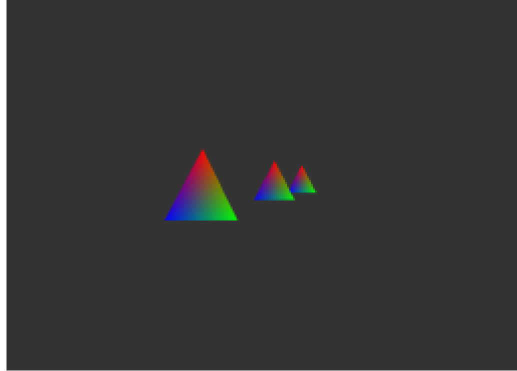
Tutorial 2 Part B: The View Matrix & Frame Rates



Summary

Now that you can move your objects around in world space, you're probably wondering how to move your viewpoint around, too. This tutorial will show you how to do so using the *view* matrix, and show an example of a simple *Camera* class to use in your OpenGL applications. You'll also see how to move around your scene at a constant rate, no matter what your frame rate is.

New Concepts

Camera Matrix, View Matrix, Camera class, framerate independence

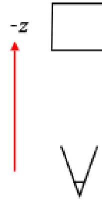
Introduction

In the last tutorial, you were briefly introduced to the view matrix, as a member variable of the *OGLRenderer* class, and a **uniform** variable of a shader, but you didn't do much with them. In this lesson, you'll learn how to manipulate the view matrix, allowing you to move around your scenes. To demonstrate this, we're going to create a *Camera* class, that responds to typical FPS game mouse and keyboard commands, and which has a function to create a valid view matrix. Then, we'll add it to last tutorial's code, to show how easy it is to add camera support to your OpenGL applications. Also, as your scenes get more complicated, you may find that your applications cannot maintain a constant framerate, so this lesson will also show you how to combat some of the possible side effects of this.

The View Matrix

Like the model matrix, the view matrix can contain translation and rotation components - but instead of transforming an individual object, it transforms everything! It moves your geometry from *world* space to *view* space - that is, a coordinate system with your desired viewpoint at the origin. Earlier versions of OpenGL combined the model and view matrices together, into one 'modelview' matrix, but OpenGL 3, and most other graphical rendering APIs, keep them separate, until being multiplied together in a vertex shader. Just as you have been using model matrices to translate and rotate your meshes, you can think of the view matrix as being the model matrix of the view point - it can contain any of the transformation components you were introduced to last tutorial, including translation and rotation.

However, there's one important thing to consider - to form a correct view matrix, the matrix transformations you perform must be *inverted*. The following example should hopefully make it clear why. Imagine a scene where your camera is at the origin, looking down the negative z axis, towards a cube that is 10 units away.



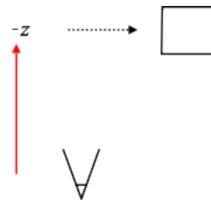
The camera matrix, the cube's model matrix, and the combined 'modelview' matrix they form, would look as follows:

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

The resulting matrix places the cube 10 units away from the camera, as expected. Now, imagine that *both* the cube *and* the camera move 10 units down the x axis. The matrices formed would look like this:

$$\begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -10 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 20 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This doesn't look right! If we were to use this matrix to render our object with, it would appear off to the right of the screen, when it should still be in the centre, as both the cube and the camera have moved along the same axis by the same amount.



Instead, we use the *inverse* of the camera matrix, so it is transformed in the opposite way:

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

That's better! Using the inverse gives us a final matrix which places the cube in front of the camera at a distance of 10 units.

Framerate Independence

In the current applications you've been writing, you probably get a constant, very high, frame rate - the exact framerate will depend on whether your graphics driver locks your framerate to the refresh rate of your monitor. But what if your game gets complex enough that the framerate becomes inconsistent?

Think about what you did last tutorial - every frame, if the *J* key is pressed down, the triangle moves 1 unit to the left. So, if your framerate is capped at 60 frames per second, your triangle moves 60 units to the left. But what if your framerate is unlimited? You might get over 1000 frames a second on such a simple scene - that'll mean your triangle would move 1000 units to the left in one second!

