****

**COLD NITES**

Alpha-1

**Base Grid Classes Design Document**

Designed & Implemented by

Yash Chamria

**Change Log**

**Version** – 0.1

**Modifier** – Yash Chamria

**Date** – 16th March, 2021

**Description** – Created the Design Document.

**Version** – 0.3

**Modifier** – Yash Chamria

**Date** – 16th March, 2021

**Description** – Modified Introduction, Design goals and the system overview. Also this design document will now incorporate all the base grid classes.

**Version** – 0.7

**Modifier** – Yash Chamria

**Date** – 16th March, 2021

**Description** – Added High level and Mid-level Architecture. Also updated the Use Case View.

**Version** – 1.3

**Modifier** – Yash Chamria

**Date** – 16th March, 2021

**Description** – Added detailed UML and sequence diagrams for all the Base Grid Character.

**Table of Contents**

1. **Introduction 4**
2. **Design Goals 5**
3. **System Overview and Behaviour 6**

**A. Base Grid Classes Common Features**

**B.** **Base Grid Actor**

**C. Base Grid Character**

**D. AIBase Grid Character**

1. **Logical View 8**

**A. High-Level Design Architecture of the Entire System**

**B. Mid-Level Design of Base Grid Classes**

**C. Detailed Design of Base Grid Classes**

1. **Process View 12**

**A. AutoRepositionToTileCentre**

**B. CheckPlayerOnTile**

**C. Character Movement**

**D. Character Rotation**

**E. Character KnockOut**

1. **Use Case View and Practice 18**

**A. For Class Inheriting from Base Grid Actor**

**B. For Class Inheriting from Base Grid Character**

**C. For Class Inheriting from AIBase Grid Character**

**1. Introduction**

Cold Nites is a Grid-style turn-based game. The player must strategically navigate through the level to survive the cold night, protecting the boy from all the mischievous elements of the city. And, there are always multiple ways to solve the puzzles along the way.

This design module will focus on the construction(architecture) and implementation of all BaseGrid Classes in the game. These base classes act as a bridge between the TileMap(Grid-based Logic) and the actual actor/character that will be spawned in the world. For instance, the player inherits from Base Grid Character, AI inherits from AIBaseGridCharacter and all the interactables (collectibles) inherits from Base Grid Actor.

Below are all the Base Classes included in this document:

BaseGrid Character - Playable and AIBase Grid Character inherits from this.

AIBase Grid Character - All the AI inherits from AIBase Grid.

BaseGrid Actor - All the interactables inherits from BaseGrid Actor.

This document will describe the architecture and design choices that make these Base Classes implementation easy to understand, reuse and debug for all fellow programmers, team members, and major stakeholders.

Below are interest points for the mentioned parties :

Programmers - BaseGrid Classes will handle the implementation of TileMap grid-based helper functions and will provide very simple functions and a consistent environment for all the inheriting classes.

Project Manager(and the Team) - All the tasks during the group meetings were assigned with everyone's and the Project Manager's agreement. This, BaseGrid design module and the code implementation will address all the concerns and will fulfill all the requirements in the game's and team's best interest.

**2. Design Goals**

The design priorities for the BaseGrid Classes are mentioned below:

The design should minimize the complexity of an inheriting class.

The design should allow the programmer to keep reusing the base class to build any specific actor/character.

The design will provide the base for all the future actors/characters in the game.

**3. System Overview and Behaviour**

The Base Grid Classes will provide a foundation, build upon the TileMap (Grid-based behaviour) helper functions and other quality of life functions to make it easy to code for inheriting classes.

**A. Base Grid Classes Common Features**

All the Base Grid Classes handle the TileMap Registration System at the base level, so any inheriting class doesn't need to worry about it.

Additionally, it provides some quality of life functions such as AutoReposition() .i.e. the actor will reposition itself on the correct Tile when the BeginPlay() is called making the Level Designer job easier. For instance, can just roughly throw props or AI and they will align automatically when the game starts.

They will also store fundamental variables such as TileMap, Player and AI stuff depending on the relevance for the class.

**B. Base Grid Actor**

Base Grid Actor is the foundational block for all the interactables in the game and inherits from AActor.

All the interactables future actors will inherit from this class.

Now, with all the common features, it provides some basic parameters such as a root component and a mesh.

Also, it provides the function to detect any actor or player.

**C. Base Grid Character**

Base Grid Character inherits from ACharacter and will be the base for all the moving characters in the game.

Playable Character and even AIBase Grid Character inherit from this class.

Other than common features, it is mainly responsible for in-game character movement and registration system.

For the movement, it also does series of 'if checks' such as if the next tile is walkable or not and handles the smooth transition as well.

