

Easy Survival RPG v5.0

Documentation



Introduction

Easy Survival RPG is a project for **Unreal Engine 4/5** that contains almost everything you need to develop your own multiplayer **Survival, RPG, Survival RPG, Adventure** and so on games. All mechanics are designed from the ground up for multiplayer.

It is based on many mechanics, ranging from basic character control and interaction with the world, to creating multiplayer sessions with saving and loading player states. This project is a very complex system that can be used to create other projects. All systems and mechanics are integrated into each other and cannot be easily extracted individually. The project should be used as a basis for your projects or as a reference for creating similar mechanics in finished projects.

This documentation contains basic information on working with multiplayer projects, as well as complete information describing all the mechanics and principles of their work, as well as guides for working with them.

arbutas#2127

Contents

Introduction

Contents

1. Basic concepts

- 1.1. Main classes
- 1.2. Working with interfaces
- 1.3. Working with data tables
- 1.4. Work with Libraries
- 1.5. Work with Gameplay Tags
- 1.6. Networking

2. Description of the main classes

- 2.1. BP_GameInstance
 - 2.1.1. Description
 - 2.1.2. Variables
 - 2.1.2.1. Game settings variables
 - 2.1.2.2. Player Character settings variables
 - 2.1.2.3. State variables
 - 2.1.3. Functions
- 2.2. BP_PlayerController
 - 2.2.1. Description
 - 2.2.2. Macros
 - 2.2.3. Variables
- 2.3. BP_HUD_Game
 - 2.3.1. Description
 - 2.3.2. Variables
 - 2.3.3. Functions
 - 2.3.4. UI_HUD
- 2.4. BP_GameSave_Settings
 - 2.4.1. Description
 - 2.4.2. Variables
 - 2.4.3. Functions
- 2.5. BP_GameSave_Session

-
- 2.5.1. Description
 - 2.5.2. Variables
 - 2.6. BP_PlayerManagerComponent
 - 2.6.1. Description
 - 2.6.2. Variables
 - 2.6.2.1. References variables
 - 2.6.2.2. Player Data variables
 - 2.6.2.3. State variables
 - 2.6.2.4. Settings variables
 - 2.6.3. Functions
 - 2.6.4. Event Dispatchers

3. Description of the main systems

- 3.1. Object interaction system
 - 3.1.1. Description
 - 3.1.2. BPI_InteractionObject
 - 3.1.3. BP_InteractionComponent
 - 3.1.4. Variables
 - 3.1.5. Functions
 - 3.1.6. Event Dispatchers
- 3.2. Items interaction system
 - 3.2.1. Description
 - 3.2.2. Structures
 - 3.2.3. Overall item information
 - 3.2.4. Classification of items
 - 3.2.5. Item data map-list identifiers
 - 3.2.6. World items
 - 3.2.7. Usable items
 - 3.2.8. BP_ContainerComponent
 - 3.2.9. BP_EquipmentComponent
 - 3.2.10. BP_HotbarComponent
 - 3.2.11. BP_CraftingComponent
 - 3.2.12. BP_FuelGeneratorComponent
 - 3.2.13. BP_TradeComponent
 - 3.2.14. Player interaction with inventory and equipment
 - 3.2.15. Player interaction with other containers
 - 3.2.16. Player interaction with a crafting component

-
- 3.2.17. Player interaction with generator component
 - 3.2.18. Player interaction with the trade component
 - 3.2.19. Player Interaction with the Hotbar
 - 3.3. Building System
 - 3.3.1. Description
 - 3.3.2. BP_Building_BaseObject
 - 3.3.2.1. Description
 - 3.3.2.2. Interfaces
 - 3.3.2.3. Variables
 - 3.3.2.4. Building object settings
 - 3.3.2.5. Functions
 - 3.3.2.6. Build collisions
 - 3.3.2.7. Build components
 - 3.3.2.8. Socket system
 - 3.3.3. Floor building object
 - 3.3.3.1. Description
 - 3.3.3.2. Variables
 - 3.3.3.3. Functions
 - 3.3.3.4. Check support settings
 - 3.3.4. Wall building object
 - 3.3.4.1. Description
 - 3.3.4.2. Functions
 - 3.3.4.3. Check support settings
 - 3.3.5. Building Component
 - 3.3.5.1. Description
 - 3.3.5.2. Variables
 - 3.3.5.3. Functions
 - 3.3.5.4. Event dispatchers
 - 3.3.6. Modular building objects
 - 3.3.6.1. Foundation
 - 3.3.6.2. Triangle foundation
 - 3.3.6.3. Ramp
 - 3.3.6.4. Wall
 - 3.3.6.5. Door frame
 - 3.3.6.6. Window frame
 - 3.3.6.7. Ceiling

-
- 3.3.6.8. Triangle ceiling
 - 3.3.6.9. Stairs
 - 3.3.6.10. Fence
 - 3.3.6.11. Roof wall left
 - 3.3.6.12. Roof wall right
 - 3.3.6.13. Roof
 - 3.3.6.14. Top roof wall
 - 3.3.6.15. Top roof
 - 3.4. Replaceable Instance System
 - 3.4.1. Description
 - 3.4.2. BP_FoliageCheckerComponent
 - 3.4.3. BP_InstancedComponent_Base
 - 3.4.4. BP_FoliageHideComponent
 - 3.4.5. BP_FoliageManagerComponent
 - 3.4.6. Configuring Replaceable Instances
 - 3.5. Dialogue system
 - 3.5.1. Description
 - 3.5.2. STR_Dialogue
 - 3.5.3. STR_DialogueReply
 - 3.5.4. BP_Dialogue_Base
 - 3.5.5. Reply conditions
 - 3.5.6. Reply events
 - 3.5.7. Dialogue scheme
 - 3.6. Quest system
 - 3.6.1. Description
 - 3.6.2. BP_Quest_Base
 - 3.6.3. BP_QuestTaskComponent_Base
 - 3.6.4. Journal notes
 - 3.7. Save & Load System
 - 3.7.1. Description
 - 3.7.2. STR_SaveData_Player
 - 3.7.3. STR_SaveData_Level
 - 3.7.4. BP_GameSave_Settings
 - 3.7.5. BP_GameSave_Session
 - 3.7.6. BPI_SaveData
 - 3.7.7. Saving & loading an object

-
- 3.7.8. Saving & loading a level
 - 3.8. Attributes system
 - 3.8.1. Description
 - 3.8.2. BP_AttributesComponent
 - 3.8.3. Character attributes
 - 3.8.4. Ability system attributes
 - 3.8.5. Status effects attributes
 - 3.8.6. Damage system attributes
 - 3.9. Advanced damage system
 - 3.9.1. Description
 - 3.9.2. BP_DamageSystemComponent
 - 3.9.3. Apply damage
 - 3.10. Advanced ability system
 - 3.10.1. Description
 - 3.10.2. BP_AbilitySystemComponent
 - 3.10.3. Working with ability blueprints
 - 3.10.4. Working with status effects.
 - 3.10.5. Working with projectiles.
 - 3.11. Footstep system
 - 3.11.1. Description
 - 3.11.2. BP_FootstepComponent
 - 3.11.2.1. Description
 - 3.11.2.2. Variables
 - 3.11.2.3. Config settings
 - 3.11.2.4. Footstep settings
 - 3.11.4. Footstep animation settings
 - 3.12. Skills tree system
 - 3.12.1. Description
 - 3.12.2. Working with skills
 - 3.13. Spell book system
 - 3.13.1. Description
 - 3.13.2. Working with spell book
 - 3.14. Swimming system
 - 3.14.1. Description
 - 3.14.2. Working with swimming system

4. Characters

-
- 4.1. Base character class
 - 4.1.1. Description
 - 4.1.2. Components
 - 4.1.3. Variables
 - 4.1.3.1. Base settings variables
 - 4.1.3.2. AI settings variables
 - 4.1.3.3. Health state variables
 - 4.1.3.4. Energy state variables
 - 4.1.3.5. Mana state variables
 - 4.1.3.6. AI state variables
 - 4.1.3.7. Interactions state variables
 - 4.1.4. Functions
 - 4.1.4.1. State functions
 - 4.1.4.2. AI functions
 - 4.1.4.3. Base interaction functions
 - 4.1.4.4. Combat interactions functions
 - 4.1.4.4. Dialogue interactions functions
 - 4.2. Character components
 - 4.2.1. AttributesComponent
 - 4.2.2. FootstepComponent
 - 4.2.3. FoliageCheckerComponent
 - 4.2.4. DamageSystemComponent
 - 4.2.5. AbilitySystemComponent
 - 4.2.6. Navigation Invoker
 - 4.3. Player character class
 - 4.3.1. Description
 - 4.3.2. Components
 - 4.3.3. Variables
 - 4.3.3.1. Customization settings variables
 - 4.3.3.2. Interactions settings variables
 - 4.3.3.3. Anims settings variables
 - 4.3.3.4. Camera settings variables
 - 4.3.3.5. Interactions state variables
 - 4.3.3.5. Rotation state variables
 - 4.3.3.6. Oxygen state variables
 - 4.3.3.7. Hunger state variables

-
- 4.3.3.8. Thirst state variables
 - 4.3.3.9. Customisation state variables
 - 4.3.3.10. Camera state variables
 - 4.3.3.11. Crouching state variables
 - 4.3.4. Functions
 - 4.3.4.1. State functions
 - 4.3.4.2. Items interaction functions
 - 4.3.4.3. Combat interaction functions
 - 4.3.4.4. Tools interaction functions
 - 4.3.4.5. Player interaction functions
 - 4.3.4.6. Base interaction functions
 - 4.3.4.7. Dialogue interaction functions
 - 4.3.4.8. Crouch interaction functions
 - 4.3.4.9. Customization functions
 - 4.3.5. Player interaction system
 - 4.3.5.1. Base player interactions
 - 4.3.5.2. Advanced player interaction system
 - 4.4. Work with animations
 - 4.4.1. Anim notifies
 - 4.4.2. Project notifies
 - 4.4.3. Anim Blueprint
 - 4.4.4. Character animations
 - 4.5. Artificial Intelligence
 - 4.5.1. Description
 - 4.5.2. Blackboard
 - 4.5.3. Patrol points
 - 4.5.4. Spawn points
 - 4.5.5. AI Perception
 - 4.5.6. Behavior trees
 - 4.5.6.1. Tasks
 - 4.5.6.2. Decorators
 - 4.5.6.3. Services
 - 4.5.6.4. Main character behavior
 - 4.5.6.5. Dynamic orders behavior
 - 4.5.6.6. Dynamic combat behavior
 - 4.5.6.7. Dynamic scared behavior

4.5.6.8. Dynamic calm behavior

4.5.6.9. Unique behaviors

5. Project settings

5.1. Collision settings

5.2. Control settings

5.3. AI navigation settings

5.4. Plugin settings

5.5. STEAM settings

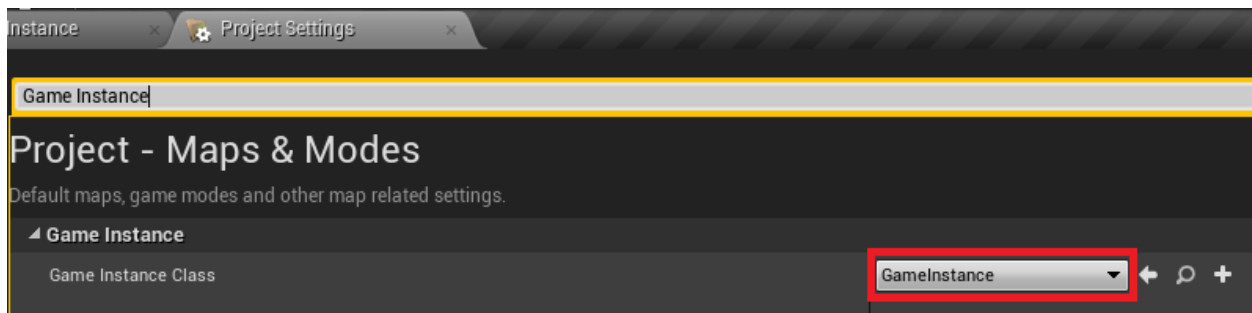
1. Basic concepts

1.1. Main classes

To create projects in **Unreal Engine 4**, you need to know the main classes that will be constantly used for specific purposes. Such classes include `GameInstance`, `GameMode`, `PlayerController`, `HUD`, and others.

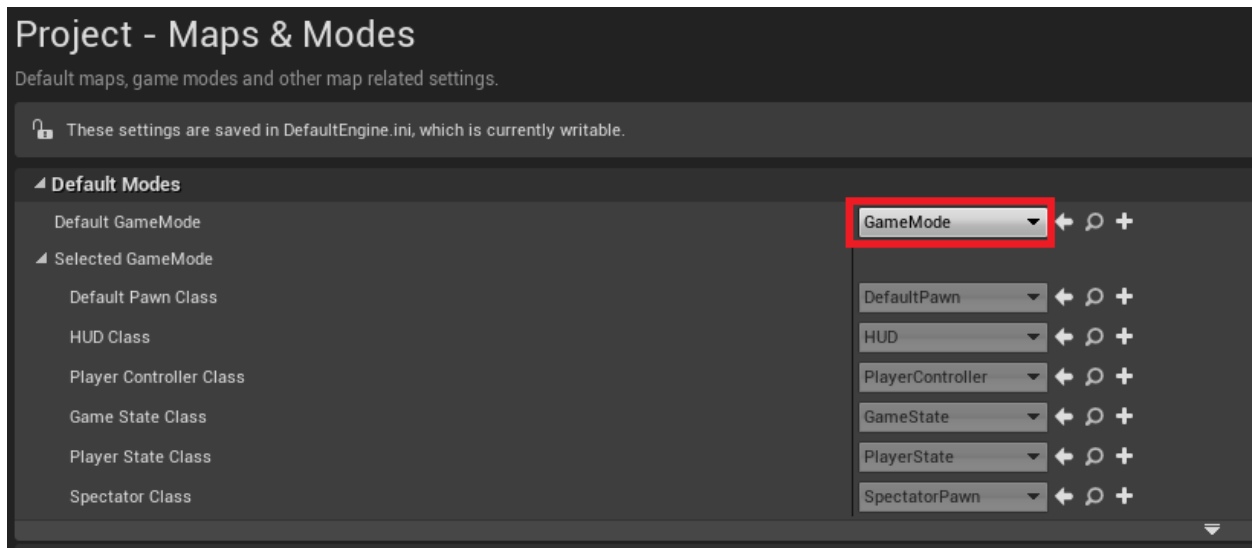
GameInstance is the main game class that is instantiated when the game starts. As a rule, it should contain methods and variables for moving between levels, methods for setting up the game, and methods for loading and saving the game. It has events that are called when the application starts and finishes, as well as when network or technical errors occur. These events can be redefined as desired.

You can change **GameInstance** in **Project Settings**.

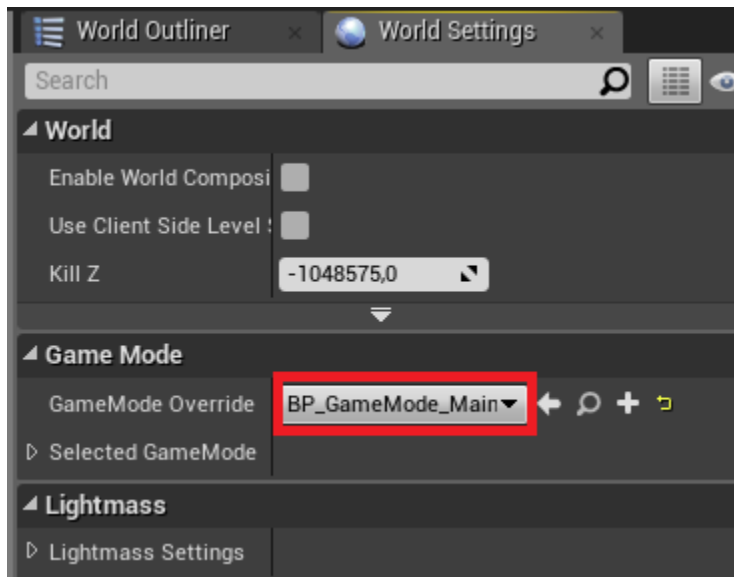


GameMode is a class that is instantiated when starting any level for a local player. It contains information about other classes that will be created with it, such as **PlayerController**, **PlayerState**, **GameState**, **HUD**, as well as information about the character that will be created by default when starting the level and that will be controlled by the player. Required for selecting the above-mentioned classes that will be created when connecting to the level. It has events that are called when a player connects or disconnects to the current level, changes the player's name, selects the spawn point and respawn point of the game character, and others. These events can be redefined as desired.

It can be set globally for the entire game in **Project Settings** in the **Maps & Modes** tab.

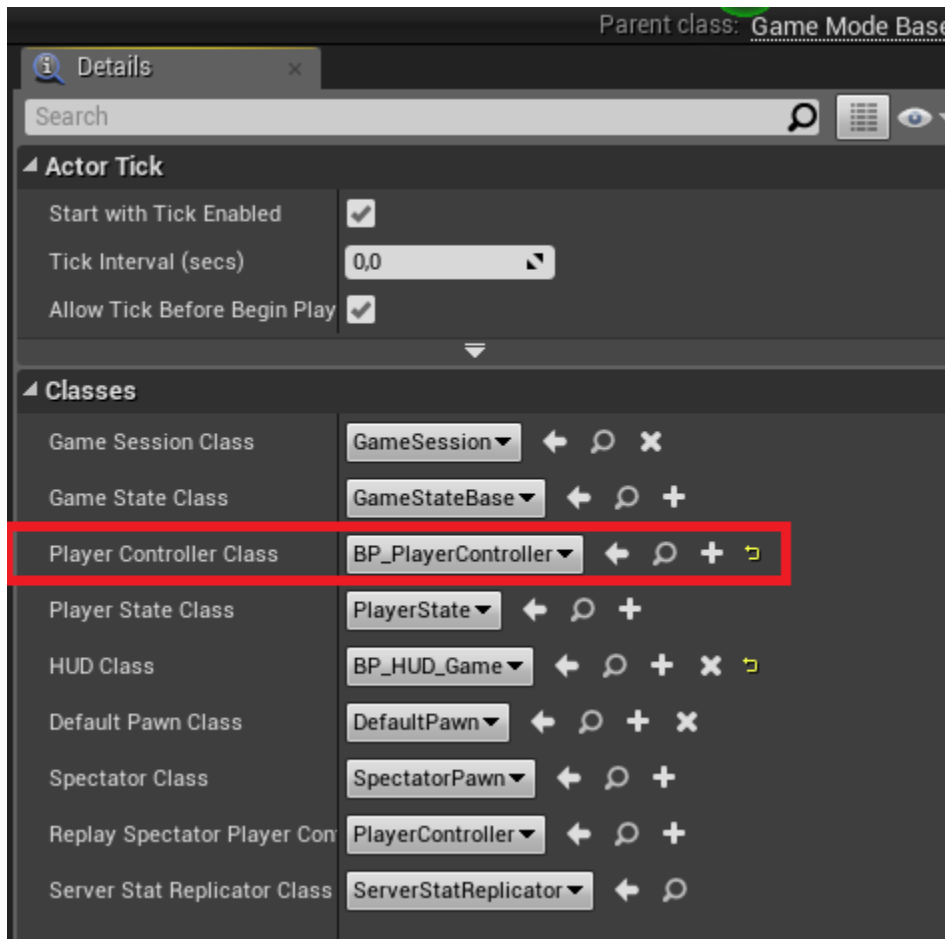


It can also be selected separately for each level in the **World Settings** window of the current level.



PlayerController is the player class that is instantiated after **GameMode**. As a rule, it should contain various player components for interacting with the world, as well as methods for controlling the character and user interface. It can also contain information about the player's status, experience, items, and so on. However, there is a separate **PlayerState** class for this purpose.

It can be set in the **GameMode** class in the **Details** tab.



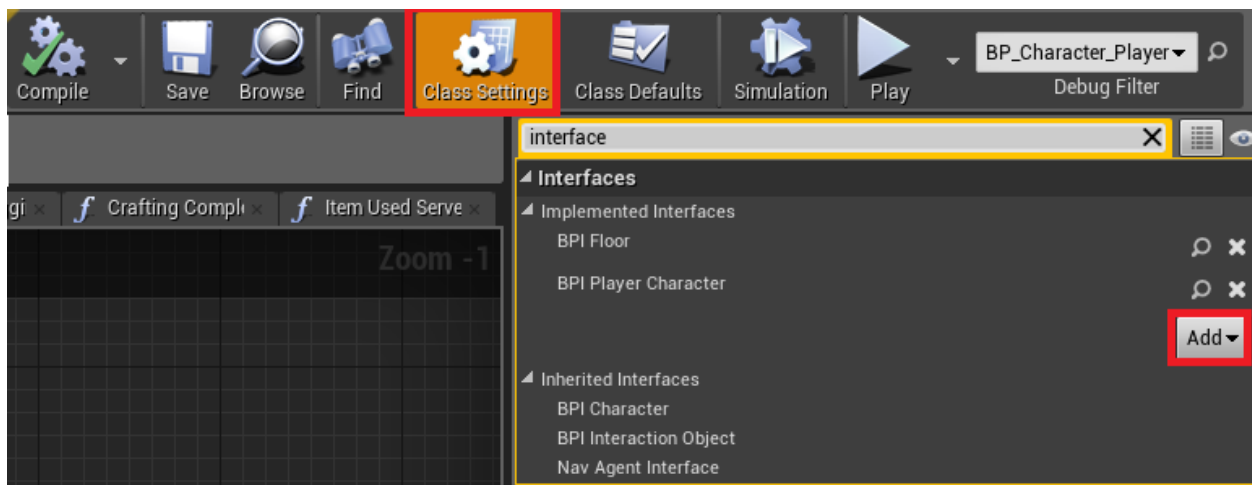
HUD is a class that is instantiated after **GameMode**. Required for creating and displaying the user interface for the player. It can also be set in the **GameMode** class.

1.2. Working with interfaces

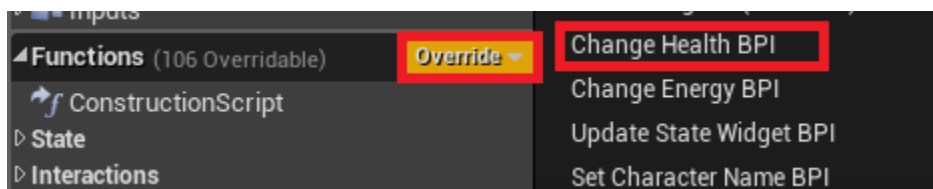
Using blueprint interfaces allows you to use the same functions to interact with different object classes. To do this, you will not need to cast the object to a specific class that has the required functions. For example, you can create an interface with functions for changing the health of an object and add it to the character's blueprint and the building's blueprint, and then write the logic for these interface functions inside these blueprints. After that, it will be possible to call the function of changing the health of the character and building, regardless of its class. Interfaces also make it easier to integrate one complex system into another.

To create an interface, click on the green **Add New** button in the **Content Browser** and select **Blueprint Interface** in the **Blueprints** section.

To add an interface for a specific actor, you need to open **Class Settings**, click on the **Add** button and select the desired interface.



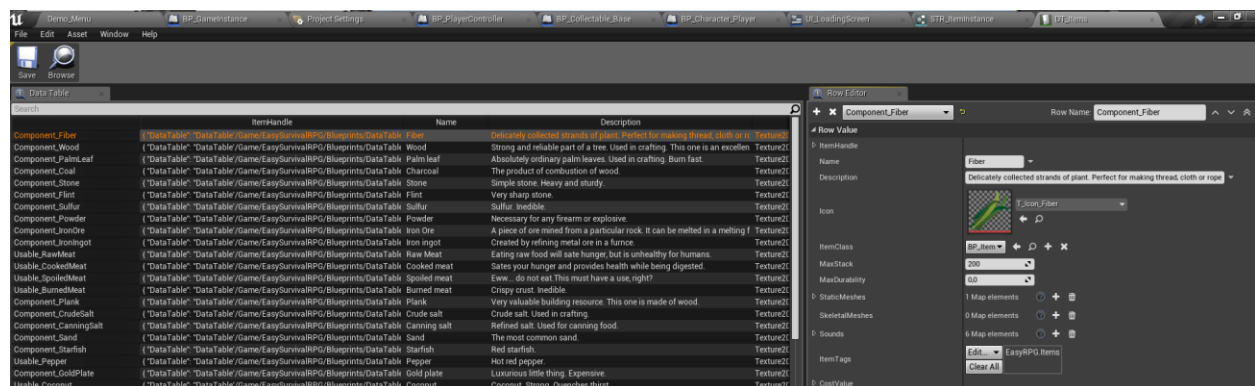
You can implement logic for an interface function by clicking on the hidden **Override** button on the function panel and selecting the desired function from the list.



1.3. Working with data tables

Data tables are needed to store a lot of information and access it from blueprints.

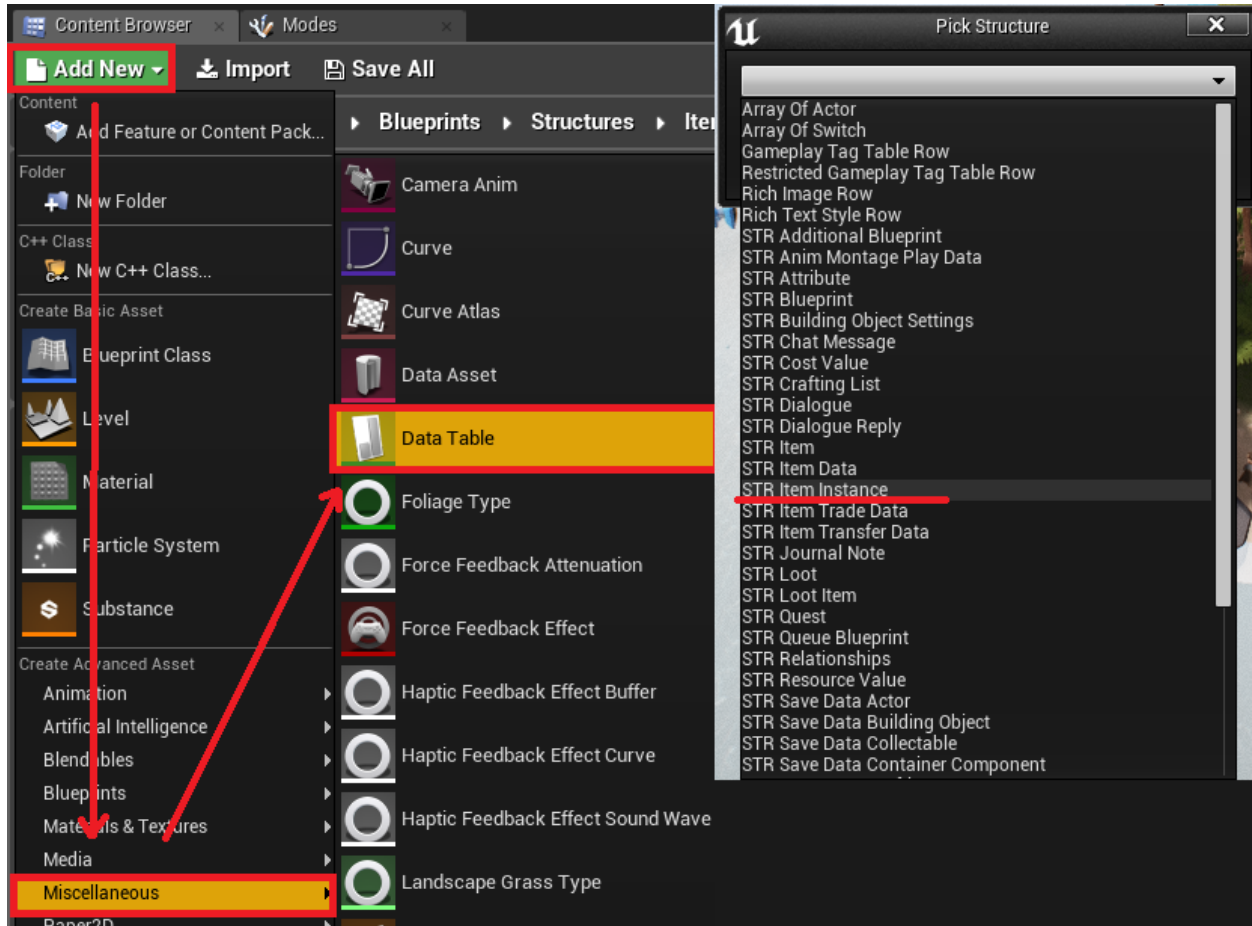
Data tables are a great solution if you plan on adding a huge number of items, dialogues, assignments, recipes, and so on to your project.



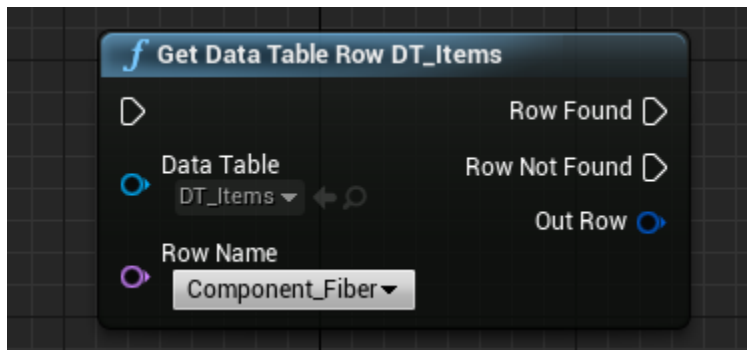
Adding a table requires a structure on which to add rows. The figure below shows the structure for an item instance used in the **Easy Survival RPG** project.



To add a table, click on the green **Add New** button in the **Content Browser** and select **Data Table** in the **Miscellaneous** section, and then select the structure on the basis of which the table is created.



You can access the data structure of a table within blueprints using the **GetDataTableRow** function.

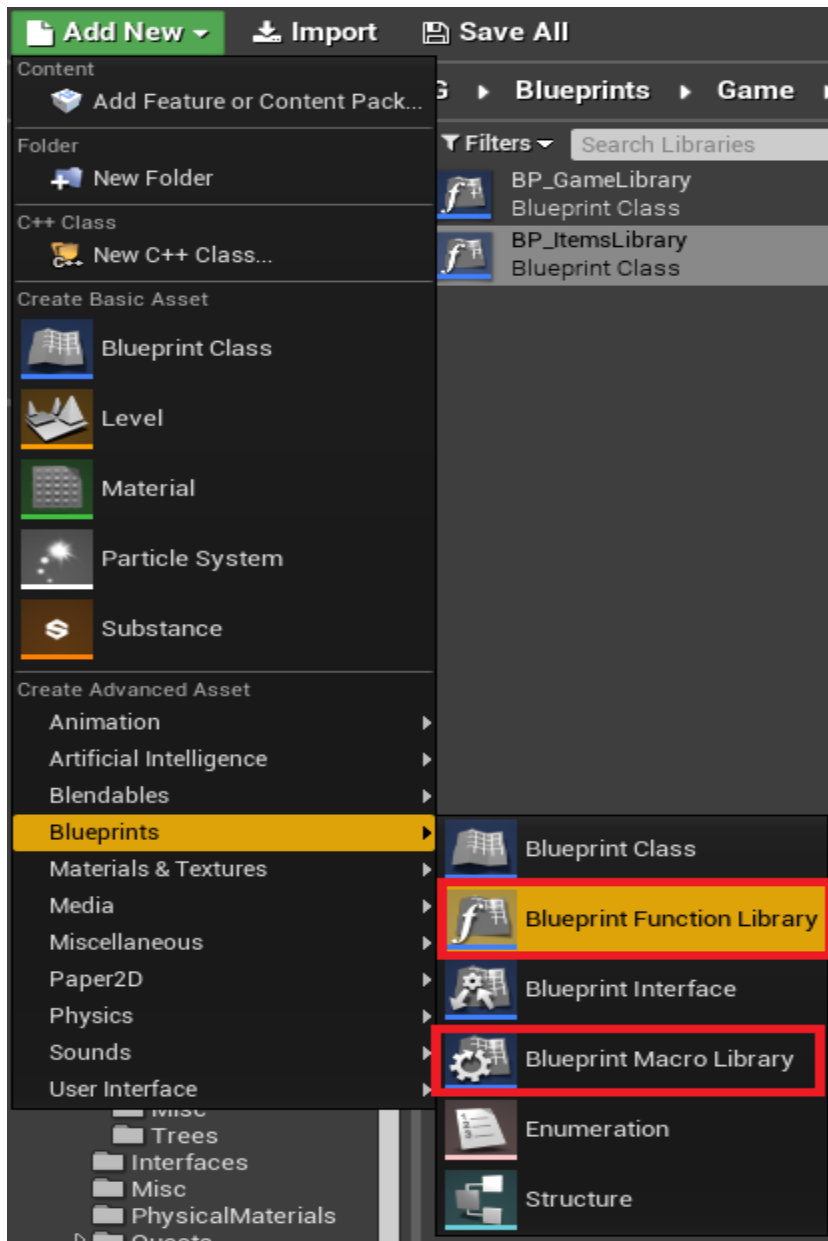


1.4. Work with Libraries

When developing large projects, you have to use the same function sequences over and over again. These sequences can be combined into a library of functions and called in any blueprint of the project.

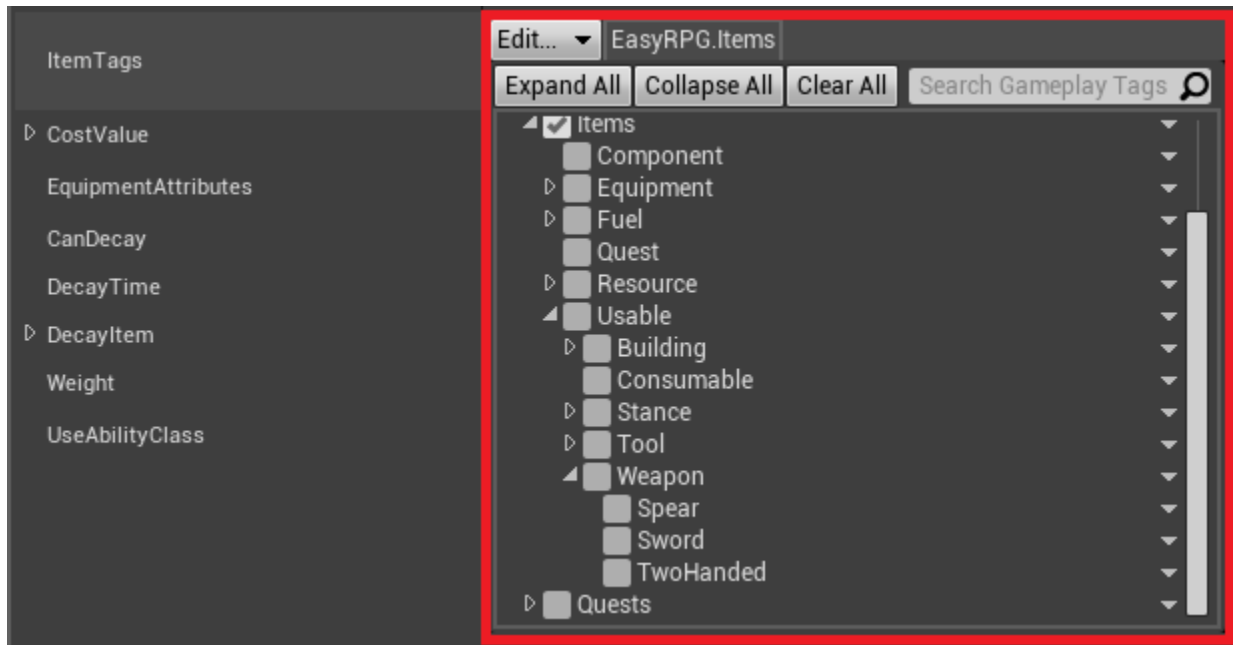
You can create libraries of functions and macros by clicking on the green button. You need to select **Add New** in **Content Browser** and then select **Blueprint Function Library** and

Blueprint Macro Library under Blueprints.



1.5. Work with Gameplay Tags

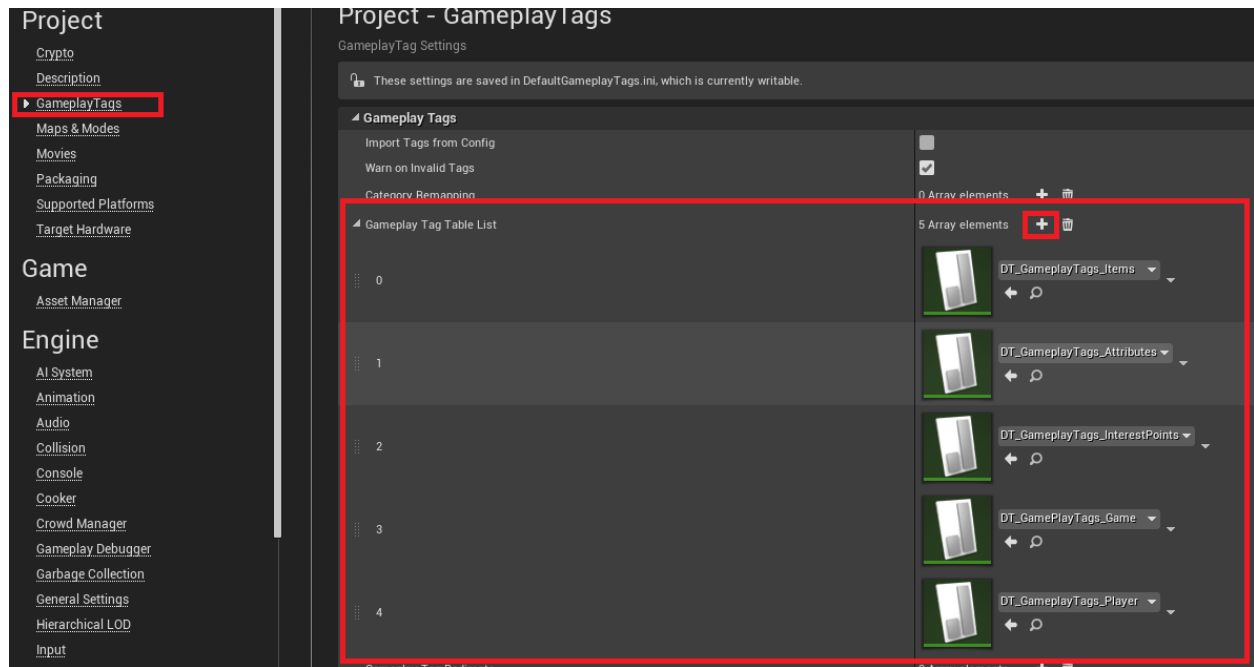
Gameplay Tags can be used for a variety of purposes, ranging from game and weather events to attributes and item types. In **Easy Survival RPG**, they are used to determine the progress of the game for the player and the world as a whole, classify attributes and classify objects



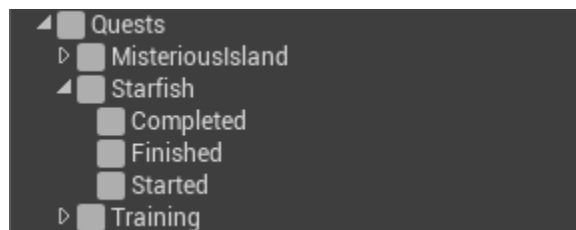
You can create data tables based on **Gameplay Tags** and add new tags to the table as your project develops.

Data Table		
Search		
	Tag	DevComment
EasyRPG.Items.Equipment	EasyRPG.Items.Equipment	Item is equipment.
EasyRPG.Items.Equipment.Head	EasyRPG.Items.Equipment.Head	
EasyRPG.Items.Equipment.Body	EasyRPG.Items.Equipment.Body	
EasyRPG.Items.Equipment.Pants	EasyRPG.Items.Equipment.Pants	
EasyRPG.Items.Equipment.Hands	EasyRPG.Items.Equipment.Hands	
EasyRPG.Items.Equipment.Feet	EasyRPG.Items.Equipment.Feet	
EasyRPG.Items.Equipment.Feet.BootDecal	EasyRPG.Items.Equipment.Feet.BootDecal	
EasyRPG.Items.Quest	EasyRPG.Items.Quest	
EasyRPG.Items.Resource	EasyRPG.Items.Resource	

You can load tag tables in **Project Settings** in the **GameplayTags** section.



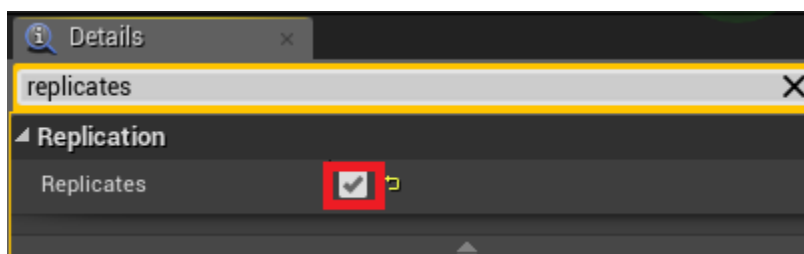
Tags can be used to check certain conditions at a level. For example, in **Easy Survival RPG** they are used for quests. Skeleton Trader will not give you a "**Shadows Mission**" until you complete the "**Starfish Mission**".



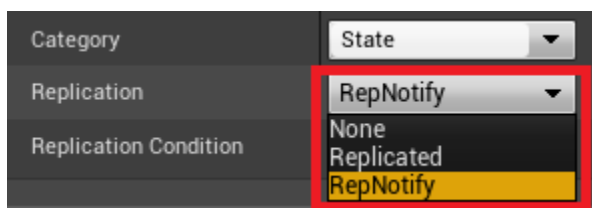
1.6. Networking

Networking in **Unreal Engine 4** is a very complex system, which will take more than a dozen pages to describe, so it is worth knowing the basic concepts of the engine that you need to follow when creating a project.

1. When you start the game, your PC is considered a server by default.
2. When connected to another PC, your PC turns into a client.
3. Objects on the server and clients may differ. When creating a new actor class, you need to specify whether it will be replicated. If replication is enabled, then creating an object on the server, it will be created on clients and the reference to the object will be shared. Only important objects that affect gameplay need to be replicated.



4. Some variables in objects also need to be replicated, otherwise the value of the variables will be different on the server and client. The replicated variable should only be changed on the server.

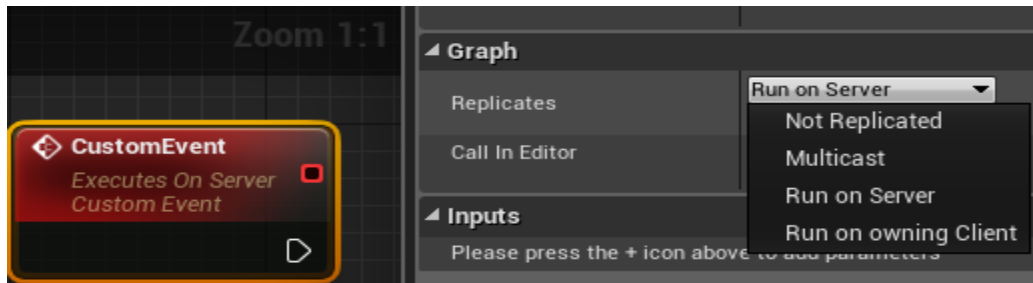


There are two types of replication for variables:

Replicated - the value of the variable is duplicated on clients when it changes on the server.

RepNotify - the value of the variable is duplicated on clients and a function is called, which can be changed at will. For example, this can be used to change the mesh of a character's weapon when the value of a certain variable changes.

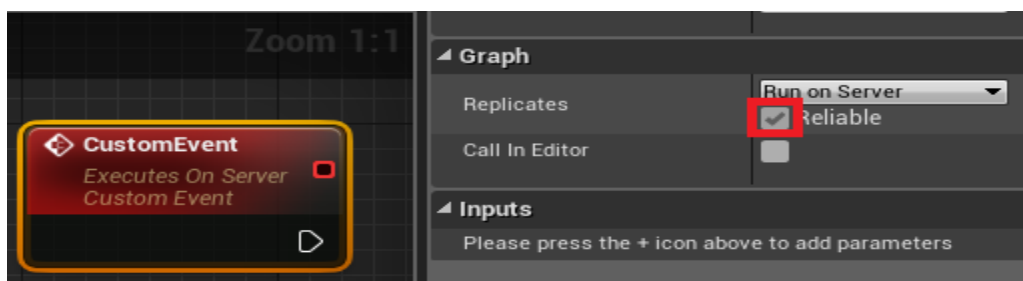
5. Some components must also be replicable if their data and variables are to be duplicated, or they will be used as arguments in server methods.
6. There are three types of server-side methods:



Run on Server - the event will only be triggered on the server machine.

Run on Client - the event will be triggered only on the client machine.

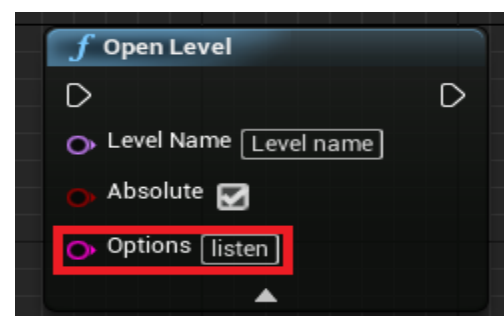
Multicast - the event will be triggered on all machines.



Reliable - the event will happen without fail, even if the network connection is poor.

Can be disabled for some cosmetic events (spawn particles, various effects, and so on, that is, everything that does not affect the main gameplay).

7. Server methods will be called only if called from the **PlayerController** class or called from a class owned by **PlayerController**.
8. For clients to be able to connect to the server, you need to open the level with the **listen** option.



2. Description of the main classes

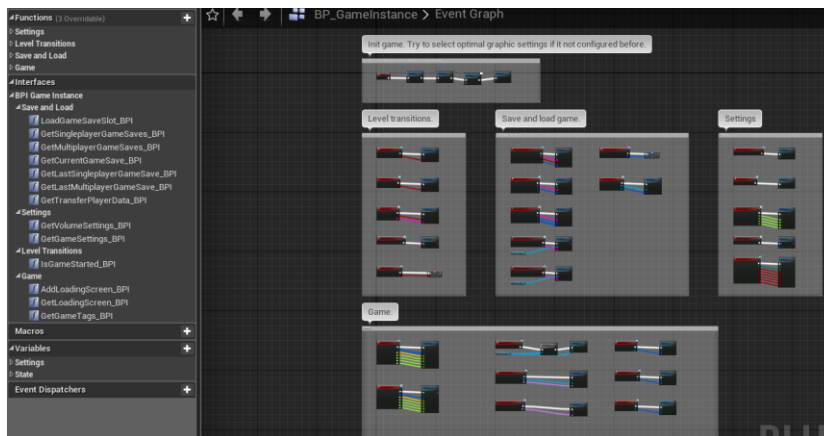
2.1. BP_GameInstance

2.1.1. Description

Inherits from the **GameInstance** class. This class contains all the functions for starting, continuing and loading the game, moving between levels and going to the main menu. Also contains functions for getting and changing custom game settings, graphics, sound and control settings.

In addition, it checks the hardware of the machine and selects the optimal graphics settings.

The class implements the **BPI_GameInstance** interface and can be easily replaced with another **GameInstance**, which also implements the functions of this interface. The functions of this interface can be called on a regular **GameInstance** class obtained using the **GetGameInstance** function.



Global gameplay features are implemented out here, such as spawning items, screen fading and manipulations with global game tags.

The **BP_GameInstance** class also contains functions for initial character customization, such as the starting character class of the player, starting items, equipment, quests,

attributes, and so on. Such functions are used in the main menu when setting up a character and when initializing the game for players.

2.1.2. Variables

2.1.2.1. Game settings variables



SettingsGameSaveSlot - the slot into which the custom game settings are saved.

DedicatedServerSlotName - the slot into which the game session data are saved.

DedicatedAutosaveMode - the autosave mode on the dedicated server.

SingleplayerStartLevel - starting level for single player mode.

MultiplayerStartLevel - starting level for multiplayer mode.

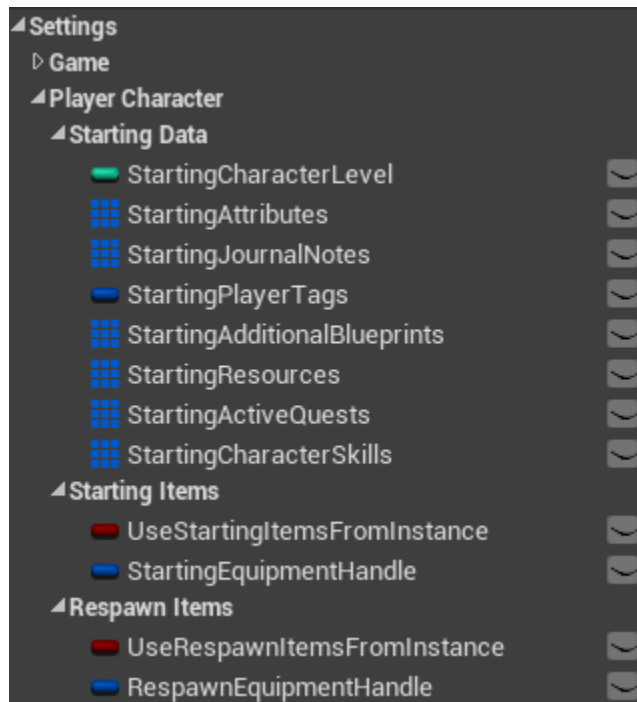
MainMenuLevel - game main menu level.

PlayerCharacterClass - default player character class for spawn and respawn functions.

Used for debugging when playing in the editor.

PlayerCharacterClasses - the list of characters available in the main menu during the initial character customization. If the list is empty the **PlayerCharacterClass** will be used.

2.1.2.2. Player Character settings variables



StartingCharacterLevel - initial level of the player character.

StartingAttributes - initial attributes of the player character.

StartingJournalNotes - initial journal notes for players.

StartingPlayerTags - initial gameplay tags for players.

StartingAdditionalBlueprints - initial additional crafting blueprints for players.

StartingResources - initial resources for players.

StartingActiveQuests - initial quests for players.

StartingCharacterSkills - initial learned skills for players.

UseStartingItemsFromInstance - if true, initial items will be taken from the game instance. Otherwise they will be taken from the selected character class.

StartingEquipmentHandle - the handle for starting equipment for players.

UseRespawnItemsFromInstance - if true, respawn items will be taken from the game instance. Otherwise they will be taken from the selected character class.

RespawnEquipmentHandle - the handle for respawn equipment for players.

2.1.2.3. State variables



IsGameStarted - an indicator that the game was launched from the main menu, and not from the level viewport.

IsMultiplayer - an indicator that the game is launched in multiplayer mode.

CurrentGameSave - the current session save object reference.

SettingsGameSave - the current settings save object reference.

CurrentLevel - name of the current level.

TransitActive - an indicator that a player character should be spawned in a specific transit point after level loading.

TransitPoint - a tag of the actor which should be a specific transit point on the loading level.

TransferPlayerData - starting player data of the player character when switching from the main menu to the game.

LoadingScreen - loading screen widget reference.

GameTags - list of current global game tags of the game.

2.1.3. Functions

Settings - the category of functions which are used for working with game settings.

Includes functions for changing and applying game settings. Also includes functions for checking hardware and selecting optimal settings for the current PC.

Level Transitions - the category of functions which are used for working with level transitions. Includes functions for starting and loading levels in singleplayer or multiplayer game modes.

Save and Load - the category of functions which are used for working with a save-load system. Includes function for saving or loading data of the level to or from the specific game save slot.

Game - the category of the functions which are used for working with the game world. Includes functions for spawning items in the world, for changing global game tags and so on.

2.2. BP_PlayerController

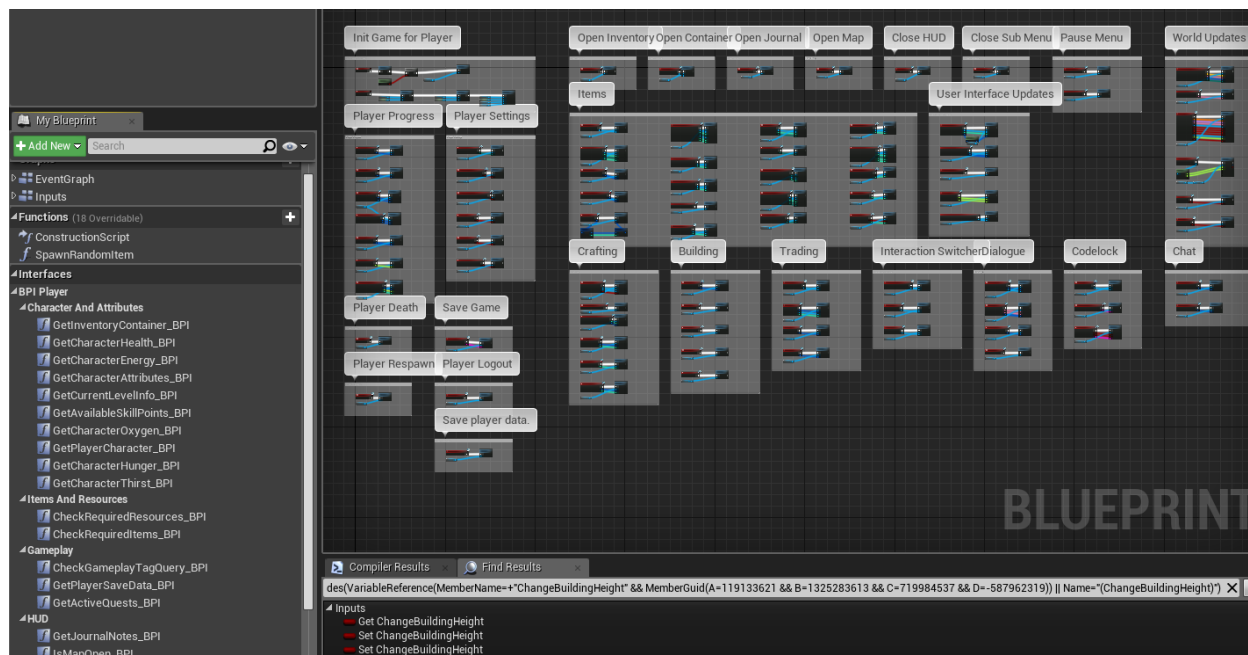
2.2.1. Description

Inherits from the **PlayerController** class. The class contains the

BP_PlayerManagerComponent control component, which includes a huge number of functions for interacting with objects, resources, crafting and other objects, as well as functions for managing the user interface. It also contains a component for building **BP_BuildComponent**, which contains functions for managing the building process and a component for interacting with **BP_InteractionComponent** objects. More details about these components are described in the corresponding sections.

