**Instructions**

Provide the code and files that were modified for each part that is applicable in separate folders named accordingly (Part A, PartB etc.). Images below sections (for ex 3d) also should match when that section is done.

**Introduction**

You will start to get a handle on working with ENTT as an ECS and exporting models at a simple level in Blender.

This is what the project looks like to begin:

A blue square with white lines

Description automatically generated with medium confidence

At the end of this work, you should have the game level and all models rendered to the screen in a static image.

Preparing to use the Vulkan API

1. Download & install the latest graphics drivers from your laptop/video card manufacturer.
2. Download & install the Vulkan SDK for your platform: <https://vulkan.lunarg.com/sdk/home>
3. Reboot your computer. (or type **taskkill /f /im explorer.exe && explorer.exe** into a command prompt)

Installing the blender add-on:

This project includes a Blender add-on script that can export a scene in Blender to a JSON and create the objs and h2bs that you will need to get models into your game. While you are provided the assets at the start, you will want to install this add-on (and maybe read through the script) is you want to make changes to the level.

1. In Blender, select **Edit->Preferences…**
2. Select the Addon option in the left-hand menu.
3. Click the Install button in the top right.
4. Select the **dropper4blend.py** in your project folder.
5. After it’s installed, it also needs to be enabled.
   1. Search from ‘dropper’ in the search bar.
   2. Make sure Dropper is checked.
6. To Export your scene, File->Export…->Dropper
   1. Select or name the JSON file you want to write.
   2. Models should be created in the local Models folder to the .blend file

If you decide to explore the exporter script and want to make changes, you will have to remove and reinstall the addon to get the changes to propagate.

Use CMake to build your Project:

1. Download & install the CMake build tool [cmake.org](file:///C:\Users\lnorr_000\AppData\Roaming\Microsoft\Word\cmake.org) (be sure to check “install for all users”)
2. Reboot your computer. (or type **taskkill /f /im explorer.exe && explorer.exe** into a command prompt)
3. Open the directory containing this document in windows explorer and select the path bar at the top.
4. Type **cmd** into the bar and a command prompt should open. Type: **cmake -S ./ -B ./build** enter.
5. This should generate a solution inside a new folder. Open it and set it as your startup project.

##### Part 1a

The base project already has some rendering setup, but we need to get some geometry data into the pipeline. We will use the helper methods in load\_data\_oriented.h to read a level file and its associated models into memory. From there we can send it off to the GPU.

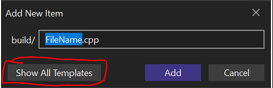
The first thing we need are **components** that represent the level data, one for data on the CPU and one for data on the GPU. Since these are for **Drawing**, these should be with the other DRAW components.

Create a component for loading the level data onto the CPU, I called my component **CPULevel**. This component needs 2 variables to hold paths, one for the path to the JSON we’re going to load and one to the folder where our model assets are. It also needs a Level\_Data to hold the data that we load.

In that same area, create another component to represent the data on the GPU, I called my component **GPULevel**. This component doesn’t need to store anything, it will only be used to create other components.

##### Part 1B

Now that the components exist, let’s create a new .cpp (or two) to house the logic of these components. Be sure to pay attention where new files are being created if you are making the files from Visual Studio. VS defaults to the following view:



You’ll want to click show all templates so you can edit the location of new file. As you can see, it defaults to being next to the project file in build, but that is a transient folder, to be deleted regularly, so not a good place for code. Also, cmake only picks up new files if they are in the Source folder somewhere.

If you explored into VulkanBuffers.cpp (hint: take a little browse through there since we’re going to be using them in a bit) you might see a macro being used at the bottom, CONNECT\_COMPONENT\_LOGIC(). You should see 3 different methods that you can use off the registry: on\_construct, on\_update, and on\_destroy. These method names offer a wonderful clue on how they are used.

Methods attached to on\_ construct and on\_destroy will be called whenever the defined component is **emplaced** or **removed**, respectively, onto an entity. Methods attached to the on\_update are called whenever **patch** is called on the registry for the defined component.

For the components we just created, we only need to do work once, so we only need on\_construct for both. Create a new method for each component and connect it in a CONNECT\_COMPONENT\_LOGIC() area. We will fill these methods out in the next part.

