Assignment 03

The Canvas assignment is **here (https://utah.instructure.com/courses/886852/assignments/12883237)** .

Requirements

- Make the main Graphics system completely platform-independent
  - All of the code must be in Graphics.cpp
  - Both Graphics.d3d.cpp and Graphics.gl.cpp must be removed from your solution and deleted
  - The Graphics.cpp file must specify the color to clear the back buffer to
    - In other words, rather than hard-coding the clear color to black you must create a platform-independent interface which accepts a color and use that
    - (It isn't necessary to let the interface specify the alpha value. You can choose to require it to be provided, or to allow it to be provided but default to 1, or not allow it to be provided and always set it to 1.)
  - To be absolutely clear, you should not add *any* platform-specific macros in Graphics.h or Graphics.cpp! Graphics.cpp must be completely platform-independent, and Graphics.h must be completely platform-independent except for the `sInitializationParameters` struct, and you should not change that struct from what was already given in Assignment 01.
- Update the mesh representation to use indexing (i.e add an index buffer)
  - The indices **must** be uint16_ts
- Change the effect initialization interface so that data can be specified (rather than being hard-coded)
  - The code that initializes an effect must be able to do so in a platform-independent way
  - The data that the initialization function requests doesn't necessarily have to be the data that the effect stores
    - Instead, the data that is requested to initialize an effect should be the easiest/simplest data required in order for the effect to create/initialize the data that it needs. If the implementation can do work to make the initialization call simpler or easier to write then it should!
    - (It should still be possible for the caller to initialize each piece of effect data! You just need to make it as easy as possible to do so.)
  - The memory that your representation of an effect takes up should be as small as possible. Don't store more data than you need!
- Change the mesh initialization interface so that data can be specified (rather than being hard-coded)
  - The code that initializes an effect must be able to do so in a platform-independent way
    - Specifically, this means that you must not require the caller to worry about winding order!
      - The caller should only have to specify data in a single platform-independent way, and your mesh implementation must take care of changing the data to be the correct winding order
  - The memory that your representation of a mesh takes up should be as small as possible. Don't store more data than you need!
- Create a second mesh in your Graphics.cpp file
  - Two distinct objects must be visible when your game is run
  - It's ok if the second one is just a simple triangle
  - It's ok to render more than two if you wish, but there must be at least two
- Create a second effect, and use a different effect for the two objects
  - It's ok to use the "standard" fragment shader that I provided in Assignment 01. In other words, it's ok to have one object with animating color using your fragment shader and one object just rendering white using the default fragment shader.
    - (The same standard vertex shader can be used for both effects)
  - It's ok to render more than two effects if you wish, but there must be at least two
- Your write-up should:
  - Show a screenshot of your game running
    - There must be at least two distinct objects with two distinct colors

- Explain how you made Graphics.cpp platform-independent:
  - What new interfaces did you have to create?
    - Where did you decide to declare these interfaces?
  - Where did you define the platform-specific implementations for these interfaces?
- Show us your code in the Graphics.cpp file that clears the back buffer color
  - (Don't show the actual interface definition or the implementation! Show the code in Graphics.cpp that *uses* your interface.)
- Show a screenshot of your game with a clear color (i.e. background) other than black
- Show code from your Graphics.cpp file that initializes an effect
  - Explain what data the user is required to specify
- Tell us how much memory a single effect takes up in both platforms (use sizeof() with a debugger rather than guessing). Is there any way to make it smaller?
  - Show us the data from your struct or class and explain why it couldn't be smaller
  - If the representation is a different size for different platforms explain why
- Show code from your Graphics.cpp file that initializes a mesh
  - Explain what data the user is required to specify
- Tell us how much memory a single mesh takes up in both platforms (use sizeof() with a debugger rather than guessing). Is there any way to make it smaller?
  - Show us the data from your struct or class and explain why it couldn't be smaller
  - If the representation is a different size for different platforms explain why

Submission Checklist

- Your write-up should follow the **standard guidelines for submitting assignments (https://utah.instructure.com/courses/886852/pages/submitting-assignments)** and the **standard guidelines for every write-up (https://utah.instructure.com/courses/886852/pages/write-up-guidelines)**
- If you change the clear color in Graphics.cpp the background should also change:
  - {1, 0, 0, 1} should make a red background
  - {0, 1, 0, 1} should make a green background
  - {0, 0, 1, 1} should make a blue background

Finished Assignments

Details

Platform-Independent Graphics.cpp

- I would advise getting this working first (before trying the other requirements for this assignment)
- If you compare your existing Graphics.d3d.cpp and Graphics.gl.cpp files from Assignment 02 they should be almost the same, with only a few differences. You need to figure out a way to get rid of those differences.
- Some of the differences will be platform-specific data and some of the differences will be platform-specific function calls. You need to figure out a good place to put both.
- There is not a "correct" way to solve this problem, and the details in the assignment are intentionally vague in an attempt to not influence you too much. This is a challenge for you to figure out your own style and solution.
  - (If you can't figure out how to get it to work, however, don't hesitate to post questions to the discussion board asking for help! I will try to give ideas and hints without giving away how *I* would do it.)