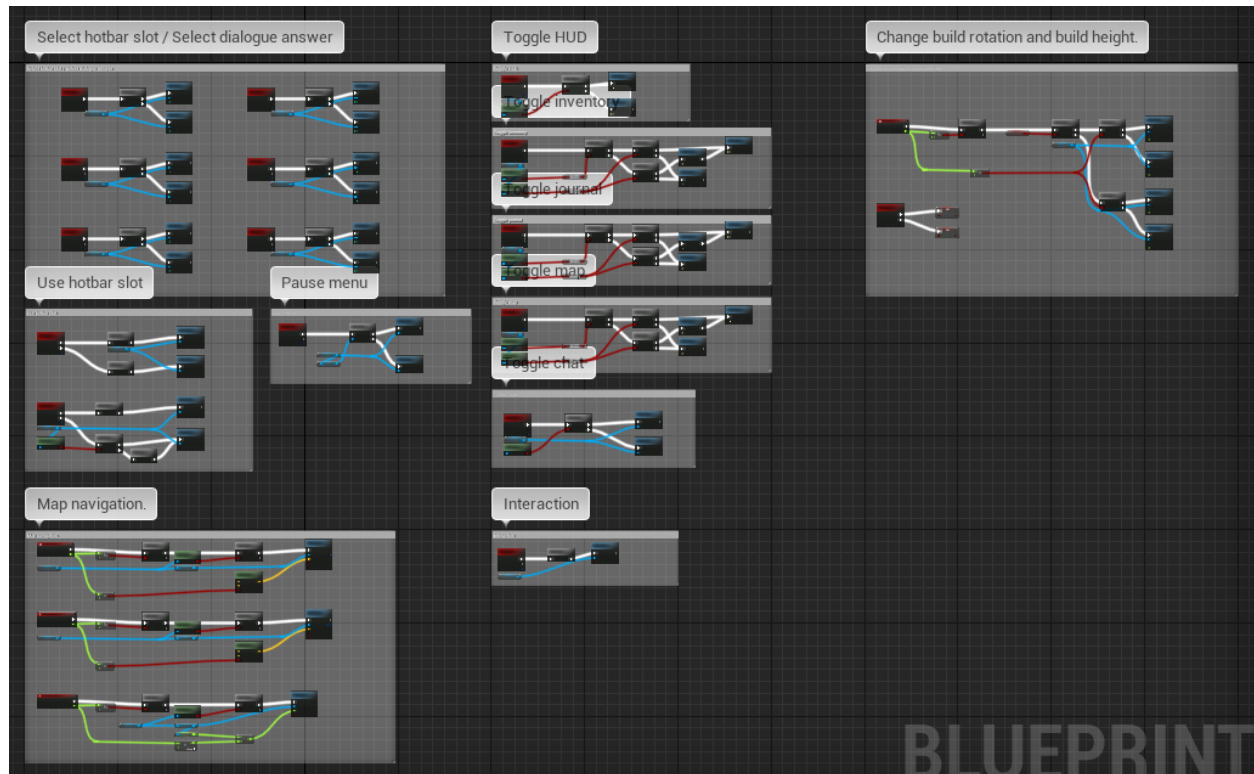
The class implements the **BPI_Player** interface and can be replaced with another **PlayerController**, which also implements the functions of this interface.

Functions of this interface can be called on the **Controller** class without being cast to the appropriate type. All these functions are in the **EventGraph** graph.



Also here are the functions for initializing the game for the player and the function for initializing all components.

The events of pressing the control buttons are in the **Inputs** graph. Such events include choosing an active hotbar slot, opening and closing inventory, and others.



2.2.2. Macros



IsGameInputs - continues the sequence of actions if the user interface is closed.

IsGameOrDialogue - selects a sequence of actions depending on whether the dialog is open or not.

2.2.3. Variables

▲ State

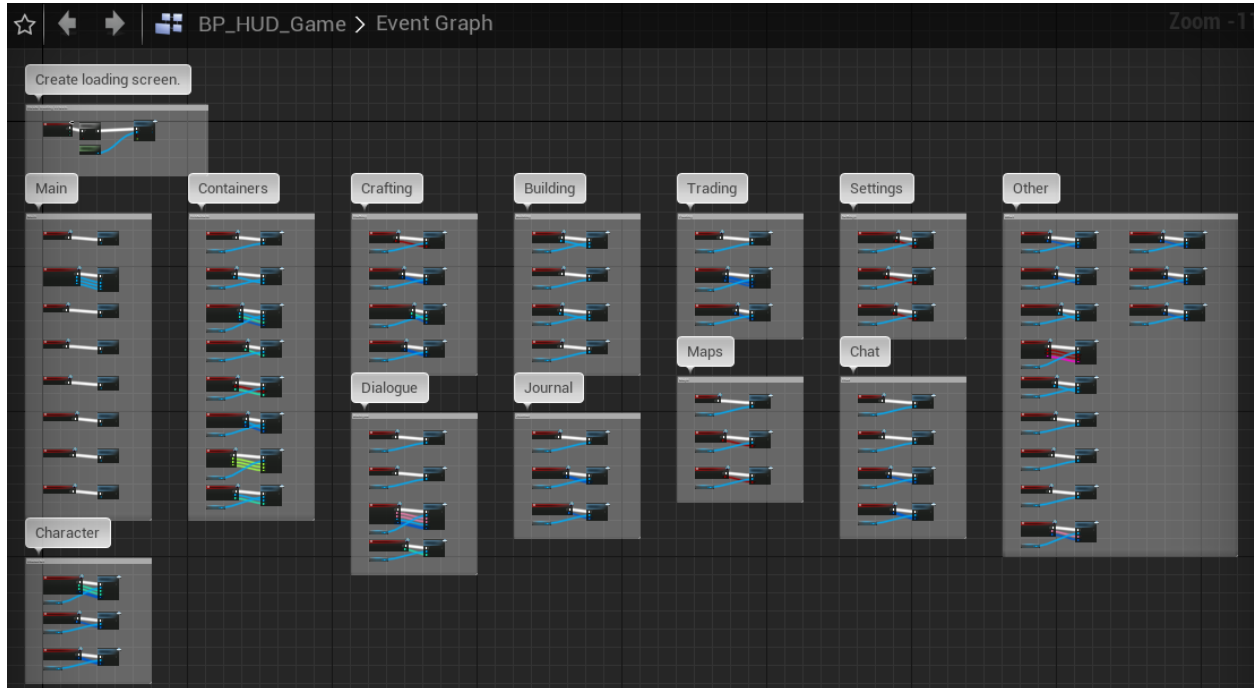
ChangeBuildingHeight

ChangeBuildingHeight - an indicator at which it is necessary to change the height of the object under the building using the mouse wheel.

2.3. BP_HUD_Game

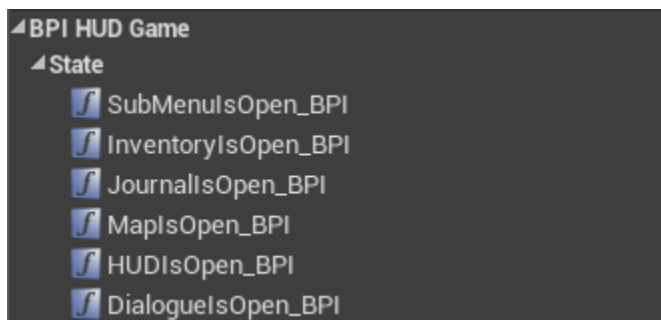
2.3.1. Description

Inherits from the **HUD** class. Contains functions for creating, updating and managing the user interface, which includes game widgets and widgets for the pause menu and the character death menu.



The class implements the **BPI_HUD_Game** interface and can be easily replaced with another **HUD**, which also implements the functions of this interface.

The functions of this interface can be called on the **HUD** class obtained using the **GetHUD** function. Most of the **BPI_HUD_Game** interface functions are called directly from the **HUD** widget. Implementation of such functions should be in the **HUD** widget.

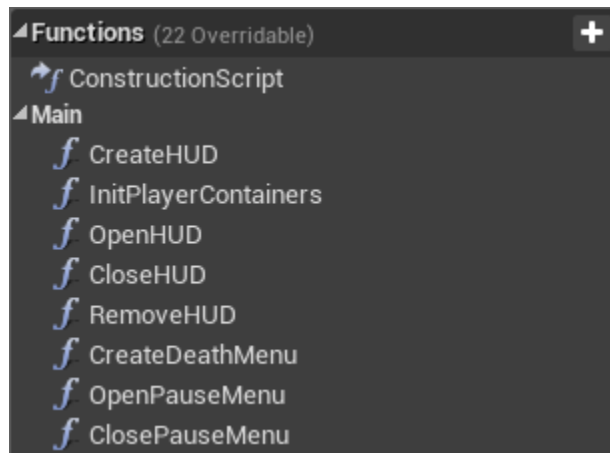


2.3.2. Variables



- **HUDWidgetClass** - class of the **HUD** widget that will be used by players for the game.
- **DeathMenuWidgetClass** - class of the death menu widget that will be used by players when the player's character dies during the game.
- **PauseMenuWidgetClass** - class of the pause menu widget that will be used by players for pauses during the game.
- **HUD** - reference to main **HUD** widget.
- **PauseMenu** - reference to pause menu widget.
- **DeathMenu** - reference to death menu widget.

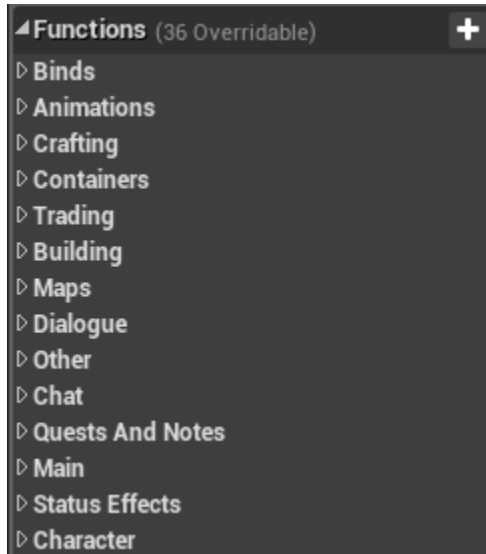
2.3.3. Functions



- **CreateHUD** - Create HUD widget and add it to the viewport.
- **InitPlayerContainers** - Init player inventory, player hotbar and player equipment widgets and create their item slots.
- **OpenHUD** - Open HUD widget.
- **CloseHUD** - Close HUD widget.
- **RemoveHUD** - Remove HUD widget from the viewport.
- **CreateDeathMenu** - Create a death menu widget and add it to the viewport.
- **OpenPauseMenu** - Create a pause menu widget and add it to the viewport.
- **ClosePauseMenu** - Remove pause menu widget from the viewport.

2.3.4. UI_HUD

This widget class has a huge number of functions for updating and managing the user interface. Function names convey a general idea and are sorted into several categories.



- **Binds** - the category of functions for updating binded variables, such as visibility of certain widgets.
- **Animations** - the category of functions for playing different reaction animations.
- **Crafting** - the category of functions for interactions with crafting interfaces.
- **Containers** - the category of functions for updating slots of inventory, equipmentt and hotbar, as well as filling them with items.
- **Trading** - the category of functions for interaction with trading interfaces.
- **Building** - the category of functions for interaction with the building menu and with the mallet menu.
- **Maps** - the category of functions for interaction with map and minimap.
- **Dialogue** - the category of functions for managing and updating dialogue widgets.
- **Other** - the category for other functions which are not included in any category. For example, these are the functions of opening a code lock, displaying pickup items, playing animations when receiving damage or blocking during a battle, and others.

-
- **Chat** - the category of functions for opening and closing chat and functions for updating messages.
 - **Quests and Notes** - the category of functions for updating the log and active quests.
 - **Main** - the category of functions for creating, opening and closing the user interface, as well as other functions - functions for creating a pause menu and a death menu.
 - **Status Effects** - the category of functions for updating actual status effects on the HUD.
 - **Character** - the category of functions for updating player character information.

2.4. BP_GameSave_Settings

2.4.1. Description

Inherits from **SaveGame** class. This is a class used to save and load game settings and to save and load slots in saved games. Loaded when the game starts from the **BP_GameInstance** class.

2.4.2. Variables

HardwareBenchmarkUsed - an indicator that the optimal graphics settings have already been set.

AutosaveMode - current game autosave settings mode.

ShowFloatingText - indicator showing pop-up text during game.

ShowCharacterState - displaying the state of the game character on the user interface during the game.

ShowEnemyCharacterStates - displaying the status of other characters during the game.

ShowMinimap - displaying the minimap on the user interface during the game.

ShowMarks - displaying the mark while playing.

ShowBuildingDurability - displaying the status of buildings during the game.

MasterVolume - the value of the overall volume of sounds.

MusicVolume - music volume value.

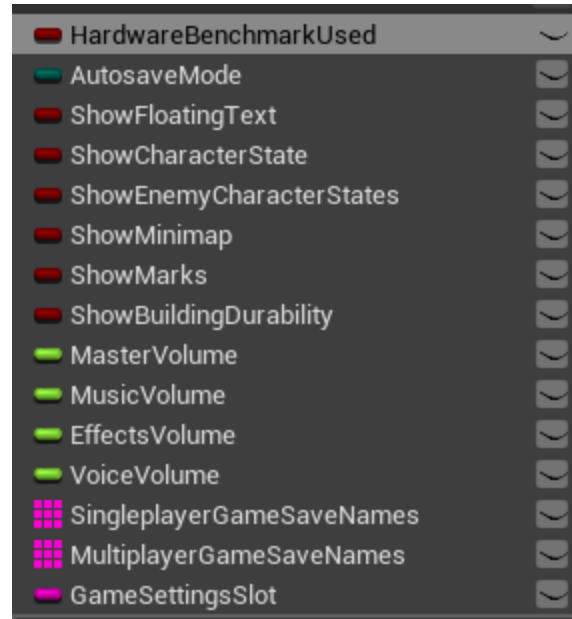
EffectsVolume - effects volume value.

VoiceVolume - voice/dialogues volume value.

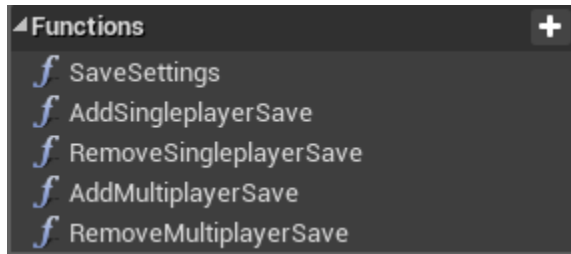
SingleplayerGameSaveNames - array of slots in which the game is saved in single player mode.

MultiplayerGameSaveNames - an array of slots in which the game is saved in multiplayer mode.

GameSettingsSlot - the slot in which the data of this class is stored.



2.4.3. Functions



SaveSettings - saving settings to a slot for saving game settings.

AddSingleplayerSave - adding a slot to the array of saved games in single player mode.

RemoveSingleplayerSave - removing a slot from an array of saved games in single player mode.

AddMultiplayerSave - removing a slot from an array of saved games in single player mode.

RemoveMultiplayerSave - removing a slot from the array of saved games in multiplayer mode.

2.5. BP_GameSave_Session

2.5.1. Description

Inherits from **SaveGame** class. Class with saved information about the game session. Used in **BP_GameInstance** to save the current game session, load and save data about the level and players.

Implements the **BPI_GameSave** interface and its functions. These functions are called by all actors and components that need to be saved and loaded. The working principle of the save and load system is explained in more detail in the following sections.

2.5.2. Variables

SlotName - the name of the slot where the save is stored.

CurrentLevel - the current level name.

TransitActive - an indicator that a player character should be spawned in a specific transit point after level loading.

TransitPoint - a tag of the actor which should be a specific transit point on the loading level.

IsMultiplayer - an indicator that the game session is running in multiplayer.

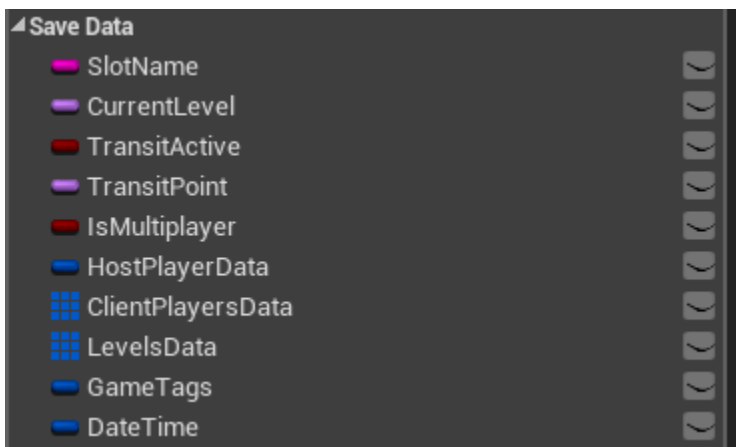
HostPlayerData - saved data about the player who is the host.

ClientPlayersData - saved data of players.

LevelsData - saved/stored level data.

GameTags - saved game tags.

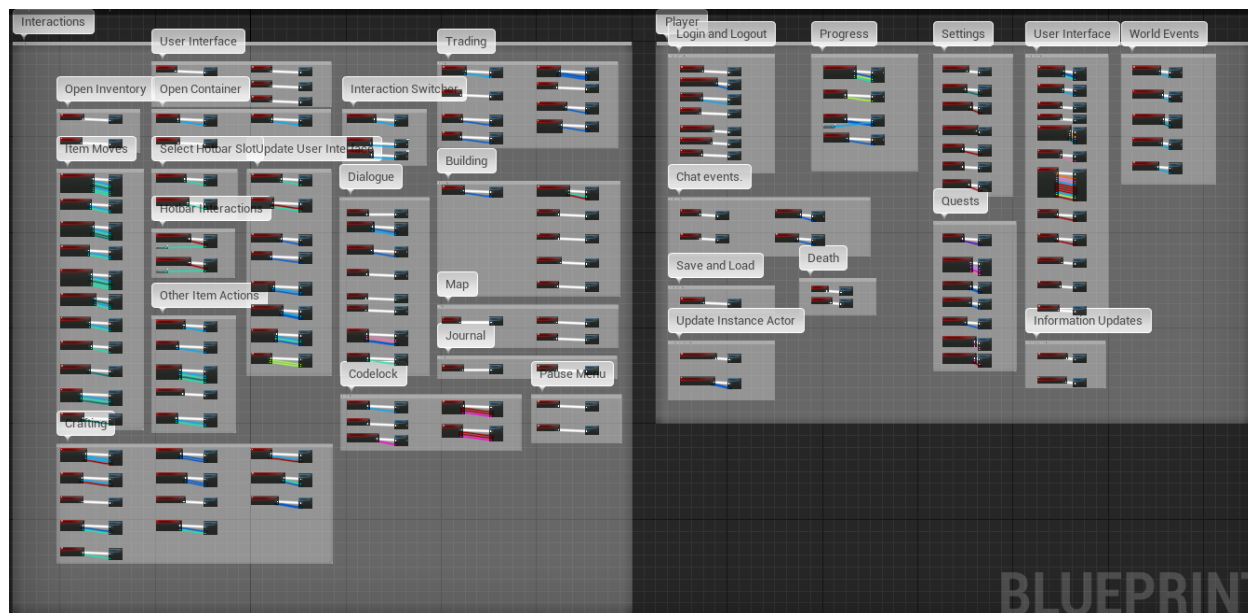
DateTime - time and date of the save.



2.6. BP_PlayerManagerComponent

2.6.1. Description

Inherits from the **ActorComponent** class. This component processes for the player the interaction of all the main systems of the project with each other in one place. Such systems include: the system for initializing the game character, the system for interacting with objects, the building system, the system for interacting with the user interface, the system for loading and saving the game, and many others. Contains all network functions that the **PlayerController** must contain in order for all systems in the project to interact in multi-user mode. It also contains information about the progress of the player, about the character's level, about the experience earned, about the available recipes for crafting, about active quests and much more. The component must replicate.



The component must be added to the **PlayerController** class and initialized there as well, along with the construction component (**BP_BuildComponent**) and the interaction component (**BP_InteractionComponent**).

2.6.2. Variables

2.6.2.1. References variables

PlayerController - reference to the player.

InteractionComponent - reference to the object interaction component.

BuildComponent - building component reference.

HUD - player UI reference.

PlayerCharacter - player character reference.

PlayerInventory - reference to the game character's inventory container.

LootContainer - open container reference.

PlayerEquipment - reference to the game character's equipment container.

PlayerHotbar - reference to the game character's equipment container.

TradeComponent - trader component reference.

CharacterCraftingComponent - reference to the crafting component of the game character.

CurrentCraftingComponent - reference to the current crafting component.

CurrentDialogue - active dialogue actor reference.

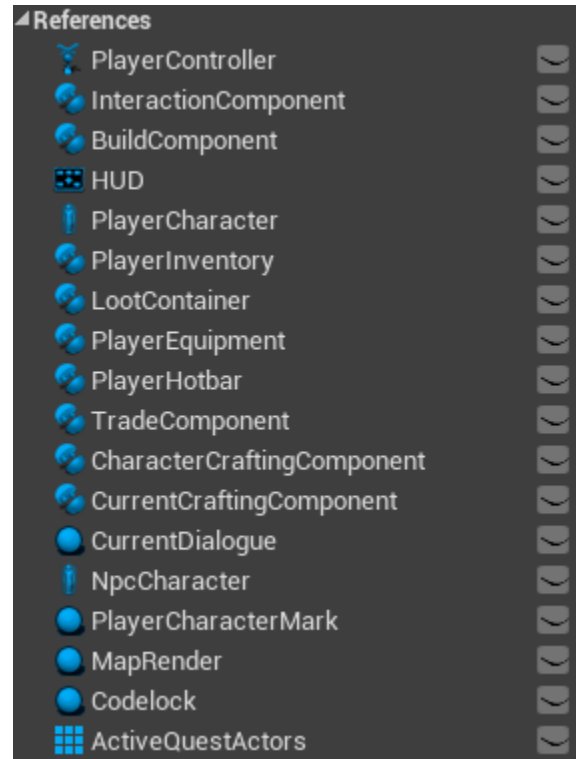
NpcCharacter - reference to the character with whom the dialogue is taking place.

PlayerCharacterMark - game character label actor reference.

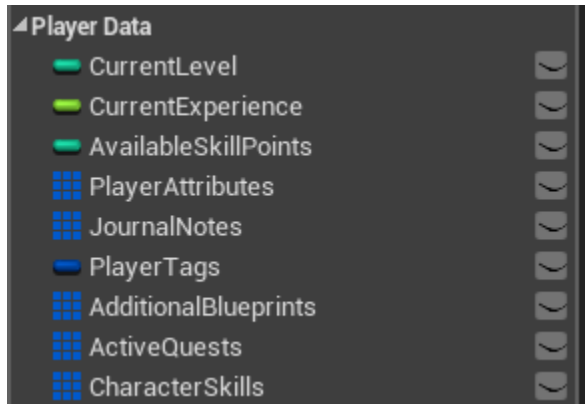
MapRender - reference to the actor that renders the map.

CodeLock - reference to active code lock.

ActiveQuestActors - an array of references to active quest actors.



2.6.2.2. Player Data variables



CurrentLevel - current character level.

CurrentExperience - current character experience.

AvailableSkillPoints - number of available skill points.

PlayerAttributes - array of player attributes.

JournalNotes - array of journal notes.

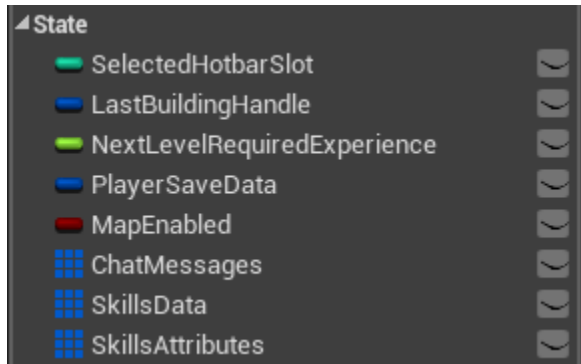
PlayerTags - active player tags.

AdditionalBlueprints - additional crafting blueprints available to the player.

ActiveQuests - array with data about active quests.

CharacterSkills - learned character skills.

2.6.2.3. State variables



SelectedHotbarSlot - active hotbar slot.

LastBuildingHandle - ID of the last placed building object.

NextLevelRequiredExperience - experience needed for the next character level.

PlayerSaveData - saved data of the player.

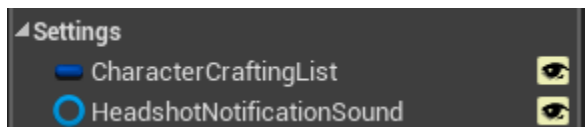
MapDisable - the indicator of the player's map being disabled.

ChatMessages - an array of messages in the player's chat.

SkillsData - learned skills data.

SkillsAttributes - attributes which are granted by learned skills.

2.6.2.4. Settings variables



CharacterCraftingList - list of crafting blueprints initially available from the start.

HeadshotNotificationSound - the sound which plays when you hit an enemy in head.

2.6.3. Functions

This component contains a huge number of functions that are sorted into different categories. The functions also have the **server** and **client** postfixes. **Server** means the function is executed on the **server** and **client** means the function is executed on the **client**. Description of each function can be found by hovering over its name.

Init Component - component initialization function.

User Interface - functions for creating, updating and managing the user interface.

Items - functions of quick interaction with objects, drag and drop between container slots, dropping, etc.

Hotbar - functions of interaction with objects in the Hotbar container.

Containers - functions for interacting with containers, displaying containers, updating slots, etc.

Crafting - functions of interaction of components during crafting, displaying a crafting window and updating crafting progress.

Trading - functions of interaction of components when trading, displaying a trading window and updating trading items.

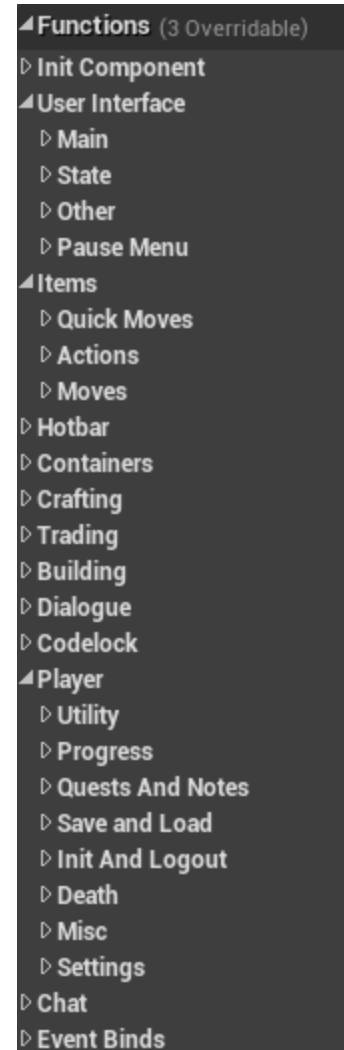
Building - functions of interaction of components during build.

Dialogue - functions for interacting with dialogues, displaying a dialogue window, selecting dialogue replicas, etc.

Codelock - functions for interacting with a code lock, code lock window display and code lock authorization.

Player - functions for initializing the player and the game character, updating progress, quests, saving and loading player data, changing settings, etc.

Chat - functions for displaying and sending chat messages.



Event Binds - functions for in-game events, for example, changing items in the inventory, chopping trees or killing other characters.

2.6.4. Event Dispatchers

OnSelectedItemChanged - occurs when an item in the active quick slot changes.

OnInventoryItemChanged - occurs when an inventory item changes.

OnPlayerTagsUpdated - occurs when player tags have been updated.

OnPlayerItemsChanged - occurs when an item in an inventory, hotbar or equipment slot changes.

OnPlayerDestructTree - occurs when a player chops down a tree.

OnPlayerFullDestructTree - occurs when the player chops down a tree stump.

OnPlayerCompleteCrafting - occurs when a player finishes crafting an item.

OnPlayerCompleteBuilding - occurs when the player completes building.

OnPlayerDestructMineStage - occurs when a player destroys a piece of mine vein.

OnPlayerDestructMine - occurs when a player completely destroys a mine vein.

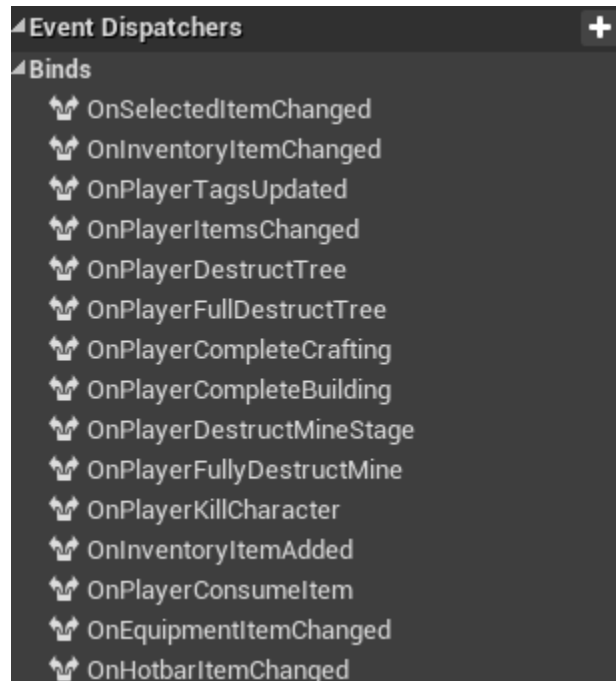
OnPlayerKillCharacter - occurs when a player kills a character.

OnInventoryItemAdded - occurs when an item is added to a player's inventory.

OnPlayerConsumeItem - occurs when a player consumes an item.

OnEquipmentItemChanged - occurs when an item of equipment changes.

OnHotbarItemChanged - occurs when an item in the hotbar changes.

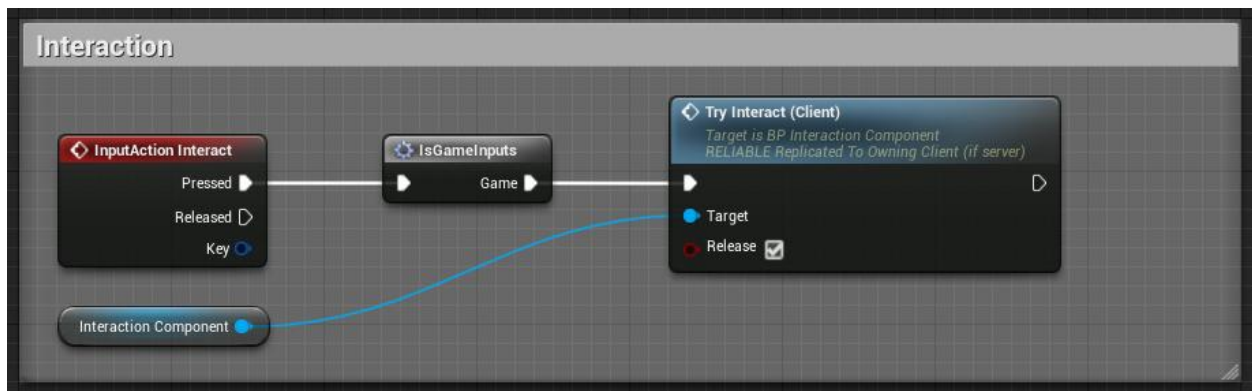


3. Description of the main systems

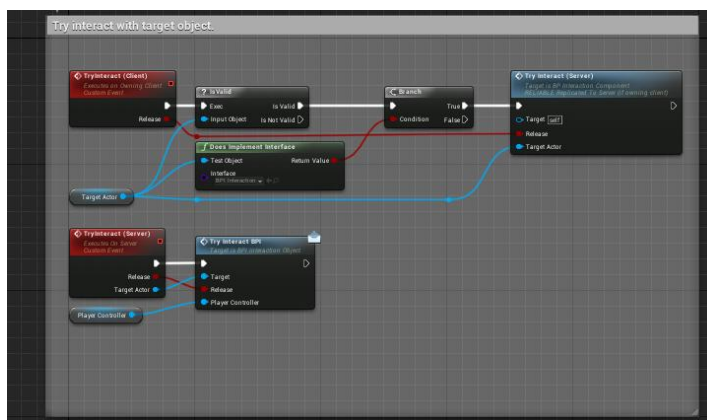
3.1. Object interaction system

3.1.1. Description

The player can interact with objects in the game world by pressing buttons to interact. To interact with an object, it must implement the **BPI_InteractionObject** interface. The search for an object with which the player can interact is carried out using the **BP_InteractionComponent**. An attempt to interact with the object occurs after the player presses a certain button and calls the **TryInteract (Client)** function from the component.

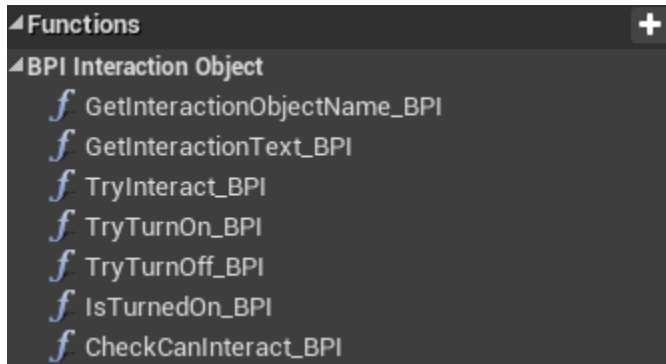


The check for the presence of a target is performed on the client's machine, and if there is a suitable target, the component calls the interaction function for the object on the server.



3.1.2. BPI_InteractionObject

Objects that the player can interact with must have the following functions implemented.



GetInteractionObjectName_BPI - returns the name of the object.

GetInteractionText_BPI - returns a description of the interaction process.

TryInteract_BPI - trying to interact with an object.

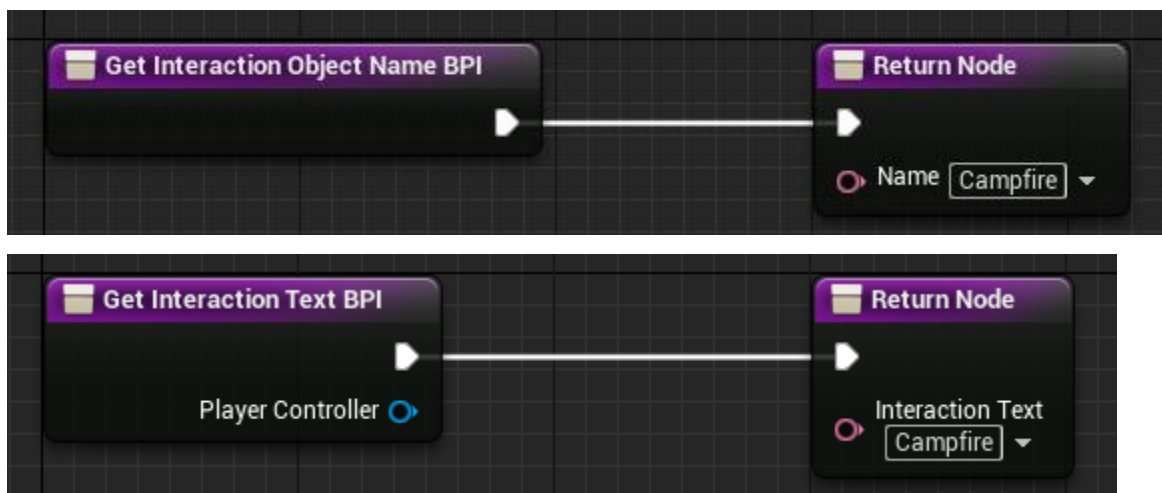
TryTurnOn_BPI - trying to turn on an object.

TryTurnOff_BPI - trying to turn off an object.

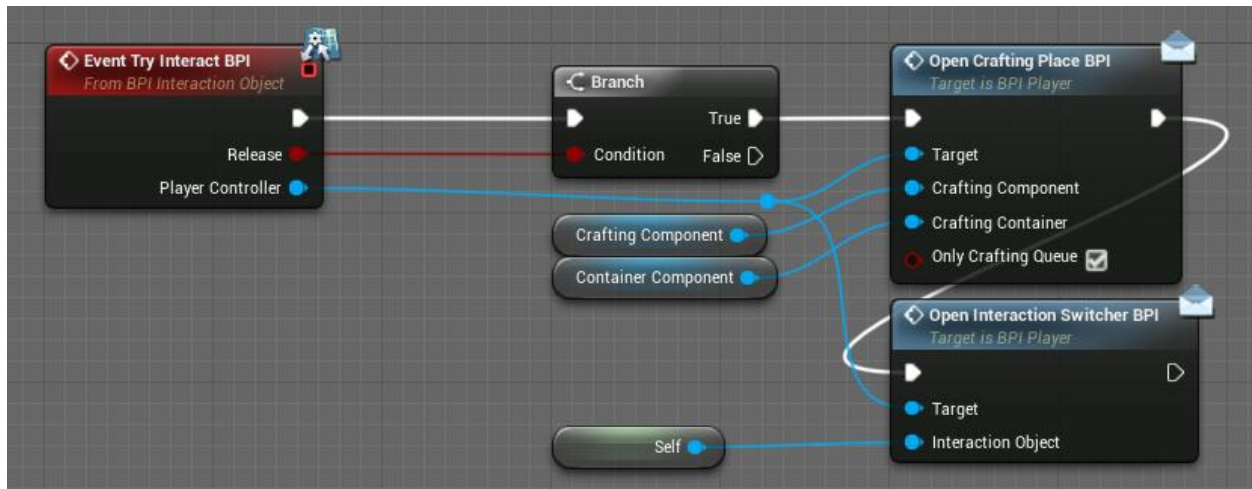
IsTurnedOn_BPI - returns true if the object is currently on.

CheckCanInteract_BPI - returns true if the object can be interacted with at the moment.

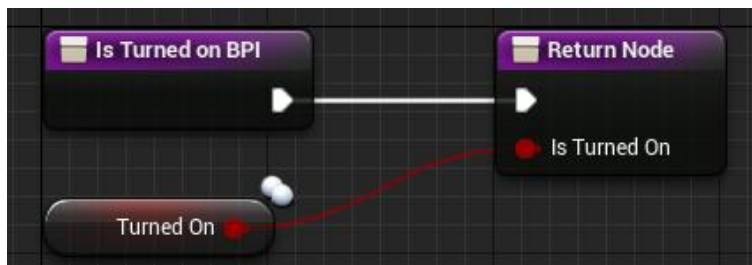
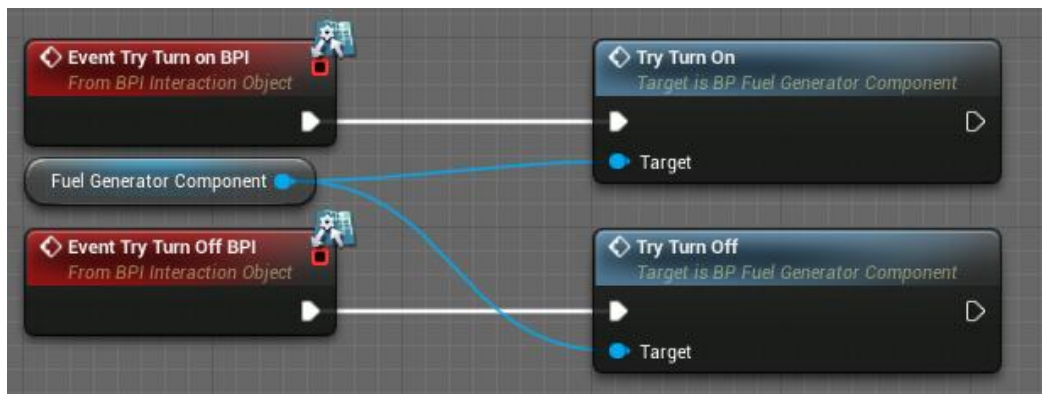
Representation of functions on the example of interaction with a campfire.



When interacting with a campfire, a container opens with a production queue and with a switch for controlling the combustion process.



The combustion process is controlled by a component **BP_FuelGeneratorComponent**.



3.1.4. Variables

PlayerController - player controller reference.

TargetActor - the target interaction actor.

IsLocalPlayer - an indicator that the component is in the possession of the local player.

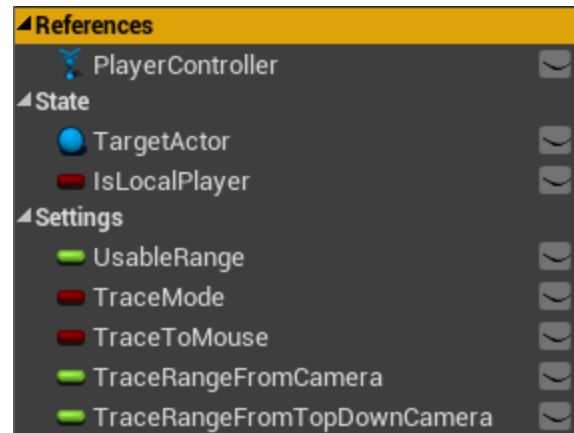
UsableRange - range of interaction from the character.

TraceMode - an indicator that the search for the target actor is performed using a trace.

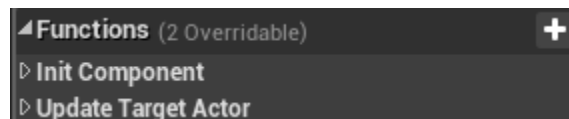
TraceToMouse - an indicator that the trace will be called at the position of the player's mouse cursor.

TraceRangeFromCamera - distance of the trace relative to the player's camera.

TraceRangeFromTopDownCamera - distance of the trace relative to the player's camera in top view mode.



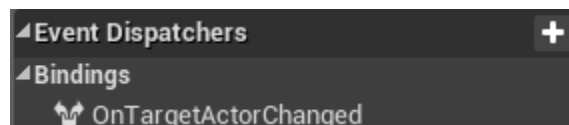
3.1.5. Functions



Init Component - a component initialization function for the player.

Update Target Actor - search and update functions of the target actor.

3.1.6. Event Dispatchers



OnTargetActorChanged - occurs when the target actor changes to another.

3.2. Items interaction system

3.2.1. Description

The system for working with items in a multiplayer game is a very complex and advanced one. It uses a large number of structures, data tables and interactions between multiple components and actors.

To store information about unique items, the **STR_ItemInstance** structure and the **DT_Items** data table made on its basis are used. The custom items data table also can be created and used in blueprints or in loot lists the same as **DT_Items**.

The **BP_Item** actor is used to place items on the level and when dropping them out of inventory.

The **STR_ItemData** structures are used to store information about items in containers. These structures are stored in the **BP_ContainerComponent**.

The player's equipment is represented by the **BP_EquipmentComponent** component, which inherits from the **BP_ContainerComponent** component.

The player's hotbar is represented by the **BP_HotbarComponent** component, which inherits from the **BP_HotbarComponent** component.

To store information about unique crafting blueprints, the **STR_Blueprint** structure and the **DT_Blueprints** data table made on its basis are used. To store different lists of crafting blueprints, the **STR_CraftingList** structure and a table based on it **DT_CraftingLists** have been created.

Crafting items using **BP_CraftingComponent**.

To store lists of items available for trade, use the **STR_TradeList** structure and the **DT_TradeLists** data table created on its basis.

Trading is carried out using the **BP_TradeComponent**.

The **BP_ItemsLibrary** library of functions is used to work with items. If you want to add new items tags don't forget to change specific functions in the library.

3.2.2. Structures

Several structures were made to work with items.

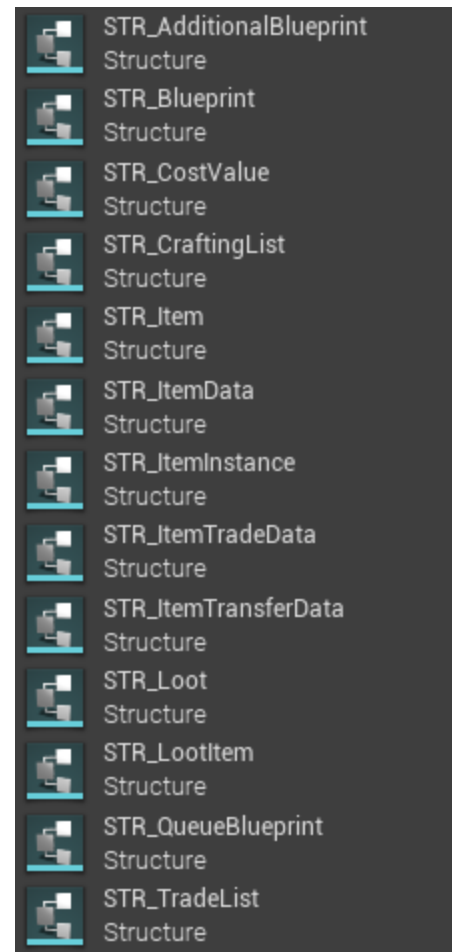
Structures **STR_Item**, **STR_ItemData**, **STR_ItemInstance**, **STR_ItemTransferData** are created to store and transfer information about items.

The **STR_ItemInstance** structure stores information about the item, index, name, description, item icon, static & skeletal meshes, etc. Based on this structure, the **DT_Items** data table information about all the project items.

The **STR_Item** structure stores information about the item index from the **DT_Items** table and its quantity.

The **STR_ItemData** structure stores complete information about an item located in the world or in a container slot.

The **STR_ItemTransferData** structure is created to transfer complete information about an item between the server and the client



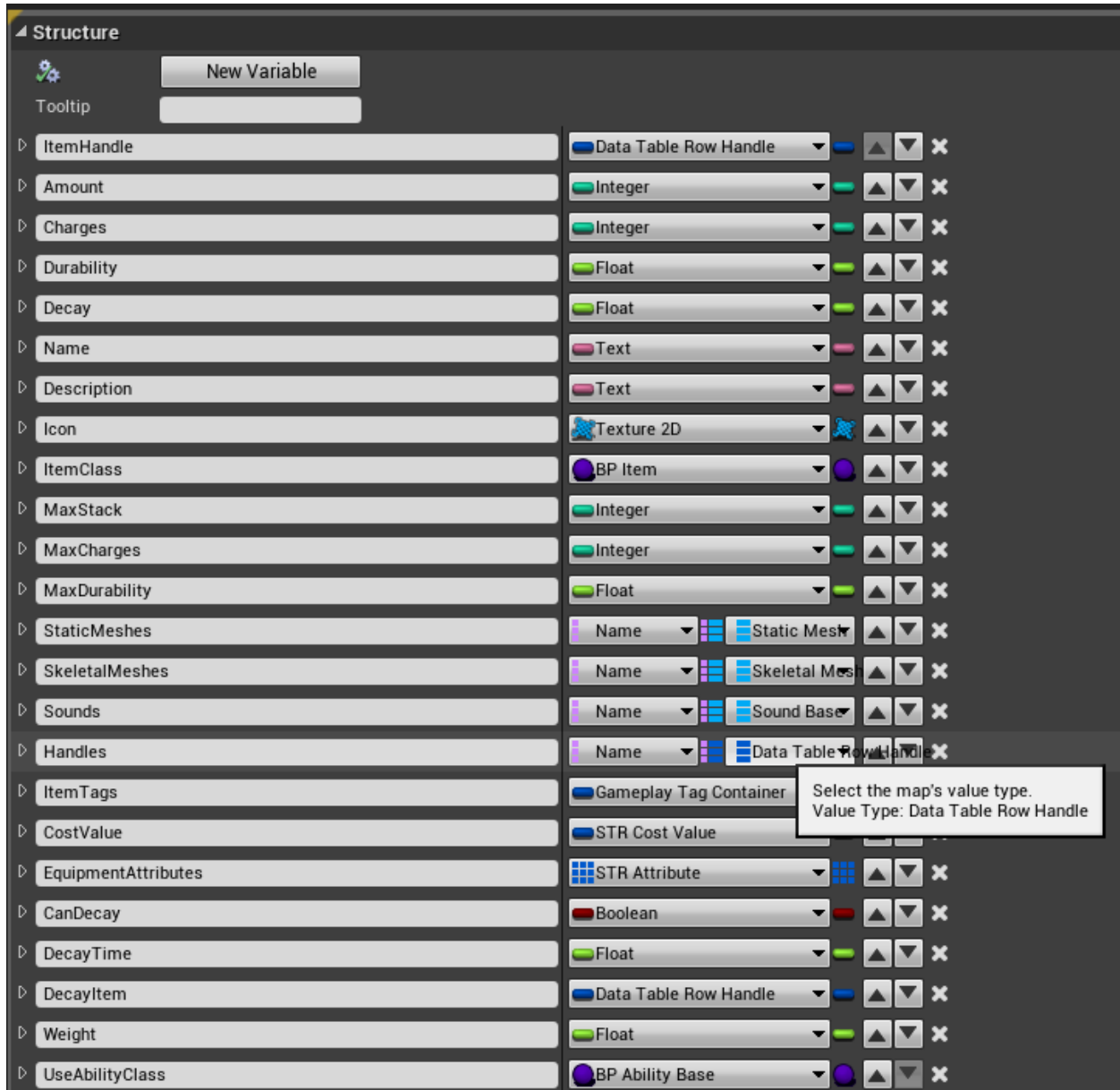
Structures **STR_Blueprint**, **STR_CraftingList**, **STR_QueueBlueprint**, **STR_AdditionalBlueprint** are created to store information about crafting blueprints for the crafting system.

Structures **STR_ItemTradeData**, **STR_CostValue**, **STR_TradeList** are created to work with the trading system.

Structures **STR_ItemTradeData**, **STR_CostValue**, **STR_TradeList** are created to work with the trading system.

3.2.3. Overall item information

Overall information about an item in the world or a container slot is stored in the **STR_ItemData** structure.



ItemHandle - item id in **DT_Items** data table.

Amount - the current amount of the item in the stack.

Charges - the current amount of charges in the item.

Durability - current value of item durability.

Decay - current decay value of the item.

Name - item name.

Description - item description.

Icon - item icon.

ItemClass - world actor class.

MaxStack - maximum number of items in a stack.

MaxDurability - maximum item durability.

StaticMeshes - the list of static meshes used by the item. For example, the "**Drop**" index is used to display the mesh of an item when it is dropped from the inventory.

SkeletalMeshes - the list of skeletal meshes used by the item.

Sounds - the list of sounds used by the item. For example, the "**Take**" index is used to play a sound when it is added to inventory.

Handles - the list of handles which are used for connecting item data with other data tables, like building data, usable item data or ability data.

ItemTags - list of item tags, to determine the type of item.

CostValue - structure that determines the value of an item when it is sold to a trader.

EquipmentAttributes - attributes of an item added to a player character if the item is equipped or selected in the player hotbar.

CanDecay - an indicator of whether a given item can decay over time.

DecayTime - decay time of the item in seconds.

DecayItem - an item that this item will transform into after decay.

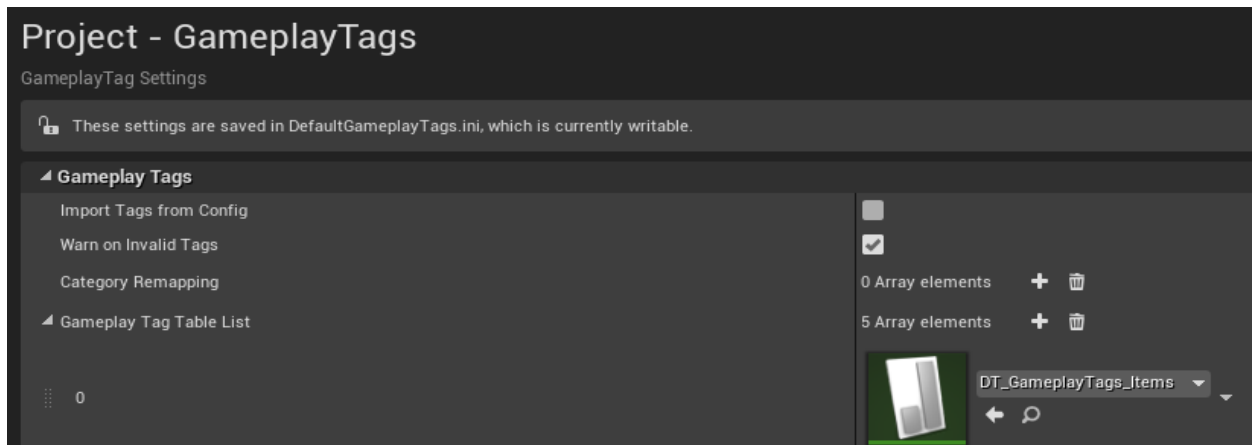
Weight - weight of one item in a stack.

