Assignment 4 – Cameras & Materials

Due by 11:59PM on Thursday, February 7th.

# Overview

You’ve (hopefully) set up an entity class and a mesh class by now. You’re going to continue fleshing out your engine by adding a Camera, as well as a basic Material system. The camera simply contains a view and projection matrix, and the smarts to update them as necessary. You’ll also need input (mouse and keyboard) to control the camera.

Materials will define the look of a particular mesh when it’s drawn, and can be shared among multiple game entities. While the diagram below shows that materials will eventually have textures, you’ll implement that piece (as well as texture mapping) in a future assignment.

Coming Soon

This assignment

# Task Overview

Here is the high level overview of tasks, which are explained in more detail on the pages that follow:

* Create a **camera class** that acts like a first person flying camera (with user input)
* Make sure the camera class properly updates its **view and projection matrices**
* Create a **material class** that represents a set of vertex and pixel shaders
* Update your entity class to also keep track of **its own material**
* Ensure you have no **warnings**, **memory leaks** or **DX resource leaks**

# Tasks

## Create a Camera Class

**Camera Data**

The camera defines where the viewer is in a 3D scene, as well as how that 3D scene is mapped onto a 2D screen. The first part, describing the viewer of the scene, is handled by a view matrix. The second half, mapping back to a 2D scene, is handled by a projection matrix. Both of these matrices will need to be created and maintained by the Camera class, so make sure you have appropriate **fields** and **get methods** for them.

The view matrix will update anytime the user wants the camera to move or rotate, and can be created easily in one of two ways: as a *look at* matrix, or a *look to* matrix. A *look at* camera would keep track of its current position and another position in space it’s looking at. A *look to* camera would keep track of its current position and a “forward” direction in space. Since you’ll be creating a basic first-person camera for this assignment, the *look t*o camera makes the most sense (although either could work depending on how to handle the math).

To create the *look to* view matrix, you’ll need to keep track of a few more fields:

* Camera’s position (vector3)
* Camera’s direction (vector3)
* Rotation around the X axis (float)
* Rotation around the Y axis (float)

A first-person camera is going to rotate around the X and Y axes, but not the Z. However, if you rotate around X, then Y, then X, then Y, etc. you’ll begin to introduce a roll around the Z axis. This isn’t ideal when you want precise control. A better way to approach the camera is to keep track of the raw rotation values on the X and Y, and recreated the matrix from those values each frame. This also allows you to easily put limits on the X rotation, so your camera doesn’t get flipped (turned upside down).

**Update Method**

Create an Update() method in the camera class. For now it’ll update the camera’s view matrix based on the current X and Y rotations (you’ll be adding to the method as we go). Inside the method, create a rotation matrix or quaternion based on the current X and Y rotation values. You can create two rotation matrices (one on each axis) and combine them, or simply use the XMQuaternionRotationRollPitchYaw() method. Apply your resulting matrix or quaternion to the default forward vector (a unit vector on the Z axis) to get the camera’s direction for the view matrix. You might find XMVector3Transform() or XMVector3Rotate() useful for this.

Lastly, call the XMMatrixLookToLH() method, passing in the camera’s current position, the direction from the step above, and the up vector. Store the result as the camera’s current view matrix.

**Keyboard Input**

Camera movement (forward, back, strafe, etc.) is usually controlled by the keyboard. You can ask Windows about the current state of a particular key using the global GetAsyncKeyState() function in the Windows.h header. This means you could even put this logic directly into the camera class if you wish. (A more advanced engine might have an input manager that gets key states once per frame, and provides a simple interface for the rest of the code base to access that information. This is a common place to use the Singleton design pattern.)

A quick example of determining if the “W” or “S” keys are currently being pressed:

if( GetAsyncKeyState('W') & 0x8000 ) { /\* Do something useful \*/ }

if( GetAsyncKeyState('S') & 0x8000 ) { /\* Do something useful \*/ }

You need the “*& 0x8000”* part because the return value from the function contains multiple pieces of information packed into a single *short* value. More information on GetAsyncKeyState() can be found here: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms646293(v=vs.85).aspx>

**Implementing Input**

Begin adding some input logic to your camera’s Update() method. This could be directly in the method or in a helper method. Here is one set of potential keys and their usage, but definitely feel free to customize. These match how flying mounts are controlled in World of Warcraft and similar games:

* W, S – Forward or backwards (based on your current rotation! NOT just pos.z += amount)
* A, D – Strafe left or right (based on your current rotation! NOT pos.x += amount)
* Spacebar - Move up along the world’s Y axis (regardless of your rotation: pos.y += amount)
* X - Move down along the world’s Y axis (regardless of your rotation: pos.y -= amount)

The only tricky part is the “based on your current rotation” piece. When pressing W, for instance, you’d probably like to move along the direction that the camera is currently facing. Luckily you should already have that vector, as it was needed for the view matrix. Backwards is just the negation of that vector. Left/right vectors can be found using the cross product of forward (camera’s direction) and the world’s up axis (0,1,0).

