Assignment 4 – Making Things Move

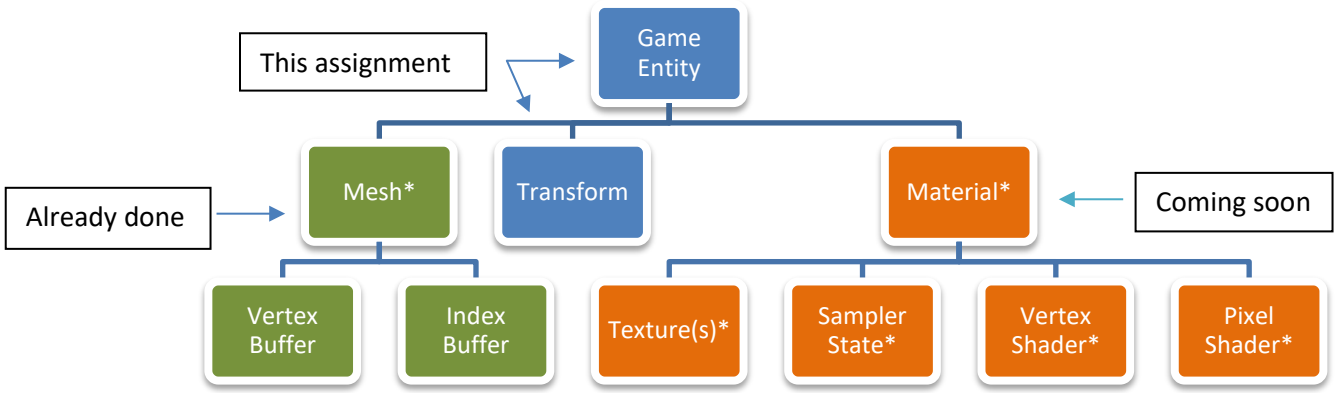
Overview

You previously worked with vertex & index buffers by wrapping them up in a Mesh class. The next step is to represent individual entities – things like platforms, projectiles or environmental objects – in your engine, many of which share the same mesh. Since each mesh might appear multiple times in a scene, it doesn't make sense for the Mesh class to keep track of a transformation. It also doesn't make sense to have multiple Mesh objects with the exact same geometry, as duplicating the same resource is wasteful.

Instead, you'll create a class to represent *transformations* (position, rotation & scale) and another to represent a single *game entity*. A *game entity* will have its own transform object 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 duplicating the resources it uses.

> *A more complex game engine might contain classes that inherit from the game entity class, or instead provide components that define the behavior of an entity.*

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



Task Overview

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

- ❑ **Alter** your C++ constant buffer struct & associated cbuffer to take a **matrix** instead of an offset
- ❑ Create classes that represents individual **game entities** and their **transformations**
- ❑ **Create** and **draw** at least 5 entities, several of which **share the same Mesh** object
- ❑ **Update** entity transformations each frame so the entities move/scale/etc.
- ❑ Display and edit entity details using **ImGui**
- ❑ Ensure you have no **warnings**, **memory leaks** or **DX resource leaks**.

Constant Buffer Alterations

For this assignment, you'll be calculating an entire transformation matrix – often called a *world matrix* – for each entity you intend to render. Though the calculation of this matrix will occur inside a transformation class of some sort, you'll need to get that matrix down to the Vertex Shader to make use of it when rendering. This means it'll need to go through your constant buffer.

C++ Constant Buffer Struct

Go to your C++ constant buffer struct, probably in a header file like BufferStructs.h. Replace the **XMFLOAT3 offset** with an **XMFLOAT4X4 worldMatrix**. The exact name is up to you, though "world" is a standard term for this specific matrix.

> *Since the sizes of both the world matrix and the remaining color tint member are multiples of 16 bytes, there shouldn't be any alignment issues.*

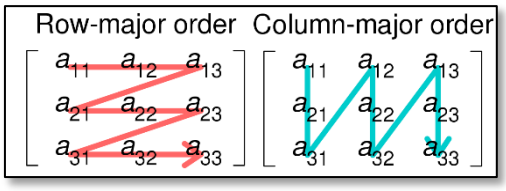
Vertex Shader cbuffer

Next, you'll need to update your cbuffer in VertexShader.hlsl to match your C++ struct. Replace the **float3 offset** here with a **matrix world**. The data type is just "matrix", though "float4x4" is equivalent.