UseAbilityClass - the actor class that will be created and activated when using the item.

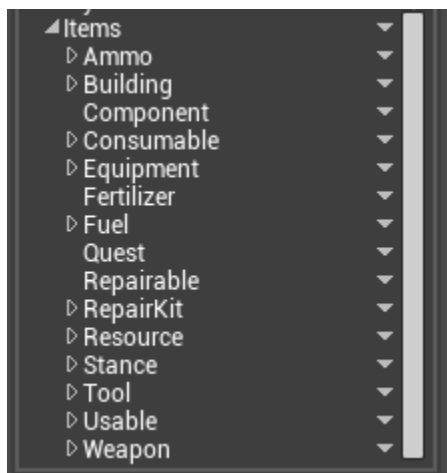
3.2.4. Classification of items

Item classification is implemented using the game tag system.

Tags for items were registered in the **DT_GameplayTags_Items** table, and then added to the project settings.



Different tags and classifications can be chosen for each item.



Ammo - an indicator that the item is an ammo or uses a specific ammo type as ammo.

Building - an indicator that the item is a building object which can be placed in the world.

The building object can be set in the **Handles** variable of the item with the **Building** name in the **DT_Items** or in your own custom items data table.

Building.BuildingPlan - an indicator that the item is a building plan and has its own building menu where you can select the building object which you want to place.

The building list can be set in the **Handles** variable of the item with the **BuildingList** name in the **DT_Items** or in your own custom items data table.

Component - an indicator that the item can be made into something else.

Equipment - an indicator that the item is equipment and used in a specific equipment slot.

Equipment.Feet.BootDecal - an indicator that the item has a specific boot decal which renders in the footstep system.

Fertilizer - an indicator that the item is fertilizer and can be used in the crop plots for increasing the plant's growing factor.

Fuel - an indicator that the item is used as fuel.

Quest - an indicator that the item is a quest item.

Repairable - an indicator that the item can be repaired by repair kits.

RepairKit - an indicator that the item is a repair kit which can restore the durability of other items. The repair value can be set in the **Durability** variable of the repair kit item.

RepairKit.Ratio - an indicator that the repair kit restores the part of durability of other items instead of specific value.

Resource - an indicator that the item is a resource and can stack in the resources variable of the container component.

Stance - an indicator that the item has a specific stance type which is used for selecting the animation stance and locomotion of the character who uses the item.

Tool - an indicator that the item is a tool and used for specific tool interactions.

Usable - an indicator that the item can be used in the player hotbar.

Usable.Abstract - an indicator that the item is not real and can be used only in hotbar slots. This item can not be added in the container components.

Usable.Charging - an indicator that the item can be charged by specific interactions.

Usable.Consumable - an indicator that the item will be consumed when used.

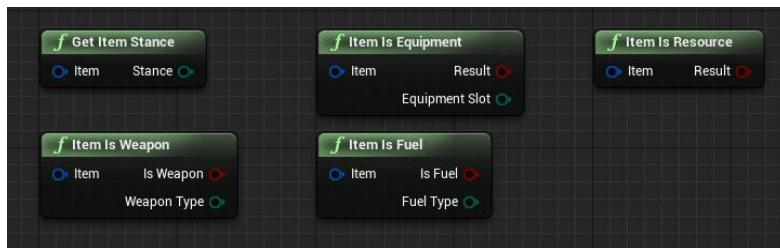
Usable.Continuable - an indicator that the item will be used again and again while you hold the main interaction button.

Usable.UsedAtStart - an indicator that the item will be used when you press the main interaction button. Otherwise the item will be used when you release the main interaction

button.

Weapon - an indicator that the item is a weapon.

Item type classifications are implemented in the **BP_ItemsLibrary**. The functions check the item tags and return specific item enumeration type. If you want to add or change items tags don't forget to change these functions in the library.



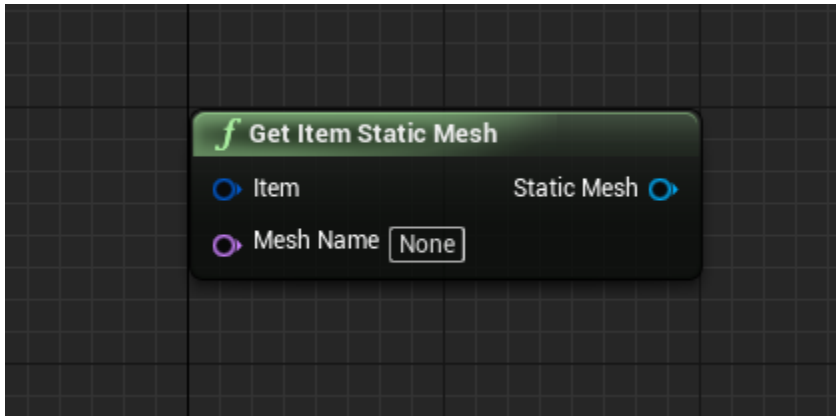
3.2.5. Item data map-list identifiers

The **StaticMeshes**, the **SkeletalMeshes**, the **Sounds** and the **Handles** variables of the item data can return objects by identifiers which are hard coded in the project.

Item Static Meshes

The item static meshes are used for displaying items in the world or for displaying default usable items of the character's weapon.

The **GetItemStaticMesh** function can return the static mesh if it is valid.



Drop - an identifier for static mesh which is used for displaying items in the world.

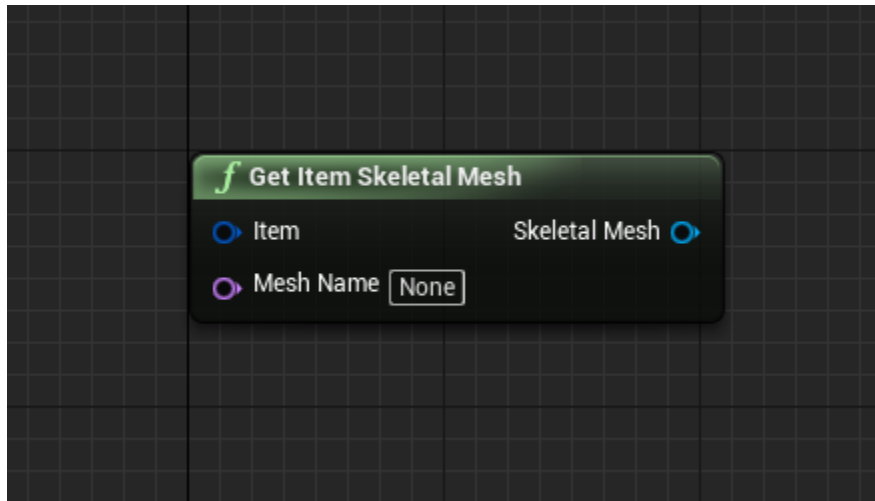
Usable - an identifier for static mesh which is used for default usable items.

Dummy - an identifier for static mesh which is used for displaying equipment items on the mannequin.

Item Skeletal Meshes

The item skeletal meshes are used for displaying equipment on the characters or for displaying default usable items of the character's weapon.

The **GetItemSkeletalMesh** function can return the static mesh if it is valid.



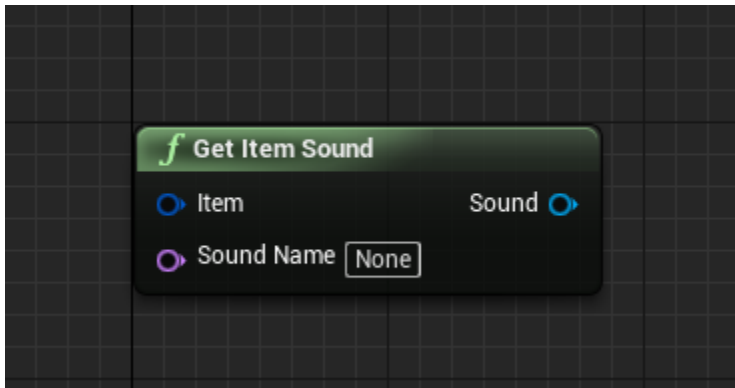
Usable - an identifier for skeletal mesh which is used for default usable items.

Jess - an identifier for skeletal mesh which is used for displaying equipment on the player character (**Jess**).

Item Sounds

The item sounds are used for playing sound when players pick up items or when items drop on different surfaces.

The **GetItemSound** function can return the static mesh if it is valid.



Take - an identifier for sound which is played when a player picks up the item.

Drop_Grass, Drop_Sand, Drop_Rock, Drop_Wood, Drop_Flesh, Drop_Splash - identifiers for sounds which are played when the item drops on a certain surface.

Impact_Grass, Impact_Sand, Impact_Rock, Impact_Wood, Impact_Flesh - identifiers for sounds which are played when a player hits a certain surface by the item.

Whoosh - an identifier for sound which is played in attack animations.

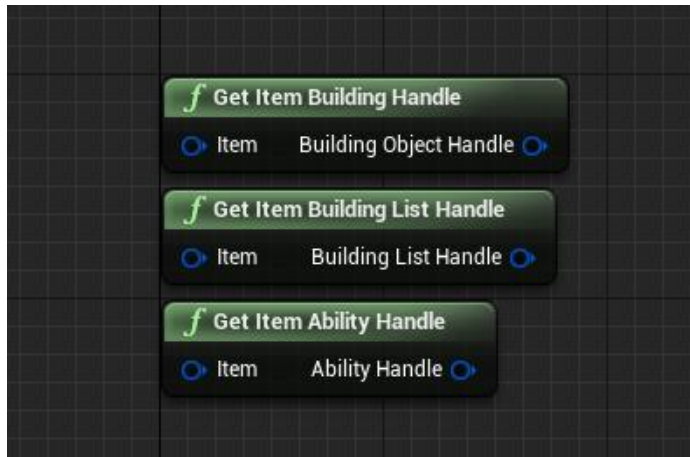
Shoot - an identifier for sound which is played in shoot animation.

DryShoot - an identifier for sound which is played in shoot animation and the item weapon has no ammo.

Item Handles

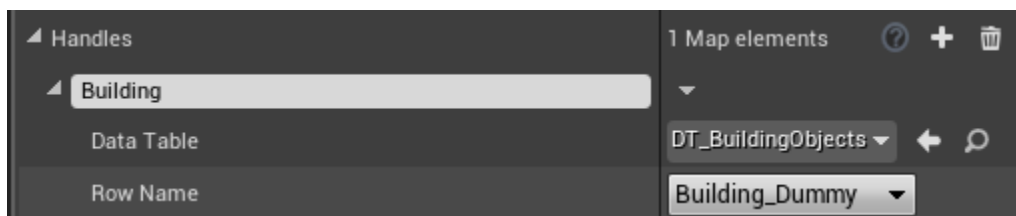
The **Handles** variable in the item data is used for connecting item data with specific data from other data tables. For example it is used for the connection of a specific item with specific building objects from a building data table.

These handles are hard coded in the **BP_ItemsLibrary**. If you want to add and use new handles, don't forget to implement such functions in the library.



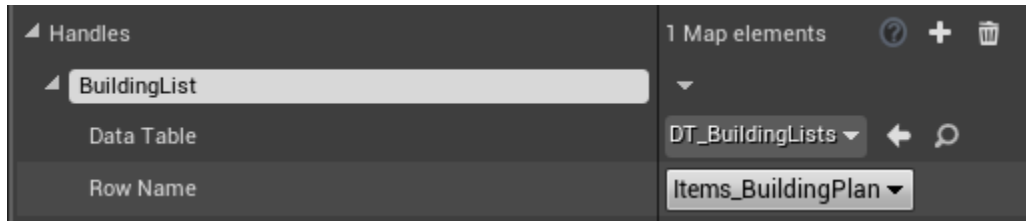
Each handle has its own name identifier.

Building - the handle to the specific building object from the building data table. It gets building object data which should be placed in the world by the item with the **Building** tag.



BuildingList - the handle to the specific building list which updates the building menu widget and allows users to select the building object for placing in the world. The item also

should have the **Building.BuildingPlan** tag.



Usable - the handle to the specific usable item data which is used for advanced usable item logic. This data overrides the default usable item of the item. Advanced usable items have different logic, meshes, particle effects, animation transitions and so on. For example: bow, assault rifle, torch.



Ability - the handle to the specific ability data which is used for the advanced ability system. This data overrides the item ability from **UseAbilityClass** variable and allows to configure ability attributes, effects and so on.



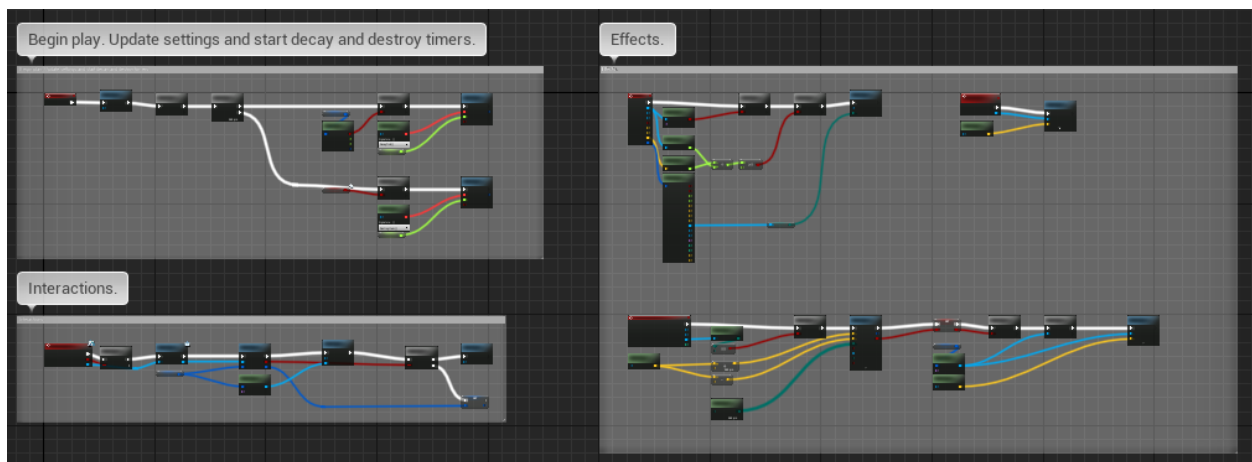
3.2.6. World items

Description

To place items on the level, use the **BP_Item** class. An object of the same class is created when an item is dropped from a container slot.

The class implements the **BPI_InteractionObject** interface so that the player can interact with it using the **BP_InteractionComponent** component.

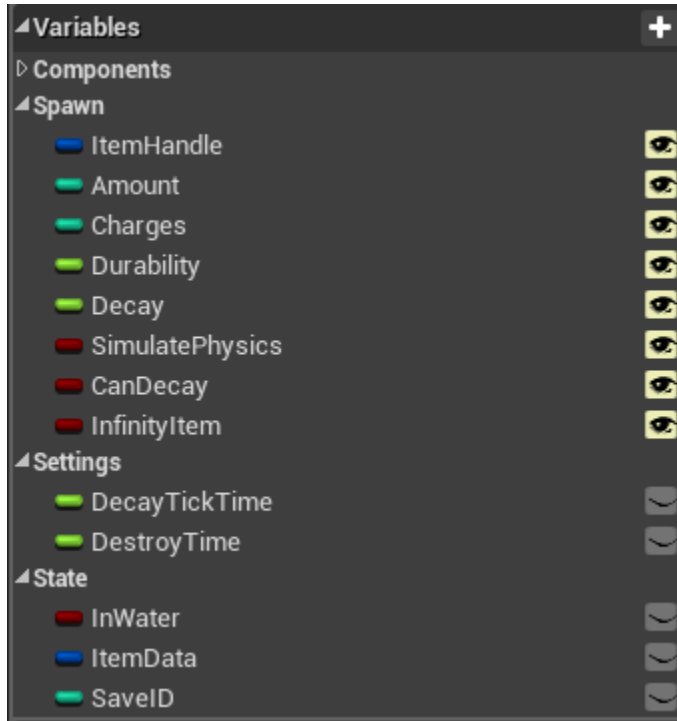
The class is replicated.



An dropped item from the inventory will be destroyed after a while, using a timer.

When an item drops to the ground or into water, it plays the sound configured in the table for that item.

Variables



ItemHandle - the spawn item id in the items data table.

Amount - the spawn amount of the item.

Charges - the spawn charges of the item.

Durability - the spawn durability of the item.

Decay - the spawn remaining decay time of the item when spawning.

SimulatePhysics - an indicator of whether physics simulation should be enabled for the item.

CanDecay - an indicator that the item can decay during time.

InfinityItem - an indicator that the item won't be removed from the game when it will be picked up.

DecayTickTime - timer time to update the decay value of the item.

DestroyTime - time after which the item will be destroyed after dropping.

InWater - an indicator that an item is in water.

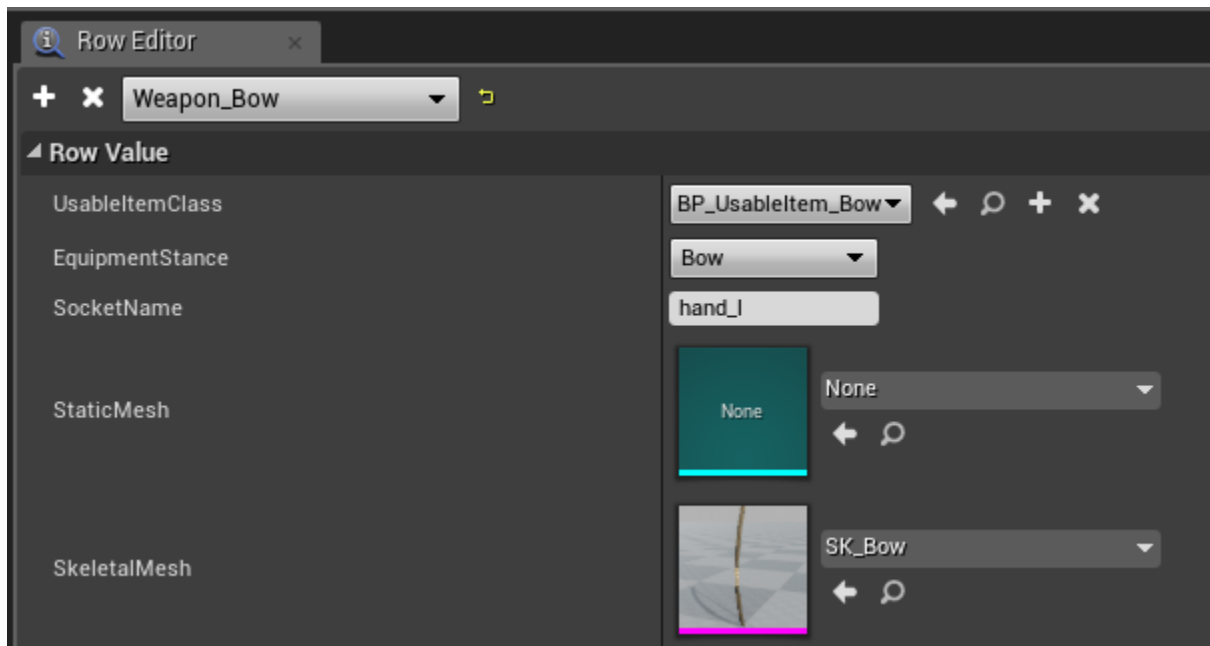
ItemData - complete information about the item, based on the values of the variables

when spawning.

SaveID - item id for the save system.

3.2.7. Usable items

The usable items are used as weapons or as tools of the characters. By default it can be a static or skeletal mesh and can be attached only to the **socket_weapon_r** socket of the main character skeletal mesh. But advanced usable items have advanced customisation options. These options can be configured in the **DT_UsableItems** and then selected for the items in the **Handles** variable by the **Usable** name.



UsableItemClass - the class of usable item blueprint. You can create your own usable item blueprint based on **BP_UsableItem_Base**, and add additional components like particles, audio components or specific logic and then use it.

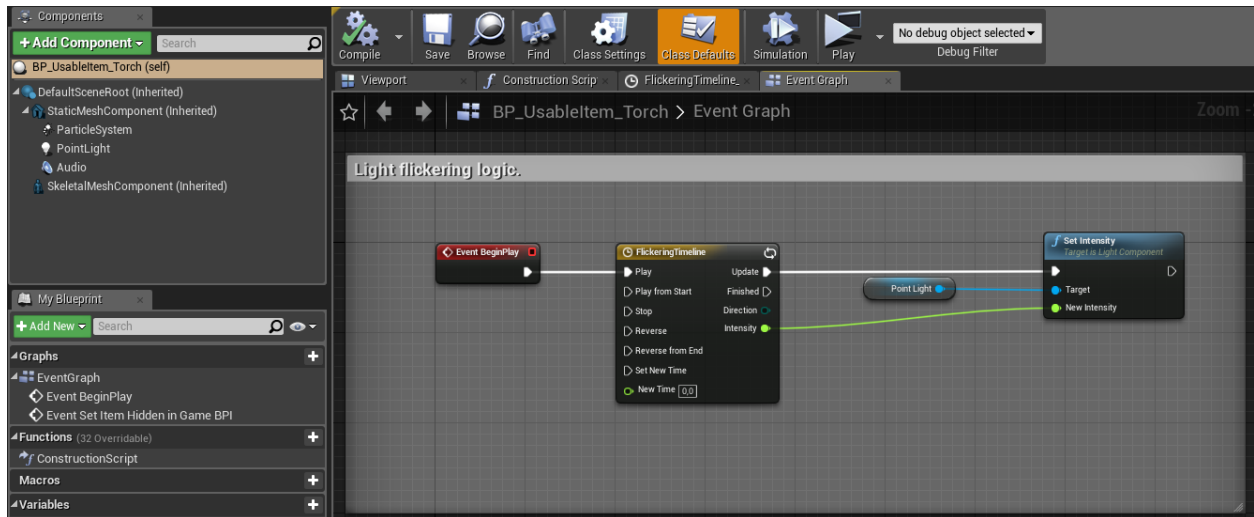
EquipmentStance - the stance for the character who used this item.

SocketName - the name of the main character skeletal mesh socket to which the usable item will be attached.

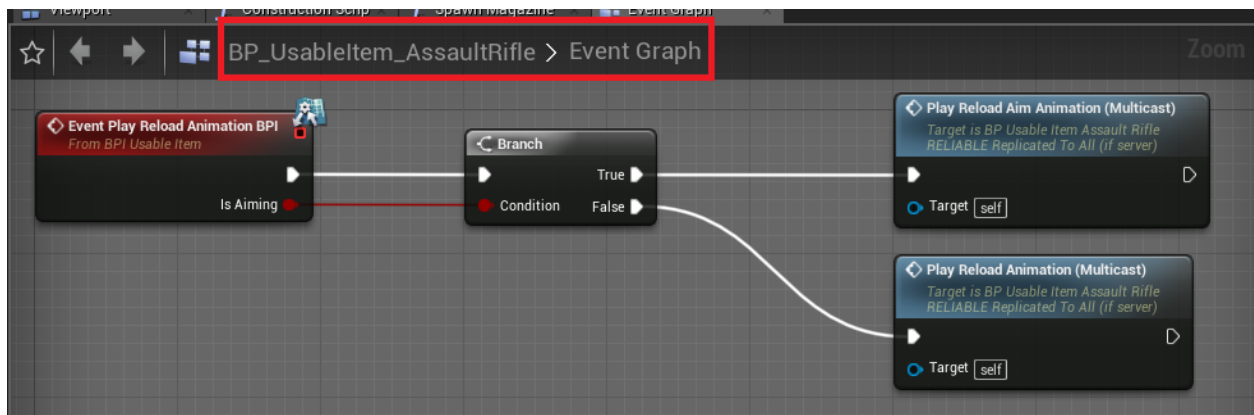
StaticMesh - the static mesh which is used for the usable item.

SkeletalMesh - the skeletal mesh which is used for the usable item.

Example of the custom usable item with additional particle and audio components. Also contains custom blueprint logic for changing torch light intensity during time.



If a usable item should play animations synchronously with character it must implement certain functions from the **BPI_UsableItem** interface.



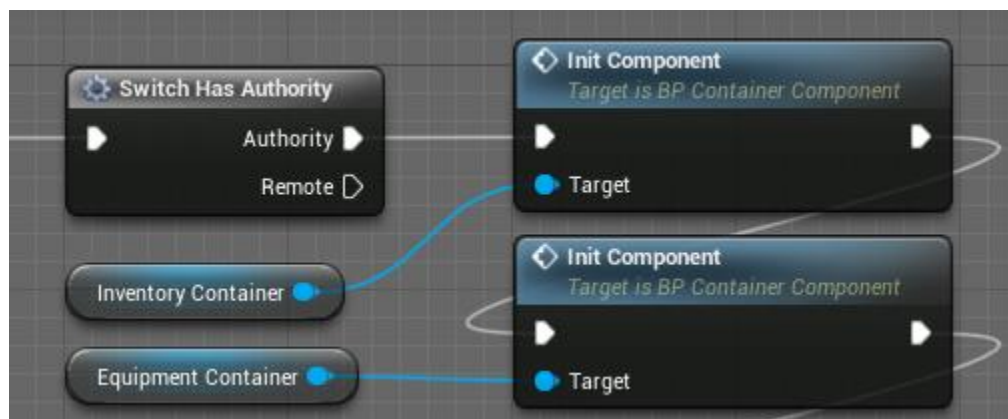
3.2.8. BP_ContainerComponent

Description

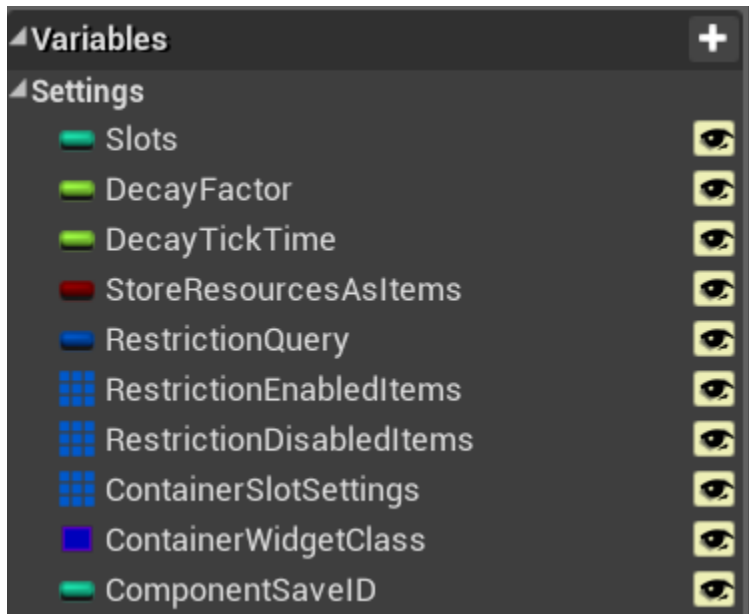
The component stores current information about items and resources in a container. Contains functions for adding and removing items and resources, functions for checking the availability of required items and resources in a container, as well as a function for calculating the total weight of items.

When changing an item in a container slot, a request is sent to change the information in the container slot in the user interface of active players who are using this component at the moment.

The component is replicated and needs to be initialized on the server.



Variables



Slots - number of slots in the container. In the player character blueprint this value is overridden by inventory slots attribute from attributes component.

DecayFactor - rate of decay of items in a container.

DecayTickTime - decay timer time.

StoreResourcesAsItems - an indicator that resource items will be added in the container slots instead of resources array.

RestrictionQuery - the tag query for possible items which can be added in the container.

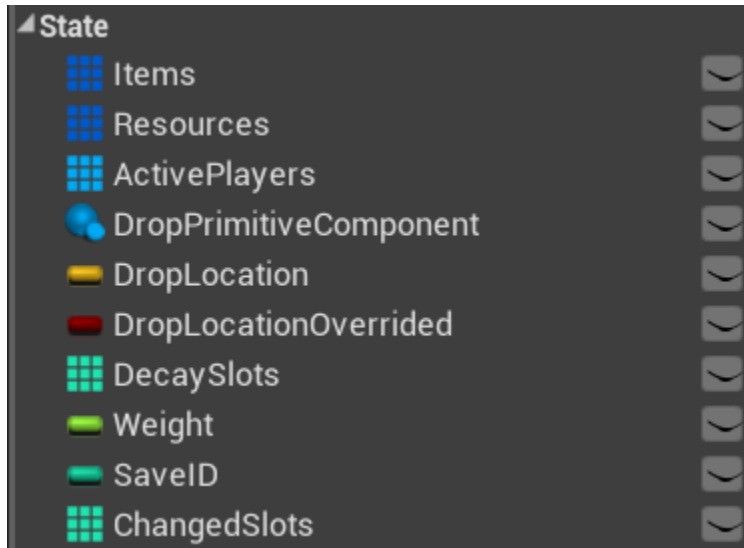
RestrictionEnabledItems - the list of items which can be added to the container. Only these items can be added in the container.

RestrictionDisabledItems - the list of items which can not be added to the container.

ContainerSlotSettings - specific settings for container slots. It includes slot name, slot description, slot background texture and slot item restrictions.

ContainerWidgetClass - custom widget class that will be opened when a player attempts to open a container. If the widget class is not set it will open the default loot container widget.

ComponentSaveID - unique component save identifier.



Items - items in container slots.

Resources - resources in the container.

ActivePlayers - players who use the container.

DropPrimitiveComponent - reference to the component whose position will be used to drop items.

DropLocation - location that will be used to create dropped items.

DropLocationOverridden - an indicator that it is required to overload the location in which the objects thrown from the container will be created.

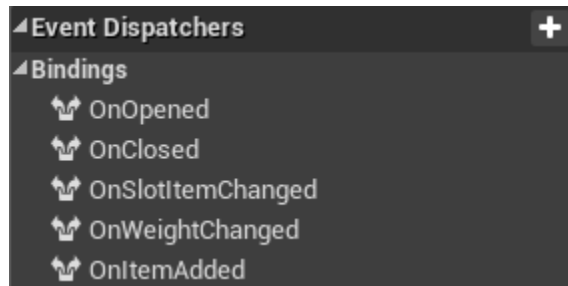
DecaySlots - an array of slots affected by the decay timer.

Weight - total weight of all items in the container.

SaveID - id for the save and load system.

ChangedSlots - the array of changed slots.

Event dispatchers



OnOpened - occurs when the container is opened by at least one player.

OnClosed - occurs when the container is closed by all players.

OnSlotItemChanged - occurs when an item in a container slot changes.

OnWeightChanged - occurs when the total weight of items changes.

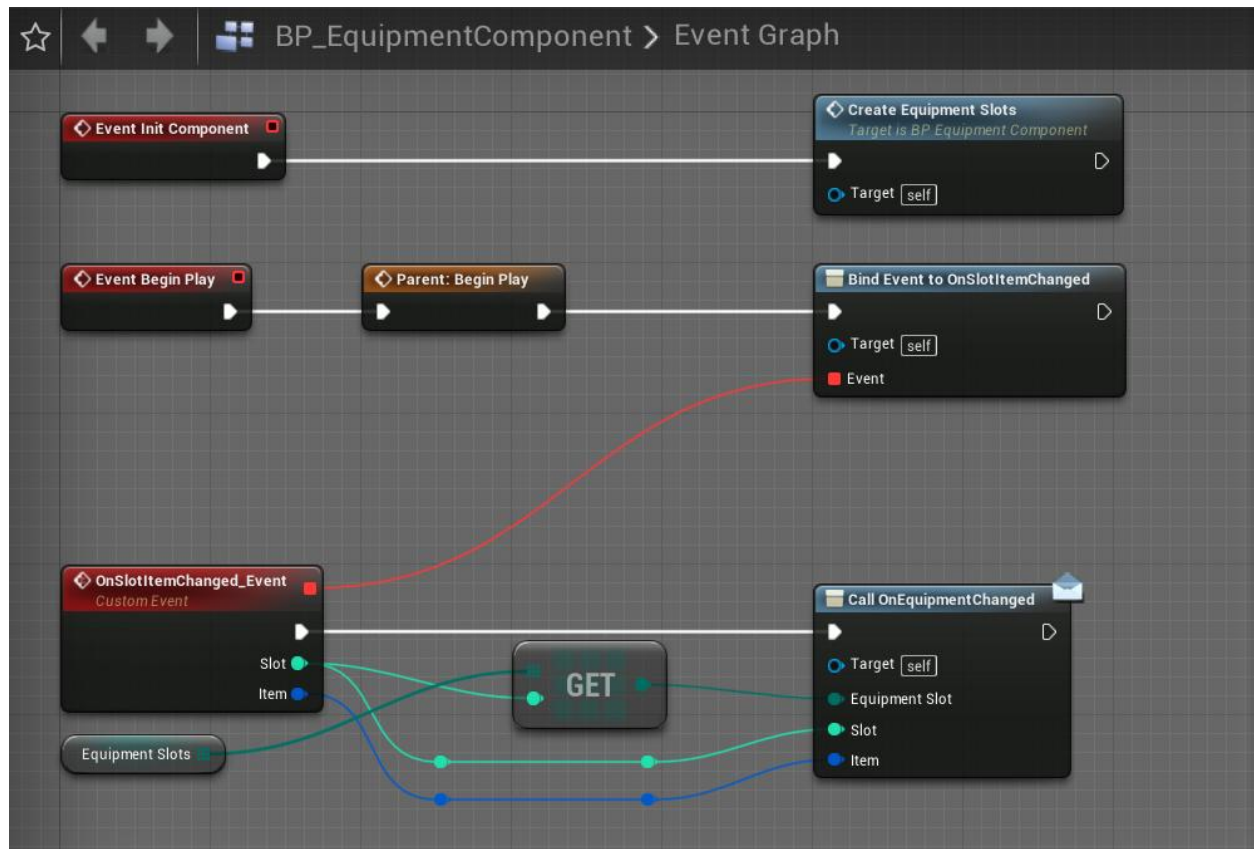
OnItemAdded - occurs when an item is added to a container.

3.2.9. BP_EquipmentComponent

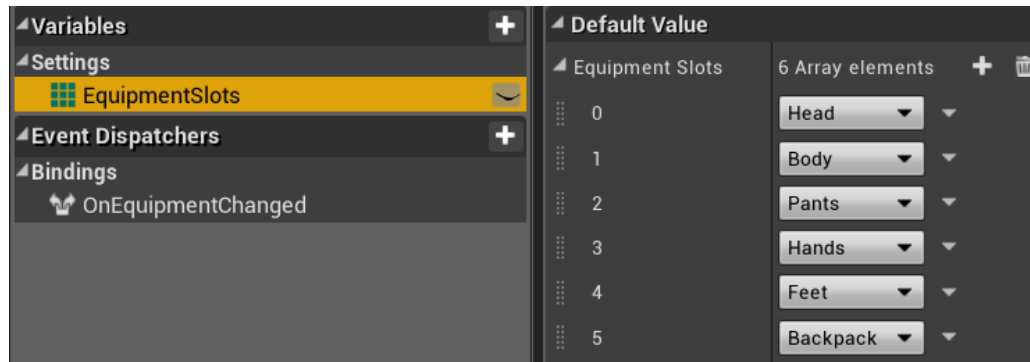
Description

Inherits from **BP_ContainerComponent**. Stores information about equipped items.

The component is replicated and must be initialized on the server.

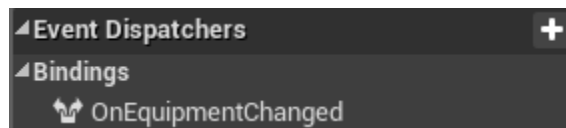


Variables



EquipmentSlots - an array that determines the amount and sequence of equipment slots. It is also used in the **Get/SetEquipmentSlot** functions.

Event Dispatchers



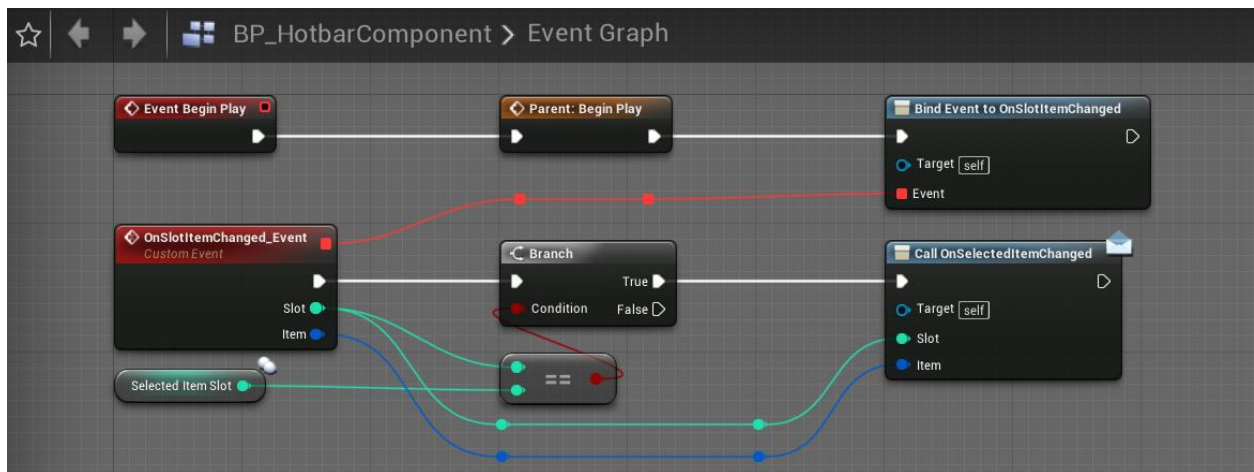
OnEquipmentChanged - occurs when an item in an equipment slot changes.

3.2.10. BP_HotbarComponent

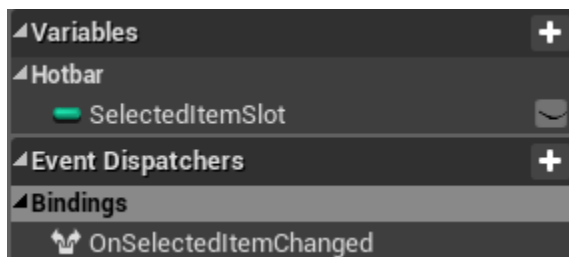
Description

Inherits from **BP_ContainerComponent**. Stores information about itemsO in the quick access panel. Contains functions and variables for selecting the current slot in the Hotbar.

The component is replicated and must be initialized on the server.



Variables



SelectedItemSlot - the index of the currently selected slot.

Event Dispatchers

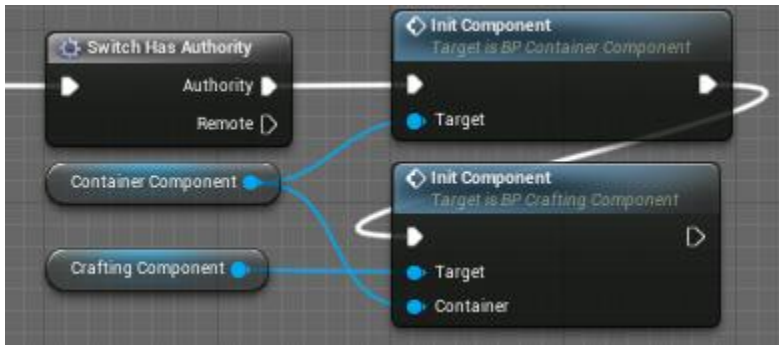
OnSelectedItemChanged - occurs when an item in the selected quick slot changes.

3.2.11. BP_CraftingComponent

Description

The component allows you to create items according to the selected crafting blueprints and add them to an initialized container. Crafting blueprints can be queued and processed in order. Several crafting blueprints can be processed at the same time, depending on the component setting. The crafting process is automatic, but can be suspended.

The component is replicated and must be initialized on the server.



Variables

Container - reference to the container from which the required items are taken and where the finished items will be added.

CraftingTickTimerHandle - crafting timer id.

ActivePlayers - queue of crafting blueprints in crafting.

BlueprintsQueue - queue of crafting blueprints in crafting.

SaveID - id for the save and load system.

CraftingListHandle - crafting list id from **DT_CraftingLists** data table.

CraftingEnabled - indicator that crafting is on.

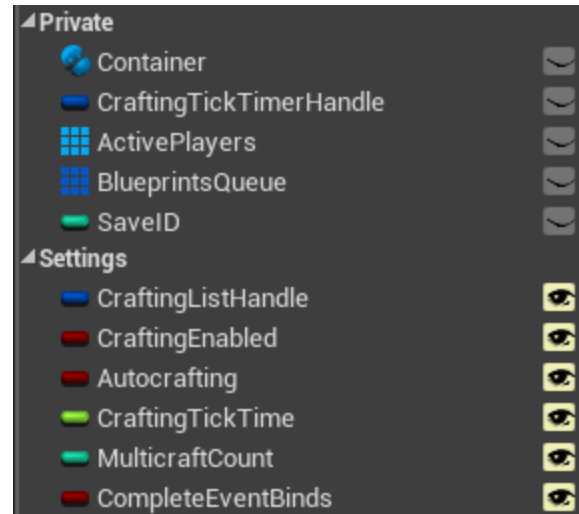
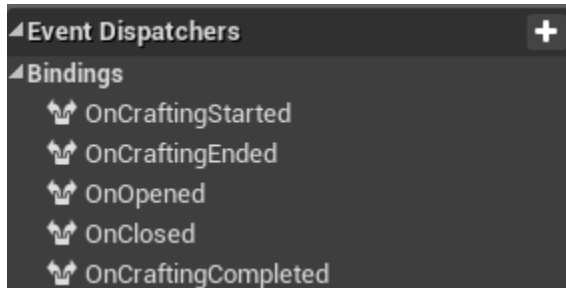
Autocrafting - an indicator that crafting will automatically start if there are items for the crafting of any blueprint from the list.

CraftingTickTime - crafting timer interval, to update progress.

MulticraftCount - number of crafting blueprints simultaneously in crafting.

CompleteEventBinds - if enabled, will trigger the event **OnCraftingCompleted**.

Event Dispatchers



OnCraftingStarted - occurs when crafting blueprint production starts.

OnCraftingEnded - occurs when crafting blueprint production ends.

OnOpened - occurs when at least one player opens.

OnClosed - occurs when closed by all players.

OnCraftingCompleted - occurs when crafting blueprint production is complete.

3.2.12. BP_FuelGeneratorComponent

Description

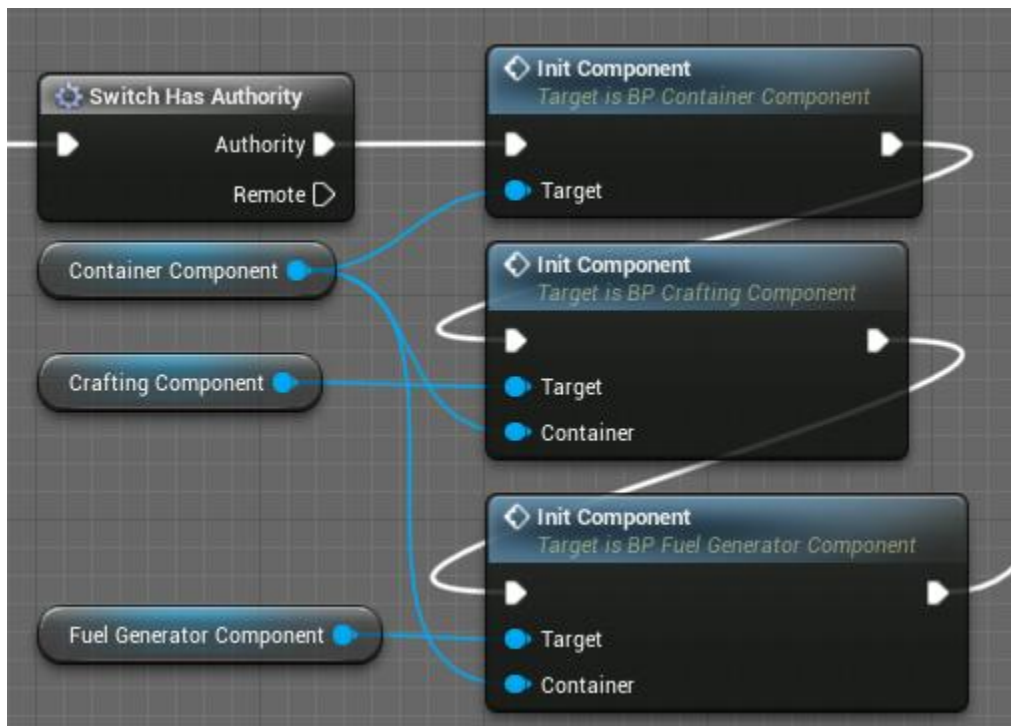
The component allows you to start crafting, depending on the availability of a certain type of fuel in the container. If fuel is available, it raises a switch-on event that can be associated with a production component.

Each type of fuel has its own burning time, after which the shutdown event is triggered, if there is no additional fuel in the container.

Has a timer to generate a burn result that triggers the event **OnGenerationTick**.

Also has an additional timer for generating burn waste, which triggers the event **OnWasteGenerationTick**. For example, in a campfire, this event is used to generate coal when burning.

The component is replicated and must be initialized on the server.



Variables

Container - reference to the container from which the fuel will be consumed.

BurnCycleTimerHandle - timer identifier for the burn cycle.

CurrentFuelType - the type of fuel used for the current burn cycle.

BurnGenerationTimerHandle - burn generation cycle identifier.

WasteGenerationTimerHandle - waste generation cycle identifier.

IsTurnedOn - an indicator of whether the waste generator is currently on or off.

SaveID - identifier for the save and load system.

FuelTypes - possible types of fuel.

FoliageCycleTime - foliage burning time.

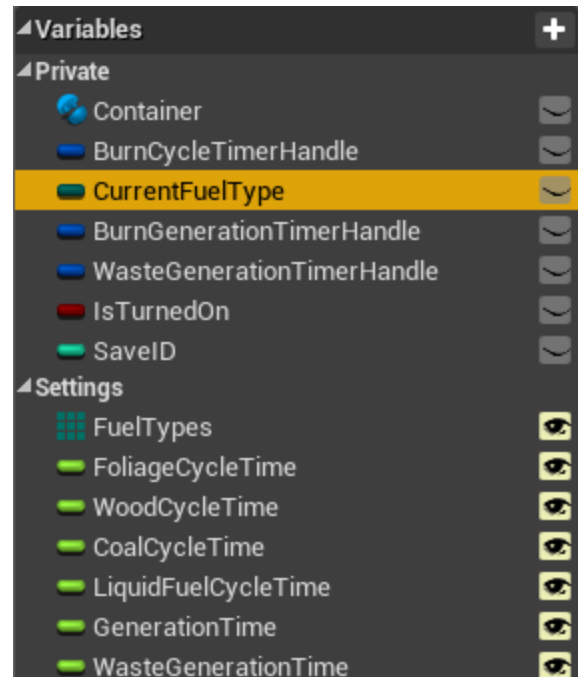
WoodCycleTime - wood burning time.

CoalCycleTime - coal burning time.

LiquidFuelCycleTime - burning time of liquid fuel.

GenerationTime - time of generating the **OnGenerationTick** event during the burn process.

WasteGenerationTime - the time when the **OnWasteGenerationTick** event was generated during the burn process.



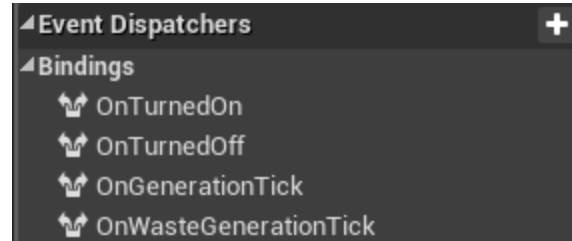
Event Dispatchers

OnTurnedOn - occurs when the burn process begins.

OnTurnedOff - occurs when the burn process ends.

OnGenerationTick - occurs when generation tick occurs.

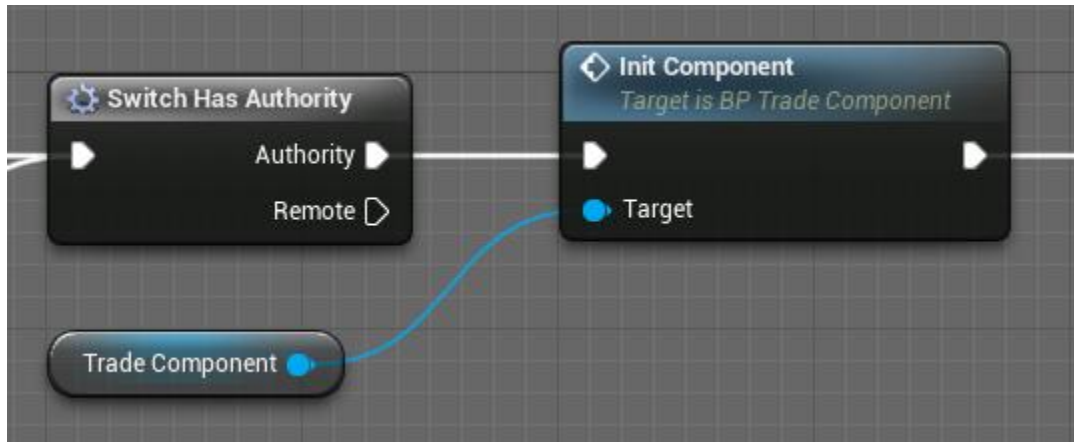
OnWasteGenerationTick - occurs when waste generation tick occurs.



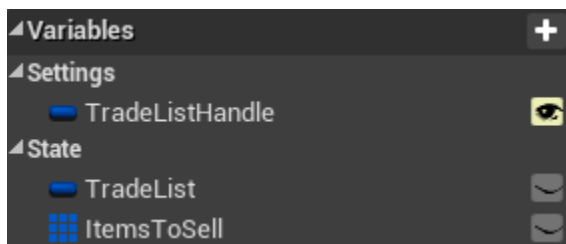
3.2.13. BP_TradeComponent

Description

A component that allows you to sell and buy items using a selected list of items for trade. The component is replicated and must be initialized on the server.



Variables



TradeListHandle - identifier of the list of items to trade from the data table

DT_TradeLists.

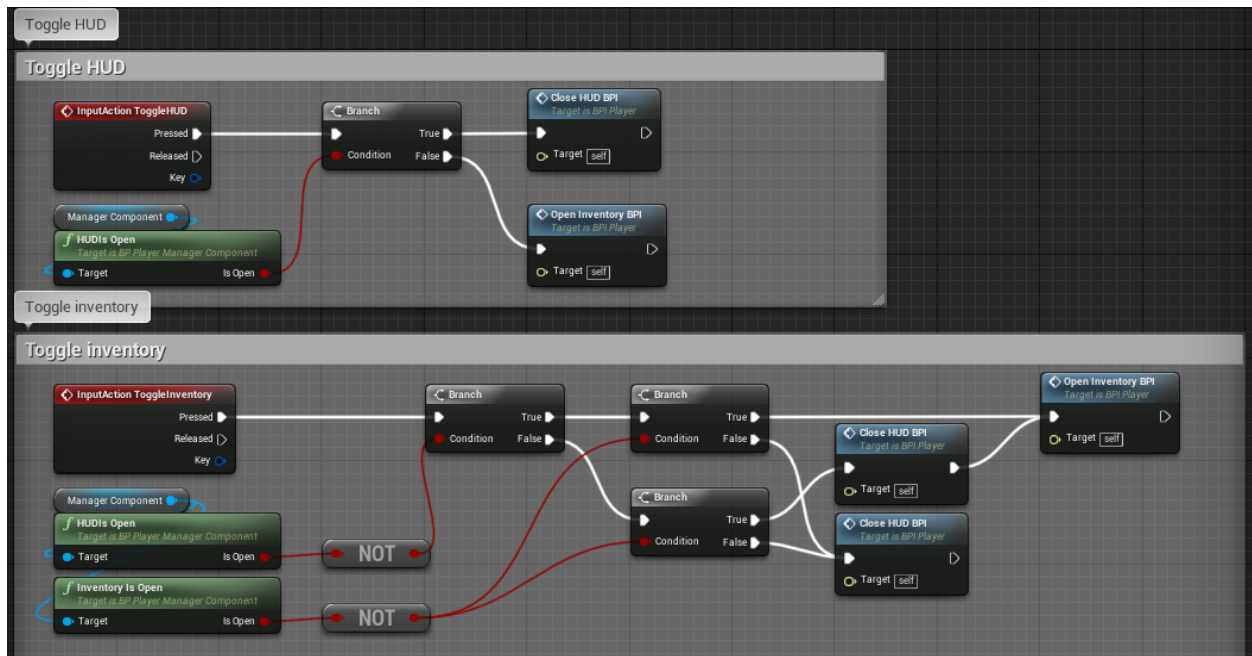
TradeList - list of items for trade.

ItemsToSell - items for sale.

3.2.14. Player interaction with inventory and equipment

Functions for interacting with inventory and equipment are in the **BPI_Player** interface and are implemented in the **BP_PlayerController** class, but the functionality of these functions is in the **BP_PlayerManagerComponent** class.

To open the inventory, you can use the buttons [I] or [Tab].



Items in inventory can be dragged to other slots by holding down the left mouse button.

Using the **[Shift]** button, you can take half the stack from the slot, and using the **[Ctrl]** button, one item from the stack. Items from the inventory can be dropped by dragging the item icon from the inventory to an empty space.

Items from the inventory can be dropped by dragging the item icon from the inventory to an empty space.

