Game Engines

MGAE-E2013

IT-University of Copenhagen

Jacob Claudius Grooss, jcgr@itu.dk

May 22, 2013

1 Introducton

This report has been written as part of a project on the Game Engines E2013 (MGAE-E2013) course on the Games course at the IT-University of Copenhagen.

For the project, I have written a program in C# (with the use of Windows Forms) that can render simple wireframe models. For the calculations, I followed the the "Overview of transforms used in rendering"[1] tutorial by Mark Nelson. The program has been written in C#. I have worked, and shared ideas, with Jakob Melnyk (jmel).

2 Features

In this section I will describe the two main features of the program: Rendering and camera movement.

2.1 Rendering

The program utilises a list of triangles created from vertices for rendering. The program calculates the location of each triangle on the screen based on the camera's variables and draws them to the screen.

2.2 Camera Movement

The camera can be moved around to display the wireframe model from different angles and positions. The controls for moving it are:

- 'a' and 'd' moves the camera on the x-axis.
- 'w' and 's' moves the camera on the y-axis.
- 'q' and 'e' moves the camera on the z-axis.
- 'j' and 'l' moves the look point of the camera on the x-axis.
- 'i' and 'k' moves the look point of the camera on the y-axis.
- 'u' and 'o' moves the look point of the camera on the z-axis.

3 Overview

In this section I will be giving an overview of the program. I will cover the classes of the program, how the rendering and math works and in which parts of the program the math is.

3.1 Classes

The program contains seven classes. They are `Matrix`, `Vertex`, `Triangle`, `Vector`, `Camera`, `WireframeRenderer` and `Program`.

Matrix, Vertex, Triangle and Vector

These four classes are computer representations of their mathematical versions.

- `Matrix` represents a matrix of varying sizes and holds floating-point values.

[1] https://blog.itu.dk/MGAE-E2013/files/2013/09/transforms.pdf

- `Vertex` represents vertices, but has four coordinate points (x, y, z and w). The fourth, w, is a dummy value so it matches the four-dimensional matrices that are used. It also has a point (an x- and y-value) that represents its position on the screen.
- `Triangle` represents a triangle in 3D space and contains the three vertices that determines the triangle's corners.
- `Vector` represents a 3D vector has three coordinate points (x, y and z).

Camera

The `Camera` class represents the camera in the model view space. It is what allows the user to see the wireframe models. It contains variables that determines its position, where it is looking, its height, width, field of view and aspect ratio.

WireframeRenderer and Program

The `Program` class starts the application. It creates a new instance of the `WireframeRenderer`, which is a subtype of WindowsForms. `WireframeRenderer` creates a window for drawing, initializes the camera and is responsible for accepting user input (and handling it correctly), updating the triangles and drawing to the screen.

3.2 Rendering

When the program is started, `WireframeRenderer` loads four `Triangle`s. These triangles represent a model of a pyramid, which is what the program renders.

When the model is to be drawn to the screen, it tells the camera to calculate its transforms. After the calculations, each triangle tells its vertices to update their screenpoint, which they do through the use of the camera transformations. Lastly, the triangles are drawn to the screen.

3.3 Math

I decided to move the calculations to the classes they were related to.

The `Matrix` class contains a simple method for multiplying matrices with each other.

The `Camera` class contains the methods needed to calculate the three camera transforms described in the tutorial PDF (location-, look- and perspective-transforms). It also contains a method for calculating the combination of these three transformations. It was implemented to make it less code-heavy to get the combined transform, for example for use in updating the screenpoints of the vertices.

The calculation of the screenpoints of the vertices happens across multiple classes. `WireframeRenderer` iterates over all triangles in the model, and tell each to update its vertices. The triangle then tells each of its own vertices to update their screenpoint. The vertices themselves handles the actual calculation of screenpoints.

It should be noted that I have omitted a part of the pseudocode that describes how to calculate the screenpoints. I decided not to ignore a point if it was outside the screen. The reason is, that while the point might not be on the screen, the line between points may. Skipping a point will therefore result in missing lines of the model.

4 Problems

In this section I will describe some of the problems of the program.

One problem arises then the camera is moved through the model and no longer looks at it. The results for the screenpoints will overflow and cause the program to crash. This could be avoided by updating only the triangles the camera is looking at.

Another problem happens when the camera is turned. At some point, the transforms will end up putting one of the closer to the oppesite side of the screen from where the rest of the model is, which means the triangle will stretch across the screen. A check to see where the rest of the triangle is would help in getting rid of this problem.

