Assignment 08

The Canvas assignment is **here (https://utah.instructure.com/courses/512907/assignments/5250590)** .

Requirements

- Change your MeshBuilder to output a binary mesh
  - You will have to move the code that reads the human-readable mesh file from run-time to MeshBuilder
  - There are two functions you may use to output binary data:
    - The C way is **fwrite() (http://www.cplusplus.com/reference/cstdio/fwrite/)** . The link shows an example of opening a file, writing to it, and closing it.
    - The C++ way is **std::ofstream::write() (http://www.cplusplus.com/reference/ostream/ostream/write/)** . The link shows an example of opening a file, writing to it and closing it.
  - There are four pieces of binary data that you must output:
    - The number of vertices
    - An array of vertex data
    - The number of indices
    - An array of index data
  - There should be exactly four calls to fwrite()/std::ofstream::write() to write out each piece of binary data
    - You should not have to iterate through each vertex or each index to write them to your binary file! You should be able to write out the entire array as a single block with a single function call.
    - (It is possible that you may have more than four calls to fwrite()/std::ofstream::write() if you complete the optional challenges, but even in that case you should still only have four calls for each of the four pieces of required data.)
- Change your run-time mesh representation to load a binary mesh
  - Your platform-independent mesh code will have to extract the four pieces of data that you wrote to your binary file
    - Your platform-specific mesh code should not have to change except to get rid of any code that changes winding order
  - You should use Platform::LoadBinaryFile() to load the entire file as a chunk of memory
  - There should be exactly four calls to extract each piece of binary data
    - You must not iterate through vertices or indices!
      - Every year I say this and every year some students still do it. **NO ITERATION! NO LOOPS!** You should not have *any* for or while loops anywhere in your mesh loading/initialization code!
  - Once the mesh is initialized you should free the Platform::sDataFromFile
- Your write-up should:

- Show us a screenshot of your game running (if you've done the optional challenge to find a recognizable mesh make sure to show it!)
- Show an example of a binary mesh file built by your MeshBuilder (a screenshot of the file in a hex editor is fine) and:
  - Tell us the order of the four things in the binary file and why you chose that order
    - Help us to be able to recognize the data by highlighting/outlining/underlining the four things in different colors
  - Tell us at least two advantages of using binary file formats (I am looking for two specific reasons, but you can include more if you wish)
    - Compare the advantages of binary formats with the advantages of human-readable formats. Why do we use binary formats at run-time in our class but human-readable formats to store the data?
- Tell us whether the built binary mesh files should be the same or different for the different platforms, and explain why. (In other words, should the binary file for a specific mesh that your MeshBuilder outputs for Direct3D be the same as the one that it outputs for that same mesh in OpenGL? If so, explain why. If not, explain why there are differences and what they are.)
- Show us how you extract the four pieces of data from binary data at run-time (show the actual code you use to do this)
  - Unless you chose to do the optional enhancements listed in the details section your code to do this should be fairly simple. If it isn't you are probably doing something wrong.
  - If you do error checking (e.g. to make sure that the data from the file is large enough) it is fine to show it, but you can also remove it for your write-up so that you just show the important parts
- Create a mesh in Maya with many vertices and indices to measure the advantages of binary files
  - A helix is a good choice for this because there are many parameters that can greatly increase the vertex/index count. Make it have as many as you can that will still fit within the limits of uint16_ts and 16-bit indices. You don't need to commit this to git; just make it temporarily to do measurements for your write-up and then delete it.
    - If you complete the optional challenge to support 32-bit indices you can also compare a mesh with 32-bit indices, but you should still do the comparison for a mesh with 16-bit indices
  - Tell us how big (in bytes) the human-readable version is on disk and how big (in bytes) the binary version is on disk
  - Tell us how long it takes to load the human-readable version and how long it takes to load the binary version
    - You can use functions from the Time project for this. Make this temporary code and revert it after you have made measurements!
    - To time the human-readable version put code in MeshBuilder. Make sure to time not just how long it takes to load the Lua file but how long it takes to extract the data. (You should start timing right before loading and end timing right before writing out the binary file.) You can put a temporary print statement with the total time that will show up in Visual Studio's Output window.

- To time the binary version put code in your run-time mesh loading code. Make sure to time not just how long it takes to load the binary file but how long it takes to extract the data. You can put in a temporary logging statement with the total time that will show up in your log file.
- Make sure to do this timing on a release build! (It doesn't matter which platform you time, though.)

