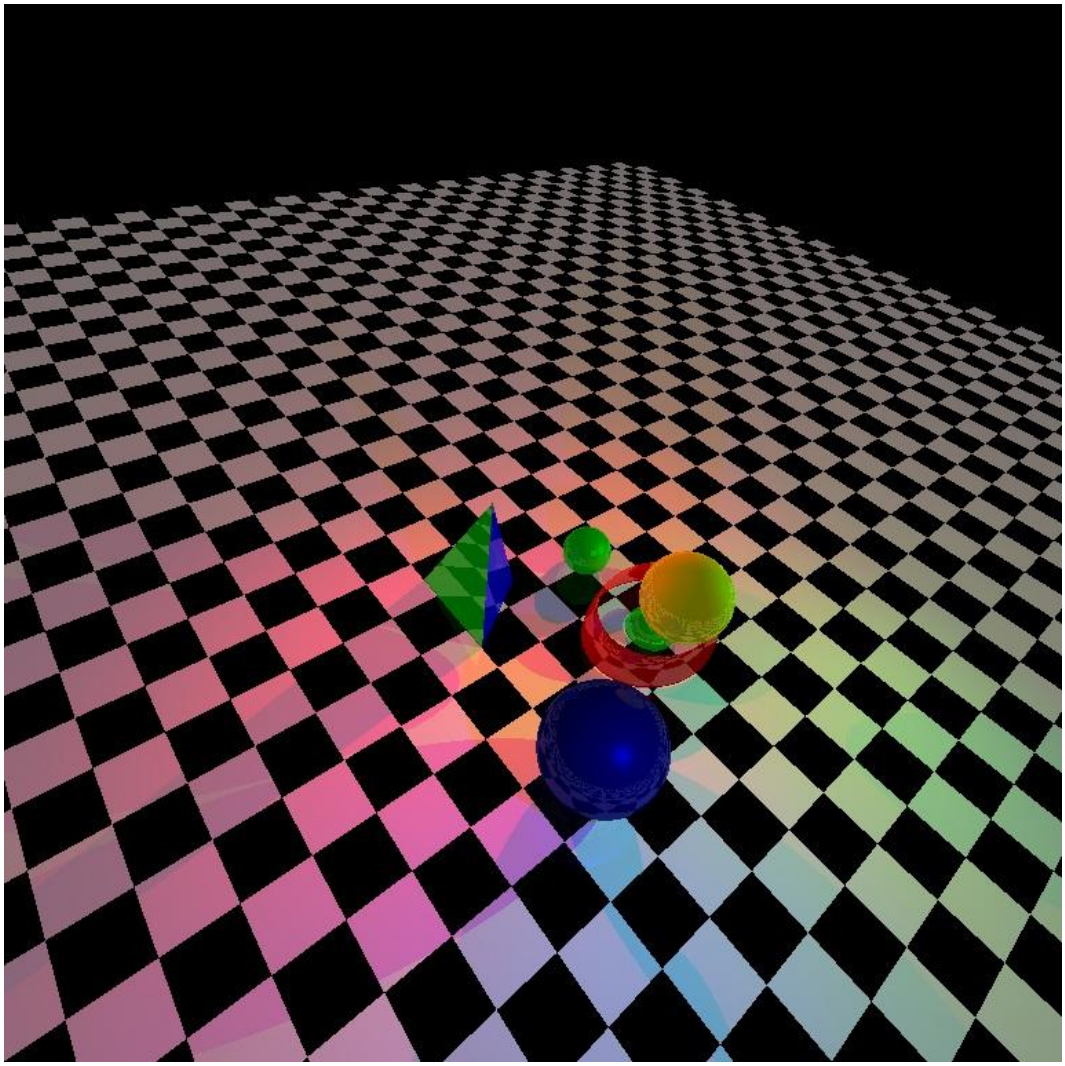
1. Basic knowledge of OpenGL (What you have learnt in assignment 1 should be sufficient)
2. Fully controllable camera (same as in assignment 1)
3. Bitmap image generation using the bitmap image header file provided for assignment 2
4. Basic idea of illumination, Phong lighting model to be more specific
5. Intersection of line with different 3D objects (e.g. ray-plane intersection, ray-sphere intersection, ray-triangle intersection etc.)
6. Multi-level reflection using ray tracing
7. Refraction using ray tracing, texture etc. (optional)



Implementation of a Fully Controllable Camera

You need to move and rotate the camera freely. Check the details from Assignment 1 if required.

Key	**Function**		**Key**	**Function**
Up arrow	Move forward		1	Rotate/Look left
Down arrow	Move backward		2	Rotate/Look right
Left arrow	Move left		3	Look up
Right arrow	Move right		4	Look down
PageUp	Move up		5	Tilt Clockwise
PageDown	Move DOwn		6	Tilt Counterclockwise



Taking and Processing Input

1. Call a function named `loadData()` from your main function. This function should read a text file named "scene.txt" containing details of different objects (shapes) and lights present in the environment. There can be three types of objects (shapes) in the file - sphere, triangle and object (shape) having a general quadratic equation in 3D. All the light sources should be considered as point light with appropriate color Check the given "scene.txt" file for further clarification.
2. Create a separate header file/src file and define the classes here. The name of the file should have your student no. as prefix (e.g. 1605123_classes.h). Include this file in the cpp file containing the main function. This file should be named similarly, having your student no. as prefix (e.g. 1605123_main.cpp).
3. Create a base class named `Object` in the header/src file mentioned in Step 2. You should define separate classes for each object (shape), and all of them should inherit the