You’ll probably want to scale your actual movement speed by delta time so your movement is independent of frame rate.

**Testing the Camera**

You should be just about ready to test the camera now. Create a camera field in Game.cpp and call Update() on it during the update phase. Pass this camera’s view matrix into the vertex shader in the draw phase rather than the existing view matrix.

In essence, you’ve just “switched” cameras. In fact, when a game switches cameras, all it’s doing is drawing the objects in the world with a different view & projection matrix.

**Mouse Input**

Mouse input is a little trickier, but you may have noticed some helper functions that come with the starter code. At the very bottom of Game.cpp are three mouse-related methods:

* OnMouseDown() – Called when a mouse button is pressed
* OnMouseUp() – Called when a mouse button is released
* OnMouseMove() – Called when the mouse cursor moves

Mouse information can be captured from the messages that the Windows OS is passing around to each active window, which is what the starter code is doing. The DXCore class is then calling the aforementioned methods whenever it detects mouse input, passing in some relevant information. The x and y parameters are the cursor’s current position. The buttonState parameter contains information on the state of various mouse buttons (left, right and middle), and can be used like so:

if( buttonState & 0x0001 ) { /\* Left button is down \*/ }

if( buttonState & 0x0002 ) { /\* Right button is down \*/ }

if( buttonState & 0x0010 ) { /\* Middle button is down \*/ }

Create two methods (or a method with two parameters) for rotating the mouse on the X or Y axis. It can simply update the camera’s raw X and Y rotation values (limiting or wrapping rotations as necessary). You may also want to scale the values, as [1 pixel = 1 radian] is probably not an ideal mouse-look speed.

The difference between the cursor’s position this frame and its position last frame is the rotation amount on the various axes. Remember that moving the mouse on the X axis should actually rotate the camera on its Y axis, and vice versa. Call your new method(s) directly from the mouse input helper methods in Game.cpp.

Since we’re relying on Windows for mouse input, and since we won’t be getting a message each and every frame, you generally *don’t* need to scale your mouse rotation by delta time here.

**Projection Matrix**

One last thing. Your camera class should have a method that updates the projection matrix, which you can call as necessary. Feel free to copy/paste the code that already exists in the starter files for creating a projection matrix. The method should take the new aspect ratio, and simply re-create the projection matrix. Make sure you call this method at least once at start up, and again whenever the window is resized.

Make sure that you’re also passing in this camera’s projection matrix to the vertex shader. Test the projection matrix by running the program and resizing the window. The objects will probably change in size, but they shouldn’t become squashed or stretched.

## Create a Material Class

Materials define how the surface of a particular mesh looks when it’s drawn: the shaders, the textures and various other drawing-related data. Your next job is to create a simple *material* class that will begin to wrap this kind of data, so that a single material definition can be shared by multiple game entities.

Again, this can be a dumb container of various DirectX structures, or it could have some smarts of its own. I’m going to describe the “dumb container” version here, but feel free to adjust the architecture if you’d like. For this assignment, your material class will only need a pixel shader pointer and a vertex shader pointer. They should be passed in to the constructor of the material, stored in fields and have corresponding Get methods. Remember that we only want to load these resources once and share them among various materials and entities.

## Connecting Materials & Entities

You only have one of each shader in your project at this point, so the material class may seem like overkill. The next few assignments will be dealing with shader code specifically, so this material system is laying the groundwork for that.

**Update your Entities**

Now that you can make materials, you’ll need to edit your Game Entity class so it keeps track of which material it will be using. This will require you to keep a pointer to material as a field, as well as accepting the material pointer in the constructor.

Right before drawing your entity, you’ll need to interact with the material’s shaders: setting data and activating the shaders themselves. While this can be done manually in Game’s Draw() for each entity, it’s much cleaner to create a method to prepare the material & shaders for the upcoming draw.

I’d suggest creating a method called PrepareMaterial() in the Entity class. It should accept the two camera matrices (view & projection) as parameters, set the basic vertex shader data (world, view and projection matrices), and then call SetShader() and CopyAllBufferData() on both the pixel and vertex shader. This will activate the shader in DirectX and also copy the local shader data to the GPU. Ensure that you’re calling these methods prior to drawing each and every entity.

**Other Architectural Options**If your entity class already has its own Draw() method, you could instead alter that method to accept the two camera matrices. Then, right before the actual DirectX draw call, you can use PrepareMaterial() to set things up.

An advanced engine might offload the material setup to a Renderer class. The Renderer might sort objects by material (or by mesh/material combinations) so it doesn’t have to swap shaders between each draw call.

## Finally

Test your program one last time. Be sure that your objects still draw and move properly with the material updates you’ve made. Your camera should also still work (mouse look, keyboard movement and projection matrix updates).

# Deliverables

Submit a zip of the entire project to the appropriate dropbox on MyCourses.