Close Combat: Swordsman

Documentation

(Unreal Engine 4 asset)

https://www.unrealengine.com/marketplace/close-combat-swordsman

Video overview of the project

Documentation written by the project author ZzGERTzZ.

Manual version : 1.0

1. Introduction:

The Project **Close Combat: Swordsman** is created to provide developers with a template for games of different genres : action games, RPG’s, slashers or fighting games. The template includes a combat system, a system for character movement, a system for managing character’s level up, and a system of skills for a character.

The package includes the following elements of gameplay:

Movement. Basic movement with a procedural lean of the character is implemented to use fewer animations while feeling the character’s response to the change of direction during the movement. There is a jumping system, as well as a system for movement in running and walking states. A combat system is also available, where it is possible to walk in a fighting stance in all directions, while making dashes or dodging an attack, etc ...

Health component. This component comprises all parameters of character’s health. It can be used not only with the character, but with any **actor**, which might need the health parameters. For example, a turret, which must receive damage when being hit. The component also has a system for health regeneration, which might be useful when a constant health regeneration is required.

Melee Weapon. The class of melee weapon for close quarter combat comprises the visual representation for the weapon, settings for the weapon damage, and types of strikes. In the project, there are 3 different swords stylized differently: *Futuristic sword, Sword of Light, Sword of Darkness.*

AI. Basic Artificial Intelligence is capable of hearing, seeing, following, and attacking the player from short range. The AI can dodge attacks, maneuver during a fight, block attacks, etc … Also patrolling over a game level is available. There is a convenient and easy way to customize the character’s behavior in a combat, which can be modified dynamically.
Also, in the project, there is a turret, which can fight at long range.

XP management component. The component comprises all necessary logic to handle leveling up of a character. It also has all necessary elements for a system of skills such as the cool-down time (time interval for the skill recovery), active and passive togglable skill types.

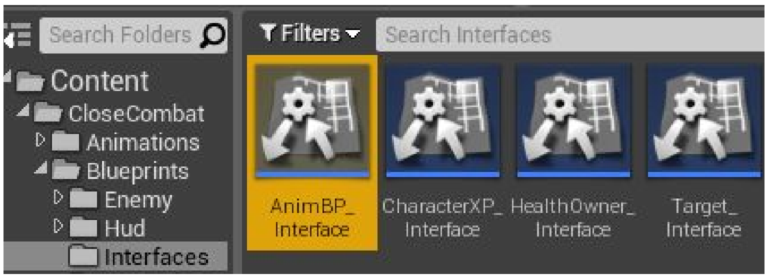
Game Interface (HUD). There is a fully animated game interface comprising HUD elements like elements of health of both character and enemies, indicator of the character level and experience points (XP points), elements of visual representation of character skills.

Combat overview. A cinematic camera in combat.

2. Basic movement

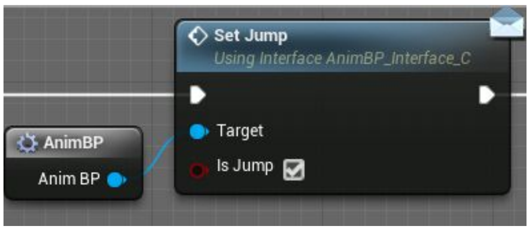
The character movement is based on the basic class for moving from "*Epic games*" **character movement**. This is why adjust all the parameters of movement in this component. The character can move in any direction and do jumps.

The Pawn animation is passed to Anim Blueprint via Interfaces



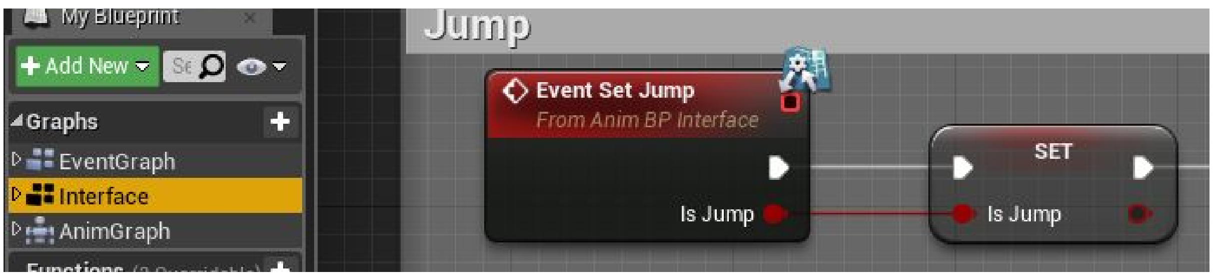
It has been done so to avoid binding the character's pawn to a particular Anim Blueprint. In future, this should simplify integration of other characters, which share the same skeleton. After the integration of the character, you can use the direct reference to Anim Blueprint via "cast to" or by adding functions to the interface "AnimBP_Interface".

Let's consider jumping to illustrate this with an example. Several calculations are being executed once you press the button for jumping (can the character jump at this moment?).

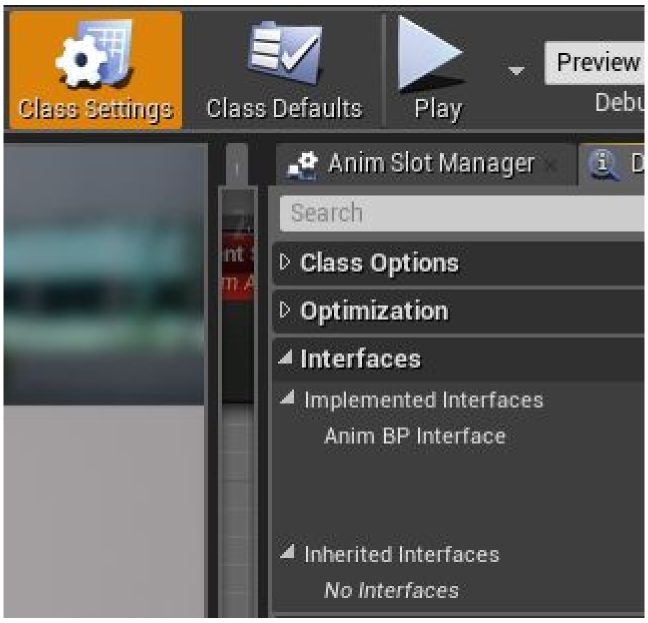


If the character starts jumping, we call the interface message "Set Jump" (it is a function, which we have created in AnimBP_Interface beforehand) to call the necessary event in Anim Blueprint, which activates the character animation and sets it in the state of jumping.

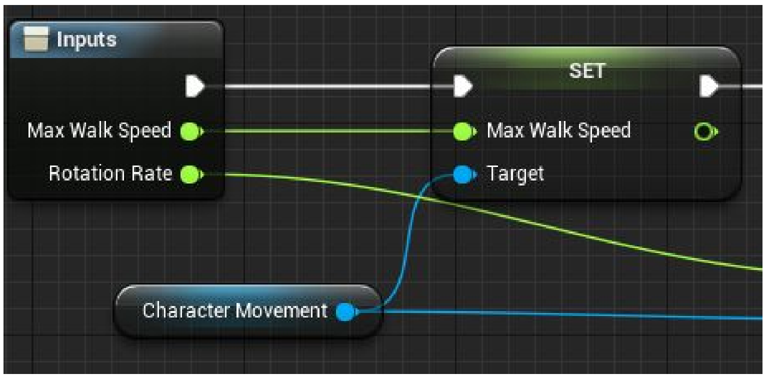
All events of interface AnimBP_Interface in Anim Blueprint are located here:



You must add the interface in “Class settings” before you can call its events.



As far as the logic of character movement is concerned, it is based on dynamic changes of parameters of class component “character movement” from “Epic Games”. For example, when transitioning from running into walking, the top speed is changed by pressing the button for walking (ALT by default).



The character animation in Anim Blueprint is changed based on the character speed.

This is how the character animation and movement are interconnected within the package “**Close Combat: Swordsman**”.

3. Health component :

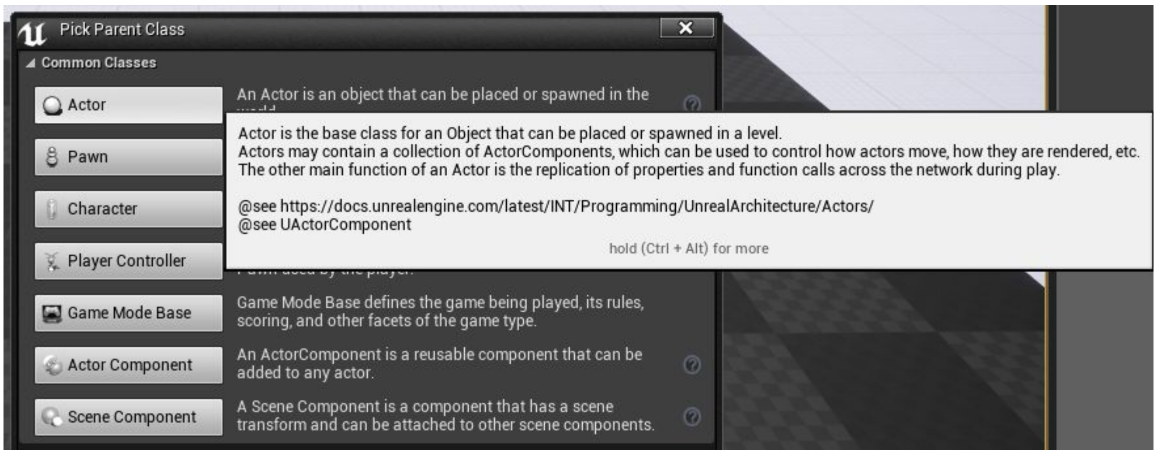
Health component is for the unified system of assigning health parameters to any actor.

This is how it works:
When you attach the component to an actor, you pass information into the component about actor’s damage (any damage). In its turn, the component computes health and, at particular settings, restores health or calls for Event Death in case there is no health left anymore. Event Death is passed via Interface to the actor, to which this component is related.

How to use health component:

Let’s consider an example of how to use health component.

1. Create an empty actor. Click RIGHT MOUSE BUTTON inside the content browser and choose Blueprint Class.



Pick Parent Class

Common Classes

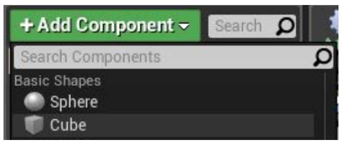
Actor	An Actor is an object that can be placed or spawned in the world
Pawn	
Character	
Player Controller	
Game Mode Base	Game Mode Base defines the game being played, its rules, scoring, and other facets of the game type.
Actor Component	An ActorComponent is a reusable component that can be added to any actor.
Scene Component	A Scene Component is a component that has a scene transform and can be attached to other scene components.

Actor is the base class for an Object that can be placed or spawned in a level.
Actors may contain a collection of ActorComponents, which can be used to control how actors move, how they are rendered, etc.
The other main function of an Actor is the replication of properties and function calls across the network during play.

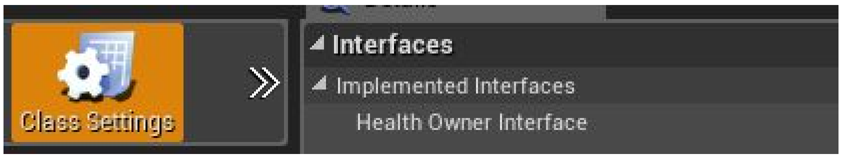
@see https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Actors/
@see UActorComponent

hold (Ctrl + Alt) for more

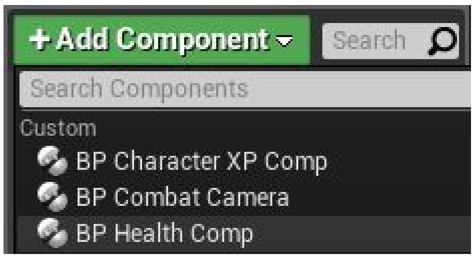
2. In this example, add a cube into our actor. It acts as a visual representation of our actor.



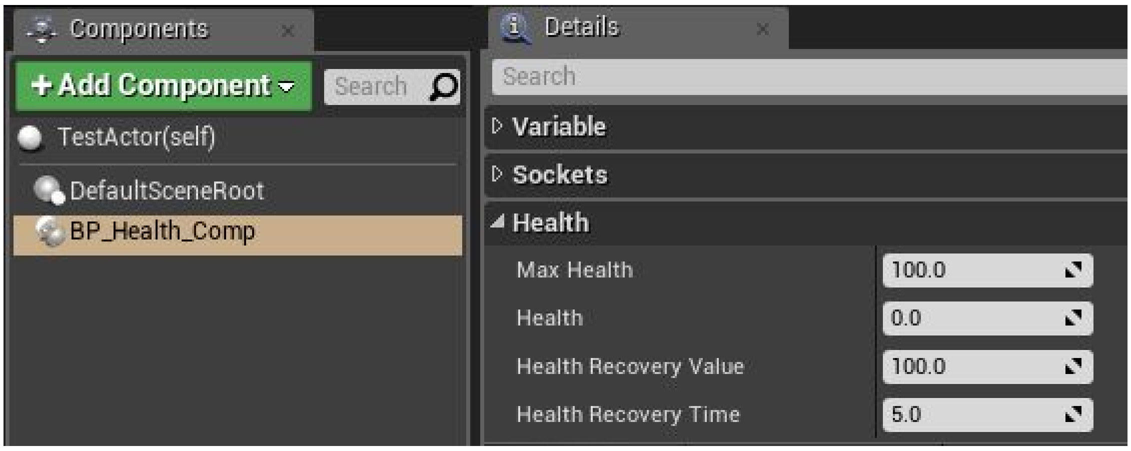
3. Since health component has the unified system, it passes events via interfaces. At first, add an interface of health component to our actor. To do this, go to class settings and add interface “**HealthOwner_Interface**”.



4. Add health component to our actor.



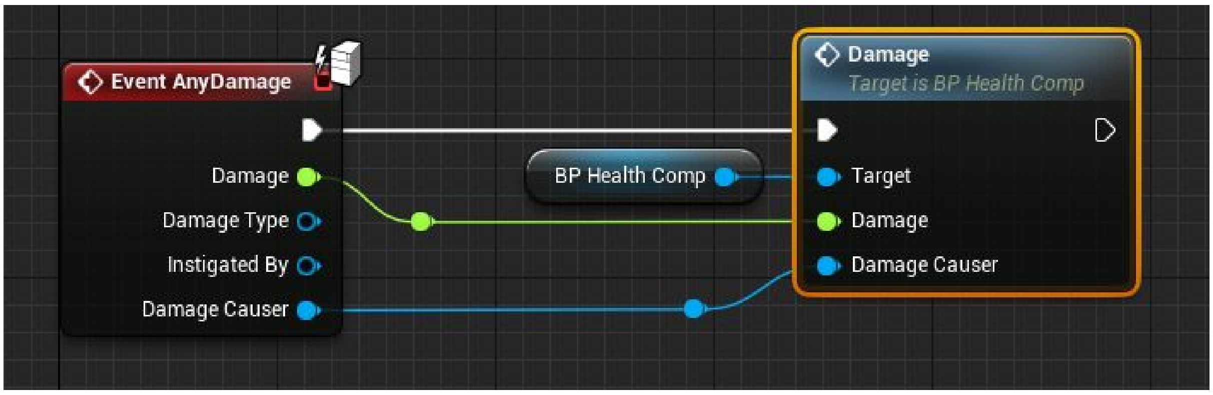
5. Customize our health component. To do this, select health component "BP_Health_Comp". Then its settings will appear in tab "Details".



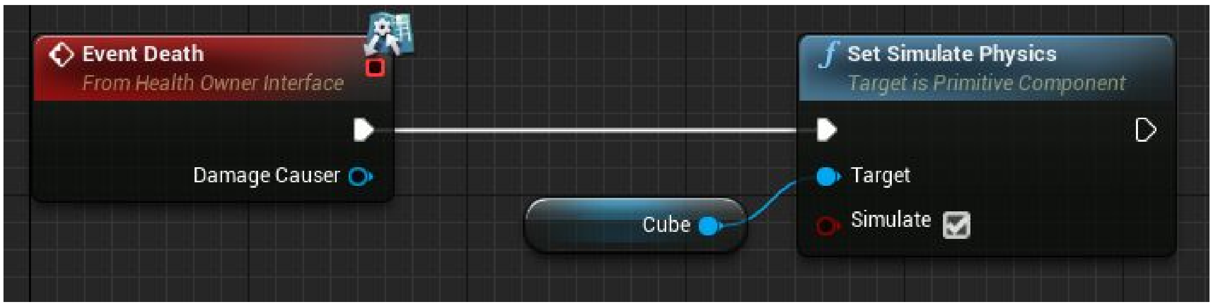
Max Health	100.0
Health	0.0
Health Recovery Value	100.0
Health Recovery Time	5.0

Parameters:
Max Health is the maximum amount of health.
Health is a current value of health.
Health_Recovery_Value determines the value of health till which the regeneration of health will continue. At zero value, the health regeneration will be on until the full health is restored.
Health_Recovery_Time is the amount of time it takes for health to regenerate. There will be no regeneration at zero value.

6. Pass the information from the damage nodes to health component. In order to do so, call event from component Damage when the actor takes damage.



7. The final step is to call Event Death. Add the event of Interface "Death" in the event graph. This event triggers your custom death logic. For our test actor, I have added Set Simulate Physics to our cube.



Now you may drag into the scene our actor Cube, and after several strikes (depending on how much health you have set), the cube will become a physical object, and it will fall.

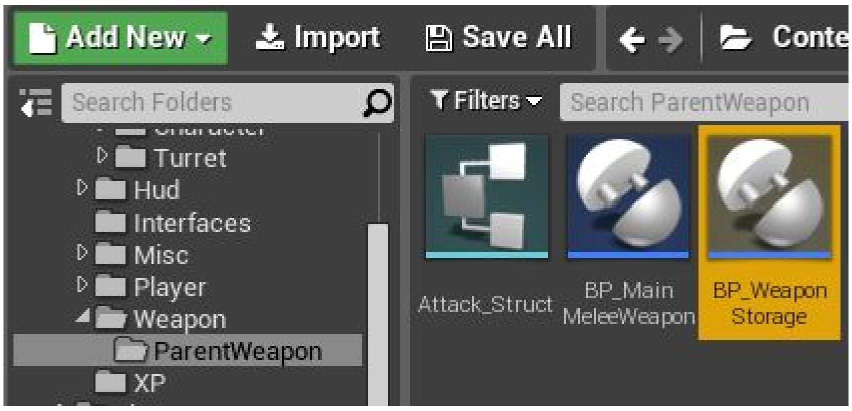
4. Melee Weapon :

The weapon in the project is also a class component. You add it to the character with the help of the node add component. This component contains information about the weapon model, the rate of strikes and damage caused.

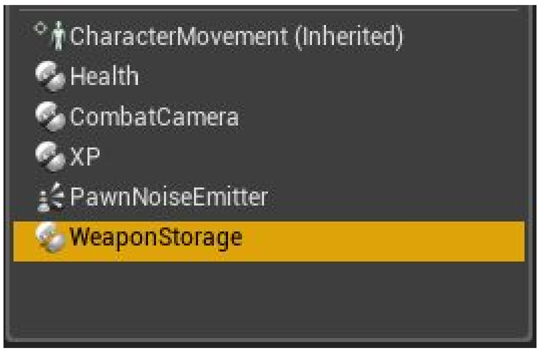
How it works: When you press the button to strike and all conditions are met (the character is not in the air, not dead, etc …), the character talks to the component and receives an animation of the strike from it, the animation speed (the speed of the attack), time moment after which the attack can be interrupted (begin moving after the swing and do not wait till the animation is over).

How to add melee weapon.

