Assignment 06

The Canvas assignment is **here (https://utah.instructure.com/courses/886852/assignments/12883241)** .

Update: If you would like a project that I have set up with Niels Lohmann's JSON library (that you used last semester in Joe Barnes's class) it is here (https://utah.instructure.com/courses/886852/files/150334757/download?wrap=1) (https://utah.instructure.com/courses/886852/files/150334757/download?download_frd=1) .

Requirements

- Create a human-readable Lua-based mesh file format
  - **Update: You may also use JSON instead of Lua if you'd prefer**
  - Refer to readNestedTableValues.lua in the **Lua examples (https://utah.instructure.com/courses/886852/pages/lua-examples)** for a general pattern of what Lua-based asset files must look like in our class
  - There isn't a single correct way to do make a good human-readable Lua-based geometry file format, and there are several different legitimate choices that you can make. There are also ways, however, that I would consider to be not as well-designed! If your format isn't easily human-readable or human-understandable then you will lose points.
    - **Update: The same design considerations for Lua apply to JSON (e.g. when to use arrays and when to use dictionaries). You will be graded based on how human-readable the file format is, regardless of whether you use Lua or JSON.**
  - Choose a file extension that you like for human-readable geometry files
- Create mesh files for your existing meshes
- Create a MeshBuilder project
- Update AssetBuildFunctions.lua so that mesh files can be built
  - (I.e. make sure that MeshBuilder.exe will be run to build your mesh files)
  - You may choose a different file extension for built mesh files from what you use for source mesh files (anticipating a future assignment when they will be in a binary format), but this is up to you
- Update AssetsToBuild.lua so that your specific mesh files that you created are built when your BuildMyGameAssets project is built
- Update your mesh representation to load the geometry data from a file
  - Change your mesh interface's factory function so that instead of taking vertex and index data as input and returning a mesh pointer as output it takes a mesh file path as input and returns a mesh pointer as output
- Update your game to to load mesh geometry from disk instead of hard-coding the data
- Your write-up should:
  - Discuss the advantages of having human-readable asset files
  - Show one of your mesh files
    - (You can either copy/paste the text, which should be human readable, or show a screenshot from a text editor. If you include a screenshot make *sure* that the text is easily-readable and not too small.)
    - Describe why you made the design choices that you did. In particular, help your readers to understand what you did to make the file readable and understandable for a human.
      - (If there was any part of the file where you struggled to decide which way would be the easiest to read and understand tell us about the different potential choices and explain why you ended up choosing what you did)
  - Show a screenshot from your game
    - (No visual changes from last week are required, but it is nice for your readers to have something to look at. You may consider changing some geometry using your new mesh files.)
  - Show a screenshot of you debugging your MeshBuilder program
    - This should be a screenshot from Visual Studio with the little yellow arrow visible to show that your MeshBuilder code is being debugged

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 position of a vertex in a mesh file, build your solution, and run the game then the change should show up (without making any code changes)
- If you remove three triangle indices in a mesh file, build your solution, and run the game then the mesh should be missing a triangle (i.e. the number of triangles should be calculated automatically and correctly, so that adding or removing a triangle is as easy as adding or removing three indices from the mesh file)

Finished Assignments

Details

Mesh File Format

- The way to create this was discussed in the class lecture
- Remember what the design goals of a human-readable file format are:
  - The file should be easy to read
  - The file should be easy to understand
  - The file should be easy to maintain
  - The file should be easy to debug (when something doesn't look correct in your game and you want to find out whether it might be a content problem)
- Remember what the design goals of a human-readable file format are *not*:
  - The file shouldn't necessarily be efficient to parse/process
    - When we create a binary file format we will make it as efficient as possible. For our human-readable files, however, if a design decision makes the file better for a human to read/understand then it is ok if it also makes it slightly less efficient for a machine.
  - The file shouldn't necessarily be as easy as possible to create
    - It isn't bad if the file format is easy to write and change for a human, and you should strive to make it easy! However, if you have to make a design decision between making it easy to write vs. easy to read/understand/debug then you should prioritize ease of reading/understanding/debugging.