Submission Checklist

- Your write-up should follow the **standard guidelines for submitting assignments (https://utah.instructure.com/courses/512907/pages/submitting-assignments)** and the **standard guidelines for every write-up (https://utah.instructure.com/courses/512907/pages/write-up-guidelines)**
- If you add a bug to a mesh (like typing "sdf") and then build assets 1) the build should fail, 2) an error message should be displayed in Visual Studio's Error List window, and 3) double-clicking that error should open the file

Finished Assignments

- **Yitong Dai** **(https://yzd0014.wixsite.com/dyt1205/blog/eae6320-binary-mesh-file)**
- **Yuxian Deng** **(http://yuxiandeng.info/Program/GameEngine2/Assignment8/Assignment8.html)**

Details

Loading the Human-Readable Mesh File in MeshBuilder

- Most of the code to read the Lua mesh file that you used at run-time should be the same when you move it to MeshBuilder, but one big required change is how errors are handled:
  - Instead of logging errors you will have to call eae6320::AssetBuild::OutputErrorMessageWithFileInfo() so that errors will show up in Visual Studio's Error List window
  - You should mostly remove asserts. You may decide to leave a few as sanity checks (for things that "should be impossible"), but an assert firing at build-time when no debugger is attached is usually not very useful. Instead, you should use traditional error checking. Remember that we don't care so much about bleeding-edge performance at build time, and so it is fine to have lots of error checking that will end up being reported as asset build errors.
- When organizing your MeshBuilder code keep in mind the four pieces of data that you will need to write out as binary data
  - In previous assignments you had to extract these same four pieces of data from the Lua file at run-time so you could create the Direct3D and OpenGL vertex buffer and index buffer. In this assignment you will be extracting them so that you can create a binary file.
  - This part of your MeshBuilder that deals with Lua, then, should be self-contained. You will input the path to the human-readable Lua file, and the output should be the four pieces of data. Once you have those four pieces of data you can output them as a binary file without worrying about

where they came from. (This would allow you, for example, to change to a different format in the future if you decide you don't like Lua after this class is over.)

Writing Binary Data

- There are several possible correct orders that you could choose to write out the four pieces of data, but also some that are incorrect
  - When deciding how to write data to a file keep in mind what you will have to do at run-time to read it in. Some of the data is independent (meaning that it could be read in any order), but some of the data must be read first before other data can be read. As long as you get these dependencies correct you can choose any order that you wish (although some orders may be more efficient than others).
- When dealing with binary data the size of numbers is critical. Programmers are sometimes careless with the size of numbers and allow the compiler to implicitly convert between them, but this can't be done with binary data. Always use explicit sizes (e.g. uint8_t, uint16_t, uint32_t, or uint64_t instead of unsigned int), and if you need to read a uint16_t make sure that you have written a uint16_t.
- Once your MeshBuilder is generating binary files you should check the results in a hex editor to make sure that the file looks like what you expect
  - I like the way that Visual Studio shows binary data, but recent versions make it really annoying to actually get a binary file open (it tries to be smart about what the file is rather than just showing it as binary data) and so I rarely use it anymore. My current preference is **HxD (http://mh-nexus.de/en/downloads.php?product=HxD)** .
  - You should know the order of the four things and how many bytes they each take up. Verify that the file has the right data in the right places. (It might be hard to recognize the float data, but you should definitely be able to find the number of vertices and indices/triangles, and the array of indices itself should also be fairly readable if you know what you're looking for.)

Extracting Binary Data

- You will have to locate the address of each of the four things starting from the base address of the chunk of binary data that was read from the file. The offsets of each of the four things from the base address will depend on 1) the order that you wrote the data out and 2) how many vertices and indices there are in the arrays.
- You should use a uintptr_t to keep track of the offsets within a binary chunk of data. A uintptr_t is an unsigned integer which any valid pointer can be converted to, and that resulting uintptr_t can be converted back to the identical original pointer.
  - To calculate the the offset to the beginning and the ending of the binary chunk of data:

```
auto currentOffset = reinterpret_cast<uintptr_t>( dataFromFile.data );
const auto finalOffset = currentOffset + dataFromFile.size;
```

