**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 also should match when that section is done.

# Introduction

You will continue to build on the project by adding new functionality. We will be adding player control and rendering only objects that need to be rendered.

At the end, you will have an entity that will be controlled by keyboard input, and enemy object being rendered, and projectiles being spawned on button press (multiple of which will appear on screen). All required setups took place in previous work so it can be used as a starting base.

## Part 1

##### Part 1A

Not everything defined in the Blender level needs to be rendered all the time (such as the projectile). Therefore, we need to adjust our rendering to only render things that are being told to render. To do that, we are going to make use a custom property that’s already on our assets, “gameType.”

Let’s start by adding a new bool to MODEL\_ENTRY and LEVEL\_MODEL to represent that a model is dynamic and not part of the static level geometry. Currently, the only objects that have assigned gameType are the dynamic objects, so you can just check to make sure it’s not ‘none’. EntityData.find("gameType") lets you find entries in json data.

ReadGameLevel and ReadAndCombineH2BS are the 2 functions that handle setting the 2 different classes we’ve updated. Update these functions to set that a model is dynamic.

##### Part 1B

Now that we can tell the models that aren’t part of the static level geometry, let’s get them hidden from the renderer until we are ready to show them. Create a new empty component in Draw namespace to represent the DoNotRender tag.

Back in the GPU level component, for each mesh that is part of a model that is dynamic, emplace the new tag on the mesh’s entity.

Now to ignore these entities for rendering, let’s modify the renderer by updating the group we created to get all the drawable entities. Remember, we mentioned the 2nd parameter of the group creation in previous work. Now we’re going to use it. Use entt::exclude<*Component*> to exclude entities that contain listed components from the group.

At this point, if you run the project, you should only see the 4 walls and the floor. The enemy, projectile, and player models should now be hidden.

A green square with red border

Description automatically generated

##### Part 1C

To be able to create new entities that use the dynamic models we just labeled, we need a way to assign these models to an entity. To do that, we’re going to create a ModelManager component that holds a list of MeshCollections. Each mesh will be represented by its own entity, so MeshCollection should contain a vector or entities. ModelManager should therefore contain a way to get these collections by name, since we will be using the blender name of the objects to find these models.

Now, before we loop through all the blender objects: create a new entity and emplace a ModelManager component on it. Grab the list of collections from that component so that we can add to it in the loop.

For each object, start a new MeshCollection. For each mesh in that object, if the object is dynamic, add that mesh entity to the collection. Once all the meshes are created, add that collection to the ModelManager under the object’s blender name, again if the object is dynamic.

I **strongly** recommend creating a new .cpp for the ModelManager to handle destruction of the MeshCollection component and adding a helper function in next part to get the copies of renderable entities. If you don’t handle the on\_destroy of the MeshCollection now, you might run into some strange behavior in the next work. This cpp will also be a good place to put helper functions for getting Model entity data.

At this point, you still should only see the floor and the walls, but you’ve completed the setup that will allow you to create new entities on the fly with models attached to them.

## Part 2

Now we will start adding Game components, so we will add some new components to the GameComponents file. Since we will be adding the player and the starting enemy entity in this part, let us go ahead and create a Player and Enemy component to tag the new entities with. You might as well add a Bullet component too to save a little time in part 4. These components will be empty as they are acting as tags to identify entities. We will also add a Transform component that holds a matrix to represent the transform of these game objects.

In main.cpp, it is time to start adding code to the GameplayBehavior method. Create two new entities, one for the Player and one for the Enemy. Emplace a MeshCollection and a Transform component of both entities. Emplace the correct tag (Player or Enemy) to these as well.

Both the Player and Enemy have entries in the default.ini to find model names that will need to be looked for, so grab a reference to the config file here. We will use the ‘model’ entry to get the name to look for in the ModelManager.

We need to fill out the MeshCollection of both the Player and the Enemy, and this process will also give us the initial Transform location. Get the single instance of the ModelManager that you added to the registry:

*registry.get<ModelManager>(registry.view< ModelManager>().front())*

This will return the first entity that had a ModelManager component, which in our use case is the only one in existence (hopefully).

Find the MeshCollection that is tied to the model’s name entry for the Player. Then, for each mesh in that collection, we are going to create a new entity that will get put into the Player’s MeshCollection. Each entity created should emplace a copy of the original’s GPUInstance and GeometryData components. Also, set the matrix in the Player’s Transform component to the transform in the first GPUInstance’s transform. This will place the player at the location defined in Blender.

Then, do the same for the Enemy entity.

If done correctly, when you run the project, the Player and Enemy should be back on the screen in the locations they were at the end of previous work you completed.

A green square with holes in it

Description automatically generated

## Part 3

Now that we’re going to start doing some more game play stuff, let’s create a component to handle all the gameplay systems. In the Game namespace, create a GameManager component. It doesn’t need any variables. Create a new .cpp for this component and define an Update method. This is where we will run our game play systems. Create a new entity in the GameplayBehavior method and attach the GameManager component to it.

In MainLoopBehavior, call patch on the GameManager before we update the Window.

The first system we are going to make in the GameManager is a system to update all the model locations. We will achieve this by updating the transforms in all the GPUInstances to the objects Transform component. So, make a view that gets all entities that have both a Transform and a MeshCollection. Then, for each of those entities, copy the matrix that’s in Transform to each mesh GPUInstance’s transform. This means that as we update the game play Transform, the models will move too.

Create a .cpp for the Player component and provide an Update method. Grab the single Input and DeltaTime components so we can use them. Using the ‘speed’ stat from the .ini file to adjust the Player’s Transform with the WASD keys. This game exists on the X/Z plane so those are the directions you should move.

Patch the player in the GameManager since that’s where we are going to be doing all our game play updates.

With this part complete, you should be able to smoothly move the player around the level.

A green square with a red border

Description automatically generated

## Part 4

Create a new Game component to represent a Firing state, that contains a cooldown. We won’t let the player fire if this component is on the entity, and it will be removed once the cooldown reaches 0.

In the player’s update, if the Firing component isn’t present, we will check the fire keys and if they are pressed, we will spawn a bullet and add the Firing component.

G\_KEY\_LEFT, G\_KEY\_RIGHT, G\_KEY\_DOWN, and G\_KEY\_UP are the keys to check for firing. The specific key will determine the velocity when we get to that part next week. For now, if a button is down and we’re allowed to fire, create a Bullet entity like you did for the Player, except the transform is the Player’s location instead of the transform from blender. After creating the entity, emplace the Firing component to the Player.

Before checking for processing the fire, if the Firing component is present, reduce the cooldown and if it’s 0, remove the component.

Bullets should be left behind as you press fire buttons while moving around, and this work is complete.

A video game with a green and blue line

Description automatically generated