Base Grid Character also provides rotation functions.

It also provides KnockOut() function in the base class.

**D. AIBase Grid Character**

AIBase Grid Character inherits from Base Grid Character and will be the base class for all the future AI Characters.

As a result, it also inherits all the movement, registration and common feature from the get-go.

It comes with additional features such as player detection, player knockout and actor in range check.

**4. Logical View**

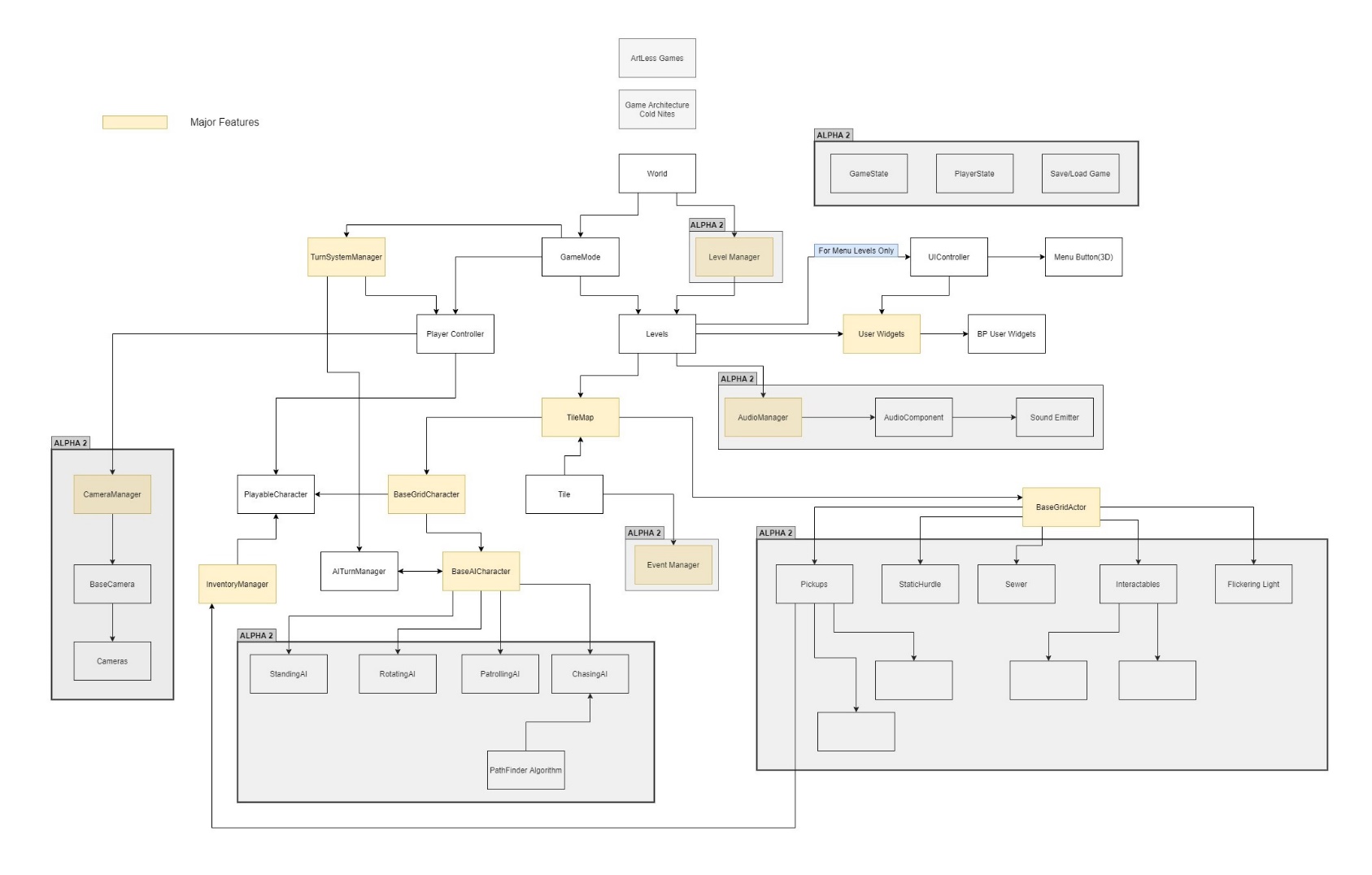
The logical view describes the high-level architecture for the entire game from all the core classes to high-level relations and interactions between them with a flow chart making it easy to read and understand.

