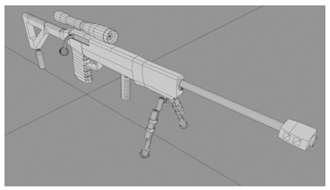
Introduction

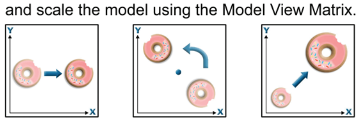
Everything you see above is a (very large) set of polygons. 3D computer graphics work by taking sets of polygons and arranging and coloring them in ways to make them look like objects we see every day.



A computer is really not much more than calculator on steroids, and a like a calculator it only understands numbers. The way the Graphics Processing Unit (AKA the GPU) creates polygons is by reading in the coordinates of corners of the polygons which are known as Verticies. In fact a file for a model of a cube could be made in a text editor

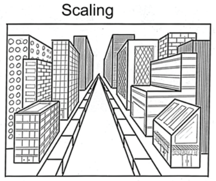
cube.3dm
```
-1.0  1.0  1.0
 1.0  1.0  1.0
 1.0 -1.0  1.0
-1.0 -1.0  1.0
-1.0  1.0 -1.0
 1.0  1.0 -1.0
 1.0 -1.0 -1.0
-1.0 -1.0 -1.0
```

When the GPU gets the verticies of the model, it can translate, rotate, and scale the model using the Model View Matrix.



Translation Rotation Scaling

Once the model has been transformed, it then has to be turned into a 2D image that can appear on your screen. This is where the Projection Matrix comes in.



The buildings in the above picture are parallel to each other, but to our eyes look like they're converging to a point. What the Projection Matrix does is transform the 3D verticies and make their positions converge on the focal point.

OpenGL is a graphics Application Programming Interface (API) that allows the programmer to interface with the GPU. Until version 2.1, OpenGL had functions to set the Projection Matrix (gluPerspective), translate (glTranslate), scale (glScale), and rotate (glRotate). When OpenGL 2.0 came out, it introduced Programmable Function Pipeline with allowed the programmer to give direction instructions to the GPU with GLSL (OpenGL Shader Language) programs.

vertex_shader.glsl
```
uniform mat4 myMatrix;

void main()
{
    //Transforming The Vertex
    gl_Position = myMatrix * gl_Vertex;
}
```

When OpenGL 3.0 came out, it got rid of all the Fixed Function Pipeline functions that automatically calculated the perspective, translation, scaling, and rotation matrices. Now if a programmer wants to use the latest version of OpenGL, they have to calculate the matrices themselves.

This means that these pieces of code:

```
GLfloat ratio = 640.f / 480.f;
gluPerspective( 45.0f, ratio, 1.f, 1000.0f );
```

```
glTranslatef( 0.f, 0.f, -20.f );
glRotatef( 45.f, 1.f, 1.f, 1.f );
glScalef( 3.f, 3.f, 3.f );
```

Need to be replaced with the proper matrix calculations.

Programmable Vertex Shaders

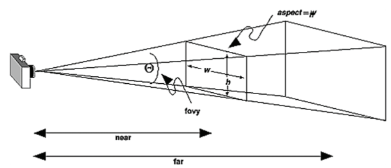
by Salvador Llamas

Methods

OpenGL vectors are 1 x 4 vectors with and x, y, z, and scale component (which is almost always set to 1). To take the vertex coordinates and put them into perspective, they are multiplied against the perspective matrix. Before OpenGL 3.0, the perspective matrix was automatically generated with gluPerspective.

$$\begin{pmatrix} x \\ y \\ z \\ s \end{pmatrix}$$

```
void gluPerspective(GLdouble fovy,
                    GLdouble aspect,
                    GLdouble zNear,
                    GLdouble zFar);
```



The perspective matrix can be manually calculated with:

Given f defined as follows:

$$f = cotangent\left(\frac{fovy}{2}\right)$$

$$\begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2\times zFar\times zNear}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Calculating the Translation Matrix

```
void glTranslated(GLdouble x,
                  GLdouble y,
                  GLdouble z);
void glTranslatef(GLfloat x,
                  GLfloat y,
                  GLfloat z);
```

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
void glScaled(GLdouble x,
              GLdouble y,
              GLdouble z);
void glScalef(GLfloat x,
              GLfloat y,
              GLfloat z);
```

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Calculating the Rotation Matrix

