COSC363 Computer Graphics

Lab07: Recursive Ray Tracing

Aim:

In this lab, we will extend the ray tracer developed in lab-06 to generate renderings of planar surfaces, reflections, patterns and textures.

I. RayTracer.cpp from Lab06

1. In Lab06, we developed a basic ray tracer for rendering a scene containing a set of spheres, and implemented ambient, diffuse, and specular reflections and shadows (Fig. 1).

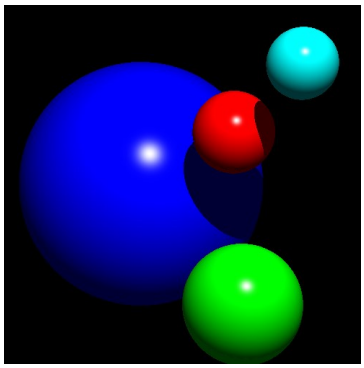


Fig. 1.

2. We will now generate reflections on the blue sphere. Reflectivity is one of the attributes of a sceneObject. The material properties of scene objects such as color, reflectivity, and specularity are specified in the `initialize()` function. Please add the following statement in this function:

```
sphere1->setReflectivity(true, 0.8);
```

The second parameter in the above function provides the value of the coefficient of reflection ρ_r (Lec08-Slide 16). For scene objects that are marked as reflective, we need to generate secondary rays along the direction of reflection at points of intersection and recursively call the `trace()` function to get the reflected colour values. This is done by including the following code segment in the `trace()` function (after computing the shadow color, and before returning the final color value). See also Lec08-Slide 20:

```
if (obj->isReflective() && step < MAX_STEPS)
{
	float rho = obj->getReflectionCoeff();
	glm::vec3 normalVec = obj->normal(ray.hit);
	glm::vec3 reflectedDir = glm::reflect(ray.dir, normalVec);
	Ray reflectedRay(ray.hit, reflectedDir);
	glm::vec3 reflectedColor = trace(reflectedRay, step + 1);
	color = color + (rho * reflectedColor);
}
```

The variable MAX_STEPS is used to limit the depth of recursion to a pre-specified value. A sample output with reflections is shown in Fig. 2. Note that with the addition of reflections, the program takes significantly more time to render a scene.

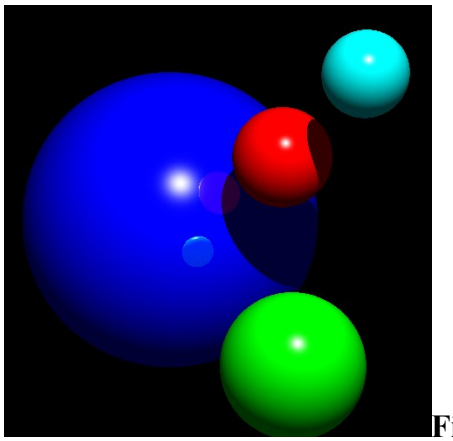


Fig. 2.

Plane.cpp

The `Plane` class is a subclass of `SceneObject`, and has a constructor that takes either three or four vertices that form the boundary vertices for a planar region. The header file `Plane.h` and the implementation file `Plane.cpp` for this class, are provided. Being a subclass of `SceneObject`, the `Plane` class must provide implementations for the functions `intersect()` and `normal()`.

Let us take a look at the functions in `Plane.cpp`. The `intersect()` function uses the following equation to find the point of intersection between the ray and the plane (Lec08-Slide 29), and returns the value of the ray parameter t at the point of intersection.

$$t = \frac{(A - p_0) \cdot \boldsymbol{n}}{\boldsymbol{d}.\boldsymbol{n}} \qquad (1)$$

The surface normal vector $\boldsymbol{n}$ of the plane is computed as $(C–B)\times(A–B)$ (Fig. 3), where A, B, C, D are the four vertices the plane defined in an anti-clockwise sense with respect to the required direction of the normal vector.