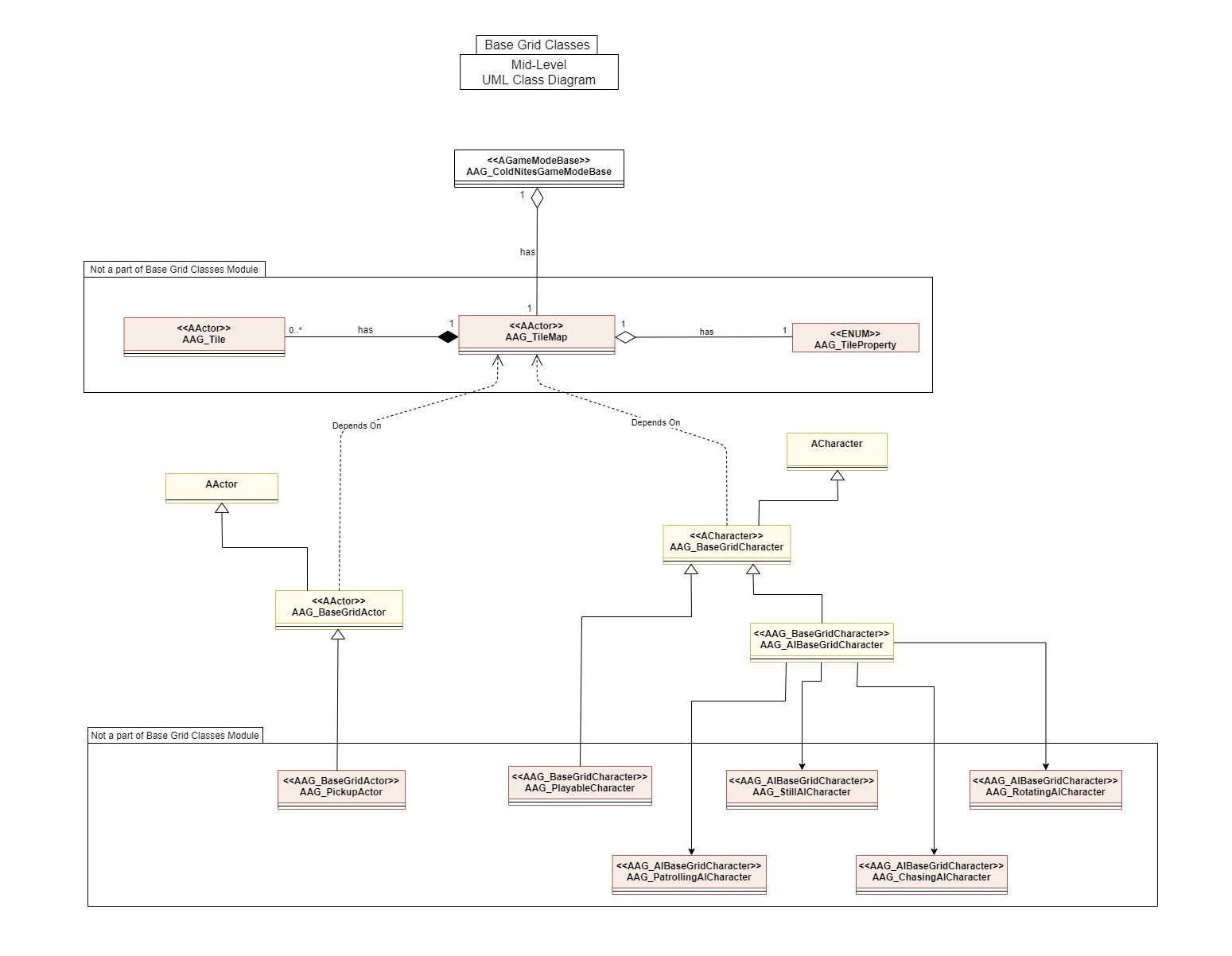
Later, it will dive deep into the high-level and detailed design for the TileMap Module, using a UML Class Diagram.