`Object` class. The `Object` class can initially have the following methods and attributes. You can add or refactor later as appropriate.

```
Object{
     Vector3D reference_point // should have x, y, z
     double height, width, length
     double color[3]
     double coEfficients[4]   // reflection coefficients
     int shine // exponent term of specular component
     object(){}
     virtual void draw(){}
     void setColor(){}
     void setShine(){}
     void setCoEfficients(){}
}
```

The derived classes can use the attributes of the `Object` class, as appropriate. Each of them, however, must override the draw method. For example, you can design a `Sphere` class as follows.

```
Sphere : Object{
     Sphere(center, radius){
          Reference_point = center
          length = radius
     }
     void draw(){
          // write codes for drawing sphere
     }
     ...
}
```

Besides, there should be another class named `Light` and it should contain the position of the point light source and its color property.

```
Light{
     Vector3D light_pos;
     double color[3];
     ...
}
```

4. In the cpp file having the main function (let us refer to it as the MAIN_FILE from now on), keep two vectors one for objects and another one for lights and make it accessible to the header file (let us refer to it as the HEADER_FILE from now on). The `extern` keyword may help in this regard. These vectors should be used to store all the objects and light sources given as input.

```
// declaration
vector <Object> objects;
Vector <Light> lights;
```

```
// populating in the loadData() function
Object *temp
temp = new Sphere(center, radius); // received as input
// set color
// set coEfficients
// set shine
objects.push_back(temp)
// similarly construct a light object, say, l
lights.push_back(l)
```

5. Besides the objects given in the input file, you need to draw a floor. So create a `Floor` class which inherits the `Object` class (just like the other shapes). You need to write its draw method so that a checkerboard of two predefined alternating colors is drawn. You can choose the colors, reflection coefficients, shine (i.e. exponent term) etc. as you like.

```
// declaration
Floor: Object{
     Floor(floorWidth, tileWidth){
          reference_point=(-floorWidth/2,-floorWidth/2,0);
          length=tileWidth
     }
     Void draw(){
          // write codes for drawing a checkerboard-like
          // floor with alternate colors on adjacent tiles
     }
}
```

```
// populating in the loadData() function
temp = new Floor(1000, 20) // you can change these values
// set color
// set coEfficients
// set shine
objects.push_back(temp)
```

Implementation of the `draw()` Method

This is trivial as for spheres and triangles. You can use OpenGL functions and your code of assignment 1 for this purpose. Think about how you can draw the floor like a checkerboard. This is quite simple as well. **However, you do not have to draw the general quadric surfaces.** You only need to show them in the bmp image file, generated by capturing the scene, as elaborated next.

Implementation of the `capture()` and the `intersect()` Methods

1. Create a method `capture()` in MAIN_FILE which will be called when you press 0.
2. Create a class named Ray in the HEADER_FILE.

```
Ray{
     Vector3D start;
     Vector3D dir;  // normalize for easier calculations

     //write appropriate constructor
}
```

3. Pseudocode of the `capture()` method may be as follows.

```
capture(){
    initialize bitmap image and set background color
    planeDistance  = (windowHeight/2.0) /
                      tan(viewAngle/2.0)
    topleft = eye + l*planeDistance - r*windowWidth/2 +
                                      u*windowHeight/2
    du = windowWidth/imageWidth
    dv = windowHeight/imageHeight
    // Choose middle of the grid cell
    topleft = topleft + r*(0.5*du) - u*(0.5*dv)

    int nearest;
    double t, tMin;
    for i=1:imageWidth
        for j=1:imageHeight
            calculate curPixel using topleft,r,u,i,j,du,dv
            cast ray from eye to (curPixel-eye) direction
            double *color = new double[3]
            for each object, o in objects
                t = o.intersect(ray, dummyColor, 0)
                update t so that it stores min +ve value
                save the nearest object, o_n
            t_min = o_n->intersect(ray, color, 1)
            update image pixel (i,j)
    save image
```

4. In the `Object` class, create a virtual method `intersect()` as follows.

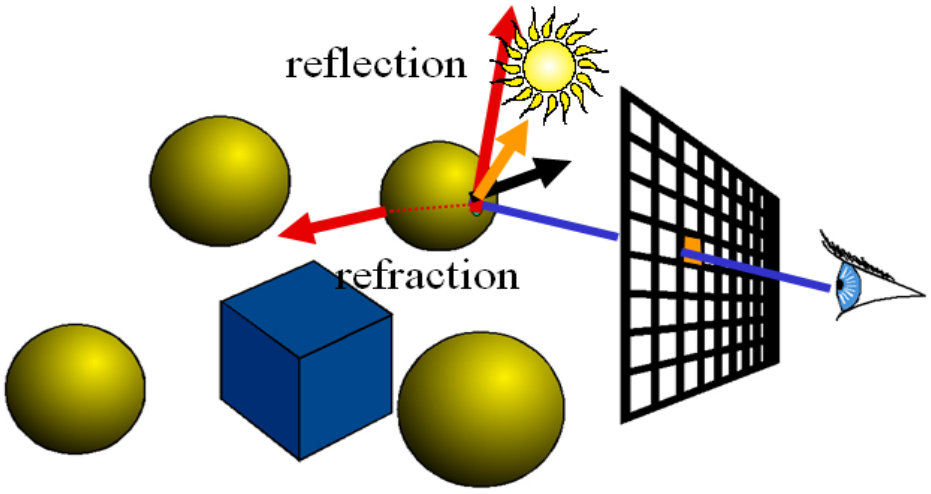
```
virtual double intersect(Ray *r, double *color, int level){
    return -1.0;
}
```

5. In each of the derived classes, override the `intersect()` method. This will vary for each different object (shape). More specifically, you have to implement ray-sphere, ray-triangle, ray-general quadric surfaces, ray-floor etc. intersections for the different objects. Some of these are discussed in the theory classes.
6. Once you have implemented the `intersect()` method of a class, say, sphere, it is encouraged to test if your program works correctly. If not, then you can manually do the computations for a custom ray passing through a particular pixel and debug your code to find what went wrong.