A smaller problem is performance-wise. Each triangle has three vertices that needs to have their screenpoints calculated. But some of these vertices are the same for the triangles (for example the top of the pyramid). This means that some vertices may be updated multiple times, resulting in a lot of extra calculations. This could be avoided by letting each triangle hold a reference to the vertices, instead of having them hold instances of the same vertex.

5 Conclusion

The program I have written allows for rendering of simple wireframe models. Furthermore, the program allows the user to move the camera around in the model view space and inspect the model from different angles. However, large models/scenes would be problematic to render due to the lack of optimization during the calculations.

5.1 Future works

There are plenty of things that could be changed in the future.

As it is now, there are a lot of unnecessary calculations, and there are cases where the calculations result in the program crashing. This could be avoided by proper optimizing of the calculations and checking if the calculations are within the expected scope.

The camera could also use some changing. The main thing to focus on would be to allow the mouse to control the camera, instead of having to use the keyboard. The keyboard is a clunky way of controlling the camera, and it not very precise. Changing how the camera rotates would also be part of this work.

6 Appendix

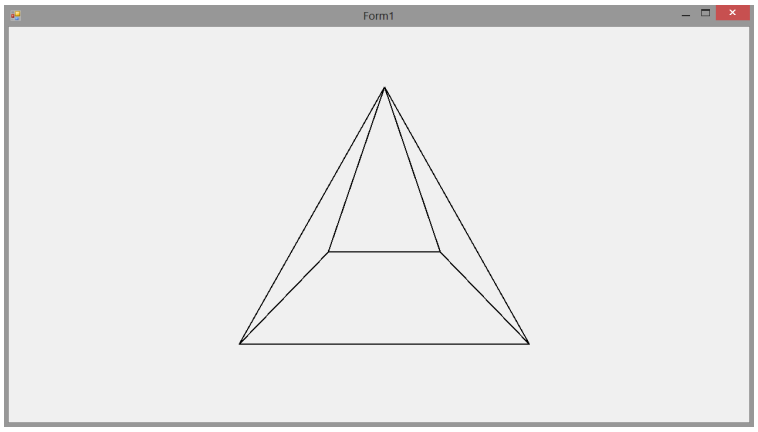

Figure 1: The program when it starts.

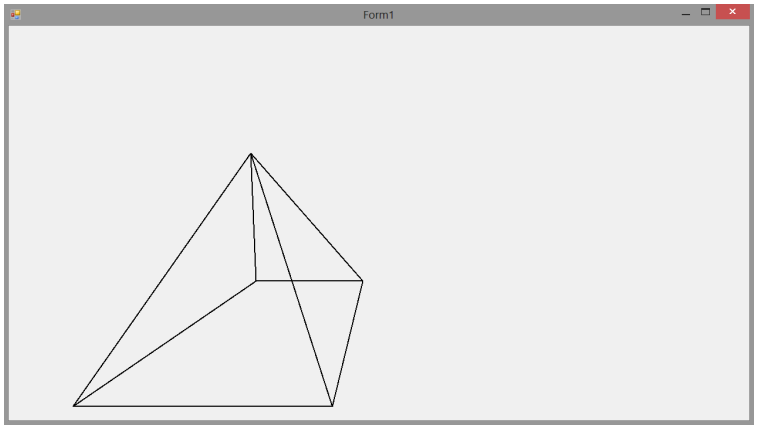

Figure 2: The camera has been moved.

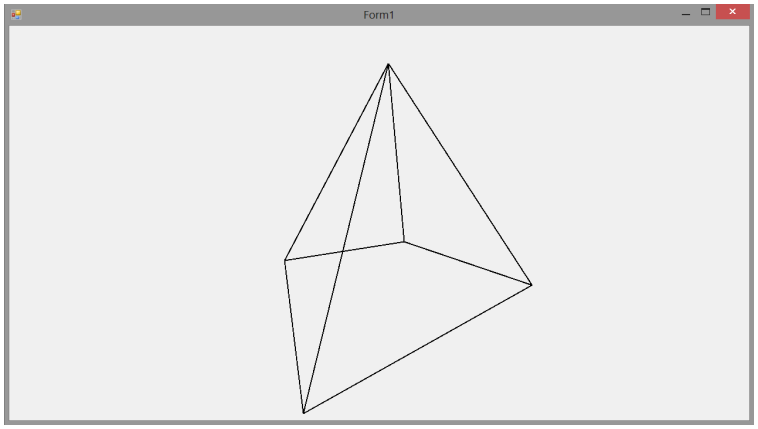

Figure 3: The camera has been moved and turned.