Assignment 3 – Making Things Move

Due by 11:59PM on Thursday, January 31st.

# Overview

Last time, you started working with vertex & index buffers by wrapping them up in a Mesh class. The next step is to use the meshes to represent individual entities in your game: things like enemies, projectiles, the environment, etc. Since a single mesh could appear multiple times on the screen, it doesn’t make sense for the Mesh class to keep track of a position (or other transformations). It also doesn’t make sense to have multiple Mesh objects with the exact same geometry in them. Duplicating the same resource in memory is wasteful and unnecessary.

Instead, you’ll create a class that represents a single *game entity*. A *game entity* will have its own individual transformations (position, rotation & scale), in addition to a pointer to the Mesh that represents the entity’s visuals. This allows you to create/load each unique Mesh exactly once and share them among multiple *game entities*, rather than each entity having its own copy of the resources it uses. Eventually your game engine could contain more-specific classes that inherit from the game entity class, or could contain components that define the behavior of the entity.

Here’s the basic structure you’ll be working on over the next few assignments:

This assignment

Already done

Coming soon

# Task Overview

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

* Create a class that represents individual **game entities** and their transformations
* Ensure that Meshes can be **shared** among multiple entities
* **Create** and **draw** at least 5 entities, with several sharing the same Mesh object
* **Update** your entity transformations each frame so the entities move/scale/etc.
* Ensure you have no **warnings**, **memory leaks** or **DX resource leaks**.

# Tasks

## Basic Game Entities

Create a *game entity* class. You can name it anything you’d like: Entity, GameEntity, Renderable, ThingWhichGoesOnTheScreenWithDirectX, etc.

**Fields**

In terms of fields, it should have a world matrix, individual position & scale vectors (math vectors, not std::vectors), some way of tracking current rotation (a quaternion perhaps, or even just 3 separate x/y/z rotation values) and a pointer to a Mesh object. Remember to define these fields as either XMFLOAT4X4 (for matrices) or XMFLOAT3 (for 3-component vectors) in the header.

The world matrix will need to represent the entity’s current position, rotation and scale. Rather than attempting to alter an existing matrix, especially one that has rotations and scales in it, it’s much easier to store the position, rotation and scale as raw numbers and recreate the world matrix as necessary.

The easiest, although least efficient, time to recalculate the world matrix is anytime the position, rotation or scale changes. Feel free to do it this way for now. However, chances are that the position and rotation will probably both change on the same frame. Here are a few other ways of approaching the problem:

* Since the world matrix isn’t actually used until the draw part of the game loop, it would be more efficient to simply recreate the matrix once per frame, after all possible updating occurs.
* An even more optimized version might keep track of whether or not any of the transformations have changed this frame, meaning the matrix is “dirty” and needs an update. You would then only update the world matrix once, and only on frames where something changed.

**Constructor**

The constructor should accept the Mesh pointer and set default values for the matrix & transformations.

**Movement Methods**

You’ll need some methods to alter the entity’s transformations (raw position, rotation & scale values). Get as in-depth as you’d like with these. You’ll probably need at least one of the following:

* Get and set methods for the matrix & transformations themselves
* Methods to directly set one or more components of each transformation
  + Like SetTranslation(), SetScale(), etc.
  + These would simply accept float values
* Methods to offset the transformations by some amount:
  + For example: a Move() method to offset the position from its current value
  + Even fancier: MoveForward() – changes position with respect to the current rotation

**Drawing Methods**

You have a few options for how drawing-related methods might work:

* Much like the Mesh class, your entity class could be a dumb container for the data to use when drawing, by way of various “get” methods in the class. This would require you to still put a lot of drawing-related code in Draw(), or a master Renderer class.
* Or the entity class could have its own Draw() method that takes care of setting buffers and issuing draw commands, which would require passing in the ID3D11DeviceContext or storing it as another field in the class. This would simplify the code in Draw() a bit, but makes more complex render decisions harder to code.

A note on drawing: An advanced engine might put most drawing code inside a *Renderer* class that makes decisions about what to render and when, performing common tasks like changing buffers as few times as possible and sorting objects based on distance. This is not something I expect you to do for these assignments, but it could come in handy for the group project later on.

**Entity Destructor**

What exactly does the entity need to clean up? If an Entity instantiates an object internally (meaning it calls *new* somewhere), then it should clean up (*delete*) that object. You won’t necessarily be doing that for this week’s assignment.

However, you will be sharing Meshes among entities. For example: 5 different entities could know about a single Mesh. Each entity should **not** be deleting that Mesh, as then you’ll be trying to delete the single Mesh 5 times (which doesn’t work so well).

Ok, so who is in charge? One rule of thumb you could go by says that the class which instantiates an object should also be the one that deletes it. In this case, your Game class is creating all of the Mesh objects, so the Game class’s destructor should be in charge of cleaning them up.

To get fancier, you could use shared pointers or some other kind of reference counting yourself. For these assignments, that would be unnecessary, but feel free to experiment if you’d like. The main goal is to cleanly prevent memory and DirectX leaks.

## Testing the Entities

Over in the Game class, create a few different entities using the various meshes from the previous assignment. You should have several entities share a single Mesh object to ensure sharing the resources works. You may find it useful to put the entities in an array or std::vector, which you can loop through when drawing. This makes deleting them at the end of the program a little easier too.

Ensure your entities are drawing to the screen. At this stage, they might all be “on top” of each other. You’ll fix that next.

## Movement

When moving objects around the screen, you do **not** update the vertices in a Mesh’s vertex buffer. That would be far too time-consuming, especially if you’re using the same Mesh for multiple entities that are at different places on the screen. Instead, we need to leverage the parallel nature of the GPU to alter each vertex’s position as it passes through the vertex shader.

This is done by passing the entity’s world matrix into the shader before the draw command. During the draw, the vertex shader uses the world matrix as part of its final position calculation. Since each vertex of a single mesh will use the same world matrix, they will all experience the same transformation, keeping the same relative shape.

Now that you have some game entities, get them moving. The **first step** to this is updating one or more transformations of each entity during Update(). You could make them move back and forth with a sine wave, rotate in place or simply just fly off the screen. Be sure to scale any transformations by the delta time parameter of Update(). This will ensure your transformations aren’t bound to the frame-rate, which can fluctuate as the program runs.

The **second step** of getting your objects to move is to pass each entity’s world matrix to the shader before drawing that entity. This will require you to do a few things before drawing *each* entity in Draw(), since each will most likely have a different world matrix:

* First, set the world matrix variable of the vertex shader using SetMatrix4x4(). This is a method from the SimpleShader class that I provided. Under the hood it’s caching the data you intend to pass to the shader until you’re ready to send it all to the GPU. It’ll look similar to this:

vertexShader->SetMatrix4x4(“world”, yourMatrixHere);

* Lastly, you must instruct the SimpleShader to actually copy the data to the GPU. Sending data to the GPU takes time, so it’s more efficient to send it all at once. The call to CopyAllBufferData() is what actually causes data to be sent, in one big chunk, over to GPU memory.

We’ll discuss the SimpleShader class (what it does for us and why we might want that) and the basics of sending data to a shader without SimpleShader soon.

## Misc. Additions

**Keyboard Input**

You don’t have to have keyboard input for this assignment. If you’d like to go that route however, look up the GetAsyncKeyState() function from the Windows.h header. It’s a simple way of determining if a particular key is currently down. More info here: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms646293(v=vs.85).aspx>

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 \*/ }

# Deliverables

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