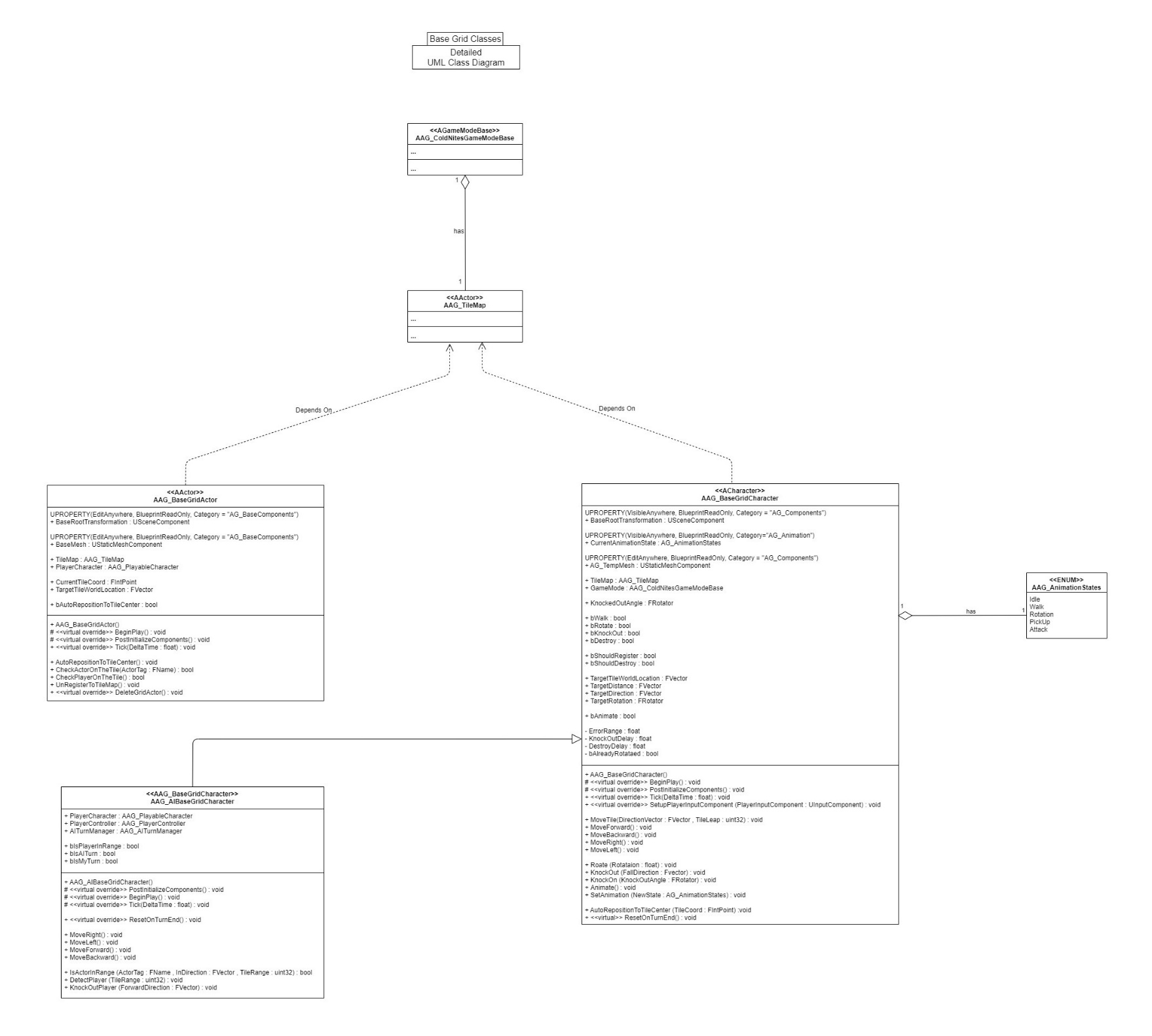
**A. High-Level Design Architecture of the Entire System**

The primary features for the Alpha 1 release:

1. **TileMap** - TileMap provides the grid-based behaviour for the game and will facilitate the event system, based on the actor present on the Tiles.
2. **Turn-Based System** - This provides the turn-based aspect for the game. It is responsible for maintaining the turn order for all the world elements(actors) and the player.
3. **Base Grid Classes** - These classes work as a foundation class for all the actors/characters spawned in the game. These classes are closely integrated with handling the TileMap(Grid-Base) Behaviour of the game.
4. **Player Character** - Player is a controllable character that inherits from BaseGridCharacter, which takes user inputs to perform appropriate moves.
5. **Inventory System** - The pickup function helps the player grab the items on the map. Inventory stores the items for the corresponding actor and will allow the player easy access to any collectible throughout the game, and it also assists in equipping the stored items.
6. **Menu Interface** - The Menu Interface will be responsible for Main Menu and Pause Menu with which the player can interact.