Vertex Shader Transformations

Since you've removed the offset variable, you'll need to adjust the Vertex Shader code to instead use the new world matrix.

> *Note: Matrices in DirectX Math use row-major storage, while matrices in HLSL use column-major storage. This means once the shader starts accessing any matrix data we send to the GPU, it'll be in a transposed form (reflected across the diagonal). We can account for this by either transposing the matrix ourselves in C++ (nope), or by reversing the order of our matrix multiplication in the shader (yup).*



To those ends, the line of code in your vertex shader that calculates the vertex's output screen position should now look like this:

```
output.screenPosition = mul(world, float4(input.localPosition, 1.0f));
```

You won't be able to really test your program until you progress further in this assignment, since you're not providing the matrix yet, but you should not have any compilation errors at this point.

Transformation Class

Create a *transformation* class. The exact name is up to you, though **Transform** is what I'd recommend.

Fields

In terms of fields, it should have individual **position** & **scale** vectors (math vectors, not std::vectors), two matrices (**world** and **worldInverseTranspose**) and some way of tracking **rotation** (individual float values for pitch/yaw/roll rotation stored as x/y/z in a 3D vector, or a quaternion instead). Define these fields as *XMFLOAT4X4* for matrices, *XMFLOAT3* for 3-component vectors or *XMFLOAT4* for quaternions.

> *Reminder: these data types are in the DirectX namespace.*

The world matrix represents 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 far easier to store the position, rotations and scale each as raw float values and recreate the world matrix when necessary.

The easiest, although least efficient, time to recalculate the world matrix is whenever any of the position, rotation or scale values change. Feel free to do it this way for now. However, chances are that multiple of these will change on the same frame. Here are other ways of approaching this:

- Since the world matrix isn't actually used until the draw phase 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 would track whether or not any of the 3 major transformations have changed, meaning the matrix is "dirty" and needs an update. You would then update the matrix right before it's actually returned by a GetWorldMatrix() method, and only if it's dirty.

Constructor

The constructor should initialize all transformation values to default values. Remember that the default scale should be (1,1,1), not (0,0,0).

You should also initialize the matrices, which can be done similarly to this:

```
XMStoreFloat4x4(&worldMatrix, XMMatrixIdentity());
XMStoreFloat4x4(&worldInverseTransposeMatrix, XMMatrixIdentity());
```

Methods

You are required to implement the following methods as part of your transformation class, as well as any other methods you might find useful. Specific information precedes each set of methods below.

Setters: These are basic *setters* and, as such, should *overwrite* (replace) the existing raw transform values. Create versions that both take the raw floats and existing vectors.

> *Notice there is no explicit setter for the world matrix: it should always be derived from these values! Depending on your implementation, world matrix creation could happen directly within these methods or in the matrix's getter. See later in this document for specifics of creating a world matrix.*

- `void SetPosition(float x, float y, float z)`
- `void SetPosition(DirectX::XMFLOAT3 position)`
- `void SetRotation(float pitch, float yaw, float roll)`
- `void SetRotation(DirectX::XMFLOAT3 rotation) // XMFLOAT4 for quaternion`
- `void SetScale(float x, float y, float z)`
- `void SetScale(DirectX::XMFLOAT3 scale)`

Getters: These *getters* should mostly just return values. The matrix's getter, however, could also *create the world matrix* when, and if, it is necessary.

- `DirectX::XMFLOAT3 GetPosition();`
- `DirectX::XMFLOAT3 GetPitchYawRoll(); // XMFLOAT4 GetRotation() for quaternion`
- `DirectX::XMFLOAT3 GetScale();`
- `DirectX::XMFLOAT4X4 GetWorldMatrix();`
- `DirectX::XMFLOAT4X4 GetWorldInverseTransposeMatrix();`

Transformers: These methods adjust the existing, raw transformation values by the specified amount. MoveAbsolute(), for instance, *adds* the specified x, y and z values to the existing position. Rotate() will do the same for rotational values. Scale(), on the other hand, should *multiply* by the given values.