Index Buffers

- In Direct3D:
  - To create:
    - An index buffer is an `ID3D11Buffer*`, just like a vertex buffer
    - The way you create it is almost identical to a vertex buffer: You create a buffer description struct and call the same CreateBuffer() function. The buffer description fields are the same as what you use for a vertex buffer except:
      - ByteWidth should be the size of the index buffer (how do you calculate this?)
      - Use `D3D11_BIND_INDEX_BUFFER` for the BindFlags
    - The initial data struct's pSysMem field should point to the index data instead of the vertex data
      - (Remember that each index should be a uint16_t)
  - To draw:
    - Before drawing the mesh you need to bind the index buffer (it doesn't matter whether you bind the vertex buffer or index buffer first). This is what I have:
      - ```
        EAE6320_ASSERT( indexBuffer );
        constexpr DXGI_FORMAT indexFormat = DXGI_FORMAT_R16_UINT;
        // The indices start at the beginning of the buffer
        constexpr unsigned int offset = 0;
        direct3dImmediateContext->IASetIndexBuffer( indexBuffer, indexFormat, offset );
        ```
    - The draw call is different when using indexing. This is what I have:
      - ```
        // It's possible to start rendering primitives in the middle of the stream
        constexpr unsigned int indexOfFirstIndexToUse = 0;
        constexpr unsigned int offsetToAddToEachIndex = 0;
        direct3dImmediateContext->DrawIndexed( static_cast<unsigned int>( indexCountToRender ), indexOfFirstIndexToUse, offsetToAddToEachIndex );
        ```
  - To clean up:
    - You should be able to figure out how to do this
- In OpenGL:
  - To create:
    - An index buffer's ID is a `GLuint`, just like a vertex buffer
    - The way you create it is almost identical to a vertex buffer: It must be created when the vertex array object is bound (I would recommend putting the index buffer creation code immediately after the vertex buffer creation code). You generate a new buffer, bind it, and allocate/assign initial data. You can copy/paste the code to create the vertex buffer and then make the following changes:
      - When you call glGenBuffers() to create the buffer use the index buffer ID instead of the vertex buffer ID
      - When you call glBindBuffer() pass in the index buffer ID and use `GL_ELEMENT_ARRAY_BUFFER`
      - When you call glBufferData() use `GL_ELEMENT_ARRAY_BUFFER` and pass in the size of the index buffer (how do you calculate this?) and the index data
        - (Remember that each index should be a uint16_t)
  - To draw:
    - You don't have to change anything to bind the index buffer (the vertex array object takes care of everything)
    - The draw call is different when using indexing. This is what I have:
      - ```
        // The mode defines how to interpret multiple vertices as a single "primitive";
        // a triangle list is defined
        // (meaning that every primitive is a triangle and will be defined by three vertices)
        constexpr GLenum mode = GL_TRIANGLES;
        // It's possible to start rendering primitives in the middle of the stream
        const GLvoid* const offset = 0;
        glDrawElements( mode, static_cast<GLsizei>( indexCountToRender ), GL_UNSIGNED_SHORT, offset );
        EAE6320_ASSERT( glGetError() == GL_NO_ERROR );
        ```
  - To clean up:
    - You should be able to figure out how to do this
        ```

Initialization with Data

Remember that a constructor in C++ should never fail (unless you use exceptions, but we don't in this class because most games don't). If there is any data initialization that can fail then you *must* have it passed in in a separate initialization function.

Effects

There shouldn't be many changes required to make this work, and all of the changes should be platform-independent. If you find yourself having to make platform-specific changes then your design probably isn't abstracted as well as it could be.

Meshes

- You will need to think carefully about what data needs to be stored. There is at least one piece of information that you probably didn't store in Assignment 02 that you will need to store for this assignment. There is also some data that you might be tempted to store that you don't have to. Try to make sure that you understand 1) what data you need in order to initialize and 2) what data you need to render and clean up. If you need data in order to initialize the mesh but you don't need it to render the mesh then you don't need to store it.
- Make sure to change all of the places you have hard-coded so that they now are flexible enough to do the right thing regardless of what data the mesh was initialized with.
- How can you make a platform-independent interface without worrying about winding order?
    - You should pick a winding order convention
        - It doesn't matter whether you choose right or left handed. Just pick something for your engine and then stick to it.
        - Don't try to create a clever way to figure out whether the input is right or left handed, and don't try to accept either and store a variable telling which it is; instead just pick one and require it! Being flexible with winding order would be good if we were writing something like Maya that can require lots of different kinds of input from lots of different users, but in an engine it just wastes processing time. If this bothers you remember that in an upcoming assignment we will switch to reading files that contains mesh data that has been created by Maya that our tools have built. We don't have to worry too much about the programmer "user", because eventually the data will be authored in a very user-friendly way and we can then build it any way that we want to.
        - It is also, incidentally, a waste of processing time to switch the handedness of the triangles that are input like you will have to do in this assignment. Don't worry! In an upcoming assignment when we have the ability to build files we will move this code from run-time to build-time so that your engine can just load the meshes as quickly as possible by doing as little processing as possible.
        - It is common in games programming to have conventions like this that can't be programmatically enforced. Can you think of anything that you could do in your mesh code to make it easier to figure out what the convention is?
    - Remember that the winding order applies to the index buffer, and not the vertex buffer. You should be able to use the same vertex data for both platforms.
    - How can you change the indices to convert from right-to-left-handed or left-to-right-handed? If you get confused trying to figure this out it can really help to draw a picture and write out the indices for both ways to see how they change. If you get stuck ask for help on the discussion board!

Rendering

Remember that an effect needs to be "bound", and then any mesh that is "drawn" will use the currently bound effect. That means that if you want to draw two different objects using two different effects you would have to do something like this:

- Bind first effect
- Draw first mesh
- Bind second effect
- Draw second mesh

Optional Challenges

- Can you figure out how to make the background color animate?
    - You know enough that you should be able to figure this out, but it's not necessarily obvious. You will have to look at some code that we haven't talked about in class or assignments.

- You should *not* have to change any shaders to do this! You should change this in C++ code.
- You should implement this in Graphics.cpp. (In a future assignment you will learn how to do this from game code instead of engine code.)
- You can use either "simulation time" or "system time". You can read the code comments to learn more about the difference. If you don't have a preference I would recommend simulation time.