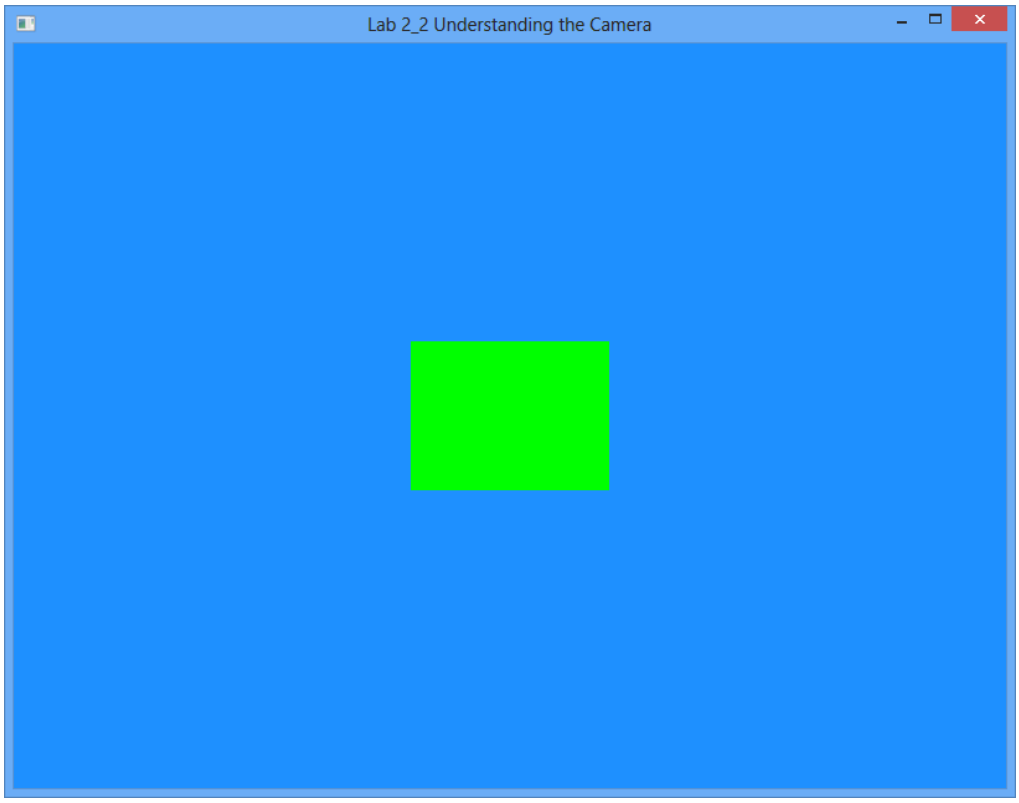
OpenGL Lab 2_2: Understanding Model View Projection

This is the third lab for Simulation and 3D Graphics. In the last lab we learnt how to use vertex and fragment shaders in the graphics pipeline, and how to link data in vertex buffer objects into shaders. This lab is dedicated to understanding a basic camera model, which we will implement using shaders. Once we understand the basic camera model we will be able to move around in 3D space. We'll also be introducing a couple of new methods from OpenTK to help us.

Before you begin this lab, backup last week's code. Then in the Program.cs file change the m_CurrentLab value to Lab.L2_2

```
private static Lab m_CurrentLab = Lab.L2_2;
```

This will run the correct window for this lab. In the solution explorer window in the Lab2 folder open Lab2_2Window.cs. Hit F5 and the solution should build and run without a problem. You should see the following window.



L22T1 Rendered a small green square to start the camera lab

This square is loaded from a ModelUtility file. Being able to load models is useful, however in industry the way models are loaded is far more efficient than using this made up file format. This make shift solution will do for now. In this case a small green square has been loaded.

Often a single model will be used many times, over and over, in different positions. To do this we create a model matrix. A model matrix can be used to transform one space to a different space. In this case we will be using a 4x4 matrix which is capable of dealing with both rotation and translation, scaling and skewing a model.

If you are unfamiliar with matrices, it would be useful to refresh your understanding. Basically matrices perform transforms on sets of points (such as vertices). Matrices can be used to rotate, translate and scale points from one space to another. We'll also used a special type of matrix to project points from a virtual "world space" into Normalised Device Coordinates.

In order to transform our model in 3d space we need to multiple each vertex by a model matrix. As matrices act on vertices the vertex shader seems an appropriate place. This variable will not change with every vertex, so it's going to be a uniform value.

