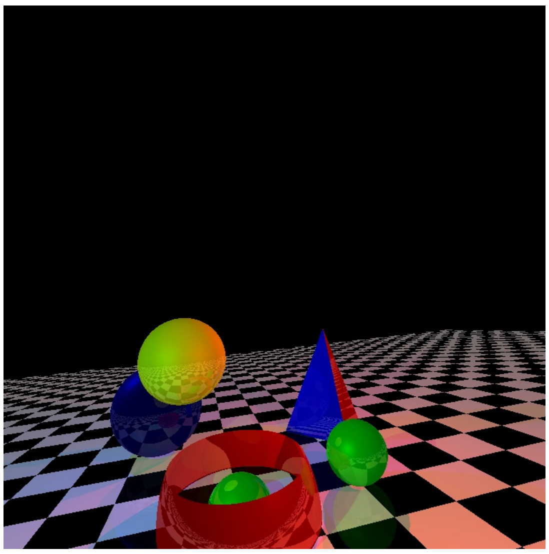
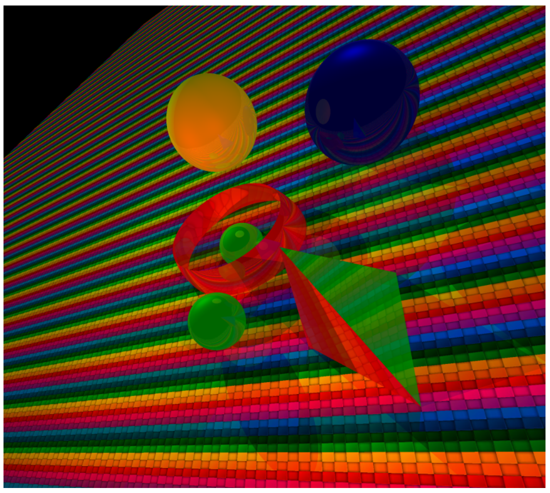
CSE410 (January 2025) - Assignment 3: Ray Tracing

Contents

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

1 Prerequisites

1. Basic knowledge of OpenGL (What you have learned 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. The intersection of lines with different 3D objects (e.g. ray-plane intersection, ray-sphere intersection, ray-triangle intersection, etc.)
6. Multi-level reflection using ray tracing
7. Using texture in OpenGL

2 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

3 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. There will be two types of light sources - point lights and spotlights. In the case of point lights, their color and position will be specified. In the case of the spotlights, their direction and cutoff angle will be specified in addition to the position and color information. 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 a prefix (e.g. 2005123_classes.h). Include this file in the cpp file containing the main function. This file should be named similarly, having your student no. as a prefix (e.g. 2005123_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] // ambient, diffuse, specular,
                           // 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. 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 can be two other classes named `PointLight` and `SpotLight`. The `PointLight` class may contain the position of the point light source and its color.

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

The `SpotLight` class, on the other hand, should contain direction and cutoff angle in addition. So it can have a `PointLight` member variable and these two other properties.

```
SpotLight{
    PointLight point_light;
    Vector3D light_direction;
    double cutoff_angle;
    ...
}
```

Alternatively, you can use the idea of inheritance and implement the classes differently.

4. In the cpp file having the main function (let us refer to it as the MAIN_FILE from now on), keep a vector for objects and one/two other(s) for lights (you can keep one for point lights and the other for spotlights or you can keep a single vector of lights - whichever suits your implementation) 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 <PointLight> pointLights;
vector <SpotLight> spotLights;

// 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)

// construct a point light object, say, pl
pointLights.push_back(pl)

// construct a spot light object, say, sl
spotLights.push_back(sl)
```

5. Besides the objects given in the input file, you need to draw a floor. So, create a `Floor` class that 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)
```

4 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.

5 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 // The 1st image you capture after running the
               // program should be named Output_11.bmp, the 2nd
               // image you capture should be named Output_12.bmp
               // and so on.
```

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.

6 Illumination with the Phong Lighting Model

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

1. When the level is 0, the purpose of the `intersect()` method is to determine the nearest object only. No color computation is required. (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 point light pl in pointLights
        cast ray_l from pl.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 += pl.color*coEfficient[DIFF]*lambertValue*
                      intersectionPointColor
            color+= pl.color*coEfficient[SPEC]*phongValue^shine *
                     intersectionPointColor
            // pl.color works as the source intensity, I_s here
        // Do the same calculation for each spot light unless
        // the ray cast from light_pos to intersectionPoint
        // exceeds cutoff-angle for the light source
        ...
}
```

