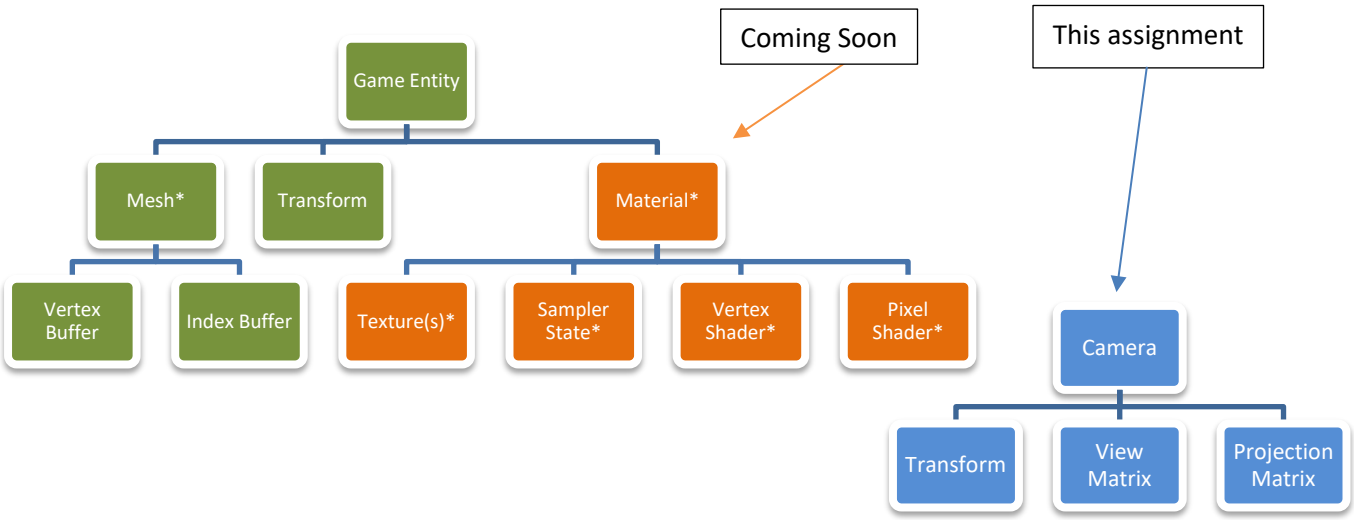
Assignment 5 – Cameras

Overview

You've set up an entity class and a mesh class. You're going to continue fleshing out your engine by adding a **camera class** that contains a transform, some user input processing and the smarts to create view and projection matrices.



Task Overview

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

- ❑ Update your Transform class to support **relative movement**
- ❑ Create a **Camera class** that acts like a first-person flying camera (with user input)
  - ❑ Ensure the class properly updates its **view and projection matrices**
- ❑ Update the **vertex shader** and **associated structs** to include camera matrices
- ❑ Create a **single camera** to ensure it works
- ❑ Add support for **multiple cameras**
  - ❑ Create at least **two different cameras** in your scene
  - ❑ Track which one is the **active camera**
  - ❑ Allow the user to **swap cameras** with ImGui
  - ❑ Use the current camera when **drawing**
  - ❑ Display **camera details** for the active camera
- ❑ Ensure you have no **warnings**, **memory leaks** or **DX resource leaks**

Task1: Transform Updates

Usually, we want 3D a camera to be able to move forward based on its current orientation. When we're writing the movement code, how do we know which direction is forward? If the camera hasn't rotated at all, its forward vector is probably still (0,0,1). As soon as any rotation occurs, however, its forward vector could be anything; hardcoding *forward movement* as a simple change of Z position is **incorrect**.

`MoveRelative(float x, float y, float z)`

Create a `MoveRelative()` method in your Transform class. The overall goal is to adjust the current position of the transform by the specified amounts *relative to the transform's rotation*. This means you **cannot** just add the values directly.

To facilitate this, get these values into an XMVECTOR. Next, you'll need to create a quaternion that represents the transform's current rotation. This can easily be done with `XMQuaternionRotationRollPitchYaw()`, which will return another XMVECTOR.

Now that you have an absolute direction and a rotation, you can rotate that direction using `XMVector3Rotate()`, which conveniently takes a direction vector and a quaternion. The resulting vector from this function is the direction you should actually move the transform. Load the existing position with `XMLoadFloat3()`, add it to your new direction and store it with `XMStoreFloat3()`.

`GetRight(), GetUp(), GetForward()`

In addition to relative movement, you might need to simply retrieve the direction an object is facing. For instance, to cast a physics ray to see what a projectile will hit, you'd need its *forward vector*. This vector is not just the world's positive Z axis (0,0,1), since the projectile might have any orientation.

