Assignment Three

# Introduction

In this assignment, you will continue to build on the project by adding new functionality. We will be adding physics and enemy logic.

At the end of this assignment, you will have enemies moving around on their own, bouncing on walls, and breaking apart on ‘death.’

# Getting started

All required setups took place in Assignment 1. Check there if there seems to be something missing.

# Assignment three

## Part 1 (15%)

While it’s nice that we can move our player, the game will not be much fun if nothing else moves. Since movement is a behavior, and behaviors are controlled by components, we need a new component to represent an entities Velocity. Velocity is a direction of movement that will be scaled by a speed, so that’s all this component needs.

Attach this component to the entities that need it (the projectiles and the enemies). The player doesn’t need it since its movement is based of input in this game. Let the vector be initialized to anything you want for now; we will put in better logic in the next parts.

Now, to get things moving, create a new system in the Game Manager (before we adjust the model positions from last time) to update entity Transforms based on this new Velocity. Since velocity will be already scaled by speed before getting to this system, only delta time is important in this system.

If done correctly, you should now see the enemy and projectiles that you create move in the direction that you initialized the velocity to, and we can move on to getting things moving in the **right** direction.

## Part 2 (15%)

To get things moving in the correct directions, let’s start with the projectiles since they are the simplest. Since we are already checking for the arrow buttons to fire a projectile, use that to set the direction in the projectile’s velocity. Notice that you can fire diagonally, so be sure to take that into consideration when building your velocity vector. The last step before setting the velocity should be to normalize and then scale by the speed value in the tuning file.

For the enemy, you will want to create the vector with a random direction. Add logic to prevent the left/right and up/down values are non-zero, since we want the enemy to move on a diagonal and not just straight left/right or up/down.

## Part 3 (40%)

Now that things are moving, it’s time for entities to start colliding with things. To start, let’s add a Gateware OBB to the MeshCollection component to represent the collider for a model. Also, create a new game component to tag entities that are Collidable.

Back in the GPU component, where we are loading from the level data, find the collider for the object and add it to the MeshCollection that you are creating for each object.

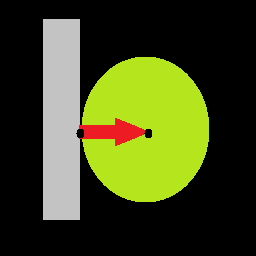
To make the walls Collidable, there is an isCollidable that is already set for tagged objects from Blender. If an object is marked as collidable here, it needs a new entity created to handle the collisions. This entity needs the Collidable tag, the mesh collection (to access the collider), and a Transform (to know where the object is). We should also create a new tag component to represent a obstacle, so we will be able to tell that we hit a wall and not something else.

With access to these OBBs, we can now make a new system in the GameManager, after we get everything moved, to handle collisions. We need to know where an entity is located, the OBB, and if it’s a collidable object. For each entity in this view, we will loop through the rest of the entries and test collisions.

To do that we will need to get our OBBs into the correct world space. For each entity you are going to test, get a **copy** of the OBB. We will modify this copy, but we don’t want to change the initial data. Luckily, Gateware provides with a handful of useful functions. First, we need to scale the extents of the collider in the case that the Transform contains scaling. You can get the scale of a matrix with one of those handy functions. Then we need to transform our center by the Transform. The part that might be new, is setting the rotation quaternion of the OBB. GQuaternion has a method to set one based on a matrix. Get that and then multiply the current rotation by the transform’s rotation. After all these steps, the OBB copies should be in the correct and same space so you can test OBB to OBB. If the result of that test is a collision, then that means the two entities have overlapped, and we can potentially handle what happens next.

The first collision we will handle is the projectiles hitting a wall. The registry provides us with any\_of and all\_of methods so we can test if an entity is assigned certain component types. Use these to determine if a projectile tagged entity collided with a wall tagged entity. In this case, mark the projectile for destruction. The registry provides a **destroy** method to do this.

The next, we will handle enemy entities colliding with walls. The response to this collision will be a little different. We want enemies to bounce of the walls, so we need to update the entities velocity based on the angle of reflection to the wall. The formula for this is: w = v – (2 \* (v ∙ n) \* n); where ‘w’ is the new velocity, ‘v’ is the current velocity, and ‘n’ is the normal of the surface that you are reflecting against. To find this normal, ClosestPointToOBBF to find the closest point on the wall from the enemy position. This will give you a point on the wall. Use this to create a vector from the wall pointing to the enemy position. This will be the ‘normal’ of the wall. I recommend setting both the y and w components of this new normal to 0 since we don’t care about verticality in our game.



At this stage, you should see projectiles disappear when they reach a wall, and the enemy model should be bouncing around inside the box.

## Part 4 (30%)

This last collision type that we will handle in this assignment is projectiles to enemies. Since we want our enemies to sustain a couple hits from a projectile before they are defeated, create a new Health component at emplace it on the enemy entity when it’s created. Initialize the health value to the value in the tuning file. With that new component, if a projectile hit an enemy, destroy the projectile, and reduce the value of the enemy’s health component. This system only handles what to do on a collision, so that’s why we are only going to reduce heath here.

Since the enemy that we have in our game has a behavior of shattering into smaller versions, we need a new component to define that behavior. Let’s call it Shatters. Our enemy will shatter a set number of times, so this new component needs to store how many more times a shatter will occur (initialShatterCount in the tuning file). It also needs a value to scale the new versions by, not currently in tuning, but consider adding an entry for this.

Create a new system after collisions are handled to check the status of all enemy entities that have a health component. In this system, if the health is 0 or less, we will handle the destruction of the enemy. If the defeated enemy Shatters, create shatterAmount new enemies the same way you created the first one, with a few differences. These new version’s Transform should be scaled by a shatter scale value (this is currently not defined in the ini and should be added). The number of shatters remaining should be decremented. If there are no shatters remaining, then that component would not be added as these new enemies won’t shatter when they are defeated.

# Summary

With this assignment complete, you should have enemy units bouncing around the level and projectiles disappearing when they collide with anything. You should be able to hit the enemy several times and then they break apart into smaller versions of the same model.

In the next steps, we will add collision to the player to reduce health and allow us to have game end conditions, win or lose.

# Resources

Documentation for the APIs used in this course are provided here. If you run into issues with any methods in your work, start here.

## ENTT

If you wish to deep dive or learn more about any given aspect of ENTT’s API, check out the ENTT wiki

<https://skypjack.github.io/entt/index.html>

## Gateware

We will be using this API occasionally throughout these assignments for simplicity’s sake. Gateware is a powerful cross-platform API often contributed to by students here at Full Sail just like you. (Designed for 3D Engine builders)

<https://gateware-development.gitlab.io/gcompiler/index.html> (Official Documentation)

*Tip: use the “--->” triple-dash operator on any Gateware proxy to have intellisense show you the actual arguments.*

# FAQ

* When adding the OBB to your mesh collections, make sure that obb is also getting copied everywhere you are copying that mesh collection.