Illumination with the Phong Lighting Model

In the `intersect()` method, after computing intersecting t of ray r, add some lighting codes. This method receives an integer `level`, as its last parameter, which actually determines if the ray currently under consideration will be reflected (and/or refracted) again or not. But this can be used to decide if a pixel should be colored or not as well.

1. When level is 0, the purpose of the `intersect()` method is to determine the nearest object only. No color computation is required then. (You could do this with some different method as well, but the computations would be fairly similar.)
2. When level > 0, add lighting codes according to the Phong model i.e. compute ambient, diffuse and specular components for each of the light sources and combine them.

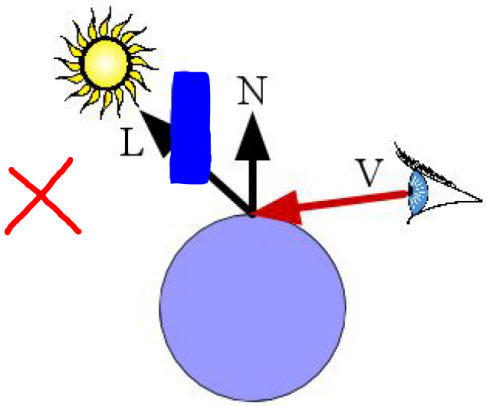


3. After computing t, you can refactor your `intersect()` method as follows.

```
double intersect(Ray *r, double *color, int level){
    // code for finding intersecting t_min

    if level is 0, return t_min
    intersectionPoint = r->start + r->dir*t_min
    intersectionPointColor = getColorAt(intersectionPoint)
    color = intersectionPointColor*coEfficient[AMB]
    calculate normal at intersectionPoint

    for each light l in lights
        cast ray_l from l.light_pos to intersectionPoint
        // if intersectionPoint is in shadow, the diffuse
        // and specular components need not be calculated
        if ray_l is not obscured by any object
```




```
            calculate lambertValue using normal, ray_l
            find reflected ray, ray_r for ray_l
            calculate phongValue using r, ray_r
        color += l.color*coEfficient[DIFF]*lambertValue*
                 intersectionPointColor
        color += l.color*coEfficient[SPEC]*phongValue^shine *
                 intersectionPointColor
        // l.color works as the source intensity, I_s here
    ...
}
```

Recursive Reflection

To handle reflection, you need to do the same calculations as camera rays (i.e. the ones cast from the eye). The recursion_level (given as input) controls how many times a ray will be reflected when incident upon objects (shapes). Say, if recursion_level is set to 2, a camera ray should be reflected by objects (shapes) and the primary resulting reflected rays should also be reflected by

other objects (shapes), but reflection will no longer have to be considered afterwards for the secondary reflected rays. You can do the following for reflection in the `intersect()` method after color computation.

```
double intersect(Ray *r, double *color, int level){
    // code for finding color components
    if level ≥ recursion_level, return t_min
    construct reflected ray from intersection point
    // actually slightly forward from the point (by moving the
    // start a little bit towards the reflection direction)
    // to avoid self intersection
    find t_min from the nearest intersecting object, using
    intersect() method, as done in the capture() method
    if found, call intersect(r_reflected, color_reflected, level+1)
    // color_reflected will be updated while in the subsequent call
    // update color using the impact of reflection
    color += color_reflected * coEfficient[REC_REFFLECTION]
}
```

Implementation of the `intersect()` Method Revisited

As you may have already understood, the `intersect()` method will vary for different objects (shapes). More precisely, ray-object (shape) intersection calculation, normal calculation, obtaining color at intersection point etc. functions will differ from one object (shape) to another. So you can write virtual functions for them and override appropriately in the derived classes.

- Floor
    - Ray-plane intersection followed by checking if the intersection point lies within the span of the floor
    - Normal = (0,0,1)
    - Obtaining color will depend on which tile the intersection point lies on
- Triangle
    - Ray-triangle intersection
    - Normal = cross product of two vectors along the edges e.g. (**b**-**a**)X(**c**-**a**)
    - Obtaining color is trivial since it is same across the whole triangle
- General Quadric Surfaces
    - Equation : $F(x,y,z) = Ax^2+By^2+Cz^2+Dxy+Exz+Fyz+Gx+Hy+Iz+J = 0$
    - Ray-quadric surface intersection (by plugging in $P_x = R_{0x} + t*R_{dx}$ and similarly P_y and P_z from the ray, into the general 3D quadratic equation)
    - If two values of t are obtained, check which one (or none or both) falls within the reference cube i.e. the bounding box within which the general quadric surface needs to be drawn. If any dimension of the bounding box is 0, no clipping along that dimension is required.

- Normal = (∂F/∂x, ∂F/∂y, ∂F/∂z) [Substitute x, y, z values with that of intersection point to obtain normals at different points]
- Obtaining color is trivial since it is same across the whole quadric surface

Memory Management

Properly manage memory irrespective of using STL or pointers (both are allowed).

Bonus Tasks

1. Implement refraction: The camera rays when incident upon an object (shape), should be refracted through it besides being reflected. You can consider the refraction coefficient for color to be half of the recursive reflection coefficient or some other suitable value (you can try to experiment and check what gives a good view).
2. Implement texture : You can download your BIIS image and paste it on the relatively darker tiles of the floor (e.g. if your tiles are black and while, choose black tiles).