**B. Mid-Level Design of Base Grid Classes**

**C. Detailed Design of Base Grid Classes**

**5. Process View**

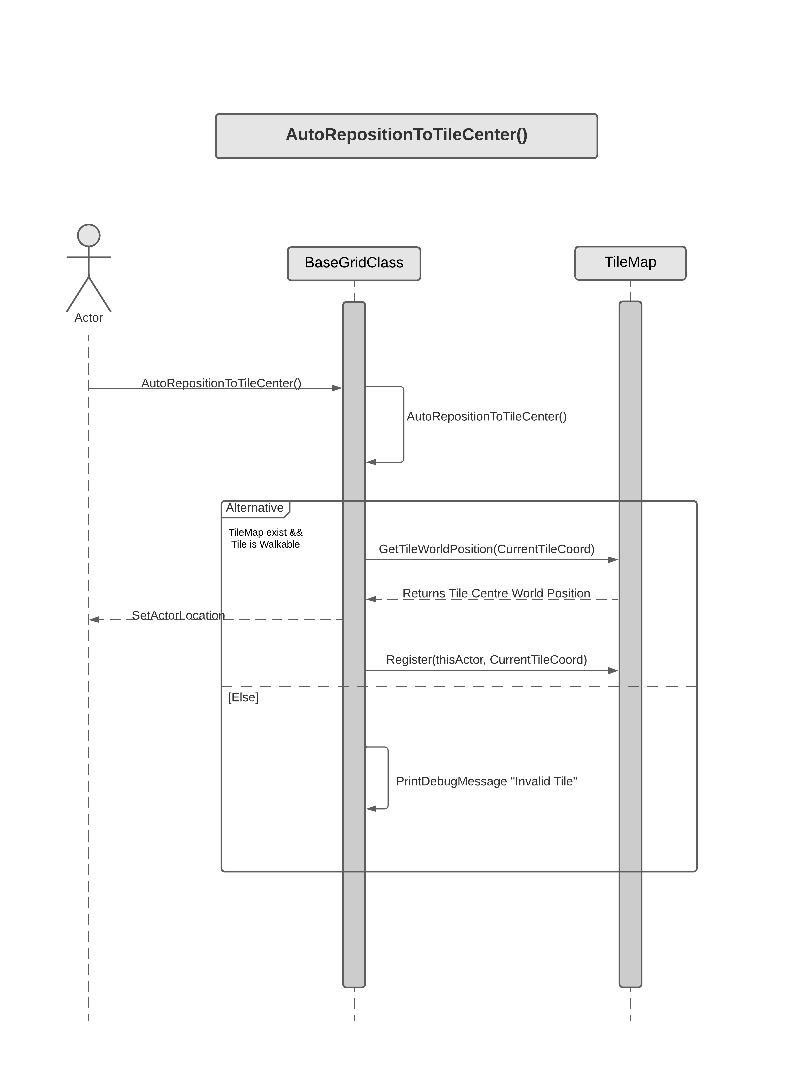
The process view will explain the relation and interaction between various cases using Sequence and Collaboration Diagrams.

**Common Feature in All the Base Grid Classes**

**A.** **AutoRepositionToTileCentre**

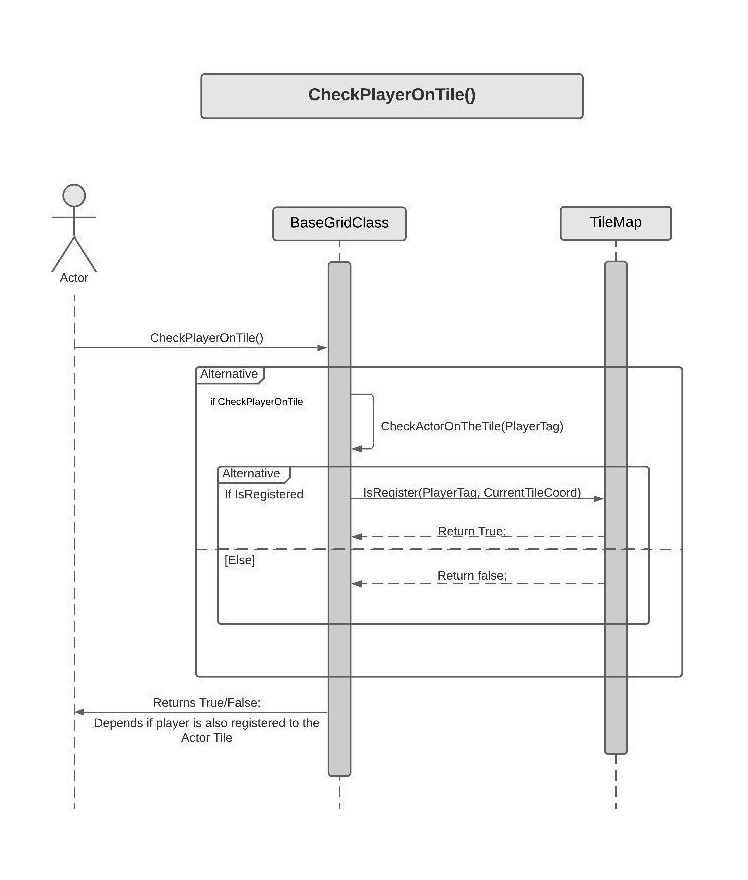
All the BaseGrid Actors/Characters AutoReposition themselves to the Tile's centre and handles the registration to the current tile respectively.

