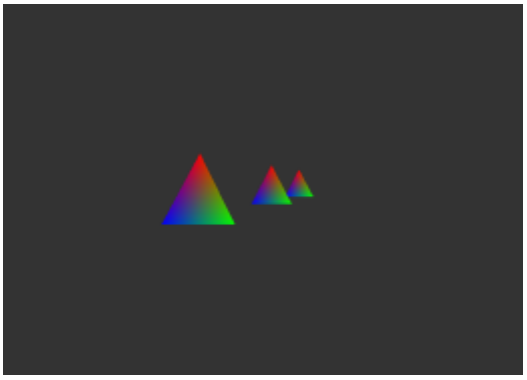
Tutorial 2 Part A: Vertex Transformation



Summary

Last tutorial you learnt how to draw a triangle on screen - now you're going to learn how to transform that triangle, using **translation**, **scaling**, and **rotation** matrices. You'll also learn about **projection** matrices, which can add a sense of depth to your rendered scenes.

New Concepts

Model matrices, Projection matrices, matrix translation, matrix rotation, matrix scaling, znear, zfar

Introduction

In the first tutorial, we rendered a triangle directly into *clip space*. That was OK for an introduction, but is not very useful in the long run! In this lesson, you'll learn how to use the vertex shader to transform vertices from their *local coordinate space* into *world* space and finally *clip* space, via the *model* and *projection* matrices.

Matrix Basics

The model and projection matrices, as well as the *view* and *texture* matrices you'll be introduced to later, are a type of *transformation* matrix - those which manipulate *homogenous coordinates*. Homogenous coordinates have an extra cooordinate, *w*, which is usually set to a value of 1.0 - remember how the vertex positions in the last tutorial had to be expanded out to 4 component vectors in the vertex shader? That is to keep them homogenous - you'll see how this extra dimension coordinate is used later. These transformation matrices are 4 by 4 *square* matrices. Here's how they're normally represented:

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

The contain 16 real numbers, arranged into 4 rows and 4 columns. Generally these elements will be stored as **floats** - graphics hardware is heavily geared towards performing floating point operations on 4 component vectors, making them also ideal for computing with matrices.

Matrix Multiplication

In your graphical applications, you will often find yourself multiplying matrices together, both in your C++ code, and your shaders. This is because it is *multiplication*, not addition, that concatenates the effects of matrices together. For example, in this tutorial you will learn how to create matrices that will translate vertex positions in space, and matrices that rotate them. In order to do a translation AND a rotation on a vector, the two transformation matrices must be multiplied together. This is done like so, with each value of the resulting matrix being the *dot product* of the relevent *row* of the first matrix and the *column* of the second:

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix} \cdot \begin{bmatrix} . & . & b_8 & . \\ . & . & b_9 & . \\ . & . & b_{10} & . \\ . & . & b_{11} & . \end{bmatrix} = \begin{bmatrix} . & . & c_8 & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix} \quad c_8 = \begin{bmatrix} a_0 b_8 + a_4 b_9 + a_8 b_{10} + a_{12} b_{11} \end{bmatrix}$$

Commutativity

The method used to multiply matrices together means that the multiplication order is not *commutative* - their ordering matters! For example, multiplying matrix a by matrix b is *not* the same as multiplying matrix b by matrix a.

$$\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix} \cdot \begin{bmatrix} 5 & 2 & 3 & 1 \\ 5 & 2 & 3 & 1 \\ 5 & 2 & 3 & 1 \\ 5 & 2 & 3 & 1 \end{bmatrix} = \begin{bmatrix} . & . & c_8 & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix} \quad c_8 = \begin{bmatrix} 0 \cdot 3 + 4 \cdot 3 + 8 \cdot 3 + 12 \cdot 3 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 2 & 3 & 1 \\ 5 & 2 & 3 & 1 \\ 5 & 2 & 3 & 1 \\ 5 & 2 & 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix} = \begin{bmatrix} . & . & c_8 & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix} \quad c_8 = \begin{bmatrix} 5 \cdot 8 + 2 \cdot 9 + 3 \cdot 10 + 1 \cdot 11 \end{bmatrix}$$

Identity Matrix

When working with transformation matrices, it may be desirable to have a matrix which does *nothing* - maybe you want to just draw something as-is at the origin? To do so, the *identity* matrix is used. It looks like this:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Multiplying any matrix m by an identity matrix i will result in a matrix identical to m. It's worth pointing out that in OpenGL, using only an identity matrix for your transformation would leave your viewpoint looking down the *negative* z axis - so to move 'forward', you will actually 'subtract'!