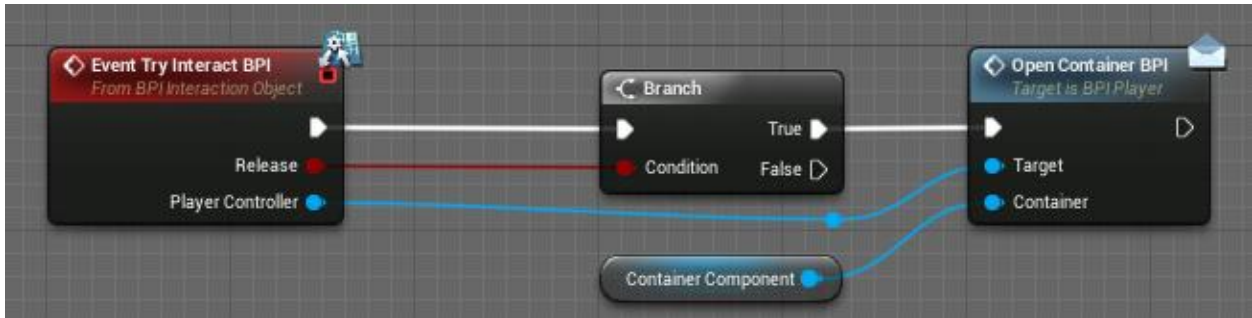
If the items are equipment, they can be placed in equipment slots by dragging or by right-clicking.

If items can be used they can be placed in Hotbar slots using drag and drop or by right-clicking.

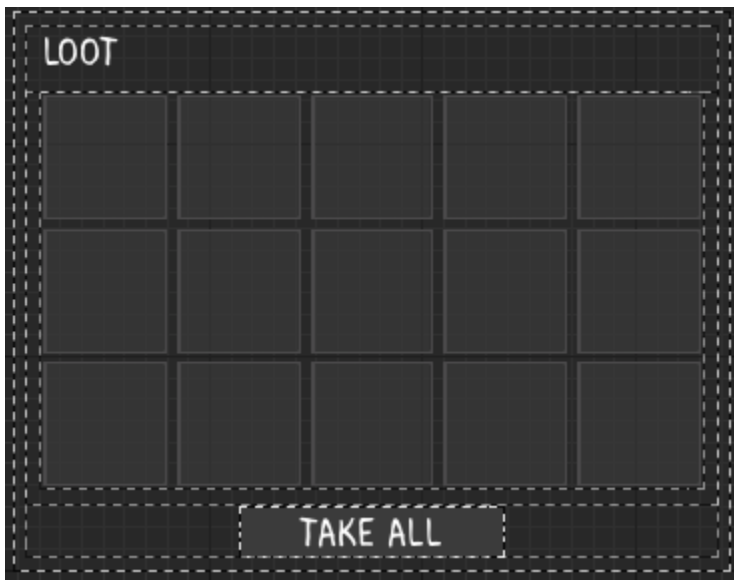
The functionality of the above actions can be found in the **UI_ItemSlot** and **UI_EquipmentSlot** widgets.

3.2.15. Player interaction with other containers

To open a container, use the **OpenContainer_BPI** function from the **BPI_Player** interface. It can be called when interacting with objects that have a container component.



To close the container, press the **[Tab]** button.



Pressing the **[Take All]** button will move all items from the container to the player's inventory.

The loot container widget functionality can be found in the **UI_LootContainer** widget.

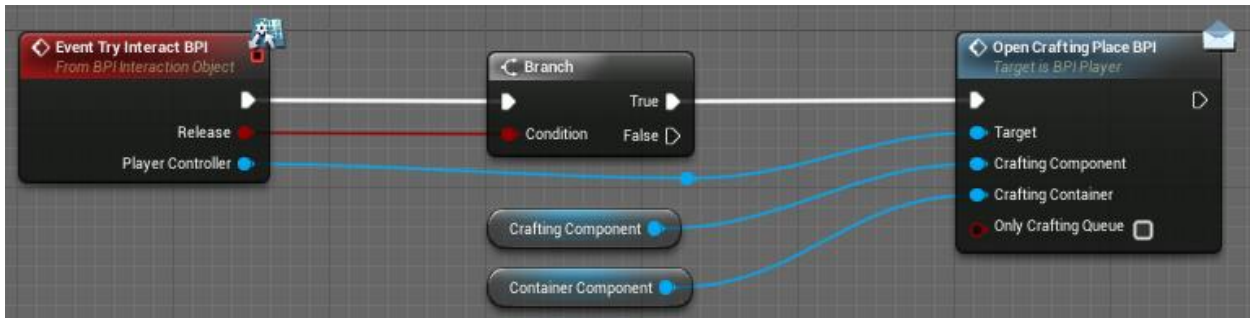
The loot container manager functionality can be found in the **UI_HUD** widget.

NOTE: You can use custom loot container widgets. For this you need to set the **ContainerWidgetClass** variable in details of the container component.

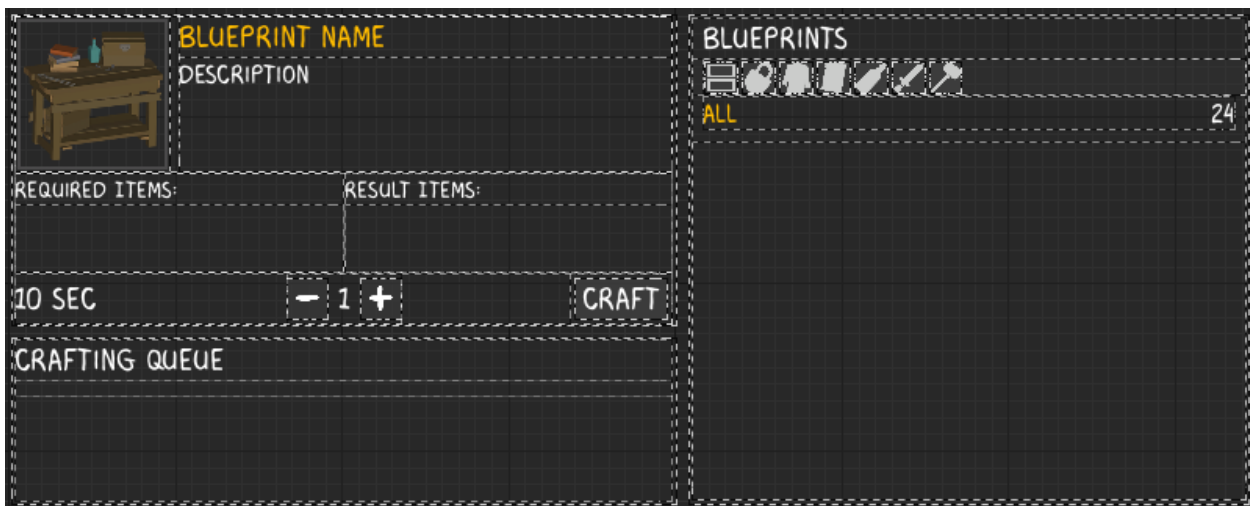
The container widget should implement the **BPI_UI_Container** interface and its functions.

3.2.16. Player interaction with a crafting component

To open a production component together with its container, use the **OpenCraftingPlace_BPI** function from the **BPI_Player** interface. It can be called when interacting with objects that have a crafting component.



You can close the inventory together with the place of production using the [Tab].



If the **OnlyCraftingQueue** variable is enabled when the **OpenCraftingPlace_BPI** function is called, then only the production queue will be displayed without a list of available crafting blueprints and information about the selected crafting blueprints.

When you click on the [Craft] button and if the required resources are available, the crafting process will start, which will take place in the **BP_CraftingComponent** component. The crafting process can be canceled by pressing the button with a cross.

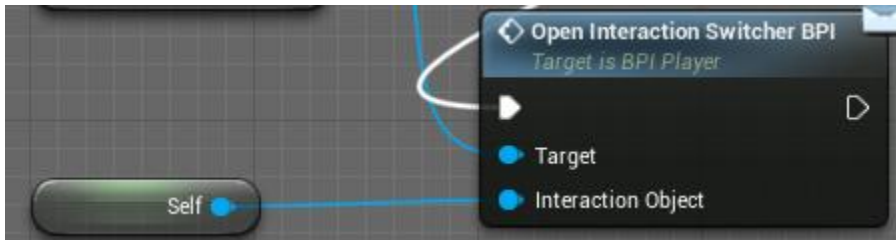
Crafting control functions are located in **BPI_Player** and are implemented in the **BP_PlayerController** class, but the functionality of these functions is in the **BP_PlayerManagerComponent** class.

3.2.17. Player interaction with generator component

The player can interact with the generator component using the **UI_InteractionSwitcher** widget.

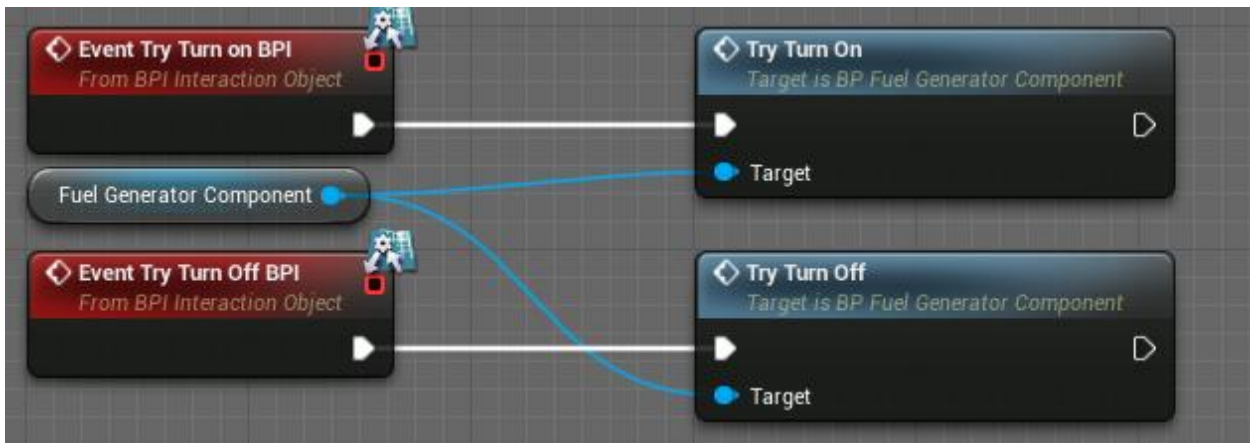


Handling events for pressing the on and off buttons are inside the blueprint of the widget. To open the switch widget, use the **OpenInteractionSwitcher_BPI** function from the **BPI_Player** interface. It can be called when interacting with objects that have a generator component.



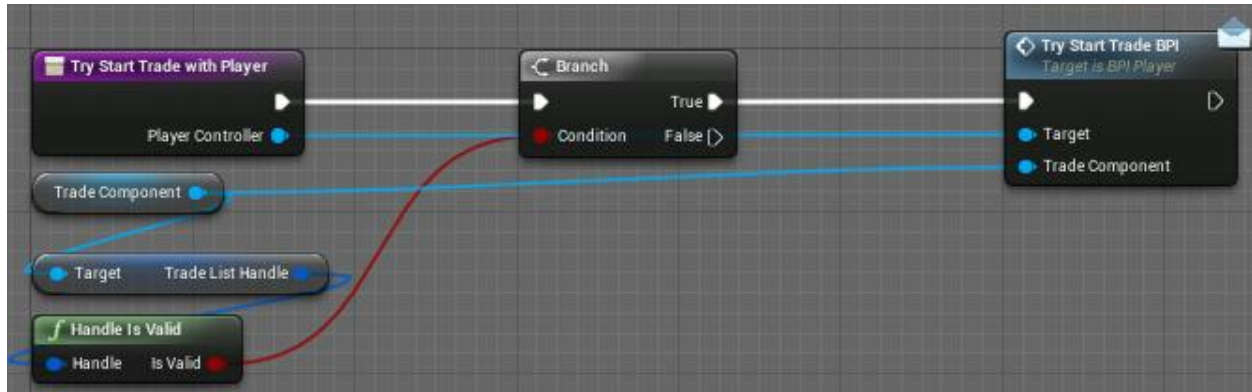
The switch control functions are located in **BPI_Player** and are implemented in the **BP_PlayerController** class, but the functionality of these functions is located in the **BP_PlayerManagerComponent** class.

An example of player interaction with a campfire.



3.2.18. Player interaction with the trade component

To open the trade component and update the list of items available for sale, use the **TryStartTrade_BPI** function from the **BPI_Player** interface. It can be called when interacting with objects that have a trade component or during dialogues with characters using the **TryStartTradeWithPlayer_BPI** function.



You can close the trading inventory using the **[Tab]** button.



When you click on the **[Sell]** button, an attempt will be made to sell the selected item at the selling price. When you click on the **[Buy]** button, an attempt will be made to buy the selected item at the specified price.

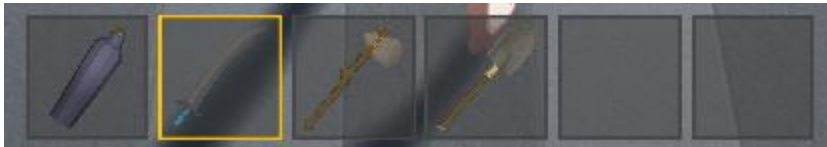
Event handling for clicking these buttons is located in **UI_SelectedTradingItemInfo**.

The trading process control functions are located in **BPI_Player** and are implemented in the **BP_PlayerController** class, but the functionality of these functions is located in the **BP_PlayerManagerComponent** class.

3.2.19. Player Interaction with the Hotbar

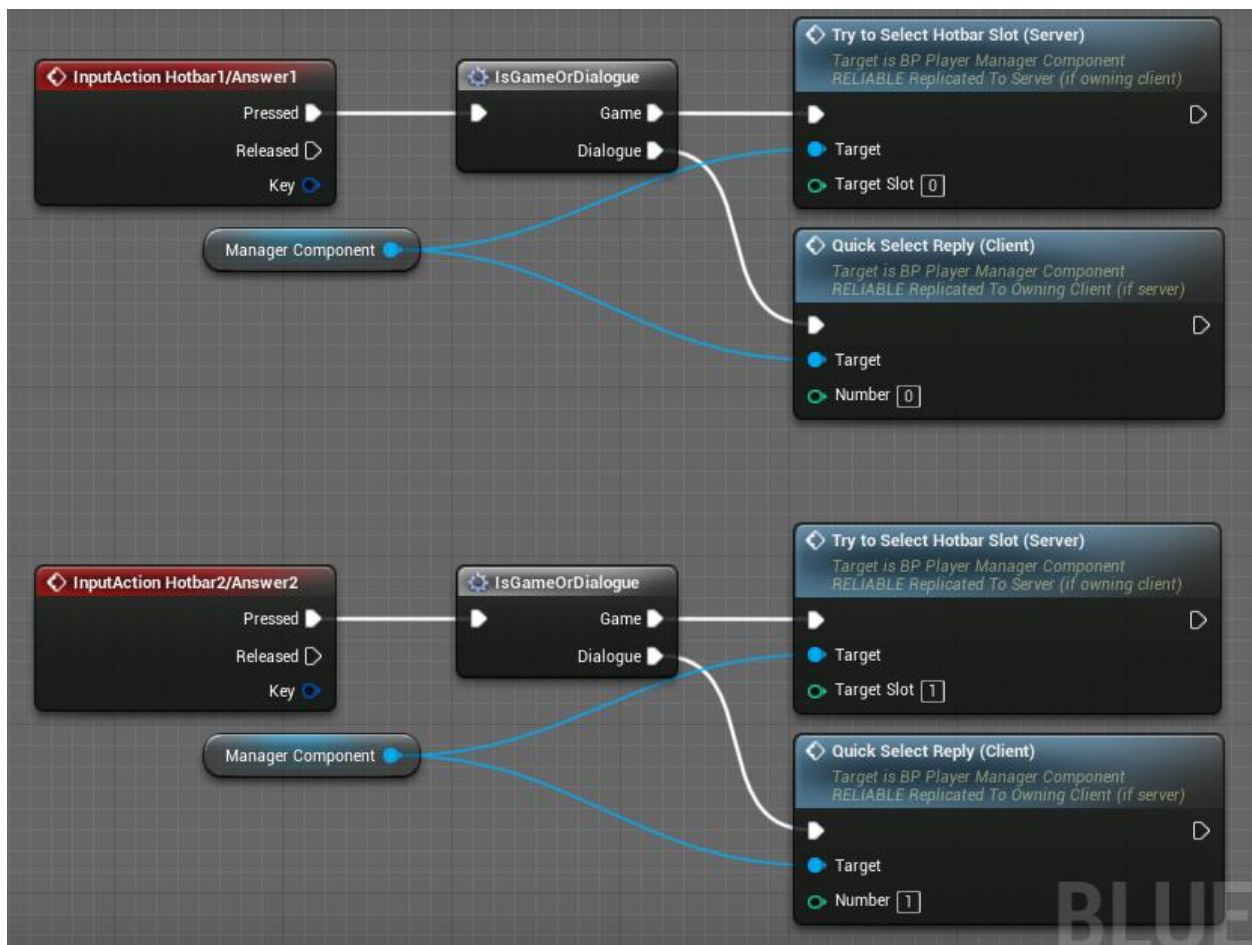
The player can interact with the quick access panel (Hotbar) and its active slot.

Items that can be used can be placed in quick access slots by dragging and dropping or by right-clicking on them.



The player can select the active slot using the buttons [1], [2], [3], [4], [5], [6].

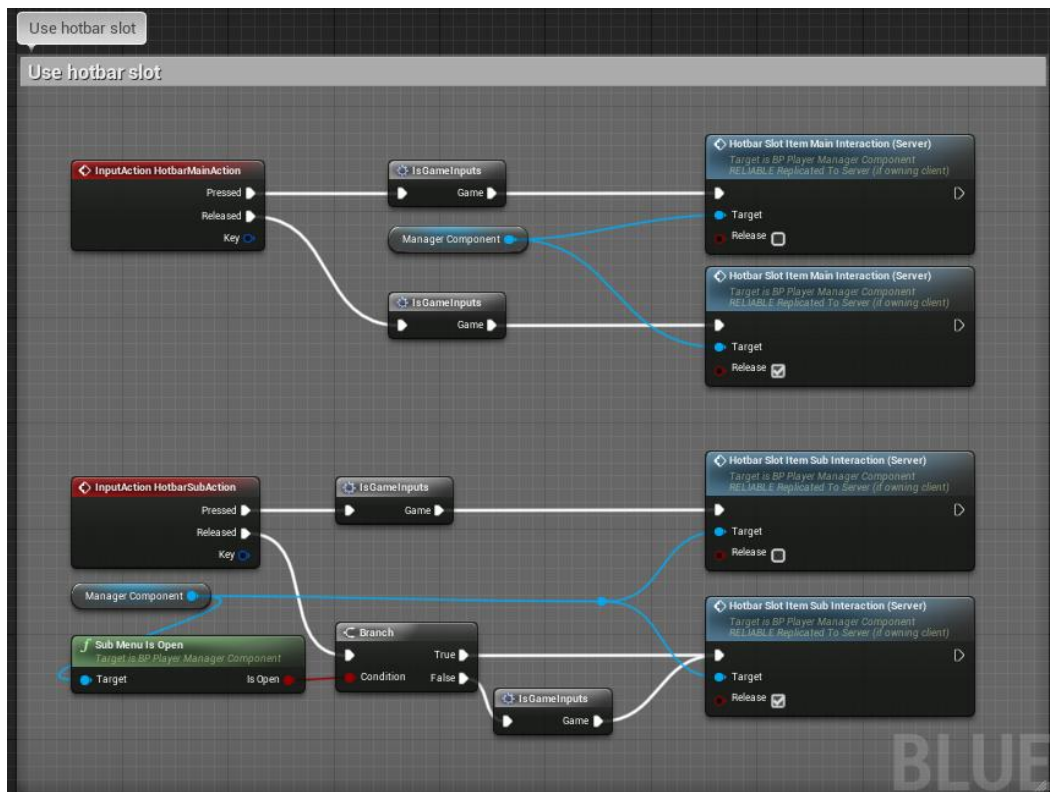
Functions for selecting the slot of the quick access bar are in **BP_PlayerController** in the Inputs graph. The switching functionality is in the **BP_PlayerManagerComponent**.



Depending on the selected item, the player can perform certain actions. Most of the used items can be used with the left mouse button, but some have an additional action.

For example, a weapon can not only attack, but also block.

The functions of player interaction with items in the quick access panel are located in **BP_PlayerController** in the **Inputs** graph. The interaction functionality is in the **BP_PlayerManagerComponent** and the **BP_PlayerCharacter** class.



3.3. Building System

3.3.1. Description

The building system allows the player to build various objects on the landscape or landscape meshes, and also allows objects to be built on top of each other if they can be connected.

The working principle is based on the interaction of the building component class **BP_BuildingComponent** with the building objects inherited from the main building class **BP_Building_BaseObject**.

The functions for controlling building processes are located in the **BPI_Player** interface and are implemented in the **BP_PlayerController** class. The building process management functionality is located in the **BP_BuildComponent** component.

The building system is referenced to the system of items and resources and the process of checking the availability of required items for build, repair and upgrade takes place in the **BP_PlayerManagerComponent**.

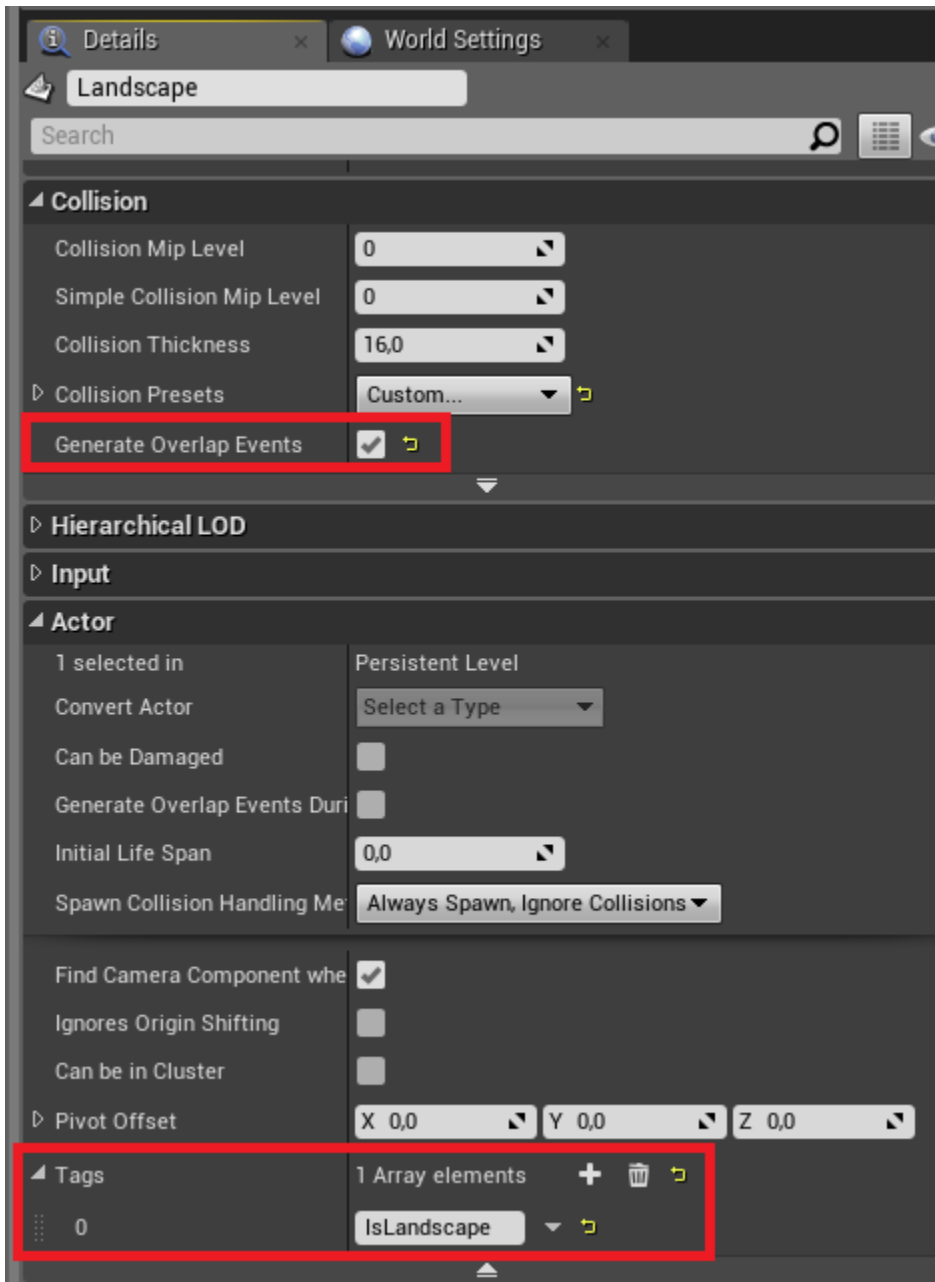
The functions to check the ability to build are in the **BP_Building_BaseObject** class, which can also be individually configured for each inherited class. The logic of snapping objects is configured using the socket system in their individual classes.

The **STR_BuildingObjectSettings** structure is used to store information about unique building objects. All building objects are stored in the **DT_BuildingObjects** data table.

The functionality for displaying the building menu for the user interface is in **BP_PlayerManagerComponent**. And the functionality of the building menu is in **UI_BuildingMenu**.

The **BPI_Ownership** interface is used to privatize the territory. It must be implemented by objects that prohibit building for other players. Building is prohibited for players who are not authorized in these facilities.

To place objects in a landscape, you need to enable **GenetateOverlapEvents** in the collision settings, and add the **IsLandscape** actor tag. A landscape can be any actor with geometry for which the above settings are specified.

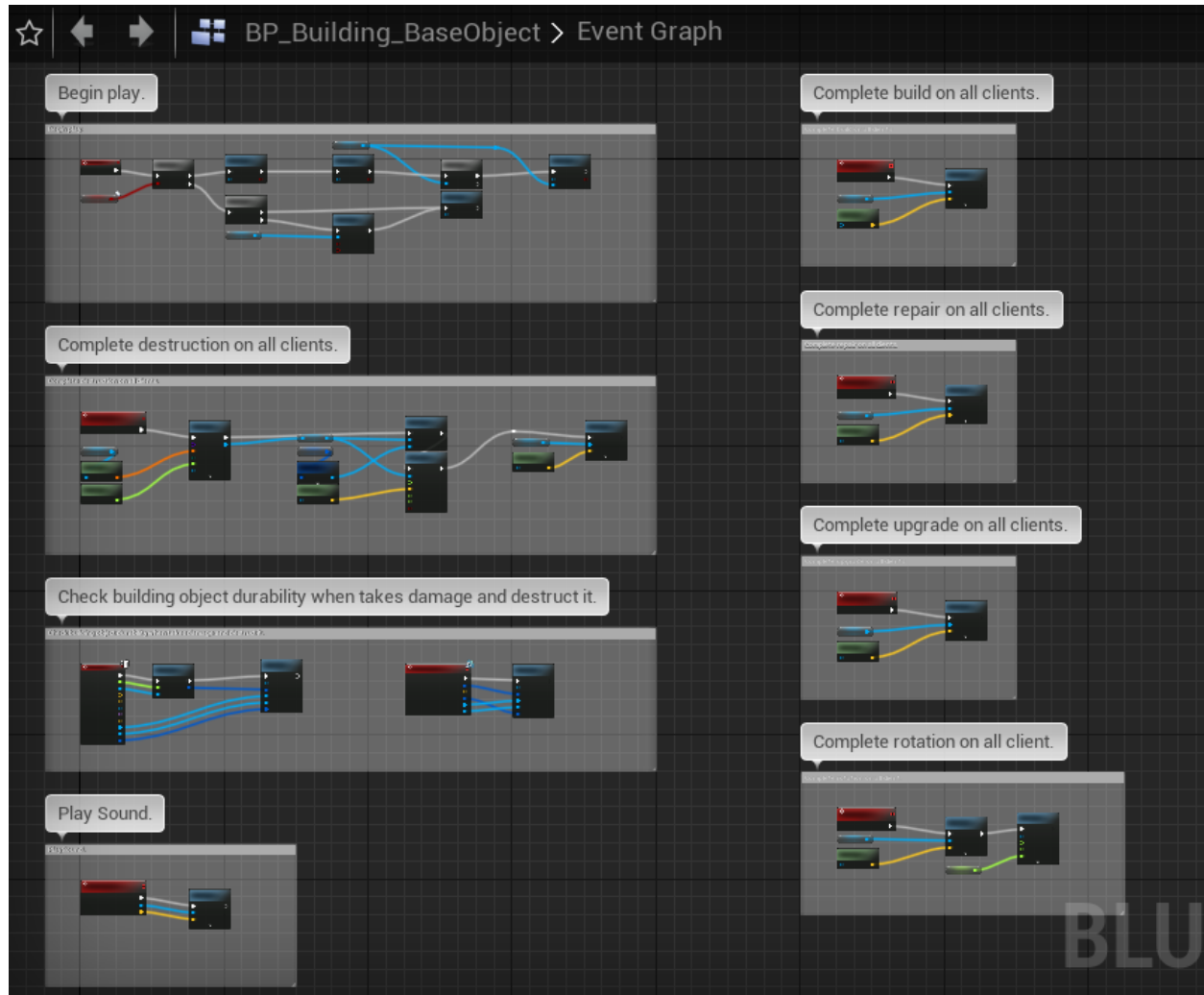


Any actor with a standard trigger collision can be used to forbid the ability to build. This actor must have a **BlockBuildingZone** tag. You can also use the already configured **BP_BlockBuildingZone** blueprint actors.

3.3.2. BP_Building_BaseObject

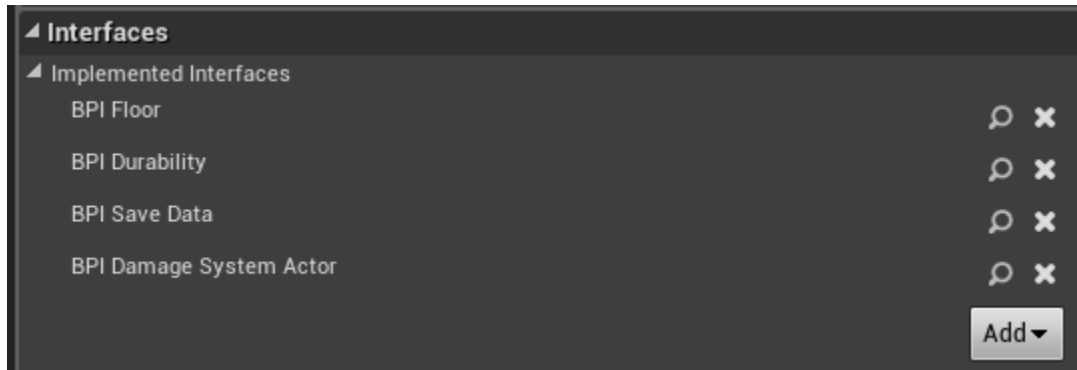
3.3.2.1. Description

BP_Building_BaseObject - the main building object class, all building objects must be inherited from it. Contains buildability check, landscape check, support check, and snap functions for modular parts.



3.3.2.2. Interfaces

The class implements the **BPI_Floor**, **BPI_Durability**, **BPI_SaveData** and the **BPI_DamageSystemActor** interfaces.

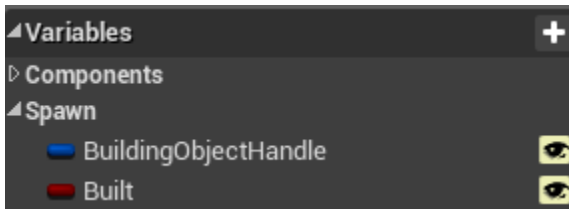


- **BPI_Floor** - interface for working with the floor of the object.
- **BPI_Durability** - interface for working with object durability.
- **BPI_SaveData** - interface for working with the save and load system.
- **BPI_DamageSystemActor** - interface for working with the advanced damage system.

3.3.2.3. Variables

Spawn

- **BuildingObjectHandle** - handle for loading object settings from the data table. If not specified, the default object settings from the building object class are used.
- **Built** - identifier that the object is already built. Must be true if the building object is placed in the editor.

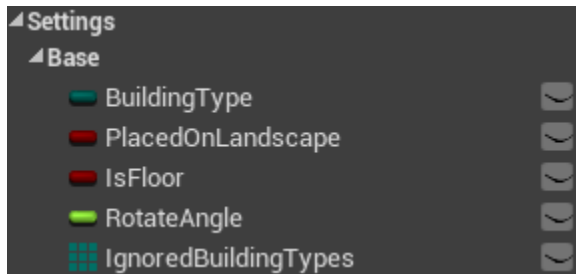


Note: Spawn variables should be configured, if an object is placed in the editor.

Settings / Base

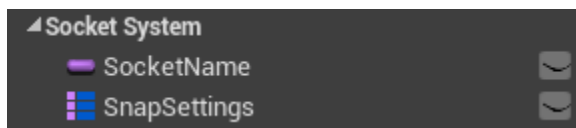
- **BuildingType** - type of building object. It is used in functions of snapping with other objects.
- **PlacedOnLandscape** - If true, the building object can be placed on the landscape or on static meshes, if they are landscapes. Otherwise, the object can only snap to other objects.
- **IsFloor** - if true, then the object is a floor and you can build objects on it that can be placed on the floor.
- **RotateAngle** - is the angle by which the object can be rotated around the Z-axis. If the rotation angle is 0, the object cannot be rotated in the object rotation mode.

-
- **IgnoredBuildingTypes** - the types of building objects that will be ignored in the buildability check function if they overlap this object.



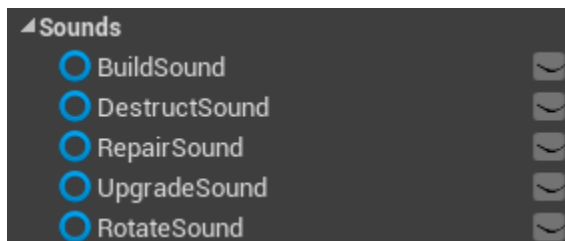
Settings / Socket System

- **SocketName** - object identifier for the socket system. Used in snapping functions with other objects.
- **SnapSettings** - settings to snap other objects to this object using the socket system. A detailed description can be found in the corresponding section.



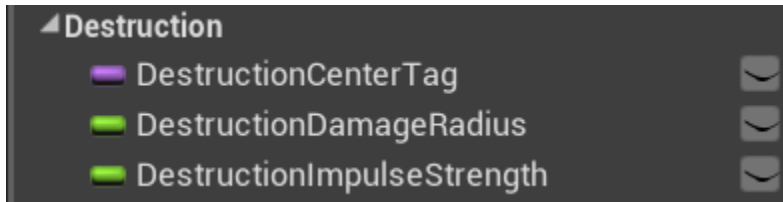
Settings / Sounds

- **BuildSound** - the sound that is played during the building of the object.
- **DestructSound** - the sound that is played when an object is destroyed.
- **RepairSound** - the sound that is played during the repair of the object.
- **UpgradeSound** - the sound that is played when the object is upgraded.
- **RotateSound** - the sound that plays when the object is rotated.

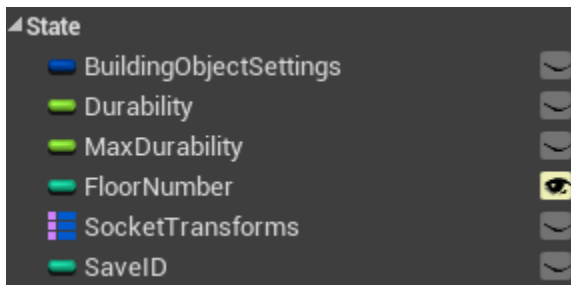


Settings / Destruction

- **DestructionCenterTag** - the tag of the scene component that is used for finding the destruction center in the **GetDestructionHurtLocation** function.
- **DestructionDamageRadius** - a value that determines destruction radius.
- **DestructionImpulseStrength** - a value that determines destruction strength.

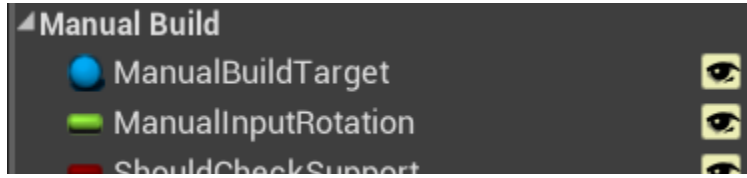


State



- **BuildingObjectSettings** - the settings of the building object. Includes name, description, icon, mesh and other important settings. Automatically loaded from the data table by the object's handle. Calls the **OnRep_BuildingObjectHandle** function when the value changes.
- **Durability** - the current durability of the object.
- **MaxDurability** - maximum durability of the object.
- **FloorNumber** - current number of the floor on which the object is placed.
- **SocketTransforms** - data about socket positions. Generated automatically when the object is initialized.
- **SaveID** - object identifier for the save and load system.

Manual Build



- **ManualBuildTarget** - target object, with which you should snap this object in the manual building mode in the editor.
- **ManualInputRotation** - input rotation angle, relative to which the object position is determined when placing in the manual building mode in the editor.
- **ShouldCheckSupport** - determines that building object should check support in the **CheckDestruction** event. Set **false** if you want to place the building object in the editor and without support or the building object will be destroyed after the loading process.

3.3.2.4. Building object settings

The building object settings are automatically loaded from the table by the handle of the **BuildingObjectHandle** variable when initializing the object in the world. If the handle is not set, the default object settings for the building class will be used. Data of object settings are stored in the **DT_BuildingObjects** table. Other data tables based on **STR_BuildingObjectSettings** structure can also be used.

Row Name	Handle	Name	Class
1 Dummy_Campfire	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Campfire	BlueprintGeneratedClass/Game/EasyBuildingSystem
2 Dummy_DoorLock	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Door Lock	BlueprintGeneratedClass/Game/EasyBuildingSystem
3 Dummy_CodeLock	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Code Lock	BlueprintGeneratedClass/Game/EasyBuildingSystem
4 Dummy_Torch	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Torch	BlueprintGeneratedClass/Game/EasyBuildingSystem
5 Dummy_Bed	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Bed	BlueprintGeneratedClass/Game/EasyBuildingSystem
6 Dummy_Foundation	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Foundation	BlueprintGeneratedClass/Game/EasyBuildingSystem
7 Dummy_Foundation_Triangle	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Triangle Foundation	BlueprintGeneratedClass/Game/EasyBuildingSystem
8 Dummy_Ramp	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Ramp	BlueprintGeneratedClass/Game/EasyBuildingSystem
9 Dummy_Wall	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Wall	BlueprintGeneratedClass/Game/EasyBuildingSystem
10 Dummy_DoorFrame	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Door Frame	BlueprintGeneratedClass/Game/EasyBuildingSystem
11 Dummy_WindowFrame	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Window Frame	BlueprintGeneratedClass/Game/EasyBuildingSystem
12 Dummy_Fence	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Fence	BlueprintGeneratedClass/Game/EasyBuildingSystem
13 Dummy_Stairs	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Stairs	BlueprintGeneratedClass/Game/EasyBuildingSystem
14 Dummy_Ceiling	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Ceiling	BlueprintGeneratedClass/Game/EasyBuildingSystem
15 Dummy_Ceiling_Triangle	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Triangle Ceiling	BlueprintGeneratedClass/Game/EasyBuildingSystem
16 Dummy_RoofWall_Left	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Left Roof Wall	BlueprintGeneratedClass/Game/EasyBuildingSystem
17 Dummy_RoofWall_Right	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Right Roof Wall	BlueprintGeneratedClass/Game/EasyBuildingSystem
18 Dummy_RoofWall_Top	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Top Roof Wall	BlueprintGeneratedClass/Game/EasyBuildingSystem
19 Dummy_Roof	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Roof	BlueprintGeneratedClass/Game/EasyBuildingSystem
20 Dummy_Roof_Top	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Top Roof	BlueprintGeneratedClass/Game/EasyBuildingSystem
21 Dummy_Window	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Window	BlueprintGeneratedClass/Game/EasyBuildingSystem
22 Dummy_Door	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Dummy Door	BlueprintGeneratedClass/Game/EasyBuildingSystem
23 Stylized_Campfire	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Campfire	BlueprintGeneratedClass/Game/EasyBuildingSystem
24 Stylized_DoorLock	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Door Lock	BlueprintGeneratedClass/Game/EasyBuildingSystem
25 Stylized_CodeLock	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Code Lock	BlueprintGeneratedClass/Game/EasyBuildingSystem
26 Stylized_Torch	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Torch	BlueprintGeneratedClass/Game/EasyBuildingSystem
27 Stylized_Bed	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Bed	BlueprintGeneratedClass/Game/EasyBuildingSystem
28 Stylized_Wooden_Foundation	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Wooden Foundation	BlueprintGeneratedClass/Game/EasyBuildingSystem
29 Stylized_Wooden_Foundation_Triangle	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Wooden Triangle Foundation	BlueprintGeneratedClass/Game/EasyBuildingSystem
30 Stylized_Wooden_Ramp	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Wooden Ramp	BlueprintGeneratedClass/Game/EasyBuildingSystem
31 Stylized_Wooden_Wall	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Wooden Wall	BlueprintGeneratedClass/Game/EasyBuildingSystem
32 Stylized_Wooden_Doorframe	("DataTable": "DataTable/Game/EasyBuildingSystem/Blueprints/DataT...	Wooden Door Frame	BlueprintGeneratedClass/Game/EasyBuildingSystem

-
- **Handle** - object identifier in the table.
 - **Name** - text object name.
 - **Class** - building object class.
 - **Icon** - building object icon.
 - **Description** - text description of the object.
 - **StaticMesh** - static model of the object.
 - **DestructibleMesh** - destructible model of the object.
 - **MaxDurability** - maximum object durability.
 - **CanBeDamaged** - can take damage and be destroyed.
 - **UseMeshSockets** - use mesh sockets instead of class sockets.
 - **CorrectMeshTransform** - corrective offset of the object model.
 - **BuildingRequirements** - identifier of the requirements for the building of the object. Selected from another table, e.g. from the requirements table.
 - **UpgradeObject** - identifier of the object settings to which this object can be upgraded.
 - **UpgradeRequirements** - identifier of the requirements for the object upgrade. Selected from another table, e.g. from the requirements table.
 - **RepairRequirements** - identifier of the requirements for the repair of the object. Selected from another table, e.g. from the requirements table.
 - **RepairFactor** - a fraction of the maximum durability to which the object is restored when it is repaired.

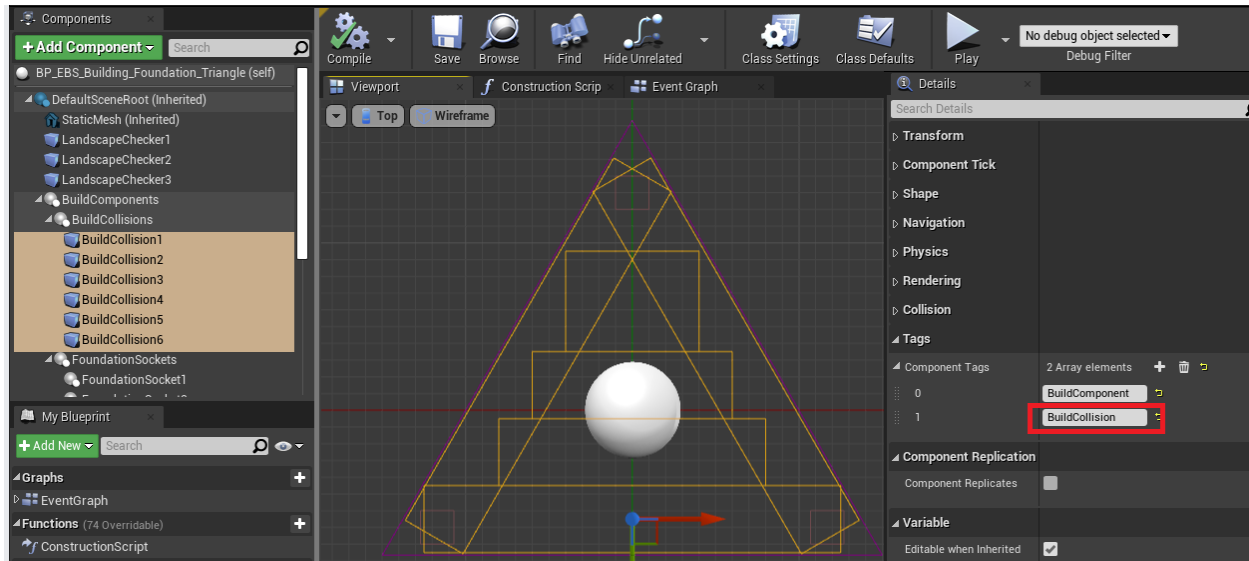
3.3.2.5. Functions

- **ManualSnapToTarget** - Snap to target object manually.
- **ManualSnapAndRotateNext** - Snap to target object with next rotation manually.
- **ManualSnapAndRotatePrev** - Snap to target object with prev rotation manually.
- **ManualRotate** - Rotate building object in place manually.
- **CheckAndAttachToTarget** - Check and attach to the target actor if it is possible.
Can be overridden in snap based child classes.
- **CheckBuildStatus** - Checks build status. Returns true if building is possible. Can be overridden in child classes.
- **CheckSnap** - Returns true if the building object can be snapped with the target actor. Can be overridden in snap based child classes.
- **GetSnapTransform** - Returns snap world transform with target actor. Can be overridden in snap based child classes.
- **CheckBuildCollisions** - Returns true if target actor does not overlap build collisions.
Can be overridden in child classes.
- **CheckActorCollisions** - Returns true if the target actor don't blocked by the target component or block building zone.
- **SetStartBuildCollisionResponseToOverlap** - Set collision response to overlap.
- **GetSnappedObjects** - Returns list of overlapping building objects.
- **CheckSupport** - Returns true if the building object has support. Can be overridden in child classes.
- **GetNearestTransform** - Returns the nearest transform to target location.
- **GetSocketTransform** - Returns socket transform by name and index. Can be overridden in snap based child classes.
- **BuildingObjectInSocket** - Returns true if the target object in one of the sockets.
Can be overridden in child snap based classes.
- **GetFloorWorldZ** - Returns world Z value of the floor. Can be overridden in floor based child classes.

-
- **CreateSocketTransforms** - Set the transform variables for the sockets. Can be overridden in snap based child classes.
 - **DestroyBuildComponents** - Remove unnecessary collisions and sockets. Destroys each component which has the **BuildComponent** tag. Can be overridden in child classes.
 - **CheckClaim** - Returns true if the building object is not claimed or claimed by target player.
 - **SetFloorNumberByTargetActor** - Set floor number by target actor. Can be overridden in child classes.
 - **CheckOwnership** - Returns true if building object owned by the target player.
 - **MakeSocketTransforms** - Make socket transforms. Can be overridden in child classes.
 - **MakeSocketTransformsPrimitive** - Make socket transforms by primitive component. Can be overridden in child classes.
 - **GetSocketTransforms** - Returns socket transforms by name. Can be overridden in snap based child classes.
 - **GetNearestSocketTransform** - Returns nearest socket transform to target location. Can be overridden in child classes.
 - **CompleteBuild** - Complete building for the player.
 - **CheckRepair** - Returns true if the target player can repair the object.
 - **CheckRepairRequirements** - Returns true if repair requirements complete for the player.
 - **CompleteRepair** - Repair building and complete repair requirements for the player.
 - **CheckUpgrade** - Returns true if the target player can upgrade the object.
 - **CheckUpgradeRequirements** - Returns true if upgrade requirements complete for the player.
 - **CompleteUpgrade** - Upgrade building and complete upgrade requirements for the player.

-
- **CompleteRemove** - Removes the building object from the game. Checks snapped object's support and removes them if needed. Destruct it, if flag is true.
 - **RemoveAttachedObjects** - Removes attached objects. For example: Door in DoorFrame. Overridden in child class.
 - **CompleteDestruction** - Check and destruct the object on the server.
 - **CheckRotate** - Returns true if the target player can rotate the object.
 - **CompleteRotate** - Rotate building object on the server.
 - **AttachTargetActorToSocket** - Attach target actor to the socket. Can be overridden in snap based child classes.
 - **GetAttachedActorBySocket** - Returns an attached actor that is associated with the target socket.
 - **CheckAndAttachToComponent** - Check and try to attach an actor to a specific socket of the component. Can be overridden in child classes.

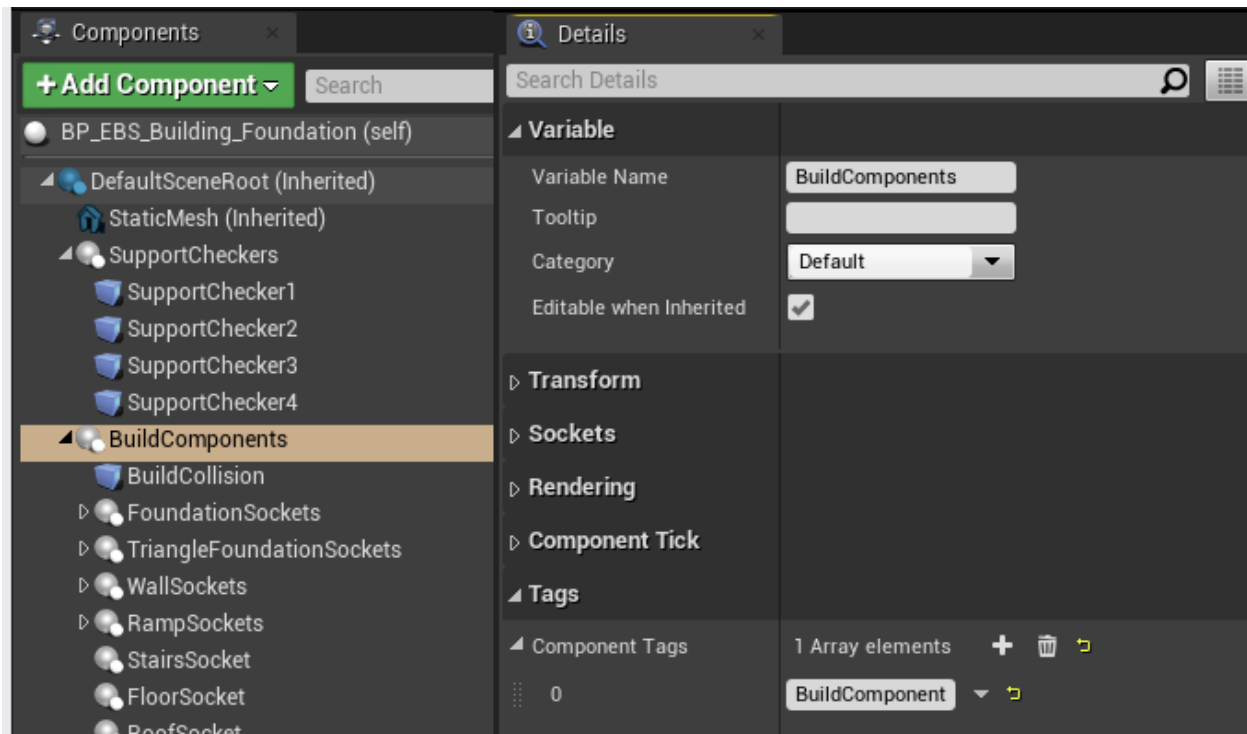
3.3.2.6. Build collisions



Building collisions are used to check the possibility of building. In building mode, if other objects overlap one of these collisions, you cannot build at the current point. You can use any collision component classes (box, sphere, capsule) and customize their sizes and offsets as desired. Building collision components must have a **BuildCollision** tag.

BuildCollision preset should be **OverlapAllDynamic**.

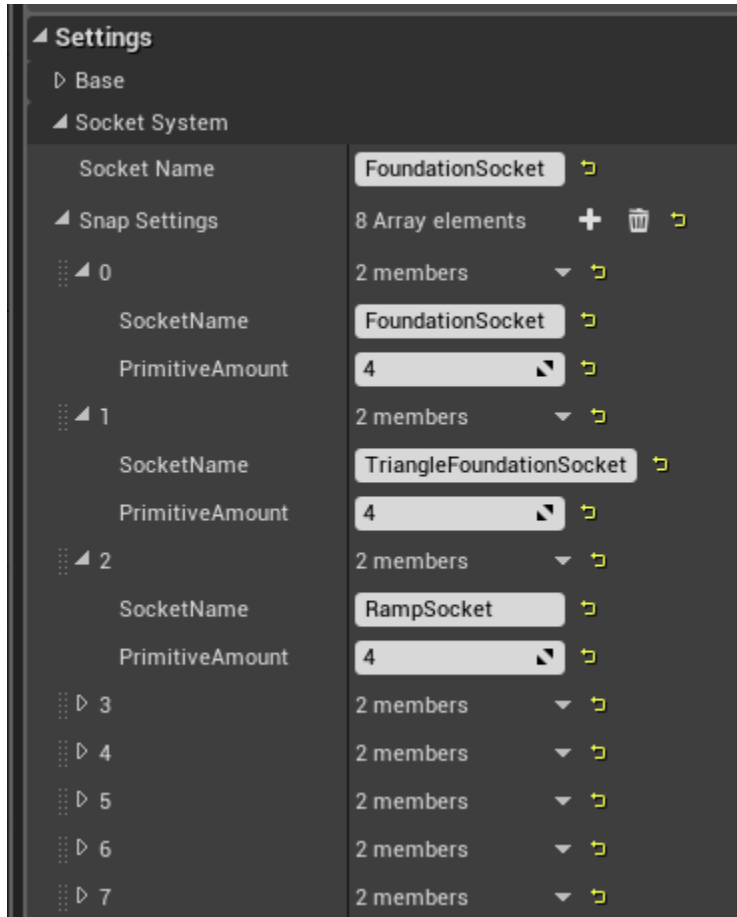
3.3.2.7. Build components



Some components of building classes are used only in building mode and are not needed later on after placement. Such components may include building collision components or socket scene components. These components can be automatically removed from the game after placing an object. In order to mark unnecessary components, you should add to them the **BuildComponent** tag.