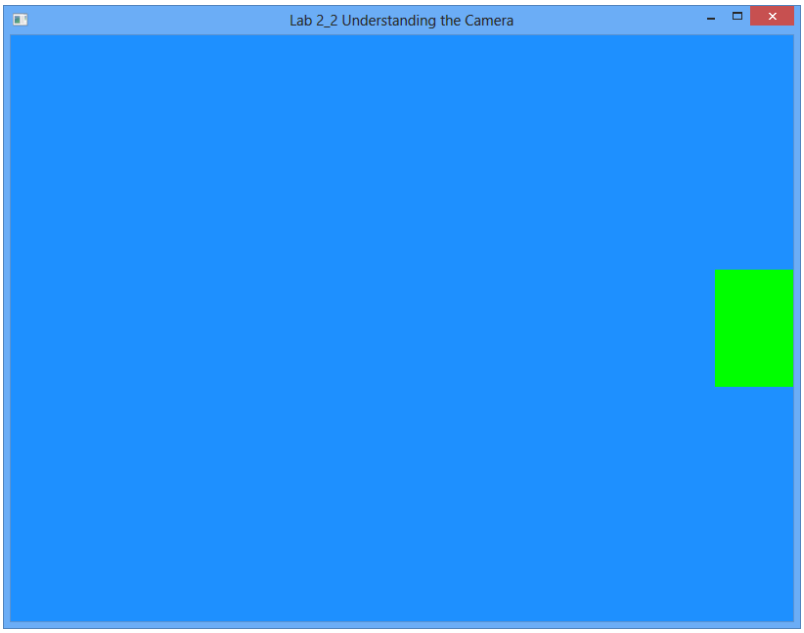
Change the vertex shader to be

```
#version 330

uniform mat4 uModel;

in vec3 vPosition;
in vec3 vColour;

out vec4 oColour;

void main()
{
        gl_Position = vec4(vPosition, 1) * uModel;
        oColour = vec4(vColour, 1);
}
```

Now we need to set this uniform matrix to a value. In the OnRenderFrame method before the draw call change the state of uModel like this:

```
int uModelLocation = GL.GetUniformLocation(mShader.ShaderProgramID, "uModel");
Matrix4 m1 =  Matrix4.CreateTranslation(1,0,0);
GL.UniformMatrix4(uModelLocation, true, ref m1);
```

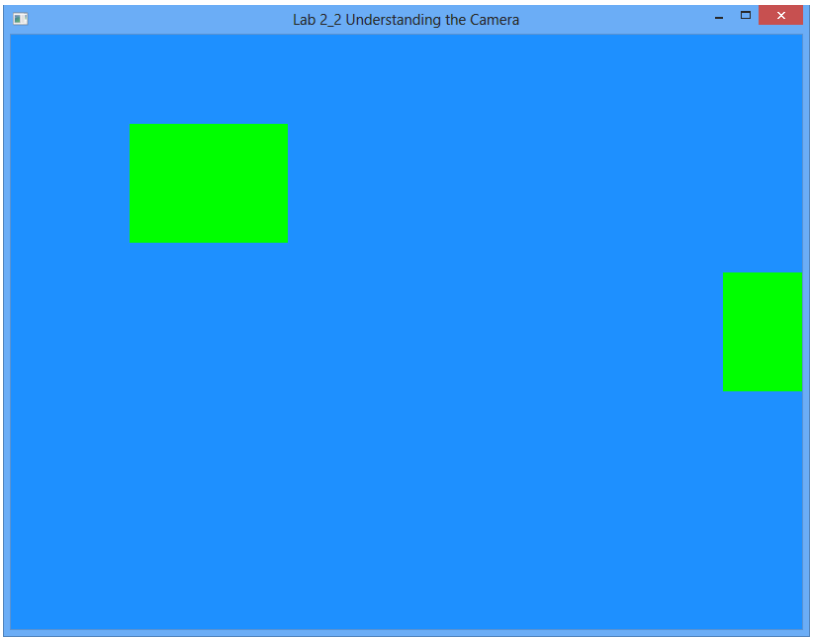
This code translates the model in the x direction by one unit. If you run your code you will see that now the square is hanging off the edge of the screen.



