Raytracer Report

Aikaterini Baousi

University of Bristol, COMS30115
ab16651@my.bristol.ac.uk

May 11, 2017

Abstract

In this report the implementation and the extensions of the raytracer project are thoroughly described. The extensions focus on providing solutions to decrease the time complexity of the project (eg. Cramer's rule,Parallel programming) and adding extra features in this implementation (e.g dynamic illumination, soft shadows, anti aliasing). The code can be executed using the make command provided by the Makefile in the initial folder.

I. Implementation

In this project the surfaces that had to be represented were stored as a set of triangles. Rays were sent out for each pixel of the camera image and the pixel's corresponding ray direction should be calculated.

The first step of this implementation was to calculate the intersections between triangles and rays. These intersections have to be both minimal and within the intersecting triangle's surface. The color of the closest intersected triangle's surface is considered to be the color of the pixel.

In order to increase the camera's mobility, Update() function was created. Thanks to this function the user has the ability to change the camera's position forwards and backwards by pressing the equivalent key(up & down). Furthermore, the user has the ability to rotate the camera to the left and right via the keyboard.

It was also necessary to incorporate a light source in this implementation. The direct light from this light source that is reaching the intersections of rays and triangles is calculated in DirectLight() function. The power of the light source is perceived as a sphere having its centre at the source's position and its radius equivalent to the distance of the surface from the light source. In addition, whether or not a surface receives direct light depends on the projection of the unit vector describing the direction from the surface to the light source to the surface's normal unit vector(i.e the dot product of the two vectors). If the two vectors have an angle of more than 90 degrees then the surface does not get direct light.Ultimately, four keys were included at this point in the Update() function in order to manipulate the lights movement forwards, backwards, right and left. These keys were W,S,A and D respectively.

The DirectLight() function was extended even more by casting more light in order to create shadows.ClosestIntersection() was also used at this point in order to determine for each point whether there is a surface blocking the light or not. In that case the pixel get zero value.Finaly, indirectLight vector was used in Draw() function to model the "bouncing" of light from the source via the surfaces to the camera.

II. Extensions

The ways the aforementioned implementation got extended are the following:

i. Cramer's Rule

Cramer's rule is a formula for the solution of a system of linear equations with as many equations as unknowns. The solutions are expressed in terms of determinants of the coefficient matrix (i.e A) and of matrices obtained from it by replacing one column by the vector of right hand sides of equations (i.e b). In more detail the Cramer's rule can be expressed as follows:

Let the linear equation system be described as $Ax = b$, where

$$A = \begin{bmatrix} a1 & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \end{bmatrix}$$

$$x = \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix}$$

$$b = \begin{bmatrix} d1 \\ d2 \\ d3 \end{bmatrix}.$$

Also let

$$A_x = \begin{bmatrix} d1 & b1 & c1 \\ d2 & b2 & c2 \\ d3 & b3 & c3 \end{bmatrix}$$

$$A_y = \begin{bmatrix} a1 & d1 & c1 \\ a2 & d2 & c2 \\ a3 & d3 & c3 \end{bmatrix}$$

$$A_z = \begin{bmatrix} a1 & b1 & d1 \\ a2 & b2 & d2 \\ a3 & b3 & d3 \end{bmatrix}$$

Then the solution of the linear system will be the following:

$$x_1 = \frac{\det A_x}{\det A}\ ,\ x_2 = \frac{\det A_y}{\det A}, x_3 = \frac{\det A_z}{\det A}$$

Cramer's rule can reduce significantly the complexity of calculating the inverse matrix in order to solve systems of linear equations such as the ones that occur when computing the closest intersection of a ray with a triangle. This extension will help in solving the bottleneck of our implementation.

ii. Anti Aliasing

Antialiasing focuses on removing the staircase effect from the computed shapes.Aliasing occurs while trying to depict continuous smooth curves and lines via small squares such as pixels.A possible solution to this problem is to use multiple samples of each pixels surrounding area, average them and fix the "missing" color information from the lines of the drawn shapes. In this implementation an extra key (key 1) is added in order to give the opportunity to the user to resolve the problem of aliasing that happens in this project as shown in Figure 2. When pressing this button the program checks in Draw() function the surrounding area of the pixel which value is computed and averages the values of the pixels. Unfortunately this procedure increases the computational complexity of the implementation and the program becomes slower.

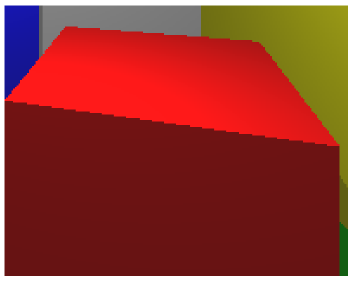

Figure 1: *Jaggies occured in this implementation.*

iii. Soft Shadows

Most generated shadows in this implementation tend to have hard edges (i.e hard shadows). In order to smooth this problem , an approach similar to the one mentioned in the anti aliasing section was taken. Once the user presses key 2 the soft shadowing effect is enabled and the shadows casted in this implementation get smoother. The existence of soft shadows is achieved by increasing the number of light sources in the program.The soft shadowing effect extended DirectLight() function, which is responsible for the illumination of the image by casting light, from even more positions . Once again the computational complexity is increase and that fact slows down the program.

iv. Further illumination of the scene

Another extension of this project was to give the user the opportunity to dynamically increase and decrease the number of casted lights in the image.Function Illuminate() was created in that scope and is responsible for manipulating the light object created by class Light.This function increases the number of casted light by a small number every time the user presses "l" key. The number of lights decreases when the user presses "b" key.

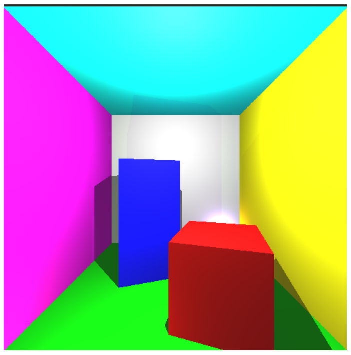

Figure 2: *Illuminated picture.*

v. Parallel Programing

penMP is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism . It uses a thread-based parallelism model in which a master thread creates a team of threads in order to execute a specific region of a program. After the team threads complete the statements in the parallel region synchronize and terminate, only leaving the master thread active. For the purpose of this project, OpenMP was used to parallelize the intensive loops of the functions.

In the main function the maximum number of threads provided by the machine is used. All the workload created by the for loops of this implementation is spread across the number of the created threads. In addition "nowait" clause is used because this implemention is embarassingly parallel because the procedures performed in each iteration are independent from each other.The overall time complexity of the implementation is decreased by this implementation.