7 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 afterward 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]
}
```

8 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 the intersection point, etc., functions will differ from one object (shape) to another. So, you can write virtual functions for them and override them appropriately in the derived classes.

8.1 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

8.2 Triangle

- Ray-triangle intersection
- Normal = cross product of two vectors along the edges e.g. $(b - a) \times (c - a)$
- Obtaining color is trivial since it is same across the whole triangle

8.3 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 $P_x = R_{0x} + t \cdot R_{dx}$ 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 = $(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z})$ [Substitute x, y, z values with that of the intersection point to obtain normals at different points]
- Obtaining color is trivial since it is same across the whole quadric surface

9 Memory Management

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

10 Adding Texture

After you are done with other things, try to bind a texture to the floor instead of making it a black-and-white checker board. You can check out the reading material in 13. For ray tracing, after you have determined the intersecting point, you have to retrieve the texture colour from the corresponding coordinate of the texture and use it. Other things will remain same. The following is a sample code for retrieving the colour from a texture at (u, v). **Note that this coordinate is texture coordinate, not the coordinate of the floor. You have to determine this from intersection coordinate.**

```
Color sampleTexture(double u, double v){
    if (!textureData || textureWidth <= 0 || textureHeight <= 0) {
        return Color(0.5, 0.5, 0.5);  // Gray fallback
    }

    // Clamp u and v to [0,1]
    u = std::max(0.0, std::min(1.0, u));
    v = std::max(0.0, std::min(1.0, v));

    // Normalized -> pixel coords
    int pixel_x = (int)(u * (textureWidth  - 1));
    int pixel_y = (int)(1.0 - v) * (textureHeight - 1));  // Flip Y

    // Safety clamp
    pixel_x = std::max(0, std::min(textureWidth - 1, pixel_x));
    pixel_y = std::max(0, std::min(textureHeight - 1, pixel_y));

    // Compute array index
    int index = (pixel_y * textureWidth + pixel_x) * textureChannels;
    int max_index  = textureWidth * textureHeight * textureChannels;
    if (index < 0 || index + 2 >= max_index) {
        return Color(1.0, 0.0, 1.0);  // Magenta = error
    }

    Color color;
    color.r = textureData[index] / 255.0;

    if (textureChannels >= 2) {
        color.g = textureData[index + 1] / 255.0;
    } else {
        color.g = color.r; // Grayscale
    }

    if (textureChannels >= 3) {
        color.b = textureData[index + 2] / 255.0;
    } else {
        color.b = color.r; // Grayscale
    }