The simplest solution, and the one we'll be using in these tutorials, is scale values by how much time has passed since the last frame. How accurately time can be measured, and the unit of measurement used, will depend on your operating system, and the functions used. For now, we'll just assume the unit of measurement is milliseconds, and the timing is accurate to within 1 ms. Now, assume we want our triangle to move 60 units to the left, no matter what our framerate is. How would we achieve this? Like this!

$$Movement = x/(1000.0/msec)$$

Where x is the 'per second' value (in this case 60 units) and $msec$ is the number of milliseconds that have passed since the last frame. So for example, if our framerate was a mere 1 FPS, we'd end up with:

$$Movement = 60.0 = 60.0/(1000.0/1000.0)$$

and if we had a framerate of 30 FPS we'd get the following (1000ms / 30 FPS = 33.333 msec):

$$Movement = 2.0 = 60.0/(1000.0/33.333)$$

In each case, we end up with 60 units being traversed over the course of 1 second. Easy! Now we can keep everything consistent across variable framerates, by using the formula to calculate framerate independent values for all of our rotations, translations, and anything else we can think up in our games! It's not perfect - rounding errors caused by the inaccuracies of floating point values mean things might be *slightly* off, but it's good enough for now. These inaccuracies can have unexpected consequences - one famous example being that certain trick jumps in *Quake III* could only be performed if the player had a constant framerate of 120FPS!

Example program

No new program this tutorial; instead we're going create a new *class*, and add a **virtual** function to the *Renderer* class you made in the last tutorial. Create a new class called *Camera* to the *NCLGL* solution project - we'll be using the *Camera* class often! For now though, we'll add a Camera to last tutorial's example program, so we can use the view matrix variable in its vertex shader.

Camera header file

Our *Camera* class will make use of the *Matrix4* class and the *Vector3* class, so we must **include** both of their header files. The class has three **protected** member variables - its position in world space, and its yaw and pitch. In case you don't know, pitch is how many degrees up or down something is facing, and its yaw is it's heading. We also have **public** accessor functions for each of these - as the code is trivial, we'll collapse them into the header file. We'll do that with a couple of **constructors** too, one **default**, and one that takes arguments to explicitly set the member variables. Finally, we have two more public functions, *UpdateCamera*, and *BuildViewMatrix*. You'll see what these do shortly.

```cpp
#pragma once

#include "Window.h"
#include "Matrix4.h"
#include "Vector3.h"

class Camera   {
public:
   Camera(void){
      yaw   = 0.0f;
      pitch = 0.0f;
   };

   Camera(float pitch, float yaw, Vector3 position){
      this->pitch    = pitch;
      this->yaw      = yaw;
      this->position = position;
   }

   ~Camera(void){};

   void UpdateCamera(float msec = 10.0f);

   Matrix4 BuildViewMatrix();

   Vector3  GetPosition() const        { return position; }
   void     SetPosition(Vector3 val)   { position = val; }

   float    GetYaw()   const           { return yaw; }
   void     SetYaw(float y)            { yaw = y; }

   float    GetPitch() const           { return pitch;}
   void     SetPitch(float p)          { pitch = p; }

protected:
   float    yaw;
   float    pitch;
   Vector3  position;   //Set to 0,0,0 by Vector3 constructor ;)
};
```

Camera.h

Camera class file

First up - *UpdateCamera*. This will read in the mouse and keyboard input, and update the member variables accordingly. It's not *quite* as easy as you'd first think, but nothing too difficult. First, we read in the *pitch* from the mouse y axis (the up and down movement of the mouse), and yaw from the x axis (the sideways movement) - the *GetRelativePosition* function of the NCLGL *Mouse* class returns how much the mouse has moved since the last game frame. We then lock the *pitch* variable to be between 90 and -90 degrees - just like in an FPS game. We also do a bit of sanity checking on the *yaw* variable, to keep it within the range 0 to 360 degrees.

```cpp
#include "Camera.h"

void Camera::UpdateCamera(float msec)  {
   pitch    -= (Window::GetMouse()->GetRelativePosition().y);
   yaw      -= (Window::GetMouse()->GetRelativePosition().x);

   pitch = min(pitch, 90.0f);
   pitch = max(pitch,-90.0f);

   if(yaw < 0)       {
      yaw += 360.0f;
   }
   if(yaw > 360.0f)  {
      yaw -= 360.0f;
   }
```