3.3.2.8. Socket system

A socket system is used to connect modular building objects with each other. The settings for this system are set with the variables of the **Socket System** category inside each modular object.

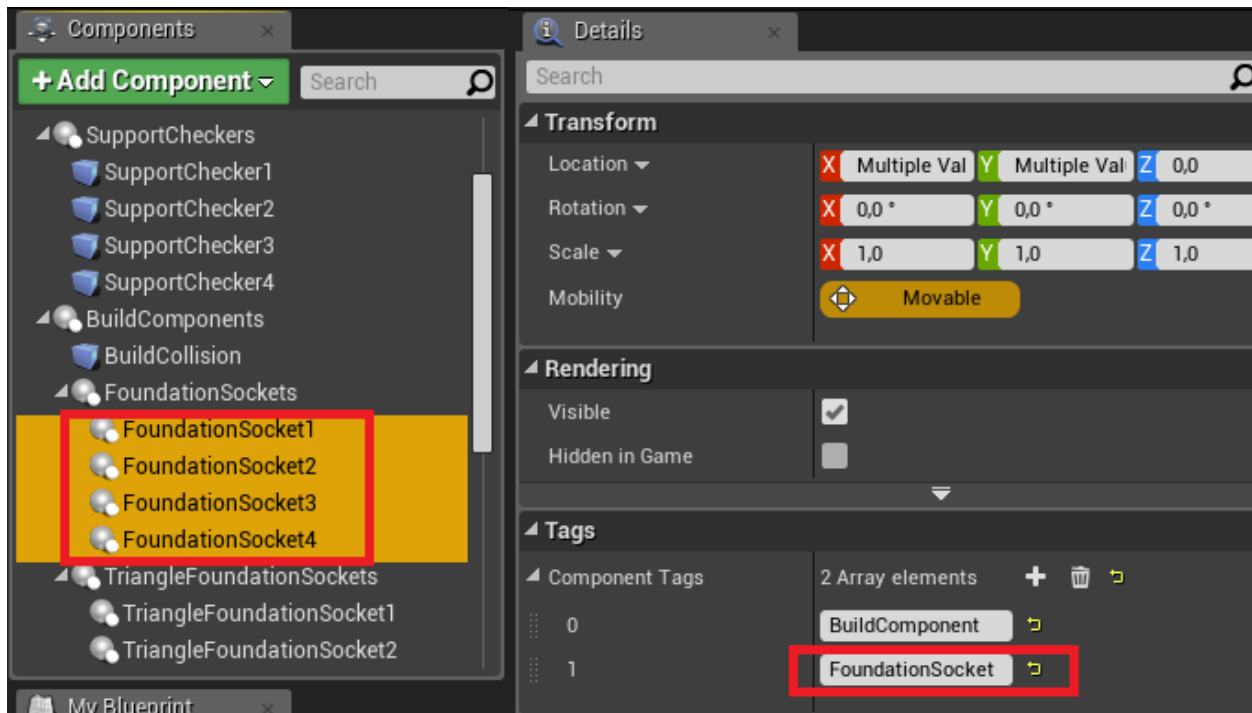


SocketName - defines the name of the socket for the object. Depending on this name the position of the component in the target object will be selected, with the same tag.

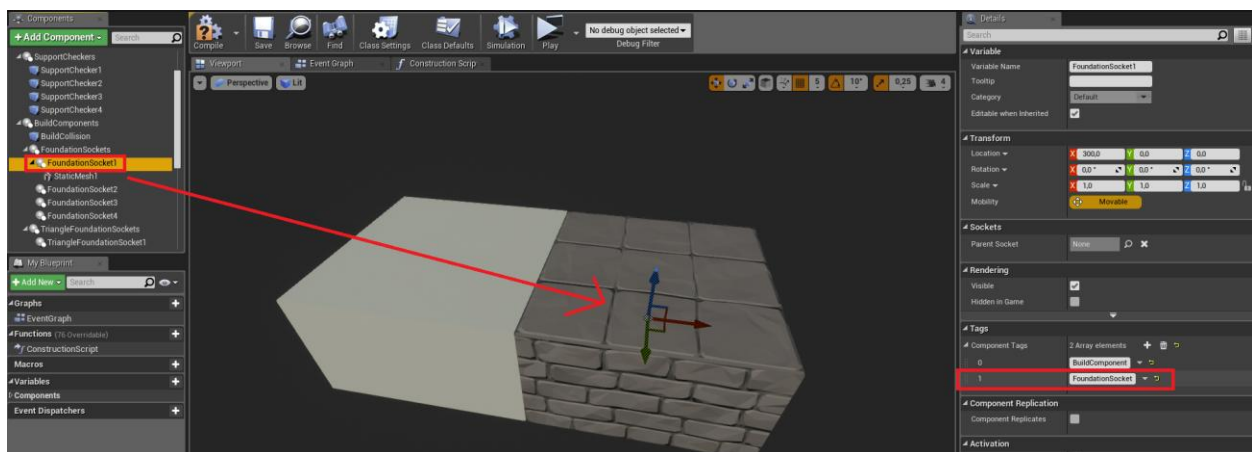
Snap Settings - an array of settings that specifies the ability to snapple with other objects. Each element in the array defines the name of a socket object that can snap to this object.

PrimitiveAmount - defines the number of sockets that are taken from the object mesh itself. It is used only when **UseMeshSockets** from the object's settings is enabled.

The component setting for the socket system determines the position of objects in the snap, as well as the name of the socket used for these objects. To configure it, move the components to the place where the other objects should be in the snapping and give them the appropriate tag.



To check the position of a socket in the object's viewport, you can use static meshes with the model of the object that is snapped to that socket.



3.3.3. Floor building object

3.3.3.1. Description

BP_Building_FloorObject - a class of the building object that can be placed on the floor or on the landscape. All floor objects must inherit from this class. Contains functions to check for the presence of the floor and remade snapping functions with objects that are floor.

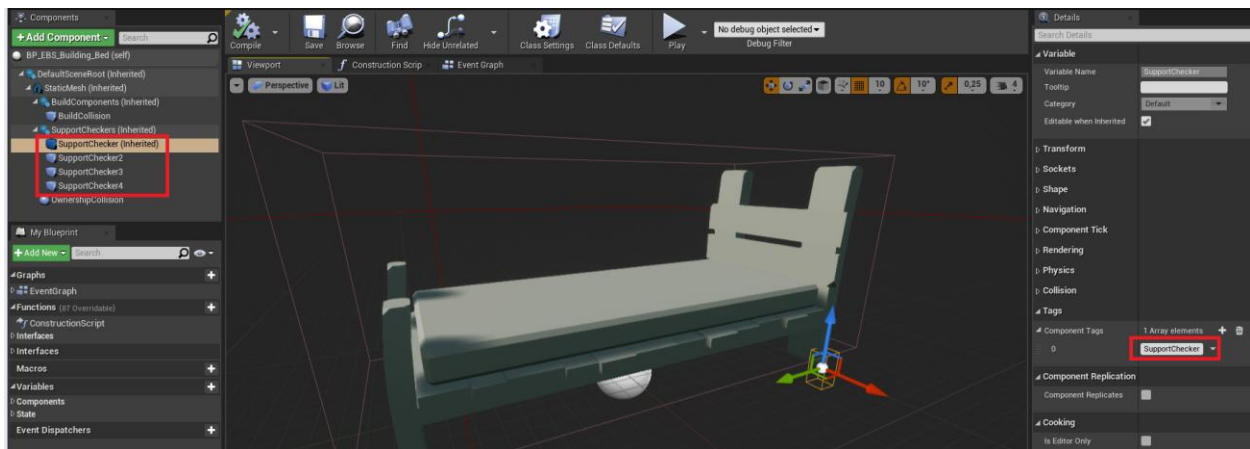
3.3.3.2. Variables

- **PlacedOnFloor** - if true, the object can be placed on the floor.
- **GridOffset** - offset of the object when the object is placed in the grid building mode.

3.3.3.3. Functions

- **GetGridOffsets** - Returns grid offsets if needed.
- **CheckFloor** - Returns true if array of actors contains floor.
- **CheckLandscape** - Returns true if array of actors contains landscape.

3.3.3.4. Check support settings



The support check is done with the support check components. These components can be any collision classes (box, sphere, capsule) with the **OverlapAllDynamic** collision preset and with the **CheckSupport** tag. When placing an object, all of these components must overlap the landscape or building object, which is considered the floor.

3.3.4. Wall building object

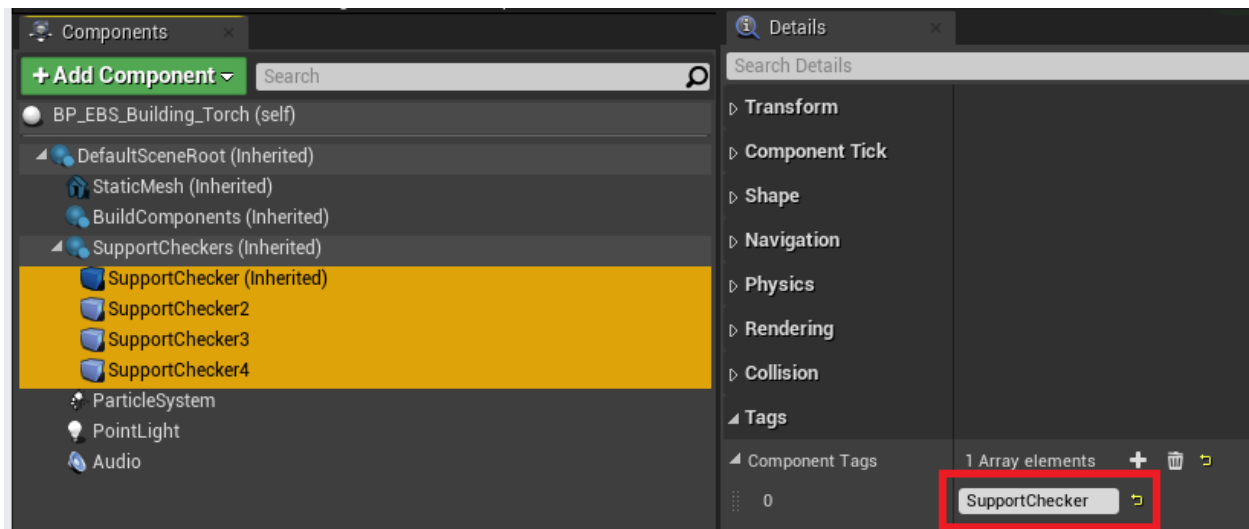
3.3.4.1. Description

BP_Building_WallObject - a class of building object that can be placed on walls. All wall objects must inherit from this class. Contains functions to check if there is a wall and reworked snapping functions with objects that are walls.

3.3.4.2. Functions

- **CheckWall** - Returns true if the array of actors contains walls.

3.3.4.3. Check support settings



The support check is done with the support check components. These components can be any collision classes with the **OverlapAllDynamic** collision preset and with the **CheckSupport** tag. When placing an object, all these components must overlap the building object, which is considered a wall.

3.3.5. Building Component

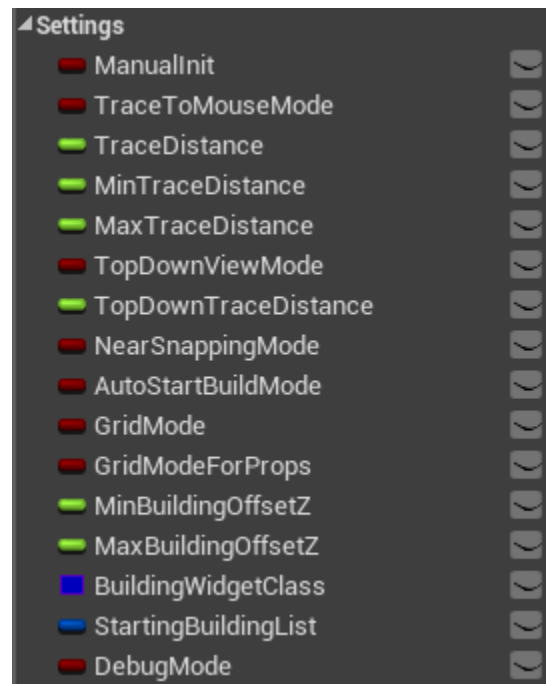
3.3.5.1. Description

BP_BuildingComponent - component that includes functionality for controlling the building process. It allows you to switch between different building modes, such as placing, repairing, upgrading, removing, destroying, and rotating. The component is easily configurable for both first-person, third-person and top-view camera placement. Features automatic initialization, creation or connection to a widget that manages the building menu and other building processes.

3.3.5.2. Variables

Settings

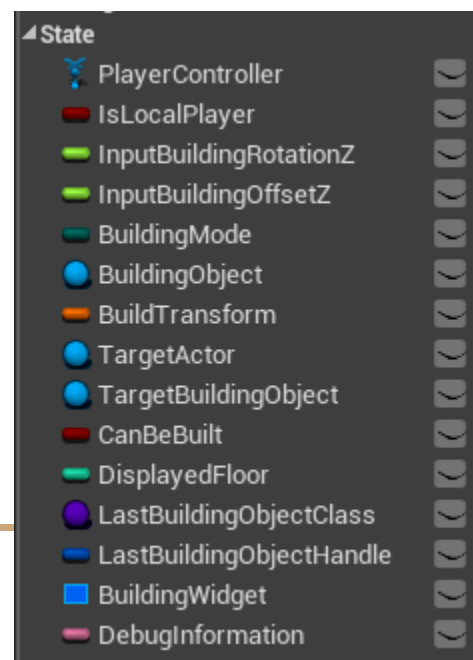
- **ManualInit** - If true, the component is not initialized automatically and must be initialized manually.
- **TraceToMouseMode** - if true, the target point or object will be searched for by trace from the player's camera to the mouse position on the screen. Otherwise, the trace is made to the center of the screen.
- **TraceDistance** - trace distance to find the target point or object.
- **MinTraceDistance** - the minimum value limiting the dynamic variation of the variable trace distance.
- **MaxTraceDistance** - the maximum value limiting the dynamic variation of the variable trace distance.



- **TopDownViewMode** - enables top view mode.
- **TopDownTraceDistance** - trace distance to find the target point or object in the top view mode.
- **NearSnappingMode** - if true, then when snapping objects to each other, the socket closest to the camera will be selected. Otherwise, you can select the desired socket by rotating, which is done with the mouse wheel.
- **AutoStartBuildMode** - if true, then after the successful placement of an object, the process of placing such an object will automatically start.
- **GridMode** - enables grid-based building mode.
- **GridModeForProps** - enables grid-based building mode for floor objects.
- **MinBuildingOffsetZ** - lower limit for object height offset during the placement process.
- **MaxBuildingOffsetZ** - upper limit for object height offset during the placement process.
- **BuildingWidgetClass** - a widget class that is created or located and bound to the building component when the component is initialized.
- **StartingBuildingList** - the list of building objects that are loaded in the menu of the building widget when the component is initialized. By default, the list of the widget itself is loaded.
- **DebugMode** - enables the debug mode, which outputs the necessary information for debugging.

State

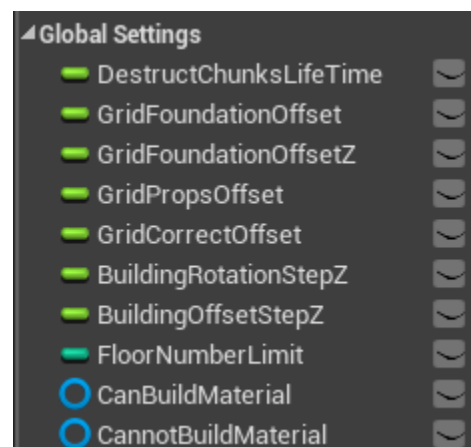
- **PlayerController** - reference to the controller of the player controlling the component.
- **IsLocalPlayer** - local player identifier. Used to update the building process only on the client machine.



- **InputBuildingRotationZ** - the input value of the building rotation in the building mode. Dynamically changed by the player.
- **InputBuildingOffsetZ** - input value of the building's Z-axis offset in the building mode. Dynamically changed by the player.
- **BuildingMode** - current building mode.
- **BuildingObject** - reference to the current building object.
- **BuildTransform** - the current position for the location of the building object.
- **TargetActor** - reference to the target actor after the trace.
- **TargetBuildingObject** - reference to the target building object after the trace.
- **CanBeBuilt** - identifier that the building object can be placed in the current position.
- **DisplayedFloor** - current displayed floor in top view mode.
- **LastBuildingObjectClass** - class of the last placed building object. Used in automatic building start mode.
- **LastBuildingObjectHandle** - identifier of the last placed building object. Used in the automatic building start mode.
- **BuildingWidget** - reference to building widget.
- **DebugInformation** - debugging information, which is formed in the debug mode of the component.

Global Settings

- **DestructChunksLifeTime** - the time of life of parts of objects when they are destroyed.
- **GridFoundationOffset** - grid offset distance for foundations.
- **GridFoundationOffsetZ** - grid offset distance along the Z-axis for the foundation.
- **GridPropsOffset** - grid offset distance for floor objects.
- **GridCorrectOffset** - corrective offset of the grid relative to the world.



-
- **BuildingRotationStep** - the step for changing the **InputBuildingRotationZ** variable. Also used in the socket system.
 - **BuildingOffsetHeigh** - the step for changing the **InputBuildingOffsetZ** variable.
 - **FloorNumberLimit** - maximum floor for the placement of objects. If the value is less than 1, there are no restrictions.
 - **CanBuildMaterial** - the material of the building object, when placement is possible.
 - **CannotBuildMaterial** - the material of the building object, when placement is not possible.

3.3.5.3. Functions

- **InitComponent** - Try to init the component and set references. Should be called manually if the **ManualInit** variable is true.
- **AutoInitComponent** - Checks and inits component automatically in the **BeginPlay** event, if the **ManualInit** variable is disabled.
- **CheckInitReferences** - Init component and check references. Repeat again if references are not valid.
- **ChangeBuildingRotationZ** - Change the **InputBuildingRotationZ** value for the building process. It can be in range 0 - 360.
- **ChangeBuildingOffsetZ** - Change the **InputBuildingOffsetZ** value for the building process. It can be in range **MinBuildingOffsetZ** - **MaxBuildingOffsetZ**.
- **ChangeMinBuildingOffsetZ** - Change the **MinBuildingOffsetZ** value for the building process.
- **ChangeMaxBuildingOffsetZ** - Change the **MaxBuildingOffsetZ** value for the building process.
- **ChangeTraceDistance** - Change the **TraceDistance** value for the building process. It can be in range **MinTraceDistance** - **MaxTraceDistance**.
- **ChangeMaxTraceDistance** - Change the **MaxTraceDistance** value for the building process.
- **ChangeDisplayedFloor** - Change the **DisplayedFloor** value and hide building actors if their floor number is greater than displayed floor.
- **ChangeTopDownViewMode** - Change top down view mode for building process.
- **ChangeGridMode** - Change grid mode for building process.
- **ChangeNearSnappingMode** - Change near snapping mode for building process.
- **ChangeBuildingMode** - Change building mode for building process.
- **ChangeDebugMode** - Change debug mode for building process.
- **StartBuildObject** - Start to build an object by building handle and activate building mode.

-
- **CancelBuild** - Cancel build and reset building mode.
 - **TryBuild** - Try to complete build the object if it can be built.
 - **TryRemove** - Try to remove the target building object.
 - **TryDestruct** - Try to destruct the target building object.
 - **TryRepair** - Try to repair the target building object.
 - **TryUpgrade** - Try to upgrade the target building object.
 - **TryRotate** - Try to rotate the target building object.
 - **FinishBuild** - Finish build and spawn the building object on the server.
 - **TryQuickInteraction** - Try to do interaction depending on current building mode.
 - **TryStartBuildObject** - Check building requirements on the server and start to build the object.
 - **DestroyBuildingObject** - Destroy building object and clear variable.
 - **CheckBuildingRequirements** - Returns true if building requirements are complete.
 - **CompleteBuildingRequirements** - Complete building requirements.
 - **SendBuildingMessage** - Send building message for player.
 - **GetBuildingMode** - Returns current building mode.
 - **GetBuildingStatus** - Returns current building status and references.
 - **GetInteractionStatus** - Returns current interaction mode and references.
 - **PrintDebugInformation** - Print debug information depending on current building mode.
 - **UpdateFloorActorsVisibility** - Hide floor actors if their floor number is greater then displayed floor number.
 - **ResetFloorActorsVisibility** - Reset floor number visibility.
 - **GetDebugInformation** - Returns debug information.
 - **UpdateProcess** - Update building process for the local player.
 - **UpdateBuildingPosition** - Update position of the current building object.
 - **UpdateTargetActor** - Update target actor and target building object.
 - **GetTraceHitResult** - Returns hit result after line trace.

-
- **SetBuildingObjectTransform** - Set building object transform. Returns true if the transform was changed.
 - **SetBuildingObjectMaterials** - Set building object materials.
 - **SetCanBeBuilt** - Set build status variable and update building object materials.
 - **UpdateBuildStatus** - Update build status for building process.
 - **GetFloorIgnoringActors** - Returns actors whose floor number is higher than the target floor number.
 - **GetFoundationTransform** - Returns correct transform for placing foundations depending on hit location and current building modes.
 - **GetPropTransform** - Returns correct transform for placing props depending on hit location and current building modes.
 - **GetCorrectHitLocation** - Returns correct hit location for building process with building Z offset.
 - **GetCorrectBuildingRotation** - Returns correct building rotation for building process in the current view mode.
 - **SetTargetActor** - Set target actor reference.
 - **SetTargetBuildingObject** - Set target building object reference.
 - **CheckFloorNumber** - Update floor number for building object and check floor number limit.
 - **TryToCreateBuildingWidget** - Try to create a building widget and add it to the viewport.
 - **ShowBuildingMenu** - Show building menu from building widget.
 - **HideBuildingMenu** - Hide building menu from building widget.
 - **UpdateBuildingList** - Update building list in the building widget.
 - **SetBuildingWidget** - Set building widget reference.
 - **ShowMalletMenu** - Show mallet menu from building widget.
 - **HideMalletMenu** - Hide mallet menu from building widget.
 - **ChangeBuildingWidget** - Change building widget class and create it for the player.

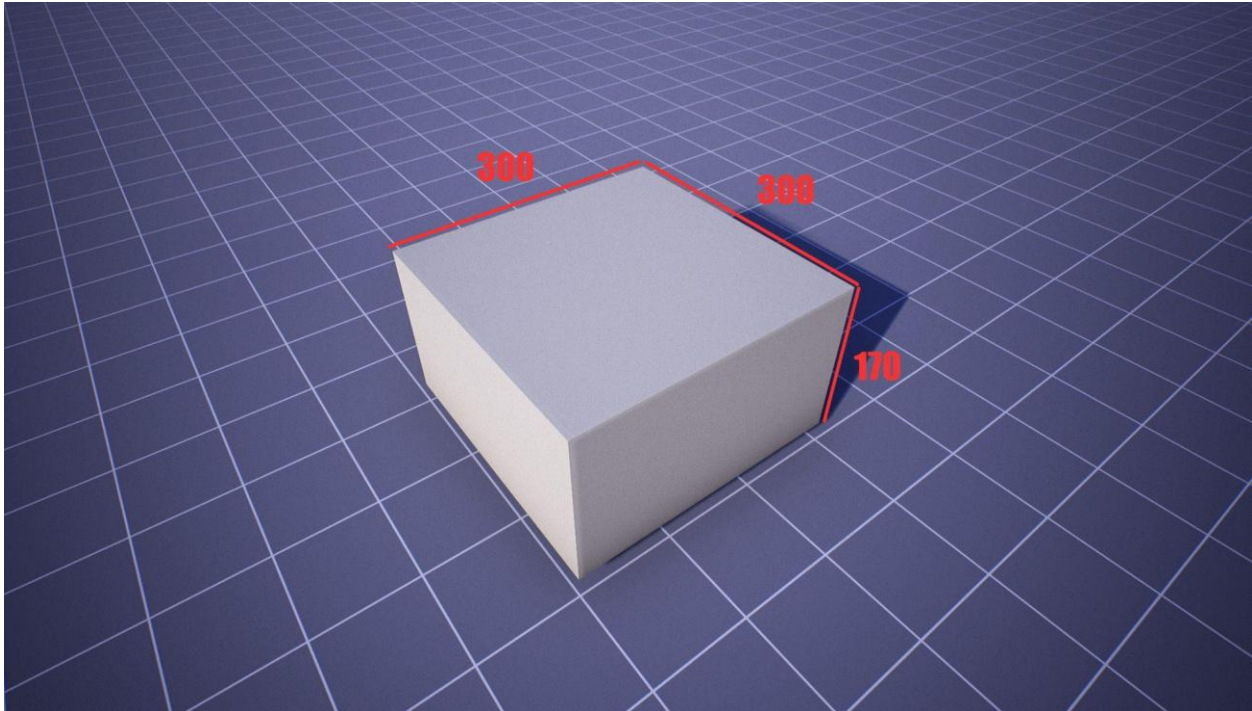
3.3.5.4. Event dispatchers

- **OnBuildingModeChanged** - calls when building mode was changed on client.
- **OnBuildingMessageReceived** - calls when building message was received on the client.
- **OnBuildingInteractionCompleted** - calls when building interaction was completed on the server.
- **OnBuildingCompleted** - call when building was completed on the server.
- **OnRemovingCompleted** - calls when removing was completed on the server.
- **OnDestructionCompleted** - calls when destruction was completed on the server.
- **OnRepairingCompleted** - calls when repairing was completed on the server.
- **OnUpgradingCompleted** - calls when upgrading was completed on the server.
- **OnRotationCompleted** - calls when rotation was completed on the server.
- **OnTargetActorChanged** - calls when the target actor was changed on the client.
- **OnTargetBuildingObjectChanged** - calls when the target building object was changed on the client

3.3.6. Modular building objects

3.3.6.1. Foundation

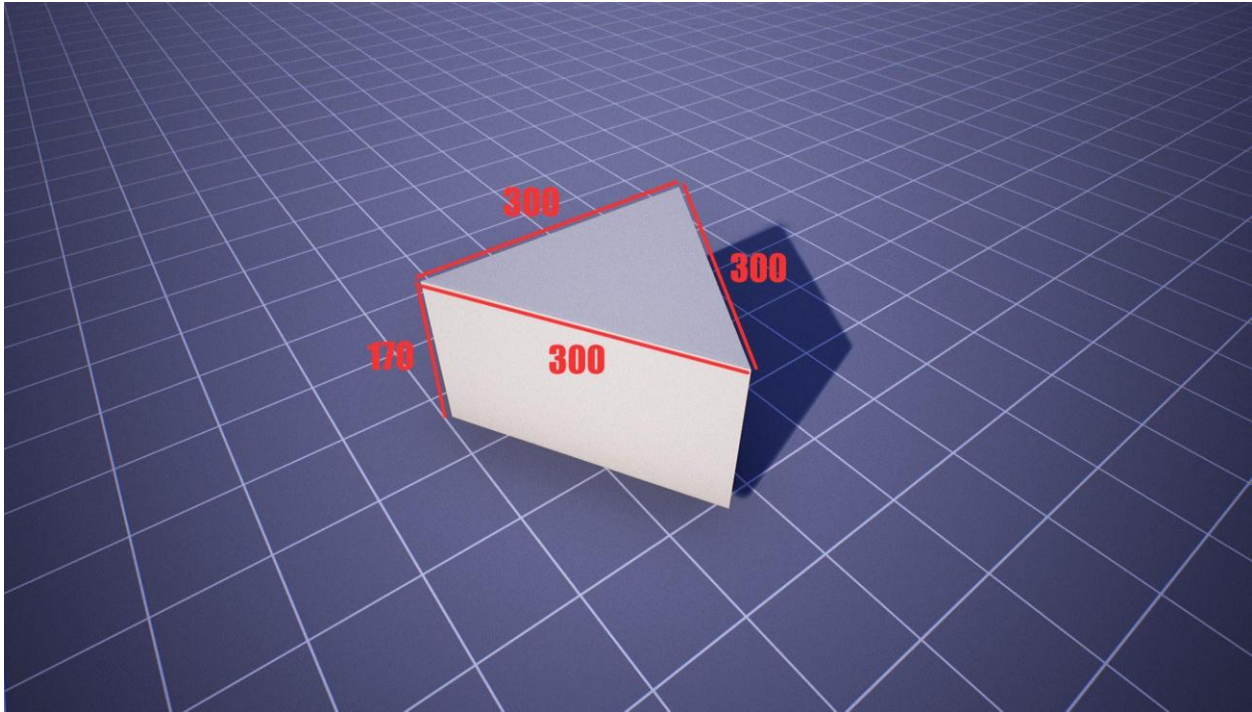
BP_Building_Foundation.



- Can be placed on landscape
- Can be a support for floor objects
- Can snap with any foundations
- Can be a snap target for any foundations, ramps, any walls, fences, stairs and any roofs

3.3.6.2. Triangle foundation

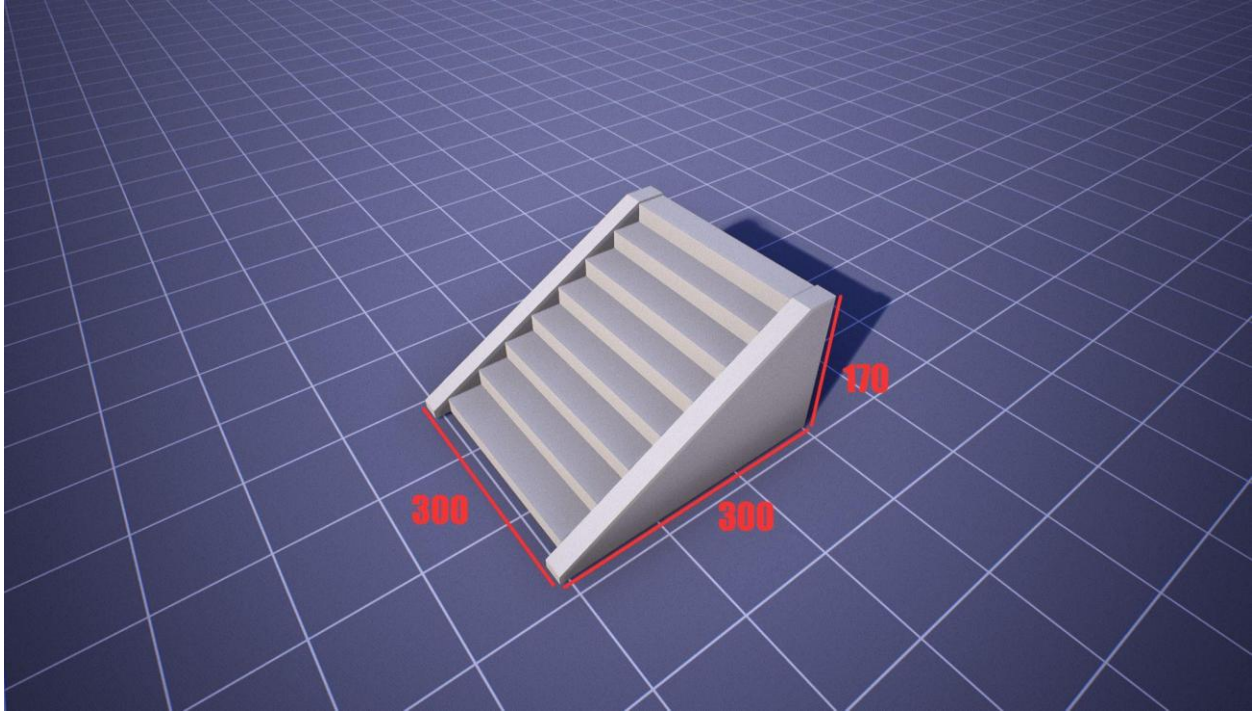
BP_Building_Foundation_Triangle.



- Can be placed on landscape
- Can be a support for floor objects
- Can snap with any foundations
- Can be a snap target for any foundations, any walls, fences and ramps

3.3.6.3. Ramp

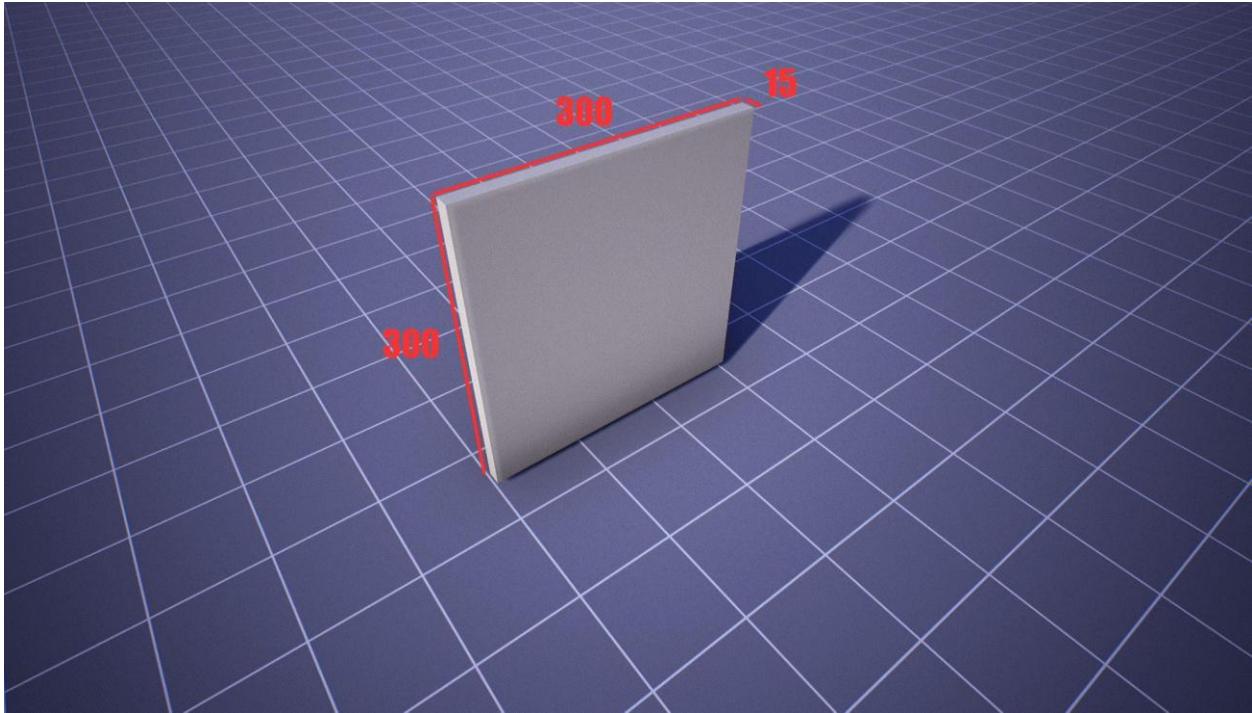
BP_Building_Ramp.



- Checks landscape support
- Can snap with any foundations

3.3.6.4. Wall

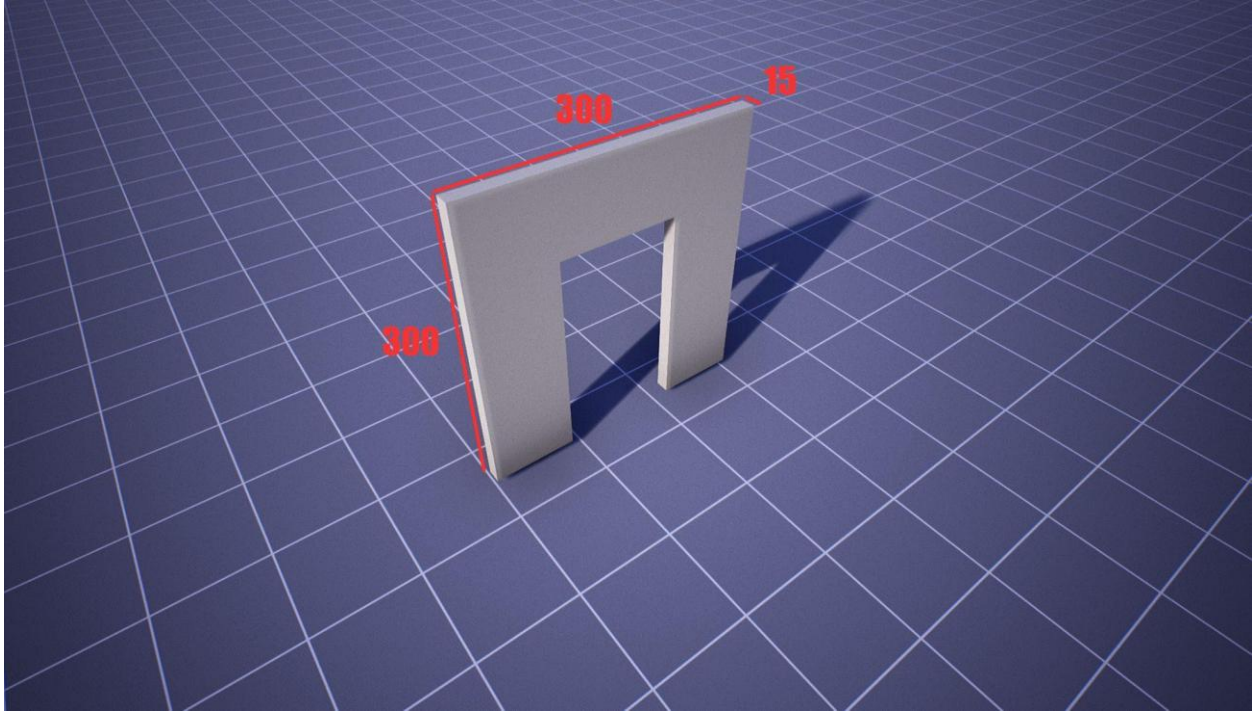
BP_Building_Wall.



- Can be a support for wall objects
- Can snap with any foundations, any ceilings, walls, doorframes and windowframes
- Can be a snap target for any walls, any ceilings, fences and roofs

3.3.6.5. Door frame

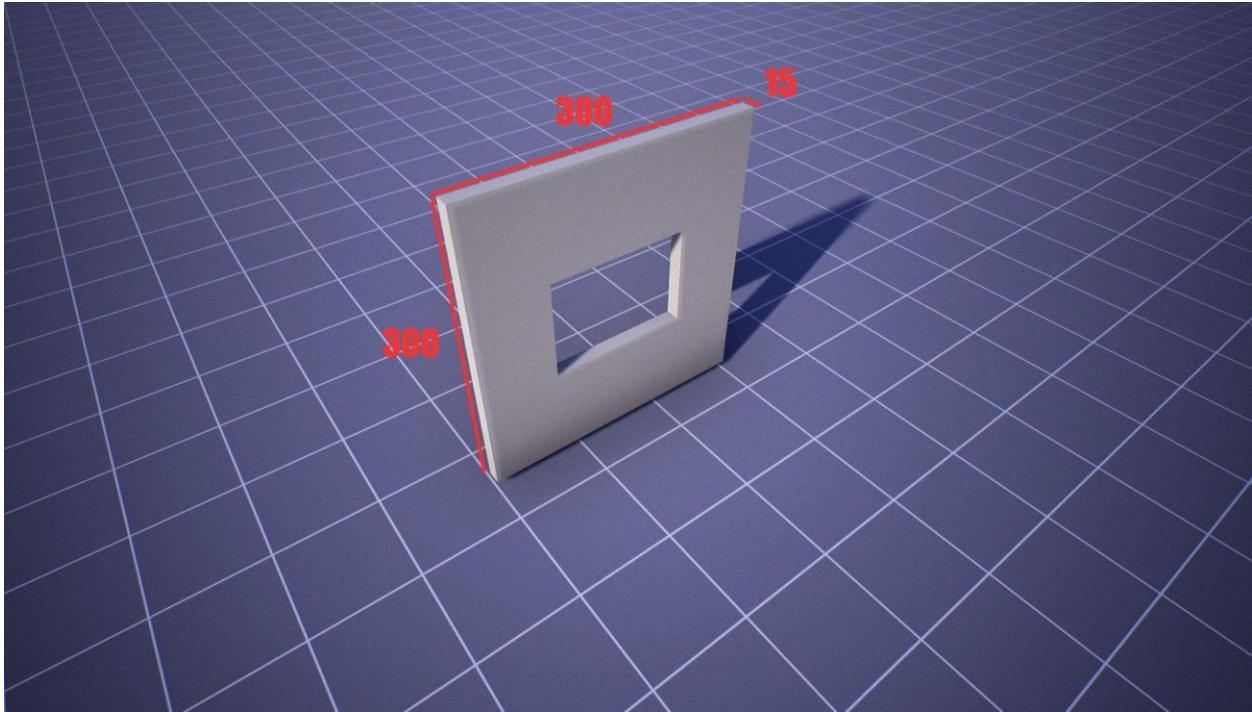
BP_Building_DoorFrame.



- Can be a support for wall objects
- Can snap with any foundations, any ceilings, walls, doorframes and windowframes
- Can be a snap target for any walls, any ceilings, fences and roofs

3.3.6.6. Window frame

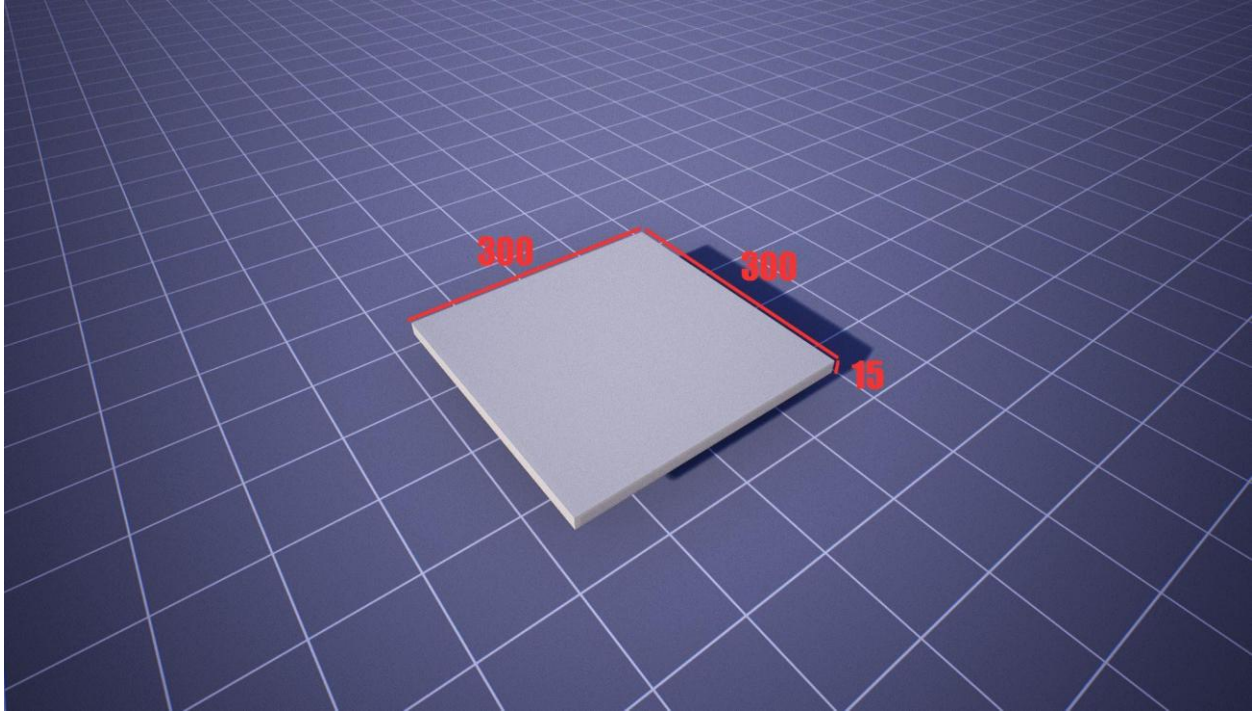
BP_Building_WindowFrame.



- Can be a support for wall objects
- Can snap with any foundations, any ceilings, walls, doorframes and windowframes
- Can be a snap target for any walls, any ceilings, fences and roofs

3.3.6.7. Ceiling

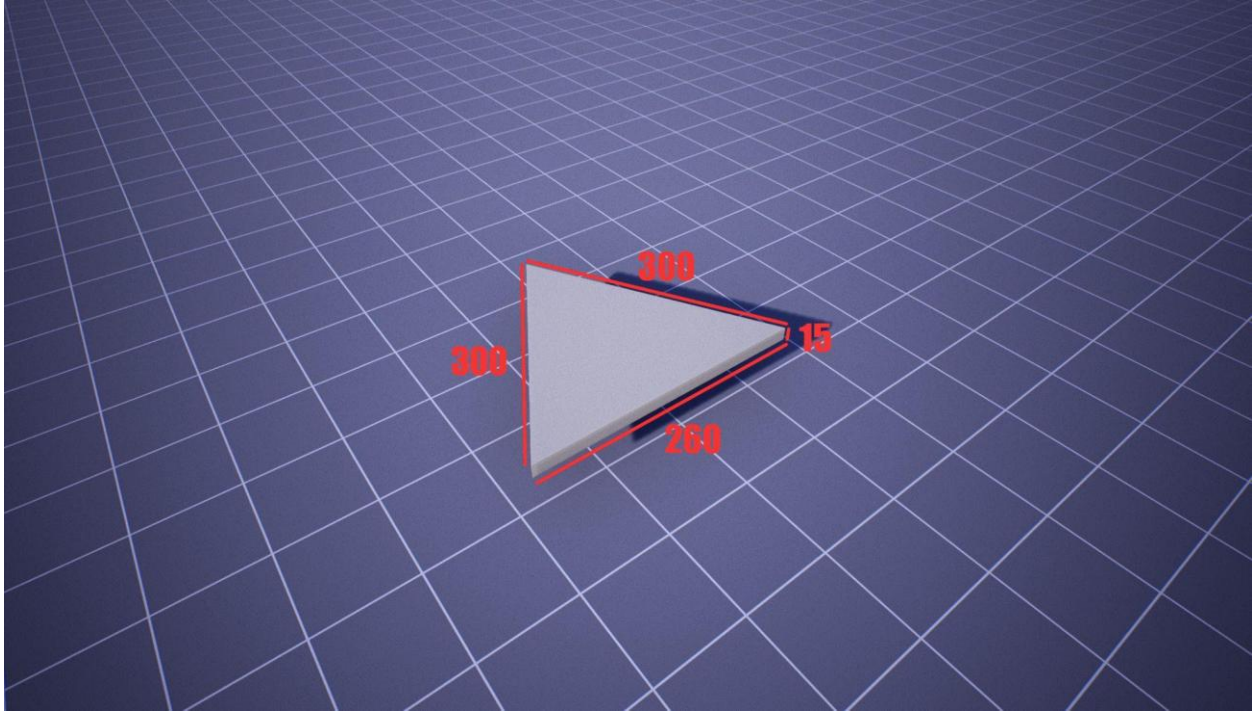
BP_Building_Ceiling.



- Can be a support for floor objects
- Can snap with any ceilings, walls, door frames and window frames
- Can be a snap target for any ceilings, any walls, fences, any roofs and stairs

3.3.6.8. Triangle ceiling

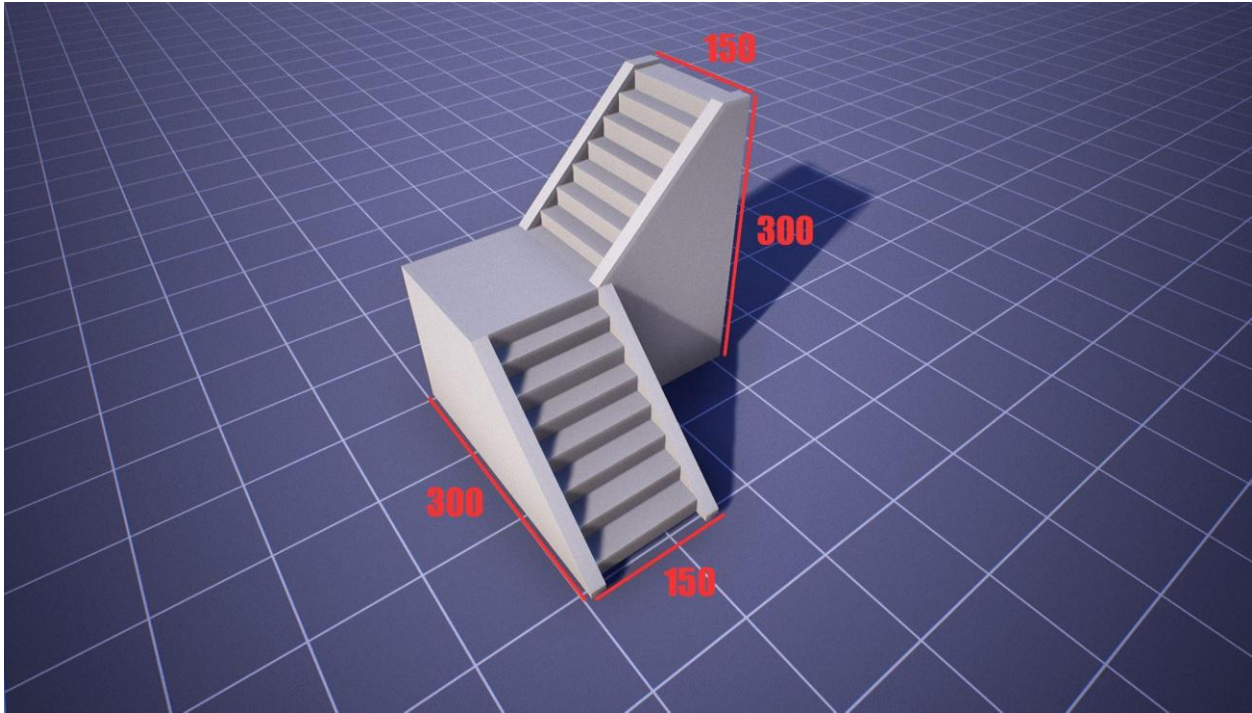
BP_Building_Ceiling_Triangle.



- Can be a support for floor objects
- Can snap with any ceilings, walls, door frames and window frames
- Can be a snap target for any ceilings, any walls and fences

3.3.6.9. Stairs

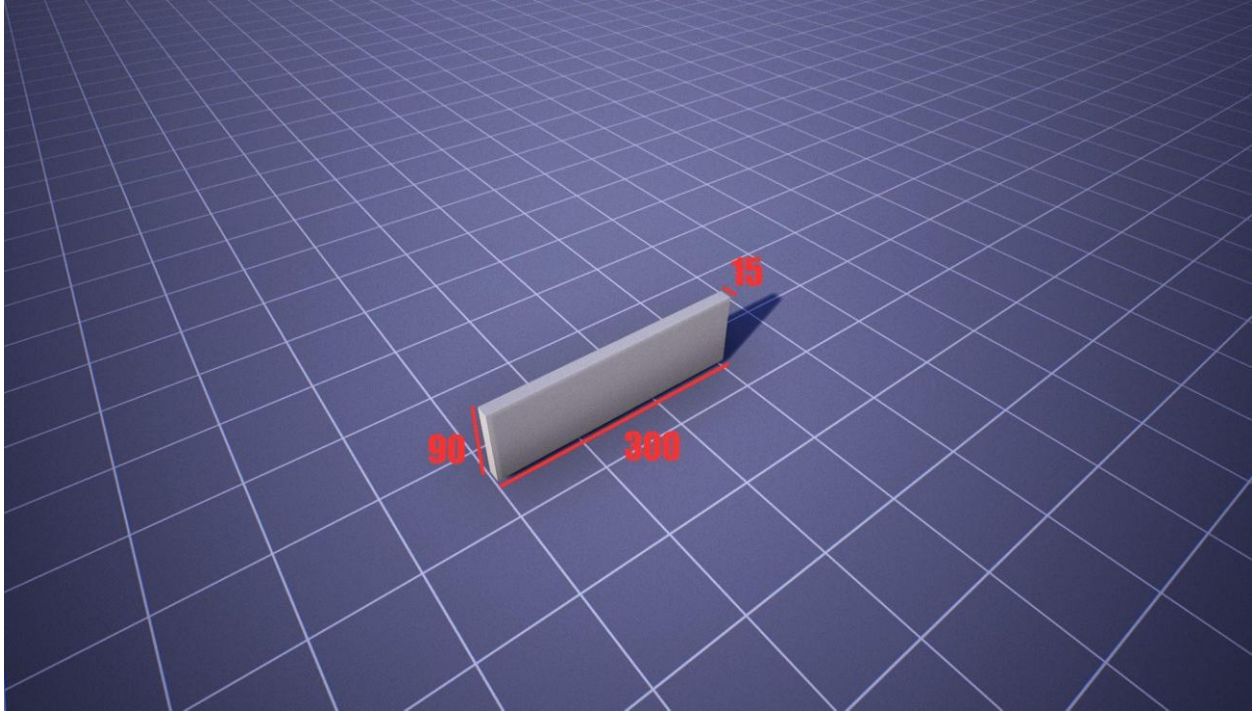
BP_Building_Stairs.