General Suggestions

PLEASE START EARLY. This is a fairly complex assignment. Trying to complete it overnight will not help. So, invest sufficient time. Hopefully, you will like the outcome!

Try to code incrementally. Follow the "scene.txt" file to prepare a test input file, say, "scene_test.txt". Gradually add objects and complex attributes e.g. recursion level to it. Always test if what you have implemented so far works properly or not.

- Initially keep only a sphere and a light source in "scene_test.txt" file.
  a. Load the center, radius, color, reflection coefficients and the exponent term of the specular component of the sphere and the position and color of the light source using the `loadData()` function.
  b. Construct a `Sphere` object and a `Light` object accordingly and insert them into `objects` and `lights` vectors respectively.
  c. Loop over the `objects` vector and call the `draw()` method for each object. Inside the `draw()` method, write how you would draw the object using OpenGL.
  d. Loop over the `lights` vector and draw them using points with appropriate color. You can add a `draw()` method for `Light` class too if you want.
  e. Run your code and check if you are getting the desired output.
  f. Implement the `capture()` method for generating a bitmap image and `intersect()` method for the objects (start with the sphere only) for the `capture()` method to work. Compute the illumination based on the Phong lighting model. Clip the color values so that they are in [0, 1] range.
  g. Run your code and check if you are getting the desired output.
- Add a second sphere to the "scene_test.txt" file.
  a. Handle reflections based on recursion level, given as input, in the `intersect()` method. While working with the reflected rays, you may want to

fix their start points a bit above the surface (instead of the exact intersection point) so that intersection with the same object (shape) due to precision constraint can be avoided.

   b. Write `draw()` and `intersect()` methods for the floor object.
   c. Handle shadows in the `intersect()` method.
   d. Run your code and check if you are getting the desired output.

- Add other objects to the "scene_test.txt" file, followed by more lights of different colors.
   a. Implement their `draw()` and `intersect()` methods.
   b. Run your code and check if you are getting the desired output.
- When you can generate desired output for all the shapes and lights, try to think of some corner cases and check if your program can handle them.

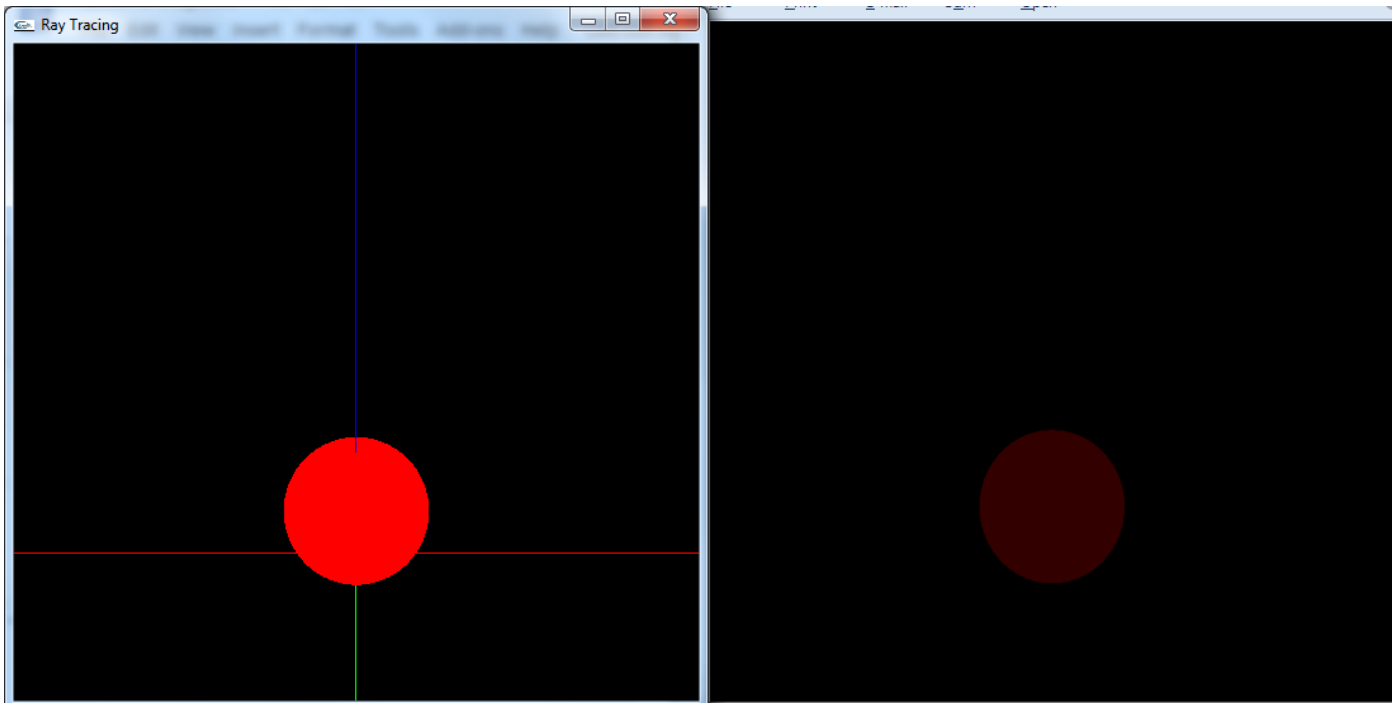
Resources/References

1. Theory class lectures, recordings on Lighting and Shading
2. Theory class lectures, recordings on Ray Casting and Ray Tracing
3. https://asawicki.info/news_1301_reflect_and_refract_functions.html
4. https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm

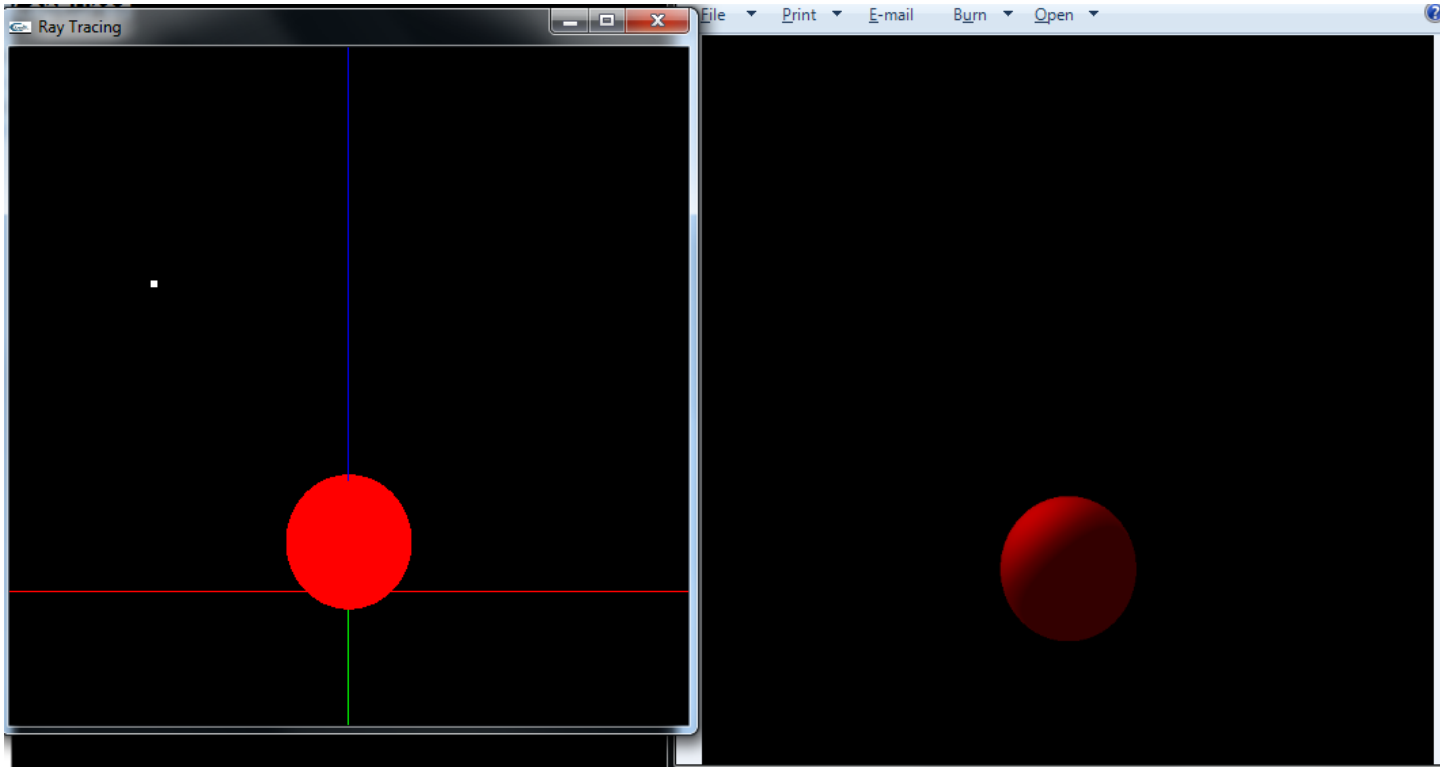
Sample I/O

You can check out some of the following intermediate outputs for reference (your code may not necessarily generate the exact same images, but you can perhaps get a brief idea from here).