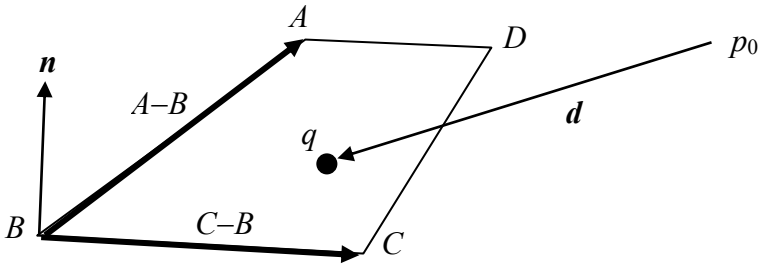


Fig. 3.

Even though the normal of a plane is independent of the point at which it is computed, we need to use the standard signature of the normal function (`normal(p)`) as specified in the `SceneObject` class.

The function `isInside(q)` implements a point inclusion test to determine whether a point *q* is inside the (triangular or quadrilateral) region on the plane's surface, given by the user specified (three or four) boundary points. This test uses a set of vectors computed using the boundary points, as shown on Lec08-Slide 31.

3. In the `initialize()` function of the ray tracer, create a pointer to a plane object as shown below. The four parameters define the vertices of the floor plane.

```
Plane *plane = new Plane (glm::vec3(-20., -15, -40),       //Point A
                          glm::vec3(20., -15, -40),       //Point B
                          glm::vec3(20., -15, -200),      //Point C
                          glm::vec3(-20., -15, -200));    //Point D
plane->setColor(glm::vec3(0.8, 0.8, 0));
```

and add this to the list of sceneObjects:

```
sceneObjects.push_back(plane);
```

Remember to add the statement `#include "Plane.h"` at the beginning of the program. You should get an output similar to that shown in Fig. 4:

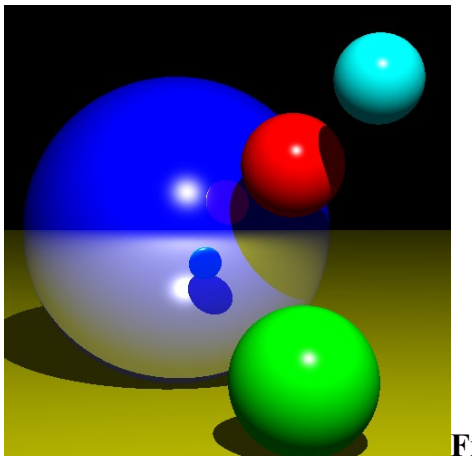


Fig. 4.

4. Please note that the plane shown in the above figure is an infinite plane. The intersect() method in `Plane.cpp` does not use the boundary vertices of the plane. We need to check if the point of intersection given by the ray parameter *t* computed by the intersect() method lies within the boundaries of the plane.

   Please edit the file "Plane.cpp" and replace the last statement (`return t;`) of the `intersect()` method with the following code:

```
glm::vec3 q = p0 + dir*t;    //Point of intersection
if( isInside(q) ) return t;    //Inside the plane
else return -1;                //Outside
```

   You should now get a finite plane bounded by the vertices, as shown in Fig. 5.