        - (Remember that, ideally, the file will be created and edited by a content-creation tool; the tool's focus will be making it as easy to write and change as possible using a GUI, and the file itself will only rarely be hand-edited by a human.)
- The important things to ask yourself for each piece of data is:
    - Is the data ordered? If so, it should be an array.
        - Sometimes the order is important (like triangle winding order). Sometimes the order is arbitrary (like vertex order). In both cases, though, there must be *some* order. When this is true an array should be used.
    - Is the data unordered? If so, then it should be a dictionary (with a reasonably human-readable name)
        - Sometimes students get confused about this and think that it is ok (or desirable) to impose an implicit order on data. This is not the case! You should not choose an implicit convention for things where you can make it explicit by using a dictionary instead of an array.
            - There may be cases where an ordering "makes sense" to you, and if so it is fine when you create files to always use this ordering, but you should still use a dictionary instead of an array.
            - Instead of asking yourself "is this data ordered?" you may find it useful to ask the opposite question "would it be ok to change the order of this data?" If it would be possible to change the order without having to change the code that reads the file and with no resulting visual differences then you should use a dictionary.
- The only case that we have encountered in our class so far where someone reasonably familiar with graphics would immediately understand an implicit order is with the x, y, and z of position. For those either a dictionary (using the letters) or an array (using an implicit ordering) would both be good and acceptable choices (you should choose based on what you think is the most readable and understandable).
- Currently the only data that a vertex has is position, and if you haven't done any graphics programming before it is not obvious what else there could be. Imagine that every vertex could have any kind of arbitrary data. In our class specifically in a future assignment we will be adding a color (so that every vertex specifies a (possibly unique) vertex and a (possibly unique) color. It is also very common for a vertex to have a "normal" vector. As you design your file format try to imagine what you will have to do to add a color to each vertex in Assignment 07; if it won't be easy to do that then your design may not be ideal.
- You should avoid redundant data
    - The number of vertices and the number of indices are both implicitly defined by their tables. You should *not* design your file format to explicitly require these counts to be defined. (Can you think of why redundant data is undesirable?)
    - Often students think about how to make the file more readable and debuggable and think that things like the total number of vertices or the ID of each vertex should be listed. This is good thinking, but the solution is not to require it as part of the format! Can you think of how to show information like this in the file without requiring it? (As a hint, remember that your file will be using Lua. What mechanism does Lua provide to give information to the human reader that is ignored by the machine interpreter?) Remember that these files will ideally be created by a program (your mesh files, for example, will be created by your Maya mesh exporter starting in Assignment 07), and so it isn't hard to make your Maya mesh exporter add any extra optional/helpful information like this automatically, even though it wouldn't be required if a human made the file by hand.

MeshBuilder

- Create a new project
    - This *must* be in the Tools/ folder (both on disk and in Solution Explorer)
    - Choose **Console Application (.exe)** for the Application Type and make sure that Precompiled Header is *not* selected in Additional Options
    - You should know how to add the appropriate property sheets
        - It's not uncommon for students to skip this step, thinking that it's not necessary, but this can cause confusing bugs, either in this assignment or the next one. Please do this.
- Use ShaderBuilder as an example, but keep in mind that your MeshBuilder will be *much* simpler:
    - You will need a main() function that calls the appropriately-templated Build() function. My advice is to just copy EntryPoint.cpp from ShaderBuilder and change every instance of "Shader" to "Mesh".
    - You will need a cMeshBuilder.h file (named whatever you want). You can start with cShaderBuilder.h if you want, but your cMeshBuilder.h (or whatever it's called) file only needs to implement the inherited virtual Build() function (you can delete the second shader-specific one and remove the unnecessary #include directives).
    - You will need a cMeshBuilder.cpp file (named whatever you want)
        - Its job is to override iBuilder::Build()
        - (You can look at how ShaderBuilder does this as a starting point, but MeshBuilder is much simpler and won't really have anything in common with ShaderBuilder's Build() function. It probably makes sense to just create the CPP file from scratch.)
        - In a future assignment we will build a binary mesh file. Until then, however, a mesh doesn't actually get "built". Instead, MeshBuilder's responsibility is to *copy* the source mesh file to the target mesh location.
            - MeshBuilder::Builder() must copy `m_path_source` to `m_path_target` (these are member variables that the base iBuilder class takes care of and so you your Build() function can just use them without worrying about how they get initialized)
            - Can you figure out how to copy a file in a platform-independent way? (I have given you a function that can do this in the code from Assignment 01. Look in the Platform project to see if you can find it.)
                - The copy should not fail if the target file already exists (it should try to overwrite the existing file)
                - The copy should update the file time of the target file
        - If an error occurs during build time it is handled differently from the way we handle errors of the game during run time
            - The ideal way for errors to be output may differ for different platforms, but in our class where we build things using Visual Studio we want any errors or warnings to show up in Visual Studio's Error List.
            - Any error should be reported using one of the functions that can be seen in AssetBuildLibrary/Functions.h. You should read over them to understand what the different versions do, but as an example this is what my reference implementation does if copying the source geometry file to the target destination fails:

                -
                  ```
                  OutputErrorMessageWithFileInfo( m_path_source, errorMessage.c_str() );
                  ```
            - (You must also return a `cResult` that indicates failure. Once you figure out how to copy a file in platform-independent way it should be obvious where you get the cResult and the error message if the copy fails.)
- There are some references that you will need to add to your new MeshBuilder project. You should be able to figure out what they are.
- There is a different project that will depend on your new MeshBuilder project (an explicit dependency, i.e. not using a reference). Do you remember how to figure this out? (Ask yourself "which project needs the MeshBuilder project to be built before it can be built?" Remember! The question isn't which project needs MeshBuilder to be built before it can be *run*; the question is which project needs MeshBuilder to be built before it can be *built*.)

AssetBuildFunctions.lua

- This is the file that defines most of the asset-building logic for our engine. It is a Lua script, and has quite a few comments that you can read if you are interested in how it works. You will have to add some new logic so that the build system knows how to deal with mesh files.
- Search in the file for "NewAssetTypeInfo"

    - The first instance in the file is a definition for a function that should be called for every asset type that can be built. You will need to call this function to define how mesh assets should be built. Make a note of what parameters the function expects.
    - Keep searching in the file for where the NewAssetTypeInfo() function gets called. You should find it being called for shaders. You can copy/paste this function call and then modify it to work for meshes.
- Make a call to NewAssetTypeInfo() that defines how mesh files should be built:
    - The first parameter is a string that defines a unique key for an asset type. If you want to follow my pattern you would use "meshes", but this can be anything that makes sense to you.
    - The next parameter is a table. This is an idiomatic use of Lua tables, and is a way to specify named parameters. There are three functions that you can specify by their names (look at how this is done for shaders to help you understand the syntax):
        - GetBuilderRelativePath()
            - This function is required, and tells the asset build process what builder executable to run in order to build the specific asset. You should be able to figure out what to return in this function.
        - ConvertSourceRelativePathToBuiltRelativePath()
            - This function is optional, and tells the build system what the target built path should be given the source content path. If you don't specify this the built file will have the same file name and relative path as the source path. Usually my recommendation would be to not change anything except possibly the file extension. For this assignment it is completely fine to not define this function. If you want to try changing the file extension look at how the function is defined for shaders and see if you can figure out what you need to do.
        - ShouldTargetBeBuilt()
            - This function is optional, and calculates extra dependencies for a specific kind of asset. I added this specifically for shaders so that the "shaders.inc" file can be used as a dependency. You should not define this function for meshes.

AssetsToBuild.lua

- Specifying which mesh files should be built is very similar to specifying which shaders should be built
- You will need to create a new named table for meshes. The key should be whatever key you specified when calling NewAssetTypeInfo() in AssetBuildFunctions.lua.
- There are two ways to specify a mesh file to be built:
    - You can do it the same way that shaders are specified, with a table for each geometry. If you do this each geometry only need a "path" (you don't need to specify any "arguments")
    - Alternatively, you can just specify the path as a string, instead of having to make a table. In other words, your mesh table can just be a list of source paths.
        - (The reason this works is because I have programmed AssetBuildFunctions.lua to look for both. Most assets only need a path, and so as a convenience I check whether a table entry is a string, and, if so, I treat it as a path. Otherwise, I check whether a table entry is another table, and then look for named entries.)

