CSCE 441 - Computer Graphics

Programming Assignment 6 - Part 1 & 2

1 Goal

The goal of this assignment is to write a ray tracer.

Starter Code

The starter code can be downloaded from the course drive.

Part 1

Task 1

Download the code and run it. You should be able to see a black screen. Make sure you write your name in the appropriate place in the code, so it shows up at the top of the window.

Here is a brief explanation of the starter code:

• There are two folders in the package; one includes all the source files and the other contains an obj file.

• The main function in “**main.cpp**” first calls the Init function.

• The **Init** function performs the following:

– First, it initializes the window.

– Then it creates an instance of the Scene class. This class is a container that holds all the scene information including the shapes and light sources.

– Next, it creates an instance of the Camera class. Currently, only the resolution of the image is passed to the camera. However, you need to modify this and pass all the necessary information.

– Then we call a function of the Camera class to take a picture of the scene using ray tracing. This is where the main loop of the ray tracer should be implemented in. This function takes the scene as the input and takes a picture of it. The picture is stored in a private variable called **renderedImage**.

– We then have the camera return the rendered image and set it to the **frameBuffer** through **memcpy**.

Note that you do not need to follow this starter code structure. If you want to write your ray tracer differently, feel free to do so. The starter code is just provided to help you.

Task 2

In this part, you will be implementing a ray tracer. Your ray tracer should support the following:

• **Shadows:** You should compute shadow rays to all the light sources.

• **Reflections:** Your code should recursively call the reflection ray, i.e., you shouldn't hard code the reflection calls. Set the level of recursion to at least 4. DO NOT HARD CODE FOUR REFLECTION CALLS.

• **Lighting:** You should compute basic lighting (ambient, diffuse, specular) as explained in the shading lectures.

You are expected to demonstrate your ray tracer in the following scene. Here are the camera properties:

• Eye = (0.0, 0.0, 7.0)

• Look at = (0.0, 0.0, 0.0)

• Up = (0.0, 1.0, 0.0)

• FovY = 45

• focal distance = 1

• Width Res = 1200

• Height Res = 800

Your scene should include two planes, four spheres, and two light sources as follows:

• Shapes

  – Sphere 1

    ∗ Position = (-1.0, -0.7, 3.0)

    ∗ Radius = 0.3

    ∗ ka = (0.1, 0.1, 0.1)

    ∗ kd = (0.2, 1.0, 0.2)

    ∗ ks = (1.0, 1.0, 1.0)

    ∗ km = (0.0, 0.0, 0.0)

    ∗ n = 100.0

– Sphere 2

  ∗ Position = (1.0, -0.5, 3.0)

  ∗ Radius = 0.5

  ∗ ka = (0.1, 0.1, 0.1)

  ∗ kd = (0.0, 0.0, 1.0)

  ∗ ks = (1.0, 1.0, 1.0)

  ∗ km = (0.0, 0.0, 0.0)

  ∗ n = 10.0

– Sphere 3 (reflective)

  ∗ Position = (-1.0, 0.0, -0.0)

  ∗ Radius = 1.0

  ∗ ka = (0.0, 0.0, 0.0)

  ∗ kd = (0.0, 0.0, 0.0)

  ∗ ks = (0.0, 0.0, 0.0)

  ∗ km = (1.0, 1.0, 1.0)

  ∗ n = 0.0

– Sphere 4 (reflective)

  ∗ Position = (1.0, 0.0, -1.0)

  ∗ Radius = 1.0

  ∗ ka = (0.0, 0.0, 0.0)

  ∗ kd = (0.0, 0.0, 0.0)

  ∗ ks = (0.0, 0.0, 0.0)

  ∗ km = (0.8, 0.8, 0.8)

  ∗ n = 0.0

  - Plane 1
    * Center = (0.0, -1.0, 0.0)
    * Normal = (0.0, 1.0, 0.0)
    * ka = (0.1, 0.1, 0.1)
    * kd = (1.0, 1.0, 1.0)
    * ks = (0.0, 0.0, 0.0)
    * km = (0.0, 0.0, 0.0)
    * n = 0.0
  - Plane 2
    * Center = (0.0, 0.0, -3.0)
    * Normal = (0.0, 0.0, 1.0)
    * ka = (0.1, 0.1, 0.1)
    * kd = (1.0, 1.0, 1.0)
    * ks = (0.0, 0.0, 0.0)
    * km = (0.0, 0.0, 0.0)
    * n = 0.0
- Light
  - light 1
    * Position = (0.0, 3.0, -2.0)
    * Color = (0.2, 0.2, 0.2)
  - light 2
    * Position = (-2.0, 1.0, 4.0)
    * Color = (0.5, 0.5, 0.5)

Your final rendering should look exactly like the image in Fig. 1.