Commit your code with the message

L22T2 Translated the square one unit to the right

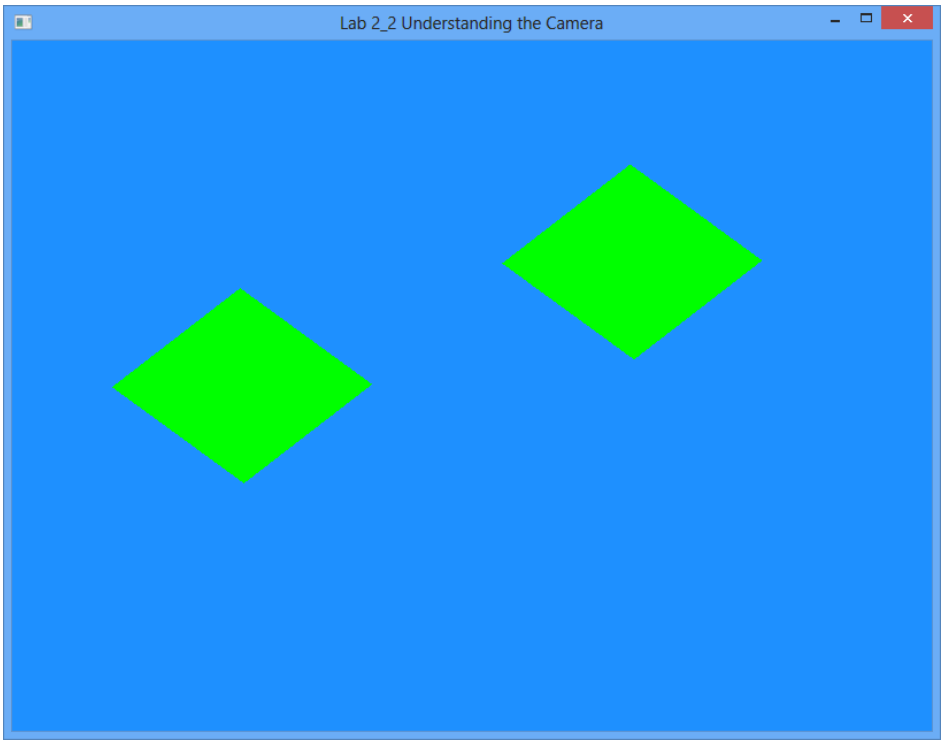
Now we can move things around it is more meaningful to render the same objects over and over, because we can render them in different places. See if you can render a second square, half a unit to the left, and half a unit up.



Commit your code with the message:

L22T3 Reused a model to render two squares by changing the model matrix

You can also store rotations in a matrix. Try translating one square in half a unit in the x direction, then rotating about the z axis by an angle of 0.8 radians (about 45 degrees). For the other square perform the rotation first and then the translation in the opposite direction (i.e. -0.5 in x).



Note that the order of translation and rotation is crucial to the outcome! You screen shot may not look exactly like this one.

Comment your code with the message:

L22T4 Two squares translated and rotated

We can also use matrices to move the camera. In actual fact, it's more like we're moving the whole world (in the opposite way we want the camera to move) around the camera. In the vertex shader add another uniform 4x4 matrix called uView and in the main function multiply apply the matrix to the vertex position like this:

```
        gl_Position = vec4(vPosition, 1) * uModel * uView;
```

We also need to keep track of the value of the view matrix in our game window, so create a Matrix4 member variable of Lab2_2Window called mView. In the OnLoad function, set that value to Matrix4.Identity. The identity matrix is a matrix that has no effect, which is what we want to begin with.

Create and initialize a Matrix4 variable called mView:

```
mView = Matrix4.Identity;
```

Create three Vector3 variables called *eye*, *lookAt* and up to specify the camera position and camera focus point and its up direction respectively, forexample:

```
Vector3 eye = new Vector3(1.2f, 2.0f, -5.0f);
Vector3 lookAt = new Vector3(0, 0, 0);
Vector3 up = new Vector3(0, 1, 0);
mView = Matrix4.LookAt(eye, lookAt, up);
```

Specify the value for the uniform variable *uView* in the vertex shader :

```
int uView = GL.GetUniformLocation(mShader.ShaderProgramID, "uView");
GL.UniformMatrix4(uView, true, ref mView);
```

Next, we need to override a method provided by OpenTK that gets called every time a key is pressed. In this case when the 'a' key is pressed we want the camera to move left. In order to do that we move the whole world right instead. Create a new method in the Lab2_2Window class.

```
protected override void OnKeyPress(KeyPressEventArgs e)
{
  base.OnKeyPress(e);
  if (e.KeyChar == 'a')
  {
      mView = mView * Matrix4.CreateTranslation(0.01f, 0, 0);
      int uView = GL.GetUniformLocation(mShader.ShaderProgramID, "uView");
      GL.UniformMatrix4(uView, true, ref mView);
  }
}
```

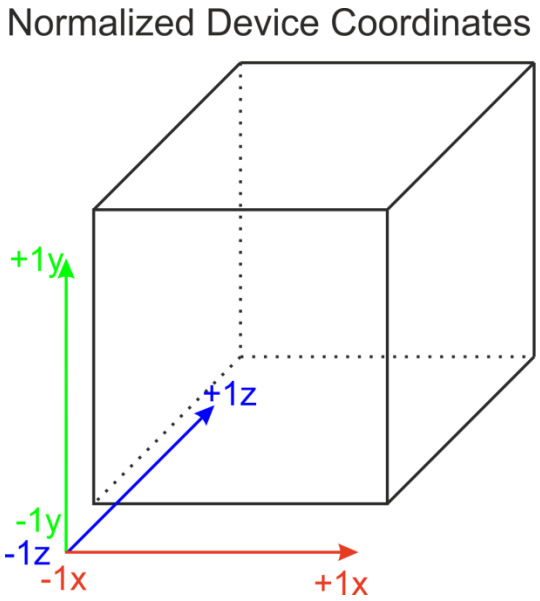
Adapt this code to use the d key to move in the opposite direction. When you can move the camera left and right comment your code with the message

L22T5 Set up vertex shader to use view matrix and set view matrix in new OnKeyPress function

Great, so now we can move the camera side to side, but chances are you did a cut and paste and now have some repeated code. If we don't nip this in the bud now it's only going to get worse! Highlight the two lines that link the view to the shader, right click and select Refactor > Extract Method (or press Ctrl R, M). Called your refactored method MoveCamera, and make sure to use it anywhere you copy and pasted the code you've replaced. You might also want to get rid of any magic numbers in your code with variables (like cameraSpeed, for example).