    return color;
}
```

You can use any texture file you want. Keep a mechanism for easy switching between checker board and texture floor.

11 Bonus Tasks

1. Instead of Phong Lighting model, use more realistic PBR (Physically Based Rendering) model. You can check out the reading materials listed in 13.
2. Do raytracing on the GPU (You do not need dedicated GPU for this, coding on integrated GPU will be enough). You can use any framework for this (e.g. OpenGL, GSL, CUDA, OpenCL, SYCL, Metal, Vulkan, DirectX etc). **Note, you must do the mandatory tasks in OpenGL. Just for this bonus task, you can use other frameworks.**

You have to implement all the mandatory tasks to be applicable for bonus marks.

12 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 point light source in the "scene_test.txt" file.
    1. Load the center, radius, color, reflection coefficients, the exponent term of the specular component of the sphere, and the position and color of the point light source using the `loadData()` function.
    2. Construct a `Sphere` object and a `PointLight` object accordingly and insert them into objects and pointLights vectors, respectively.
    3. 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.
    4. Loop over the pointLights vector and draw them using points with the appropriate colors. You can add a `draw()` method for the pointLight class too, if you want.
    5. Run your code and check if you are getting the desired output.
    6. 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.
    7. Run your code and check if you are getting the desired output.
- Add a second sphere to the "scene_test.txt" file.
    1. 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 constraints can be avoided.

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

- Add other objects to the "scene_test.txt" file, followed by more point lights of different colors.
    1. Implement their `draw()` and `intersect()` methods.
    2. Run your code and check if you are getting the desired output.

- Add a spotlight to the "scene_test.txt" file.
    1. Construct a spotLight object and insert it into the spotLights vector.
    2. Draw each spotlight in the spotLights vector, just like point lights
    3. Add codes in the `intersect()` methods of different objects so that they can handle interaction with spotlights. You can start with the sphere and gradually work with the other objects. Note that, the additional check you need to do is whether the light ray from light_pos of the spotlight is within the cutoff angle or not.
    4. Run your code and check if you are getting the desired output. If it does, add more spotlights.

- When you can generate the desired output for all the shapes and lights, try to think of some corner cases and check if your program can handle them.

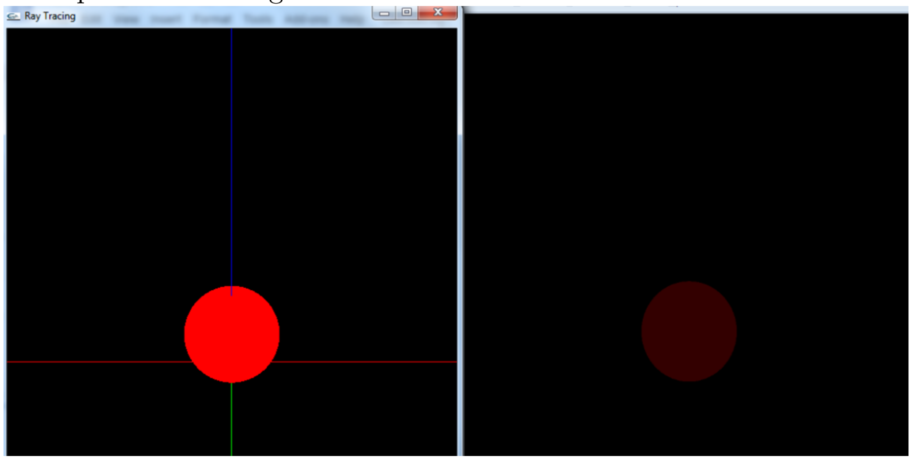
13 Resources/References

1. Ray-Triangle Intersection
2. Reflection and Refraction
3. Texture
4. PBR
5. PBR 2

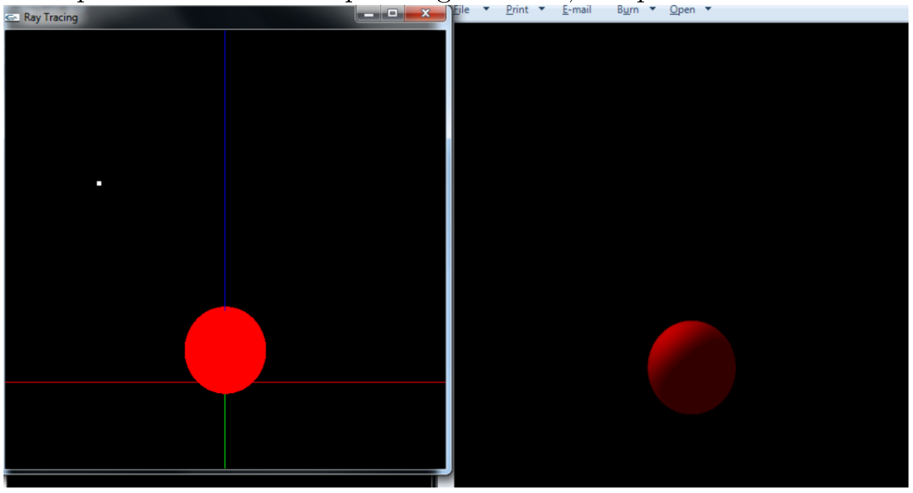
14 Sample I/O

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).

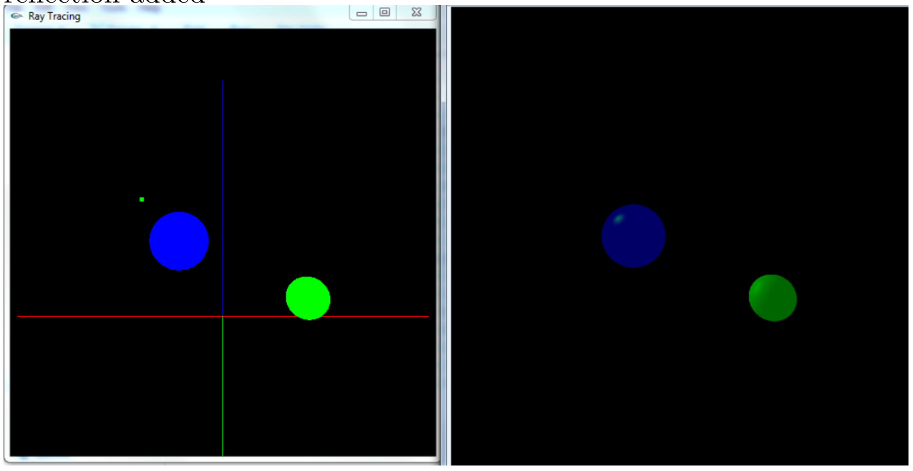
1. One sphere and no light source