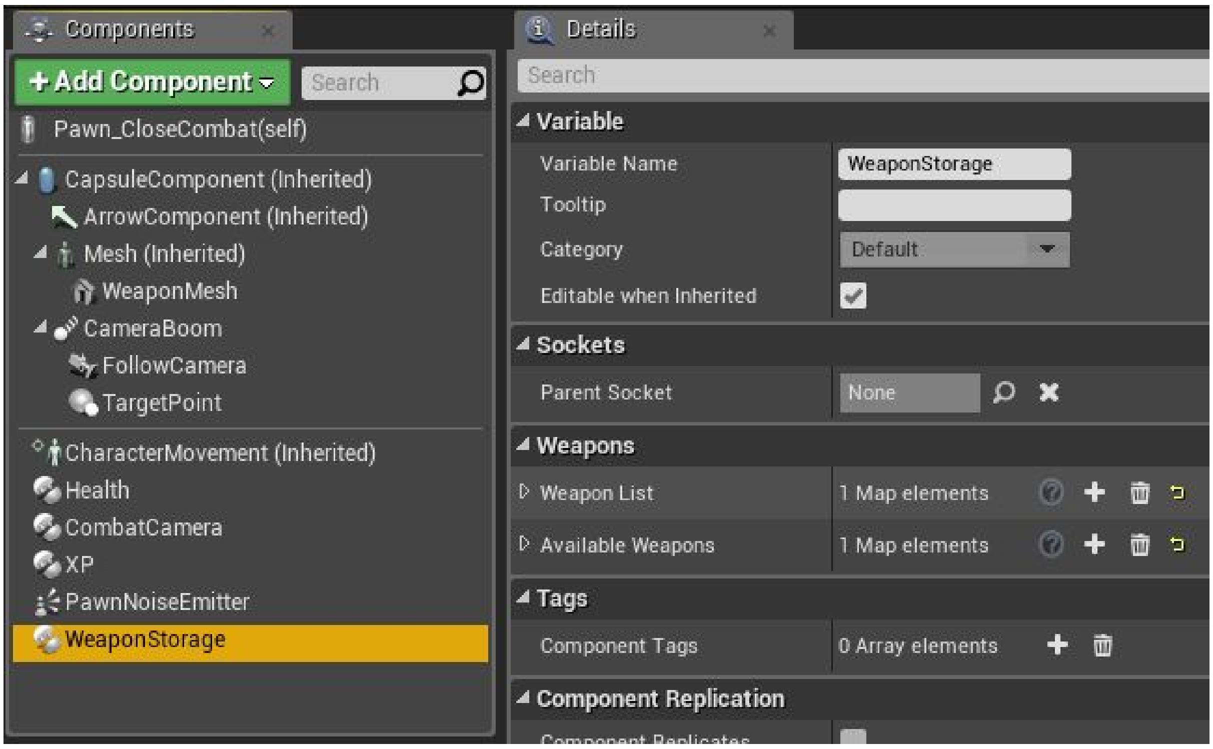
In the second update, there is a special component “BP_WeaponStorage” for creating and storing weapon in the project. It is in the folder of the weapon class.



The next step is to add the component to the Pawns (player and enemy). By default, these components have already been added and renamed (deleted prefix “BP_”).
It has been renamed as “WeaponStorage” in the Pawn by default.



Select the component and find section “Weapons” in tab “Details”. There you will see two map-variables:



Components

+ Add Component

Pawn_CloseCombat(self)

- CapsuleComponent (Inherited)
  - ArrowComponent (Inherited)
  - Mesh (Inherited)
    - WeaponMesh
  - CameraBoom
    - FollowCamera
    - TargetPoint
- CharacterMovement (Inherited)
- Health
- CombatCamera
- XP
- PawnNoiseEmitter
- WeaponStorage

Details

Variable

Variable Name	WeaponStorage
Tooltip	
Category	Default
Editable when Inherited	✓

Sockets

Parent Socket	None

Weapons

Weapon List	1 Map elements
Available Weapons	1 Map elements

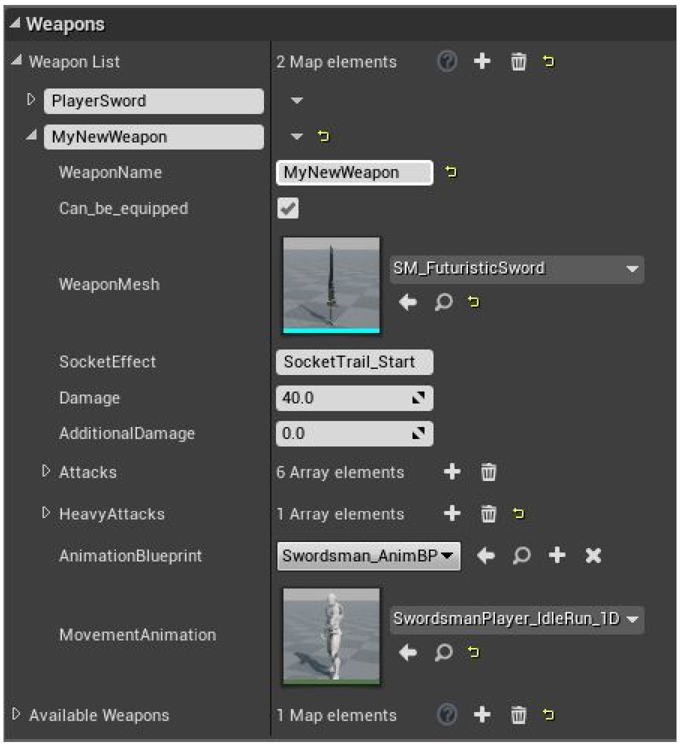
Tags

Component Tags	0 Array elements

Component Replication

WeaponList is a list of all weapon that can be added to the character. *AvailableWeapons* is a list of all available weapon (kind of weapon in inventory). In the list, it is possible to store weapons in which you modified their initial parameters (upgrade).

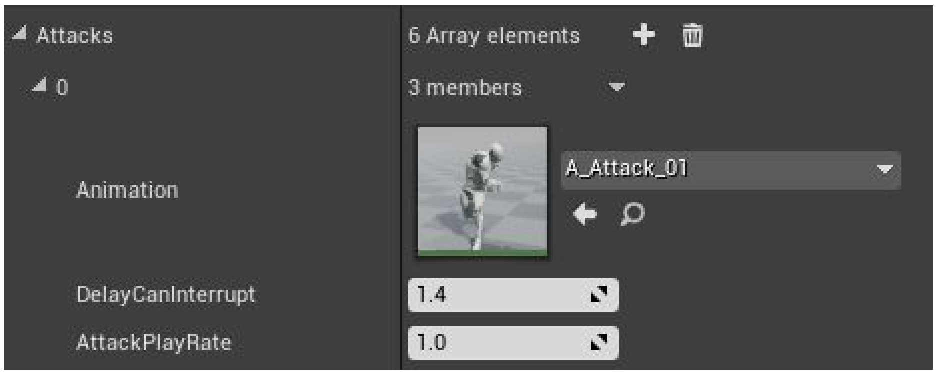
By default, there is a weapon “PlayerSword” in the list. To add a new weapon, add the weapon-structure in the map-array (button with the plus icon). Specify the name of your new weapon. Pay attention that weapons cannot have identical names (map-variable).
Let’s consider weapon parameters:



Weapons

Weapon List	2 Map elements
PlayerSword	
MyNewWeapon	
WeaponName	MyNewWeapon
Can_be_equipped	✓
WeaponMesh	SM_FuturisticSword
SocketEffect	SocketTrail_Start
Damage	40.0
AdditionalDamage	0.0
Attacks	6 Array elements
HeavyAttacks	1 Array elements
AnimationBlueprint	Swordsman_AnimBP
MovementAnimation	SwordsmanPlayer_IdleRun_1D
Available Weapons	1 Map elements

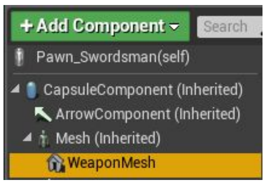
WeaponName is the name of a weapon. You must specify the same name as in the array. In this example, it is MyNewWeapon.
Can be equipped Whether the current weapon can be put on the back (update 1 - https://youtu.be/mzYNtDsmEOg). The variable is needed for a weapon which does not have a 3d model. For instance, hand-to-hand combat.
Weapon mesh is how the weapon looks like, choose its 3D mesh.
Socket Effect is a point from where the particles will be played (sparks)
Damage is a base damage value for the weapon's strike.
AdditionalDamage is an additional damage value. It is better to add additional damage via additional events (described below).
Attacks is a list of attack types with the weapon. By default, they follow one after the other (you may choose a combo).



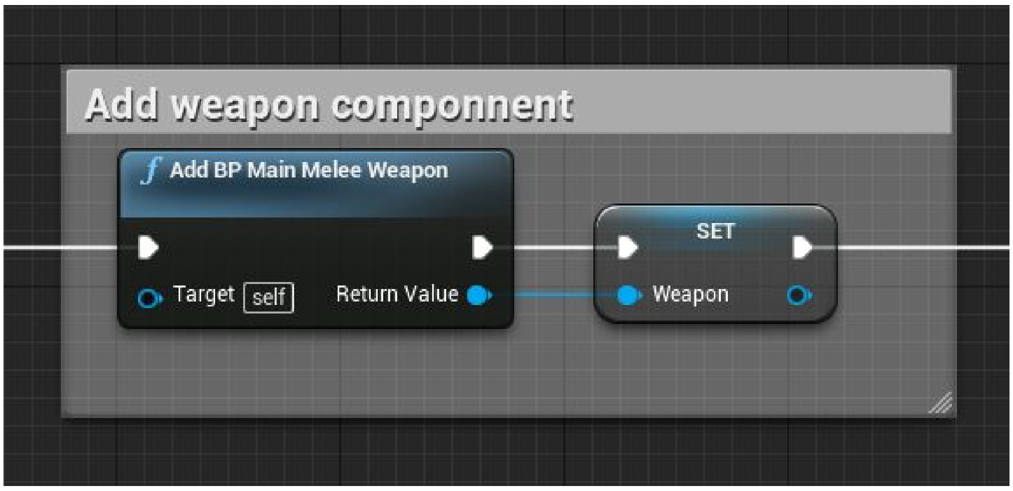
Attacks	6 Array elements
0	3 members
Animation	A_Attack_01
DelayCanInterrupt	1.4
AttackPlayRate	1.0

Animation is an animation of the strike.
Delay Can Unterrupt is a time-delay after which the animation of the strike can be interrupted with an animation for movement.
Attack PlayRate is the initial animation playrate for the strike (the speed of the attack).

The further step is to add the weapon to our character. For that, create a mesh and attach it to the character's weapon socket (Socket_Weapon). This component is present there by default.

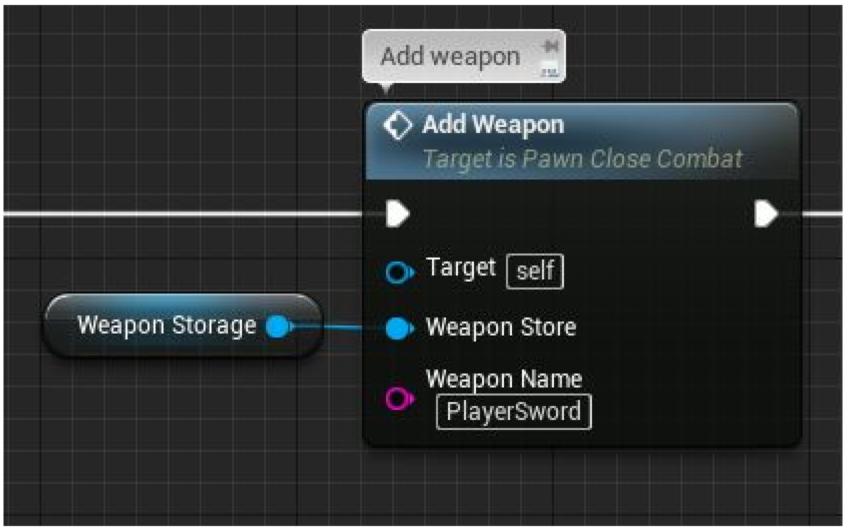


Then add the weapon component. By default, it is added with the event Begin Play.



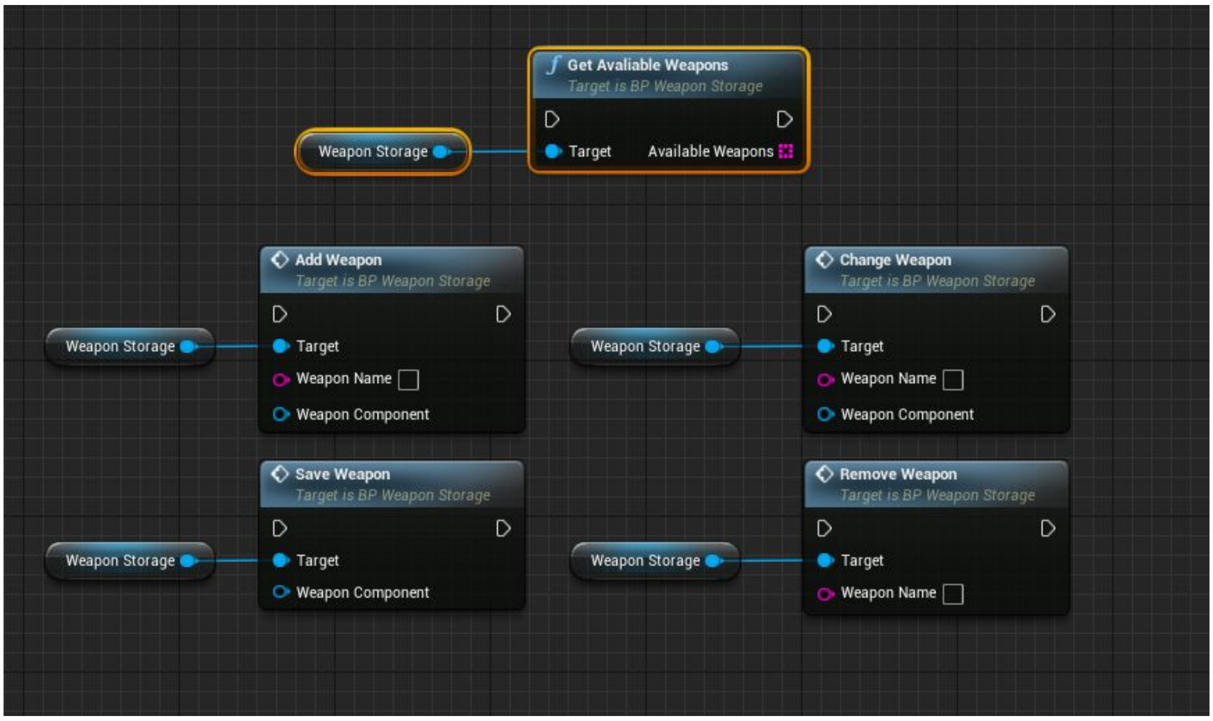
When adding the component, specify the variable (variable type **BP_MainMeleeWeapon**) for your weapon, so that later we can reference this weapon of ours from any place. The component contains the logic of close quarters combat. The component "WeaponStorage" stores parameters for the weapon component, so that a weapon can be modified when rewriting these parameters.

In the next step, specify for the weapon component what weapon to use. In order to do it, call function "Add Weapon" in the Pawn and write the name of your weapon. We have discussed the creation of it above.



By default, it is "PlayerSword". The function writes parameters from the list "WeaponList" in the weapon component.

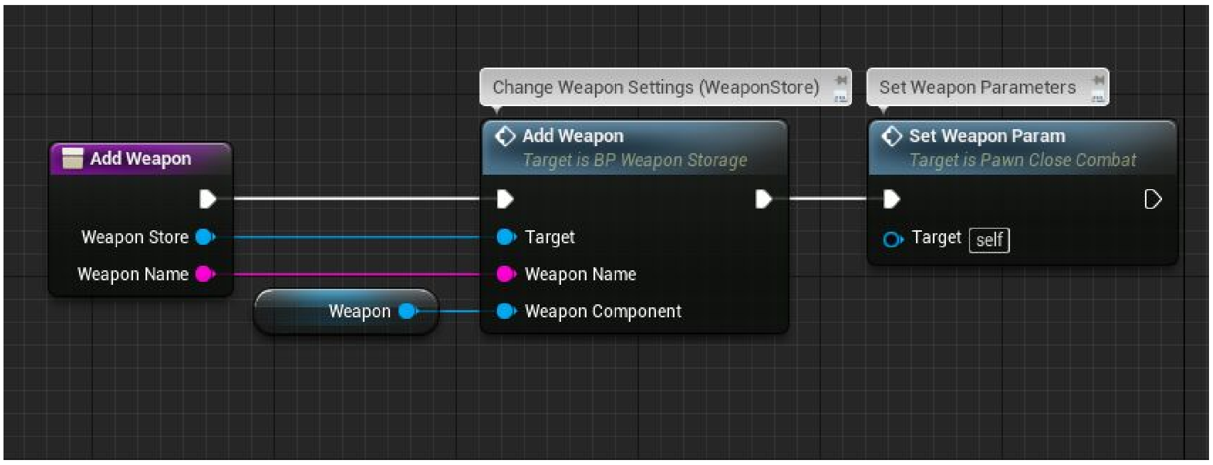
Let's consider all events when working with "WeaponStorage":



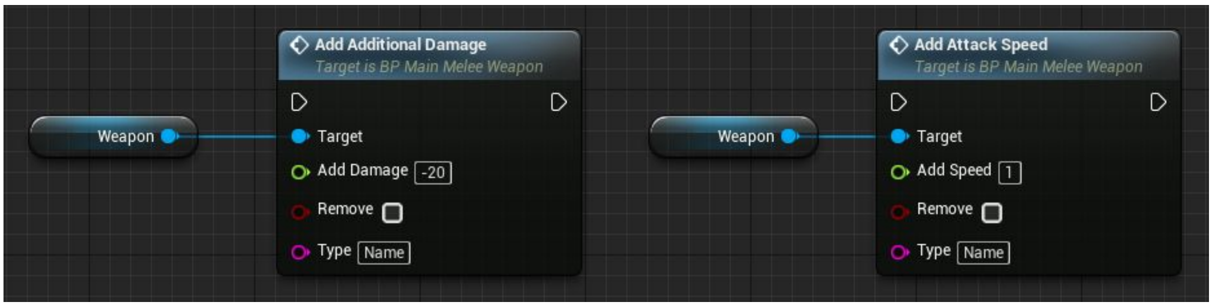
Save Weapon. It saves the current weapon by adding it to the list "AvailableWeapons".

Change Weapon. It replaces the current weapon with the specified one from the list of available weapons (“AvailableWeapons”).
Remove Weapon. It removes the specified weapon from the list of available weapons (“AvailableWeapons”).
Get Available Weapons. It returns a list of weapons available to the player.

Functions “AddWeapon” and “ChangeWeapon” in the component “WeaponStorage” write new parameters in the weapon component. However, in order to apply the new parameters to the character, you need to call the function “SetWeaponParam” right after the parameters have been changed. For convenience, functions “Addweapon” and “ChangeWeapon” have been created in the Pawn. There, right after the parameters have been applied, the function “SetWeaponParam” is called.