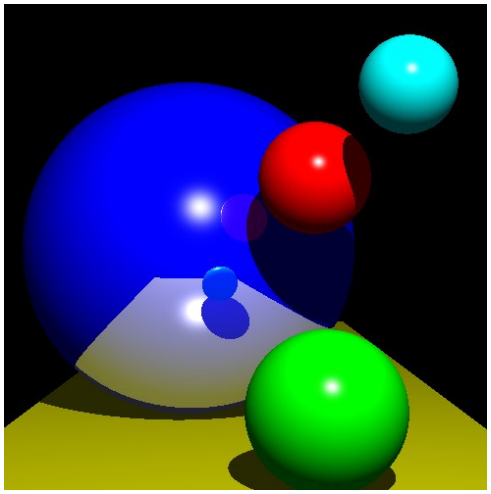


Fig. 5

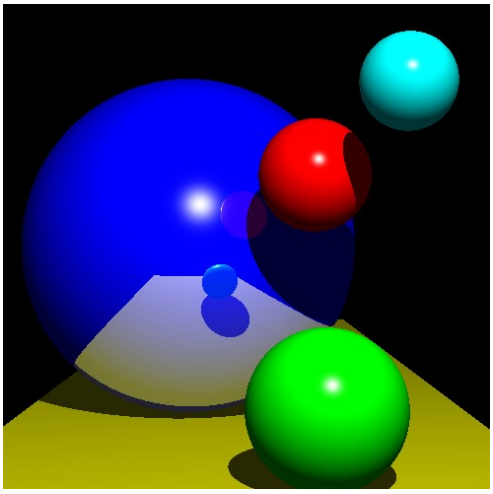


Fig. 6

Specular reflections from the floor plane can be disabled (Fig. 6) by setting the specularity property of the floor plane in the initialize() function:

```
plane->setSpecularity(false);
```

5. We will now generate a stripe pattern on the floor plane by modifying the plane's material colour value at each point. We will use the method shown in Fig. 7 to generate the pattern.

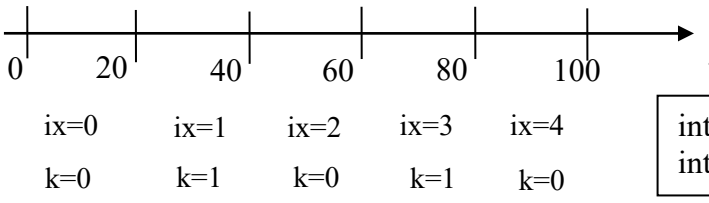


Fig. 7

A range of values along any coordinate axes (*x* in the above example) can be subdivided into sub-ranges of fixed widths (20 in the above example) by using an integer division of the coordinate value by the width. Using a modulo operation (2 in the above example) we can convert the range to a repeating pattern of integers corresponding to the number of colour values of a pattern.

We will use the above method to convert the z-coordinate values on the plane into values 0 or 1, and assign colour values to those points. Include the following code segment in the `trace()` function after the statement "obj = sceneObjects[ray.index]":

```
if (ray.index == 4)
{
    //Stripe pattern
    int stripeWidth = 5;
    int iz = (ray.hit.z) / stripeWidth;
    int k = iz % 2;                    //2 colors
    if (k == 0) color = glm::vec3(0, 1, 0);
    else color = glm::vec3(1, 1, 0.5);
    obj->setColor(color);
}
```

The above code assumes that the index of the plane is 4. Please use the correct index from the initialize() function, based on the position of the plane in the sceneObjects list. The code modifies the colour of the plane specified in Section 3 above (page 3) with two alternating colours. The output with the stripe pattern is shown in Fig. 8.

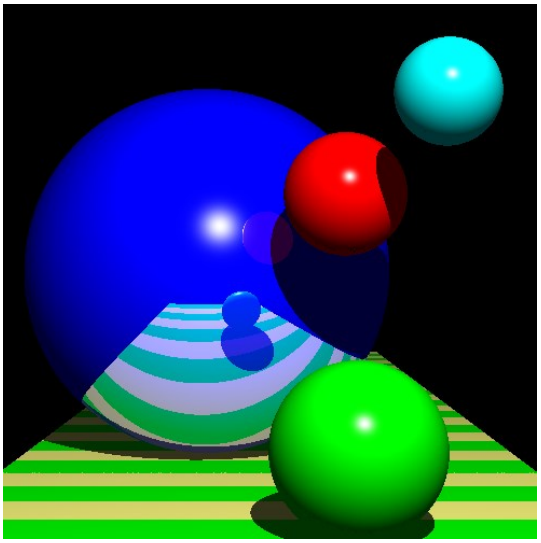
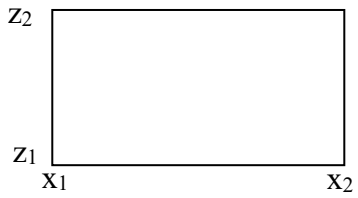