Vertex Transformation

Vertices are transformed to their final clip space position in the vertex shader, by multiplying their position vector by the matrix formed by multiplying the model, view, and projection matrices together. A vector is multiplied by a matrix in the following way (remember, we make our 3 component vertex positions homogenous by adding a 1!):

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x m_0 + y m_4 + z m_8 + w m_{12} \\ x m_1 + y m_5 + z m_9 + w m_{13} \\ x m_2 + y m_6 + z m_{10} + w m_{14} \\ x m_3 + y m_7 + z m_{11} + w m_{15} \end{bmatrix}$$

The Model Matrix

The model matrix is used by the vertex shader to transform incoming vertices from their *local space* (the coordinate values defined in arrays or loaded in from a file when creating a mesh) to *world space* (the global coordinate system that determines where objects are in relation to each other in the scene). All of the vertices for a mesh will be transformed by the same model matrix, and can contain any combination of translation, rotation, scale, and shear information required to transform an object into world space. By multiplying together transform matrices, a model matrix can be made to translate, rotate, and scale your objects to whereever you want in world space.

Translation

A translation matrix has the following properties. It has a value of 1.0 down its diagonal, and has a translation component down the right side - this is how much the vertex will be translated by on each axis:

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

So, for example, we could translate the *origin* (0,0,0) of a mesh in local space to a world space position of (10,10,10) using the following matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 10 \\ 0 & 0 & 1 & 10 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \cdot 1 + 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 10 \\ 0 \cdot 0 + 0 \cdot 1 + 0 \cdot 0 + 1 \cdot 10 \\ 0 \cdot 0 + 0 \cdot 0 + 0 \cdot 1 + 1 \cdot 10 \\ 0 \cdot 0 + 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 10 \\ 10 \\ 10 \\ 1 \end{bmatrix}$$

Rotation

A rotation matrix rotates a vector around an *axis* - defined as a *normalised* 3 component vector with components x, y, z. It is defined as follows:

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

Where c is the **cosine** of the angle to rotate the vector by, and s is its **sine**.

Take this example: imagine we have a simple line, starting at the origin, and extending 10 units down the x axis. If we wanted to rotate this line so it points down the z axis, we would need to rotate the end point of the line -90 ° by the y-axis (*0*, *1*, *0*) - think of the rotation axis as a spindle through the object by which it rotates.

$$\begin{bmatrix} 0^2+c & 0 \cdot 1 - 0s & 0 \cdot 0 + 1s & 0 \\ 1 \cdot 0 + 0s & 1^2 + c & 1 \cdot 0 - 0s & 0 \\ 0 \cdot 0 - 1s & 1 \cdot 0 + 0s & 0^2 + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 10 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} 10 \cdot (0^2+c) & 0 \cdot (0 \cdot 1 - 0s) & 0 \cdot (0 \cdot 0 + 1s) & 1 \cdot (0) \\ 10 \cdot (1 \cdot 0 + 0s) & 0 \cdot (1^2 + c) & 0 \cdot (1 \cdot 0 - 0s) & 1 \cdot (0) \\ 10 \cdot (0 \cdot 0 - 1s) & 0 \cdot (1 \cdot 0 + 0s) & 0 \cdot (0^2 + c) & 1 \cdot (0) \\ 10 \cdot (0) & 0 \cdot (0) & 0 \cdot (0) & 1 \cdot (1) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 10 \\ 1 \end{bmatrix}$$

With c = cosine(-90) = 0, and s = sine(-90) = -1

Scaling

Multiplying a vector by a scaling matrix scales its values on a per-axis basis - meaning the result gets closer to or further away from the origin. It is defined as follows:

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

Where x, y and z are the scaling factors for each axis - they can be negative! To build on the previous example, imagine we now want our 10 unit line to be 100 units long. The line is now pointing along the z axis, so the following scaling matrix will make the 10 unit line 100 units long.