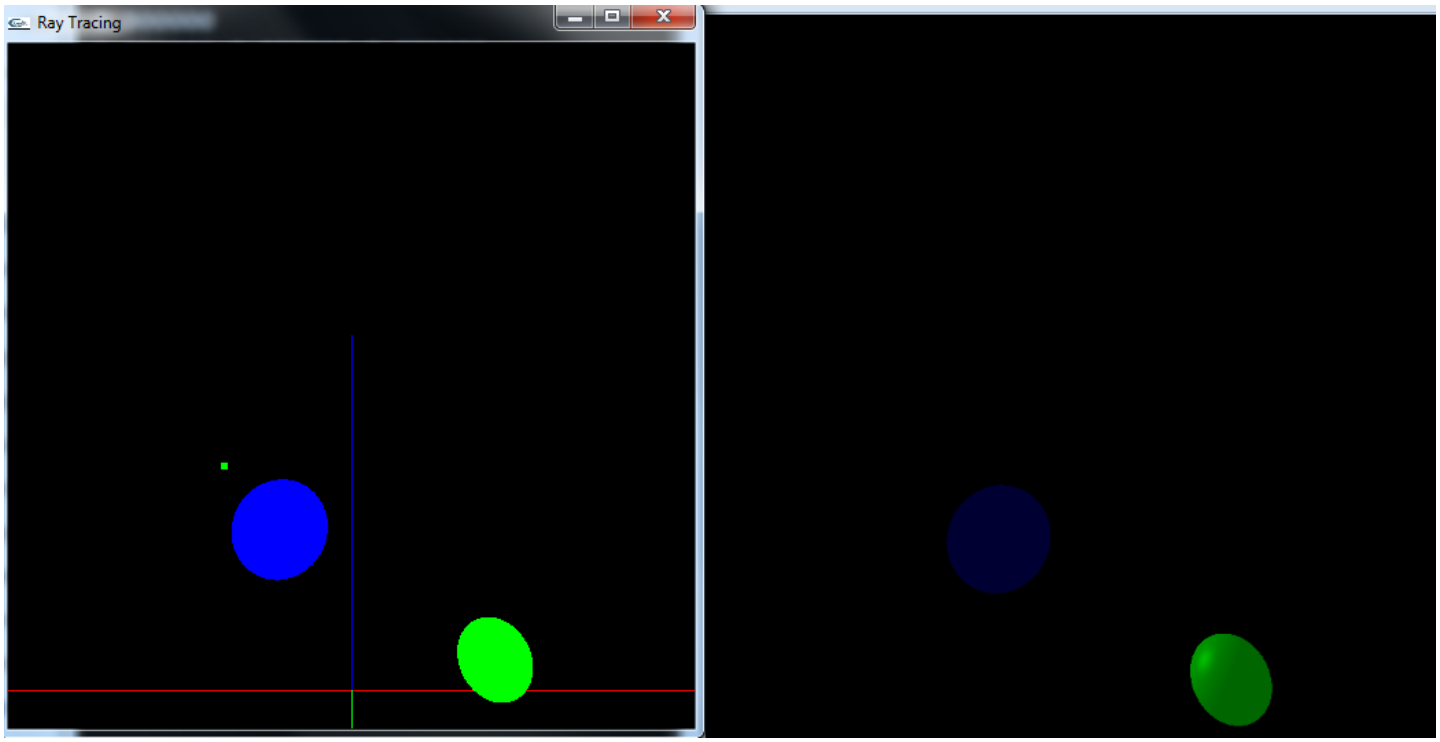
1. One sphere and no light source



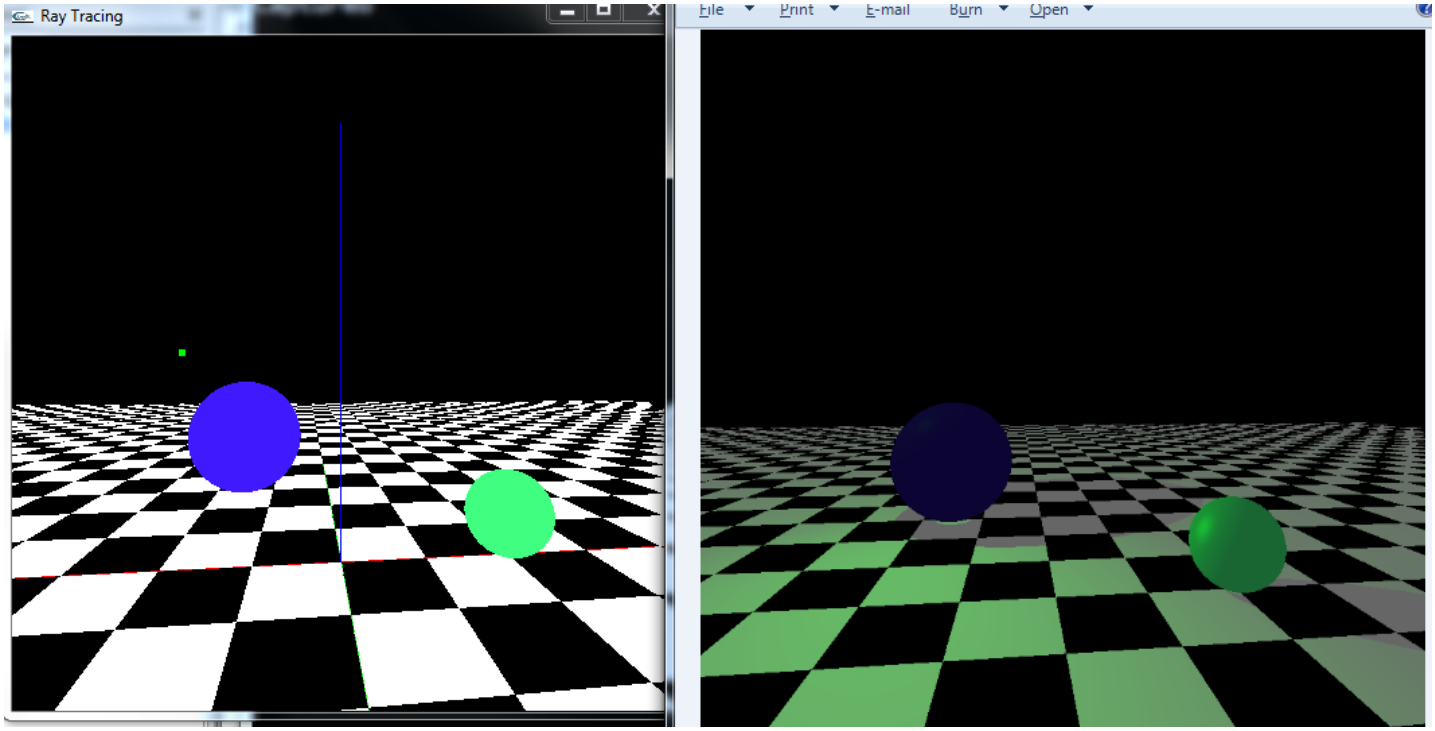
2. One sphere and one white light source