L22T6 Refactors camera code and eliminated magic numbers

The last thing to address the concerns cameras is the projection matrix. The projection matrix is used to project a space of our choosing into normalized device coordinates. In a previous lab we discussed how OpenGL uses something called normalized device coordinates, and up to now the space we've been drawing in has been a 2 by 2 by 2 square with the origin at the centre.



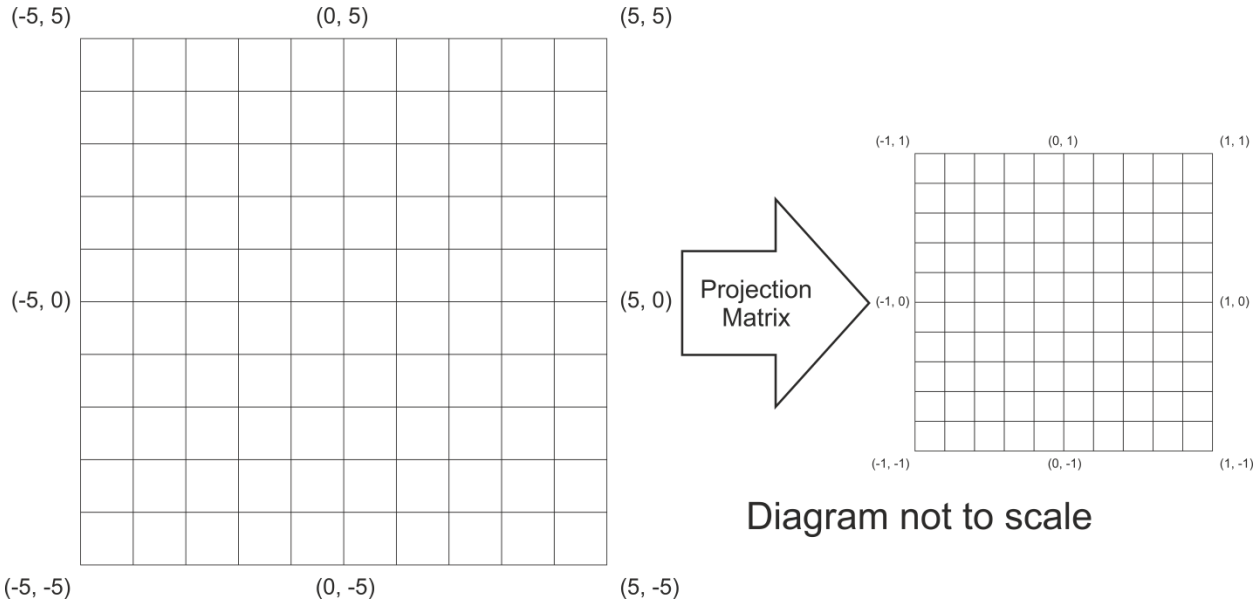
Next you will see how we can use matrices to create a camera and move around in a 3D space that we choose. Behind the scenes everything is still being converted to OpenGL's 2 by 2 by 2 Normalized Device Coordinates, but using matrices make this much easier for us.

Create another matrix in the shader called uProjection, and use that to transform the vertex in model-view space into projected space.