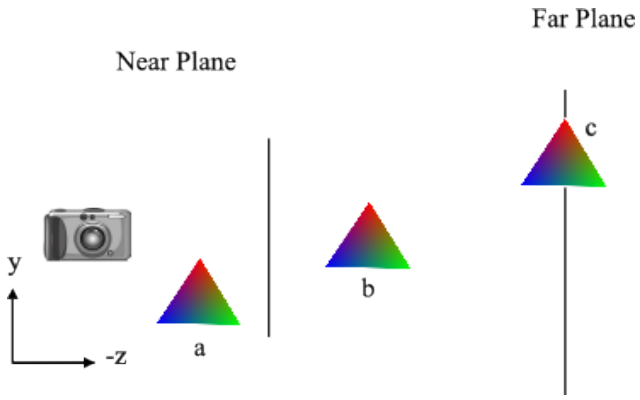
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \cdot 1 & 0 \cdot 0 & 10 \cdot 0 & 1 \cdot 0 \\ 0 \cdot 0 & 0 \cdot 1 & 10 \cdot 0 & 1 \cdot 0 \\ 0 \cdot 0 & 0 \cdot 0 & 10 \cdot 10 & 1 \cdot 0 \\ 0 \cdot 0 & 0 \cdot 0 & 10 \cdot 0 & 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 100 \\ 1 \end{bmatrix}$$

The Projection Matrix

The projection matrix takes our world coordinates, and maps them into *clip space*, which maps vertices to a space which stretches from -w to w one each axis, where w is the vertices own homogenous w component. Any primitives past those values will either be culled (if the entire primitive is outside of this range) or *clipped* (if part of the primitive will still be visible). This matrix is also used to 'project' the 3D world we define via our transformation matrices and vertex coordinates onto a flat plane - essentially, you can think of this flat plane is our 2D monitor screen. As part of this projection process, it is possible to add a sense of *perspective* to the scene - The model and camera matrices have no real mathematical workings that will make an object that is moving away from the camera get smaller. There are a variety of ways of calculating the values for a projection matrix, but they can be grouped into two basic types, the *orthographic* projection, and the *perspective* projection.

Near and Far Planes

We only have so much precision in a floating point value, so we can't really have a view that truly goes off into infinity. Instead, we must constrain what is seen on screen by using a *near* and a *far* plane - nothing in front of the near plane is drawn, and nothing behind the far plane is drawn. The greater the space between these two planes, the fewer bits of accuracy we have to play with in our scene, so its good practise to limit the near and far planes to only be as large as needed. Technically, the projection matrix defines **six** of these so-called 'clipping' planes, as there's a plane for the left, right, top, and bottom sides of our scene, too. Together, these planes are known as a *frustum* - you'll do more with frustums later in the module.



Triangle a *is in front of the far plane, and will not be seen, while triangle* c *actually intersects the far plane - parts of triangle* c *will be culled!*

Orthographic Projection

The simpler of the two projection matrix types is the orthographic matrix. As with drawing directly into clip space, the orthographic projection is entirely *parallel* - there is no perspective added to the scene. It does, however, allow the creation of a cuboid viewing area around the origin, using values to determine the maximum viewing range per axis. These maximum viewing ranges are then 'squished' to go from -1 to 1 by the matrix - the orthographic matrix doesn't change a vertices' w component, so the resulting clip space goes from -1 to 1. The most common orthographic projection matrix is defined as follows:

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & \frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & \frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So, instead of having a viewable space the ranges from -1 to 1 on each axis as with going directly to clip space like we did in the previous tutorial, we have several options when using an orthographic matrix. We could have a viewing area that goes from -100 to 100 on each axis, simply by using the following values:

near = -100, far = 100, left = -100, right = 100, top = 100, bottom = -100

You should be able to see how using the following values would form an identity matrix, and thus keeping clip space as the resulting coordinate space:

near = -1, far = 1, left = -1, right = 1, top = 1, bottom = -1

Or we could create a projection matrix that creates a space that matches that of the view area, using these values:

near = -1, far = 1, left = 0, right = screen width, top = 0, bottom = screen height

This one is particularly useful as it allows pixel perfect placement of objects on screen - each unit in space equals one pixel on screen. It is commonly used to draw the HUD, menus, and text in games, either using the screen width and height directly, or by using 'virtual canvas' - The orthographic projection used in id Software's *Doom 3* always uses a width of 640 and a height of 480, no matter what the actual screen resolution. This allows artists to define exactly whereabouts on screen the health bar etc should be, in a screen resolution independent way.