> *Note: It's called MoveAbsolute() because it should **not** take the object's orientation into account. A future assignment will have you implement a MoveRelative() method that will do just that.*

- `void MoveAbsolute(float x, float y, float z)`
- `void MoveAbsolute(DirectX::XMFLOAT3 offset)`
- `void Rotate(float pitch, float yaw, float roll)`
- `void Rotate(DirectX::XMFLOAT3 rotation)`
- `void Scale(float x, float y, float z)`
- `void Scale(DirectX::XMFLOAT3 scale)`

Making a World Matrix

You'll need to create your matrices somewhere in this class. An `UpdateMatrices()` helper function would be great here; call it in the getters for matrices. To actually create a world matrix, create separate translation, rotation and scale matrices, multiply them together and store the result. You'll find the following functions of the DirectX Math library useful for this, all of which return an XMMATRIX:

- XMMatrixTranslation() – Creates a translation matrix
- XMMatrixScaling() – Creates a scale matrix
- XMMatrixRotationRollPitchYaw() – Creates a rotation matrix from pitch/yaw/roll values.
    - The parameters go pitch/yaw/roll (which map to x/y/z axis rotations)
    - The method name refers to the order rotations are applied: roll, then pitch, then yaw.

You generally want to apply Translation, then Rotation, then Scale. However, given the way these matrices are represented internally, your multiplications will be written in the opposite order, like so:

```
// Note: Overloaded operators are defined in the DirectX namespace!
// Alternatively, you can call XMMatrixMultiply(XMMatrixMultiply(s, r), t))
XMMATRIX world = scale * rotation * translation;
```

Store the final XMMATRIX as an XMFLOAT4X4 like so, then store the inverse transpose of it:

```
XMStoreFloat4x4(&worldMatrix, world);
XMStoreFloat4x4(&worldInverseTransposeMatrix,
	XMMatrixInverse(0, XMMatrixTranspose(world)));
// You'll use the inverse transpose matrix in a future assignment!
```

Basic Game Entities

Create a *game entity* class. The exact name is up to you: Entity, GameEntity, Renderable, etc.

Fields

You really only need two fields here: a **Transform object** and a **shared_ptr to a Mesh** (*not* a ComPtr as your Mesh class isn't itself a Direct3D object).

> *Should the Transform be a regular object, a pointer or a shared_ptr? You've got options! Ultimately, you'll have a get function that allows someone on the outside of the class to access and alter the transform, and we don't want that to create a copy. This means you'll need the get function to return a regular pointer, a shared_ptr or a reference.*
>
> *Regular pointers are easy: just return the address of your field. If you want to use a shared pointer instead, then your transform field should probably be stored as a shared_ptr initially. Or, you could return a reference instead, which skips the whole pointer business entirely.*

Constructor

The constructor should accept a Mesh shared_ptr and save it.

Destructor?

There isn't really much for a destructor to do here. As the Mesh object is not instantiated by this class, it should NOT be deleted by this class! In general, a class shouldn't delete an object it didn't create.

> *Case in point: say a Mesh pointer is shared between two Game Entities; if the destructor of each entity tried to delete that Mesh pointer, the first would succeed but the second would crash, as the Mesh has already been deleted.*

Getters

Create getters for both fields:

- `std::shared_ptr<Mesh> GetMesh()`
- **One** of the following:
  - `std::shared_ptr<Transform> GetTransform() // Shared pointer version`
  - `Transform* GetTransform() // Raw pointer version`
  - `Transform& GetTransform() // Reference version`

We need to ensure that, when accessing the transform from outside this class, we're changing the entity's *actual transform* and not a *copy* of it, so one of the above Transform getters is necessary.

Draw Method

You have a few options for how to structure drawing a single entity:

- Your entity class *could* just be a dumb container for the data to use when drawing, by way of the various "get" methods in the class. This setup requires a lot of repetitive drawing-related code in a big loop in Game::Draw() – setting constant buffers & filling them with entity-specific data.

- Your entity class could have its own Draw() method that takes care of setting all the buffers and issuing draw commands, which would require passing in a reference to a few D3D objects: the *context* and the *constant buffer* resource. This would simplify the code in Game::Draw().

A note on drawing: Having each entity draw itself is a very clean way of structuring your code. However, an advanced engine would put most drawing code inside a Renderer class that makes decisions about what to render and when, performing common tasks like object culling, 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 be useful in the final project.

Drawing

It is up to you exactly how you want to structure rendering an individual entity, though the second option outlined above will probably result in less code overall inside Game::Draw(). Regardless of *where* you put the code, you'll need to do *all of the following* to **render a single entity**:

- Set the correct Constant Buffer resource for the Vertex Shader stage (done in assignment 3)
- Collect data for the current entity in a C++ struct (done in assignment 3, but now updated to hold the world matrix from the entity you're about to draw)
- Map / memcpy / Unmap the Constant Buffer resource (done in assignment 3)
- Set the correct Vertex and Index Buffers (done in assignment 2)
- Tell D3D to render using the currently bound resources (done in assignment 2)

When it's time to render the next entity, you need to do *all of the above* again. For each entity!