Additional events in the weapon:

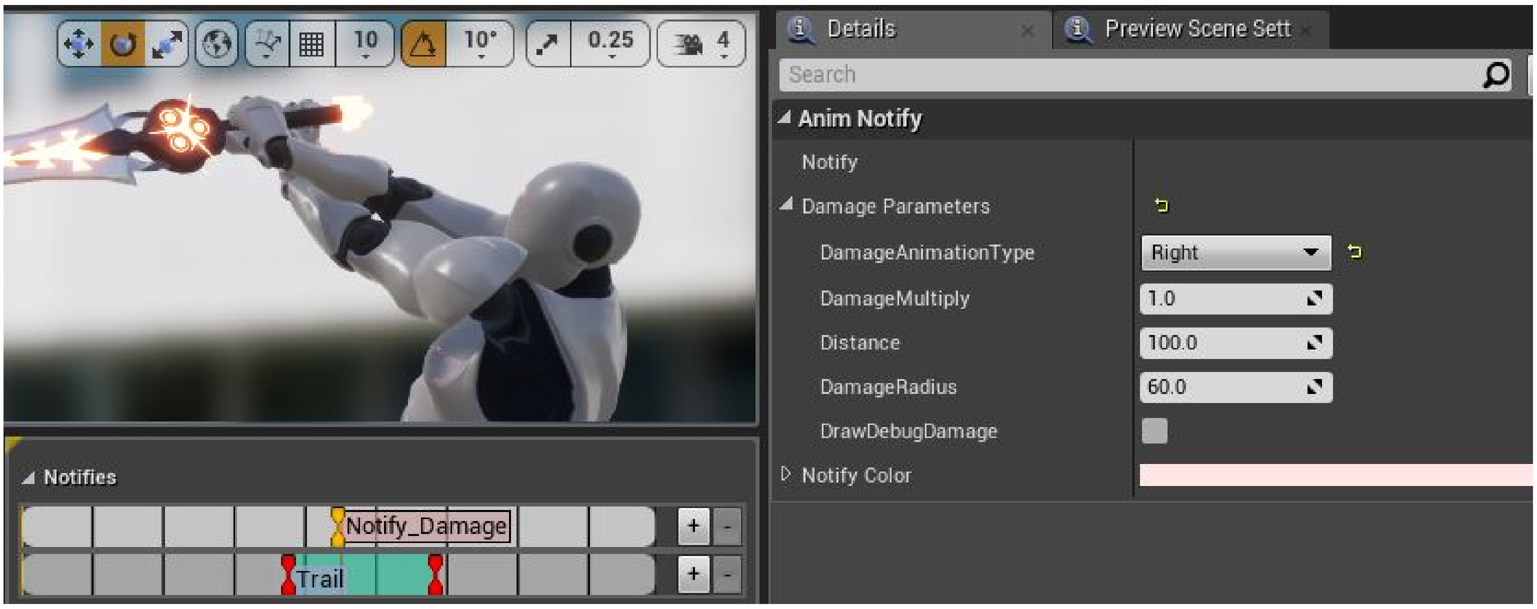


Add Additional Damage is an event for an increase or a decrease of extra damage dealt by the weapon. For instance, due to the character’s level up or various skills. Function “MultiIncrease” is used.
Add Attack Speed is an event for an increase or a decrease of the attacking speed. For instance, due to the character’s level up or various skills. Function “MultiIncrease” is used.

Inflicting damage (“Notify_Damage”)

Inflicting damage on actors is done via the event “**CauseDamage**” in weapon class, but the information about the damage is passed through the animation via Notify.

For instance, open any animation of attack and you will notice “**Notify_Damage”** along the notify timeline.



Anim Notify

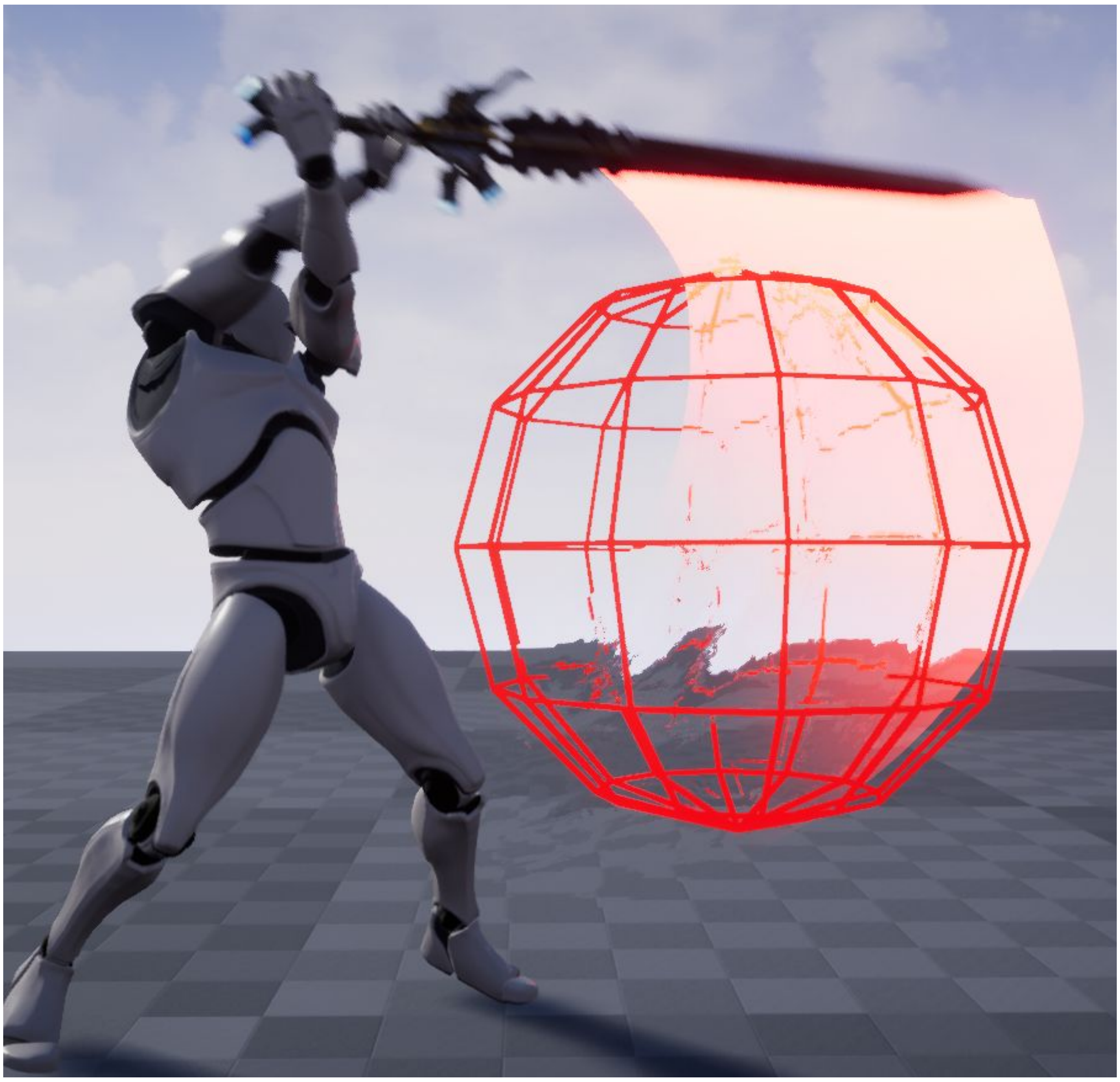
Notify	
Damage Parameters	
DamageAnimationType	Right
DamageMultiply	1.0
Distance	100.0
DamageRadius	60.0
DrawDebugDamage	☐
Notify Color	

Notify_Damage must be in the frame of damage infliction. To position the notify, click the RIGHT MOUSE BUTTON on needed frame and choose **Notify_Damage**. Select the notify and you will see its settings in tab Details. Open the list of parameters “Damage Parameters”:

Damage Animation Type is an animation type of the damage experienced by the target (to the left, to the right, or backwards). The animation is either selected manually in ***Enemy_Param*** in **AI** settings, or by filling the table.
DamageMultiply is a damage multiplier from the current strike.

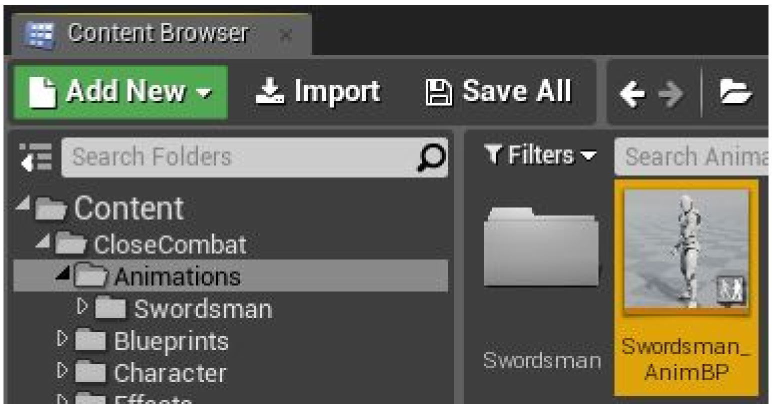
Damage to targets is dealt according to the following principle. A sphere of a given radius is created near the attacking actor. If the sphere intersects actors with tags target (enemy groups), then they receive damage. You may switch off the check of targets and the damage will be applied to all actors.

Distance is a distance from the character to the center of the sphere.
DamageRadius is a radius of the sphere, which intersects the actors.
DrawDebugDamage is a visual representation of the sphere for the convenience of tweaking.

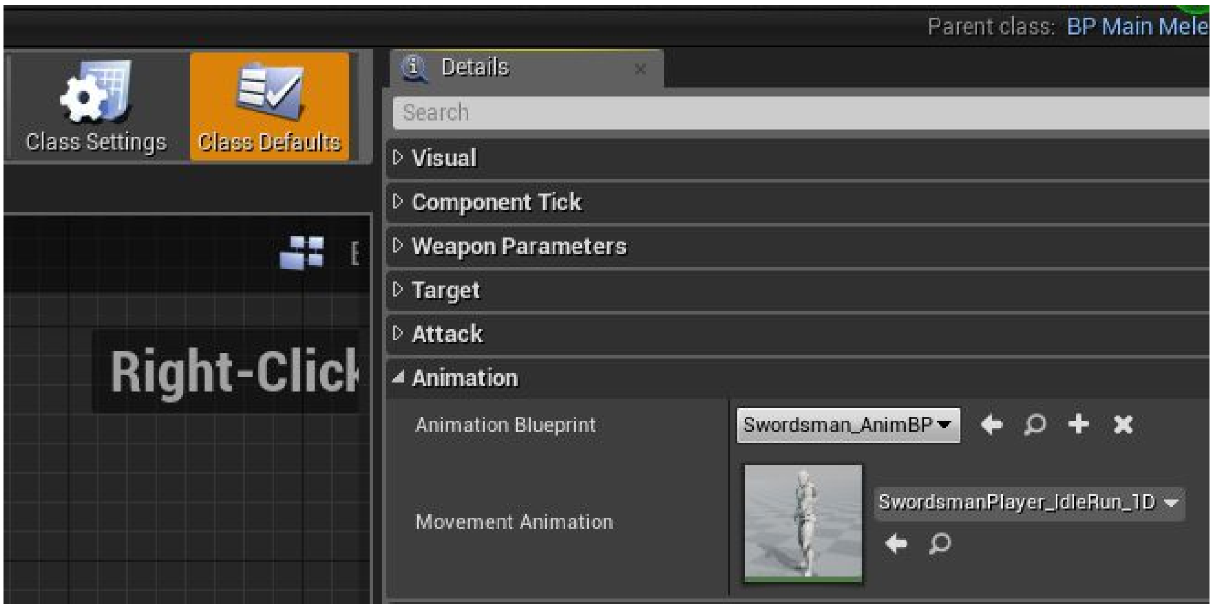


Animation of a character with a weapon.

All animation of a character (movement, behavior in a combat, animation of skills...) are in Anim Blueprint.



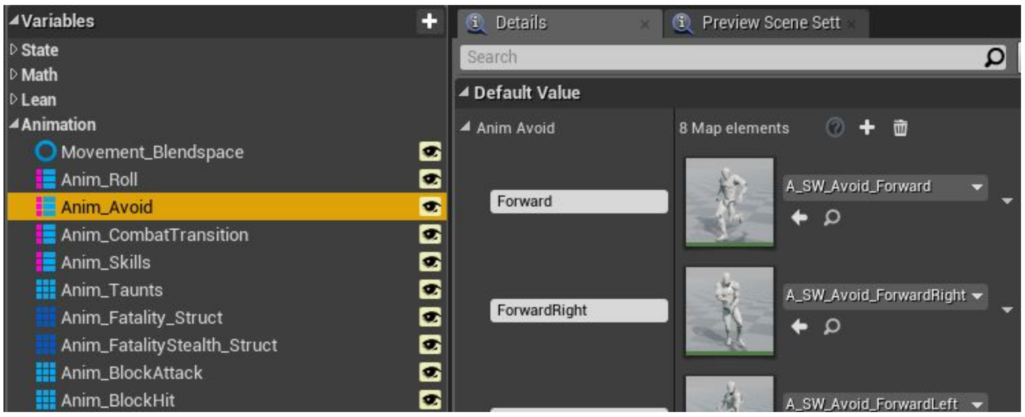
The structure is done in such a way, that the weapon class has a reference to the Anim Blueprint and changes all animation of the character. It is quite convenient when adding new weapon types. For instance, you are planning to make a glaive. For that, simply copy the Anim Blueprint and specify new animations there. Followed by that, specify which Anim Blueprint to use in weapon class.



Animation Blueprint: Swordsman_AnimBP
Movement Animation: SwordsmanPlayer_IdleRun_1D

Animation Blueprint is an animation blueprint, which will be used when the current weapon is added.
MovementAnimation is a choice of a blend space for movement. Enemies may have their own animations for movement, but same combat animations.

All additional animations are listed in category "Animation"



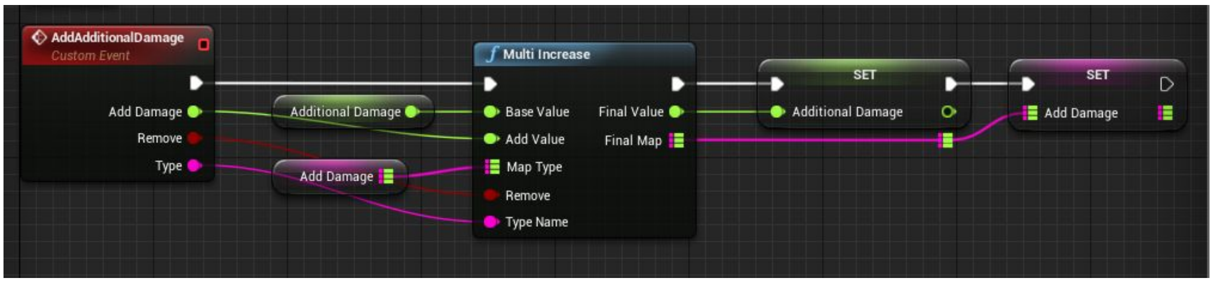
Anim Avoid: 8 Map elements
Forward: A_SW_Avoid_Forward
ForwardRight: A_SW_Avoid_ForwardRight
A_SW_Avoid_ForwardLeft

Library of functions "CloseCombat_FunctionLibrary"

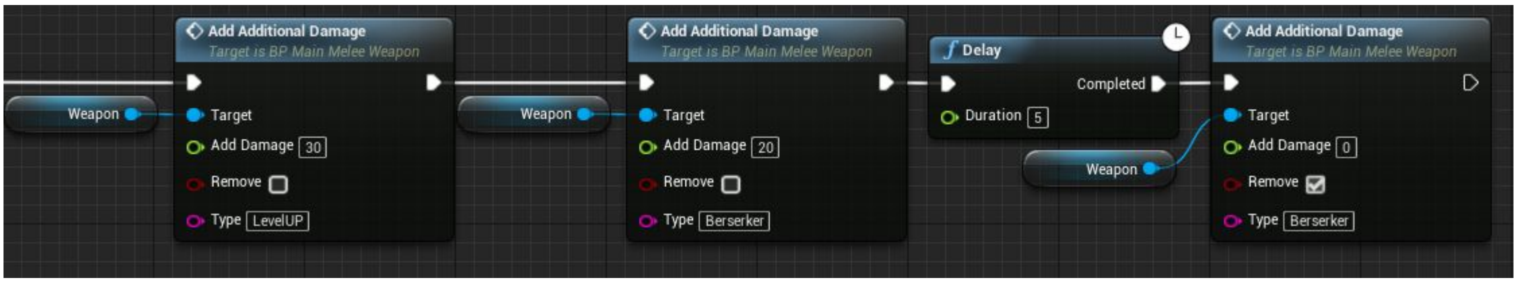
This library contains functions that might be useful outside of a single actor.

Function "MultiIncrease"

One of such function is ***MultiIncrease***. The function is for changing values from different inputs. For instance, the damage caused by a weapon can get increased due to a skill and because of a level-up. However, since the increase due to a skill can be temporal, this is why you need the help of ***MultiIncrease*** function.



The function works in the following way. There is an input value (Base Value), to which another value (Add Value) needs to be added or subtracted. Then the current value (Add value) is written in map variable under some key (name). A Boolean variable Remove is responsible for adding changes or removing changes made. In practice, it looks like that:



In the beginning, the weapon damage is increased by 30 points type of “LevelUp”, then extra 20 points of damage type of “Berserker” are added, and, after 5 seconds, from the total value of damage a value type “Berserker” is subtracted and it amounts to 20 points. If Boolean variable Remove=true, there is no need to specify the value, since it is already in the database.
For the sake of convenience, in all components where function “MultiIncrease” is used, events are implemented. The names of the events start with the word “Add”.

Function “TargetTags”

The function is for adding or removing tags in an actor. It works with arrays, so that multiple tags can be added or removed at the same time.

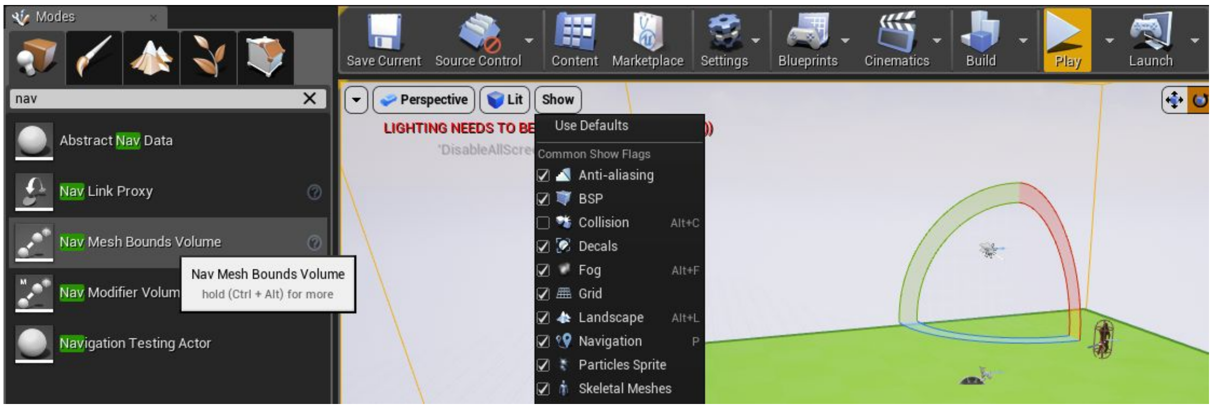
Function “CheckTargetTag”

This function checks whether specific tags are assigned to a specific actor, and, if it is so, returns a Boolean value.

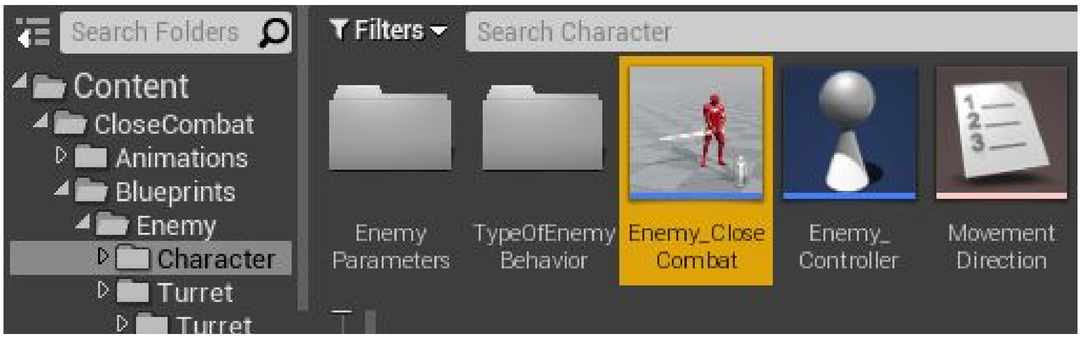
Artificial Intelligence (enemies)

In the project, there are enemies capable of patrolling surroundings. If they detect a target, they can attack it in close-quarter combat. For the start, let’s consider adding an enemy to the scene. There are several ways to do it. We will discuss the method of direct insertion of an enemy into the scene and the method of spawning of an enemy via Blueprint.

