**Lab 2 Report**

*Name: Bingjing Xu Major:DMT Contact:* [*bingjing@kth.se*](mailto:bingjing@kth.se)

**1.0 Before we do the experiment…**

In this lab, we want to implement a Ray tracer, which draws images of 3D scenes by tracing the light rays reaching the simulated camera.

In order to realize the final goal, we still have to build a new project with SDL framework. Next step, we import the data of test model---‘Cornell Box’ into our own project. And we named the data file ‘TestModel.h’. After all those steps finish, we can start to write our own code.

* 1. **Ray tracing with Triangle Surface Representation**

Initially, we assume that the camera is aligned with the coordinate system of the model with x-axis pointing to the right and y-axis pointing down and z-axis pointing forward into the image. And we should set up the position of the pinhole camera:

vec3 cameraPos(x,y,z); // we should test different values to get the best positon of camera.

Also, we want to confirm the direction where the light reaching every pixel comes from. Therefore, we use a formula to describe the relationship between the direction and the pixel’s position:

vec3 direction (x - SCREEN\_WIDTH / 2, y - SCREEN\_HEIGHT / 2, focalLength);

To get the effect of ray tracing, now we have to calculate the closest intersection for every pixel. If there is a closest intersection, we save the index of triangle in the memory, then draw the corresponding color on this pixel.

However, in the process, there are some details we should pay attention to:

For the equation (7), (8), (9), (11), there are some mistakes about the range of these variables. So we have to correct them as following to get the correct scene in the end.

**u>=0; (7)**

**v>=0; (8)**

**u+v<=1 (9)**

**t>0 (11)**

Hint: If you use the equation in the paper, the final effect of your picture may have unknown black line on the border of every triangle.

And we get the scene after running our code as figure 1 shows:



figure 1 Cornell box after ray tracing

* 1. **Moving the Camera**

Since we have a simple visualization of our 3D scene, we are interested in moving the camera around.

Now we want to implement such functions:

If the user presses left and right arrow keys, the camera will rotate around y-axis;

While the user presses up and down arrow keys, it will rotate around x-axis;

We put the piece of code into update();

if( keystate[SDLK\_UP] )

{

// Move camera forward

xaw = 5.0f\*M\_PI/(-180.0f);

right=glm::vec3(1,0,0);

down= glm::vec3(0,cos(xaw),-sin(xaw));

forward=glm::vec3(0,sin(xaw),cos(xaw));

R=glm::mat3(right,down,forward);

cameraPos=R\*cameraPos;

}

if( keystate[SDLK\_DOWN] )

{

// Move camera backward

xaw = 5.0f\*M\_PI/180.0f;

right=glm::vec3(1,0,0);

down= glm::vec3(0,cos(xaw),-sin(xaw));

forward=glm::vec3(0,sin(xaw),cos(xaw));

R=glm::mat3(right,down,forward);

cameraPos=R\*cameraPos;

}

if( keystate[SDLK\_LEFT] )

{

// Move camera to the left

yaw = 5.0f\*M\_PI/180.0f;

right=glm::vec3(cos(yaw),0,sin(yaw));

down= glm::vec3(0,1,0);

forward=glm::vec3(-sin(yaw),0,cos(yaw));

R=glm::mat3(right,down,forward);

cameraPos=R\*cameraPos;

}

if( keystate[SDLK\_RIGHT] )

{

// Move camera to the right

yaw = 5.0f\*M\_PI/(-180.0f);

right=glm::vec3(cos(yaw),0,sin(yaw));

down= glm::vec3(0,1,0);

forward=glm::vec3(-sin(yaw),0,cos(yaw));

R=glm::mat3(right,down,forward);

cameraPos=R\*cameraPos;

}

if we still keep our picture size as 500\*500, then the responding time for every update would be too long. So as to decrease the time and test the code, we change the size of picture for example as 100\*100.

const int SCREEN\_WIDTH = 100;

const int SCREEN\_HEIGHT = 100;

* 1. **Illumination**
     1. Direct Illumination

Following the instructions on the the lab2 paper, we can easily get a result as follows:

Notice that there is no overwrite division’/’ for 3D vectors in GLM library, so be careful using ‘/’ when you write your algorithm.



figure 2 direct light source

However, this is not what our eyes perceived in the real world. Thus we need a diffuse reflection model for this.

So we implement such piece of code in Draw();

vec3 colorx = Light(closestintersection);

int index = closestintersection.triangleIndex;

vec3 colory = triangles[index].color;

//add the diffuse effect

vec3 color = colorx \* colory;

The result after adding it to skeleton.cpp:



figure 3 diffuse reflection

* + 1. Moving the light

The principle is just the same as that of moving the camera, so we skip the details.

Core code:

if( keystate[SDLK\_w] )

{

glm::vec3 forward(0,0,-0.1);

if(lightPos.z>-1.0)

lightPos += forward;

}

if( keystate[SDLK\_s] )

{

glm::vec3 back(0,0,0.1);

if(lightPos.z<1.0)

lightPos += back;

}

if( keystate[SDLK\_a] )

{

glm::vec3 left(-0.1,0,0);

if(lightPos.x> -1.0)

lightPos += left;

}

if( keystate[SDLK\_d] )

{

glm::vec3 right(0.1,0,0);

if(lightPos.x<1.0)

lightPos += right;

}

if( keystate[SDLK\_q] )

{

glm::vec3 up(0,-0.1,0);

if(lightPos.y> -1.0)

lightPos += up;

}

if( keystate[SDLK\_e] )

{

glm::vec3 down(0,0.1,0);

if(lightPos.y< 1.0)

lightPos += down;

}

* + 1. Direct Shadow

Assume that there is a closer surface between direct light source and the intersected point, shadow will form because the light was blocked by the closer surface.

So we need implement closest intersection again to see if there exist another surface be direct light source and the intersected point. If it does, then give a black color to the point.

*The code is as follows:*

// direct shadow

Intersection closestintersection2;

vector<vec3> solutions2(30);

CalculateIntersaction(triangles, in.position, Radium, solutions2);

if (ClosestIntersection(solutions2 , closestintersection2, in.triangleIndex)==true)

{

if (closestintersection2.distance < RR )

directlight= glm::vec3(0,0,0);

}



figure 4 direct shadow

* + 1. Indirect Illumination

Finally, we only need the add indirect illumination to our 3D scene. Since the indirect illumination is so expensive to implement so we just use an approximate model to describe indirect illumination.

Let N be the power per surface area of this constant indirect illumination:

N = 0.5f\*vec3(1,1,1);

So the total incident illumination should be

Direct Illumination + Indirect Illumination;

After implementing this, we will get the final effect of our Cornell Box scene:



figure 5 final result