Steps

Here are the steps to take to properly implement the ray tracer:

- Fill in the Camera class to have the necessary member variables. Currently, the instance of the camera class in the Init function of **“main.cpp”** is created with only width and height information.

You should set it up using the camera parameters provided above.

• Set up the scene with the information provided above by filling in the Shape and Light vectors in the Scene class. This can be done in the constructor of the Scene class.

To do this, you need to first add appropriate member variables to the Shape, Sphere, Plane, and Light classes. For example, **Light class** should at least have the position and color as member

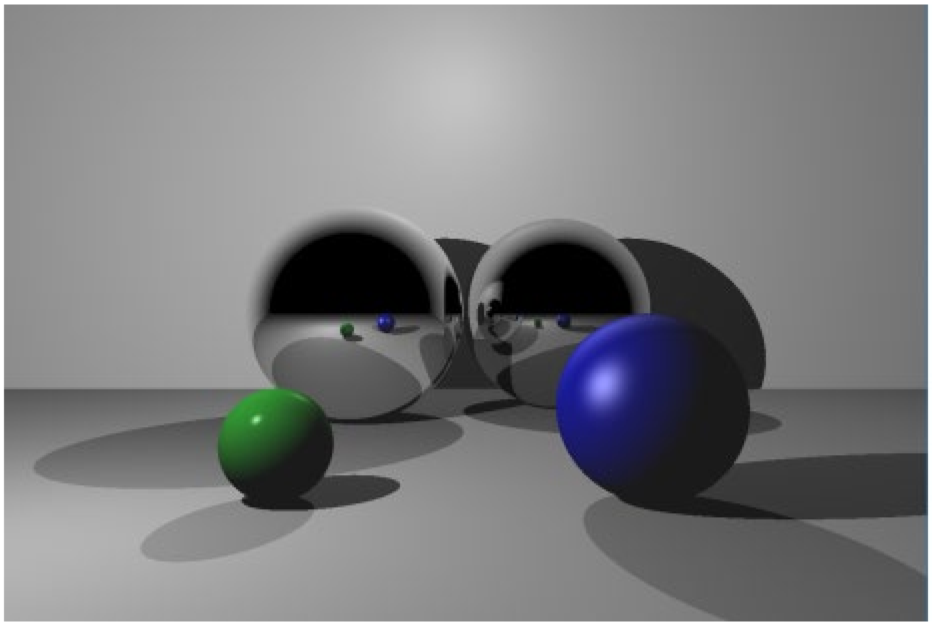
Figure 1: The result of rendering the scene.

variables. Note that, Shape is the base class for the Sphere and the Plane and has access to all their member variables.

Once all the classes are properly set up, you can create instances of **Sphere**, **Plane, and Light classes** based on the information provided above and then push them into the appropriate vector.

• Next, you should fill in the **TakePicture** function of the camera class. In this function, you should loop over all the pixels. For each pixel, you first create a ray pointing from the camera to the pixel.

You then call a function to get the color of the ray. Most of the computations are done inside this recursive function. The returned color of the ray is then used to set the color of the **renderedImage** array at that pixel.

Suggestions

Since the ray tracer has several features, it would be better if you build your system incrementally. For example, you can do the following (see Fig. 2 for the outcome of each step):

1. Render a sphere without any shadows or recursive calls.

2. Add a plane to the scene.

3. Shoot shadow rays to be able to render shadows.

4. Implement the recursive reflection call and add a reflective sphere to the scene to test it.

Task 3

In this part, you will be rendering the bunny.

This requires implementing ray triangle intersection. Also since the bunny is a triangle mesh consisting of a large number of triangles, checking the intersection against every triangle for every pixel is going to be time-consuming.

So we speed up the rendering by computing a bounding box around the bunny. Before intersecting against every triangle, we first check the intersection against the box. If the ray does not hit the box, we won't be checking the intersection with the triangles.