Camera.cpp

To move the camera about the game world, we're going to use the common *WASD* keyboard input for the x and z axis. To do so we're going to form a rotation matrix using the *yaw* variable, and use it to multiply a direction vector pointing down the negative z axis - rotating it to point the direction the camera is facing. The vector is then multiplied by the *msec* variable to scale it by the frame rate, and added to the camera *position* variable.

```cpp
   if(Window::GetKeyboard()->KeyDown(KEYBOARD_W)) {
      position += Matrix4::Rotation(yaw, Vector3(0.0f,1.0f,0.0f)) *
                  Vector3(0.0f,0.0f,-1.0f) * msec;
   }
   if(Window::GetKeyboard()->KeyDown(KEYBOARD_S)) {
      position -= Matrix4::Rotation(yaw, Vector3(0.0f,1.0f,0.0f)) *
                  Vector3(0.0f,0.0f,-1.0f) * msec;
   }
```

Camera.cpp

In case you don't quite follow, here's how this works. In the case where we have a *yaw* value of 0°, the vector (0,0,-1) stays the same, and the position variable will move down the negative z axis. However, if we were to have a *yaw* value of 45°, the vector (0,0,-1) would be rotated to a value of approximately (-0.70761,0,-0.70761), meaining the movement would be diagonal. If yaw were 90°, the vector would equal (-1,0,0), and movement would instead be down the negative x axis.

For the sideways movement, we do much the same. This time, however, the *Vector3* being rotated has a non-zero x axis, rather than the z axis.

```cpp
   if(Window::GetKeyboard()->KeyDown(KEYBOARD_A)) {
      position += Matrix4::Rotation(yaw, Vector3(0.0f,1.0f,0.0f)) *
                  Vector3(-1.0f,0.0f,0.0f) * msec;
   }
   if(Window::GetKeyboard()->KeyDown(KEYBOARD_D)) {
      position -= Matrix4::Rotation(yaw, Vector3(0.0f,1.0f,0.0f)) *
                  Vector3(-1.0f,0.0f,0.0f) * msec;
   }
```

Camera.cpp

To move up and down, we're going to use *shift* and *space*, respectively. We want the camera to move up and down no matter what its orientation is, making an easy calculation - we just increment or decrement the y axis by *msec*, giving us 1000.0 units per second of y axis movement.

```
32    if(Window::GetKeyboard()->KeyDown(KEYBOARD_SHIFT)) {
33        position.y += msec;
34    }
35    if(Window::GetKeyboard()->KeyDown(KEYBOARD_SPACE)) {
36        position.y -= msec;
37    }
38 }
```

Camera.cpp

The *BuildViewMatrix* function will form the view matrix required for our vertex shaders. As explained earlier, this view matrix is the *inverse* of the matrix created from the camera's position and rotation values. However, inverting a 4·4 matrix is very computationally expensive - look up the code somewhere, it's nasty! We could use the **inverse** function of GLSL to hide the tricky details of inverting a matrix if we wanted, but instead we're going to do something a little bit sneaky. We can just *negate* the *pitch*, *yaw*, and *position* of the camera when forming the matrix, which does just the same as the inverting the matrix, but is a far cheaper operation!

We build the actual matrix using three temporary matrices - two to rotate by the negated pitch and yaw, and one to translate by the negated position. Remember, the order of matrix multiplications is important! If we had the translation matrix before the rotation matrices, we'd be rotating and translating *around* the origin at a distance of the camera position, not rotating *at* the camera position. Also, take note of the axis each of the rotations is performed around.

```
39 Matrix4 Camera::BuildViewMatrix()  {
40    return   Matrix4::Rotation(-pitch,  Vector3(1,0,0)) *
41             Matrix4::Rotation(-yaw,    Vector3(0,1,0)) *
42             Matrix4::Translation(-position);
43 };
```

Camera.cpp

Using the class

With the class successfully compiled in your *NCLGL* project, you can start to use it in your programs. Add a *Camera* member variable to the *Renderer* class for the last tutorial. You also need to implement the **virtual function** *UpdateScene*, inherited from the *OGLRenderer* class. It has the following function signature:

```
1 virtual void UpdateScene(float msec);
```