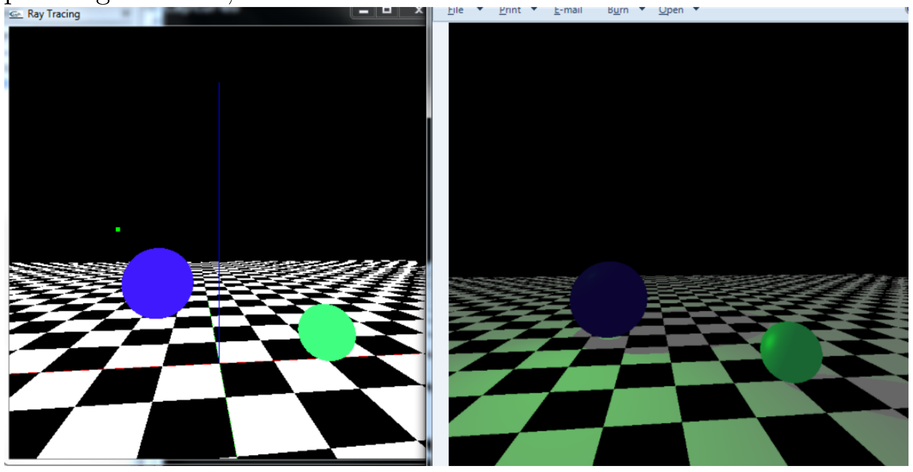
2. One sphere and one white point light source, no specular reflection



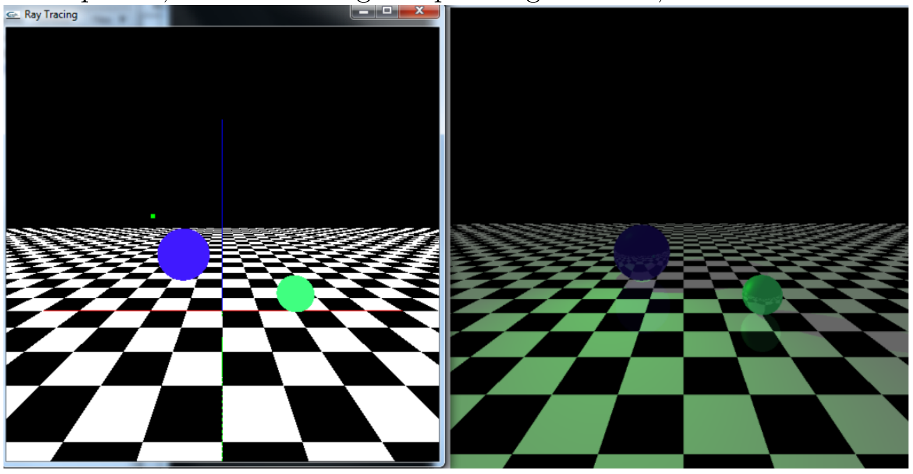
3. Two spheres (one pure blue, one pure green) and one green point light source, specular reflection added



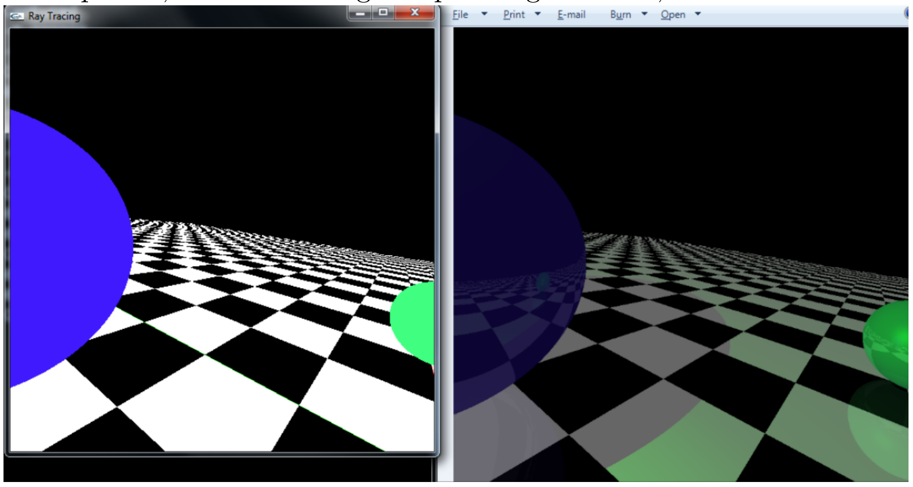
4. Two spheres (with slightly different color and coefficients from before), floor and one green point light source, reflection level = 1



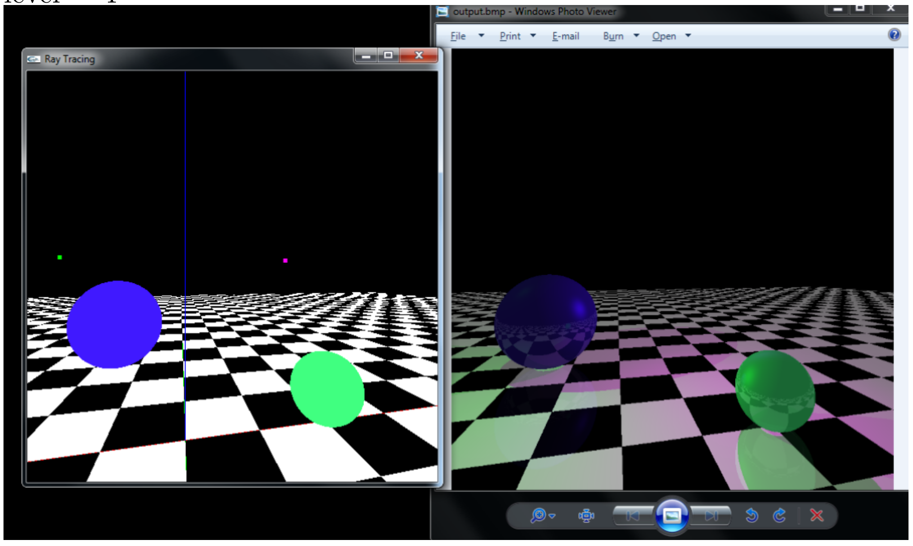
5. Two spheres, floor and one green point light source, recursion level = 2



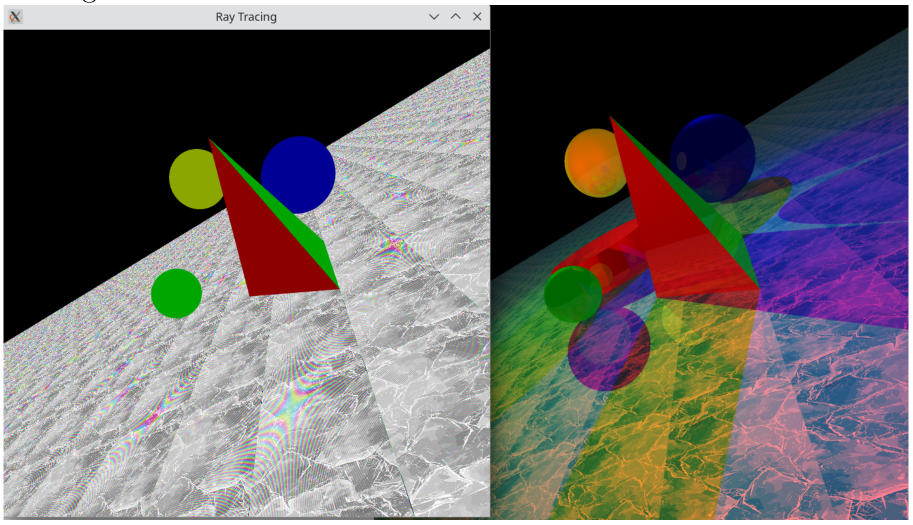
6. Two spheres, floor and one green point light source, recursion level = 4



7. Two spheres, floor and one green, one violet/pink (red+blue) point light source, recursion level = 4



8. Adding a custom texture to the final scene



15 Marks Distribution

- File I/O, Camera control, Memory management, Drawing in OpenGL etc. - 20%
- Ray-object intersection (ray casting) - 30%
- Recursive reflection (ray tracing) - 15%
- Illumination - 20%
- Texture - 15%
- Bonus - 25% (PBR - 10%, GPU - 15%)

16 Submission Guidelines

1. Create a folder having the same name as your 7 digit student id. If your student id is 2005xxx, then the name of the folder will be 2005xxx.
2. Rename all your source files so that they have your student id as prefix (e.g. 2005XXX_Main.cpp, 2005XXX_Header.h etc.).
3. Put the source files in the folder created in step 1 and zip the folder.
4. Upload the zip file (2005XXX.zip) on Moodle.

17 Submission Deadline

Deadline will be on 4 July 2025, 11:59 PM.