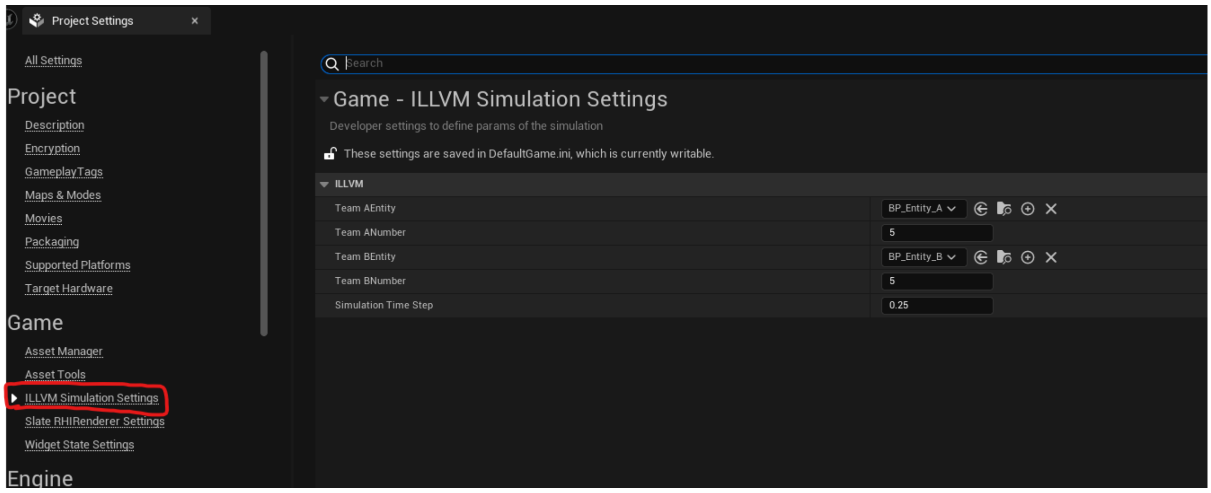
Project Notes

- There isn't any controlled pawn in the scene, so I recommend to execute in simulation mode.
- Two types of entities: Team A(Red) and Team B (Blue).
- Team A States:
    - Normal: Red
    - Receive hit: Yellow
    - Attack: Green
    - Dead: **Black**
- Team B States:
    - Normal: Red
    - Receive hit: Purple
    - Attack: Cyan
    - Dead: **Black**
- Some parameters of the simulation can be configured through project settings.
    - Class of entities to spawn
    - Number of entities to spawn
    - Simulation time step period



Game - ILLVM Simulation Settings

Developer settings to define params of the simulation

These settings are saved in DefaultGame.ini, which is currently writable.

ILLVM

Team AEntity	BP_Entity_A
Team ANumber	5
Team BEntity	BP_Entity_B
Team BNumber	5
Simulation Time Step	0.25