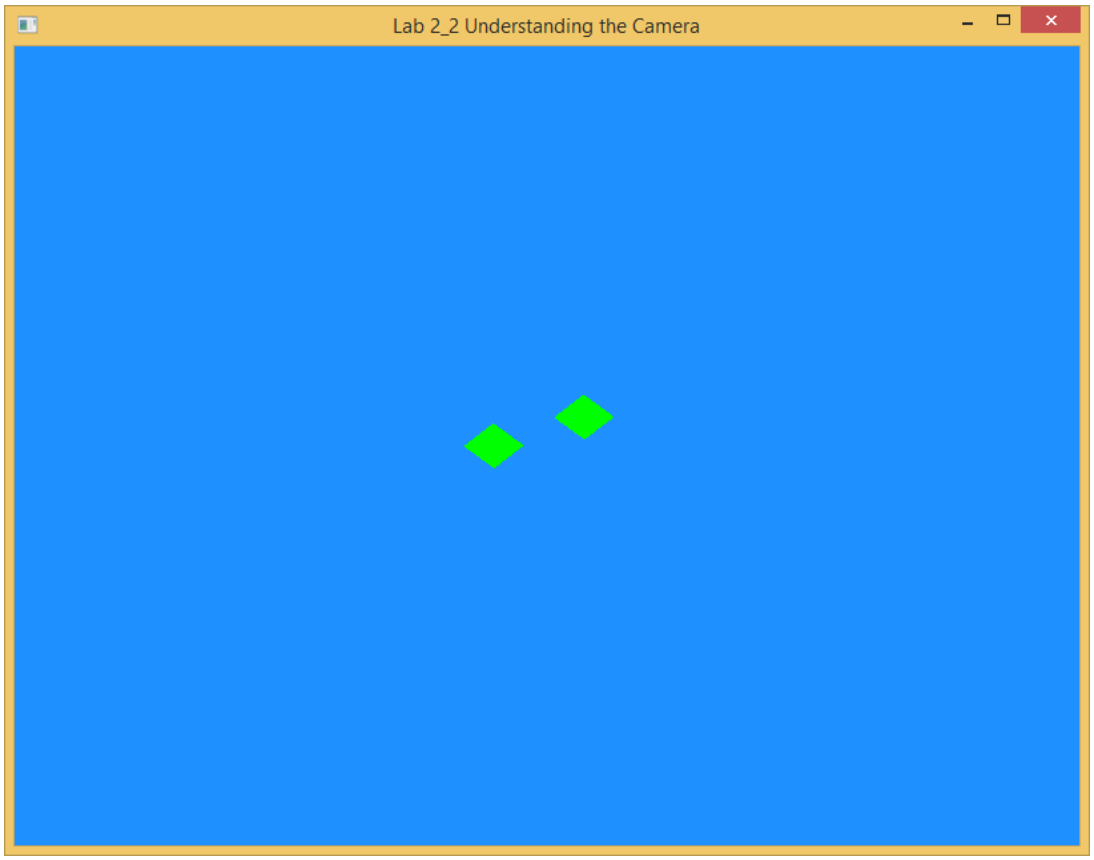
We need to assign the projection matrix a value. We can do this in the OnLoad method.

```
int uProjectionLocation = GL.GetUniformLocation(mShader.ShaderProgramID, "uProjection");
Matrix4 projection = Matrix4.CreateOrthographic(10, 10, -1, 1);
GL.UniformMatrix4(uProjectionLocation, true, ref projection);
```

This creates an orthographic projection that is ten units wide and ten units high, with near and far boundaries at -1 and 1. Effectively instead of having a window that goes from -1 to 1 in x and y we now have a window that goes between -5 and 5 in both x and y.



If you run your code, you should see that the scene has been expanded, and the green squares are relatively smaller.



There is a second square to the right of the first that was previously outside of OpenGL's normalized device coordinates. One thing that should be noted, but may not be immediately obvious is the during

the projection process the z value has been flipped. –z now goes into the screen and +z comes out of the screen, comment your code with the following log message.

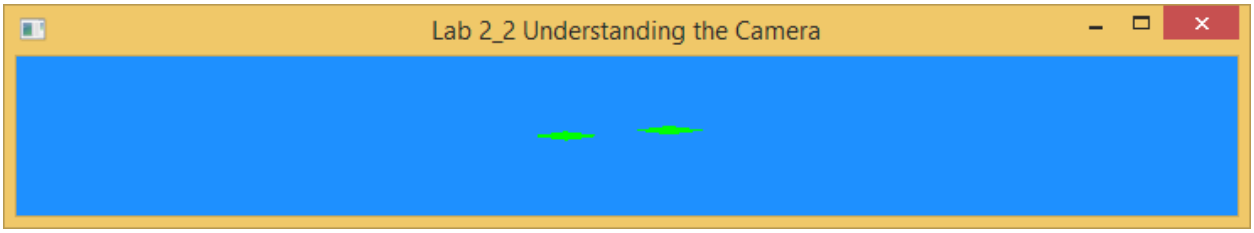
L22T3 Added a projection matrix to the shader to increase the size of the viewing volume

There has been something that isn't quite right for the last few labs. None of my squares are very square! The reason is that OpenGL's square coordinate system is being stretched into a rectangular window. We could use the projection matrix to fix that!

OpenTK provide us with a method that we can override that is called whenever the window is resized, but before we override that method take a minute to resize your current window and notice the effect. Add this code to the Lab2_2Window class.

```
protected override void OnResize(EventArgs e)
{
  base.OnResize(e);
  GL.Viewport(this.ClientRectangle);
}
```

This doesn't change the projection, but it does tell OpenGL that the size of the window it's filling (which affects the rasterization process) has changed. Play with the size of the window and notice the difference.



This is better, but it's still not exactly what we want.