Testing the Entities

Over in the Game class, create **at least five** different entities using the various meshes from the previous assignment. You should have several entities **share a single Mesh** object to ensure sharing Meshes works. You may find it useful to put all of your entities in an array or std::vector, which you can now easily loop through when drawing.

Seriously – I don't want to see a bunch of code copied & pasted five times in Game::Draw().

Ensure your entities are drawing to the screen. At this stage, they might all be "on top" of each other.

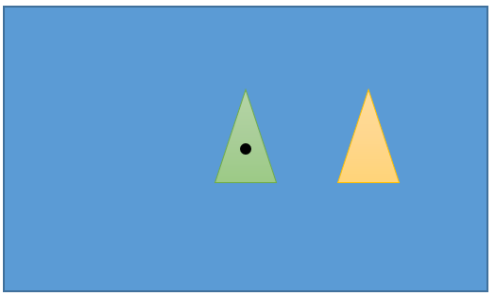
Movement

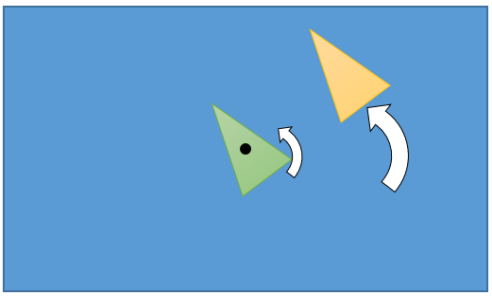
When moving objects around the screen, you do **not** update the vertices that live in a Mesh's D3D vertex buffer resource. That would be far too time-consuming, especially if you're using the same Mesh for multiple entities that are at different places on the screen. Instead, we can leverage the parallel 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, each vertex shader instance 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 *drawing*, get them ***moving***. This means 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, pulsate by scaling up & down or just simply fly off the screen (slowly, for testing). Be sure to scale all transformations by the delta time parameter of Update(). This will ensure your transformations aren't bound to the frame-rate, which can and will fluctuate as the program runs.

> *Note: The vertices of your meshes were probably hardcoded so the shapes were distributed all around your window. This was expected in previous assignments. However, it does mean that the vertices of those meshes aren't centered on the origin. This will affect how they look when a transformation is applied.*
>
> *In the examples below, there are two separate meshes: a green triangle with vertices centered on the origin (black dot) and a yellow triangle with vertices defined in the right half of the window. After applying the same rotation to both objects, the objects appear to rotate differently. In reality, they're both doing the same thing: rotating around their origin; the difference is in the positions of their vertices relative to that origin.*



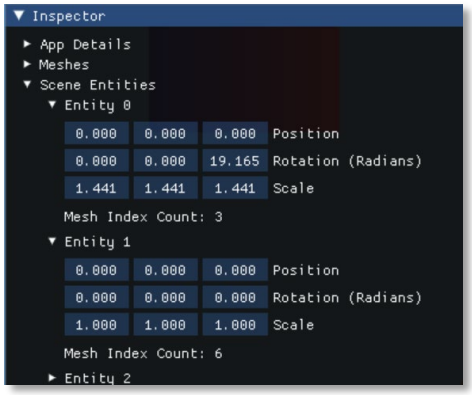


ImGui

List of all entities in your user interface, as well as their transform data: position, rotation and scale. See the screenshot to the right for an example of the data I'm looking for, though your overall layout may look different.

> *My example also displays the number of indices of each entity's mesh, which is not required.*

Use `ImGui::DragFloat3()`, or a similar function, to allow the user to both see and edit this data.



Odds & Ends

Keyboard Input

You don't have to have keyboard input for this assignment. However, if you'd like to begin playing around with input, feel free to use the included Input class in the starter code. For example: you could drive a single entity around the screen by altering its transform when various keys are pressed.

Don't go too crazy here. Future assignments will handle camera movement using WASD and the mouse, so you'll probably need to remove any input-related code you create now.

Transformation Hierarchies

You could absolutely use this Transform class to create a hierarchy (a tree of parent/child relationships) of objects. While it's outside the scope of this assignment, I'd be happy to give more information if you're interested. The math itself is quite simple, and creating a tree of Transform objects isn't hard either. The tricky part is ensuring that child transforms are updated (or marked "dirty") when necessary.

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.