- Can snap with foundation and ceiling

3.3.6.10. Fence

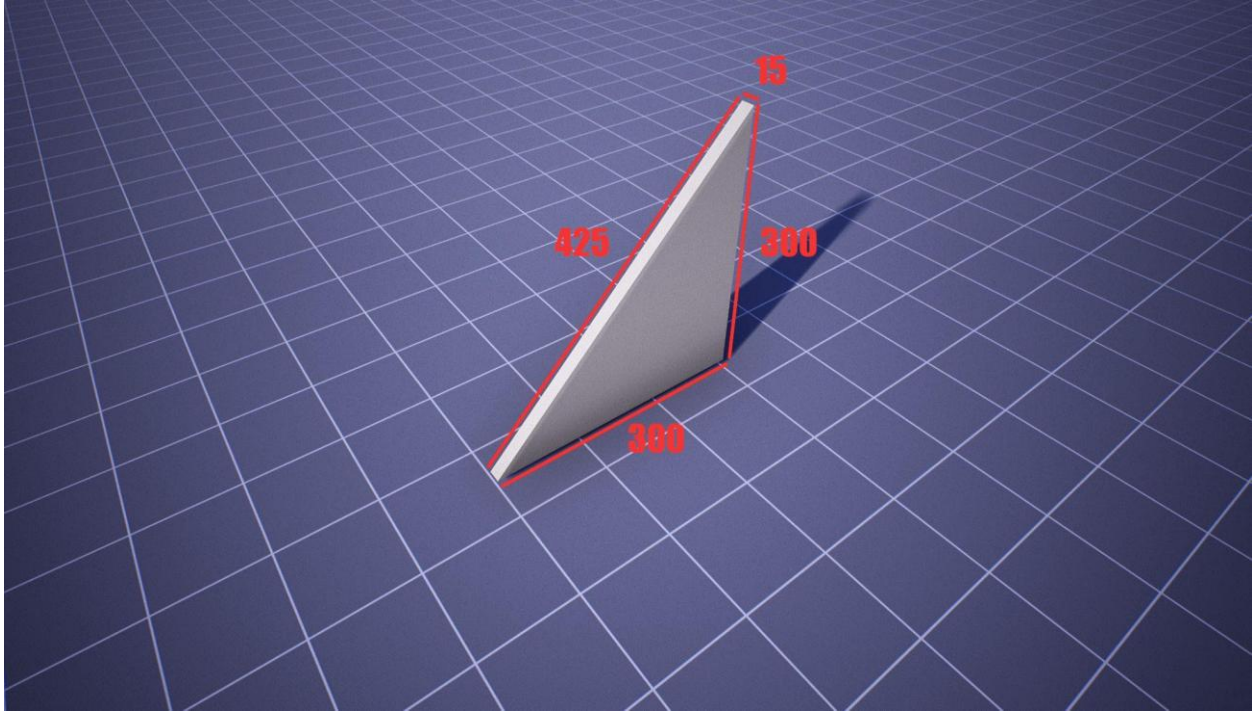
BP_Building_Fence.



- Can snap with any foundations, any ceilings, walls, door frames and window frames.

3.3.6.11. Roof wall left

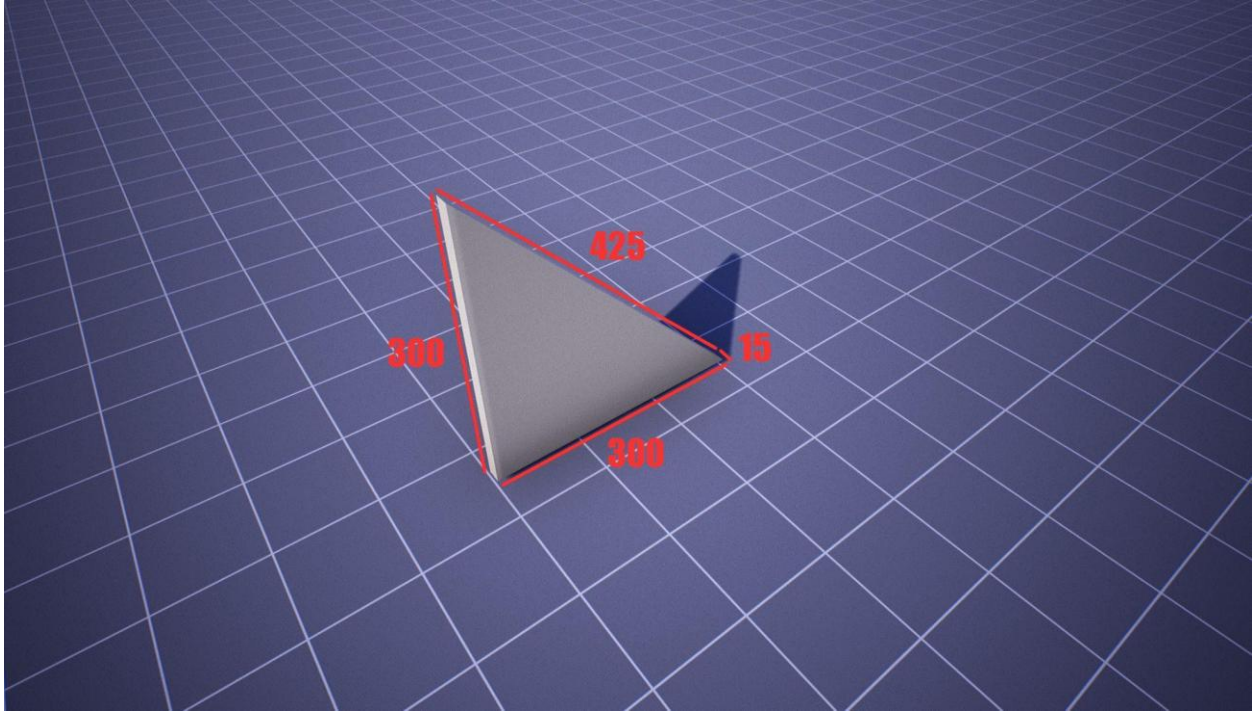
BP_Building_RoofWall_Left.



- Can be a support for wall objects
- Can snap with any foundations, any ceilings, walls, doorframes and windowframes
- Can be a snap target for roofs

3.3.6.12. Roof wall right

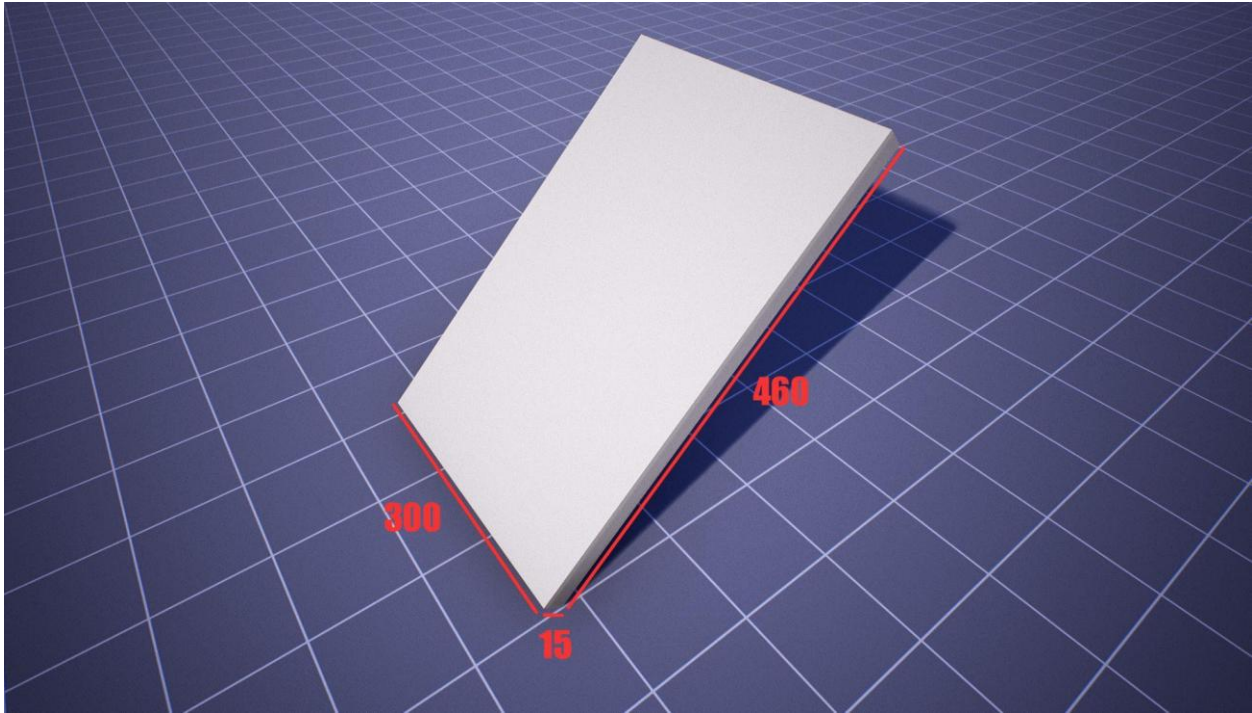
BP_Building_RoofWall_Right.



- Can be a support for wall objects
- Can snap with any foundations, any ceilings, walls, doorframes and windowframes
- Can be a snap target for roofs

3.3.6.13. Roof

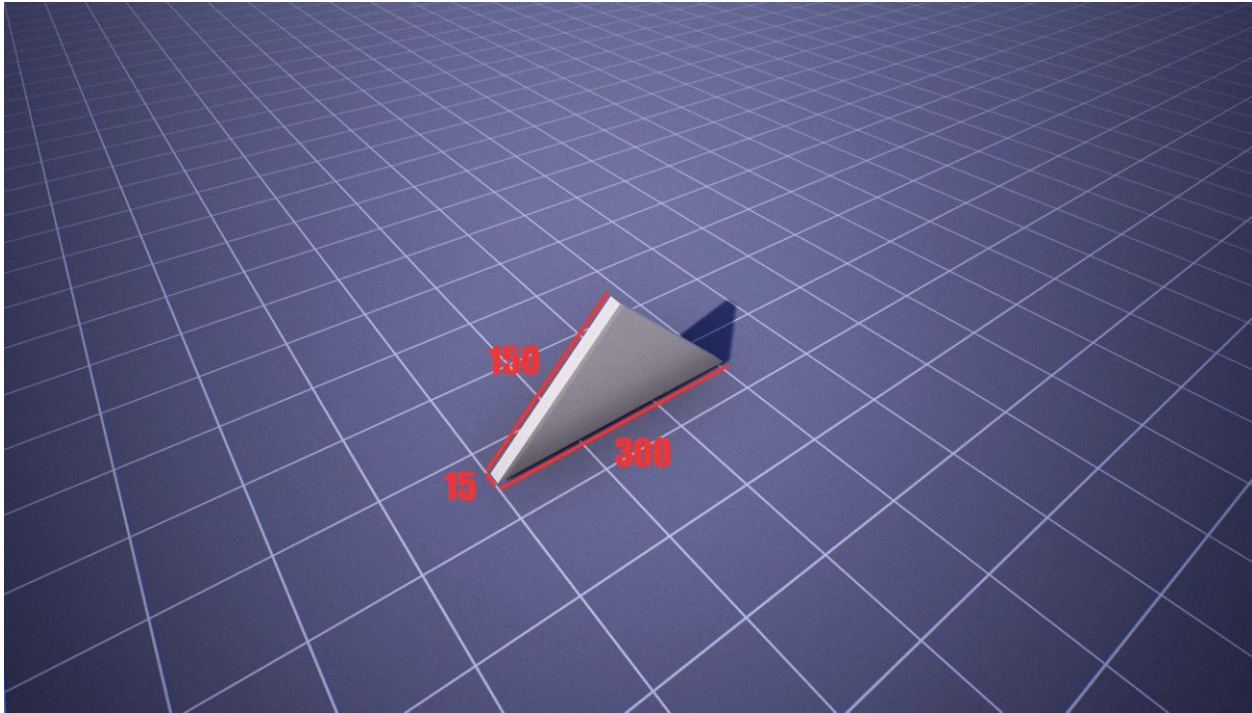
BP_Building_Roof.



- Can snap with foundations, ceilings, walls, doorframes, windowframes and roof walls
- Can be a snap target for roofs

3.3.6.14. Top roof wall

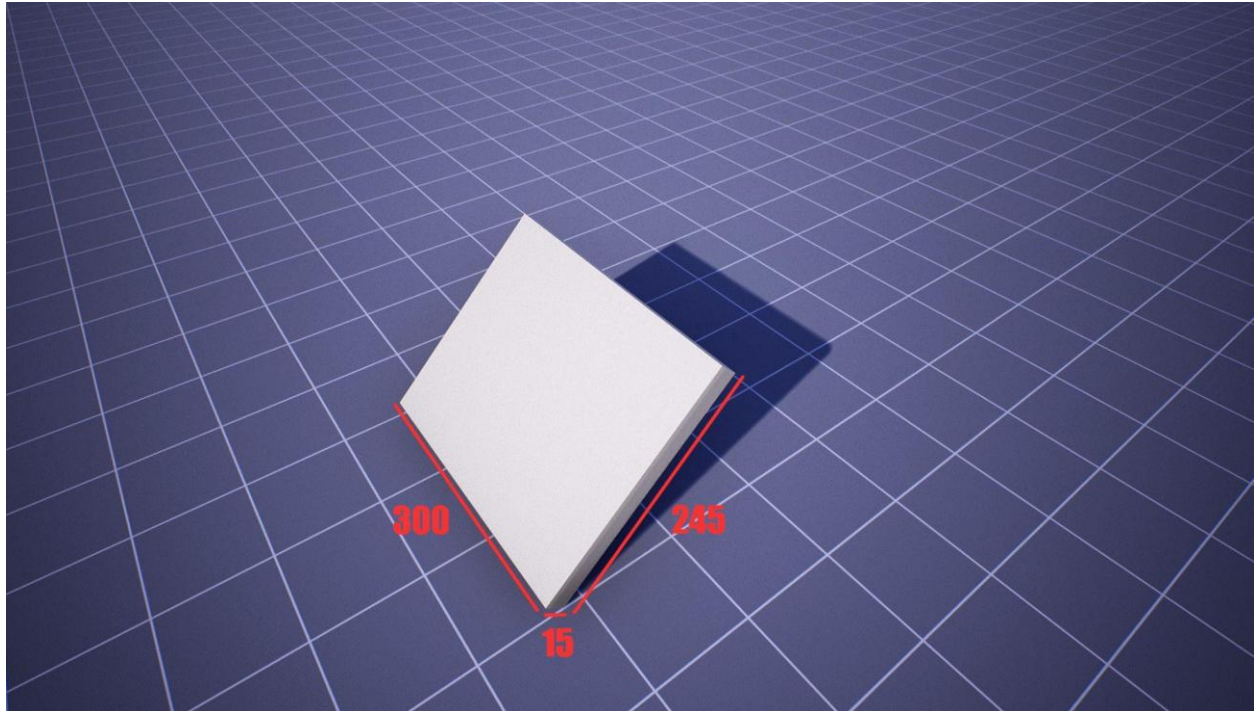
BP_Building_RoofWall_Top.



- Can be a support for wall objects
- Can snap with any foundations, any ceilings, walls, doorframes and windowframes
- Can be a snap target for roofs

3.3.6.15. Top roof

BP_Building_Roof_Top.



- Can snap with foundations, ceilings and top roof walls
- Can be a snap target for top roofs

3.4. Replaceable Instance System

3.4.1. Description

To optimize the huge number of objects located in the world with which you can interact, a system of instances was created. Trees, interactive ferns and harvested resources are placed in the world using the **Foliage Tool**, and in the game the system automatically replaces them with the necessary actors.

The general functionality of dynamically replacing an instance with an actor is in the **BP_InstancedComponent_Base** class. For each type of object that will be replaced during the game, you need to create a child class with individual settings.




The search for meshes that need to be replaced with actors is performed using the **BP_FoliageCheckerComponent** class. This component must be added to the character blueprint.

Additional collision types were created and configured to determine the objects to be replaced.

Name	Default Response
StaticFoliage	Ignore
FoliageChecker	Ignore
DynamicFoliage	Ignore

▲ Preset

You can modify any of your project profiles. Please note that if you modify profile, it can change collision behavior. Please be careful when you change currently existing (used) collision profiles.

Name	Collision	Object Type	Description
 StaticFoliage	Collision Enabled (Query and Physics)	StaticFoliage	Needs description
 FoliageChecker	Collision Enabled (Query and Physics)	FoliageChecker	Needs description
 DynamicFoliage	Collision Enabled (Query and Physics)	DynamicFoliage	Needs description

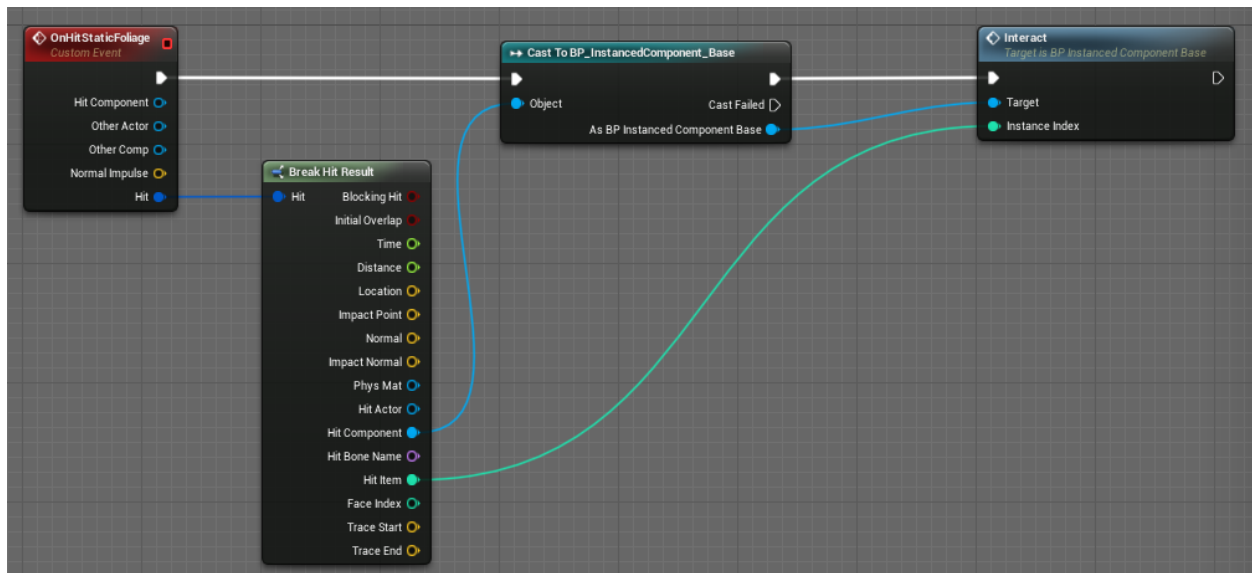
The collision of the **StaticFoliage** type must be assigned to the meshes, which will be dynamically replaced with actors. It must definitely block a collision with **FoliageChecker**.

Only the **BP_FoliageChecker** actor should have a **FoliageChecker** collision.

Collision type **DynamicFoliage** designed for physics interactive ferns.

The foliage checker actor should not affect the tree foliage. Trees should be replaced only when they get hits from damage functions.

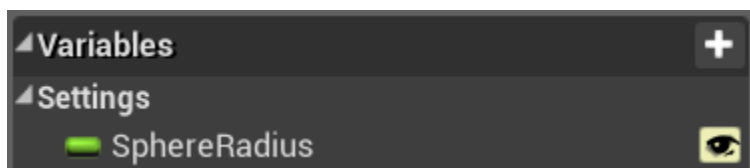
The **BP_FoliageCheckerComponent** class finds and automatically replaces instances with actors. It creates an actor with a collision of a certain radius and moves it to the position of the game character by tick. When colliding with instances, the **OnHitStaticFoliage** event is called, which calls the function of interacting with **BP_InstancedComponent_Base** and replaces the instance with an actor.



This component must be added to the player character class.

This component is not replicated.

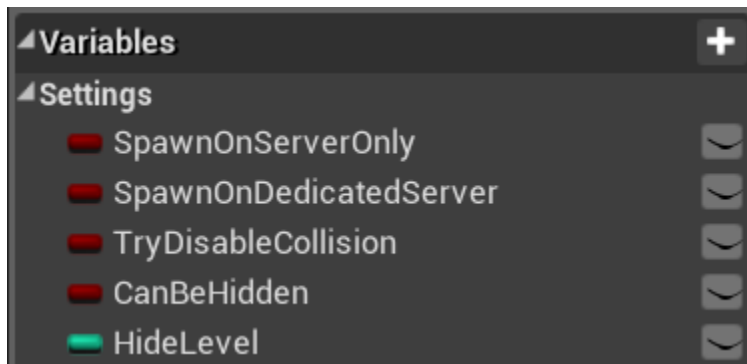
You can configure the interaction radius in the **SphereRadius** variable of the **BP_FoliageCheckerComponent** class.



Instance meshes must have a collision, which the checker collision can collide with. You can add or set up collisions inside the mesh itself.

3.4.3. BP_InstancedComponent_Base

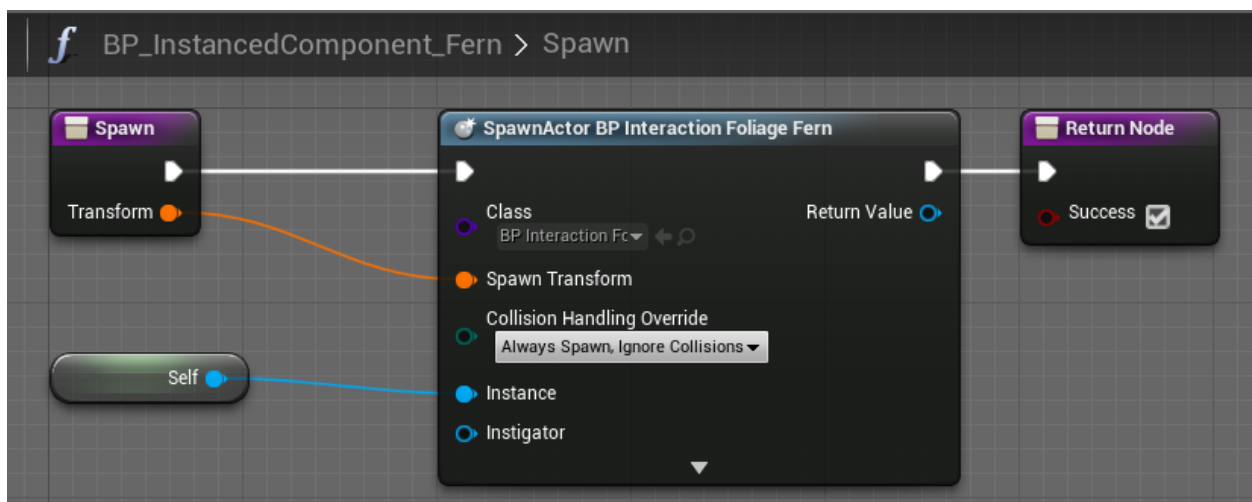
The **BP_InstancedComponent_Base** class provides general functionality for replacing instances with actors. For each type of object that will be replaced during the game, you need to create a child class with individual settings.



The **Interact** function removes the instance and calls the **Spawn** function to spawn the appropriate actor. The **SpawnOnServerOnly** variable should be true, if the actor is replicated. The **SpawnOnDedicatedServer** variable should be false, if the actor is not replicated and does not affect gameplay.

The **Spawn** function must be overridden individually for each type of foliage.

For example, this is how it will look for interactive ferns.

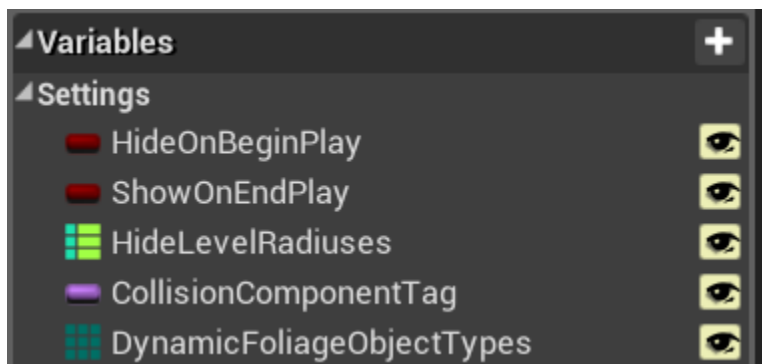


The component also provides functions to hide or show instances. The **CanBeHidden** and the **HideLevel** variables determine hide settings for the type of foliage.

3.4.4. BP_FoliageHideComponent

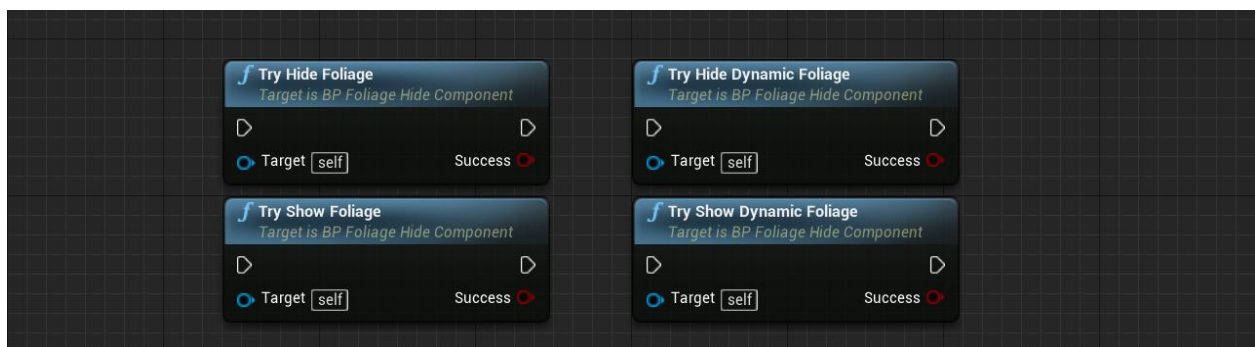
The **BP_FoliageHideComponent** contains functionality that allows to hide or show foliage instances and dynamic foliage actors near the owner. It allows you to configure hide radius for foliage with different hide levels.

For example you can add the component to the foundation building object and when you place it on the ground it hides large grass and ferns under it.



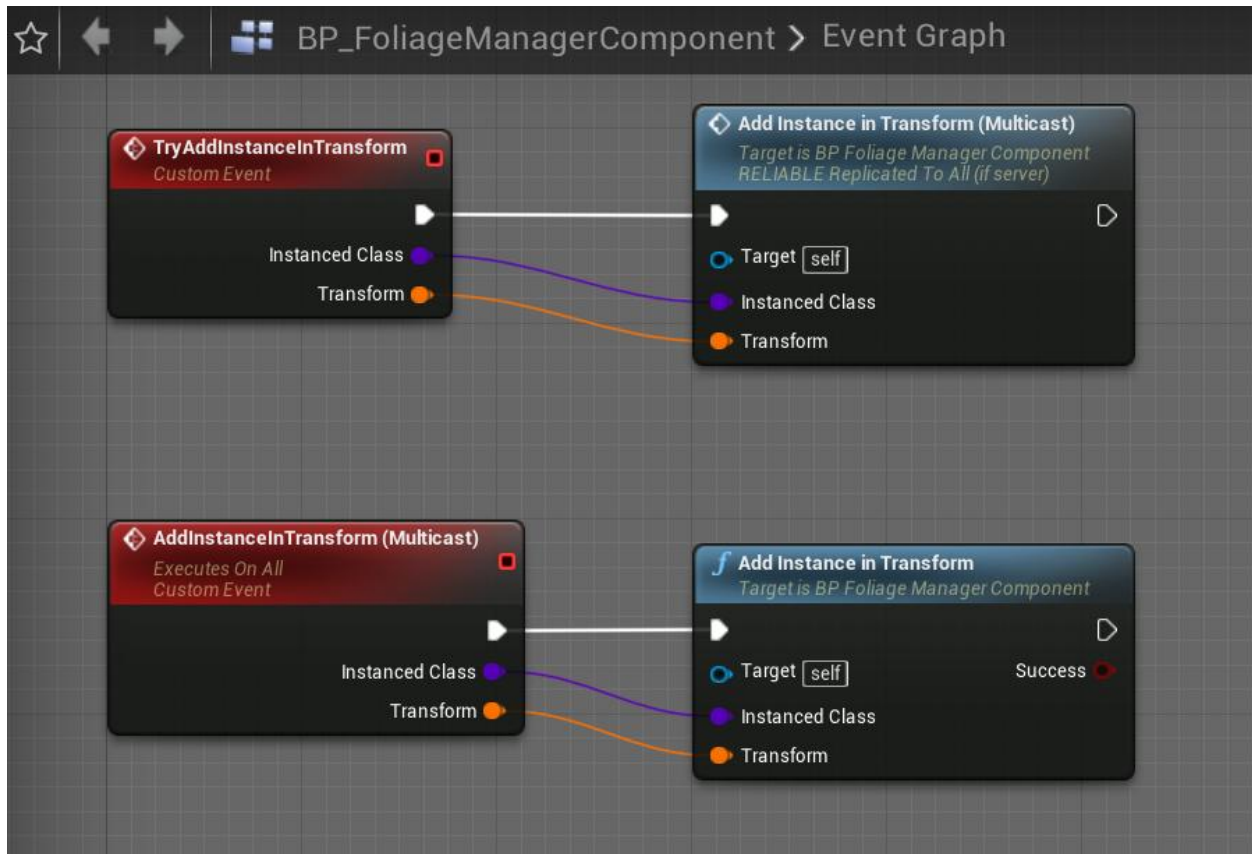
Add this component to the actor that should hide foliage and configure the **HideLevelRadiuses** variable. It will hide or show foliage in a specific radius for specific foliage level. To use specific collision component bounds use the **CollisionComponentTag** variable. Add the same value to tags of the component that should be used.

By default it will hide foliage on the begin play event and show back on the end play event automatically. To hide or show foliage manually use the functions on the screenshot below.



3.4.5. BP_FoliageManagerComponent

The **BP_FoliageManagerComponent** provides functions for quick access to the instanced foliage actor and for adding and removing instances in multiplayer.

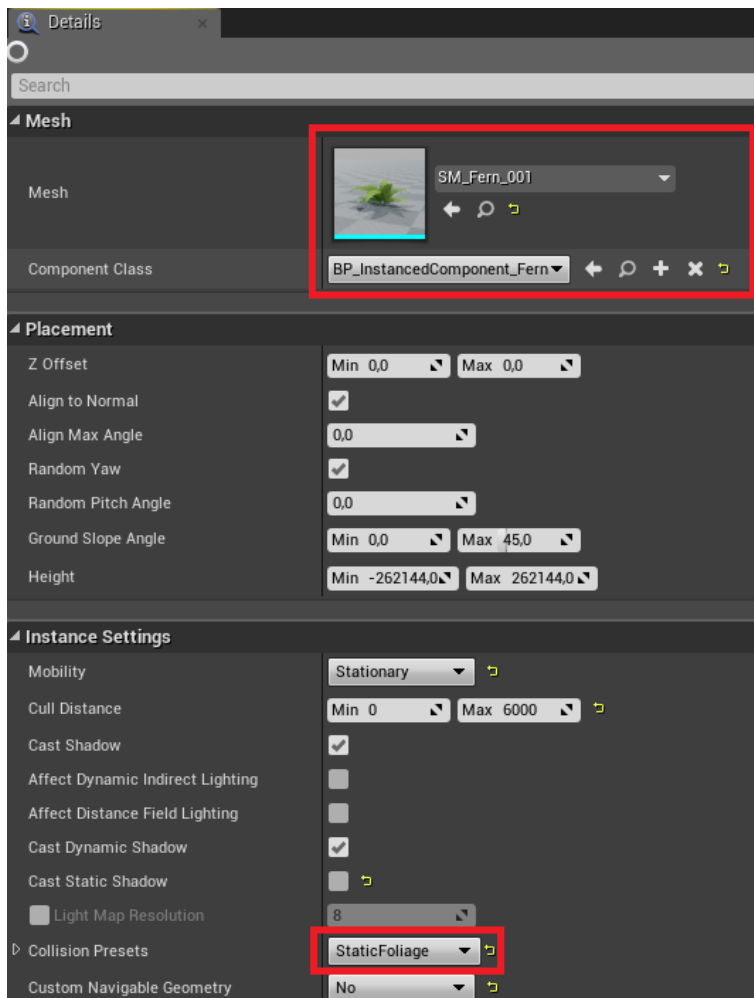


This component is replicated and should be added to the **GameState** class.

3.4.6. Configuring Replaceable Instances

To add a new type of replaced instance, you must:

1. Create an actor that will replace the instance.
2. Create a child class from **BP_InstancedComponent_Base**. In this class, you need to override the **Spawn** function. Set the **SpawnOnServerOnly** variable to true if the replaced actor is replicated. Set the **SpawnOnDedicatedServer** variable to false if the replaced actor is not replicated.
3. Create **Foliage Type** customize it using the created component, select the collision type **StaticFoliage**.



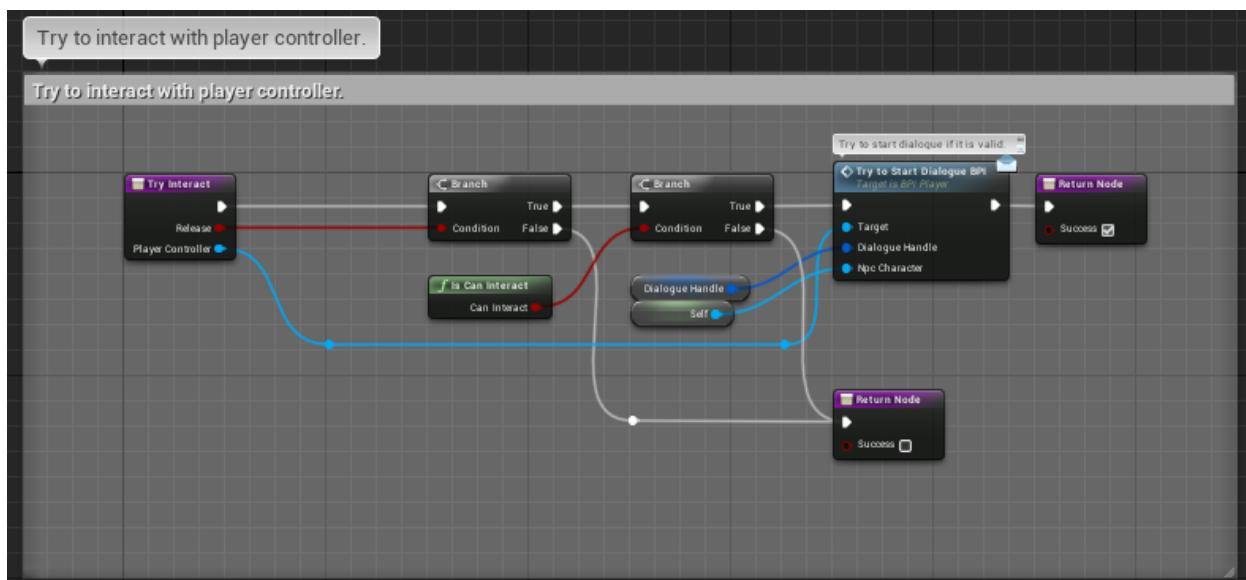
3.5. Dialogue system

3.5.1. Description

The dialogue system allows the player to communicate with the NPC using the replicas, receive the necessary information, advance through the plot, trade, etc. Events triggered when playing replicas, as well as the conditions for the appearance of these replicas, are written in the class inherited from **BP_Dialogue_Base** individually for each character with which you can interact.

To store data about dialogs and their replicas, **STR_Dialogue** and **STR_DialogueReply** structures have been created, respectively. For dialogs, the **DT_Dialogues** data table is used, and for dialog replicas, the **DT_DialogueReplies** data table is used.

For a NPC with whom you can talk, you need to implement the **BPI_InteractionObject** interface and then you can interact with him to start a dialogue. It is also necessary to implement the functions of starting, ending and playing dialog lines by a character from the **BPI_Character** interface. They are already implemented for the base character.



You can start a dialogue with a **NPC** if he can interact and the dialogue ID exists in the **DT_Dialogues** table.

Dialog control functions are located in the **BPI_Player** interface and are implemented in the **BP_PlayerController** class. The functionality is located in the **BP_PlayerManagerComponent** component.

The visual interface of dialogs and individual replicas is in **UI_Dialogue** and **UI_DialogueReply**.

3.5.2. STR_Dialogue

The **STR_Dialogue** structure is used to store data about the dialogue with a specific character. Based on it, the **DT_Dialogues** data table was created, which stores data about dialogues with all characters.

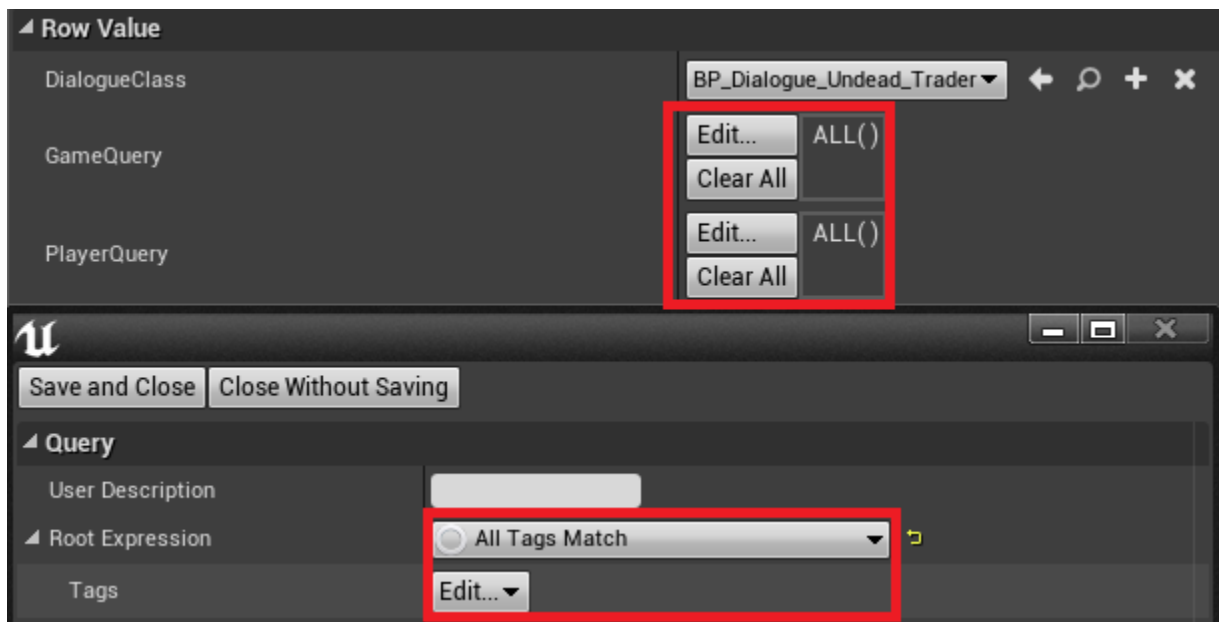


DialogueClass - the class of the dialog actor that will be created when the dialog starts.

GameQuery - requirement for game tags to start a dialogue.

PlayerQuery - requirement for player tags to start a dialogue.

FirstReplies - identifiers for selecting the first replica from the **DT_DialogueReplies** table from which the dialoG can start.

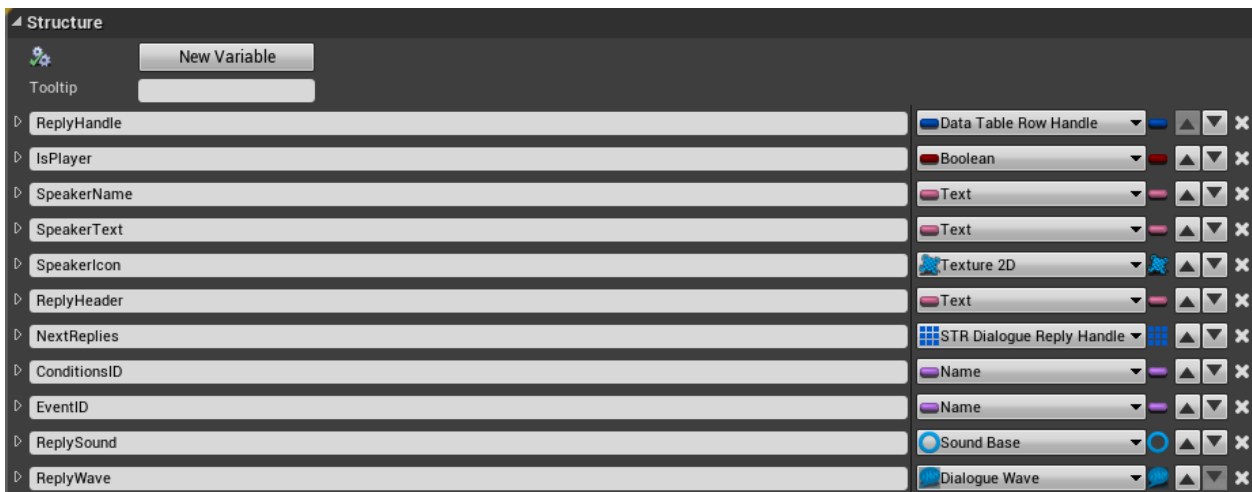


If there are no specific requirements for starting a dialogue with a character, then

GameQuery and **PlayerQuery** must be set as in the picture above, otherwise the dialogue will not start.

3.5.3. STR_DialogueReply

The **STR_DialogueReply** structure is used to store data about character replies. Based on it, the **DT_DialogueReplies** data table was created, which stores the data of all character replies.



ReplyHandle - replies identifier in the **DT_DialogueReplies** table.

IsPlayer - player selectable reply.

SpeakerName - override the character name of the reply.

SpeakerText - override the character icon of the reply.

SpeakerIcon - override the character icon of the reply.

ReplyHeader - override the replica text in select mode.

NextReplies - the next possible replies.

ConditionsID - condition id for the appearance of the reply.

EventID - id of events occurring during reply plays.

ReplySound - sound played during a reply.

ReplyWave - localized sound played during a reply.

3.5.4. BP_Dialogue_Base

BP_Dialogue_Base - it is the base class for dialogs. It includes the functionality of managing the dialogue process, namely, the choice of replies and their sequence. Provides a choice of replies based on conditions, and triggers events when replaying specific replies.

To create unique dialogs with your own conditions and events, you need to create a new class based on **BP_Dialogue_Base**. This class can then be selected for the structure in the **DT_Dialogues** table.

Conditions for the appearance of replies in the dialogue are checked by the **CheckConditions** function depending on the **ConditionsID** in the reply structure. This function can be overridden and unique conditions can be written in a new class for dialog. Read more about this in the **Reply Conditions** section.

Events when playing replies in a dialog are called by the **DoDialogueEvent** function depending on the **EventID** in the reply structure. This function can be overridden and unique events can be written in a new class for the dialog.

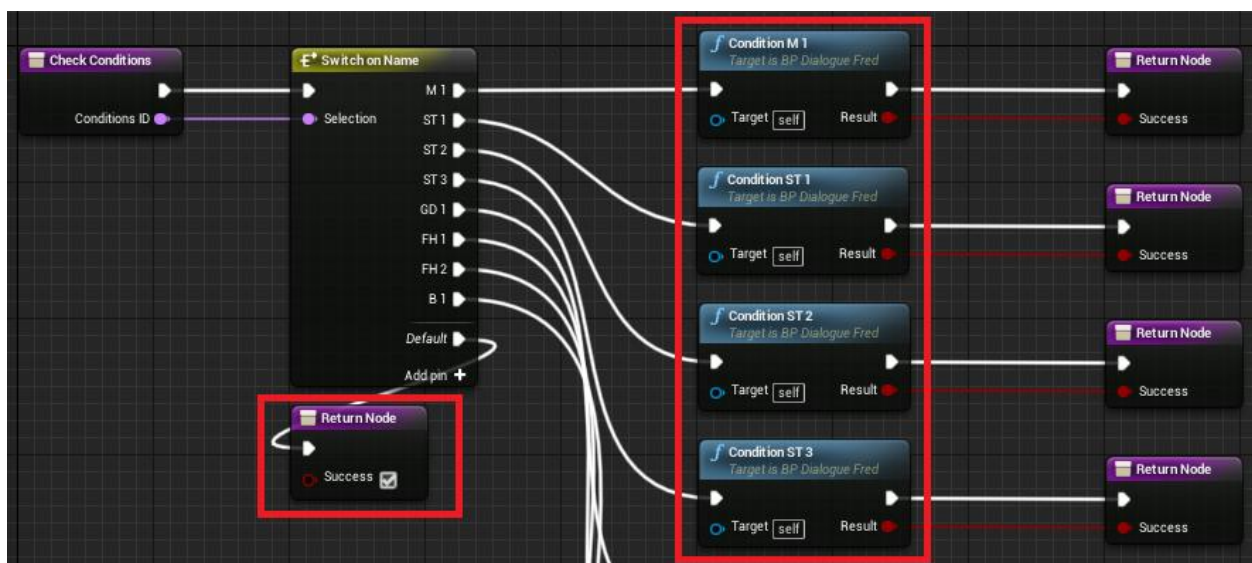
Read more about this in the **Reply Events** section.

3.5.5. Reply conditions

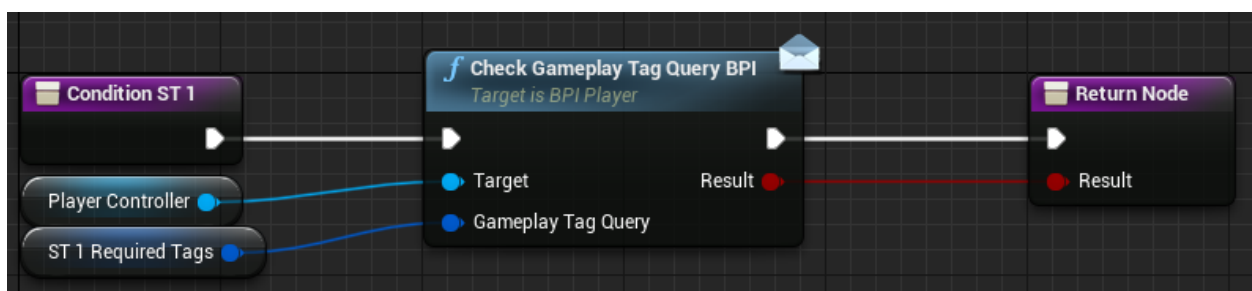
The conditions required for the appearance of certain replies must be written individually for dialogue with each NPC. Verification of reply conditions is performed using the **ConditionsID** from the **STR_DialogueReplies** structure.



Checking conditions must be performed in the overloaded **CheckConditions** function of the character dialog class. If no match is found or no identifier is found, the function should return **true**.



In the conditions, you can check the presence of certain tags or resources of the player, check the level of the game character, etc.

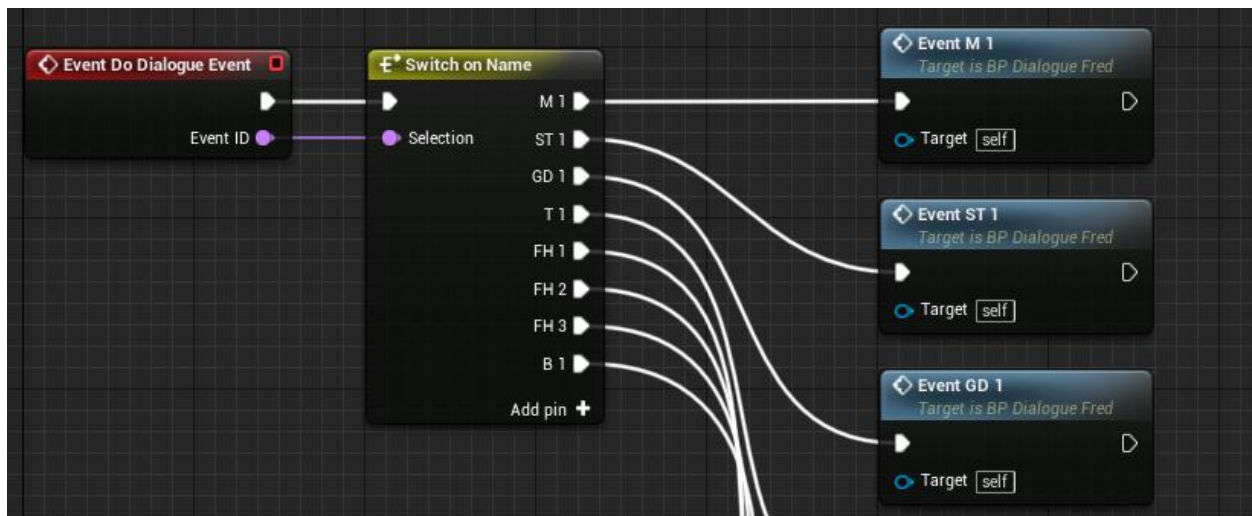


3.5.6. Reply events

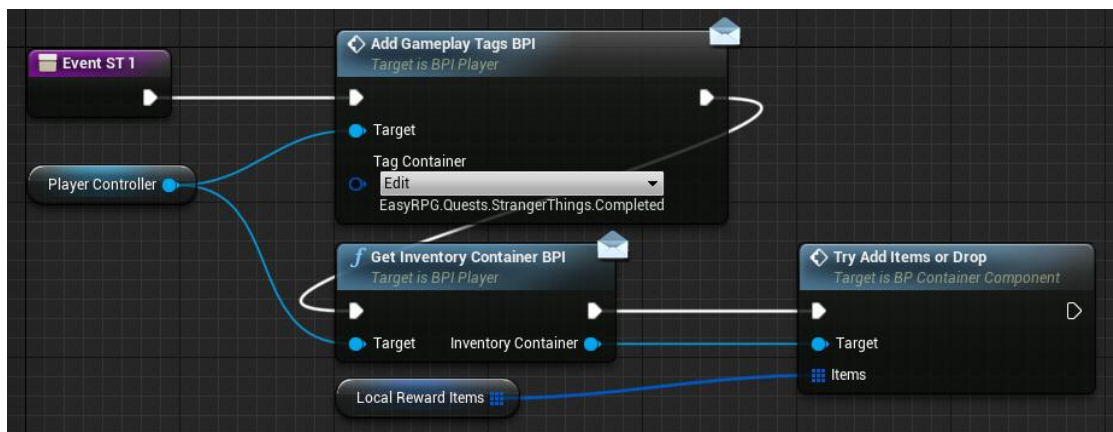
Replica events occur when a replica is played back. They are called using the **DoDialogueEvent** function by the identifier from the **STR_DialogueReply** structure.



These events must be registered individually in the dialog character class.



These events can be used to add tags or resources to the player, add new crafting blueprints or journal quests, and much more.



To simplify the design of dialogs, you can use a diagram. They clearly show the sequence of replicas in dialogues, the conditions for the appearance of certain replicas, as well as the events that are triggered when they are played.

3.6. Quest system

3.6.1. Description

The quest system allows you to set certain tasks for the player and track the progress. It is also possible to add different quests to the journal and display active tasks on the main screen.

Base class for quests: **BP_Quest_Base**. All other quests must be created on its basis. The assignments can include several stages. To complete each stage, you must complete one or more tasks.

Base class for tasks: **BP_QuestTaskComponent_Base**. On the basis of this class, basic tasks have been created that can be used in quests. Other tasks also need to be created based on this class.

Functions for adding quests and journal notes are located in the **BPI_Player** interface and implemented in the **BP_PlayerController** class. The functionality is located in the **BP_PlayerManagerComponent** component. The functionality for managing active quests is also there.

The **STR_JournalNote** structure has been created to store data about notes in the journal. All journal notes are stored in the **DT_JournalNotes** table.

The visual interface of the active quests window is in **UI_ActiveQuests**.

The visual interface of the journal is in **UI_Journal**, and for notes in the journal in **UI_JournalNote**.

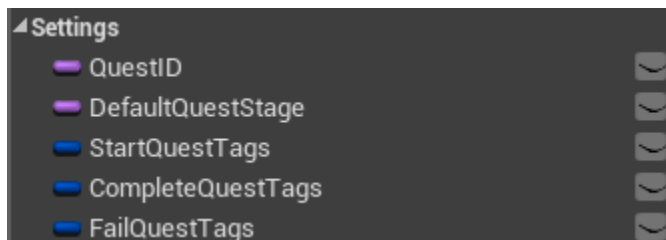
arbutas#2127

3.6.2. BP_Quest_Base

BP_Quest_Base - this is the base quest class. It includes the functionality of managing the quest execution process, namely, its initialization, checking the progress of tasks and the execution of task stages. Asvignments can include an unlimited number of stages.

To create a new quest, you need to create a new class based on **BP_Quest_Base**. Each stage of a quest can include several active tasks.

To set up a quest, you need to set up variables:



QuestID - quest id.

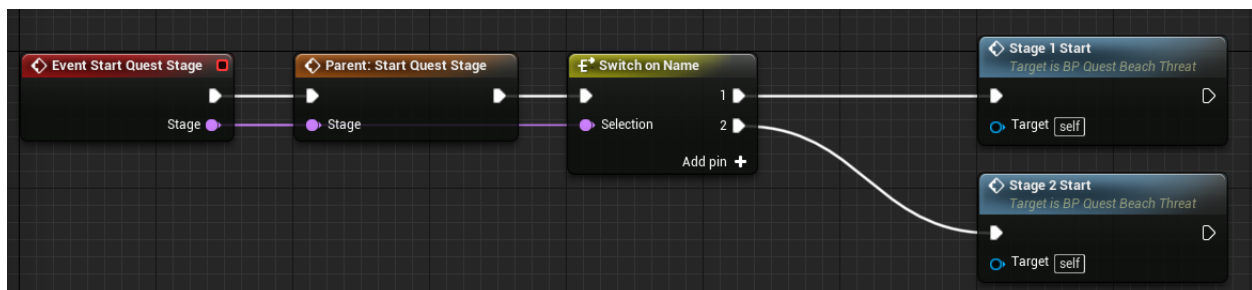
DefaultQuestStage - name of the initial stage of the quest.