Perspective Projection

In 3D games such as first person shooters, it is common to use a perspective projection. It is this added perspective *forshortening* that makes objects get bigger and smaller as they get closer and farther away from the viewpoint, and that makes parallel lines extending into the distance appear to converge at their *vanishing point*. The perspective projection is commonly defined as follows:

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

where f = cotangent ($\frac{fov}{2}$) , *aspect* = screen Width / screen Height, *zNear* and *zFar* are the near and far planes, and *fov* is the vertical *field of vision* - how wide the angle of vision should be, in degrees. The larger this value, the more objects to the side of the view normal will be visible. It was once common for FPS games to have a field of vision of 45 °, but many modern console FPS games have vertical fields of vision as low as 30 ° either way.

The perspective divide

You'll notice how the perspective projection matrix has a value of -1 at m_{11}. If you follow the math that multiplies a vector by a matrix, you'll see that this will put the *negated* z value of a input vector in the output vector's w value. This becomes important due to the *perspective divide*. After the vertex shader completes, the x, y, and z components of the output vector are divided by the w component, turning the 'clip space' coordinates into their final 'normalised device coordinates'. It is this divide which makes objects in the distance seem smaller - the further away something is, the larger the w component will be, resulting in smaller x, y and z values post-divide, moving everything closer to the centre of the screen the further away it is. You will also notice how we have a translation in the z-axis of our matrix - this is due to a side effect of the perspective divide. If we divide the z axis by w (which is just the z axis value, really), then z will always equal 1 after the perspective divide. This is *bad* as it means we lose the ability to determine if one triangle is behind another. So, we add a translate derived from the near and far planes, so that even once our translated z has been divided by the 'untranslated' w value, we still end up with a unique value to represent how far 'into' the scene a vertex is, and which still maps the z-axis from -1 to 1.

To show the effects of the perspective divide, here's the result of performing the perspective divide on some vectors, representing vertex positions. If we were to form a perspective matrix with a zNear of 1, a zFar of 100, a vertical field of vision of 45.0, and an aspect ratio of 1.33 (for example, from a screen resolution of 800 by 600), we'd get the following matrix:

$$M = \begin{bmatrix} 1.81 & 0 & 0 & 0 \\ 0 & 2.41 & 0 & 0 \\ 0 & 0 & -1.02 & -2.02 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Multiplying vectors that are 10 units to the right of the origin, and varying distances from the viewpoint by this perspective matrix, we get the following:

$$A = M \cdot \begin{bmatrix} 10 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 18.1 \\ 0 \\ -3.0 \\ -1 \end{bmatrix} B = M \cdot \begin{bmatrix} 10 \\ 0 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 18.1 \\ 0 \\ -4.0 \\ -2 \end{bmatrix} C = M \cdot \begin{bmatrix} 10 \\ 0 \\ 99 \\ 1 \end{bmatrix} = \begin{bmatrix} 18.1 \\ 0 \\ -103.0 \\ -99 \end{bmatrix} D = M \cdot \begin{bmatrix} 10 \\ 0 \\ 100 \\ 1 \end{bmatrix} = \begin{bmatrix} 18.1 \\ 0 \\ -104.0 \\ -100 \end{bmatrix}$$

After the perspective divide step, we'd get the following vectors:

$$A' = \begin{bmatrix} -18.1 \\ 0 \\ 3.04040 \end{bmatrix} B' = \begin{bmatrix} -9.0 \\ 0 \\ 2.03030 \end{bmatrix} C' = \begin{bmatrix} -0.18289 \\ 0 \\ 1.04060 \end{bmatrix} D' = \begin{bmatrix} -0.1810 \\ 0 \\ 1.04040 \end{bmatrix}$$

If you look at the four post-perspective divide vectors, you'll see that their x axis position converge towards 0 the further away they are - it is this that makes geometry smalller in the distance, and what makes the 'vanishing point' effect on parallel lines. You should also see that the z axis is also effected - the effects of the perspective divide are non-linear, so the difference in z axis between vectors A' and B' is far greater than the difference between C' and D', something which will become important later, when looking at *depth buffers*.