- The first thing that you decided to write out will be located at the base address, so this is easy. As an example let's assume that the first thing in your binary mesh file is the number of vertices as a uint16_t. You could then extract that information like this:
  - ```
    const auto vertexCount = *reinterpret_cast<uint16_t*>( currentOffset );
    ```
- If the vertex count is a uint16_t then you know that the *next* piece of information (whatever it is) will be located 2 bytes after the base address. (You must understand the previous sentence in order to complete this assignment. If it's confusing to you then re-read it as many times as necessary until you understand what it's saying and why.)
  - If the next piece of data is the number of indices stored as a uint16_t then you could write:
    - ```
      currentOffset += sizeof( vertexCount );
      const auto indexCount = *reinterpret_cast<uint16_t*>( currentOffset );
      ```
  - If, on the other hand, the next piece of data was the array of vertices then you could write:
    - ```
      currentOffset += sizeof( vertexCount );
      const auto* const vertexArray = reinterpret_cast<VertexFormats::sMesh*>( currentOffset );
      ```
- The vertex count and index count will have fixed sizes that will be the same for every mesh, but the size of the actual vertex and index arrays will be different. You can calculate how many bytes an array takes up in your block of data by taking into account how many bytes a single element takes up as well as how many elements there are in the array.

Optional Challenges

- You may want to change the mesh file extension for binary files
  - Your human-readable mesh files are different than your binary mesh files, and it can be useful for humans to use different extensions (so that when you see a file name you can tell immediately whether it is a source file or a built file)
  - You can make your build system do this by changing the ConvertSourceRelativePathToBuiltRelativePath() function in AssetBuildFunctions.lua. Can you figure out how to do this?
  - (You may also want to do the same thing for shaders)
- Can you figure out how to align the four pieces of binary data optimally?
  - You can write out binary data any way that you want, but certain data will work better with certain alignments when you read it in. In this current assignment it doesn't really matter because we free the data after we have initialized the mesh (because the API keeps its own copy), but you could still align things optimally as a fun challenge to figure out how. In order to do this you 1) may need to add "padding" to your file, and 2) may want to be a bit more careful about the order of data, taking size and alignment into account (because this may prevent the need for padding in some cases)
  - You should use alignof() to get the ideal alignment of the different kinds of data. (For this specific assignment the vertex struct is the main thing you will worry about, because its ideal alignment
      ```

could change if you make changes in future assignments. Still, it is good practice to use alignof() for everything instead of hard-coding it. If you complete the optional challenge to use 32-bit indices remember that the ideal alignment for indices will be different for different meshes.)
    - The Platform::LoadBinaryFile() function uses malloc(), and that aligns its allocations in a way that should be good for all kinds of data (in practice this means it will be aligned to an address which is a multiple of 8). This means that you can consider the start of the file to be aligned, and you only need to worry about alignment of data of different sizes within the file (just like the way data of different sizes within a struct or class can get padding).
    - If you add padding when writing you'll need to figure out how to deal with it when reading the file in. It's possible to make it implicit (you calculate it at run-time the exact same way that you did at build-time), or you can write it out explicitly in the file. If you write it in the file think about the smallest size you can make it (how big could the padding possibly be?) and take that into account when figuring out the ideal alignment for everything else.
    - The actual contents of any padding doesn't matter, but you should always try to make your generated files deterministic (in other words, the same file will always be generated). A good choice is to use zero for any padding, but you can also use some recognizable bit pattern.
    - Remember that you should only add padding if a piece of data wouldn't be aligned. If the alignment is already ideal then should should just write out the data with no extraneous padding.
- Can you figure out how to support 32-bit vertex indices?
    - We have used 16-bit indices to keep our mesh sizes as small as possible. This is fine for most meshes, but you may encounter cases where a mesh has more vertices than can be indexed with only 16 bits and these cases would be unusable in our class with our current code.
    - If you want to support 32-bit indices then you need to update your code to be able to use either 16-bit or 32-bit indices. (In other words, do *not* just change everything to use 32-bit indices. Instead, still use 16-bit indices whenever possible, but allow 32-bit indices to be used if the mesh is too big.) At build-time you would need to calculate whether 16 bits are sufficient (how do you do this?) and then generate an array of either uint16_ts or uint32_ts. You could then either 1) write out something in the binary file that makes it clear which kind of indices are being used and then extract that data at run-time, or 2) require the run-time to figure out which kind of indices are used the same way that it was calculated at build time.
        - One of the trickiest aspects for students to figure out is how to represent an array of indices that can be different types. As a hint, think about using a void*.
    - This is a fun change to make but it may be more difficult than you expect. If you try this *please* make sure that you first commit your working changes that fulfill the assignment's requirements (just using 16-bit indices). Then, if something goes wrong you can just give up and revert everything to get back to a working finished assignment.
- Can you find (or create) a mesh that is a recognizable object (i.e. that isn't just a Maya polygon primitive)?
    - The easiest way to do this is to find something online, but if you have objects that you've made or that you have permission to use from your project you can use that too
        - **Here (http://www.turbosquid.com/Search/3D-Models?keyword=&search_type=free&media_typeid=2&file_type=105)** is one possible site that you could

use

- This is an example from my reference implementation using a crocodile mesh that I found online (I added vertex colors for the eyes):
  -