StartQuestTags - tags added to the player at the start of the quest.

CompleteQuestTags - tags added to the player when complettnng a quest.

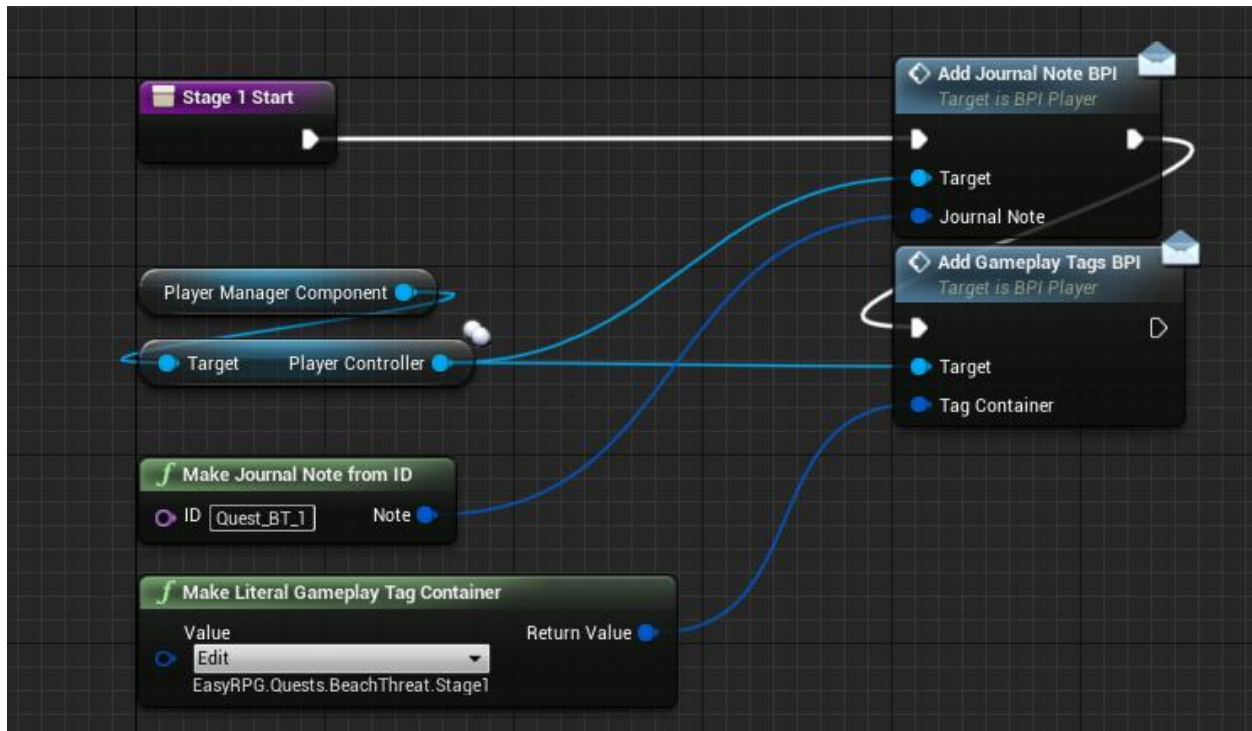
FailQuestTags - tags added to the player when the task fails.

To customize the quest, three functions must also be modified: **StartQuestStage**, **InitQuestStage** and **CompleteQuestStage**. The **StartQuestStage** function takes the quest to the specified stage and calls the **InitQuestStage** function, where the active tasks for this stage are initialized. When the quest starts, automatically moves the quest to the stage specified in the **DefaultQuestStage** variable.

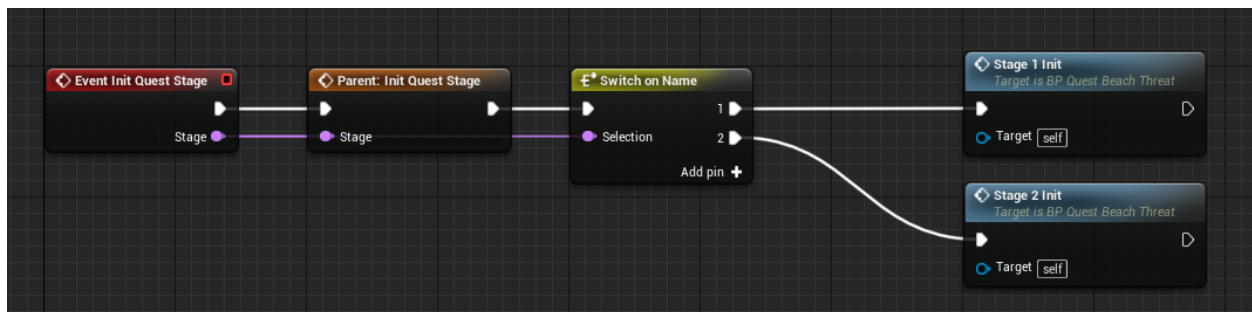


The **CompleteQuestStage** function is called automatically when all active tasks are completed or when one of the main tasks is completed.

By modifying the **StartQuestStage** function, you can, for example, add notes to the journal, add game tags to the player, give out items, etc, at the beginning of the quest stage. The same can be done at the time of the stage execution in the **CompleteQuestStage** function.



Tasks are initialized in the **InitQuestStage** function. Tasks can be added and configured in the quest components window.



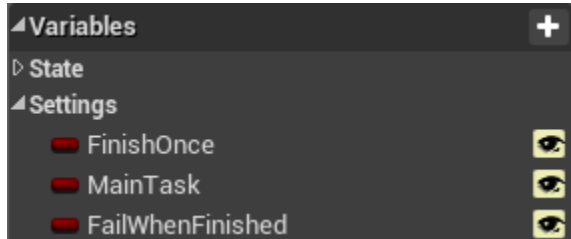
To start a quest in the game, you need to call the **StartQuest_BPI** function from the **BPI_Player** interface for the target player.



The quest will be created in the game and added to the active quests.

3.6.3. BP_QuestTaskComponent_Base

BP_QuestTaskComponent_Base - base class for quest tasks.



FinishOnce - the task is completed only once.

MainTask - the task is the main one. The quest stage ends immediately when such a task is completed..

FailWhenFinished - upon completion of the task, the quest will fail.

Based on **BP_QuestTaskComponent_Base**, the main tasks are created and configured that can be used in quests.

Main tasks

BP_QuestTaskComponent_PlayerTags - task that completes if the player has or receives the required game tags.

BP_QuestTaskComponent_GameTags - a task that completes if the game has or receives the required game tags.

BP_QuestTaskComponent_ItemRequired - a task that completes if the required item is or appears in the player's containers.

BP_QuestTaskComponent_ConsumeItem - a task that completes if the player uses the required item a certain number of times.

BP_QuestTaskComponent_CutTree - a task that ends when the player chops down a tree a certain number of times.

BP_QuestTaskComponent_DestructMine - a task that ends when the player destroys an ore vein a certain number of times.

BP_QuestTaskComponent_KillCharacter - a task that ends when the player kills the NPC a certain number of times.

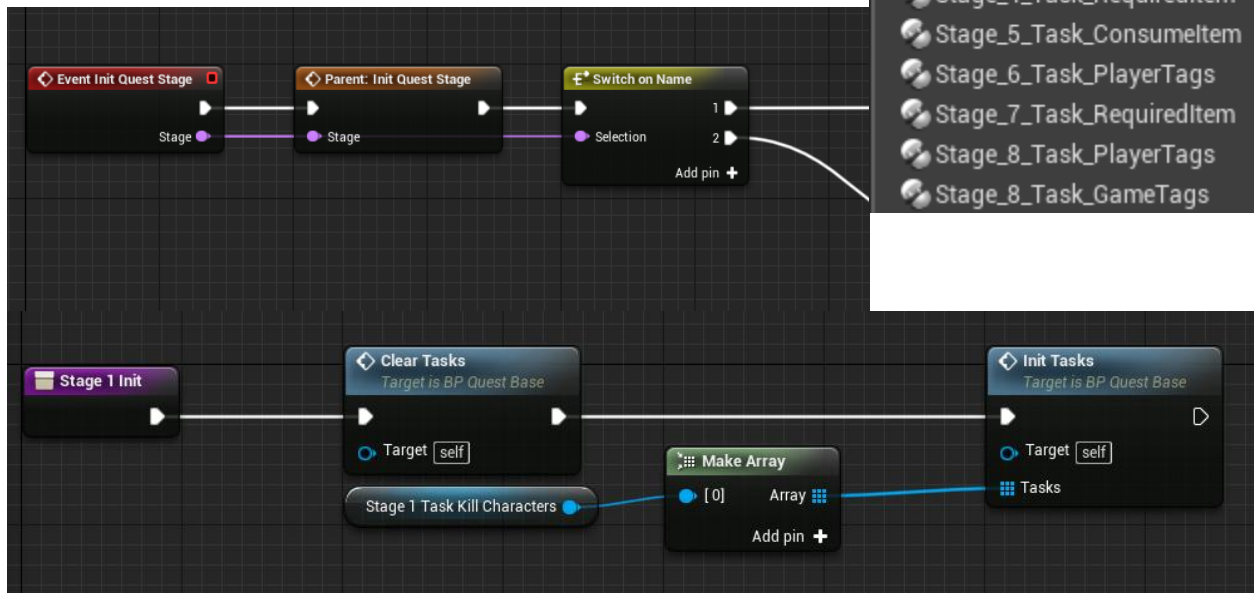
BP_QuestTaskComponent_Building - a task that ends when the player completes the building of a structure a certain number of times.

BP_QuestTaskComponent_Crafting - a task that completes when the player completes the creation of items according to the crafting blueprint a certain number of times.

BP_QuestTaskComponent_TimeRemaining - a task that completes after a specified time.

Other tasks for jobs must also be created based on **BP_QuestTaskComponent_Base**
BP_QuestTaskComponent_Base.

Tasks can be immediately added to the task blueprint components window and initialized in the **InitQuestStage** function. Before initializing tasks for the task stage, you must call the **ClearTask** function to clear the list of previous active tasks. Tasks can also be created and configured when they are initialized using the **AddComponent** nodes.

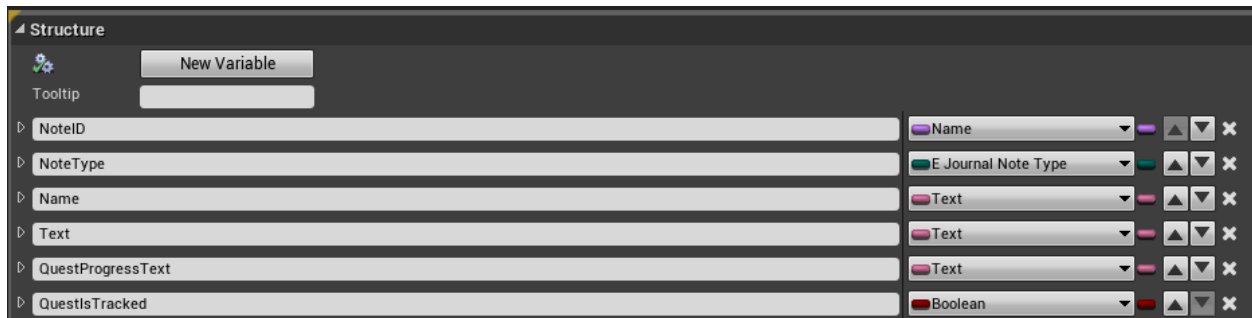


3.6.4. Journal notes

The **STR_JournalNote** structure has been created to store data about notes in the journal. All notes are stored in the **DT_JournalNotes** table.

Functions for adding quests and notes are located in the **BPI_Player** interface and are implemented in the **BP_PlayerController** class. The functionality is located in the **BP_PlayerManagerComponent** component. The functionality for managing active quests is also there.

STR_JournalNote



NoteID - journal note id.

NoteType - type of journal note.

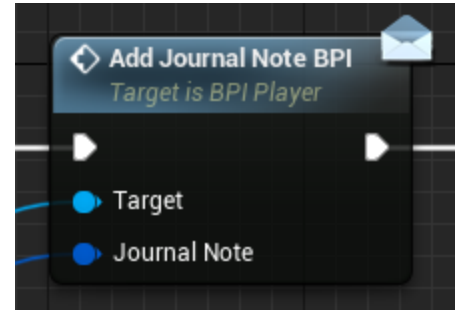
Name - name of journal note.

Text - text of journal note.

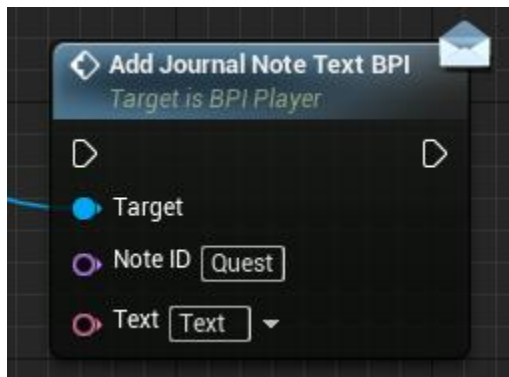
QuestProgressText - the current progress of the quest to be added in the journal.

QuestIsTracked - an indicator that note is being tracked in the active quests window.

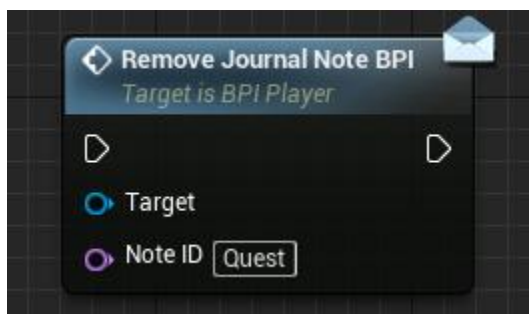
To add a journal note, call the **AddJournalNote_BPI** function from the **BPI_Player** interface for the player. The entry will be added to the log with the **NoteID**. If there is already an entry in the journal with the same **ID**, the text of the new entry will be added to the existing one.



To add text to an existing journal note, call the **AddJournalNoteText_BPI** function from the **BPI_Player** interface for add with a specific **ID** for the target player.



To delete a note with a specific id is necessary to call a function **RemoveJournalNote_BPI** from **BPI_Player** interface to target player.



To associate a journal note with an active quest, the **NoteID** value from the **STR_JournalNote** structure must match the **QuestID** from the class of the desired quest.

3.7. Save & Load System

3.7.1. Description

The save and load system is a very complex system, the functionality of which extends to many classes and components.

To save and load game settings and information about game sessions, the **BP_GameSave_Settings** class is used.

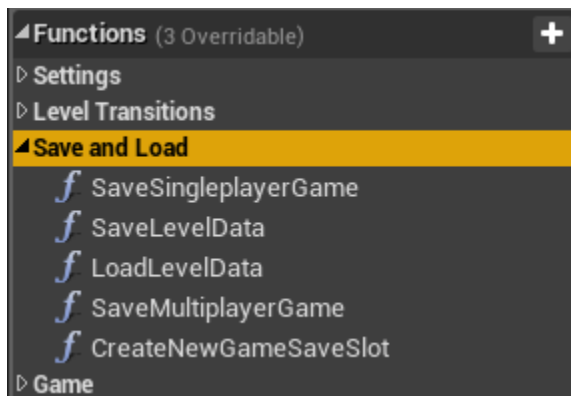
To save and load data about the game session, use the **BP_GameSave_Session** class.

The structure is used to store information about the player - **STR_SaveData_Player**.

The structure is used to store information about the game level - **STR_SaveData_Level**.

For an actor or component to be saved and loaded, it must implement the **BPI_SaveData** interface. Each type of object requires its own structure to save.

The functionality of saving and loading the game is located in **BP_GameInstance** in the functions of the **Save** and **Load** category.



3.7.2. STR_SaveData_Player

The **STR_SaveData_Player** is a structure that stores information about the player to save and load.



PlayerName - player name.

HairTypeIndex - character hair type index.

HairColorIndex - character hair color index.

PantsColorIndex - character pants color index.

SkinColorIndex - character skin color index.

Level - current player level.

Experience - current exp level.

AvailableSkillPoints - available ability points.

Attributes - player attributes.

Health - character health value.

Energy - character energy value.

Hunger - character hunger value.

Thirst - character thirst value.

InventoryItems - items in the character's inventory.

EquipmentItems - character equipment.

HotbarItems - items in the character's hotbar.

JournalNotes - player journal notes.

PlayerTags - player tags.

AdditionalBlueprints - additional player blueprints.

IsAlive - indicator that the player's character is alive.

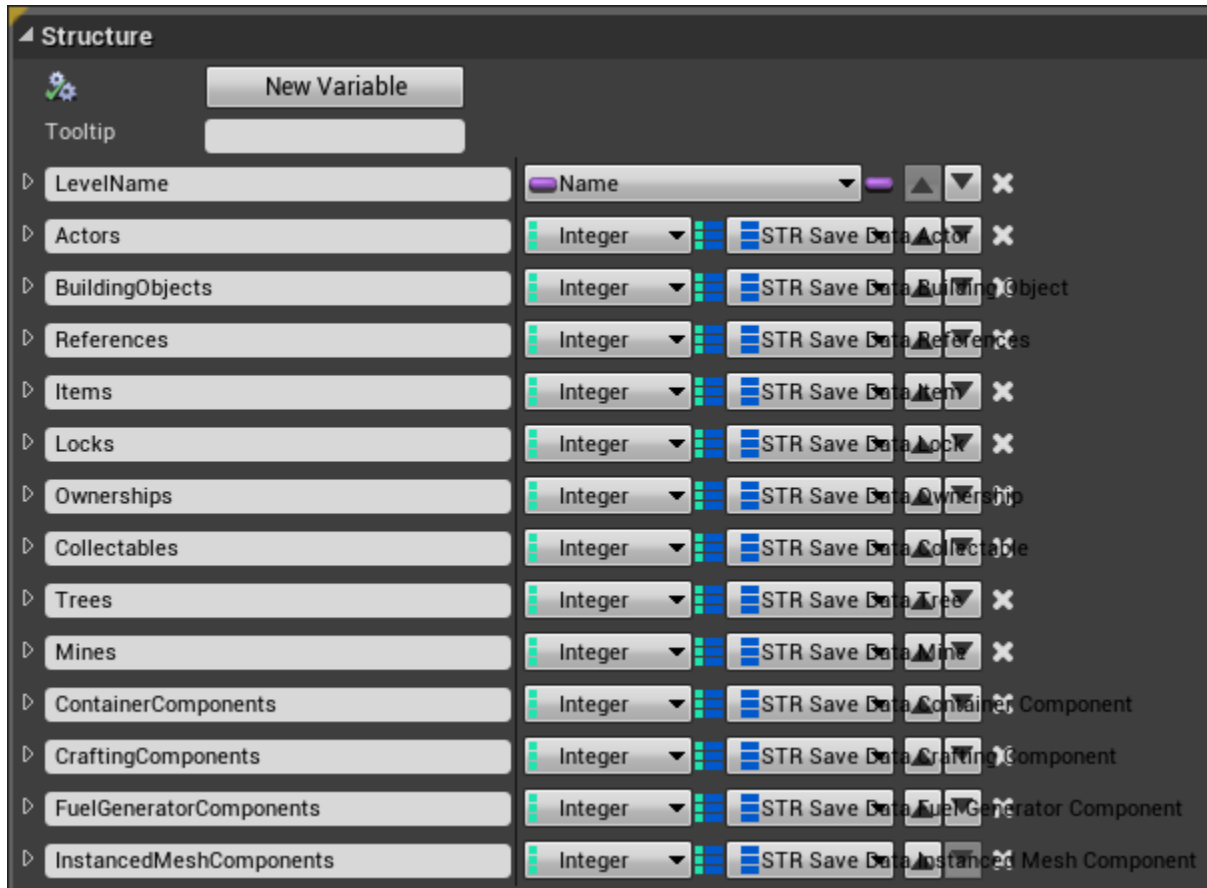
Transform - player character position.

Resources - player resources.

ActiveQuests - active player quests.

3.7.3. STR_SaveData_Level

STR_SaveData_Level - structure that is used to store data about objects at the level.



LevelName - level name.

Actors - main data of actors at the level.

BuildingObjects - building object data at the level.

References - data on the relationships of objects at the level.

Items - items data at the level.

Locks - level lock data.

Ownerships - claim level data (example - totems).

Collectables - data on collected objects at the level.

Trees - tree data at the level.

Mines - ore vein data at the level.

ContainerComponents - container component data at the level.

CraftingComponents - crafting component data at the level.

FuelGeneratorComponents - fuel generator component data at the level.

InstancedMeshComponents - instance component data at the level.

3.7.4. BP_GameSave_Settings

Description

Inherits from **SaveGame** class. This is a class used to save and load game settings and to save and load slots in saved games. Loaded when the game starts from the **BP_GameInstance** class.

Variables

HardwareBenchmarkUsed - an indicator that the optimal graphics settings have already been set.

AutosaveMode - current game autosave settings mode.

ShowFloatingText - indicator showing pop-up text during game.

ShowCharacterState - displaying the state of the game character on the user interface during the game.

ShowEnemyCharacterStates - displaying the status of other characters during the game.

ShowMinimap - displaying the minimap on the user interface during the game.

ShowMarks - displaying the mark during the game.

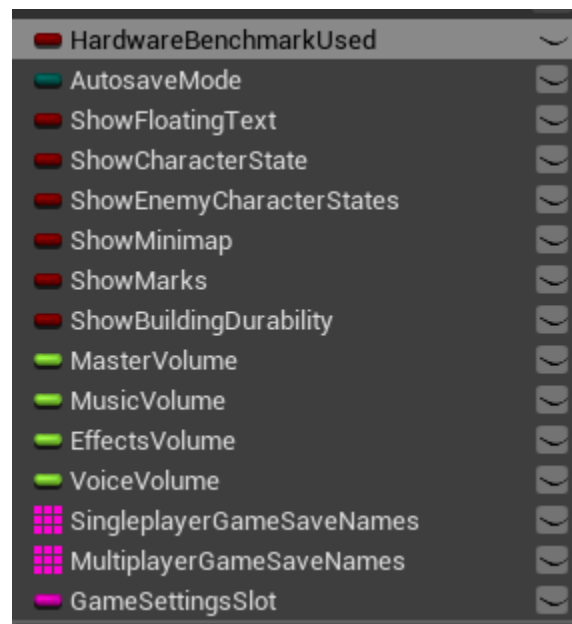
ShowBuildingDurability - displaying the status of buildings during the game.

MasterVolume - the value of the overall volume of sounds.

MusicVolume - music volume value.

EffectsVolume - effects volume value.

VoiceVolume - voice volume value.

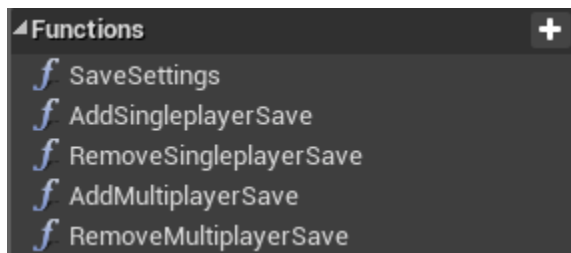


SingleplayerGameSaveNames - an array of slots in which the game is saved in the singleplayer mode.

MultiplayerGameSaveNames - an array of slots in which the game is saved in the multiplayer mode.

GameSettingsSlot - the slot in which the data of this class is stored.

Functions



SaveSettings - saving settings to a slot for saving game settings.

AddSingleplayerSave - adding a slot to the array of saved games in the singleplayer mode.

RemoveSingleplayerSave - removing a slot from an array of saved games in the singleplayer mode.

AddMultiplayerSave - add the slot to the array of saved games in the multiplayer mode.

RemoveMultiplayerSave - remove the slot from the array of saved games in the multiplayer mode.

3.7.5. BP_GameSave_Session

Description

Inherits from **SaveGame** class. Class with saved information about the game session. Used in **BP_GameInstance** to save the current game session, load and save data about the level and players.

Implements the **BPI_GameSave** interface and its functions. These functions are necessary for working with temporary data during the execution of the save and load functions. They are called by actors and components, which must be saved and loaded.

Variables

SlotName - the name of the slot where the save is saved.

CurrentLevel - current save level.

IsMultiplayer - multiplayer session indicator.

HostPlayerData - saved data about the player who is the host.

ClientPlayersData - saved player data.

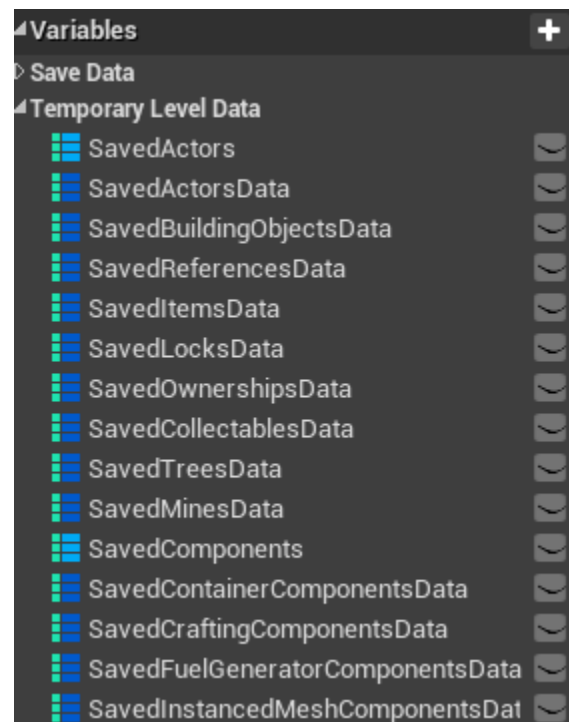
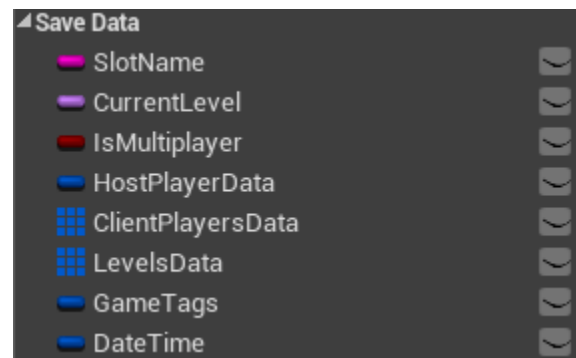
LevelsData - saved level data.

GameTags - saved game tags.

DateTime - save time and date.

Temporary Level Data - variables for temporary data to form the

STR_SaveData_Level structure. Shaped and used during the game save and load process.



Functions

InitGameSave - initializes basic save parameters.

GetClientPlayerData - getting save data for a specific player.

SaveData - write to save slot.

SaveCurrentLevelData - saves data about the current level from temporary variables.

GetLevelData - get save data of the specified level.

ClearTemporaryLevelData - removes temporary variables with level data.

LoadTemporaryLevelData - loads level data into temporary variables.

UpdateClientData - updates save data for the specified player.

UpdateClientsData - updates save data for all specified players.

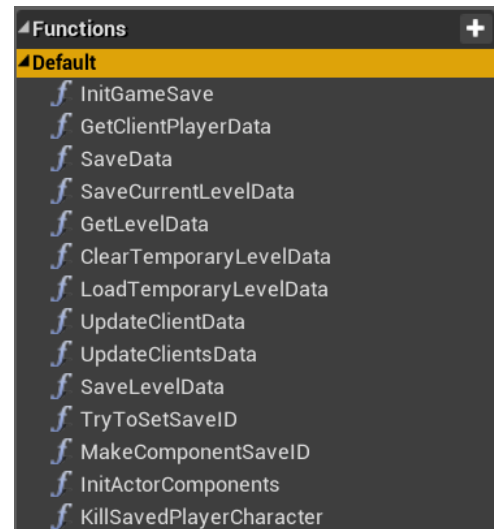
SaveLevelData - save or update data for target level.

TryToSetSaveID - try to set the save ID variable for target save object.

MakeComponentSaveID - returns component save ID.

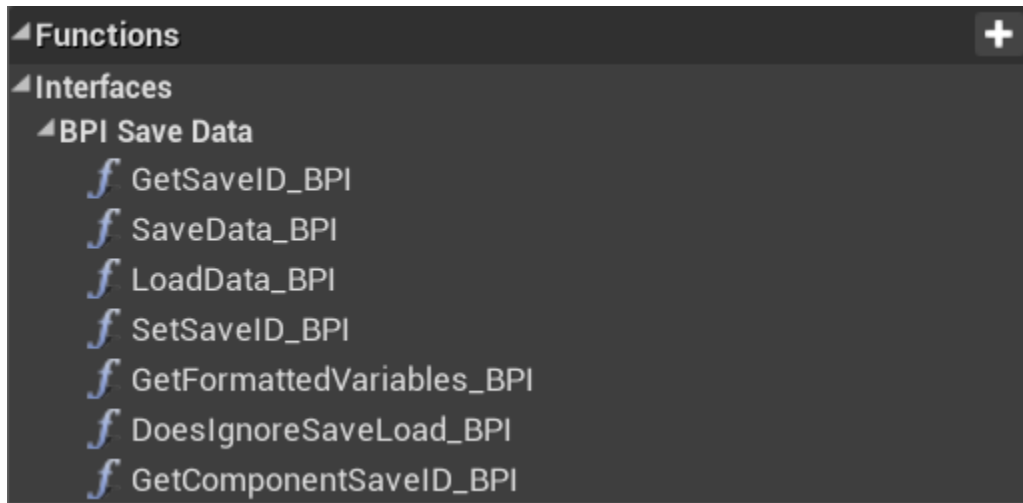
InitActorComponents - init actor components with the unique Save IDs for saving or loading process.

KillSavedPlayerCharacter - update save data for specific player and set alive status to false.



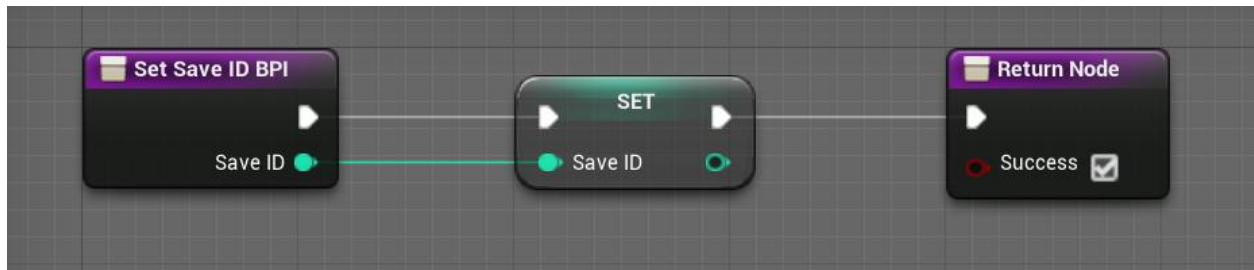
3.7.6. BPI_SaveData

BPI_SaveData - the interface required to store the data of an actor or component at the level.

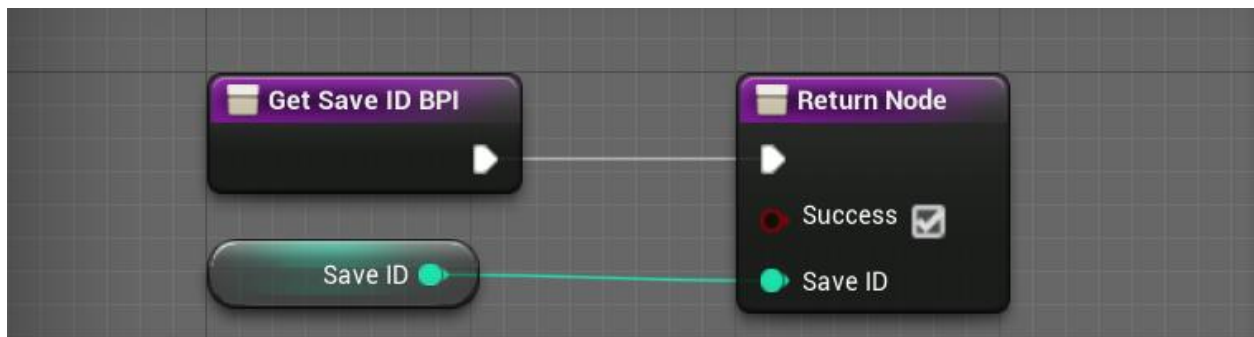


All interface functions must be implemented in an actor or component to save its data using the save and load system.

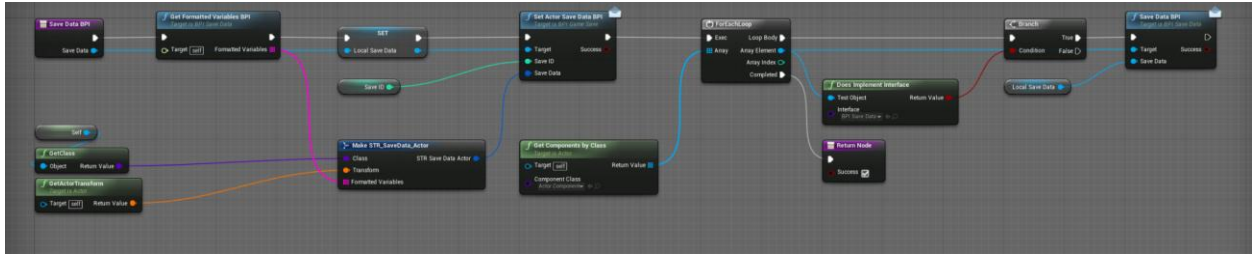
SetSaveID_BPI - sets the save identifier for the object.



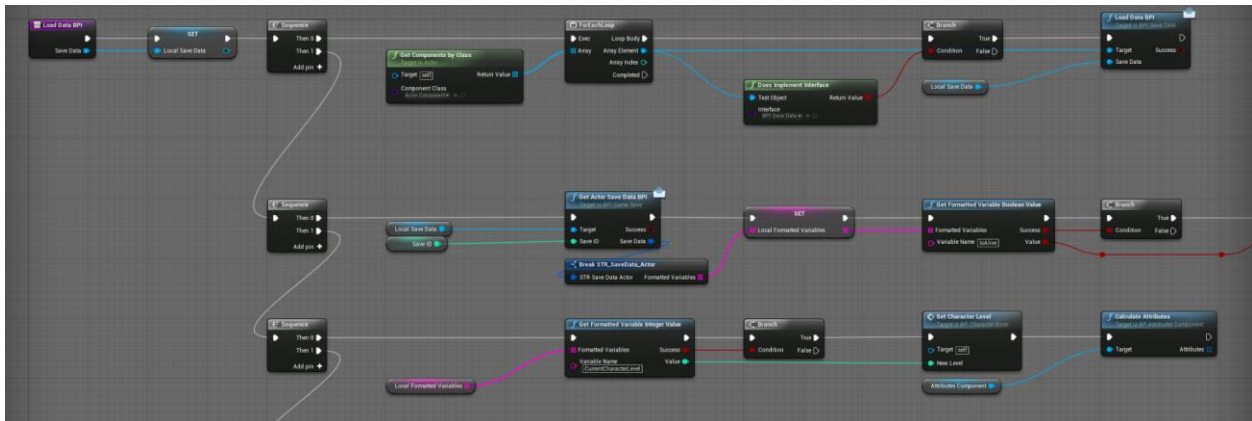
GetSaveID_BPI - gets the save id of the object.



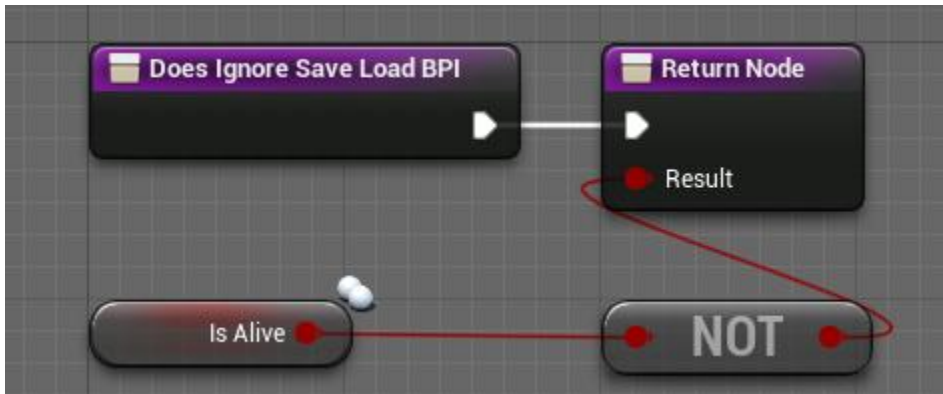
SaveData_BPI - saving object data. Various structures are used to store data. For example, the function of saving an item at the level uses the **STR_SaveData_Actor** and **STR_SaveData_Item** structures. The data from these structures is transferred to the session save instance using the **BPI_GameSave** interface, and stored in temporary level data. After saving all objects of the level, temporary data forms the structure of the **STR_SaveData_Level** level and is saved.



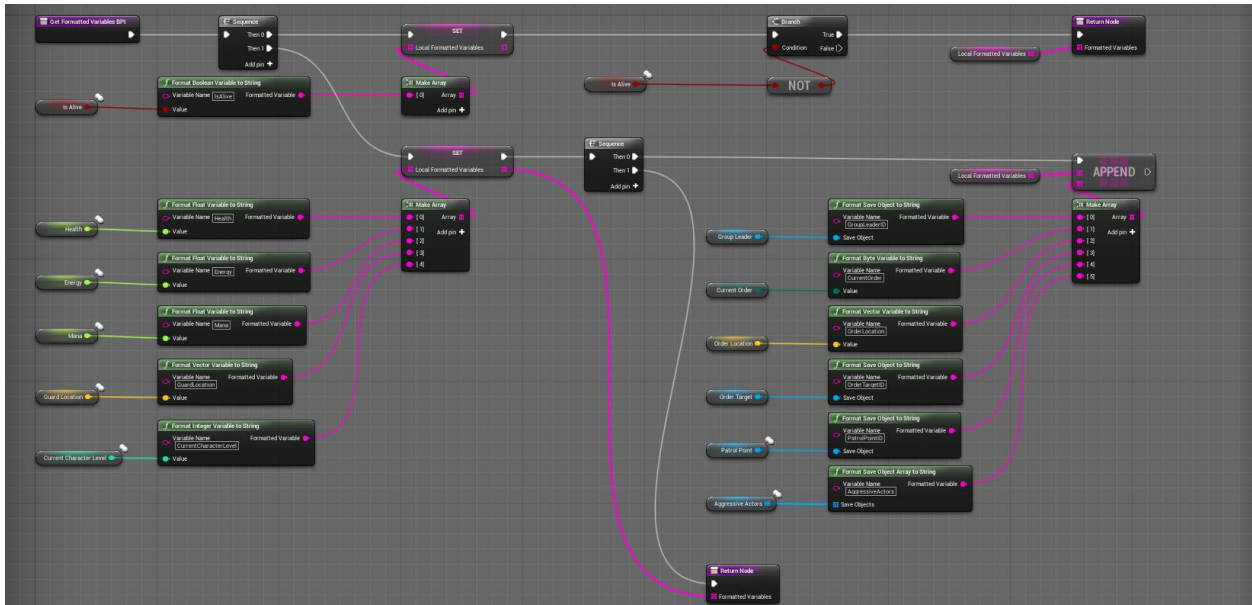
LoadData_BPI - loading object data and initializing loaded data.



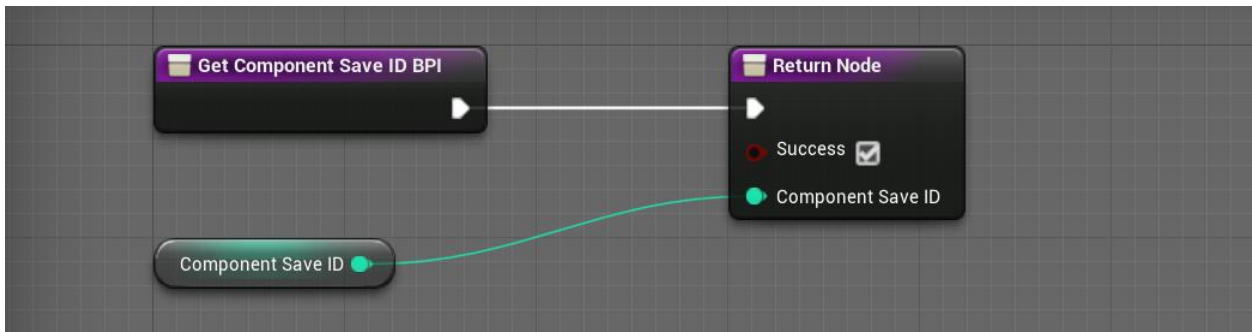
DoesIgnoreSaveLoad_BPI - determine that the object is used in the save / load system.



GetFormattedVariables_BPI - format simple variables that should be saved in the object. This function can be used for saving simple variables in save data without creating special structures.

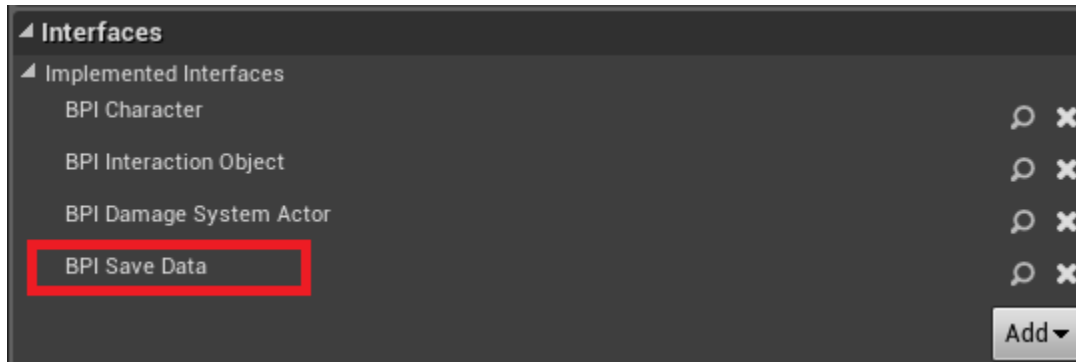


GetComponentSaveID_BPI - returns unique **ComponentSaveID**. This function should be implemented only in actor components that implement the **BPI_SaveData** interface.

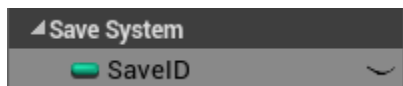


3.7.7. Saving & loading an object

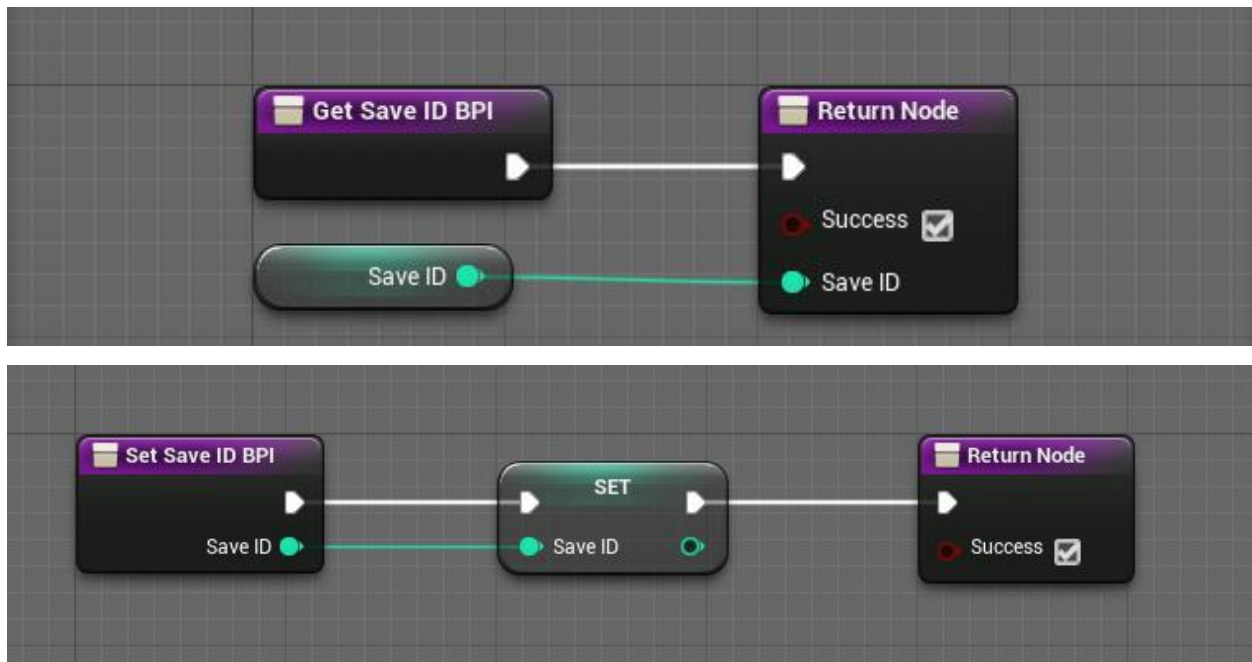
The save object must implement the **BPI_SaveData** interface.



This object also must have the **SaveID** integer variable.



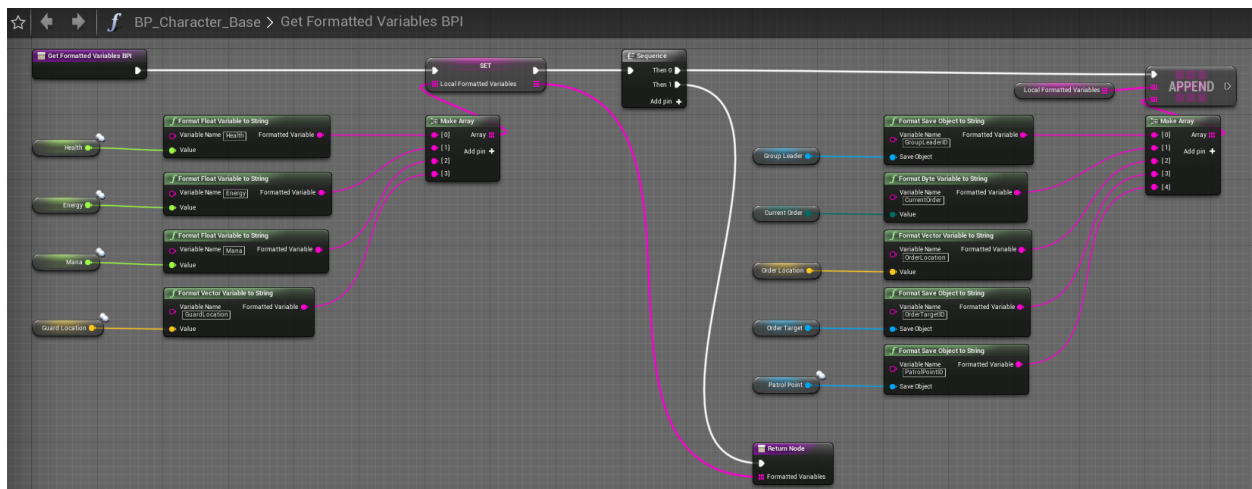
The **GetSaveID_BPI** and the **SetSaveID_BPI** functions are the same for all save objects.



Before saving the saving data should be prepared.

There are two methods for saving data of the object.

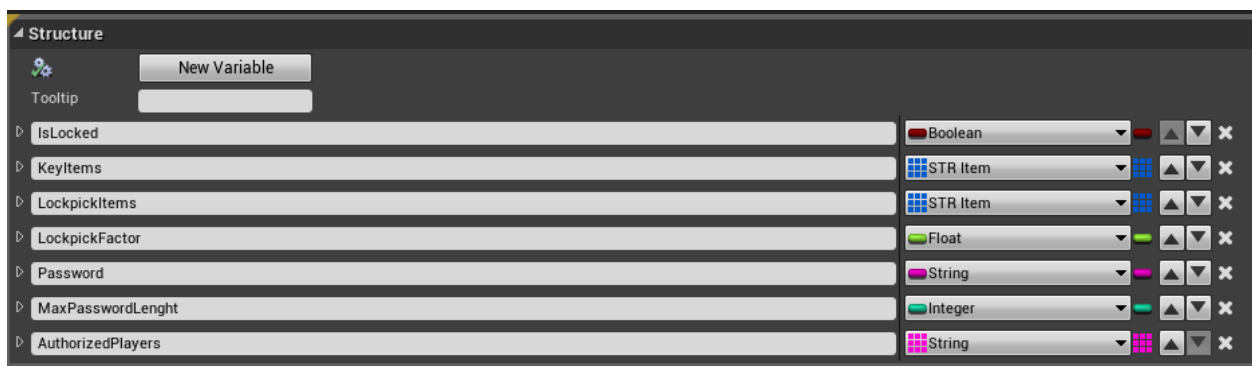
First method for simple variables (like boolean, integer, float, string and such arrays) can be configured in the **GetFormattedVariables_BPI** function of the saving object.



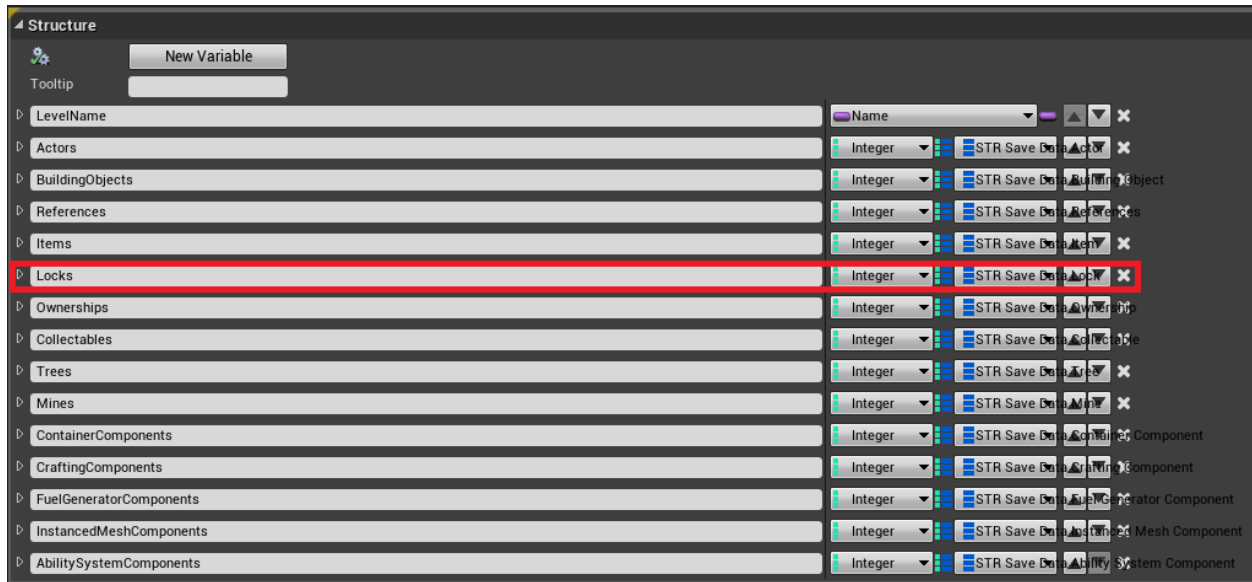
Second method for complex variables (like different structures) requires adding a special save structure to the level save structure.

NOTE: Always try to use the formatted method for simple variables, because adding new variables to **Game Save** blueprints corrupts the saves.

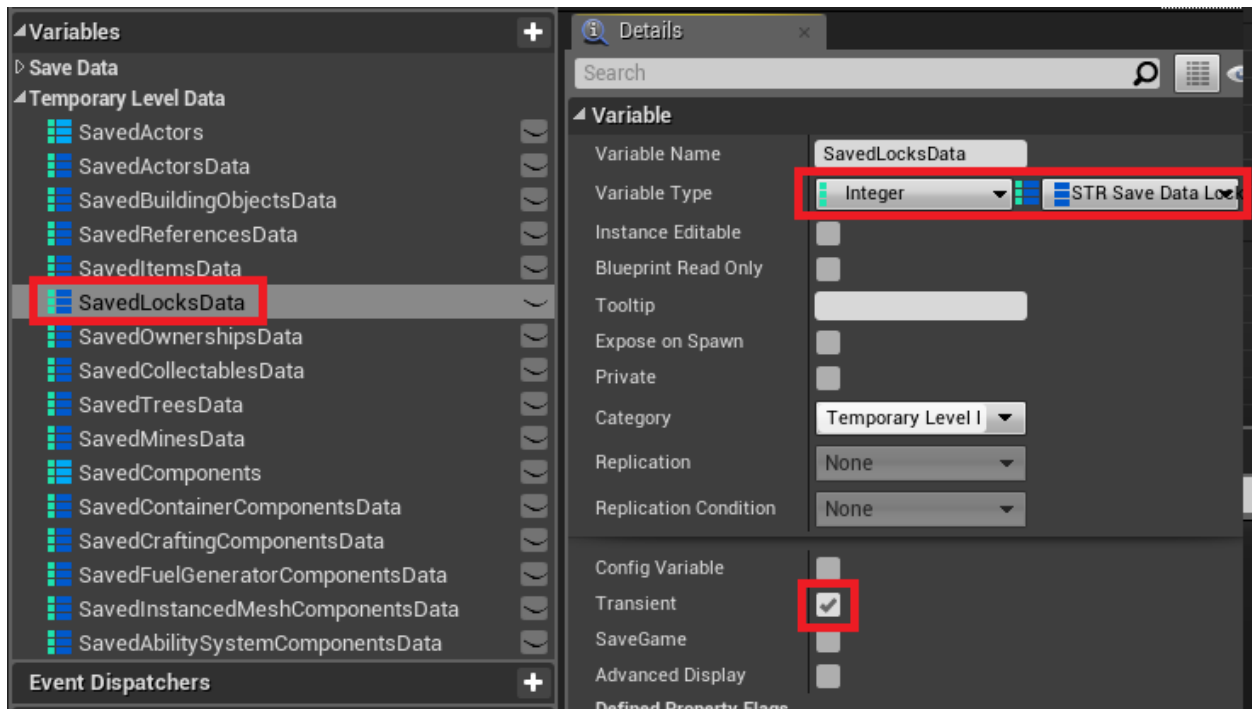
For example the **STR_SaveData_Lock** structure.