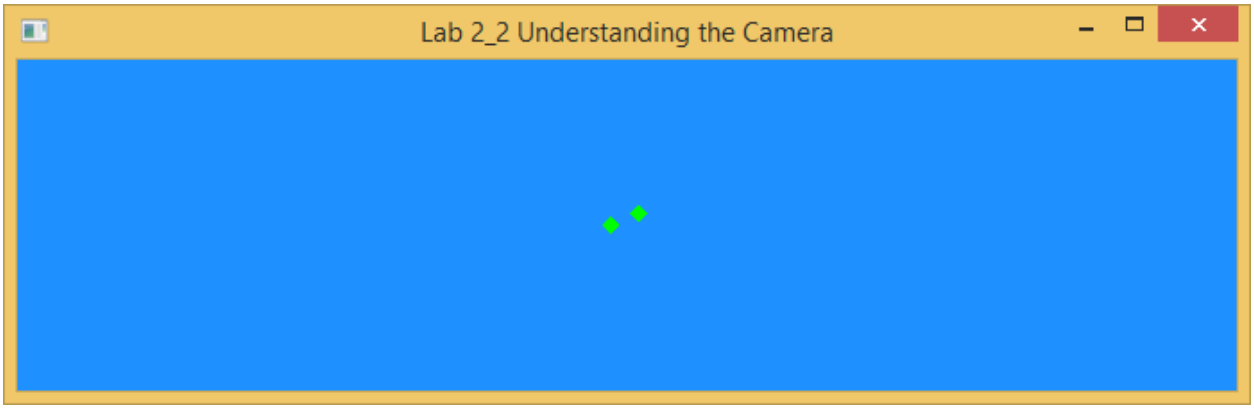
Comment your code with the message

L22T4 Can change the viewport on resize, but the aspect ratio still isn't right

Below the GL.Viewport call add the following code.

```
if (mShader != null)
{
  int uProjectionLocation = GL.GetUniformLocation(mShader.ShaderProgramID, "uProjection");
  int windowHeight = this.ClientRectangle.Height;
  int windowWidth = this.ClientRectangle.Width;

  if (windowHeight > windowWidth)
  {
     if (windowWidth < 1) { windowWidth = 1; }

     float ratio = windowHeight / windowWidth;
     Matrix4 projection = Matrix4.CreateOrthographic(ratio * 10, 10, -1, 1);
     GL.UniformMatrix4(uProjectionLocation, true, ref projection);
   }
   else
   {
      if (windowHeight < 1) { windowHeight = 1; }

      float ratio = windowWidth / windowHeight;
      Matrix4 projection = Matrix4.CreateOrthographic(10, ratio * 10, -1, 1);
      GL.UniformMatrix4(uProjectionLocation, true, ref projection);
   }
}
```

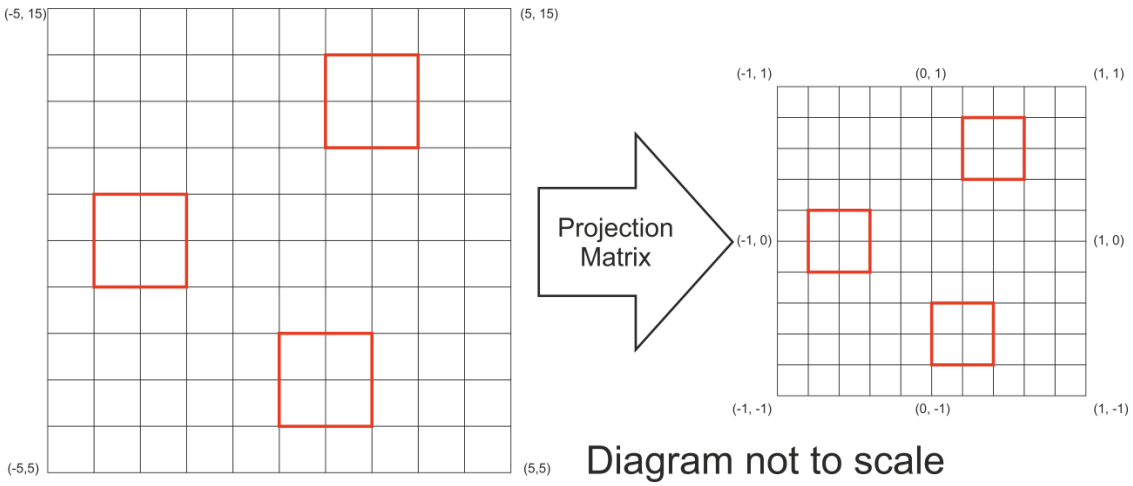
Read through this to try and get an understanding of the code. The intended purpose of this code is to check the width and height of the window, and ensure that when we set up the orthographic projection matrix that the smaller dimension is set to 10 units, and the larger dimension is set to the appropriately proportioned amount. This means that when no matter what the width and height of your window are the green squares will now always be square. There are ***two*** bugs in this code, but hopefully by understanding the code and what it should do you will be able to find them.