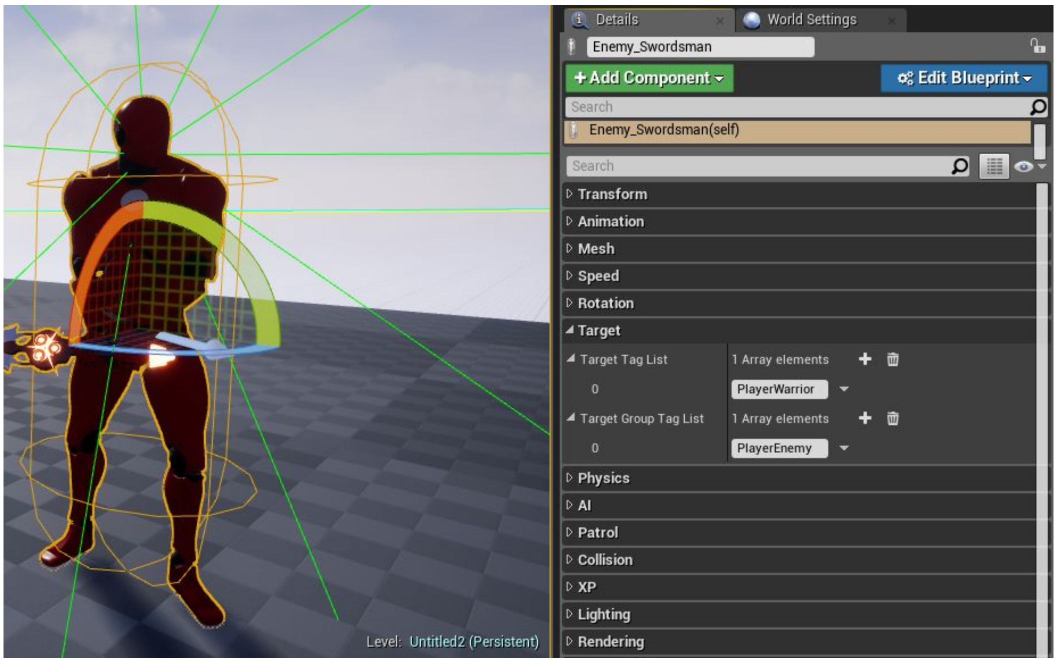
The first thing is to determine the navigation inside your level. This is done by adding volume “*Nav mesh Bounds Volume*”, which should comprise your scene where potential enemies can move.



The next step is to place your enemy via “click, drag and drop” from the content browser.

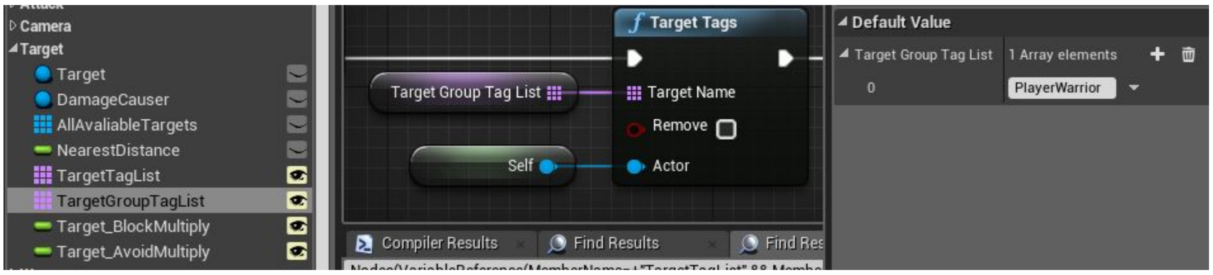


Select the actor and go to its settings.

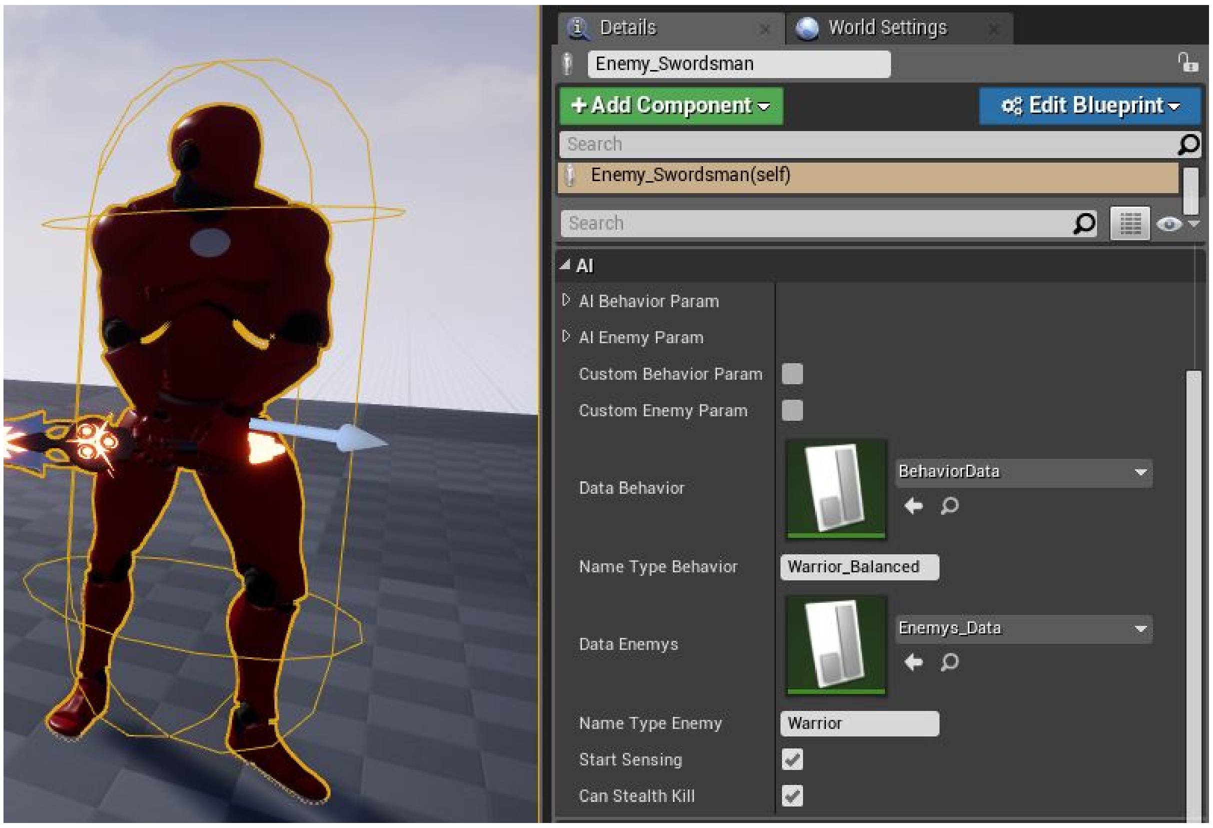


Target Tag List: 1 Array elements
0: PlayerWarrior
Target Group Tag List: 1 Array elements
0: PlayerEnemy

Specify the target for the character and to which group it belongs first. The default target for the character is the group “**PlayerWarrior**”, which is the default group of the player’s character. The character belongs to the group “**PlayerEnemy**” by default, which is a group hostile to the player. It is very important to take into account the information of the group, since player’s targeting is done based on the name tags of the actor. The tags are assigned at the event Begin Play via described above function “Target Tags”.



It is possible to assign multiple groups at the same time, in other words, you can make any number of enemy or allied groups.
The next step is "AI" settings.

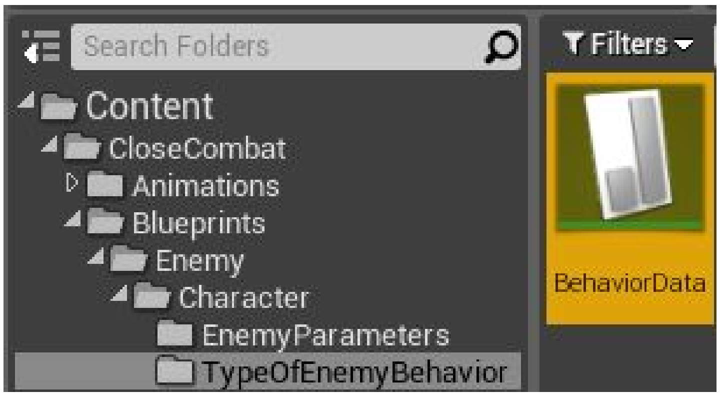


AI Behavior Param	
AI Enemy Param	
Custom Behavior Param	☐
Custom Enemy Param	☐
Data Behavior	BehaviorData
Name Type Behavior	Warrior_Balanced
Data Enemys	Enemys_Data
Name Type Enemy	Warrior
Start Sensing	✓
Can Stealth Kill	✓

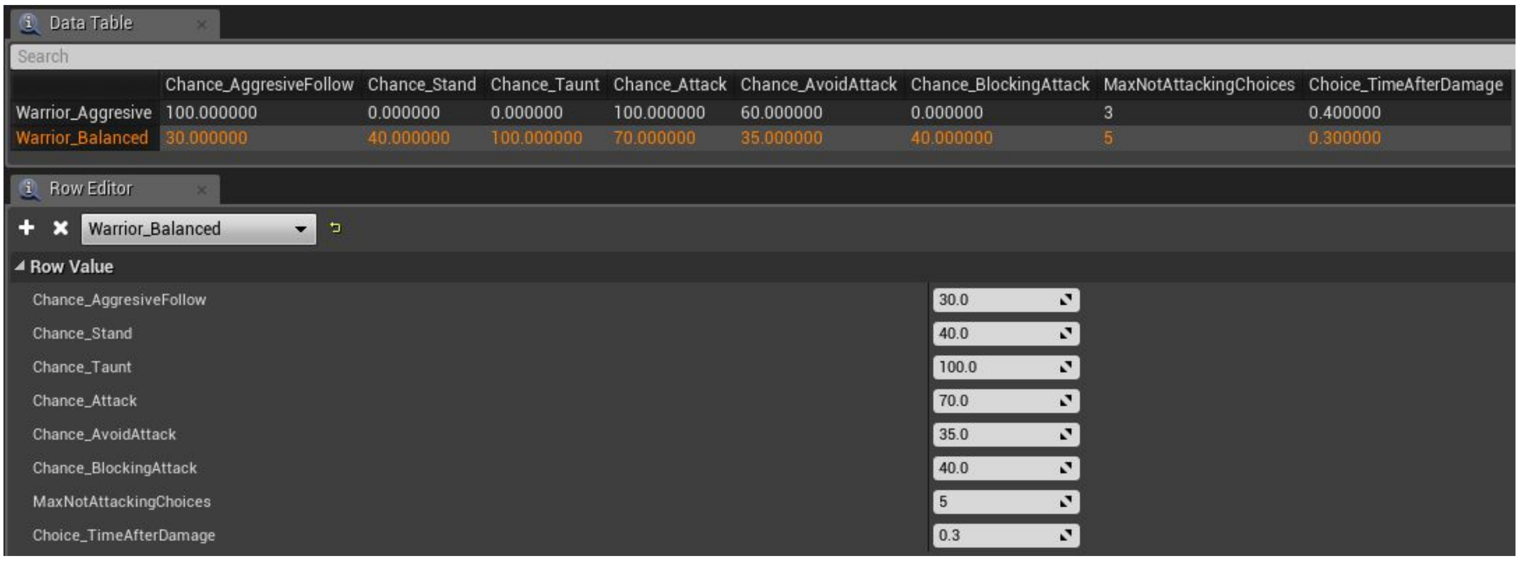
In this section, there are main settings of behavior in a combat, as well as parameters of the character such as health, amount of damage that goes through blocking, amount of damage that is applied when dodging.
Start Sensing. Whether the character is able to see and hear from the start.
Can Stealth Kill . Whether the character can be killed with the skill "StealthKill".
Custom Behavior Param. Whether to ignore the table and specify all parameters manually. Upon enabling this parameter, open the list *AI Behavior Param* and specify parameters of character's behavior.
Custom Enemy Param. Whether to ignore the table and specify all parameters manually. Upon enabling this parameter, open the list *AI Enemy Param* and specify parameters of character's behavior.

Data tables are used in the project for the convenience of filling in and storing data regarding characters behavior in combat.
When a character finds a target and stays within the combat range (6 meters by default), it makes decisions every 0.3 second at percentage rate (0-100) compared to the specified

parameters in structure **AI_BehaviorParam**. The structure is assigned data from the table or from manually specified values (based on settings presented above).
Let's open the table and discuss settings for behavior in a combat.



There are 2 types of pre-made behavior in the project by default.
Warrior_Balanced is a character behavior balanced in attack and defense.
Warrior_Aggresive is an aggressive behavior set at constant attack.
You may have any amount of pre-made tables, which you may either specify beforehand or modify dynamically with the help of events in character. Fro that, use **ChangeBehavior** for behavior modification and **ChangeEnemyParam** for modification of parameters.



	Chance_AggresiveFollow	Chance_Stand	Chance_Taunt	Chance_Attack	Chance_AvoidAttack	Chance_BlockingAttack	MaxNotAttackingChoices	Choice_TimeAfterDamage
Warrior_Aggresive	100.000000	0.000000	0.000000	100.000000	60.000000	0.000000	3	0.400000
Warrior_Balanced	30.000000	40.000000	100.000000	70.000000	35.000000	40.000000	5	0.300000

Warrior_Balanced

Chance_AggresiveFollow	30.0
Chance_Stand	40.0
Chance_Taunt	100.0
Chance_Attack	70.0
Chance_AvoidAttack	35.0
Chance_BlockingAttack	40.0
MaxNotAttackingChoices	5
Choice_TimeAfterDamage	0.3

Description of behavior parameters:

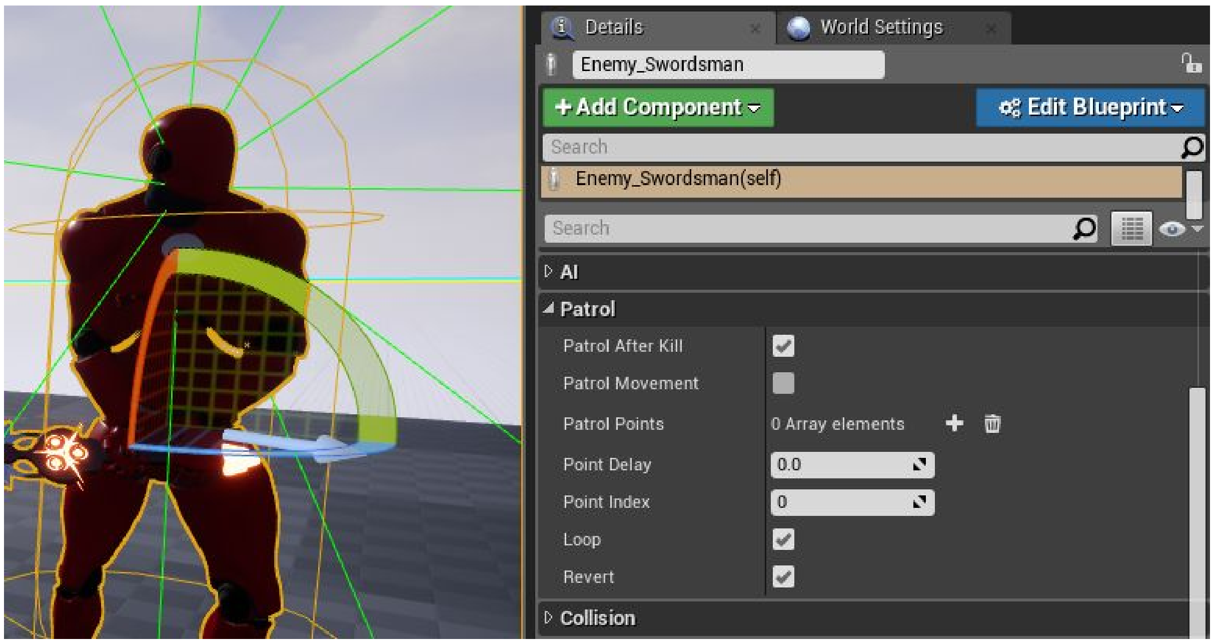
All parameters are in percentages (0-100%). For example, the first parameter from the top is the parameter of Chance_AggressiveFollow. It is equal to 30%, which means that the chance to make a decision to get closer to the target is 30 out of 100, and the decision-making is every 0.3 second.
Chance_AggresiveFollow is a chance to make an aggressive step towards the target followed by an attack.
Chance_Stand is a chance to stop.
Chance_Taunt is a chance to taunt the target after the stop.
Chance_Attack is a chance to initiate the attack if the target is within 3 meters. In case you do not get the chance, the character does a quick dash.

Chance_AvoidAttack is a chance to dodge an attack by rolling.
Chance_BlockingAttack is a chance to use a block.
MaxNotAttackingChoices is the maximum amount of decision-making not related to attacking. After decision-making not related to attacking is over, the character starts getting closer to the target and will attempt an attack. Once the attack is done, the decision-making not related to attacking starts all over.
Choice_TimeAfterDamage is a time interval. If a character takes damage, then the time interval determines when the character is able to make decisions again.

The character makes decisions in a combat based on these parameters. You may refer to comments in blueprint for a more detailed breakdown. Every part of the source code has detailed commentary.

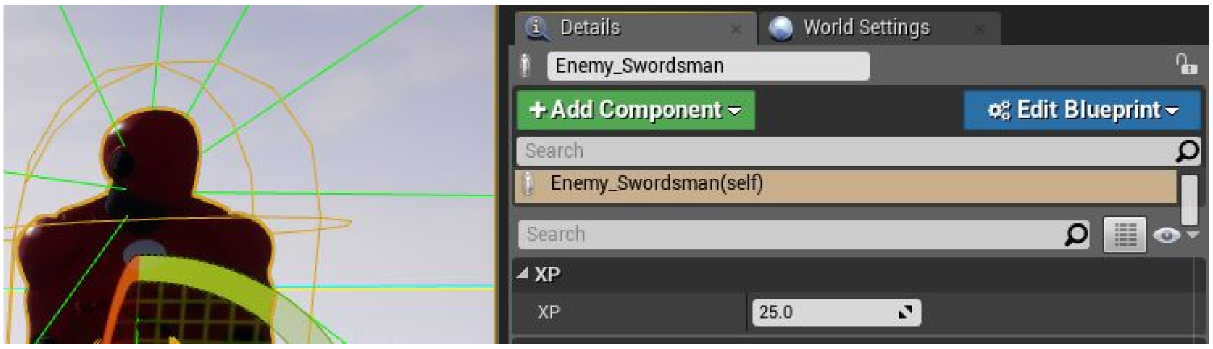
Patrolling

The character can patrol different area of a game level. The patrolling is based on specified key points. All parameters of patrolling are specified in a corresponding tab in character's details.



Patrol After Kill	✓
Patrol Movement	
Patrol Points	0 Array elements
Point Delay	0.0
Point Index	0
Loop	✓
Revert	✓

Patrol After Kill. Whether the character does or does not continue patrolling after killing the target.
Patrol Movement. Whether the character will do patrolling.
Patrol Points are key points for patrolling. They contain coordinates. Specify current character's location in the first point. There can be an arbitrary amount of points.
Point Delay determines the amount of time the character can stay in key points.
Point index is the current point where the character is going to (it is an index of array of key points).
Loop. Whether the character returns in the first key point and goes on patrolling upon reaching the last point.
Revert. Whether the character visits previous points on his way to the first key point.
The parameter XP contains the amount of experience awarded for killing this character.