Loading Meshes

- There are four pieces of data that you need to extract from the Lua file in order to create platform-specific vertex and index buffers:
    - An array of vertex data
    - The number of vertices in the array
    - An array of indices
    - The number of indices in the array
- Your code from Assignment 05 should expect the game to pass in these four pieces of information. The change for this assignment will be to have the game pass in a path to a mesh file to your factory function, that function will extract the four pieces of data from the file, and then it will pass on those four pieces of data to your platform-specific initialization code.
- I would strongly encourage you to review **ReadTopLevelTableValues.cpp** and **ReadNestedTableValues.cpp** from the **Lua examples (https://utah.instructure.com/courses/886852/pages/lua-examples)**
    - Seriously! At least read through both files quickly (including the comments) before starting this part of the assignment. You will be glad that you did :-)
- **Update:** If you want to use JSON instead of Lua I have created a **ReadNestedTableValues.json (https://utah.instructure.com/courses/886852/files/150334756/download?wrap=1)** **(https://utah.instructure.com/courses/886852/files/150334756/download?download_frd=1)** which shows the same thing as the ReadNestedTableValues.lua example
    - I have never used the library before so I can't give you much advice. The following is a small code sample (without sufficient error checking or reporting (note that the library throws exceptions!)) just to show you how it could be used with our engine code, and hopefully your experience from Joe Barnes's class will help you to figure out the rest:
    -
      ```
      eae6320::Platform::sDataFromFile dataFromFile;
      eae6320::Platform::LoadBinaryFile( "data/readNestedTableValues.json", dataFromFile );
      const auto parsedFile = nlohmann::json::parse( static_cast<const char*>( dataFromFile.data ),
              static_cast<const char*>( dataFromFile.data ) + dataFromFile.size );
      if ( parsedFile.is_object() )
      {
              const auto textures = parsedFile["textures"];
              if ( textures.is_array() )
              {
                      for ( const auto& texture : textures )
                      {
                              if ( texture.is_string() )
                              {
                                      const auto textureValue = texture.get<std::string>();
                              }
                      }
              }
              const auto parameters = parsedFile["parameters"];
              if ( parameters.is_object() )
              {
                      const auto brightness = parameters["g_brightness"];
                      if ( brightness.is_number() )
                      {
                              const auto brightnessValue = brightness.get<float>();
                      }
                      const auto speed = parameters["g_speed"];
                      if ( speed.is_number() )
                      {
                              const auto speedValue = speed.get<float>();
                      }
              }
      }
      ```
    - You should be able to figure out how to add the necessary references. Note that I have made an Includes.h file that you can #include.
    - There is a slight problem with the license file dependencies that I will have to fix in future years when I am better prepared from the start of the semester. It may not end up in the install location the first time you build in this assignment, but when we move the human-readable parsing code to MeshBuilder in a future assignment I believe it will work correctly.

- The hardest part of this assignment will be dealing with the Lua stack. Notice in ReadNestedTableValues.cpp how I have a different function for every nested table, and how I write comments to help me keep track of where I am (and what values are at the top of the stack). You obviously do not need to use my same coding style, but it can *really* help to do something similar: By keeping a single function for each level I only need to worry about one thing at a time. Consider doing something similar, especially when you are starting out. Write your code one step at a time, and write checks (like `lua_istable()`), to try and verify that you are getting the data that you expect.
- Something that students often get confused about is when to iterate through a table and when to look up keys explicitly:
  - Remember that *you* designed the file format, and so if you know what to expect in a table you don't have to iterate
  - If a table is an array, and that array can have a variable number of values, then you will need to iterate
    - For example: Every mesh can have a different number of vertices, and so you will have to get the length of the table and iterate through each one.
    - As a counter example: If you decide to use an array for position x, y, z coordinates, then you do *not* need to iterate. You know that there will always be exactly three values, so you can just look them up directly.
  - If a table is a dictionary then you do *not* need to iterate. Instead, look up the keys that you expect to find.
    - (This means that a table could contain keys that you ignore if you are not looking for them. This is ok, especially for our class!)
  - If you *do* decide to iterate through an arbitrary table (i.e. a dictionary) remember that the order is ***NOT DETERMINISTIC*** :-)
    - The order that you add keys to a table is not (necessarily) the same order that they will be returned to you
    - (If you explicitly look up keys in a dictionary instead of iterating you will not have to worry about this potential "gotcha")
- Remember that arrays in Lua are 1-based (the first element is at 1 and not 0)
  - This means than when you are writing Lua code in C you may need to add or subtract 1 to convert indices. This is easy to do but also very easy to forget.
- Doing this part of the assignment is not necessarily difficult, but bugs can be very confusing and frustrating. There isn't really a way to examine the Lua stack in a debugger, and it's easy to get confused about where you are. Remember to take things one step at a time, and if you run into problems ask for help on the discussion board!
- There are different ways you may have swapped the winding order for one platform in Assignment 05, but the way that I intended for you to do it was to swap in the platform-specific initialize function. There is a different way you could do it in this assignment that is probably better, although you may also keep it the way you had it with no change.
  - In a future assignment we will move all of the Lua code out of the run-time code and into the MeshBuilder project, and its job will be to create a binary file. Then your run-time code can just load the data directly without doing any kind of processing.
  - To anticipate this change, you can swap the indices when you are loading them from the Lua file. Then, when you pass the index array to your platform-specific initialize functions they will already be in the correct order.
  - This means that you would put #ifdef preprocessor statements in your platform-independent [MESH].cpp file, which I usually recommend against. This is one case where that is probably a good choice because it can make things more efficient.