Part 2a

Let’s get these components emplaced so you can test each step as you add it. Find the area in GraphicsBehavior() in main.cpp, right after the renderer is emplaced, and that’s where we’ll emplace our new components.

This is where we will touch on the GameConfig and using the default.ini to get data. If you look directly above the area, we are going to add our components, you will see a call to the (\*config) to get the name of the Shaders this project is using. Now, if you open the defaults.ini file, you will see a section labeled Level1. This is the data that we should be loaded for our levels. Using the defaults.ini to change data to use means that you don’t have to recompile the project to see the changes.

We need to have our CPULevel loaded before the GPULevel, so be sure to emplace the CPULevel (with the paths gathered from the ini) before emplacing the GPULevel component.

Part 2b

Now, let’s put some work into these new components, starting with the CPU level. This component only needs to do one thing, and that’s call LoadLevel that’s located in Level\_Data. To do this, you’ll need to get access to the Level\_Data that is part of the component that is currently being **emplace**d. As you may have seen in other areas of the project, ENTT’s registry has a method called **get** that will return the requested component from any entity. Luckily the required signature of the construct function has an entity in it. This is the entity that the component is being emplaced on. Once you have the instance of the component we are working with, we can use the data as part of it to fill out the call to LoadLevel.

The thing that might seem out of place is the GW::SYSTEM::GLog that’s needed for that call. Simply create an instance of a GLog in this method, and you can name it whatever you’d like in the Create. It is recommended at the start to also EnableConsoleLogging on the log so that level loading information will be printed to the console. This could prove useful.

LoadLevel returns a bool for whether it was successful or not, so it’s not a terrible idea to catch this return to possibly handle a failure.

Part 2c

With LoadLevel successfully called, the Level\_Data in the CPULevel component should now have all the data from the game level. Now, we just need to create some data for the GPU. In the GPULevel component, we’ll need access to the Level\_Data that’s in the CPULevel. Since both components will be on the same entity, we can try and get the CPULevel from the entity that the GPULevel is being emplaced on. In addition to **get**, the registry also has a method called **try\_get**. This acts like a get but won’t fail if the requested component doesn’t exist and instead return a null pointer. While, in this case, it’s almost guaranteed that the CPULevel will exist at time of emplacement for the GPULevel, you want to get in the habit of using error proofing code.

Once we have successfully retrieved the instance of the CPULevel component, we can move on to creating our buffers for the GPU.

Part 2D

You have been provided working components for both a Vulkan Vertex buffer and Index buffer. If you look at these components and the methods provided for them, you should see that both construct and destroy do the boilerplate setup for Vulkan of these buffer types, and it’s the Update that the data is gathered and sent to the GPU.

Looking at both Update\_VulkanVertexBuffer and Update\_VulkanIndexBuffer, you will see these components getting another component into a variable named **cpuBuffer**. This component that it’s getting is the data from the CPULevel that is needed for that buffer. You might notice that the components that are requested aren’t defined structs like we’re using in other places.

In the GPULevel construction method, we need to emplace both a vertex and index buffer component, but also emplacing the data that those buffers need before updating those buffer components with a call to the registry’s **patch**.

All these rendering components are going onto the same entity, so create this series of calls:

* Emplace a buffer component on the entity that also has the GPULevel component
* Emplace the data required for that buffer onto the same entity.
  + Look at the buffer’s update call and see what it’s going to be looking for.
  + Look at Level\_Data and find the variable that stores the required data (verts or indices).
* Patch the buffer component to run the update the method to load the data onto the GPU.
  + registry.patch<BufferComponent>(entity)

Once this is written, you should have vertex and index buffers loaded onto the GPU from information gathered from the game level data that was exported from Blende.

If you were to run the project now, and everything was written correctly, you still won’t see anything. However, they are on the GPU if you feel like digging around in RenderDoc Resource view for the 2 buffers (not important and not really recommended at this stage).

Part 3A

Now that we have the buffers populated, we can start organizing our data into ways for the renderer to put things on the screen.

You have been provided two components that will be used to define what and how to draw various models: GeometryData and GPUInstance.

GeometryData contains the information required to make a call to Draw using the index and vertex data. There is also an overloaded < operator, this will be used later to sort our list of Geometry data.