AutoRepositionToTheTileCentre() checks if the actors current tile is valid/walkable or not. Then it gets the TileWorldPosition which returns the tile centre and sets the actor to that position.



**B.** **CheckPlayerOnTile**

An Actor can check if the player is in range or not, just by calling CheckPlayerOnTile(). Internally, it will call a CheckActorOnTile() passing player tag as a parameter. This functions 'if checks' if the Current Actor Tile, also consist of the player character in the RegisterActor array. If true, the CheckPlayerOnTile returns true or vice versa.

****

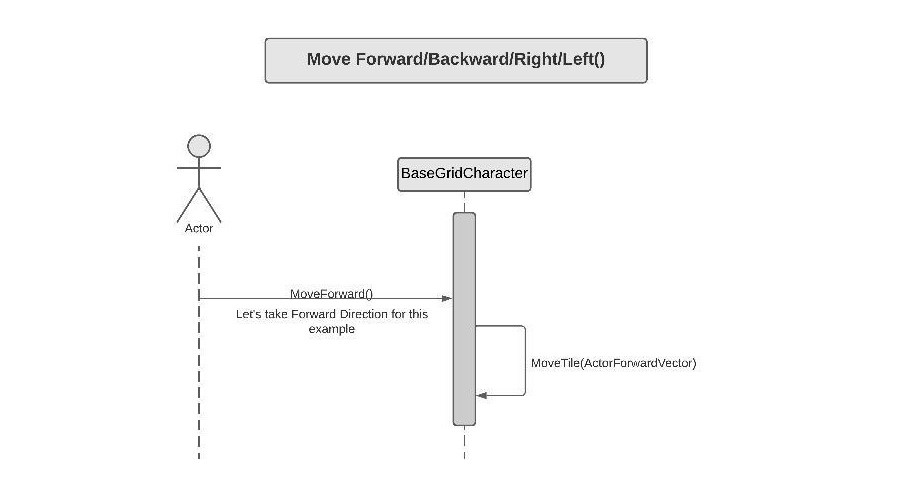
**Base Grid Character Feature**

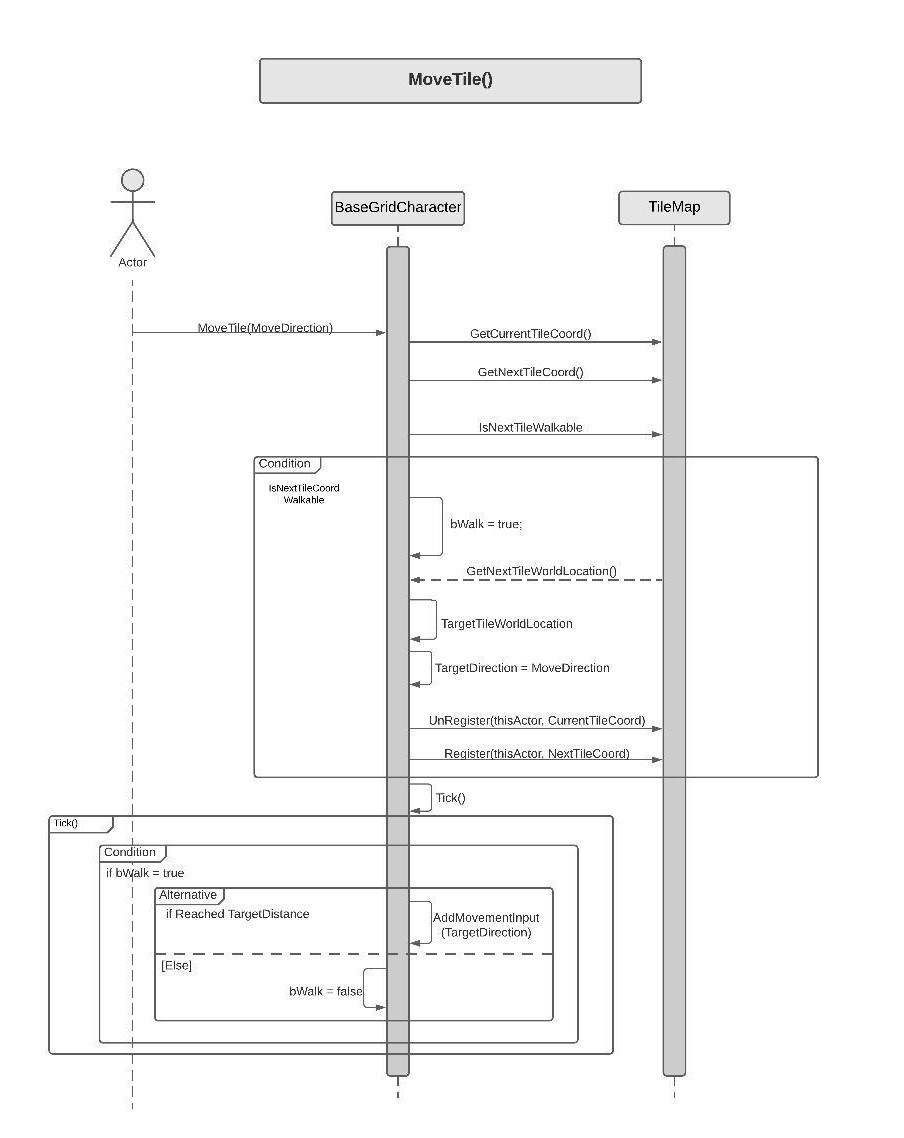
**C. Character Movement**

MoveTile is the main function used for the character movement throughout the game. Even, Move Foward/Backward or Move Left/Right internally calls move tile passing the appropriate direction.