Example program

The example program for this Tutorial will allow us to render 3 triangles in either perspective or orthographic mode. These triangles can then be translated, rotated, and scaled using the model matrix, controlled by the keyboard. Previous versions of OpenGL provided in-built support for the manipulation of the projection and model matrices via its matrix *stack* - but OpenGL 3 does away with that, forcing you to handle all matrix-based functionality yourself. Instead, we'll be using the *Matrix4* nclgl class to handle our matrix needs. As matrices are so intrinsic to the correct rendering of graphical scenes, the *OGLRenderer* class that these tutorials inherit from has *model* and *projection* matrices as member variables, along with a *view* matrix, which you'll be using in the next tutorial. In your Tutorial2 solution, create a *Renderer* class that inherits from *OGLRenderer*, and a text file called *Tutorial2.cpp*. We'll be reusing last tutorial's fragment shader, but writing a new vertex shader, so create a text file called *MatrixVertex.glsl* in the *Tutorial2* folder.

Renderer header file

Our *Renderer* class for this Tutorial is pretty similar to that of Tutorial 1. This time, we have 3 new **protected** member variables - these will control the scale, rotation, and position of our rendered triangles. We also have **public** accessors for each of these, that will be used in our main loop - we'll define these in the header file, as they're so simple. Finally, there's two additional **public** functions that will be used to switch between perspective and orthographic projections.

```
1  #pragma once
2
3  #include "./nclgl/OGLRenderer.h"
4
5  class Renderer : public OGLRenderer {
6  public:
7     Renderer(Window &parent);
8     virtual ~Renderer(void);
9
10    virtual void   RenderScene();
11
12    void  SwitchToPerspective();
13    void  SwitchToOrthographic();
14
15    inline void SetScale(float s)       { scale = s;}
16    inline void SetRotation(float r)    { rotation = r;}
17    inline void SetPosition(Vector3 p)  { position = p;}
18
19 protected:
20    Mesh*    triangle;
21
22    float    scale;
23    float    rotation;
24    Vector3  position;
25 };
```

Renderer.h

Renderer Class file

The **constructor** and **destructor** for the *Renderer* class in this tutorial are similar to Tutorial 1 - but note the new vertex shader on line 6, and the call to our *SwitchToOrthographic* function on line 15.

```
1  #include "Renderer.h"
2
3  Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
4     triangle        = Mesh::GenerateTriangle();
5
6     currentShader   = new Shader("MatrixVertex.glsl",
7                                  "../Shaders/colourFragment.glsl");
8
9     if(!currentShader->LinkProgram()) {
10       return;
11    }
12
13    init        = true;
14
15    SwitchToOrthographic();
16 }
17
18 Renderer::~Renderer(void)  {
19    delete triangle;
20 }
```

Renderer.cpp

The next two functions in this Tutorial switch the projection matrix between a perspective projection, and an orthographic projection, using the nclgl *Matrix4* class functions *Perspective* and *Orthographic*, which simply create the matrices described earlier. *Perspective* takes 4 parameters - a near and far z, an aspect ratio (remember how aspect is used in the perspective matrix formation?) and a horizontal field of vision.

```
21 void  Renderer::SwitchToPerspective()  {
22    projMatrix = Matrix4::Perspective(1.0f,10000.0f,
23                  (float)width / (float)height, 45.0f);
24 }
```

Renderer.cpp

Orthographic takes 6 parameters - one for each axis and direction, in the order *back*, *front*, *right*, *left*, *top*, *bottom*.

```
25 void  Renderer::SwitchToOrthographic() {
26    projMatrix = Matrix4::Orthographic(-1.0f,10000.0f,
27    width / 2.0f, -width / 2.0f,height / 2.0f, -height / 2.0f);
28 }
```

Renderer.cpp

Both matrix projections have a far plane value of 10,000 - enough for a large scene. Perspective has a near plane value of 1.0 - this is a fairly common default. Due to the way the depth buffer values are calculated, the closer a near plane value is to 0, the lower the depth buffer precision is. The orthographic projection, however, has -1 as its near plane. Why? Orthographic projections are often used to draw text and HUD information on screen, and it is intuitive for such elements to have a depth of 0. So, to ensure such items are drawn, a negative near plane value is often used with orthographic projections.

