3D Wireframe Renderer

Game Engines E2013

Jakob Melnyk, jmel@itu.dk

October 22, 2013

1 Introduction

This report is the result of a smaller project to implement a 3D wireframe renderer in the Game Engines, E2013 course at IT-University of Copenhagen.

The project had the following constraints: Transforms and rendering must be implemented in software without using DirectX or OpenGL functions. The use of an existing line-drawing API and an existing library for basic matrix math was allowed.

The wireframe renderer is implemented in C# (using Windows Forms) based on the transforms and pseudo-code from the "Overview of transforms used in rendering"[1] tutorial by Mark Nelson. The implementation of the matrix (and multiplication of matrixes) is taken from dev.bratched.com[2].

I have worked closely on the design of the code for the renderer with Jacob Grooss (jcgr@itu.dk) on this project.

2 Features

The core of the wireframe renderer takes a list of triangles (consisting of three vertices) and renders them to a Windows Forms window according to the Near, Far, Aspect Ratio and Field of View variables for the camera.

The renderer loads a hardcoded pyramid shape of five vertices making up four triangles. The position of the camera and the point the camera is looking at are changeable using the controls described in 2.1.

2.1 Camera controls

Movement of the camera position and look point is done in 10-unit intervals. This interval is hardcoded and should probably be a changeable variable or a smaller value.

The controls for the camera are:

- a/s/q decrease the value of the x/y/z position of the camera, respectively.
- d/w/e increase the value of the x/y/z position of the camera, respectively.
- j/k/u decrease the value of the x/y/z position of the camera's look point, respectively.
- l/i/o increase the value of the x/y/z position of the camera's look point, respectively.

3 Overview of the Program

The **Vector**, **Matrix**, **Vertex** and **Triangle** are just object representations of the respective math/geometry concepts. I have implemented the wireframe renderer using the column-vector convention.

The **Camera** class represents the camera of the renderer. The constructor defines the Near, Far, Aspect Ratio and Field of View variables, but these can be changed using the properties of the

[1] https://blog.itu.dk/MGAE-E2013/files/2013/09/transforms.pdf
[2] http://dev.bratched.com/en/fun-with-matrix-multiplication-and-unsafe-code/

camera. It also contains the calculations for the transformation matrices.
Currently the movement of the camera is very poorly implemented. Better camera movement would be actual rotation (instead of the look point only moving on one axis at a time) and the user being able to move the camera around with the mouse.

3.1 Renderer

The constructor of the **Renderer** instantiates a new camera and uses it to open a Windows Forms window. Then it gets the pyramid from the **Loader** and starts the first draw/paint.

Whenever the renderer is drawing to the screen, it first makes the camera calculate all of the transforms (more on this in section 3.2). Then it asks each triangle to update itself according to the camera (triangle then makes its three vertices update according to the camera) and then it draws the three lines making up the triangle.

I have chosen not to draw each point (only the lines), because it makes no difference to how the wireframe looks when rendered in my implementation. Each vertex could be a thicker/larger point than the line itself, but I do not believe this is necessary.

3.2 Math

The update function of the vertex contains most of the pseudo-code from section 4 of the overview document mentioned in section 1. I have chosen to update the screen point of the vertex at a lower level, because I believe it makes the code much cleaner.
It does come with a drawback, however. In my implementation I do not skip updating the screen point (as per "*if any of .x/.y/.z are outside the range [-1,1], skip to next vertex*"), so the figure may appear deformed if the camera is positioned in a particular way. A way around this problem could be to force calculation of the screen points if the screen point is null or (0, 0).

I have not implemented frustum culling (for the reasons stated above), but would do so if I also implemented the skipping of the vertices outside of the the range [-1, 1].

I have chosen to do an optimization when multiplying the three transforms camera transforms (*perspectiveTransform * cameraLookTransform * cameraLocationTransform*). I calculate the resulting matrix of this chain of transforms every time the renderer is about to draw and cache it to decrease the amount of necessary calculations on each draw. While it does not matter when there are only 20 updates to calculate[3], these calculations could take a very long time to do in case there are very many vertices.