Renderer.h

As you can see, it takes in a single **float** as a parameter. This will represent the number of milliseconds that have passed since the last time *UpdateScene* was called. Then, in your *Renderer* class file, add the following function definition:

```
1  void Renderer::UpdateScene(float msec)  {
2     camera->UpdateCamera(msec);
3     viewMatrix    = camera->BuildViewMatrix();
4 }
```

Renderer.cpp

This function's purpose should be a bit clearer, now. It updates your new *Camera* class with the appropriate number of milliseconds, and then builds a new view matrix, ready for sending to your shaders. As you saw in the last tutorial, the code to send a matrix to a shader is a bit unwieldy, so we're also going to create a new **protected** function in *OGLRenderer*, called *UpdateShaderMatrices*,

with the following code:

```
1 void OGLRenderer::UpdateShaderMatrices()  {
2    if(currentShader) {
3       glUniformMatrix4fv(
4       glGetUniformLocation(currentShader->GetProgram(),
5       "modelMatrix"),1,false, (float*)&modelMatrix);
6
7       glUniformMatrix4fv(
8       glGetUniformLocation(currentShader->GetProgram(),
9       "viewMatrix"),1,false, (float*)&viewMatrix);
10
11      glUniformMatrix4fv(
12      glGetUniformLocation(currentShader->GetProgram(),
13      "projMatrix"),1,false, (float*)&projMatrix);
14
15      glUniformMatrix4fv(
16      glGetUniformLocation(currentShader->GetProgram(),
17      "textureMatrix"), 1,false, (float*)&textureMatrix);
18   }
19 }
```

OGLRenderer.cpp

The OpenGL function calls should be fairly familiar to you - we used them last tutorial to set the model and projection matrix. This time we're going to set the *view* and *texture* matrices, too, in exactly the same way (you'll see how to use the texture matrix next tutorial!). This is just a way to cut down the amount of repetitive code in your *Renderer* class. You can use it instead of the two calls to **glUniformMatrix4fv** starting on line 28 in tutorial 2, handily also updating our new view matrix!

Finally, we must call *UpdateScene* in our game loop, so modify your loop in Tutorial2.cpp as follows:

```
1    while(w.UpdateWindow()
2    && !Window::GetKeyboard()->KeyDown(KEYBOARD_ESCAPE)){
3       //Tutorial 2 code goes here...
4       ...
5       ...
6       renderer.UpdateScene(w.GetTimer()->GetTimedMS());
7       renderer.RenderScene();
8    }
```

Tutorial2.cpp

We now call *UpdateScene*, but how does it get the correct number of milliseconds? Fortunately, the NCLGL has a *Timer* class that is instantiated behind-the-scenes when a *Window* is created. The *Timer* class is a small wrapper around the high performance counter on your CPU, which additionally keeps track of the number of milliseconds since the the last call to its function *GetTimedMS*, which it calls when it is first instantiated. So, as long as you only use the timer to time one thing, in this case the time since the last frame, you'll always have an accurate counter to use to keep your *Camera* moving at the correct speed.

Tutorial Summary

If you re-run your second tutorial program after making the changes, you should be able to switch to perspective mode, and move around your simple triangle scene using the mouse to alter your pitch and yaw, and the *W*, *A*, *S*, *D*, *Shift*, and *Space* keys to move around the scene. Not too exciting, but you now know what a view matrix is, how to use it, and how to keep your calculations correct even when the framerate is variable. Next tutorial, you'll see how to use texture maps to make your geometry look more realistic, and also take a look at how to use the *texture* matrix.

Further Work

1) What happens if you multiply the view matrix by a scaling matrix? Does it do what you thought it would?

2) How about if you change the order of the rotations in the *BuildViewMatrix* functions?

3) The *Camera* class currently uses a simple value of *msec* to work out how far to move. This might be too slow or too fast for your game, so try adding in a *speed* member variable to the class - you could even have a separate speed per axis! How would you use the *speed* variable with the *msec* parameter of *UpdateCamera*?

4) The Camera class currently creates view matrix rotations about the X and Y axis. Try adding in a *roll* member variable to the *Camera* class, which rotates about the Z axis. Does it do what you thought it would? How about when you roll 90 degrees?