```
void glRotated(GLdouble angle,
               GLdouble x,
               GLdouble y,
               GLdouble z);
void glRotatef(GLfloat angle,
               GLfloat x,
               GLfloat y,
               GLfloat z);
```

The rotation matrix is fairly complex. The original glRotate took in the x, y, and z components of a vector a rotated it around the given angle. The rotation matrix is calculated like so:

$$\begin{pmatrix} x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ yx(1-c)+zs & y^2(1-c)+c & yz(1-c)-xs & 0 \\ xz(1-c)-ys & yz(1-c)+xs & z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Where $c = cos(angle)$, $s = sin(angle)$, and $\|(x, y, z)\| = 1$

Results

The Final Vertex Transformations

Final = Perspective * Translation * Rotation * Scaling

$$\begin{pmatrix} 4.37 & -1.69 & 2.75 & 0.00 \\ 3.66 & 5.83 & -2.25 & 0.00 \\ 0.93 & -1.52 & -2.42 & 22.04 \\ 0.93 & -1.52 & -2.41 & 20.00 \end{pmatrix} = \begin{pmatrix} 1.81 & 0.00 & 0.00 & 0.00 \\ 0.00 & 2.41 & 0.00 & 0.00 \\ 0.00 & 0.00 & -1.00 & 2.00 \\ 0.00 & 0.00 & -1.00 & 0.00 \end{pmatrix} * \begin{pmatrix} 1.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 1.00 & -20.00 \\ 0.00 & 0.00 & 0.00 & 1.00 \end{pmatrix} * \begin{pmatrix} 0.80 & -0.31 & 0.51 & 0.00 \\ 0.51 & 0.80 & -0.31 & 0.00 \\ -0.31 & 0.51 & 0.80 & 0.00 \\ 0.00 & 0.00 & 0.00 & 1.00 \end{pmatrix} * \begin{pmatrix} 3.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 3.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 3.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 1.00 \end{pmatrix}$$

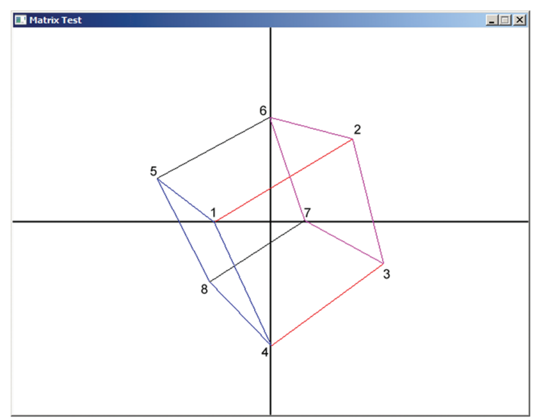
Pre Transformation

Vertex	X	Y	Z	S
1	-1.00	1.00	1.00	1.00
2	1.00	1.00	1.00	1.00
3	1.00	-1.00	1.00	1.00
4	-1.00	-1.00	1.00	1.00
5	-1.00	1.00	-1.00	1.00
6	1.00	1.00	-1.00	1.00
7	1.00	-1.00	-1.00	1.00
8	-1.00	-1.00	-1.00	1.00

Post Final * Vertex Transformation

X	Y	Z	S
-3.31	-0.09	17.17	15.14
5.43	7.24	19.04	17.00
8.81	-4.41	22.08	20.04
0.06	-11.74	20.21	18.17
-8.81	4.41	22.01	19.96
-0.06	11.74	23.87	21.83
3.31	0.09	26.92	24.86
-5.43	7.29	25.05	23.00

*All values rounded to the nearest hundreth



Manual Calculation of 3D cube vertices to 2D

Summary

3D computer graphics are made by taking a set of polygons and combining them to create 3D shapes that resemble objects in 3D space. Taking the vertices of the 3D shapes and multiplying them against a perspective matrix produces vector positions that be used to create 2D images that can be shown on the screen.

Older versions of graphics APIs calculated transformation matricies for the programmer. Now they need to be calculated manually. The OpenGL 2.1 documentation is freely available and gives the equations needed to replace the deprecated API calls.

With the matricies calculated, they are then multiplied against verticies of a model to get 2D coordinates used for drawing the shape on a 2D canvas

Conclusions

Replacing the fixed function matrix calculations with manual calculation was fairly painless. OpenGL 2.1 is well documented and calculating the matrices was just a matter of plugging in the variables.

The ability to calculate the matrices would also have uses for fixed function OpenGL. Being able to do the matrix calculations manually would allow the programmer to do the matrix calculations in RAM and send the matrix once. This would allow for fewer calls to the command buffer and a boost in performance.

The ability to calculate transformation matricies would also be needed to z sort polygons. When there are polygons with transparency they need to be rendered back to front, and knowing how to calculate the vertex depth when rendered is needed to do so.

There are bound to be more applications to manually calculating the matricies. It the world of computer graphics, it is a very valuable skill to have.

Acknowledgements

Web Kinesia LCC. "Computer Graphics." *Web Kinesia.* Web Kinesia LCC, 2003. Web. 7 May 2010. <http://www.webkinesia.com/online/graphics/notes/viewing2.php>.

Wright, Richard S., Benjamin Lipchak, and Nicholas Haemel. *OpenGL Superbible: Comprehensive Tutorial and Reference.* Upper Saddle River, NJ: Addison-Wesley, 2007. Print.