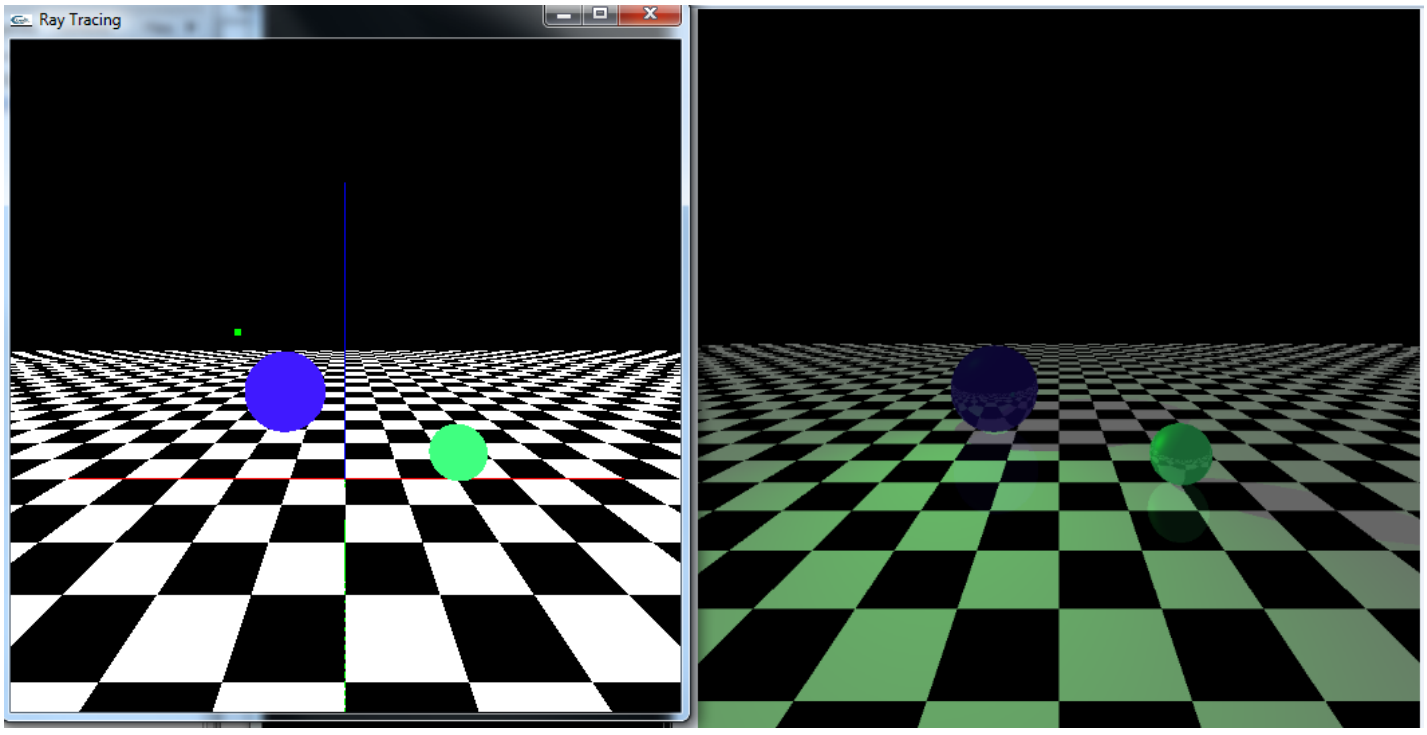
3. Two spheres (one pure blue, one pure green) and one green light source



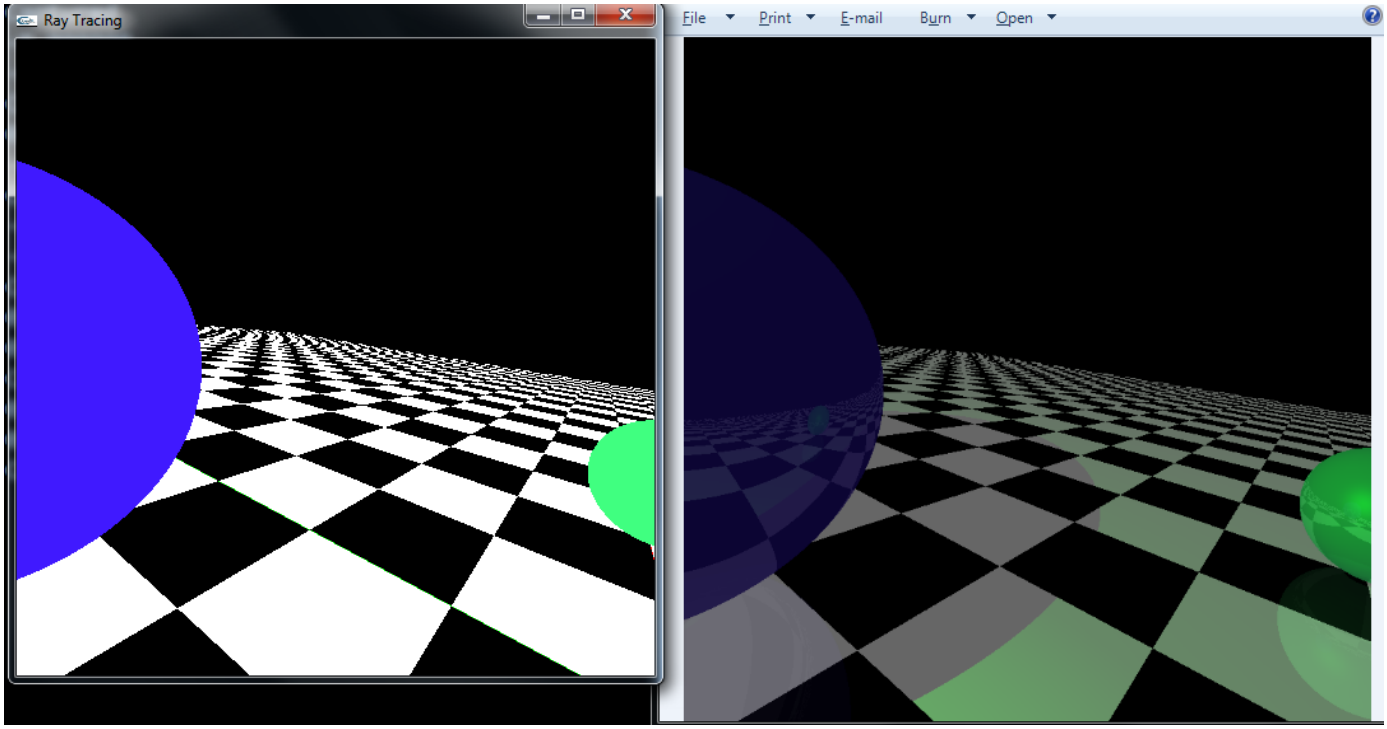
4. Two spheres (having all of r,g,b components), floor and one green light source