Finally, we have the *RenderScene* function of our tutorial's *Renderer*. Lines 34 and 37 demonstrate how to update a matrix **uniform** variable of a shader. The **glUniformMatrix4fv** function calls are a bit long winded - the first parameter is the variable name, the second is how many matrices to update (it is possible to have an array of matrices!), the third is whether the matrix should be *transposed*, and the fourth is a pointer to the matrix data. Beginning on line 40, we render three triangles in the world, progressively further away from the origin. Line 46 shows how the model matrix for an object can be formed by multiplying several transformation matrices together, in this case, a translation, rotation and a scale, all of which are controlled by the **local variables** of the *Renderer* class. Line 50 then sends this concatenated model matrix to the shader, and 53 draws the current triangle, at the position determined by the model matrix.

```
29 void Renderer::RenderScene()  {
30    glClear(GL_COLOR_BUFFER_BIT);
31
32    glUseProgram(currentShader->GetProgram());
33
34    glUniformMatrix4fv(glGetUniformLocation(currentShader->GetProgram()
35       , "projMatrix"),  1,false, (float*)&projMatrix);
36
37    glUniformMatrix4fv(glGetUniformLocation(currentShader->GetProgram()
38       , "viewMatrix"),  1,false, (float*)&viewMatrix);
39
40    for(int i = 0; i < 3; ++i) {
41       Vector3 tempPos = position;
42       tempPos.z += (i*500.0f);
43       tempPos.x -= (i*100.0f);
44       tempPos.y -= (i*100.0f);
45
46       modelMatrix = Matrix4::Translation(tempPos) *
47                     Matrix4::Rotation(rotation,Vector3(0,1,0)) *
48                     Matrix4::Scale(Vector3(scale,scale,scale));
49
50       glUniformMatrix4fv(glGetUniformLocation(
51       currentShader->GetProgram(), "modelMatrix"), 1,false,
52       (float*)&modelMatrix);
53       triangle->Draw();
54    }
55
56    glUseProgram(0);
57
58    SwapBuffers();
59 }
```

Renderer.cpp

Main file

Our main function is quite long this time, but still pretty simple! We want to be able to rotate, scale and translate our triangles, so we need key checks for each of these. The + and - keys control the *scale* local variable, *I*, *J*, *K*, *L*, *O* and *P* control the *position* local variable, and finally the *left* and *right* arrow keys control the *rotation* local variable. These are then sent to the *Renderer* each frame by the accessor functions we declared earlier, and finally the *RenderScene* function is called. Note how the position *z*-axis is set to *-1500.0f* - we draw 3 triangles, each 500 units closer than the last, so the final triangle will have a z-axis position of 0.0f.

```cpp
#pragma comment(lib, "nclgl.lib")

#include "./nclgl/window.h"
#include "Renderer.h"

int main() {
	Window w("Vertex Transformation!",800,600,false);
	if(!w.HasInitialised()) {
		return -1;
	}

	Renderer renderer(w);
	if(!renderer.HasInitialised()) {
		return -1;
	}

	float scale    = 100.0f;
	float rotation = 0.0f;
	Vector3 position(0, 0, -1500.0f);

	while(!w.UpdateWindow() && !keyboard->KeyDown(KEYBOARD_ESCAPE)){
		if(Window::GetKeyboard()->KeyDown(KEYBOARD_1))
			renderer.SwitchToOrthographic();
		if(Window::GetKeyboard()->KeyDown(KEYBOARD_2))
			renderer.SwitchToPerspective();

		if(Window::GetKeyboard()->KeyDown(KEYBOARD_PLUS))  ++scale;
		if(Window::GetKeyboard()->KeyDown(KEYBOARD_MINUS)) --scale;

		if(Window::GetKeyboard()->KeyDown(KEYBOARD_LEFT))  ++rotation;
		if(Window::GetKeyboard()->KeyDown(KEYBOARD_RIGHT)) --rotation;

		if(Window::GetKeyboard()->KeyDown(KEYBOARD_K))
			position.y -= 1.0f;
		if(Window::GetKeyboard()->KeyDown(KEYBOARD_I))
			position.y += 1.0f;

		if(Window::GetKeyboard()->KeyDown(KEYBOARD_J))
			 position.x -= 1.0f;
		if(Window::GetKeyboard()->KeyDown(KEYBOARD_L))
			 position.x += 1.0f;

		if(Window::GetKeyboard()->KeyDown(KEYBOARD_O))
			position.z -= 1.0f;
		if(Window::GetKeyboard()->KeyDown(KEYBOARD_P))
			position.z += 1.0f;

		renderer.SetRotation(rotation);
		renderer.SetScale(scale);
		renderer.SetPosition(position);
		renderer.RenderScene();
	}

	return 0;
}
```