4 Conclusion

The wireframe renderer works to a degree that allows models to be rendered as wireframes (if they are hardcoded). Because of the problem with screen point calculations (and no clipping), rendering an entire scene is problematic with the current implementation. Navigating a scene is also rather cumbersome due to the poor camera controls. A mouse implementation would offer a large improvement.

[3]Because every triangle is asked to update its vertices, even shared vertices are updated multiple times. This could be fixed by having a list of all vertices and not updating them once per triangle, but instead running through the entire list of vertices.

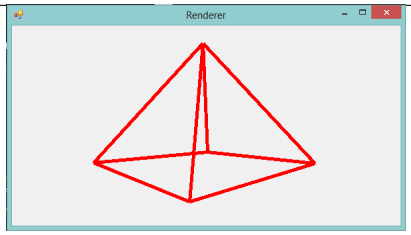

Figure 1: The starting position of the renderer.

4.1 Performance

The hardcoded pyramid contains 5 vertices and 4 triangles, which limits the amount of necessary calculations per call to the *Renderer_Paint* method. This limits the impact that can be felt from the lack of culling / clipping (even if the camera is away from the view object). To properly test the impact, there would need to be more than one object and/or more complex objects.

5 Future Work

My suggestions for future work done on this project would be to test performance on larger scenes, load triangles/vertices from a file, do frustum culling and to improve camera controls.

5.1 Test performance

The hard-coded pyramid does not stress the renderer at all, so it is hard to know if optimizations are necessary. Using a larger wireframe (such as a ship, a person or a detailed building) would stress the renderer in a way that would show this. Hard-coding these would be fairly cumbersome, so I would suggest this is done in combination with my next suggestion.
If optimizations are necessary, I would suggest implementing frustum culling (or clipping) as mentioned in section 3.2.

5.2 Load triangles/vertices from a file

In the current implementation of the renderer, it is much too cumbersome to render detailed objects as they need to be hard-coded. I suggest implementing a way to import triangles and vertices from a file, so that models made in 3D modelling software could be imported.

5.3 Better camera controls

Currently the camera controls are quite limited, as they do not do rotation of the view and the position of the camera can only be moved on the x/y/z-axis. Utilizing the mouse instead of only using the keyboard would also offer a large improvement.

6 Appendix

6.1 Screenshots

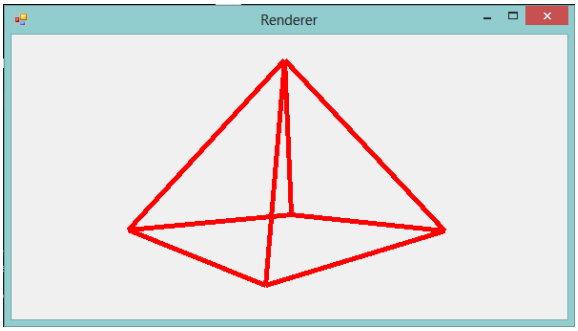

Figure 2: The starting position of the renderer.

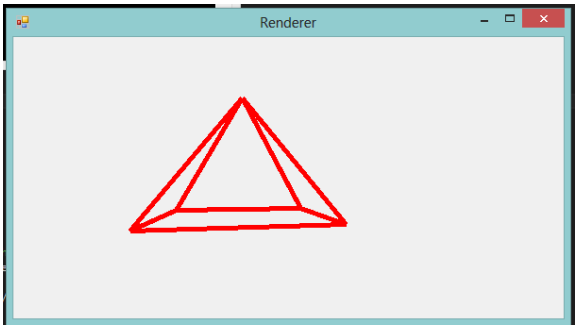

Figure 3: The camera position moved and the camera lookpoint changed.

6.2 Video

I have included a video of the wireframe renderer in action along with this document. Additionally I have uploaded the video to Youtube. Youtube Link: http://youtu.be/3KjRljro1zk