Details | World Settings

Enemy_Swordsman

+ Add Component | Edit Blueprint

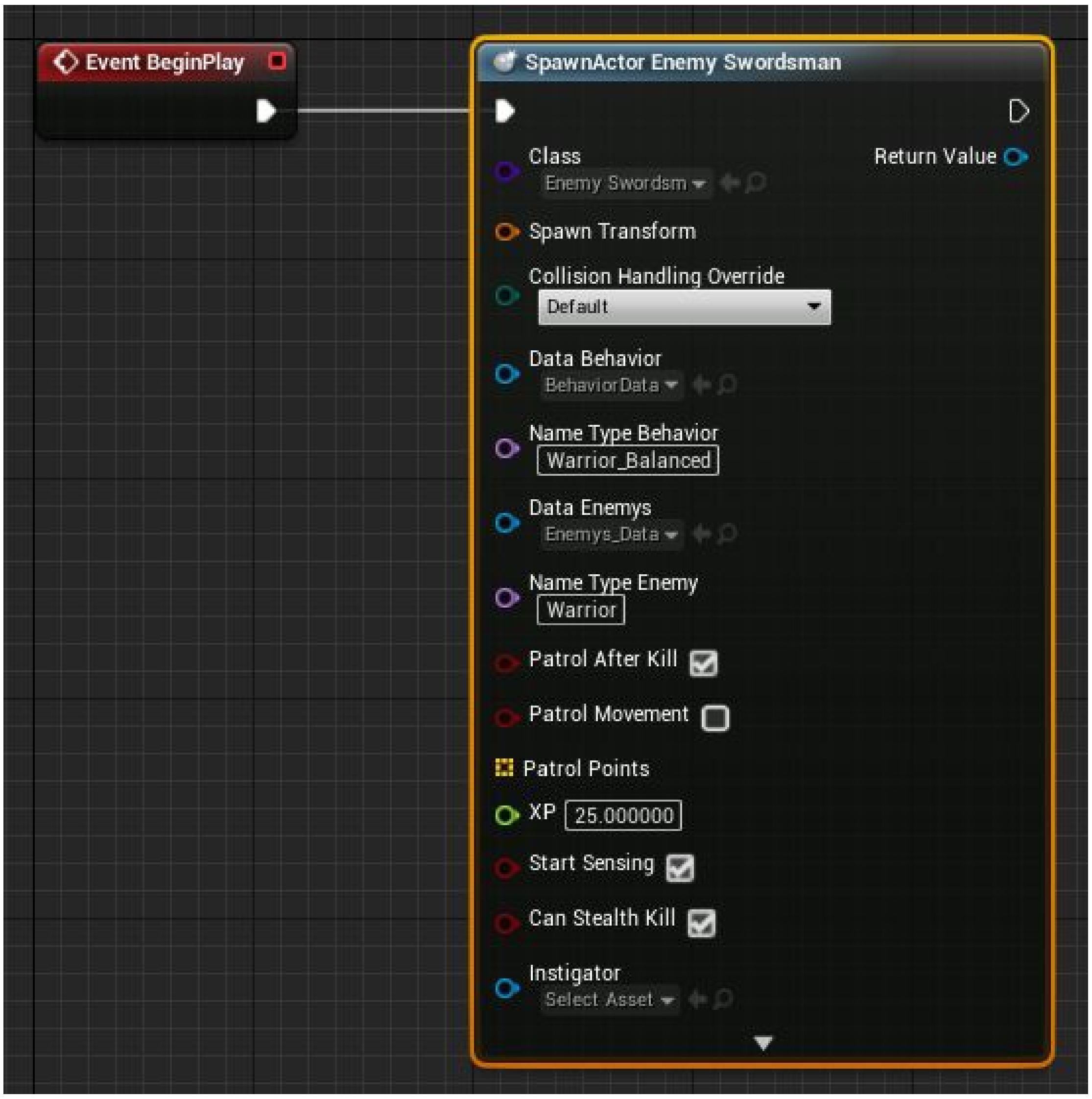
Enemy_Swordsman(self)

XP

XP: 25.0

Spawn of enemies via blueprint.

Simply use node “**Spawn actor from class**”. When choosing class “**Enemy_Swordsman**”, all settings described above will appear. Make sure to specify **Spawn Transform**, which is a location in the scene where the enemy appears.


Event BeginPlay

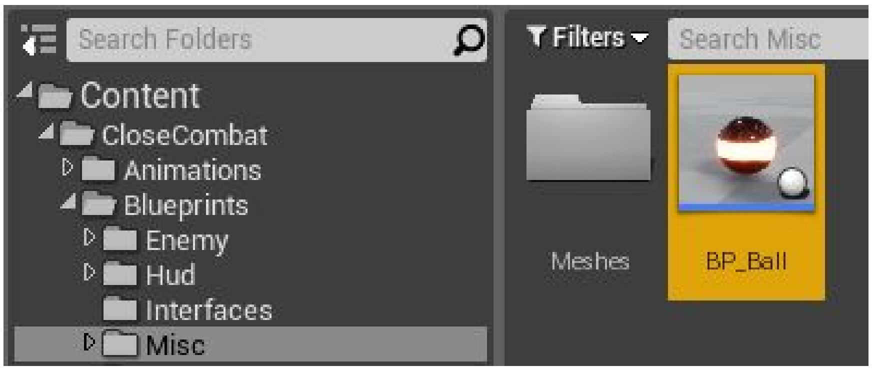
SpawnActor Enemy Swordsman

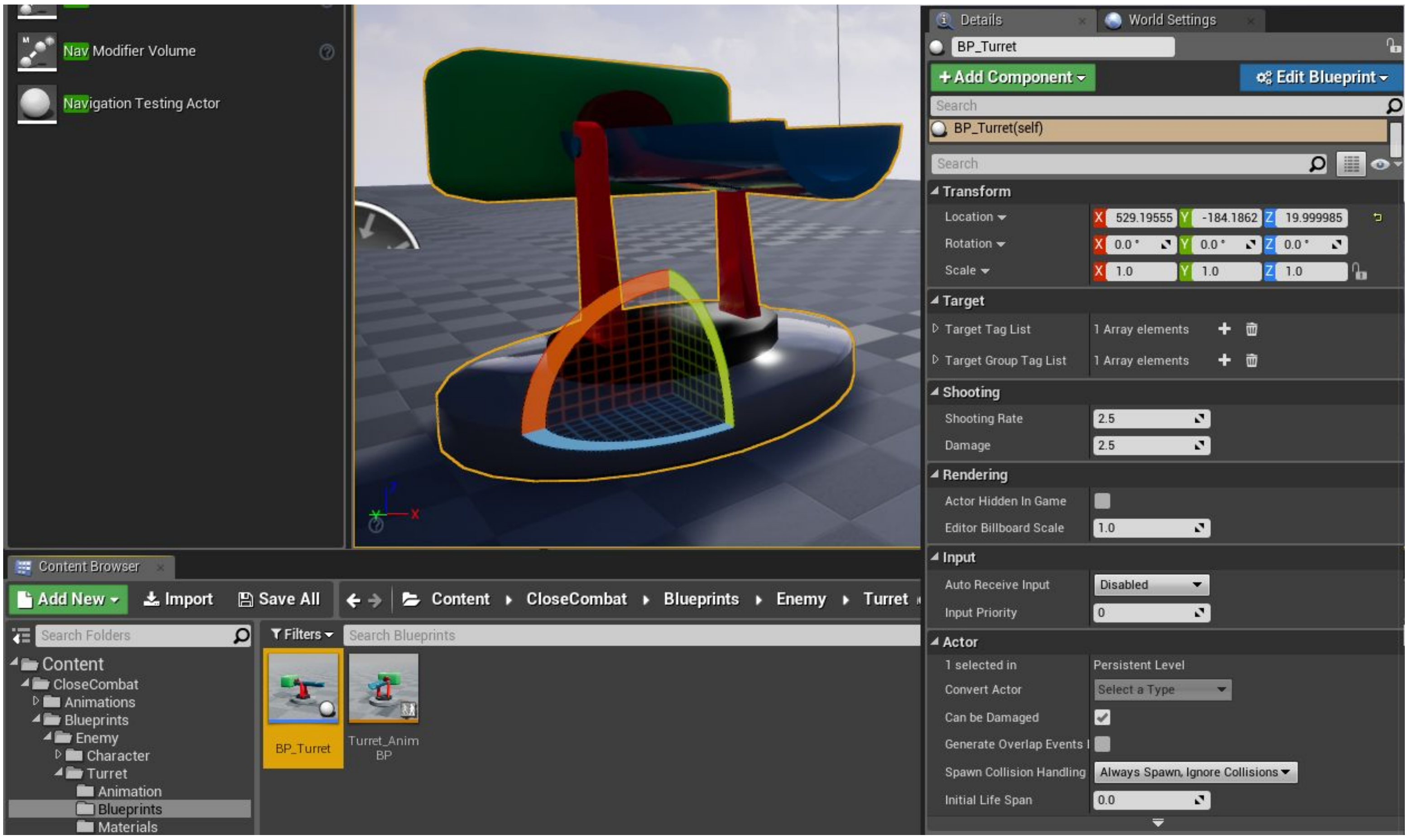
Class: Enemy Swordsm
Return Value
Spawn Transform
Collision Handling Override: Default
Data Behavior: BehaviorData
Name Type Behavior: Warrior_Balanced
Data Enemys: Enemys_Data
Name Type Enemy: Warrior
Patrol After Kill: ✓
Patrol Movement: ☐
Patrol Points
XP: 25.000000
Start Sensing: ✓
Can Stealth Kill: ✓
Instigator: Select Asset

This is how you can adjust parameters of attack in tab “**Attack**” such as:
AttackDelayMin is the minimum time delay between attacks.
AttackDelayMax is the maximum time delay between attacks.
AttackTimeRotation is a speed of rotation towards the target during an attack.
AttackTimeLookAtTarget is a time interval after which a character making turn, stops turning towards the target. It might be useful when implementing sneaking behind the back logic.

Turret

Everything described above is applied to making a turret for a game level. The turret shoots with balls with parameters of a projectile.





Transform

Location	X 529.19555	Y -184.1862	Z 19.999985
Rotation	X 0.0 °	Y 0.0 °	Z 0.0 °
Scale	X 1.0	Y 1.0	Z 1.0

Target

Target Tag List	1 Array elements
Target Group Tag List	1 Array elements

Shooting

Shooting Rate	2.5
Damage	2.5

Rendering

Actor Hidden In Game	☐
Editor Billboard Scale	1.0

Input

Auto Receive Input	Disabled
Input Priority	0

Actor

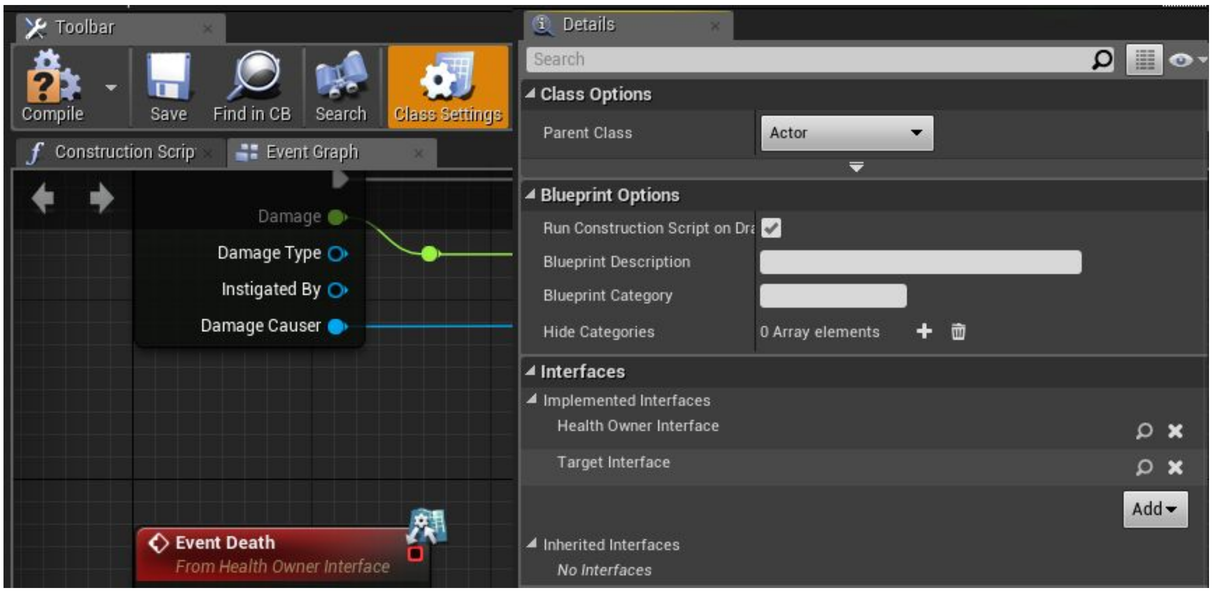
1 selected in	Persistent Level
Convert Actor	Select a Type
Can be Damaged	✓
Generate Overlap Events I	☐
Spawn Collision Handling	Always Spawn, Ignore Collisions
Initial Life Span	0.0

Parameters **Target:**
Target Tag List is a list of names of target groups for the turret.
Target Group Tag List is a list of group names the turret is related to.

Parameters **Shooting:**
Shooting Rate is the shooting rate.
Damage is the damage value caused by each shot.

Making of a new enemy.

Let’s consider an example of making a new enemy for close quarter combat. Take the actor Cube, which we have made in section “***How to use health component***”, or follow the instructions in that section and make it, if you have not made it yet.
The interaction between targets also happens with the help of interfaces. This is why the first thing is to attach the interface “**Target_Interface**” to our new enemy.

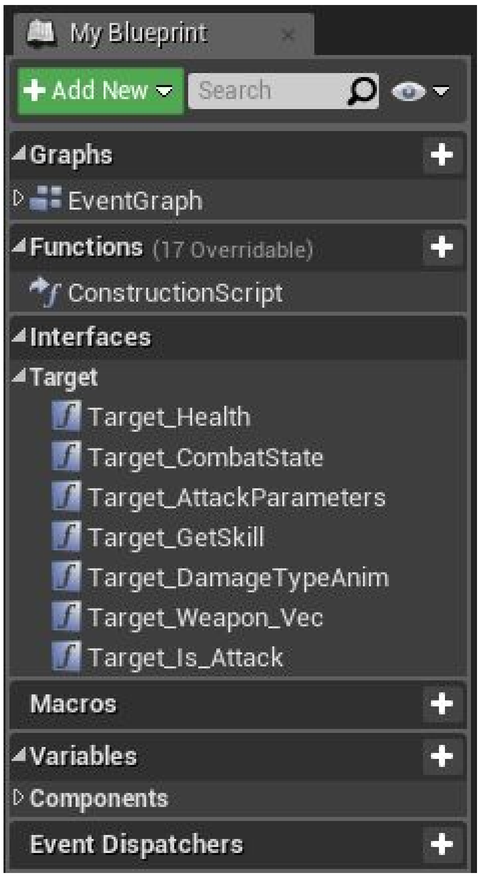


Class Options
Parent Class: Actor

Blueprint Options
Run Construction Script on Dr: ✓
Blueprint Description
Blueprint Category
Hide Categories: 0 Array elements

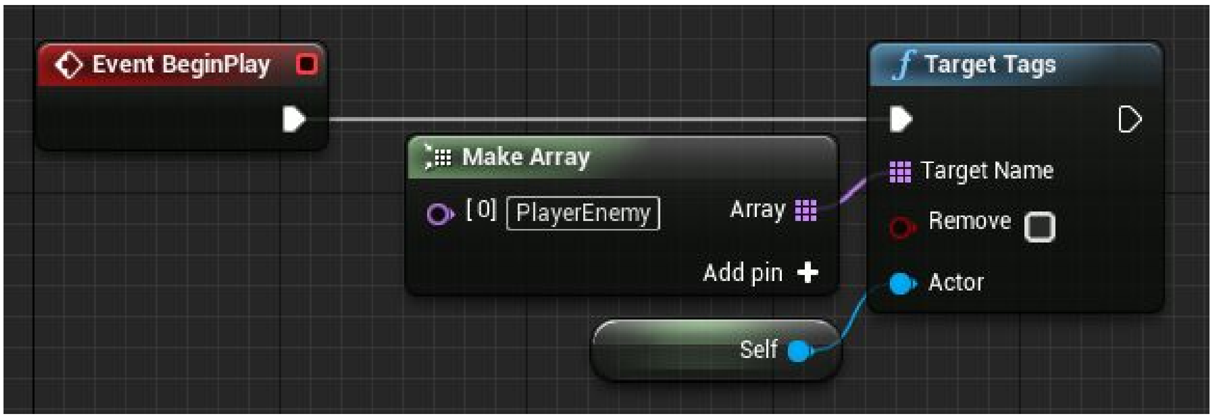
Interfaces
Implemented Interfaces
Health Owner Interface
Target Interface
Add
Inherited Interfaces
No Interfaces

After adding the interface, a new category “Target” will appear in tab interfaces.

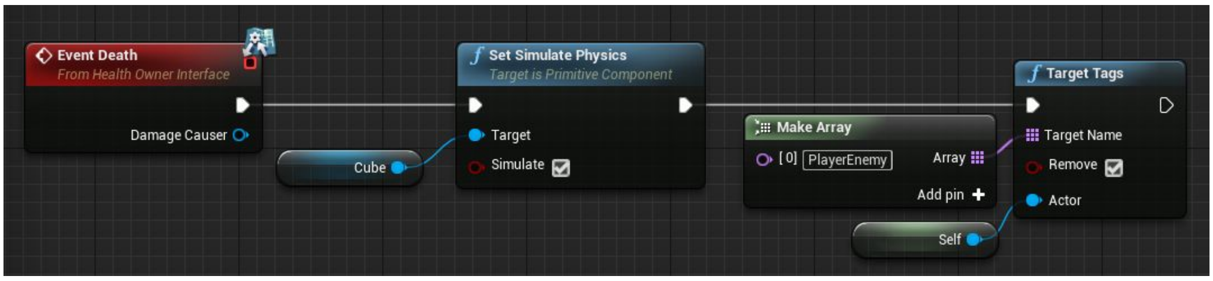


Graphs
EventGraph
Functions (17 Overridable)
ConstructionScript
Interfaces
Target
Target_Health
Target_CombatState
Target_AttackParameters
Target_GetSkill
Target_DamageTypeAnim
Target_Weapon_Vec
Target_Is_Attack
Macros
Variables
Components
Event Dispatchers

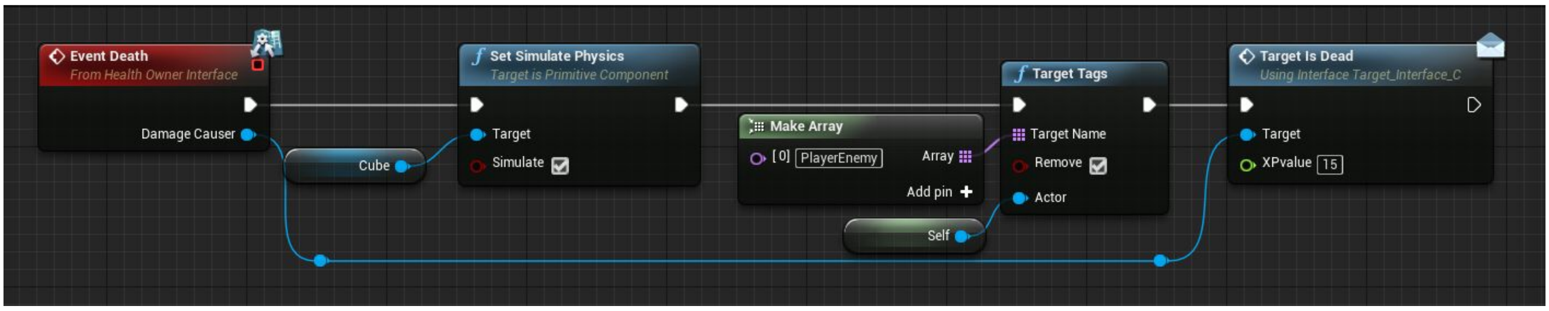
Now specify to our actor the group it belongs to. For that use function "**TargetTags**" and specify the group PlayerEnemy (it is a hostile group to the player by default).