To this end, create the following methods: GetRight(), GetUp() and GetForward(). Each will take no parameters and return an XMFLOAT3. Much like MoveRelative() above, each of these functions will take a particular vector and rotate it by the transform's orientation. These will then return the result. The vector that each function starts with will depends on the purpose of that function:

- GetRight() – Rotate the world's right vector (1,0,0) by the transform's pitch/yaw/roll.
- GetUp() – Rotate the world's up vector (0,1,0) similarly.
- GetForward() – Rotate the world's forward vector (0,0,1) similarly.

If you'd like to be more efficient, you could track these vectors with fields of the class and only update them when the rotation changes. Then these getters would simply return the corresponding field.

Task 2: Camera Class

Create a Camera class in your project. Since you'll need it for user input later on, be sure to include the `"Input.h"` header file in Camera.cpp.

Fields

Since the camera will need to move and rotate, it makes sense for the camera to have its own **Transform** field. It'll also need XMFLOAT4X4 fields for a **view matrix** and a **projection matrix**.

For customization, you could have *float* fields for things like **field of view** angle (radians), **near** & **far** clip plane distances, **movement speed** and **mouse look speed**. You could also have a *bool* (or enum) to track whether the projection is perspective (like our eyes) or orthographic (like an isometric game).

A note on field of view: Please don't go crazy high with this value in your final assignment submission. Something between 45 and 90 degrees would be fine (quarter pi and half pi, respectively, in radians).

Constructor

The constructor will need an **aspect ratio** and an **initial position** at a bare minimum (so it doesn't always start at the origin). Add any other customization options you'd like: things like, **starting orientation**, **field of view**, near/far clip plane distances, **movement speed** and **mouse look speed** are common camera options. Initialize the camera's transform with the appropriate parameters. The constructor should call `UpdateViewMatrix()` and `UpdateProjectionMatrix()`, see below, passing aspect ratio into the latter.

Getters

You'll need a way to get the **view** and **projection** matrices outside the camera, as these will be passed to shader variables. No *set* necessary, as the whole point is to derive these from the camera's data.

`UpdateProjectionMatrix(float aspectRatio)`

This method simply calls one of the DirectX Math projection matrix methods and stores the resulting matrix. This method will only be called at start up (the camera class constructor) and when the window resizes; it should not be called every frame. For a perspective projection, which is our default, use `XMMatrixPerspectiveFovLH()`.

Parameters to `XMMatrixPerspectiveFovLH()` are the Field of View angle (in radians), the aspect ratio, the near clip plane distance and the far clip plane distance. Use a small number like 0.1f or 0.01f for the near clip. Far clip should be a small as possible while still being big enough for your scene; usually no larger than 1000. The aspect ratio is a parameter to UpdateProjectionMatrix(), but the rest of the data can be hardcoded or parameterized as you see fit.

`UpdateViewMatrix()`

This will probably be called once per frame from `Camera::Update()`. It needs to create a view matrix, most likely using `XMMatrixLookToLH()` – that's **Look To**, not Look At. Make sure you also call this once in Camera's constructor to initialize the view matrix.

You need a *position* and a *direction*, as well as a *world up vector* to stabilize the camera's roll. The first two can be grabbed directly from the camera's transform: *position* is the transform's position and *direction* is the transform's forward vector.

`Update(float dt)`

This method is the workhorse of the Camera class. It's where you will process user input, adjust the transform and update view matrix appropriately.

Basic Input

Use the Input singleton class included in the starter code here. Be sure you've included "Input.h". Wherever you want to check for input, start by getting a reference to the input manager's instance:

```
Input& input = Input::GetInstance();
```

Use this object to check for keyboard and/or mouse input as necessary. A quick example of determining if the 'W' or 'S' keys are currently being pressed:

```
if (input.KeyDown('W')) { /* Do something useful */ }
if (input.KeyDown('S')) { /* Do something useful */ }
```

Non-Character Keys

Windows has a concept of "Virtual-Key Codes": values that represent various input across multiple devices like the mouse and keyboard. Here are some examples of checking "virtual keys" that don't have a particular character associated with them:

```
if (input.KeyDown(VK_SHIFT))   { /* Shift is down */ }
if (input.KeyDown(VK_CONTROL)) { /* Control is down */ }
```

The complete list can be found here: https://docs.microsoft.com/en-us/windows/win32/inputdev/virtual-key-codes

Processing Keyboard Input

Begin adding input logic to your camera's `Update()` method. Here is one set of potential keys and their usage, which match swimming & flying mount controls in World of Warcraft. Feel free to customize the exact keys, but do ensure you've got these exact types of movements.

- W, S – Forward or backwards (This is relative movement)
- A, D – Strafe left or right (Also relative movement)
- Spacebar - Move up along the world's Y axis (Absolute movement)
- X - Move down along the world's Y axis (Absolute movement)

Don't forget to scale your actual movement speed by delta time so your movement is independent of frame rate. You could even check modifier keys like shift and control to speed up or slow down the movement this frame, though that's not a requirement.