In MoveTile, Base Grid Character gets his CurrentTileCoord and NextTileCoord from the TileMap. Then it checks if the NextTileCoord is Walkable or not. If true, it sets a bWalk bool to true. And also sets the TargetWorldLocation and TargetDirection based on the passed direction. Also, if a character is not opt-out for registration, it will unregister the character at the current tile and register it to the next tile.

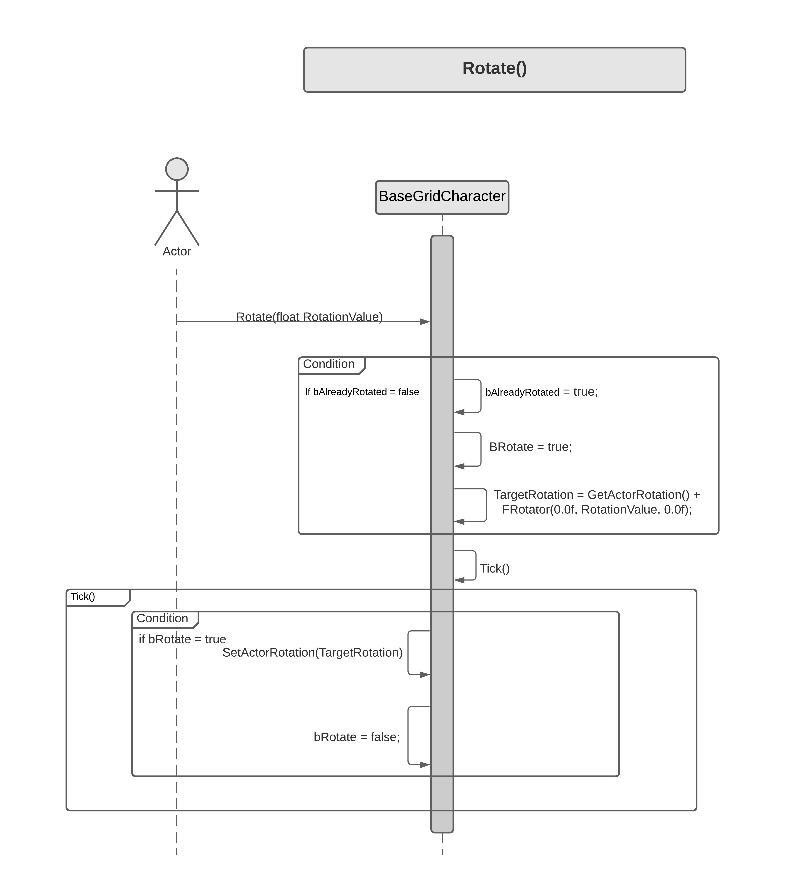
Tick() handles the smooth transition of the character from the current tile to the next. If bWalk is true, AddMovementInput keeps applying the force until the character reaches the desired tile position. Once reached it will set bWalk back to false.

****

****

**D. Character Rotation**

Rotate() handles the character Rotation in the game. Mainly it works with a series of bool. On calling Rotate, it checks if the player is not already rotated for that turn. If not, it will set bAlreadyRotate and bRotate to true. It will also set the TargetRotation. In Tick, if bRotate is true, it will set it to false and will SetActorRotation(TargetRotation).