The next step is to pass parameters of health via interface, so that the attacker knows the value of health of the target. Open function "Target_Health" in tab interface "Target" and specify Parameters of our health component. Now the player can do targeting and see the target's health.
Further on, it is necessary to clean names of groups when the actor dies, so that after the death of the target, the player will not be able to target it anymore. For that, copy nodes for adding a group, but specify parameter Remove.



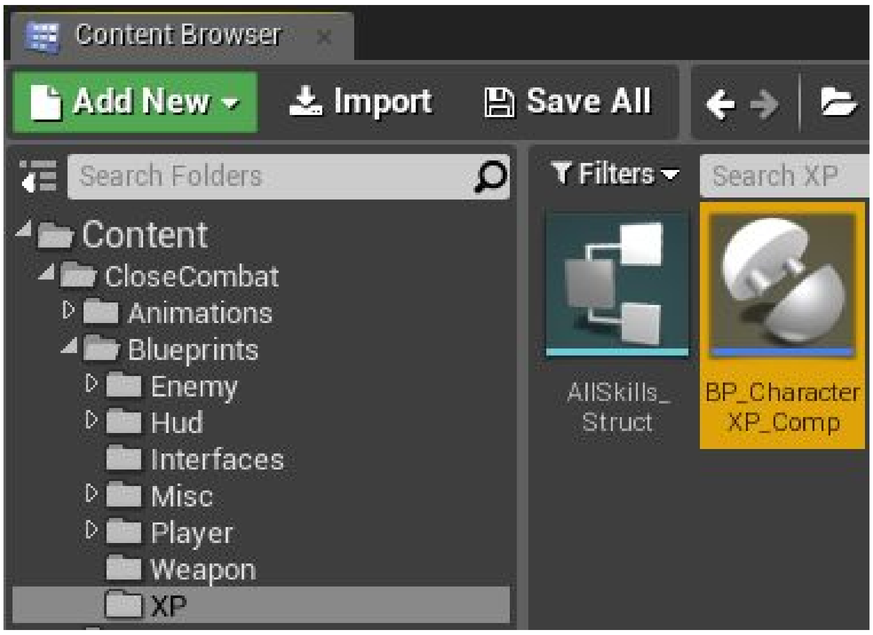
The next step is to pass information about the death of the actor. To do so, simply pass interface "Target is dead" to the attacker at the Event Death.



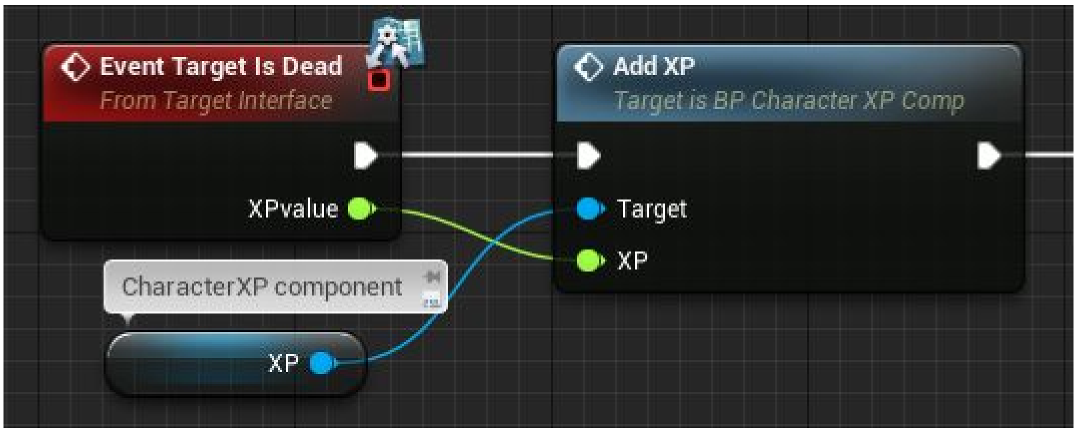
Also, you can specify experience received for killing current actor. That is all, our simple actor is ready. Now characters can target him, kill him and get experience for doing it. For more complex and advanced enemies, refer to commentaries within the code of enemy actors "**Enemy_Swordsman**" for close quarter combat encounters and "**BP_Turret**" for fighting at long range respectively.

Experience (XP) management component.

The component is for managing experience and leveling-up the character. Add it just like you add health component. The interaction between the component and a character is done via interface “**CharacterXP_Interface**”.

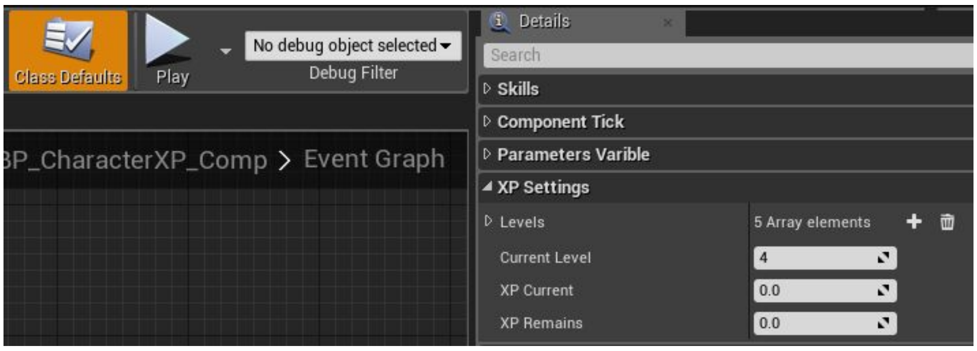


This is how it works. The component has a structure of a character’s level, which is changed according to the index value (level) in the array of all levels after certain amount of experience is collected. By default, experience is added when enemies are killed. This is done with the help of event “Add_XP”.



In the example of the enemy actor just created by us, when the actor dies, it sends a message to the interface of the attacker about the death and experience received. In its turn, the attacker responds to the event by passing information about the experience in component “**CharacterXP_Interface**”. The component calculates the experience. If the experience amount is sufficient to increase character’s level, the component increases the level of the character by updating its parameters from the structure and information is displayed in game interface (HUD).

Let’s consider experience level structure. Open component “**CharacterXP_Interface**” and go to “**Class Defaults**”. We are only interested in tab “**XP Settings**”.

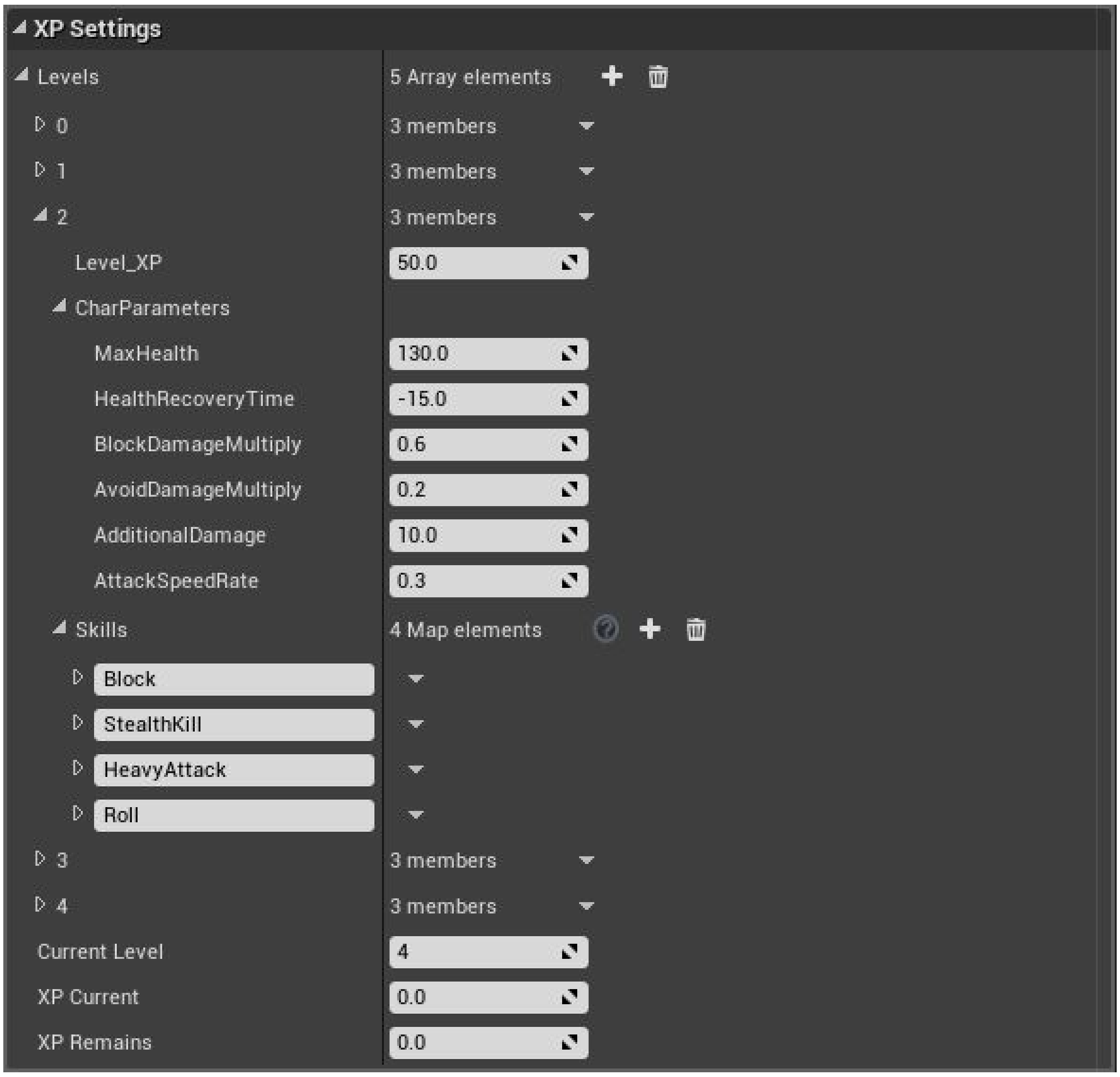


Levels	5 Array elements
Current Level	4
XP Current	0.0
XP Remains	0.0

Levels is an expandable list of structure sets for each level of the character.
Current Level is the current level of the character. Input 1 to begin with the first level.

Leave XP Current and XP Remains as zeroes. These are variable used in calculation of experience.

XP level structure:

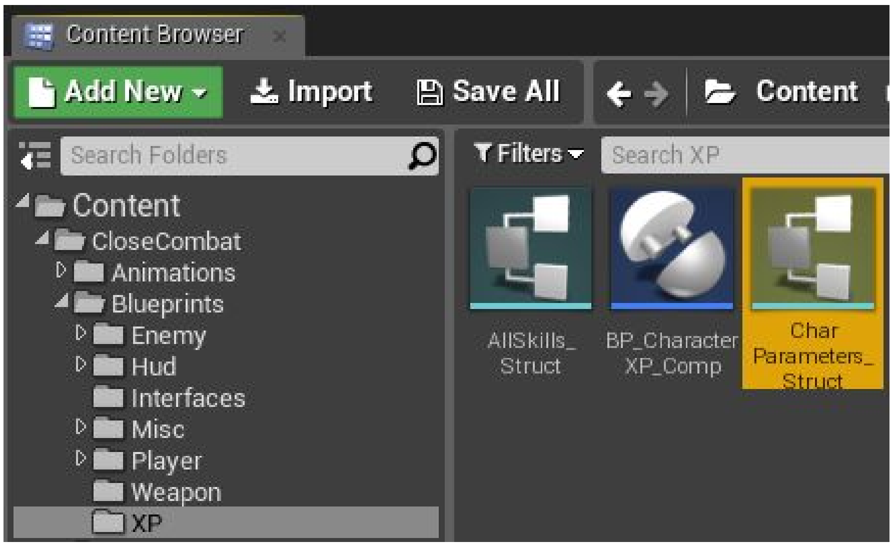


XP Settings

Levels	5 Array elements
0	3 members
1	3 members
2	3 members
Level_XP	50.0
CharParameters	
MaxHealth	130.0
HealthRecoveryTime	-15.0
BlockDamageMultiply	0.6
AvoidDamageMultiply	0.2
AdditionalDamage	10.0
AttackSpeedRate	0.3
Skills	4 Map elements
Block	
StealthKill	
HeavyAttack	
Roll	
3	3 members
4	3 members
Current Level	4
XP Current	0.0
XP Remains	0.0

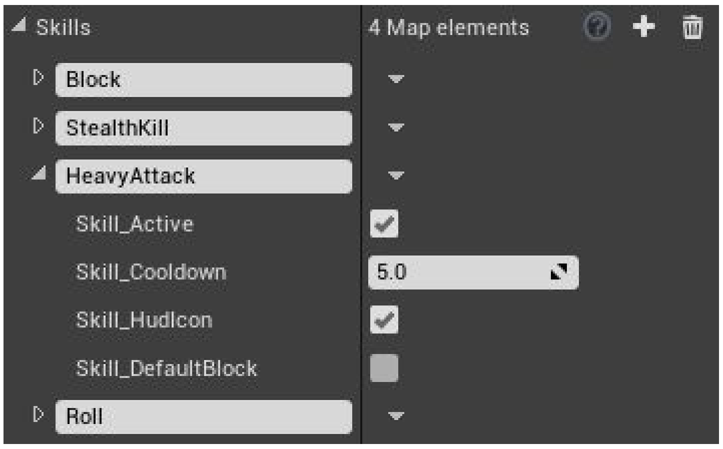
In this example, we discuss in details the 2nd level of the character. You may create any amount of levels that you need.
Level XP is an amount of experience needed to get to the 3rd level.
CharParameters is a structure of character parameters at the current level (the 2nd level). It stores information such as the amount of health the character must have, how much less time it needs to get his health fully restored, how much damage the block absorbs, weapon

damage buff, etc … You may have extra parameters, just expand the structure “**CharParameters**”.



The structure and other parameters of the level are passed to the character when the level is up at the event “**Add_XP**” via interfaces. Refer to commentaries within the source code. *Skills* is an array of skill structures of the character in the current level.

.



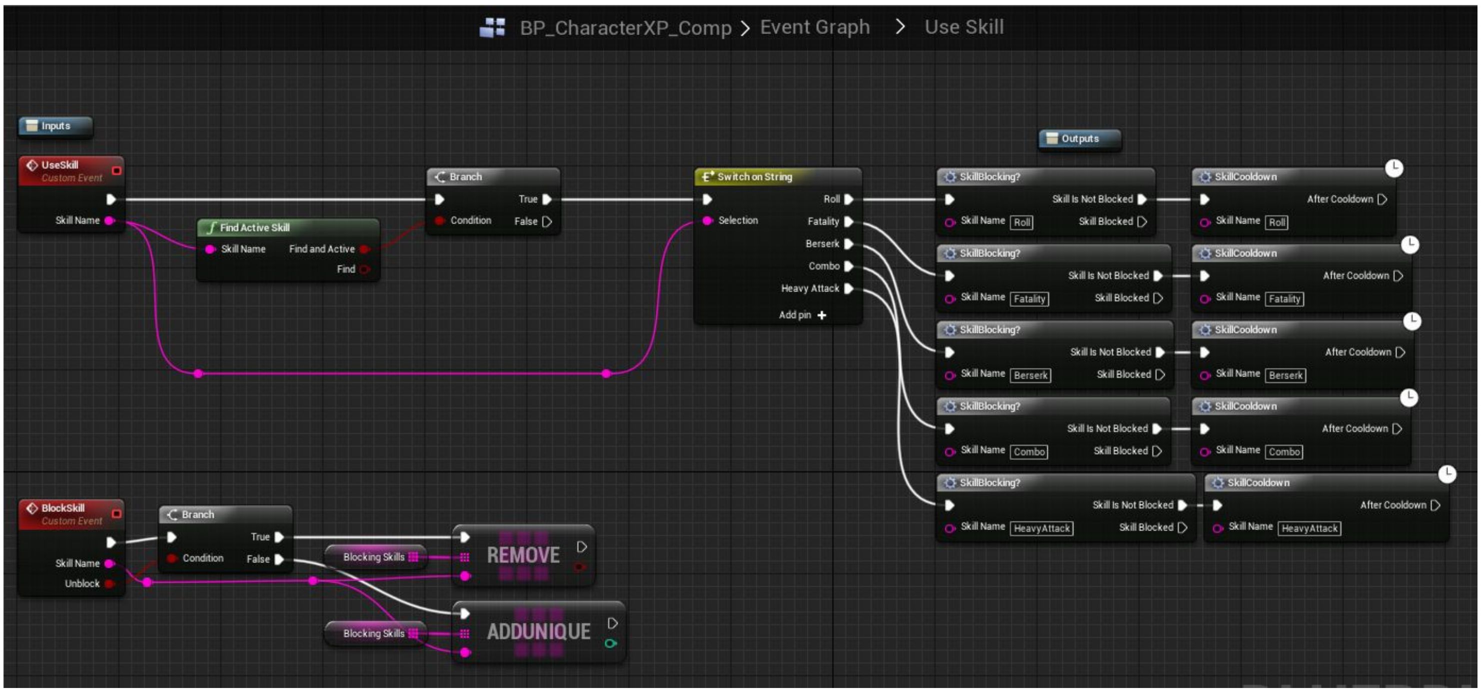
Skill_Active	✓
Skill_Cooldown	5.0
Skill_HudIcon	✓
Skill_DefaultBlock	

In the example of the skill “HeavyAttack”:
Skill_Active. Whether the skill is enabled in the beginning.
Skill Cooldown is a cool-down time for re-activation of the skill.
Skill_HudIcon. Whether the skill has a visual representation on game interface (HUD).
Skill_DefaultBlock. Whether the skill is blocked in the beginning. This parameters is for switchable skills like, for instance, the fatality. The fatality is activated only after you have activated it and delivered the final blow to a target. This is why the fatality is initially blocked. It is activated by pressing its activation button (by default, Key 2).

These are all the parameters that are passed across character’s XP levels by default. If you add extra parameters that you need, fill the array for all XP levels of the character.

Character’s Skills

You may use various skills of the character. For that, you must specify them in XP management component “**CharacterXP**” of the character first. By default, the skills are added at the event “**Use Skill**”.

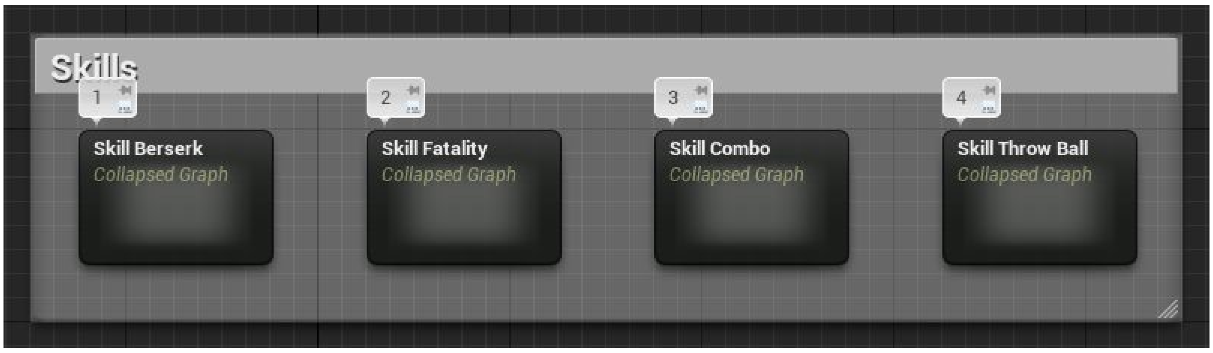


Our newly added skill must have unchangeable name. Add new skills similarly to existing skills.