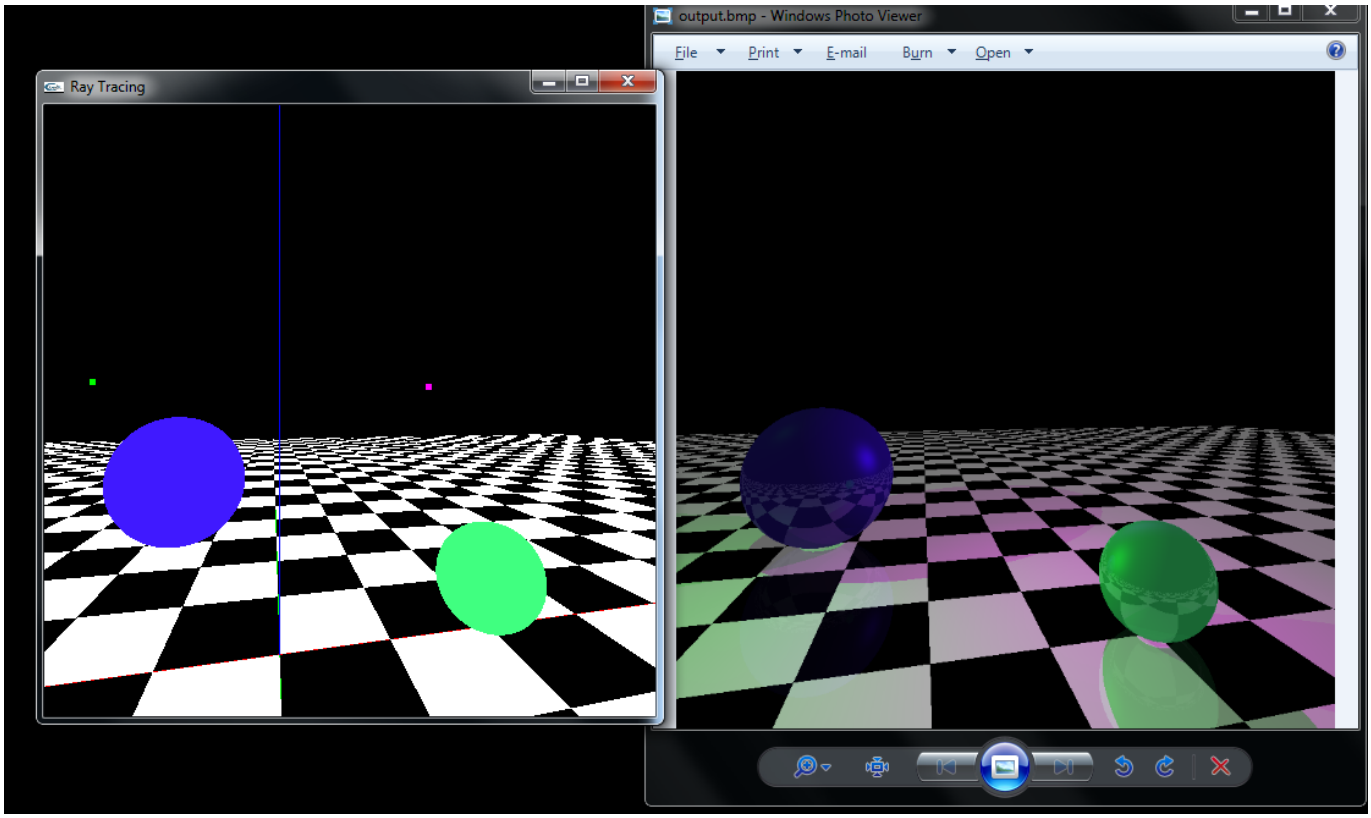
5. Two spheres (having all of r,g,b components), floor and one green light source, recursion level = 2



6. Two spheres (having all of r,g,b components), floor and one green light source, recursion level = 4



7. Two spheres (having all of r,g,b components), floor and one green, one violet/pink (red+blue) light source, recursion level = 4



Marks Distribution

Check out the Offline-3 tab of the CSE 410-Continuous evaluation google sheet shared with you.

Submission Guidelines

1. Create a folder having the same name as your 7 digit student id. If your student id is 1605xxx, then the name of the folder will be 1605xxx.
2. Rename all your source files so that they have your student id as prefix (e.g. 1605XXX_Main.cpp, 1605XXX_Header.h etc.).
3. Put the source files in the folder created in step 1 and zip the folder.
4. Upload the zip file (1605XXX.zip) on Moodle.

Submission Deadline

13th week, Sunday 2:25 PM (No Extension)

Expected date of 13th week, Sunday : July 04, 2021.