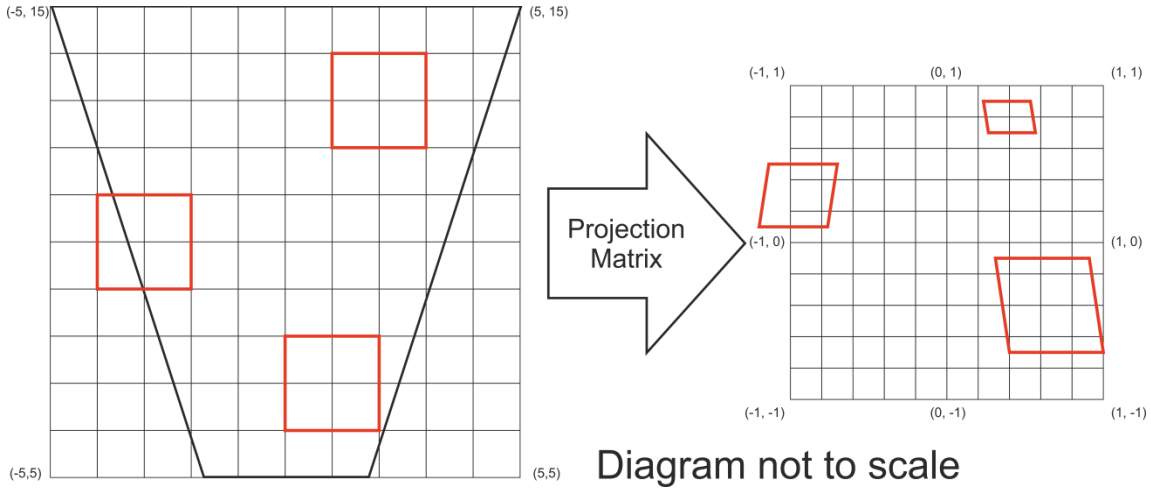
When you've found the bugs comment your code with the following (completed) log message.

L22T5 Window resizes properly. The bugs were …

Orthographic projections are really useful for creating things like heads up displays, but they don't give us any depth cues at all. Two objects of the same size will appear the same size after an orthographic projection no matter how far away they are.



Let's get some *perspective*! A perspective projection squashes a frustum shape into normalized device coordinates. This gives additional depth cues, and makes everything a lot more "three dimensional".



Remove the orthographic matrix setup and set the projection matrix using the line.

```
Matrix4 projection = Matrix4.CreatePerspectiveFieldOfView(1, (float)ClientRectangle.Width /
ClientRectangle.Height, 0.5f, 5);
```

You will have to do this in the OnLoad method and the OnResize method. The parameters are the field of view for the y axis, the screen aspect ratio, the near clipping plane and the far clipping plane. In the resize method you no longer have to worry about whether the window is taller than it is wide or not, because the CreatePerspectiveFieldOfView method takes care of that for you.

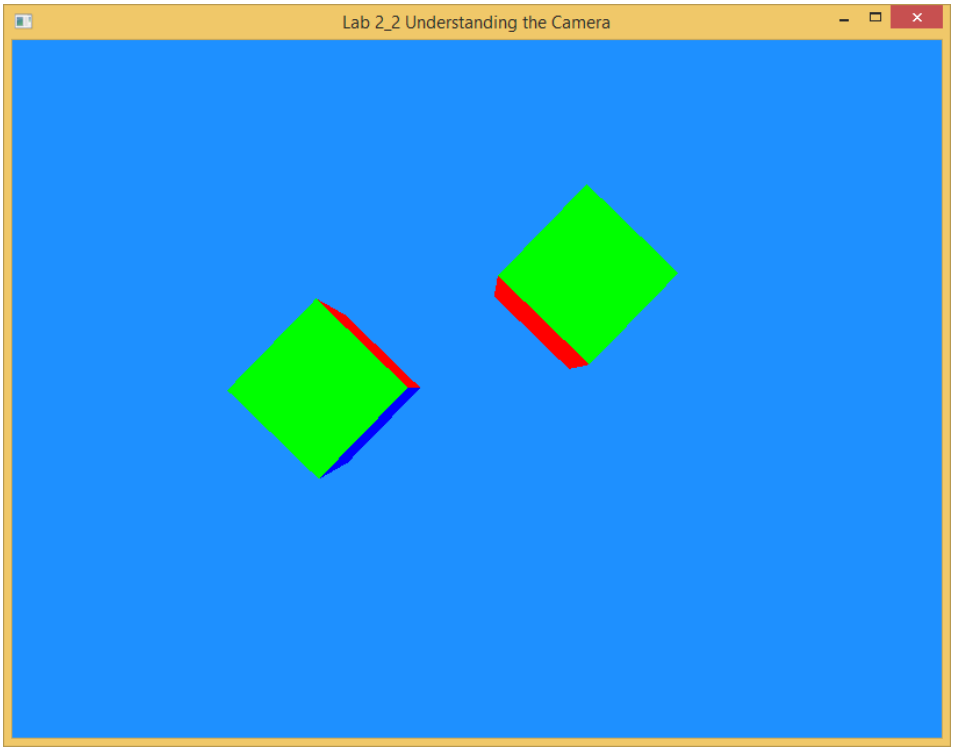
Run your code, and you'll see a blank screen. That's because the squares are closer than the near clipping frame. To remedy this you could move the squares deeper into the scene, but it's easier to move the camera (i.e. change the view matrix).