Tutorial2.cpp

Vertex Shader

Our vertex shader is similar to last tutorial, but with 3 new **uniform** variables, one for each of our matrices. To transform the incoming vertices to the correct position, we must multiply them by a combined 'Model View Projection' matrix. Remember, matrix multiplication is **not** commutative, so the order in which we multiply matrices together matters - in this case we multiply them in *reverse* order to create the correct matrix. The *viewMatrix* variable will be set to an *identity* matrix, and so will not change the *mvp* variable. You'll see how to use this variable to transform vertices into a space local to a camera viewpoint in the next tutorial.

```
1  #version 150 core
2
3  uniform mat4 modelMatrix;
4  uniform mat4 viewMatrix;
5  uniform mat4 projMatrix;
6
7  in  vec3 position;
8  in  vec4 colour;
9
10 out Vertex {
11    vec4 colour;
12 } OUT;
13
14 void main(void)   {
15    mat4 mvp    = projMatrix * viewMatrix * modelMatrix;
16    gl_Position = mvp * vec4(position, 1.0);
17    OUT.colour  = colour;
18 }
```

MatrixVertex.glsl

Running the Program

When running the program, you should be met with a scene of 3 triangles, in an orthographic projection. You'll notice that even though our for *RenderScene* **for** loop brings each triangle 500.0f units closer to the 'camera', each triangle is the same size. Contrast this with the perspective projection of the scene you see when you press the *2* key, where the triangles further away become smaller. We can also translate our triangles using *I*,*J*,*K* and *L*. Again, note how orthographic and perspective projections have different results - no matter how far we move our triangles away from the screen, they stay the same size, but in a perspective projection they get smaller and smaller - in both cases they will eventually disappear, as they hit the far z distance.

Tutorial Summary

With the completion of this tutorial, you should know how to transform your meshes, by translating them in world coordinates, as well as rotating and scaling them. You should also now know that the order in which matrices are multiplied together is important. Finally, you should have an idea of how to apply a projection matrix to your scene. Orthographic projections are useful for 2D games, and for the drawing of in-game HUDs, while perspective projections are more useful for 3D games. As the demo program shows, it is easy to use both orthographic and perspective projections in a single application, so you should be starting to see how games can use both to render their scenes. Next tutorial, we'll take a look at the third transformation matrix - the *view* matrix. This will allow us to translate and rotate our viewpoint around our scene, independently of our model matrix.

Further Work

1) What happens if you change the order of the matrix multiplications in the *RenderScene* function on line 43? Why is this?

2) When drawing in-game HUDs, it is useful to be able to target the screen directly, so that model space translations are equal to pixels on screen. What projection matrix will perform this?

2) Some games use the *field of vision* parameter of a perspective matrix to create visual effects - think of the narrowing of the fov when using the iron sights in an FPS game, or the fish-eye lens effect in *Aliens vs. Predator*. Add a controllable *fov* member variable to the *Renderer* class, in a similar manner to the triangle rotation member variable.