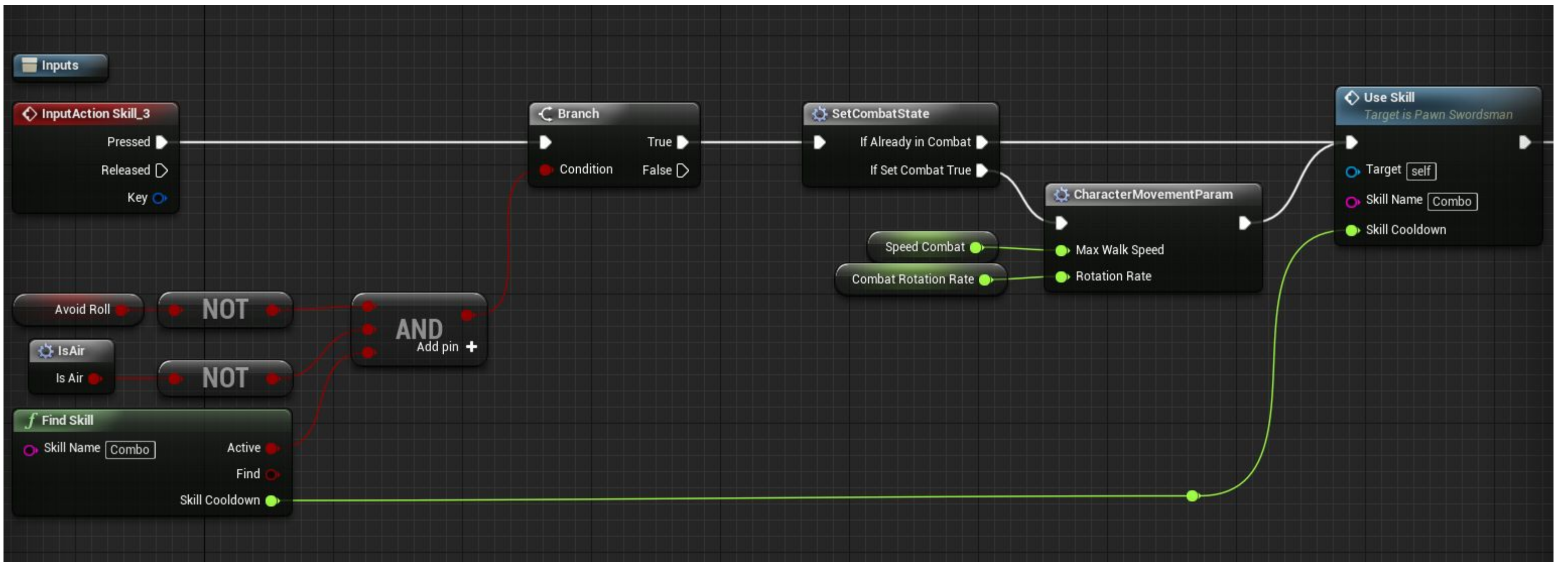
Types of skills

By default, there are 3 types of skills:
Active are skills that are executed by pressing a button.
Passive are skills that must simply be present in the structure of a level to be executed.
Switchable are skills that are executed only in case they are enabled with a switch beforehand.

Let’s consider all 3 types of skills based on existing examples of skills.
Active skill. A series of 5 consecutive strikes “**Combo**” is an active skill and is binded to Key 3.



When pressing the key, the search for the skill (together with extra conditions) is being done in component “**Chatacter_XP**”. This component is renamed as **XP** to make it shorter. Function “**Find Skill**” is implemented to do the searching for skills.



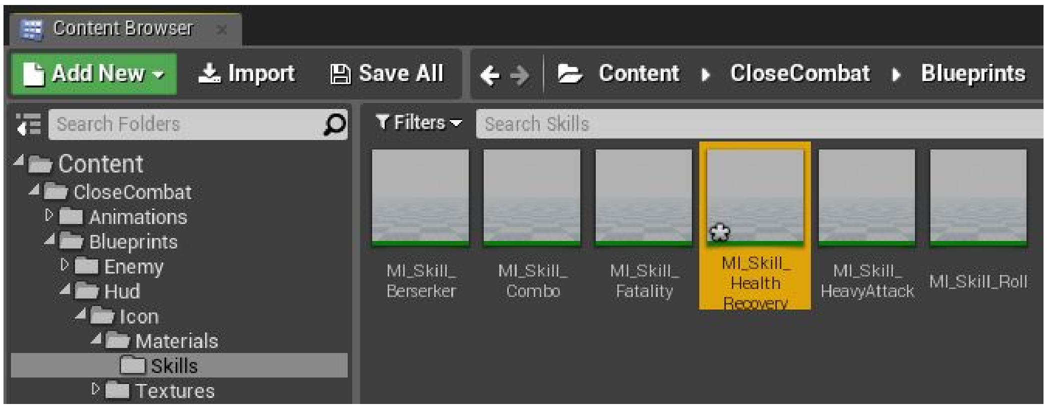
If the current skill is found and enabled (not in the cool-down state), then the skill is executed with the help of function “**Use Skill**”. The function triggers both the cool-down timer for the skill in XP management component and an animation in game interface. Actual logic of the skill goes after function “**Use Skill**”: triggering of a combo animation and various effects.

Passive skill. Stealth kill from behind does not require a cool-down, this is why it is a passive skill. When targeting and closing the distance, the skill “**StealthKill**” is being searched for. If it is present in the structure of the level, then there is an option to kill an enemy from behind.

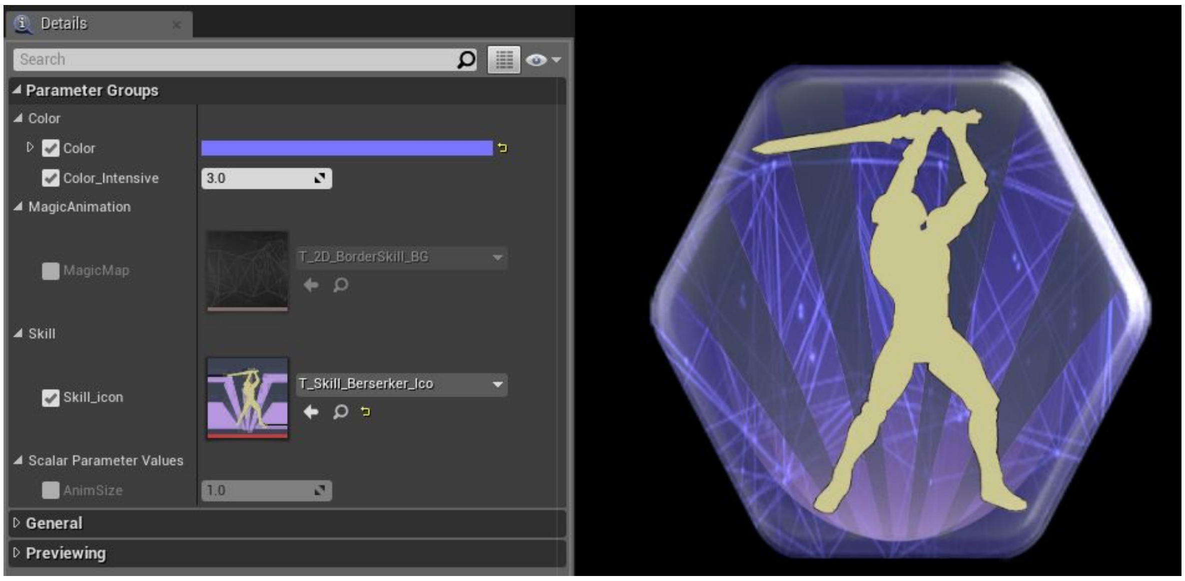
Switchable skill. The fatality has both cool-down time and enabled/disabled state. The latter is done with the help of the blocking of the skill. The fatality works as follows. If such skill is present in the structure of the level, and it is not blocked, and it is enabled (not in cool-down state), then during the final strike to an enemy, the fatality is activated. By default, blocking /unblocking is switched by pressing Key 2.

Visual representation of skills.

There is a unified visual scheme for all skills in form of material “**M_2D_Skill**”.

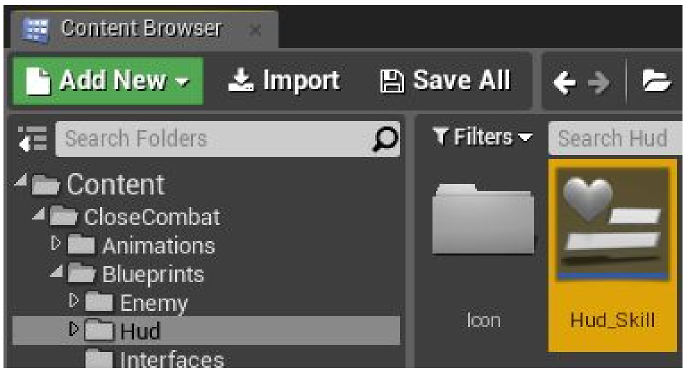


Instances for each skill are in folder Skills. To create a new icon for your skills, click RIGHT MOUSE BUTTON on material “**M_2D_Skill**” and choose Create Material instance. You create an instance, which has settings for visual representation of the skill.



Color is a background color of the material.
Color Intensive is the intensity of a color (brightness).
MagicMap is a texture for a background animation.
Skill_Icon is an icon for the skill on the foreground.
AnimSize is a parameter for filling the background during the skill cool-down.
Demonstration of the material.

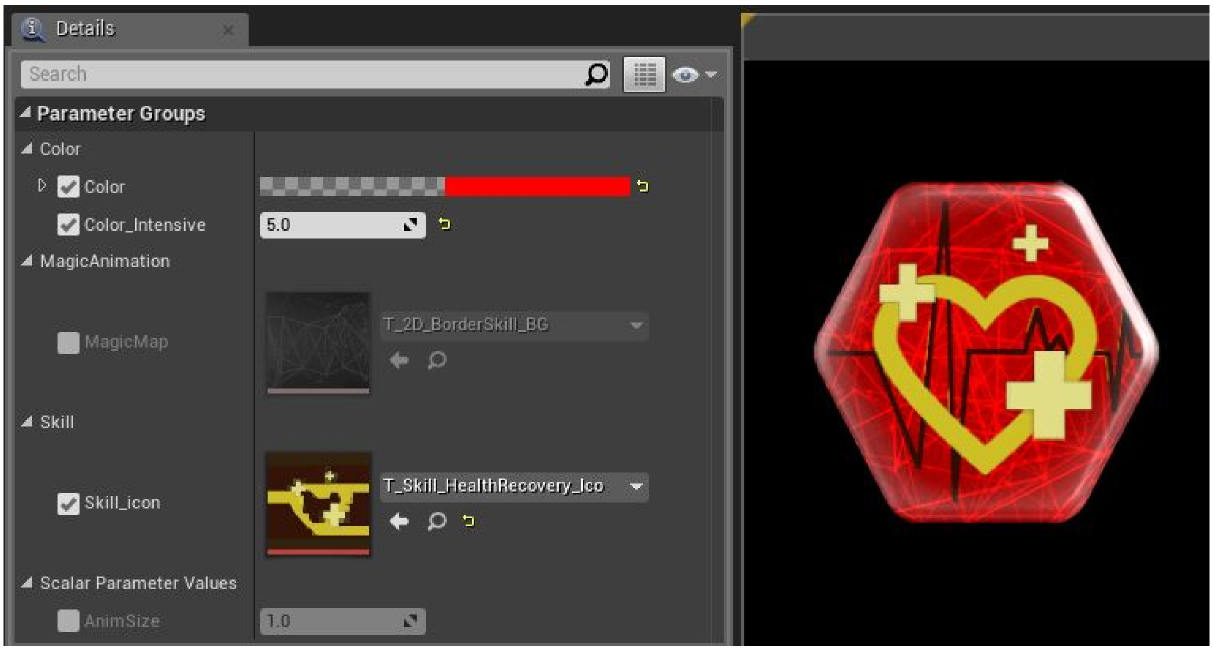
After you are done adjusting the visuals for your skill, you need to place it into the database of visuals for skills in widget “**Hud_Skill**”.



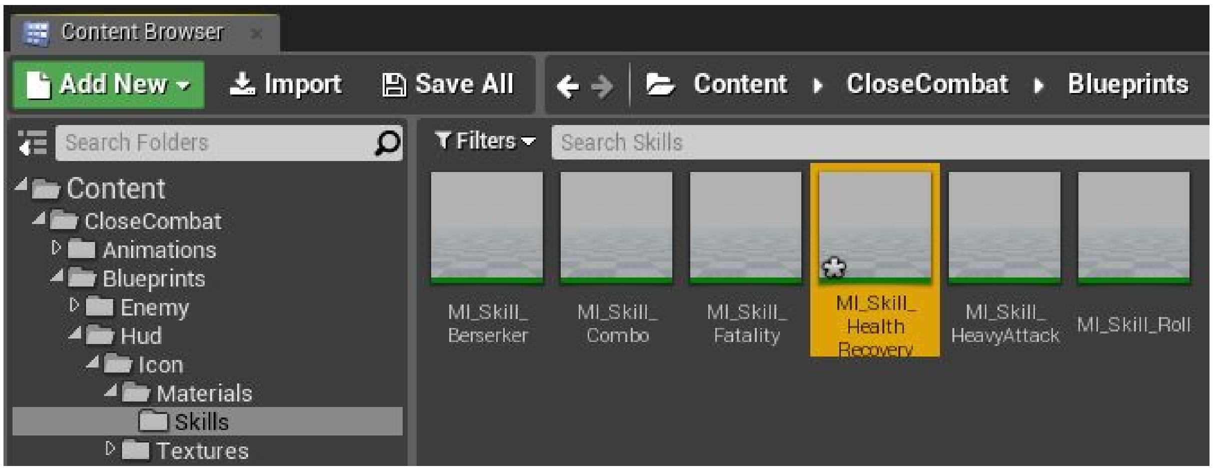
Open “**Hud_Skill**”, select map variable “**Skill_Visual**” and add a new index. Specify the name of the skill on the left. It must be the same name used in XP management component. Specify your created material on the right. Now your new icon is in the database. If a skill with the same name is added, the game interface (HUD) will display your skill.

Creation of a new skill.

Let’s consider creation of a new skill. We are going to create a skill that fully restores health and has a cool-down time of 20 seconds after it has been used. At first, we create its visual representation (material_instance) as it has been described above. Here is what I have got. The icon is available in the project:

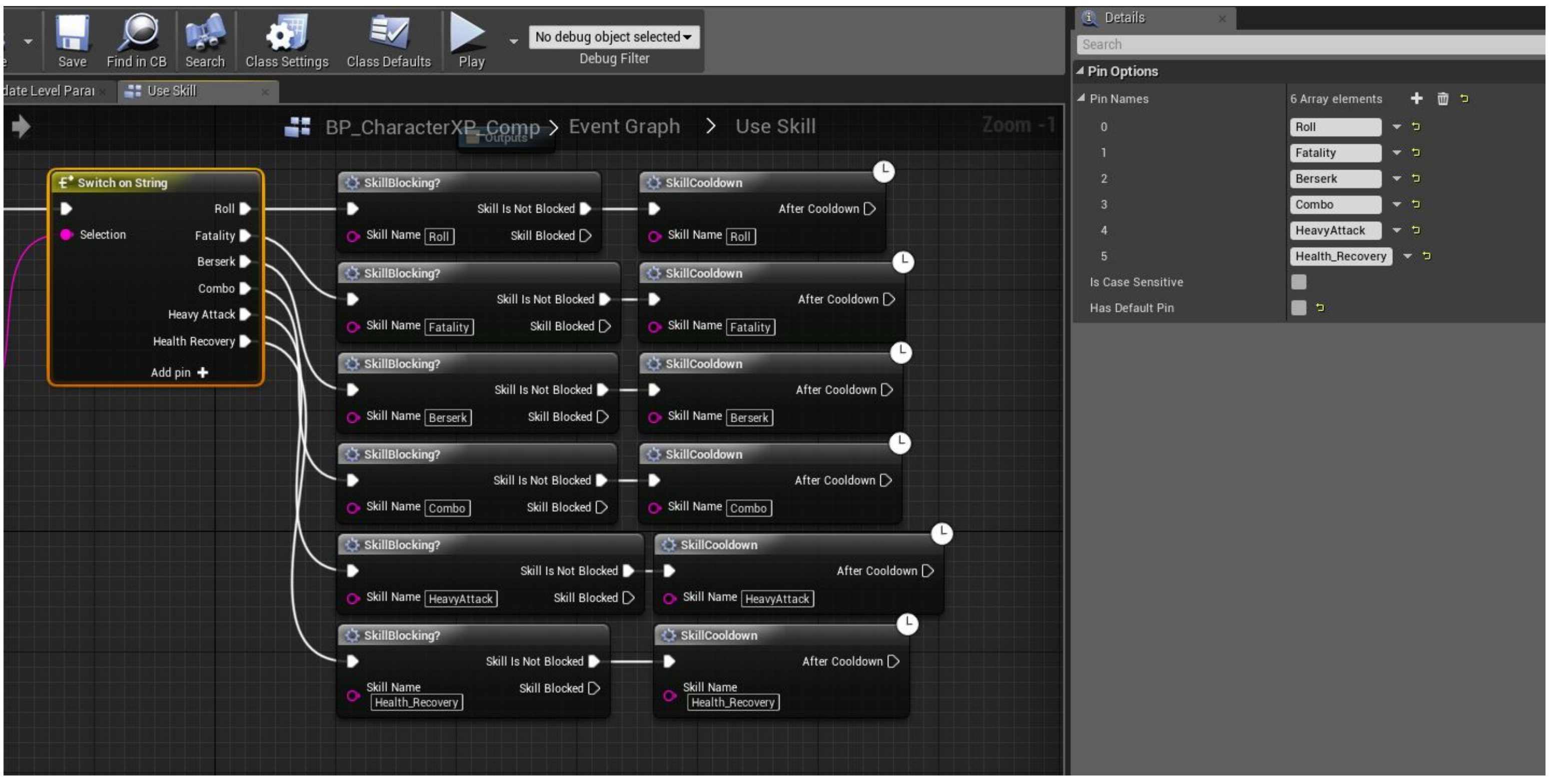


For the unified structure, place your material together with other materials for skills.



Further on, add our material in the list of visual representation of skills in widget “**Hud_Skill**” as it is described above. Add “**Health_Recovery**” to the name of the skill.

Next, add our skill in component “**CharacterXP**”. Go to event “**Use_Skill**” and add “**Health_Recovery**” by analogy with other skills.



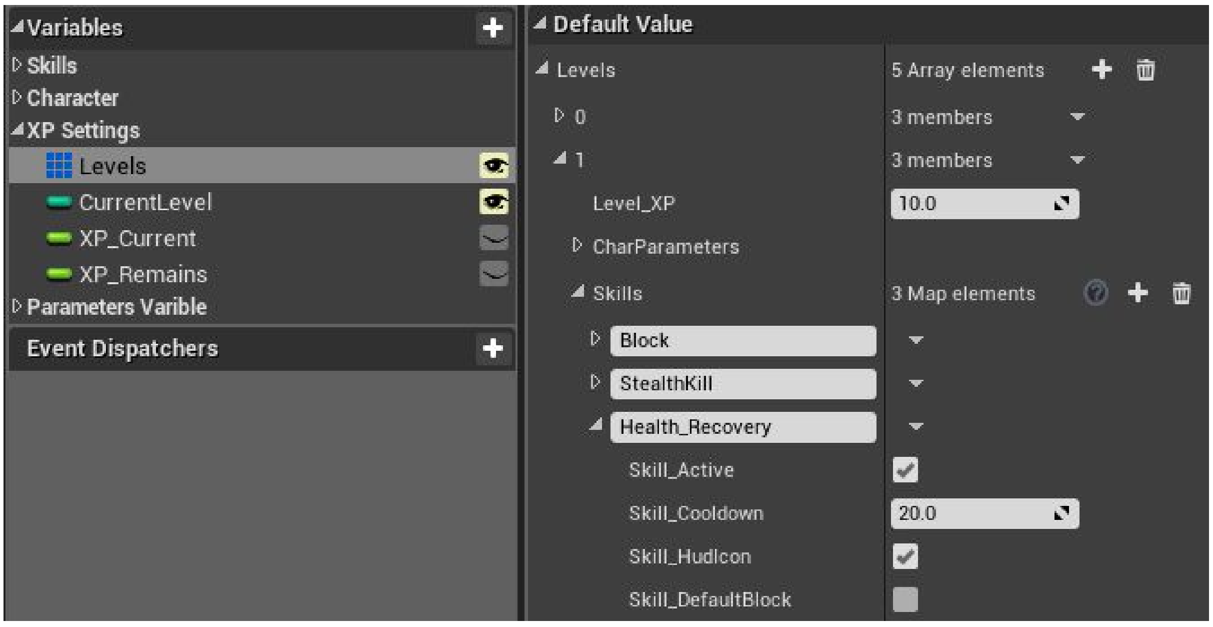
Details

Pin Options

Pin Names	6 Array elements
0	Roll
1	Fatality
2	Berserk
3	Combo
4	HeavyAttack
5	Health_Recovery
Is Case Sensitive	☐
Has Default Pin	☐

Your result should look like as in figure above, at the bottom after “**HeavyAttack**”.