In the OnLoad method replace

```
mView = Matrix4.Identity;
```

With

```
mView = Matrix4.CreateTranslation(0, 0, -2);
```



The perspective view shows that the green squares we've been working with are actually cubes!

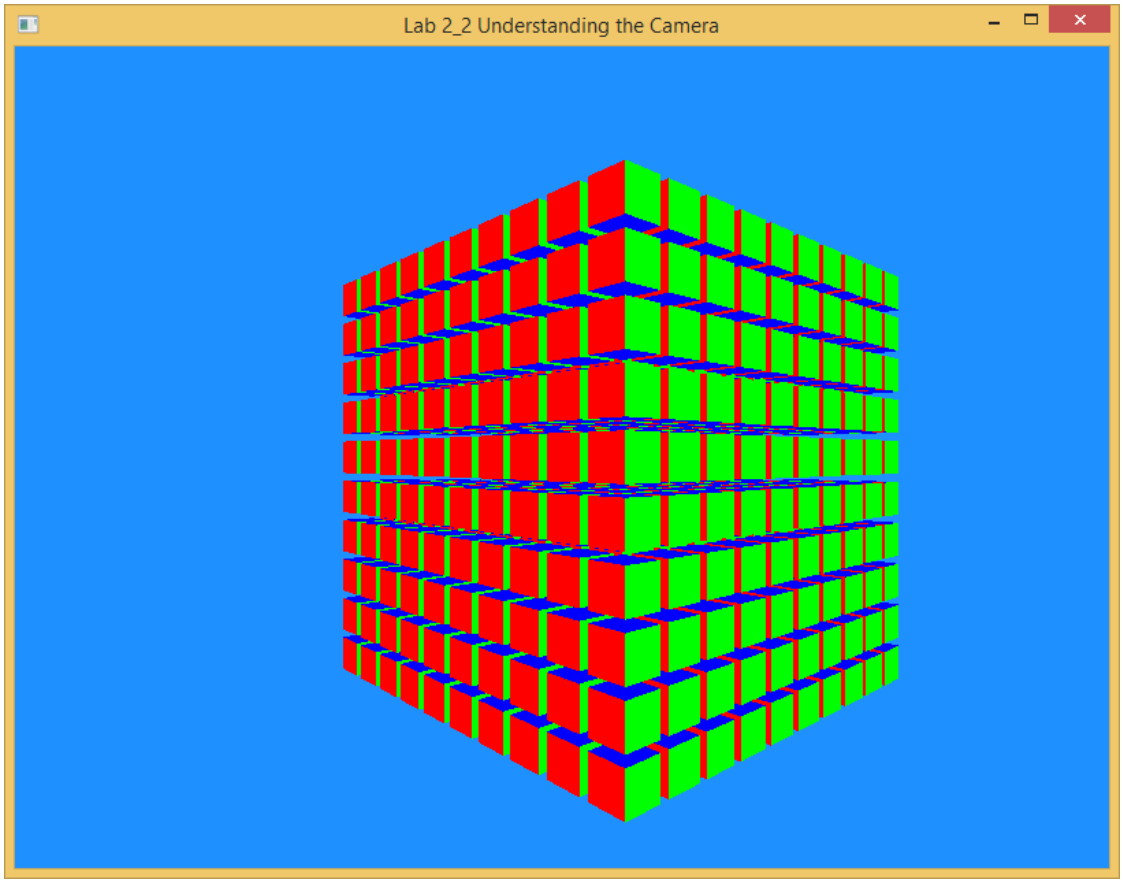
Comment your code with the message

L22T6 Created a perspective camera

Next modify the OnKeyPress method to allow you to rotate the camera about the y axis, and move through the screen in x and z. Comment your code with the following log message.

L22T7 Created a user controlled perspective camera

Finally, let's make the scene a bit more interesting. Use what you've learnt to make a cube of cubes.



Once you've done that – comment your code with the following log message.

L22T8 Made a cube of cubes

Summary

In this lab we've learnt about the model-view-projection matrix, how the order of matrix operations is important and how to set up both orthographic and perspective projections. We also added new OpenTK functions that are called when keys are pressed and when the window is resized.