3D Renderer

By: Niels Justesen (noju)

Introduction

In this projects I have created a 3D wireframe renderer which is written in C++ and uses the SDL library. The projects also uses the SDL_draw library since there is no function for drawing lines in SDL. In this renderer I have placed a number of so called double pyramids. Their location are found using the following for loop to construct a kind of spiral.

```
for(int i = 0; i < 100; i++){
        DrawableObject *obj2 = obj->clone();
        obj2->setPosition(new Point3D((1000)*cosf(i), (1000)*sinf(i), i*-100) );
        objects->push_back(obj2);
}
```

It is possible to rotate all the elements in the world simultaneously in each of the three axes as well as scaling. Rotations are made by pressing the arrow keys and the WASD keys while pressing the control key. Scaling is done by pressing the up and down keys while pressing the shift key. Using the arrow keys and the WASD keys while not pressing the control or shift keys you control the camera by increasing or decreasing the acceleration of the camera.

Architecture

I have reused the game loop from my previous 2D platformer project and it is located in the main file. This project also uses an input handler to register which keys are down. As the draw function is in the main file it is here the rendering happens but with help from a transform handler. The transform handler contains a lot of transformation functions such as `rotateObjectX(DrawableObject *_obj, float _degrees)` which rotates an object a number of degrees around the x axis. The transform handler has similar function for points and vectors and additionally functions for scaling, translating and viewspace and perspective transformations. To calculate these transformations it uses the classes Point3D, Vector3D, Matrix41 and Matrix44. The Matrix41 is a 4x1 matrix and Matrix44 is a 4x4 matrix. For larger project it would make more sense to have a single Matrix class with variable sizes and dimensions, but for this small project this solution seemed simple. The Transform handler also contains the function of multiplying matrices which is used in all the other functions.
As it can be seen in the rotation function above I have created a class called DrawableObject. Drawable objects simply has a list of triangles and a global position. The Triangle class has three points each with a relative position.

Affine transformations

The affine transformations are as described previously represented as matrices. While affine transforms can be done by only three equations we need a 4x4 matrix to do the job. The fourth dimension is called a dummy dimension which we call w[1].
The equations for a translation looks like this:

$$x' = x + t_x$$
$$y' = y + t_y$$
$$z' = z + t_z$$

Which looks like this in the transform handler.

```
float mat44[4][4] =
       {
              { 1.0f, 0.0f, 0.0f, _offsetX },
              { 0.0f, 1.0f, 0.0f, _offsetY },
              { 0.0f, 0.0f, 1.0f, _offsetZ },
              { 0.0f, 0.0f, 0.0f, 1.0f },
       };
```

Rendering

The rendering consist of a series of transformations. The main file is looping through all drawable objects and drawing them one at the time. Each object is cloned and assigned new positions to its points after each transformation. The advantage of doing this is that it is still is an object when it has been projected into 2D which allows for different possibilities such a clicking on an object very easy to implement.
After the cloning, the following steps are performed:

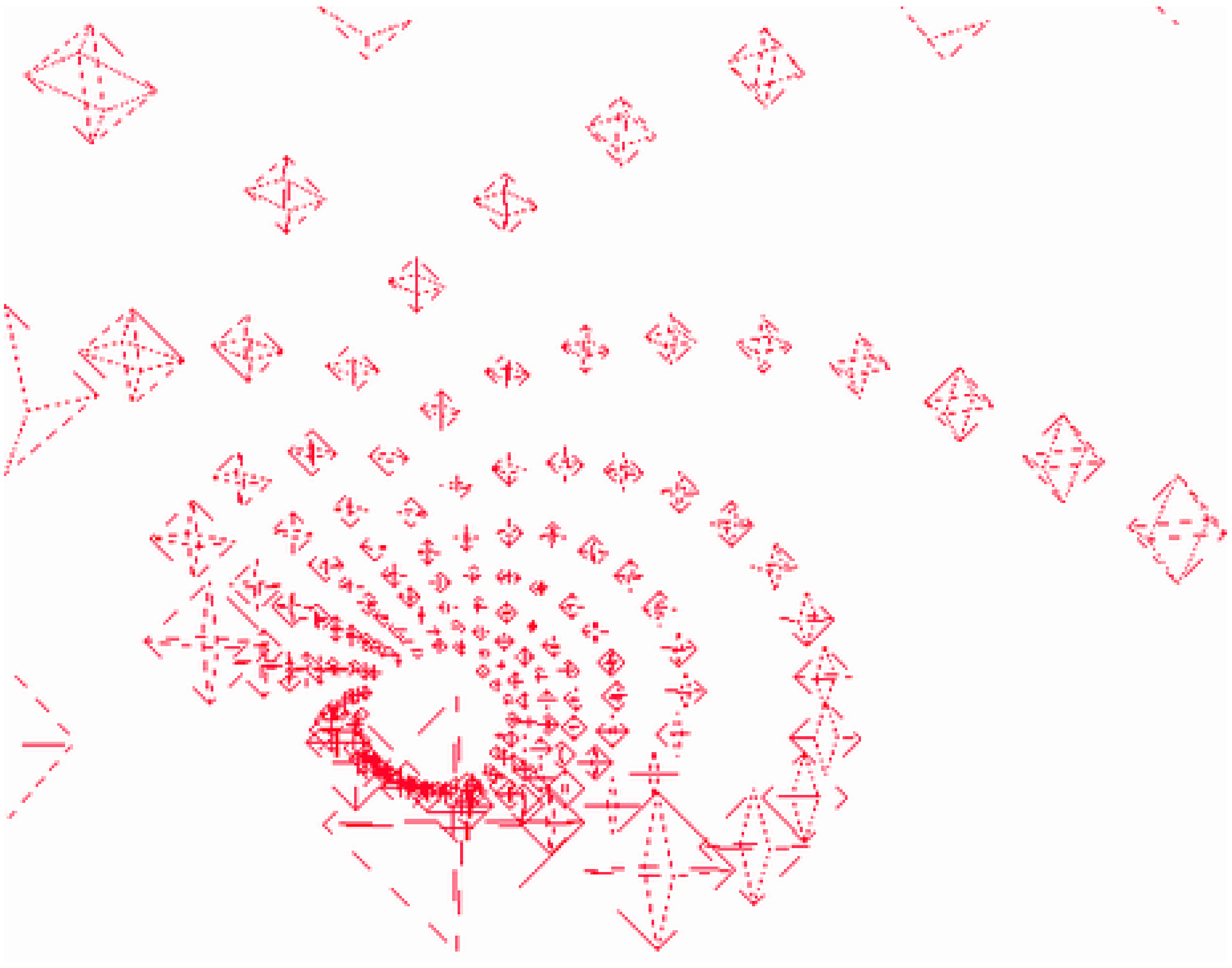
Step 1: All points of an object are translated from relative to global coordinates.
Step 2: All points of an object are translated relative to the camera position.
Step 3: All points of an object are transformed relative to camera rotation.
Step 4: All points of an object are transformed into normalized device coordinates which is a mapping of the viewing space into a cube with coordinates -1 to 1 along each axis.
Step 5: All points of an object are now in 2D but needs to be mapped to the screen size.

Step 3 requires quite a lot of vector math which is implemented in the camera class. It has a series of functions like `getViewPlaneNormal()` and `getIntermediateOrthogonalAxis()` that are used in this step.

[1] http://itu.dk/~mjas/engines/2012/3d_transform_overview.pdf

Optimizations

The renderer can render about 200 objects of 8 triangles (1600 triangles and 4800 points) on the screen smoothly with a frame rate of 24 frames per second on a 2.70 GHz dual core processor. By adding more objects it begins to slow down.
The renderer could be optimized several ways. Between each step in the rendering phase I am currently assigning the new coordinates to the the clone. this could of course just be done when the final screen coordinates are found. One of the cool thing about using matrices is that we can multiply all the transformations in a chain but this is of course not happening in this solution as I am assigning the new coordinates in between.
Clip space culling could be implemented which ignores points outside the viewspace region. Points outside the range of -w’ and w’ should be culled of. This would probably the most efficient optimization as it saves us from doing quite a lot of operations.



A screenshot of the renderer with 200 objects.