We will be using the same camera and lights as task 2. Use the following material for the bunny:

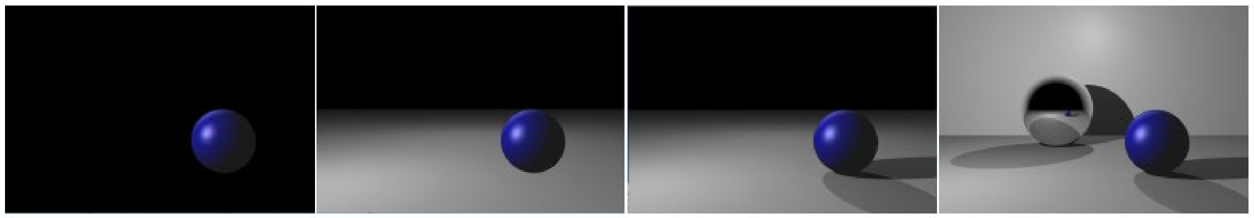

Figure 2: From left to right the outcome of steps 1, 2, 3, and 4 in Sec. 3.2.2.

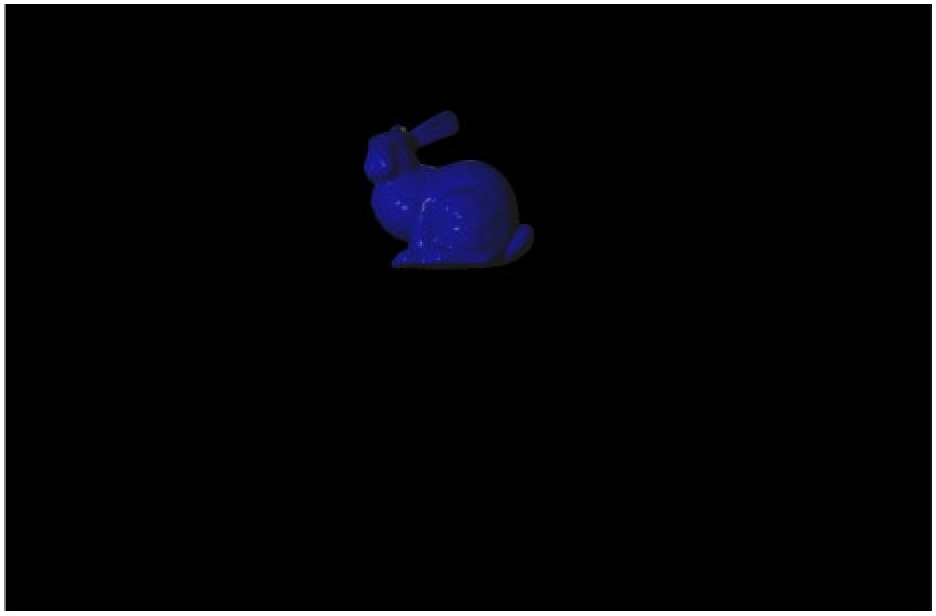
Figure 3: The expected output of Task 3.

- ka = (0.1, 0.1, 0.1)
- kd = (0.0, 0.0, 1.0)
- ks = (1.0, 1.0, 0.5)
- km = (0.0, 0.0, 0.0)
- n = 100.0

Test your code in the release mode (the performance is much better than the debug mode). You should be able to see a bunny like the one shown in Fig. 3. Have a flag in your code to determine the scene. Using the flag, you should be able to switch between the bunny and the other scene in Task 2.

Steps

Here are the steps to take to properly implement this part:

- You need to create a triangle class, similar to the plane and sphere classes. The triangle class should have appropriate variables to hold the parameters of a triangle. It should also have functions for computing ray triangle intersection and computing the normal. For ray triangle intersection, you will implement Muller Trombore approach. For normal computation, you basically have to interpolate the normal at the three vertices to obtain the normal at the intersected point. You will use barycentric coordinates for this. Note that these coordinates are already computed with Muller Trombore approach.

• You should load the triangle mesh in the obj file and write the result into a vector of type triangle. You can use the function provided in the starter code of assignment 4 to do this.

• Now you should implement the bounding box approach for faster ray tracing. The easiest way to do this is to create a new class.

Let's call this class BVH. This class should be inherited from the shape class. The idea is that this class will hold all the triangles, compute their bounding box, and will have an intersect and get normal function, similar to all the other primitives. The intersect function for this class, first intersects a ray with the bounding box. If the ray intersects with the box, it will then intersect it with every triangle; otherwise, the ray does not hit the bunny.