Then specify at what level the skill becomes available and its settings. To do so, go to **Class defaults** or select variable “**Levels**”. Go to the first index (the first level), then open the list of skills and add our “**Health_Recovery**”. This is what you should get:



Default Value

Levels	5 Array elements
0	3 members
1	3 members
Level_XP	10.0
CharParameters	
Skills	3 Map elements
Block	
StealthKill	
Health_Recovery	
Skill_Active	✓
Skill_Cooldown	20.0
Skill_HudIcon	✓
Skill_DefaultBlock	☐

Settings of your skill:

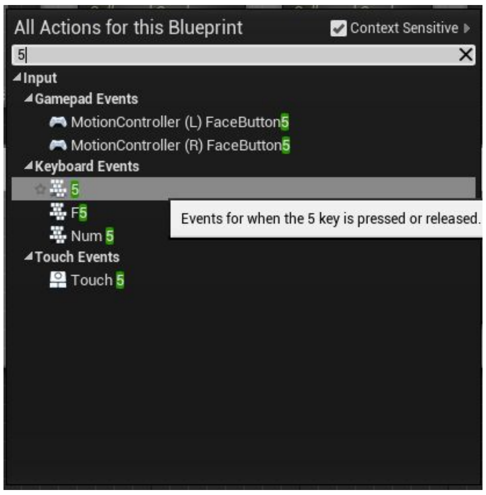
Skill_Active is enabled, since the skill must be enabled from the beginning.

Skill_Cooldown is equal to 20 seconds to re-activate the skill as it was planned.

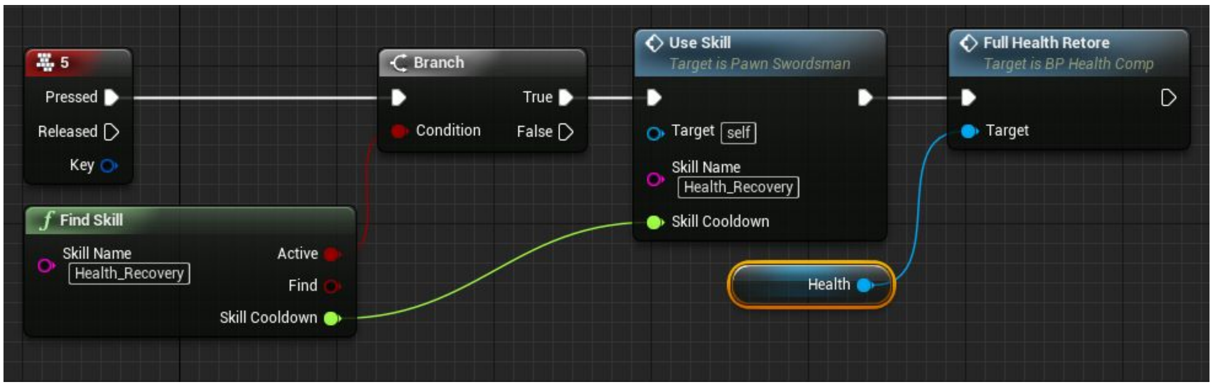
Skill_HudIcon is enabled, since the skill is active and it has an icon.

Skill_DefaultBlock is disabled, since the skill does not require blocking.

Now let’s add actual logic of health recovery to the skill. Go to the character “**Pawn_Swordsman**” and add the input event for Key 5.

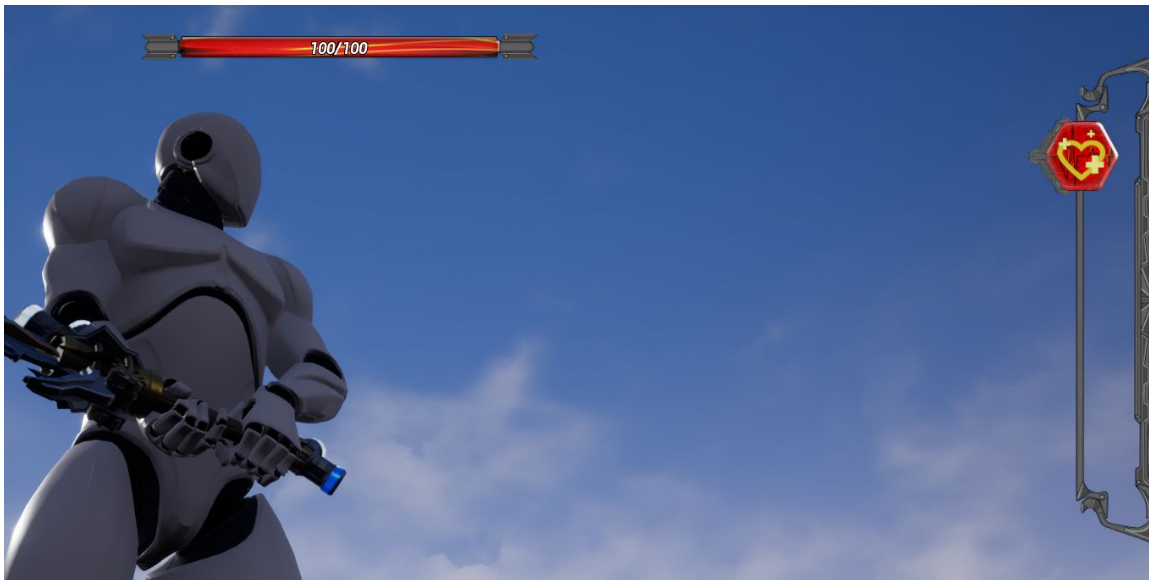

All Actions for this Blueprint
Context Sensitive
5
Input
Gamepad Events
MotionController (L) FaceButton5
MotionController (R) FaceButton5
Keyboard Events
5
F5
Num 5
Touch Events
Touch 5
Events for when the 5 key is pressed or released.

We add the following logic at this event:



This is what happens step-by-step:
Look for the skill called “**Health_Recovery**”. If it is found and active (the variable Active will be returned only if the skill is found, no need to double-check when running Find), then the skill is used with the help of function “Use Skill”. This function starts the skill cool-down timer and the animation of the game interface. The actual health recovery occurs after function Use Skill. Event “**Full Health Restore**” from health component fully restores health.

Compile “**Pawn_Swordsman**”, launch the game, and you will see the new active skill for health restoration, which is enabled by pressing key 5. All other skills work based on the same principle.

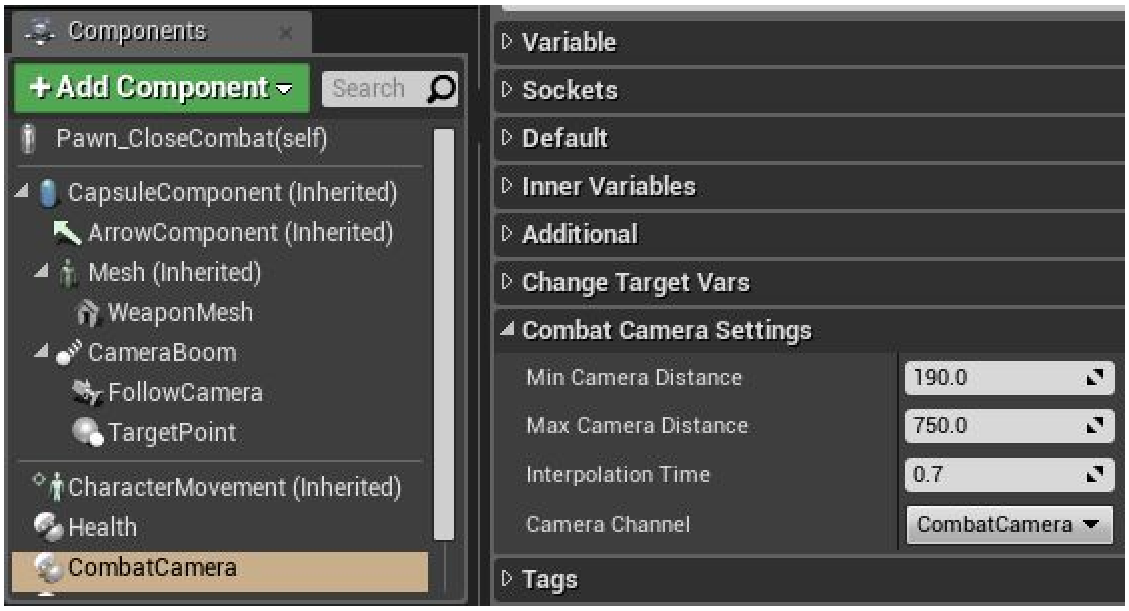


Combat camera.

Combat camera is for additional overview or perspective between two targets. By default, it can be used if the character is locked on the target. The camera keeps two targets in sight (on the screen). Demonstration.

Camera settings.

Let’s discuss camera settings. Select combat camera component. You only need category “**Combat Camera Settings**” in tab “**Details**”:



Components

+Add Component

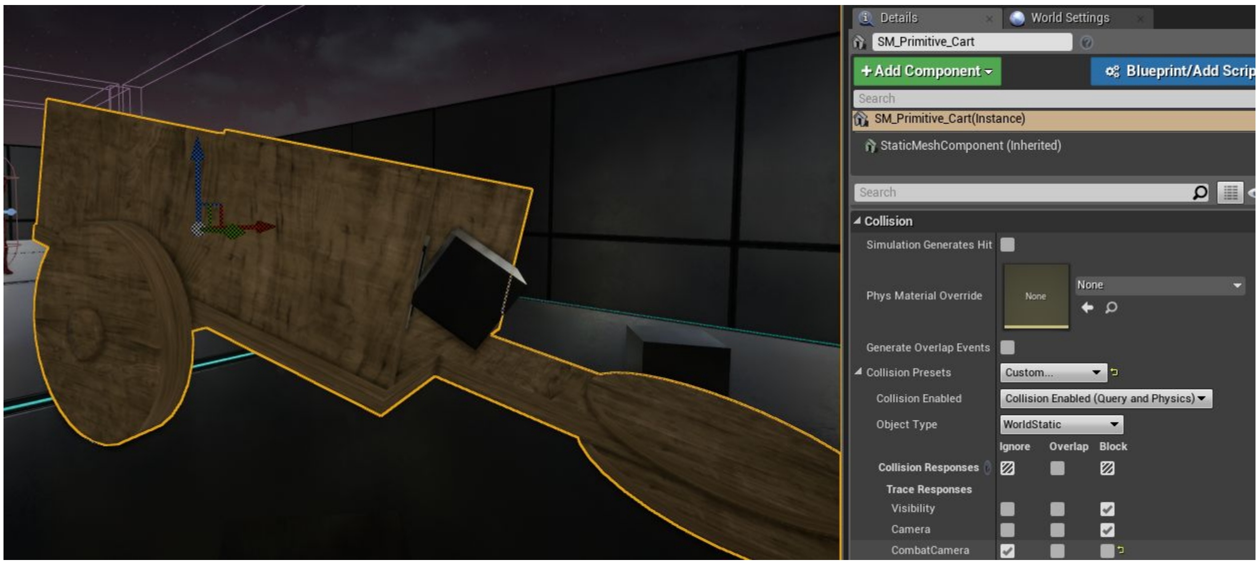
Pawn_CloseCombat(self)
- CapsuleComponent (Inherited)
  - ArrowComponent (Inherited)
  - Mesh (Inherited)
    - WeaponMesh
  - CameraBoom
    - FollowCamera
    - TargetPoint
- CharacterMovement (Inherited)
- Health
- CombatCamera

Variable
Sockets
Default
Inner Variables
Additional
Change Target Vars
Combat Camera Settings

Min Camera Distance	190.0
Max Camera Distance	750.0
Interpolation Time	0.7
Camera Channel	CombatCamera

Tags

Min Camera Distance is a distance between the camera and the center between two targets at their minimum distance.
Max Camera Distance is a distance between the camera and the center between two targets at their maximum. The maximum distance is 7.5 meters.
Interpolation Time is a time interval for a smooth transition of the camera from the combat mode into the basic mode.
Camera Channel is a channel for the camera. By default, an additional channel is created for the combat camera. It is needed to ignore some objects. For instance, there is a box in the level (a pole, some prop, …). During a combat, the camera should not collide with such object and spoil the view, but when there is no combat, it must collide with the object. Therefore, in the object collision channel specify the channel “Combat Camera” to ignore this object.



Details | World Settings

SM_Primitive_Cart

+Add Component | Blueprint/Add Scrip

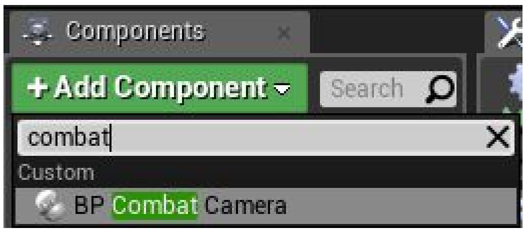
SM_Primitive_Cart(Instance)
- StaticMeshComponent (Inherited)

Collision

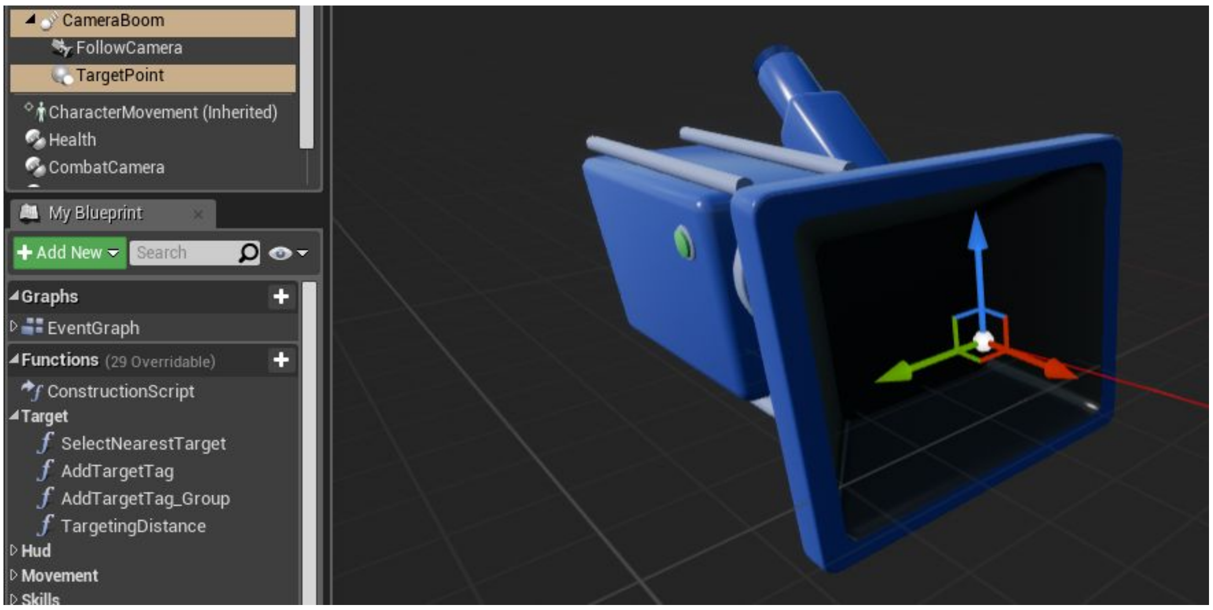
Simulation Generates Hit	☐		
Phys Material Override	None		
Generate Overlap Events	☐		
Collision Presets	Custom...		
Collision Enabled	Collision Enabled (Query and Physics)		
Object Type	WorldStatic		
	Ignore	Overlap	Block
Collision Responses	▨	☐	▨
Trace Responses			
Visibility	☐	☐	✓
Camera	☐	☐	✓
CombatCamera	✓	☐	☐

How to add combat camera

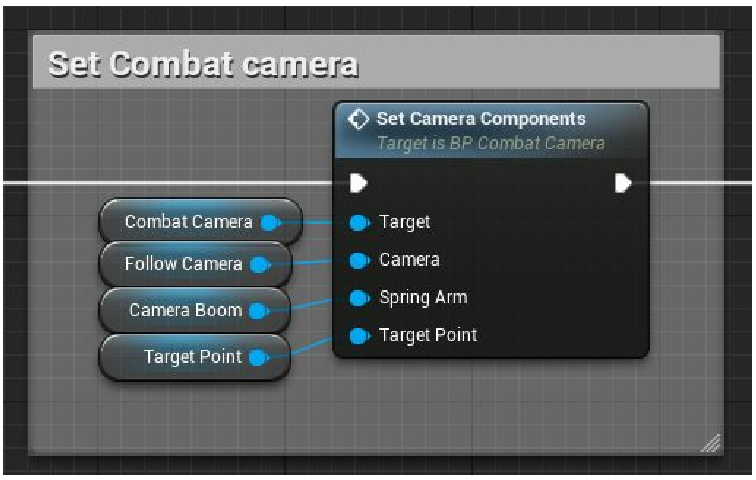
Combat camera is a component and can be added to any actor, which uses a spring arm and a camera. To add combat camera, choose it in the list Add Component.



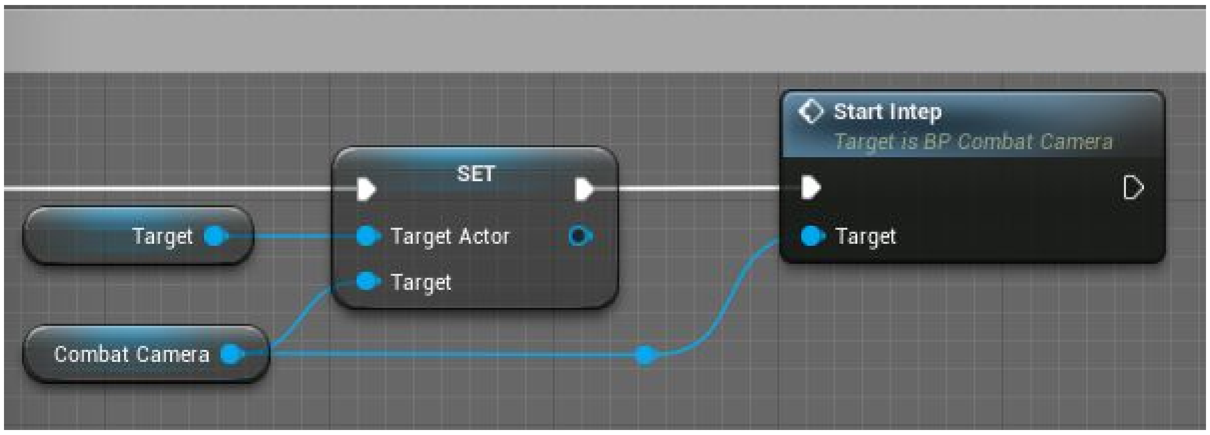
Followed by that, a child component of “Scene component” for the spring arm must be created. In the project, this component is renamed as “TargetPoint”. The component is the location of the end point of the spring arm. This is needed to return the camera in its base position. The component itself must be located in the same coordinates as the camera.



The last step is to pass information about components into combat camera component. The logic of the combat camera must know what components to work with. It is better to be done at the event Begin Play. Call function “Set Camera Components” from combat camera component and specify the components.



To make sure combat camera works correctly, it must be given a reference to the target. The camera itself is enabled at the event “Start Interp”. The event is binded to button “V” by default.



Refer to commentaries inside Blueprint classes for a more detailed description of the source code. Also, read pop-up tips when pointing the mouse cursor at the parameters.

Thank you for your attention.
With best wishes, ZzGERTzZ.