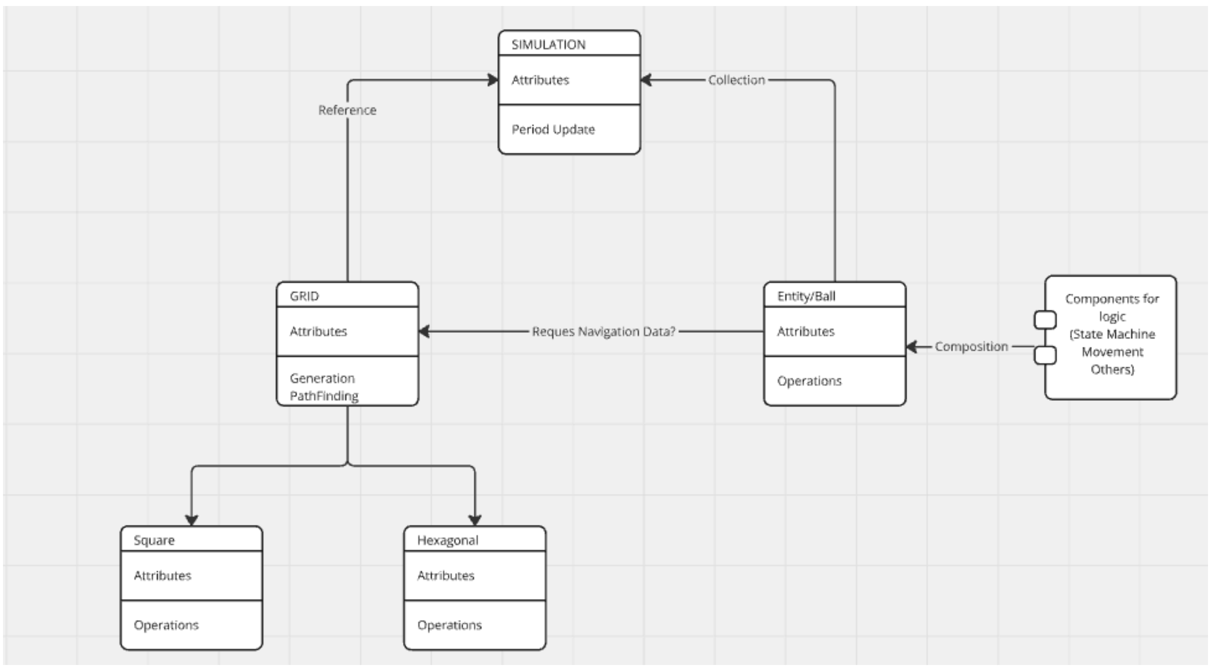
Development Notes

In this section of the document I want to explain how I resolved the proposed test.

First steps

I have been developing first/third person games for a long time, so I'm not really familiar with the concept of simulation games. So the day before, I spent some time watching videos of games as TFT to visualize and understand the proposal of the exercise.

Once I had the main idea, I started creating the project and repository. In the meantime I analyzed the requirements of the documents and prepared a first draft of the expected classes that I would need to develop it.



The image is just a quick UML draft (please don't be puristic with the UML syntax) to visualize how the test architecture would be.Let me explain a little bit more about it:

- Of course I needed an actor to represent the balls (Entity) and from my point of view, entities should always be constructed through composition, so that part of the diagram is pretty straightforward.
- I wanted to try the hexagonal grid but I knew that it would cost a time that could exceed the 8 hours, so I thought of creating a common parent class grid that would contain the methods that would be used by other classes. This way If I had enough time to implement it, I would only need to change the actor in the scene instead of rewriting the whole code (Spoiler: I didn't have time to do it)
- Finally I was sure that I would need a class to update the simulation and probably that class must be responsible for deciding the rules of the simulation. So I was pretty clear that Simulation would be a subsystem (also subsystems are trending).

Implementation

GRID

My approach to start writing the code was: First elements to fill the simulation, then the simulation subsystem. So the first classes implemented were **AILLVMGrid** and **AILLMVSquareGrid.**

I created them as an actor because I thought it's more useful if a hypothetical designer wants to move the grid around the world. Also I thought it was a good idea to not generate the grid in runtime and I used the actor's OnConstruction method to generate the grid in editor time.

I didn't want to lose too much time generating the grid, so I decided on a straightforward solution to manage the grid: a bidimensional TArray of **FUILLVMTile.** Square and hexagonal grids could be managed using two indexes and I would only need to change the methods for generation and pathfinding. I thought of using **FUILLVMTile** as a more complex structure for the Tile data, but finally this design led me to use it just for the world position.

I added a method to show the grid representation through the unreal debug helpers. It could be slightly annoying if you move the grid constantly because I left the flag for persistent lines to be true, but I think it is enough to see the generated logical grid.

Entities

As I mentioned before I was clear that the balls had to be constructed through composition, although to simplify some parts of the logic, I added data to the entities as the current grid position or the current target to pursue and attack.

There are three components in the entities:

- **UILLMVStateMachineComponent:** Responsible for managing the entity through all simulation possible states, such as moving, attacking, receiving damage or death. To represent each state I have defined **UILLMVState** which is an instanced object in the component properties. To update the logic of each state I have inherited a child class for all expected behaviors.
    - **UILLMVIdleState**
    - **UILLMVMoveToTargetState**
    - **UILLMVHitState**
    - **UILLMVAttackState**
    - **UILLMVDeadState**
- **UILLMVState** defines an interface which allows updating the simulation with three methods: 'Enter', 'Update' and 'Exit'.
- **UILLMVHealthComponent:** Responsible of managing the amount of attacks that the entity can suffer,
- **URealTimeMovementComponent:** Responsible for updating the entity location in world space in real time. (I didn't realise that the component was not following the nomenclature ILLMV and when I renamed it the entities blueprint corrupted, so I'm sorry about keeping that name)

To change the entity visualization through the states I declared 4 blueprintimplementable events where the material color is changed. Usually I prefer to encapsulate the blueprintimplementable events through an intermediate method where you can put more logic but in this case I didn't consider it important.

The entities are spawned through the simulation subsystem.

Simulation Subsystem

To emulate the simulation in the server I created a world subsystem. This subsystem can be configured through the project settings and its main responsibility is to update the entities every timestep execution. To simulate the time step I just added a timer that executes a callback every defined time period.

The subsystem stores the spawned entities in three collections, one per team and additional collection that contains all the entities to simplify the iteration over it. Everytime the timer callback is called, the method calls the custom update method of the entities, in which the entities just call the update method of the state machine.

The rules for the simulation had to be defined in this subsystem, so it is responsible for assigning to each entity the other team's target to pursue and attack.