Fig. 8

6. Texture mapping requires a mapping of the coordinates (x, y, z) at the point of intersection (ray.hit) to a pair of texture coordinate values in the range 0-1. Regions on a two-dimensional plane can be easily mapped to this range using a simple linear transformation as shown in Fig. 9 below (Lec08-Slide35) .



texcoords = (ray.hit.x – x1)/(x2–x1);
texcoordt = (ray.hit.z – z1)/(z2–z1);

Fig. 9

The files "TextureBMP.h", and "TextureBMP.cpp" required for loading a texture in bitmap (.bmp; 24 bits per pixel) format, are provided. Add the statement `#include "TextureBMP.h"` at the beginning of the program. Also, add the declaration statement for a texture object at the beginning of the program.

```
TextureBMP texture;
```

In the `initialize()` function, include the statement

```
texture = TextureBMP("Butterfly.bmp");
```

On page 4, we had provided the code for modifying the colour values of the plane using an "if" statement to generate a stripe pattern. Please modify this code block as follows:

```
if (ray.index == 4)
{
    //Stripe pattern
    ...   //code given on page 4
    ...
    obj->setColor(color);

    //Add code for texture mapping here
    float texcoords =
    float texcoordt =
    if(texcoords > 0 && texcoords < 1 &&
      texcoordt > 0 && texcoordt < 1)
    {
      color=texture.getColorAt(texcoords, texcoordt);
      obj->setColor(color);
    }

 }
```

In the space shown in the code snippet above, add statements for computing the texture coordinates as given in Fig. 9, for a region of the floor plane shown in Fig. 10. The expected output of the ray tracer is shown in Fig. 11.

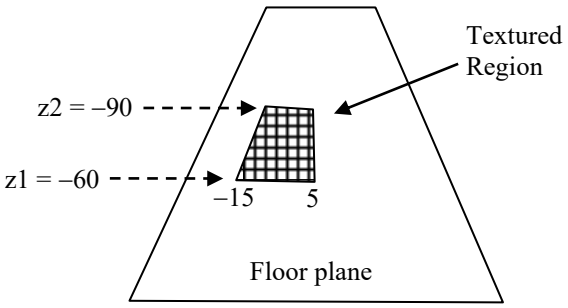


Fig. 10

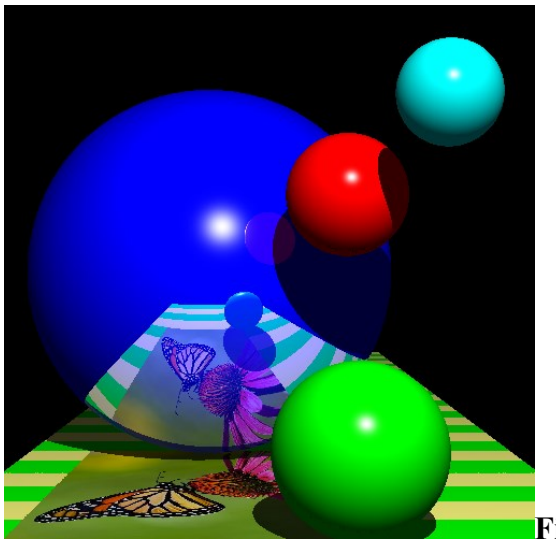


Fig. 11.

7. Please go through the code in `Sphere.cpp` and familiarize with the methods used for computing normal vectors and points of intersection. Please compare the implementations with the equations given on Slide [8]-30.

Ref:

Lec08: COSC363 Lecture slides "Lec08 Ray Tracing".

II. Quizzes

Quiz-06 closes this week: 11:55pm**, 5-May-2023**.

Quiz-07 closes on 12-May-2023.