• You'll push this class as an element in your shape vector. Because this class has the intersect and get normal functions, the rest of the code will know how to handle everything.

Part 2

Task 1

In this task, you will implement a bounding volume hierarchy, as discussed in class.

For this, you should build upon the previous BVH class. Instead of having a single box that stores all the triangles, you are going to have a tree structure, where each node is going to have a bounding box, and the leaves are going to hold the triangles.

There are two main steps to this: reconstructing the BVH and intersecting the BVH.

You should test your code against the version with a single bounding box. You should be able to see at least an order of magnitude speed up.

Steps

• The first step is to create the BVH.

Your BVH should have the necessary variables to build a tree. Basically, at every level, you divide the triangles into two groups and assign them to the left and right branches of the tree. When you create a new node, you should immediately create the bounding box and store it in the node. This is going to be used later during the ray BVH intersection. A few notes regarding this process:

– The division of the triangles into two groups happens based on their spatial position. Basically, we want half of the triangles that are spatially close to each other in one group and the rest in the other group. Since we are working in a 3D space, the division at each depth (level) is done based on one axis. For example, we use the "x" axis for the first level, "y" for the second, "z" for the third, "x" for the fourth, and so on.

– At every level, we would like to first sort the triangles based on their spatial position along the axis of interest (e.g., "x" in the first level). Since triangles have three vertices, we can first compute the average of the three vertices and perform the sorting on the axis of interest of the averaged position. Once the triangles are sorted, we assign half of them to the left node, and the other half to the right node. For example, if we have 100 triangles, after sorting, we assign the first 50 to the left and the last 50 to the right node.

– For sorting, you can use an appropriate function from the standard library (no need to implement sorting yourself).

– You continue dividing the tree until you reach a certain criteria. In our case, the criteria is to have the number of triangles in the node be smaller than a pre-defined number. This pre-defined number should be a parameter in your code. For testing, you set this parameter to 3.

– Reorganizing the whole triangle vector during sorting could be computationally expensive. Instead, you can assign an index to each triangle in the triangle vector, e.g., 1 to 100 for the case with 100 triangles. Then during sorting, you only move the indices around. In the end, you will have the triangle vector (untouched) and reorganized index array (based on the sorting). Each node will have a beginning and end values that points to the index array and basically determine which triangles are inside the node.

• Next, you will be implementing the intersection function. This function recursively intersects the ray with the bounding box until it reaches the leaf node. At that point, it will intersect the ray with every triangle in the leaf node.

Deliverables

Please follow the instructions below or you may lose some points:

• You should include a README.txt file that includes the parts that you were not able to implement, any extra part that you have implemented, or anything else that is notable. Note that the README file should be in "txt" file format and you should place it in the root folder next to the "src" folder and "CMakeLists.txt" file.

• Your submission should contain folders "src" as well as "CMakeLists.txt" file. DO NOT include the "build" folder.

• Zip up the whole package and call it "Firstname Lastname.zip". Note that the zip file should extract into a folder named "Firstname Lastname". So you should first put your package in a folder called "Firstname Lastname" and then zip it up. The zip file structure should be exactly as follows:

"firstname lastname.zip"

– "firstname lastname"

∗ "src"

∗ README.txt

• Submit the package through Canvas.

Ruberic

Part1 - Total credit: [100 points]

[5 points] - Camera set up and primary rays generated

[5 points] - Plane is intersected correctly

[5 points] - Sphere is intersected correctly

[5 points] - Depth test is done properly

[5 points] - Basic local lighting is computed

[5 points] - Normals of spheres and planes computed correctly

[5 points] - Light, view, and reflection vectors are computed correctly

[5 points] - Shading is computed using Phong model

[10 points] - Shadow rays are calculated and incorporated

[10 points] - Reflection is supported

[5 points] - Reflection is computed recursively

[05 points] - Color of reflected vector is combined with local Phong shading

[05 points] - Triangle mesh is loaded properly

[10 points] - Triangle is intersected correctly

[5 points] - Triangle normal is computed properly

[5 points] - Bounding box is computed correctly

[5 points] - Bounding box is intersected correctly

Part2 - Total credit: [100 points]

[50 points] - Create bounding volume hierarchicy (BVH)

[45 points] - Correctly intersect with BVH

[05 points] - Return the normal correctly