GPUInstance represents each individual mesh instances and the data needed to position color that mesh. These components will be used to populate the Storage Buffer managed by VulkanGPUInstanceBuffer component.

Working with an ECS involves attaching behaviors to entities, and in this case, the behavior we want is to be rendered. Therefore, our next step is to create entities and add these components to them so they can be rendered once we add that logic. Since the information we need to create these components per mesh is located within Level\_Data, we will build these entities in the GPULevel component after the spot we created the buffers.

Inside of Level\_Data, the blenderObjects member is where you will find all the objects that was in the Blender scene. You can use the BLENDER\_OBJECT to get the LEVEL\_MODEL data for each object. From there, you need to loop through every mesh in the model (in the provided scene, all the models only have one mesh, but you should add support for multi-mesh models).

For each mesh, you will need to create a new entity in the registry, then attach a GeometryData and GPUInstance component to that entity. Between the BLENDER\_OBJECT, LEVEL\_MODEL, MESH, and MESH::drawInfo, you should be able to find all the information you need to fill out these components. Note that for the materials and the index starts, the mesh’s info is relative to the model, so it needs to take the model information into account to get the final information.

Part 3B

Now that we have entities with the required components for rendering, we need to use this information in the renderer. Using data-oriented rendering, to make our draw calls, we need to know what needs rendering. Right now, we are going to render all the models in the level, which is a good start to make sure the renderer and camera is setup properly. In a future assignment we will limit what we render to only the entities that should be rendered (ex no bullets if none have been fired).

In the Update\_VulkanRenderer method, located in VulkanRenderer.cpp, find where the vertex and index buffers are bound. We will be building our draw instructions right after that. There are two primary ways in ENTT to get an iterable list: view and group. A ‘view’ is fast to construct and best used for iterating over a single component type or multi-component sets where the entities in the set always have all the components in the set (example: if an entity has a Transform component it also always has a Health component). A ‘group’ is more optimized to iterate mult-component sets in general. It also offers the ability to sort the list, which is why we are going to use a group to create our draw instructions since we need our models sorted to keep all the similar geometries together to use instanced drawing.

The first parameter in creating a group is the list of components that make up the set inside a entt::get<Component1, Component2>. The second parameter is a list of components that will mark an entity to be excluded from the set if it’s present, using entt::exclude<Component3>. We won’t use this second parameter here, but it will be used in the future, so make a note of it.

This is essentially, our first ‘system’ part of ECS. We need to create a group that gives us all the entities that contain the components we need to perform a certain behavior. In this case, the behavior is to render and the required components are the ones we created before, GeometryData and GPUInstance.

After this group is created, we need to *sort* it by the data in GeometryData:

instances.sort<GeometryData>([](const GeometryData& a, const GeometryData& b){

return a < b;

});

Looking at the GeometryData structure, you will see that the < operator looks at the indexStart of the data. This means that the sort will put all entities that have the same indexStart together, meaning all meshes that share the same model.

Part 3C

GeometryData represents all the information to make a Draw call. GPUInstance represents the data that needs to be sent to the GPU into a StructuredBuffer. To make the draw work, we need two matched lists to keep this data in sync. GPUInstance data can be stored simply in a vector, as it will be put into the VulkanGPUInstanceBuffer after we build the list. For the GeometryData, we also need an instance count to represent the number of times that specific mesh is rendered. Therefore, let’s use a map for GeometryData and an int to represent the instance count.

Loop through all the entities in the group, add that entity’s GPUInstance to the vector and either add or update the map entry for the entity’s GeometryData.

After these lists are created, emplace the GPUInstance vector onto the renderer’s entity and then update the VulkanGPUInstanceBuffer. The VulkanGPUInstanceBuffer will look for a GPUInstance vector to update its buffer.

Part 3D

Now that we have our structured buffer updated and have a list of GeometryData to draw, we can finally render our scene.

After the to bind the descriptor sets, loop through all the GeometryDatas that we gathered and make a call to vkCmdDrawIndexed. Be sure to look at the Vulkan docs to see what each parameter represents. The trickiest part is the final parameter that represents the first number to use for the instance count in the shader. This number needs to be updated so that the instance calls match up with the array of GPUInstances on the GPU.

After this is done correctly, you should now see the scene as it appears in blender rendered in your window when you run the project!

A screenshot of a video game

Description automatically generated