> *Though you're hardcoding input keys directly inside your camera for this assignment, you may want to refactor this setup for the larger project later, as you might need multiple cameras or cameras that aren't tied to user input. Given a proper GetTransform(), code elsewhere could move a camera.*

Processing Mouse Input

Getting mouse button and positional data is relatively straight forward as well:

```
if (input.MouseLeftDown())
{
   int cursorMovementX = input.GetMouseXDelta();
   int cursorMovementY = input.GetMouseYDelta();
   /* Other mouse movement code here */
}
```

Use the mouse to handle the camera's rotation. Ideally this only happens if one of the mouse buttons is pressed. If so, rotate the camera's transform based on how much the mouse has moved.

To determine how far the mouse has moved since last frame, use `GetMouseXDelta()` and `GetMouseYDelta()`. Scale each of those values by your **mouse look speed**. Since mouse input occurs in "real time" already, there is no need to scale by delta time.

Next, rotate your transform using these final X and Y movement values. However, note that the **rotation parameters go in the opposite order**: moving the mouse in the X direction (left and right) actually rotates around the Y axis (yaw), and moving in the Y direction (up and down) rotates around the X axis (pitch).

While Y rotation can continue infinitely, you probably want to **clamp the X rotation** so the camera doesn't flip upside down. Do that here using the current pitch rotation value from the transform. The limits are ½ pi and -½ pi; if the values go outside that range, clamp them to the appropriate bound.

Wrapping Up the Update() Method

Call `UpdateViewMatrix()` at the end of the camera's `Update()` to ensure the view matrix actually matches the camera's current transform.

Creating & Updating the Camera

The Game class needs a camera field, which should be a shared_ptr as you'll eventually need to pass it around. Create the camera during Game's initialization (or constructor). To calculate the window's aspect ratio, use the following:

```
(float)this->windowWidth / this->windowHeight
```

In `Game::OnResize()`, you'll need to call `UpdateProjectionMatrix()` on the camera to keep the camera's aspect ratio up to date with the window's new aspect ratio.

Lastly, be sure you're updating your camera in `Game::Update()`.

Using View and Projection when Drawing

You can't visually test your camera yet, as you're not using any of its matrices when drawing. To facilitate this, you'll need to get the two camera matrices to the GPU so the vertex shader can use them.

Begin by updating both your C++ struct (`VertexShaderExternalData` or similar) and the cbuffer definition inside VertexShader.hlsl. Both of them will need to accept two more matrices: the view and projection. While in the Vertex Shader, update the final output position to include these new matrices:

```
// Multiply the three matrices together first
matrix wvp = mul(projection, mul(view, world));
output.screenPosition = mul(wvp, float4(input.localPosition, 1.0f));
```

Next, you'll need to copy the camera's matrices to your `VertexShaderExternalData` struct alongside the world matrix and color tint. If your render setup is in `Game::Draw()`, this should be straightforward.

If, however, you've given your GameEntity class its own `Draw()` method, you should update that method to accept a pointer to a Camera (a very common implementation, as 3D rendering requires a camera). Then `GameEntity::Draw()` can fill out the new `VertexShaderExternalData` struct members.

Testing the Camera

If everything is hooked up correctly, you should be able to move around your 3D scene as expected. If you can't see anything at first, ensure you're not starting your camera at the origin (which is on the same plane as your geometry). Adjust your movement and mouse look speeds, as well as any other camera options, as necessary!

Note: Any of your geometry that overlaps will have some Z-fighting at this stage. This is expected, as it all probably has a Z position of zero. If you'd like to fix that, simply adjust each entity's Z position a bit to compensate.

Task 3: Multiple Cameras

One camera is great; more are better. Your next task is to create two (or more) distinct camera objects and allow the user to swap between them at runtime. Below is an overview of the steps to achieve this:

- Store a vector of camera shared pointers now instead of just one
    - Also track which of those cameras is the current (active) camera in a separate variable
- Create multiple cameras during initialization
    - Each should have different starting positions
    - Each should have different a different field of view
    - Store one of them as the active camera
- Update and draw using the *active camera*
- When resizing the window, update the projection matrix of *ALL cameras*

At this point, everything should still work, though only with one of your cameras.

UI Updates – Camera Swap & Details

The last thing to do is update your UI to handle camera details, including the ability to swap between cameras. First, give the user a set of cameras to choose from. This could be a set of radio buttons, a list box, a dropdown menu, "next"/"previous" buttons or some other easy way of swapping.

Upon receiving user input here, change your active camera to the one the user chose. Since you should be storing the cameras in a vector or other collection, it should be easy to link their choice to a particular camera and update your active camera variable. You should be able to swap between the cameras at runtime now!

Add some camera-related details to your UI. You only need to display the data (position, field of view and/or other user information) of the active camera.

Deliverables

Submit a zip of the entire project to the appropriate dropbox on MyCourses. Remember to follow the steps in the “Preparing a Visual Studio project for Upload” PDF on MyCourses to shrink the file size.