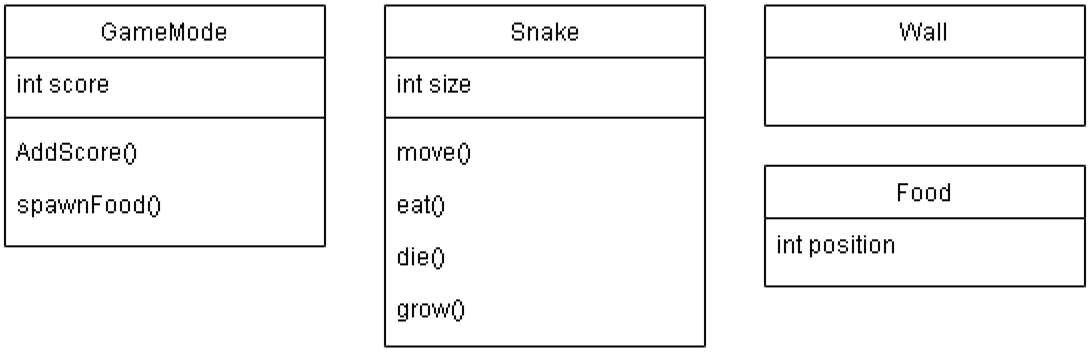
Documentation - Snake Clone

The following project features an implementation of a clone of the well-known Snake game in Unreal 4.27.

While planning the project, the class diagram initially looked like this:



The player-controlled snake would detect collisions with objects like food and wall, and then call the corresponding methods to eat, die or grow. Game Mode class would handle the score count and random food spawn during the game. Wall and Food do not accomplish much on their own, and are rather object that snake can collide into. Classes also do not interact much with each other. However, while doing the project, the game structure has evolved a lot, to incorporate things like snake body movement, which will be explained in more detail on the following pages.

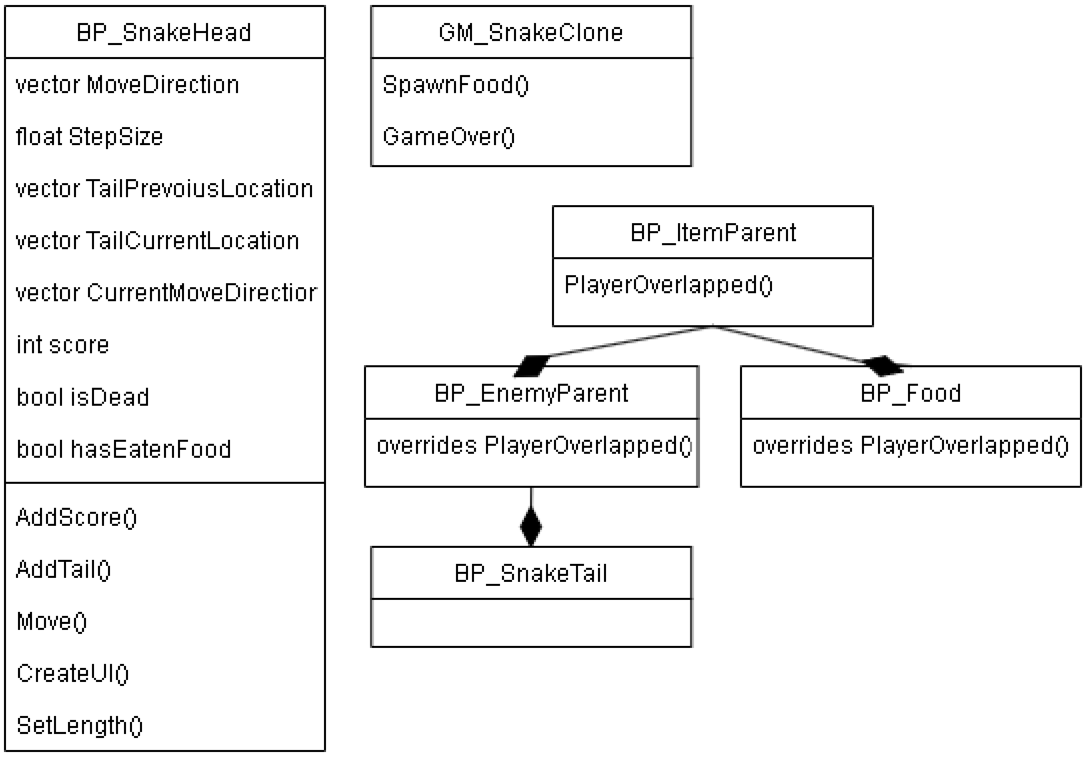
Code/Naming Conventions

To not clutter the project repository, standard Gitignore file for Unreal Engine provided by GitKraken was used. After some research on the best practices when working with Unreal and Blueprints, following rules were established:

Blueprints should be given clearly understandable names according to the following naming convention: **AssetType_AssetName**. For example, the blueprint SnakeHead should be called **BP_SnakeHead**, and so forth. Additionally, Blueprints should be **sorted in the corresponding folders** inside of the content folder, for example all blueprints that belong to snake character should be in the Snake folder, game maps should be in Maps folder, etc.

When coding with blueprints, it is necessary to have understandable, straightforward code flow. This should be achieved through **keeping the execution white line straight** by adjusting blueprints accordingly, and using comments if necessary. The variables should have understandable names. Additionally, a blueprint should have **not more than 50 nodes** to remain comprehensible, and unused nodes should be deleted.

Over the course of development, the class diagram has expanded to the following structure:



The snake is divided in two separate parts, the head and the tail. Head can be controlled by the player, eat food and die. The tail only follows the head and poses a danger to the player. Each tick, the head moves in the direction of the current moving vector. The player can control the snake by changing the **movement vector** with wasd or arrow buttons. However, there is a safety check that forbids to change the direction of the snake backwards, to prevent the snake from immediately crashing into itself. After eating some food, a tail object is spawned on the stored previous position of the head with his next move position being the current head position. New food also spawn after eating the previous one, and there is a safety check to prevent the food from spawning on the edges of the playing field in the walls. The tails and their current and future positions are saved into an array and they each **move to the position of the next one** each tick, so they end up always following the snake head.

Furthermore, an **ItemParent** class exists that provides functionality of spotting overlaps with the player. It then calls an empty function PlayerOverlapped(). The Food class can later **override** that function to call the functions AddScore() and AddTail() in the Snake class. Instead of having a separate class for walls, an **EnemyParent** class exists, which also inherits the player detection ability from the ItemParent class, and **overrides** the function PlayerOverlapped() to call the GameOver function in the Game Mode. The walls can then be created as objects of the EnemyParent class. The snake tail inherits the ability to kill the player form the EnemyParent as well.

To ensure that the game mechanics are functioning as intended, three **Unit tests** have been set up. The first unit test checks the snakes ability to eat food, by directing the snake towards already placed food object. The test is passed if the eating, growing and scoring methods are called successfully. The second tests checks the game over condition, by letting the snake run into the wall. If the GameOver method is executed, the test is passed successfully. The third test checks the scoring system, by letting the snake eat some of the placed food and **comparing** the integer Score with an expected result. If the score has changed from 0 to 1, the test is passed successfully.

Due to all game objects being 3D, the food mesh might sometimes look somewhat bigger than it is, especially if it spawns on the edges of the playing zone, because of the angle the square mesh to the camera.

In general, it is advisable to play the game in the **standalone game window**, (or run the .exe in the build) for the best experience, as playing the Unreal editor requires a mouse click to be able to insert inputs to control the snake.