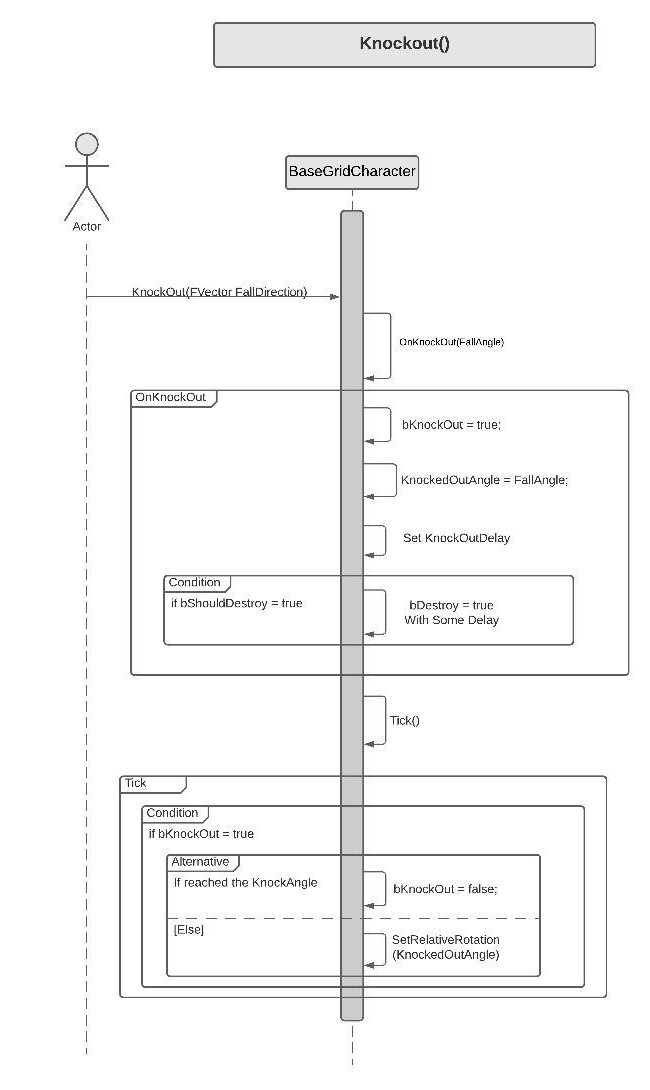
****

**E. Character KnockOut**

Calling KnockOut on a character will set its static mesh horizontally. Internally, It will convert the passing direction vector to rotator and calls OnKnockOut.

OnKnockOut only sets bKnockOut to true, sets appropriate KnockoutAngle, and destroy timer will be trigger if the character is opt-in for bshouldDestroy.

In Tick(), if bKnockOut is set to true, character mesh rotation will be set at KnockoutAngle.



**6. Use Case View and Practice (Application)**

The use case will focus on showing the uses of Base Grid Character at different stages in the game and will explain its application so that this module can act as a guide/reference for someone not quite familiar with the BaseGridCharacter codebase.

**A. For Class Inheriting from Base Grid Actor**

Just inheriting from the BaseGridActor will hook up the inherited actor for TileMap Registration System and will handle AutoReposition and other small stuff.

There is already BaseMesh available to set any desired mesh in inherited class.

For the inherited Actor there is already a helper function to check if the player is on the Tile. For Instance,

*if (CheckPlayerOnTheTile())*

*{*

*//Player->DoPickup*

*}*

**B. For Class Inheriting from Base Grid Character**

Inheriting from the BaseGridActor will hook up the inherited actor for TileMap Registration System, will handle AutoReposition and some other small stuff. It also comes with nifty functions such as MoveTile, MoveLeft/Right/Forward/Backward, Rotate, KnockOut, etc.

Any inherited class can just call MoveRight to move right and BaseGridCharacter will handle all the checks -

*MoveRight();*

Similarly, to rotate call,

*Rotate(float rotation);*

To KnockOut,

*KnockOut();*

One can opt for being destroyed or not on KnockOut by setting

*bShouldDestroy = false/true;*

**C. For Class Inheriting from AIBase Grid Character**

Since AIBaseGrid Character inherits from BaseGridCharacter, it incorporates all the grid movement, registration and other small stuff by default. Other functions added for AIBase specifically are Actor/Player Detection and to KnockOutPlayer and it will be more tightly integrated with AITurnManager in future iterations.

Any Inheriting AI can knock out the player character quite simply –

*if (IsActorInRange("AG\_PlayableCharacter", GetActorForwardVector(), 1))*

*{*

*KnockOutPlayer(GetActorForwardVector());*

*}*

*Or*

*if (DetectPlayer(AITileRange)*

*{*

*KnockOutPlayer(GetActorForwardVector());*

*}*