How to Debug Build Tools

- Starting in this assignment and continuing through the rest of the semester you will be writing other programs besides the game EXE, and it is important to know how to debug them in case something goes wrong
- Set MeshBuilder as the startup project in your solution
  - Right-click the MeshBuilder project in Solution Explorer and choose **Set as StartUp Project**
  - Now, when you start debugging with Visual Studio (e.g. by pressing F5 or the green arrow) it will run MeshBuilder instead of MyGame
  - (Try it! Set a breakpoint in your EntryPoint.cpp main() function, and then step through the code to see what happens. Debugging will work, but the program will exit really quickly after outputting "error : An asset builder must be called with at least 2 command line arguments (the source path and the target path), but none were provided".)
- You need to set the proper command arguments so that you can debug MeshBuilder the same way that it will be run when you build assets
  - Open AssetBuildFunctions.lua
  - Search for the BuildAsset() function
  - You should be able to read the comments in this function to get a general idea of what it is doing. The important part to find in order to debug is this:

    -
      ```
      -- The following line can be uncommented to see what command line is being executed
      -- (this can be used, for example, to figure out what command arguments to provide Visual Studio
      -- in order to debug a Builder)
      -- print( commandLine )
      ```
  - Uncomment the indicated line, so that every command that is executed while building will also be printed in the Output window
  - Now, build your BuildMyAssets project, and look at the Output window
    - You should see several lines of ShaderBuilder.exe commands and at least two lines of MeshBuilder.exe commands. Each command will look something like:
      - theProgramToBeRun.exe arg1 arg2 arg3 ...
    - Choose one of the lines with the asset that you want to debug
    - (After building the first time you will no longer see the commands because the assets will already be built. If you want to see them again you can either delete your temp/ folder or make some other whitespace change to AssetBuildFunctions.lua)
  - Right-click your MeshBuilder project in Solution Explorer and choose "Properties"
  - In the tree view to the left select **Debugging**
  - In the **Command Arguments** section copy/paste everything exactly from the printed command except for the MeshBuilder.exe path
    - MeshBuilder.exe is the actual "command", which is set to $(TargetPath) by default in Visual Studio. You can leave this without changing it.
    - You need to copy and paste everything else as the command arguments
- You should now be able to successfully debug your MeshBuilder project!
  - (The steps above may not be sufficient for some tools. In a future assignment you may also have to add some things to the Environment section.)

Optional Challenges

- Our meshes so far have been flat and 2D. Can you add any depth?
  - The easiest thing to do is a floor plane. It is still a flat rectangle, but it is oriented along the XZ axes instead of the XY axes, and makes the scene feel more 3D.
  - It is also (relatively) easy to make a 3D cube. If you want to try here are some things to keep in mind:
    - A cube is just a 6 rectangles put together. Try adding one side at a time, making sure that it looks correct.
    - The winding order is important, but it is more confusing in 3D than in 2D. You need to think about which way the triangle is facing, and order the vertices accordingly. The front face will be facing the camera, and the vertices should be ordered the same as the 2D rectangles that we have been making. The back face, however, will be facing the opposite direction (away from the camera), and so it will be ordered backwards.
      - If you haven't done graphics programming before the explanation that I've given above might not be sufficient for you to understand what I mean. If you are confused but want to try and make a cube feel free to ask for more clarification in a class lecture or on the discussion board and I will try to help.