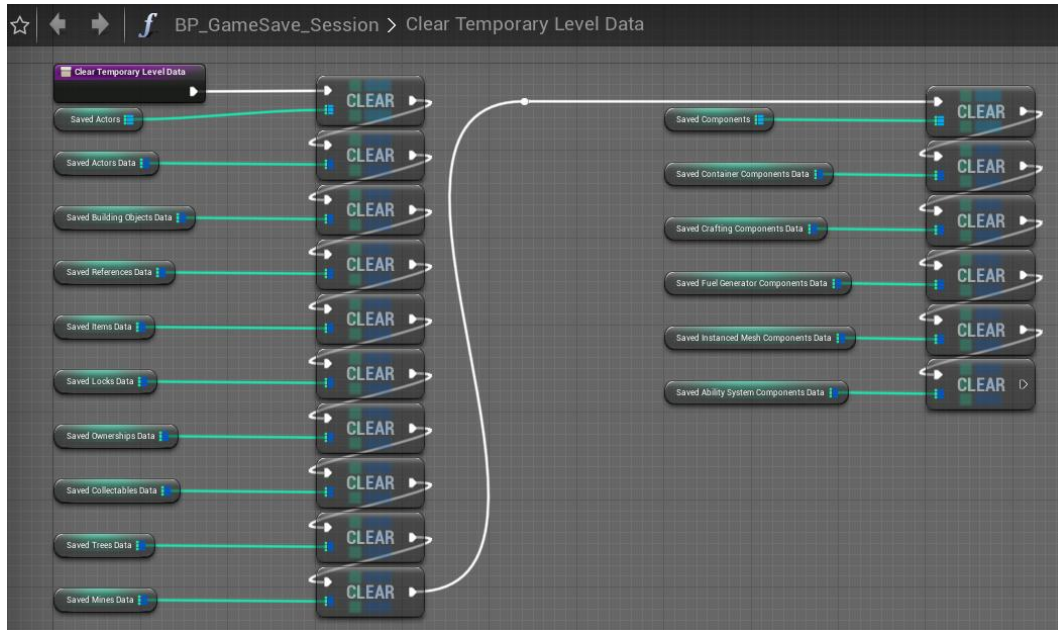
First, need to add the new structure to the level save structure. (**STR_SaveData_Level**)



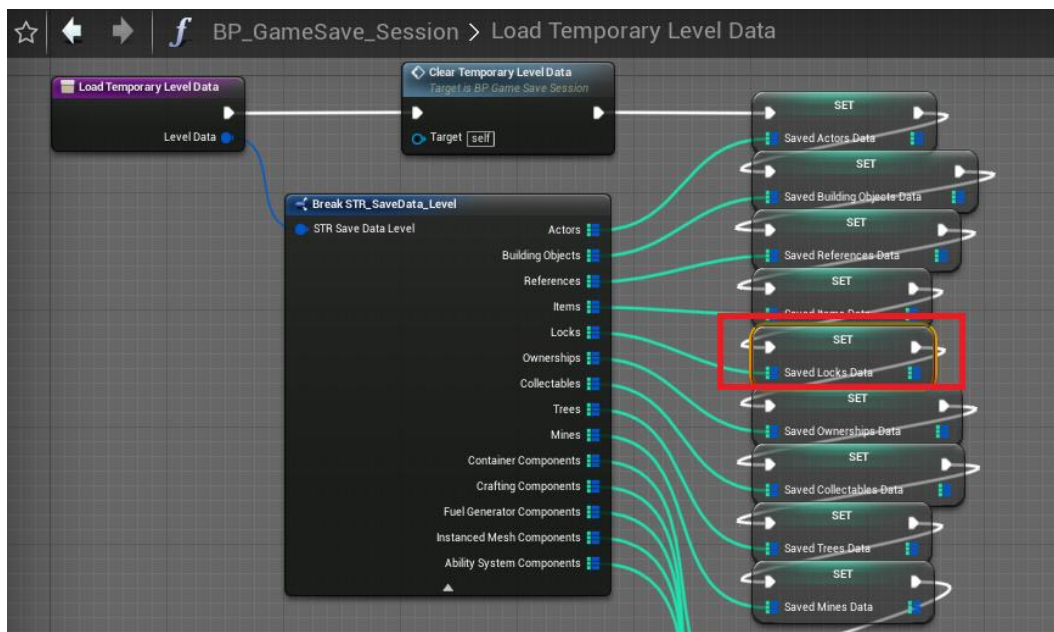
Then need to add an map array to the **BP_GameSave_Session** for the new save structure.



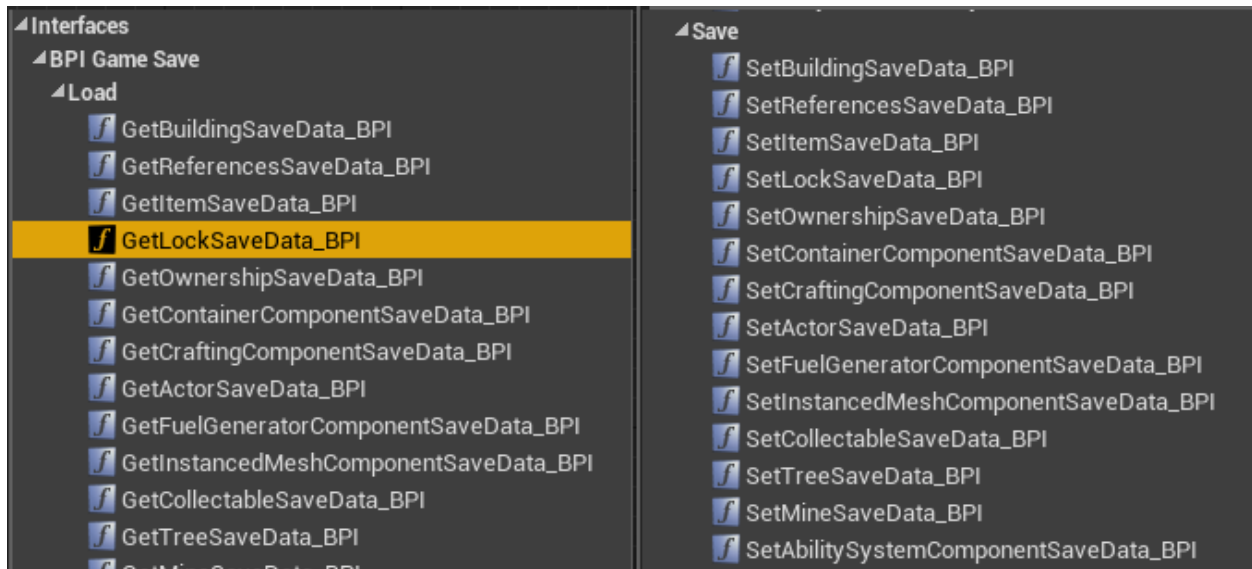
The temporary arrays should be cleared before level saving and level loading functions so need to clear the new array in the **ClearTemporaryLevelData** function of the **BP_GameSave_Session** blueprint.



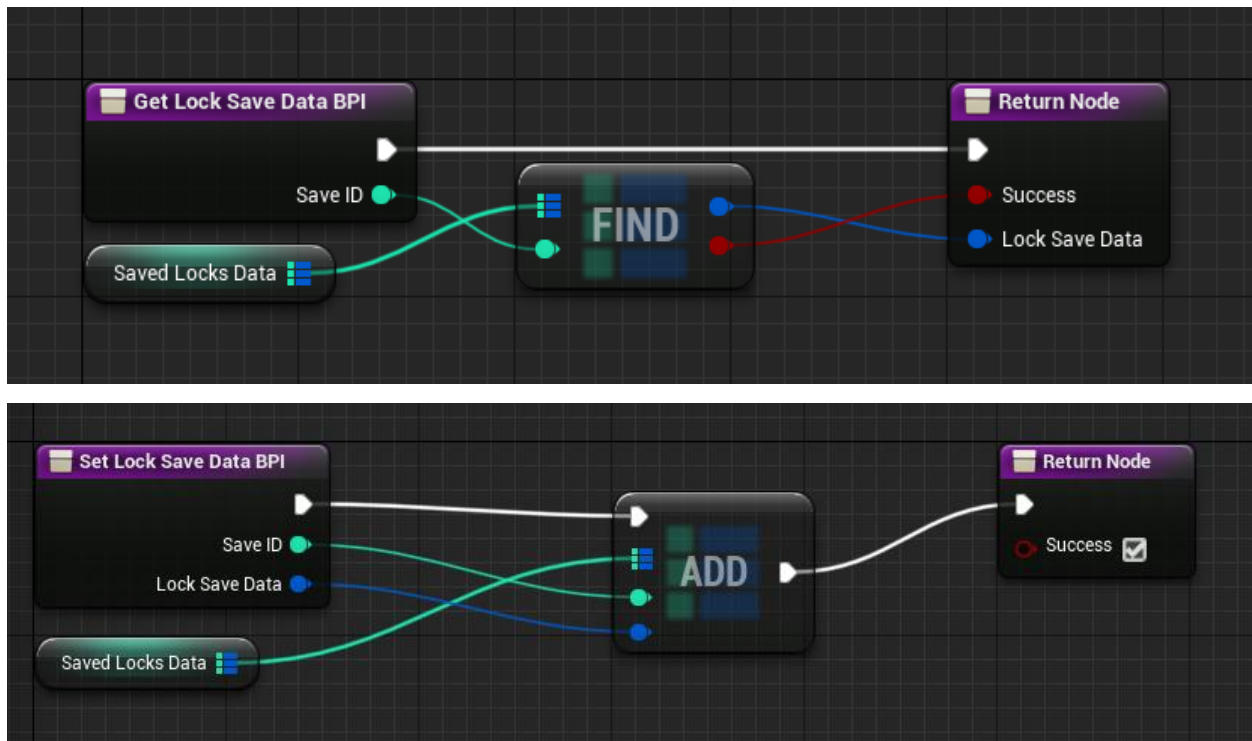
Also need to configure the **LoadTemporaryLevelData** in the **BP_GameSave_Session** blueprint.



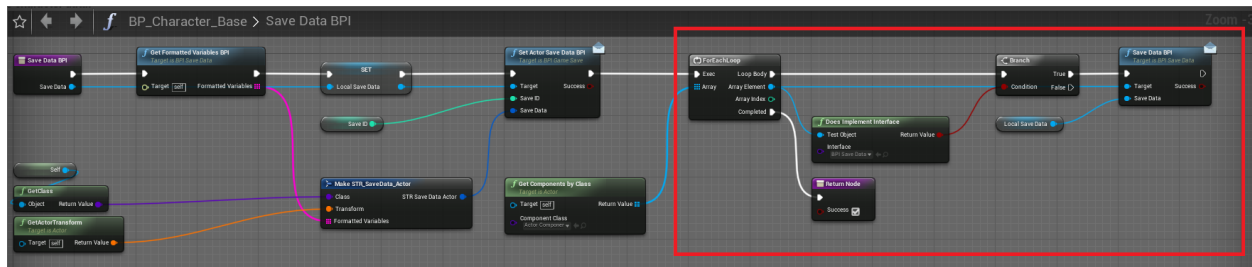
This method also requires special set and get functions in the **BPI_GameSave** interface for saving data of such structures.



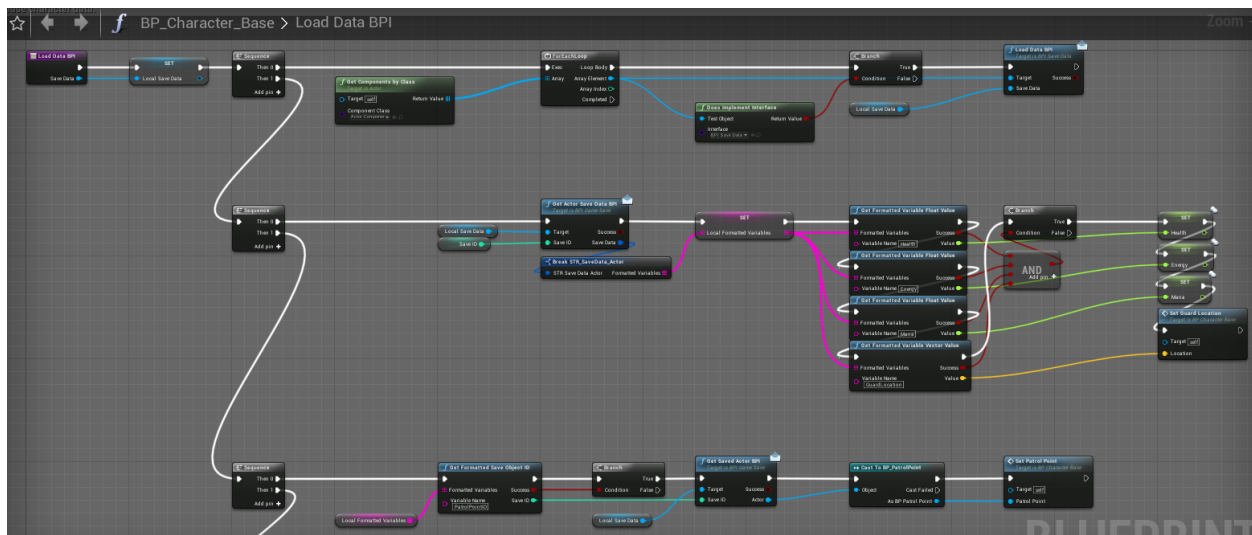
Then need to implement the **GetSaveData** and the **SetSaveData** functions in the **BP_GameSave_Session** blueprint.



The **SaveData_BPI** function with formatted implementation.

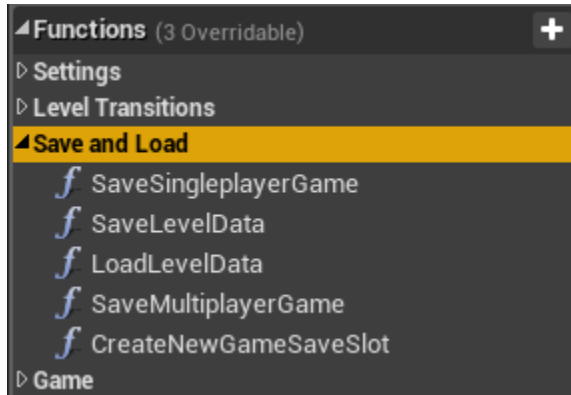


The **LoadData_BPI** function with formatted implementation.

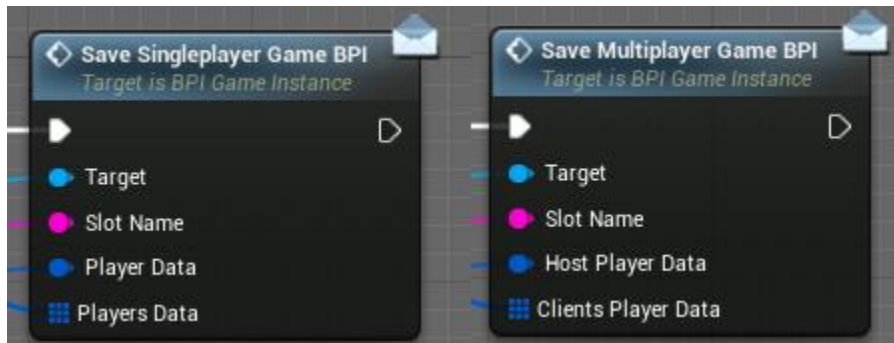


3.7.8. Saving & loading a level

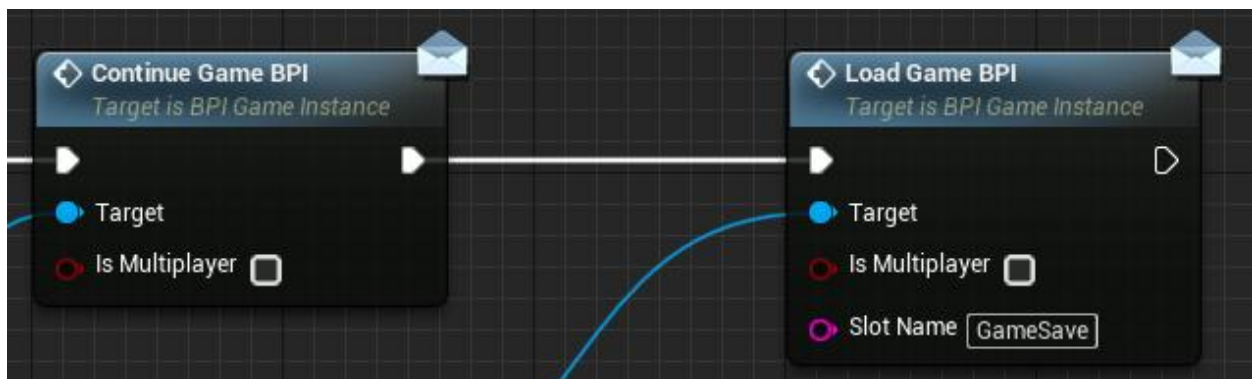
Functions for saving and loading level objects are located in the **BPI_GameInstance** interface and are implemented in **BP_GameInstance**.



The **SaveSingleplayerGame_BPI** function is used to save the game in single player mode, and the **SaveMultiplayerGame_BPI** function is used to save the game in multiplayer mode. These functions call the **SaveLevelData** function.



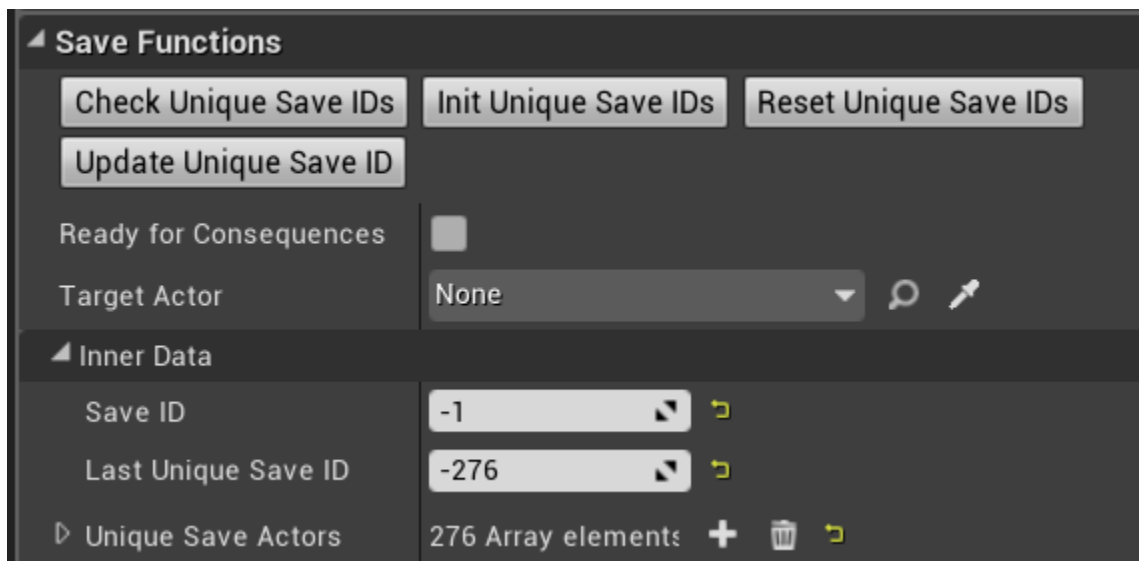
The functions **ContinueGame_BPI** and **LoadGame_BPI** are used to load the game.



The data of the current level is saved in the **SaveLevelData** function in the **BP_GameInstance** class. This function goes through all objects with the **BPI_SaveData** interface, the data of which needs to be saved, then forms this data in the **STR_SaveData_Level** structure and saves it in the current saving **BP_GameSave_Session**.

The level data is loaded in the **LoadLevelData** function in the **BP_GameInstance** class. The function removes all objects with the **BPI_SaveData** interface from the game, if they have wrong **SaveID**, and then loads and creates objects from the **STR_SaveData_Level** structure.

All actors placed on the level must have a unique **SaveID** or they will be removed by the loading process. To set a unique **SaveIDs**, place the **BP_LevelUpdater** on the level and click the **InitUniqueSaveIDs** button in the details. Use the **CheckUniqueSaveIDs** button to check that all unique **SaveIDs** are set correctly. If for some reason the data for placed on level actors load wrong use the **ResetUniqueSaveIDs** button.



To update details of specific actor for existing game saves select this actor in the **TargetActor** variable and use the **UpdateUniqueSaveID** button.

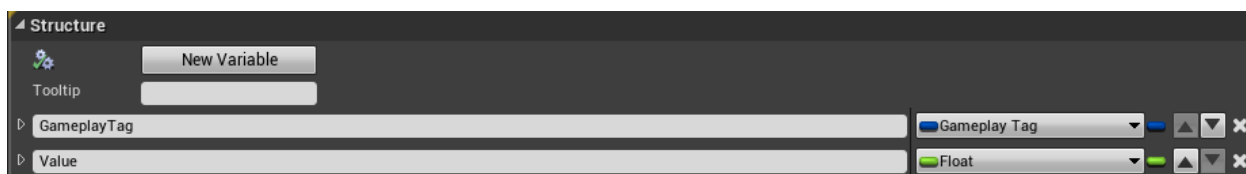
3.8. Attributes system

3.8.1. Description

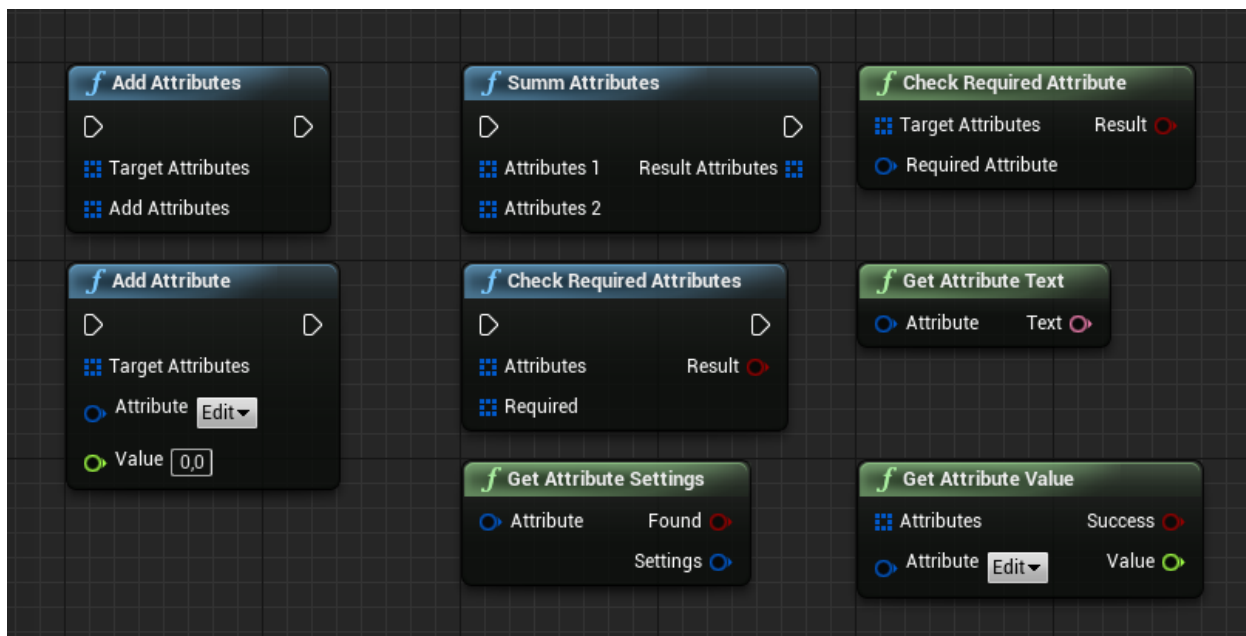
The functionality for working with character attributes is located in the **BP_AttributesComponent** component. These attributes include maximum health and energy, damage, defense, strength, agility, stamina, and other attributes. The component allows you to get and calculate the values of the attributes.

The attribute system is based on gameplay tags that are stored in the **DT_GameplayTags_Attributes** table. You can easily add new attributes, characteristics, skills and perks to it.

The **STR_AttributeValue** structure is used to store the attribute data.



Functions for performing calculations with attributes are located in the **BP_GameLibrary**.



The **STR_AttributesSettings** structure and **DT_Attributes** data table are used for advanced attributes settings. The information about attributes, which will be displayed in the user interface, can be configured in this data table. The row name for a certain attribute should be equal to its tag.

Data Table	
Search	
	Tag
EasyRPG.Attributes.Base.MaxHealth	{ "TagName": "EasyRPG.Attributes.Base.MaxHealth" }
EasyRPG.Attributes.Base.HealthRegeneration	{ "TagName": "EasyRPG.Attributes.Base.HealthRegeneration" }
EasyRPG.Attributes.Base.MaxEnergy	{ "TagName": "EasyRPG.Attributes.Base.MaxEnergy" }
EasyRPG.Attributes.Base.EnergyRegeneration	{ "TagName": "EasyRPG.Attributes.Base.EnergyRegeneration" }

Each attribute can affect other attributes. These dependencies also can be configured in the data table.

Row Editor	
+ x EasyRPG.Attributes.Charac Row Name	
Row Value	
Tag	Edit EasyRPG.Attributes.Characteristics.Intelligence
Name	Intelligence
Description	Affects the base damage to ore veins and trees when gathering r
ValueType	Integer
Visible	<input checked="" type="checkbox"/>
Dependencies	4 Array elements + -
0	2 members
1	2 members
2	2 members
GameplayTag	Edit EasyRPG.Attributes.Base.MaxMana
Value	10,0
3	2 members
GameplayTag	Edit EasyRPG.Attributes.Base.ManaRegeneration
Value	0,25

For example 1 intelligence attribute grants 10 max mana and 0.25 mana regeneration.

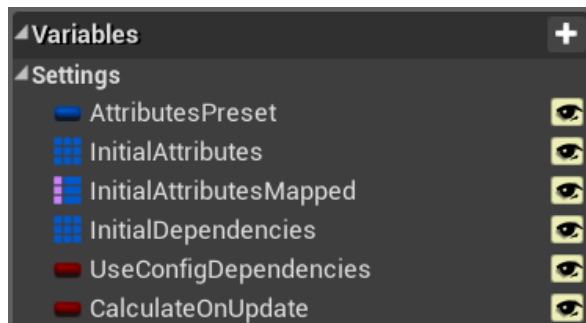
Only attributes that will be displayed in the user interface, or those that affect others, can be configured.

3.8.2. BP_AttributesComponent

Description

The functionality for working with character attributes is located in the **BP_AttributesComponent** component. These attributes include maximum health and energy, damage, defense, strength, agility, stamina, and other attributes. The component allows you to get and calculate the values of the attributes.

The component contains a list of initial attributes and attribute dependencies that can be easily customized. Initial attributes and dependencies can be loaded using the



AttributesPreset from the **DT_Attributes_Presets** data table.

There are also several additional attribute lists that can be updated during the game. These lists include the list of attributes selected by the player, the list of attributes given by equipment items, the list of attributes given by status effects, and the list of attributes of the item selected in the hotbar. If the **CalculateOnUpdate** variable is **true** the **CalculateAttributes** function is executed whenever any of these attribute lists changes, otherwise the function will be executed only once on the next update tick.

The **CalculateAttributes** function also calls the **OnAttributesChanged** event.

The **STR_AttributeDependence** structure is used to store attribute dependency data.

The attributes component allows you to customize the dependencies of some attributes on others. Attribute dependencies can be configured in the **InitialDependencies** variable. By default initial dependencies variable is not set. Overall dependencies are calculated in the **CalculateAttributeDependencies** function and it is called every time when any list of attributes is updated in the **CalculateAttributes** function. In addition the component uses

config attribute dependencies from the **DT_Attributes**. Config dependencies and initial dependencies are combined. For using only initial dependencies you need to disable the **UseConfigDependencies** variable.

3.8.3. Character attributes

Character attributes can be configured individually for each character in the attributes component of the character. Some of these attributes are very important and used for some character systems. These attributes have the **EasyRPG.Attributes.Base** prefix.

MaxHealth - an attribute that affects the maximum health variable of the character.

HealthRegeneration - an attribute that affects the health regeneration variable of the character.

MaxEnergy - an attribute that affects the maximum energy variable of the character.

EnergyRegeneration - an attribute that affects the energy regeneration variable of the character.

MaxMana - an attribute that affects the maximum mana variable of the character.

ManaRegeneration - an attribute that affects the mana regeneration variable of the character.

Damage - an attribute that affects the overall damage which the character can deal.

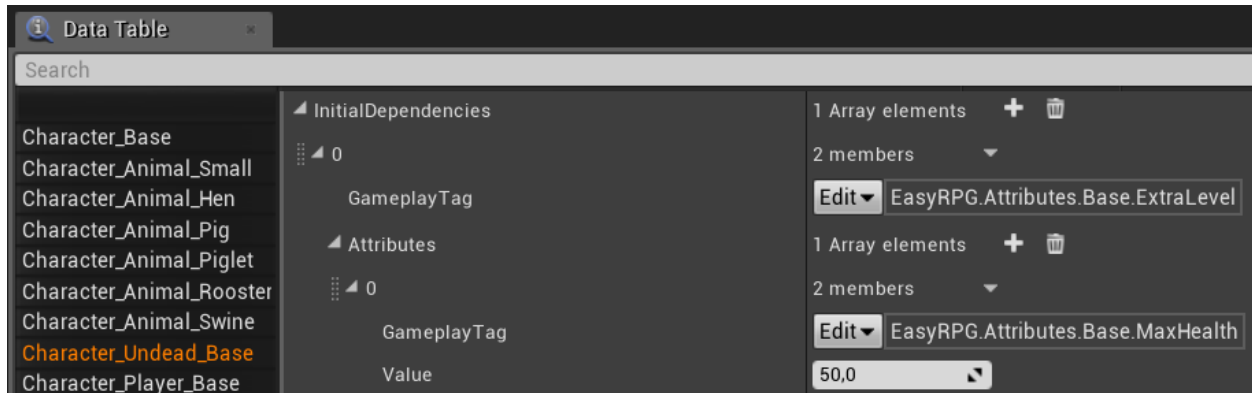
Armor - an attribute that affects the overall character damage resistance.

AttackRange - an attribute that affects trace distance in the **TraceDealDamage** function.

Level - initial level of the character.

Extra level - difference between initial and current level of the character.

NOTE: The level and extra level attributes can be used for the level scaling system. For example you can spawn enemy characters which level depends on the level of the player and their attributes will scale. The screenshot below shows how you can scale the health of the character depending on character level.



3.8.4. Ability system attributes

Ability attributes are used for advanced ability settings. It can be configured individually for each ability in the **DT_Abilities**. Some of these attributes are very useful. The ability attributes have the **EasyRPG.Attributes.AbilitySystem** prefix.

Cost - attributes which check the certain state of the character before using ability.

For example: Cost.Mana = 10 checks the current mana variable of the character and breaks using ability if current mana of the character is less than 10.

Trace.Distance - attribute which is used in the trace functions in the ability blueprint.

Projectile.Damage - attributes which are used for dealing damage when projectiles call the hit functions.

Projectile.Radius - attributes which are used in sphere trace functions of the projectile.

Projectile.Speed - attribute which sets initial velocity of the projectile.

Additional status effect attributes can be added in the

DT_GameplayTags_Attributes_AbilitySystem.

3.8.5. Status effects attributes

Status effect attributes are used for advanced effect settings. It can be configured individually for each status effect in the **DT_StatusEffects**. Some of these attributes are very useful. The ability attributes have the **EasyRPG.Attributes.StatusEffect** prefix.

Duration - an attribute which sets the duration of the status effect.

TickInterval - an attribute which sets the tick interval of the status effect.

Damage - attributes which set damage variables in the status effect blueprint.

Health - attributes which set health variables in the status effect blueprint.

Energy - attributes which set energy variables in the status effect blueprint.

Mana - attributes which set mana variables in the status effect blueprint.

Hunger - attributes which set hunger variables in the status effect blueprint.

Thirst - attributes which set thirst variables in the status effect blueprint.

Oxygen - attributes which set oxygen variables in the status effect blueprint.

Additional status effect attributes can be added in the

DT_GameplayTags_Attributes_StatusEffects.

3.8.6. Damage system attributes

Damage system attributes are used in the advanced damage system. Each attribute is a bonus damage modifier or bonus damage resistance for a special damage type. These attributes have the **EasyRPG.Attributes.DamageSystem** prefix.

Damage.Overall - attributes which are bonus modifiers to the default damage type.

Damage.Melee - attributes which are bonus modifiers to the melee damage type.

Damage.Ranged - attributes which are bonus modifiers to the ranged damage type.

Damage.Explosion - attributes which are bonus modifiers to the explosion damage type.

Damage.Fire - attributes which are bonus modifiers to the fire damage type.

Resistance.Overall - attributes which are resistances to the default damage type.

Resistance.Melee - attributes which are resistances to the melee damage type.

Resistance.Ranged - attributes which are resistances to the ranged damage type.

Resistance.Explosion - attributes which are resistances to the explosion damage type.

Resistance.Fire - attributes which are resistances to the fire damage type.

Additional damage system attributes can be added in the

DT_GameplayTags_Attributes_DamageSystem.

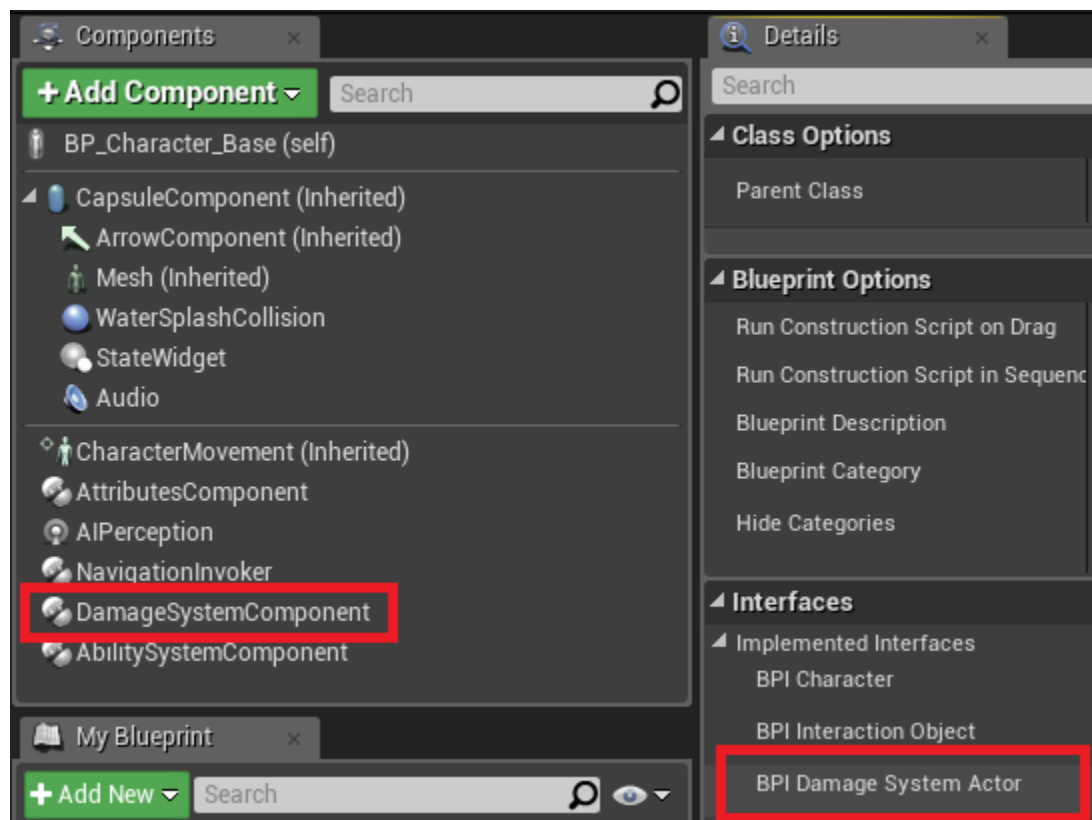
3.9. Advanced damage system

3.9.1. Description

The advanced damage system allows to use different damage types for dealing damage. The damage size increases depending on bonus damage attributes of the damage causer and decreases depending on resistance attributes of the target. Also this damage system allows users to use special hit flags when damage hits like critical hit, interrupt animations and so on.

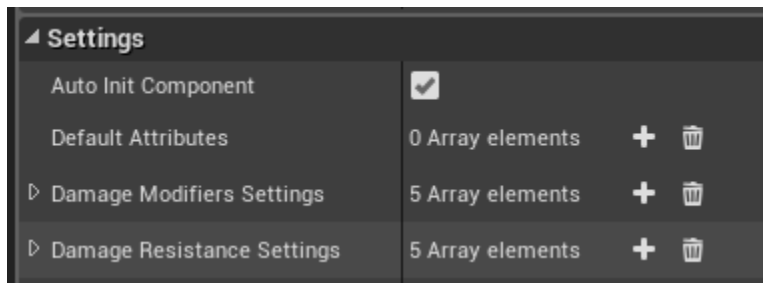
For working with the advanced damage system your actors must have

BP_DamageSystemComponent. Also actors which can take damage should implement the **ApplyPointDamage_BPI** function from **BPI_DamageSystemActor** interface.

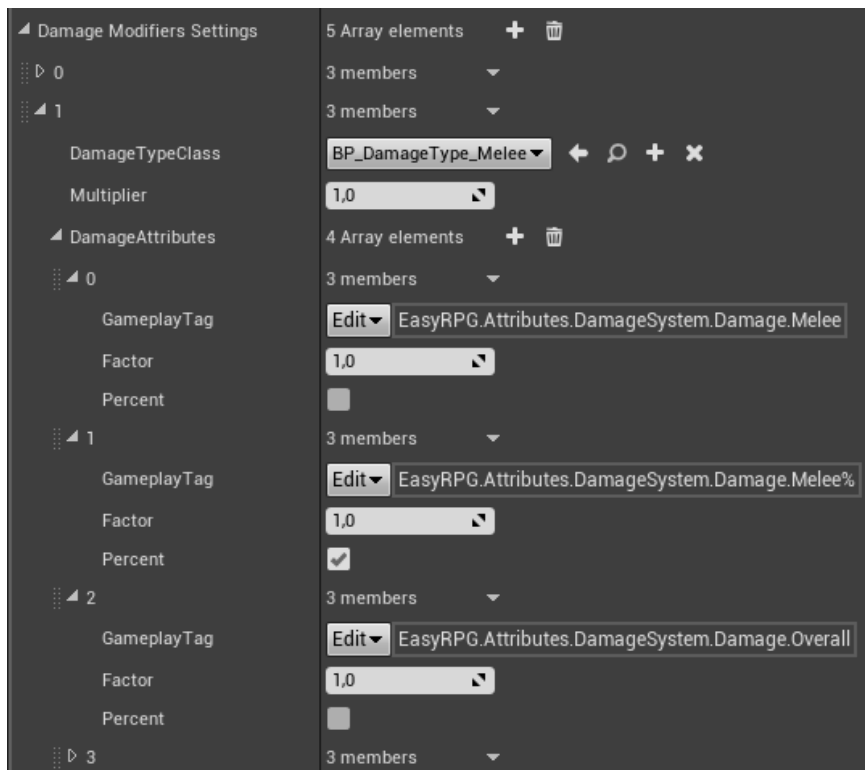


3.9.2. BP_DamageSystemComponent

The **BP_DamageSystemComponent** allows you to use different damage types and damage modifiers in apply damage functions. Damage modifiers and resistances for the certain damage type can be easily configured in the settings section of the damage system component in the actors. By default modifiers and resistance attributes are taken from the attributes component, but it can be set manually.



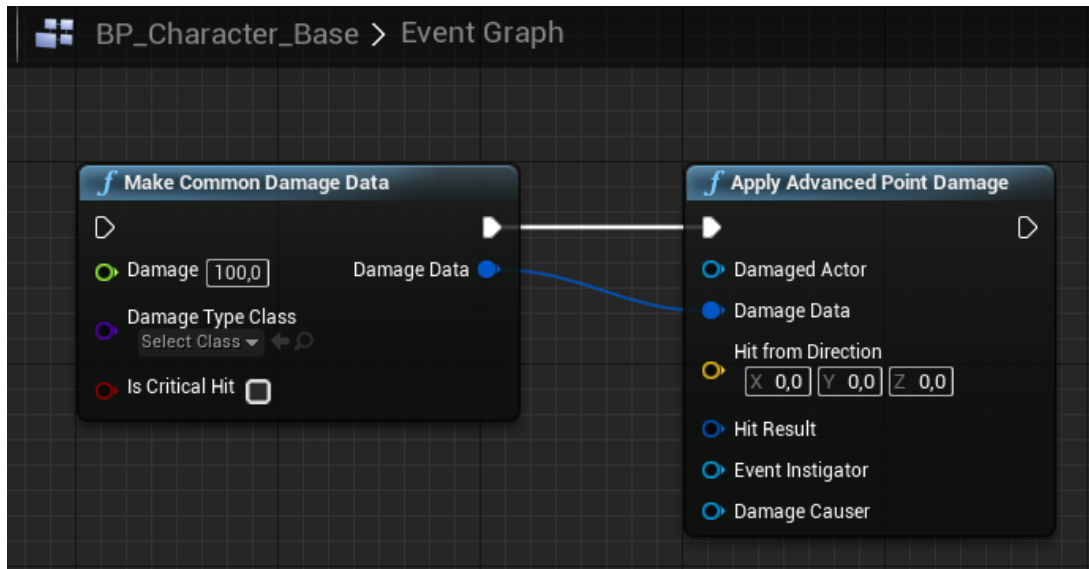
Each damage type by default has few modifiers and resistance attributes. Attributes which affect on damage type can be added or changed to another. These settings can be set up individually for each actor.



The attributes which are used in the damage system are in the [Damage system attributes](#) section.

3.9.3. Apply damage

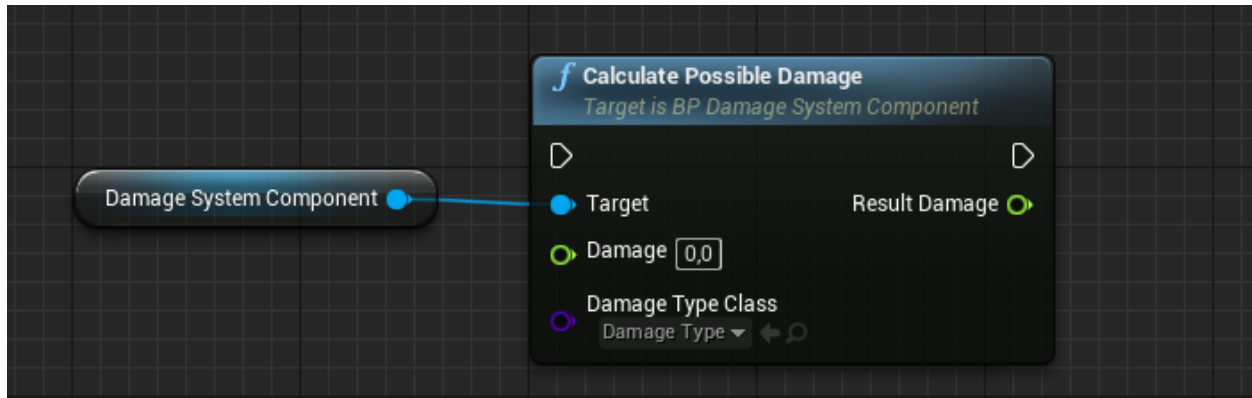
The **ApplyAdvancedPointDamage** function is used for dealing damage in the advanced damage system. The difference with the default engine **ApplyPointDamage** function is the **STR_DamageData**. This structure is used for transferring overall damage data (damage value, damage type and different damage modifiers) to the apply damage function.



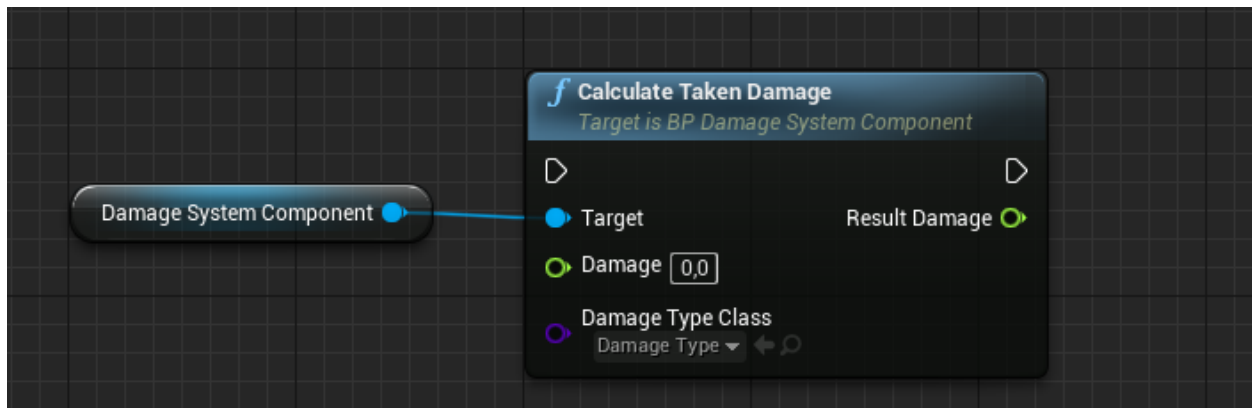
The damage data also can be made manually. Each additional modifier can be used for different purposes.



The **CalculatePossibleDamage** function of the damage system component can be used for calculating possible damage for a specific damage type. This function calculates damage depending on damage modifier attributes for the specific damage type. Also this function should be called before the **ApplyAdvancedPointDamage** function.



The **CalculateTakenDamage** function of the damage system component can be used for calculating taken damage of the specific damage type. This function calculates damage depending on damage resistance attributes for the specific damage type. Also this function should be called after the **ApplyAdvancedPointDamage** function in the implementation of the **ApplyPointDamage_BPI** function.



3.10. Advanced ability system

3.10.1. Description

The advanced ability system allows characters to use different abilities during the game. The effect of the ability could be anything, starting from health restoration for the character from some potions or food and ending with throwing meteorites from the sky. Also the system allows characters to apply different status effects, which can increase character attributes, heal, deal damage and so on.

The **BP_Ability_Base** is a base ability class. Each ability should be based on it. Abilities can be configured in the **DT_Abilities** using base ability classes and ability attributes. Abilities have their own cooldown identifier. If a cooldown is set and it is run, the character cannot use ability. By default the cooldown of the ability is the handle name. It can be changed manually in the data table. Also several abilities can have the same cooldown.

The **BP_StatusEffect_Base** is a base status effect class. Each status effect should be based on it. Status effects can be configured in the **DT_StatusEffects** using base status effect classes and status effect attributes. The status effect icon, name and description also can be configured in the data table. By default the status effect actor is not replicated, but if you need to replicate visual or audio effects in the world you need to turn it on.

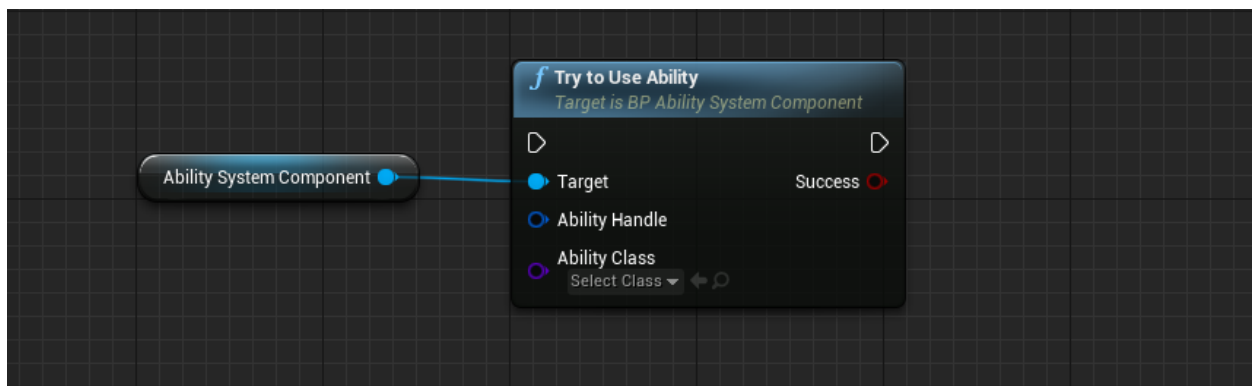
The **BP_AbilitySystemComponent** allows characters to use abilities and applies status effects. This component should be added to each character blueprint.

Some additional functions for the ability system are in **BP_AbilitySystemLibrary**. It contains functions for working with abilities and status effects without ability system component reference.

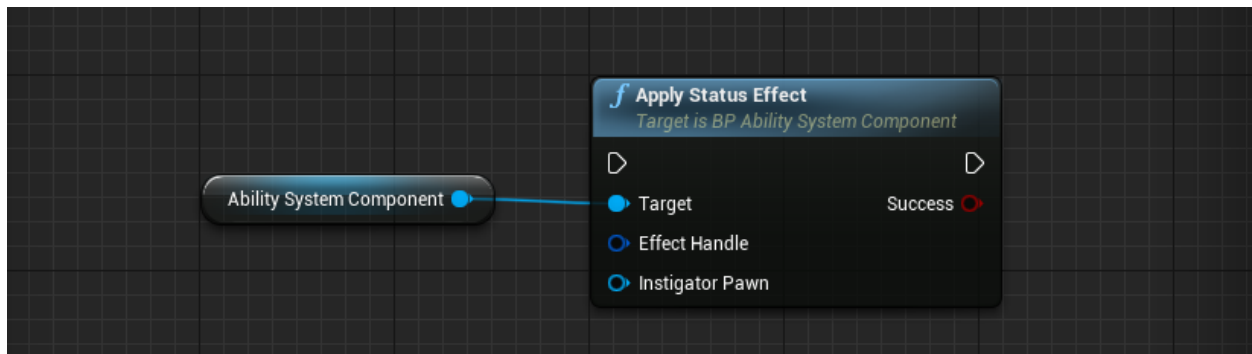
3.10.2. BP_AbilitySystemComponent

The **BP_AbilitySystemComponent** allows characters to use abilities and checks their cooldowns and other conditions before using. This component should be added to each character blueprint. Also this component allows to apply different status effects to the owning character. The active status effects save during the game save and can be loaded.

For using ability you need to call the **TryToUseAbility** function. First it checks ability cooldown, ability costs and other conditions and then uses ability if everything is okay.



For applying the status effect to the character you need to call the **ApplyStatusEffect** function. The function spawns a status effect actor and registers it in the system.



The **OnStatusEffectAdded**, the **OnStatusEffectUpdated** and the **OnStatusEffectRemoved** events are used for updating active status effects on the user interface. The **OnStatusEffectsAttributesChanged** event is used for updating status effects attributes in the character's attributes component.

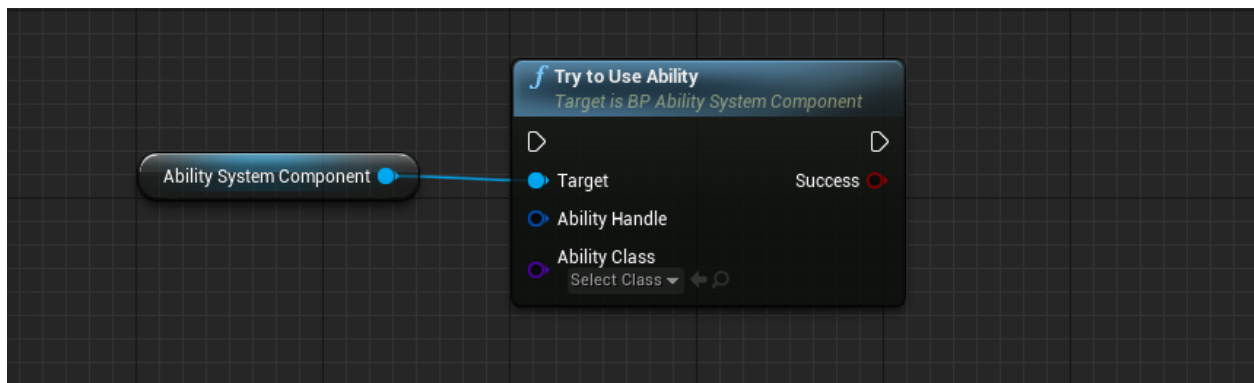
Events	
On Status Effect Added	+
On Status Effect Updated	+
On Status Effect Removed	+
On Status Effects Attributes Changed	View

3.10.3. Working with ability blueprints

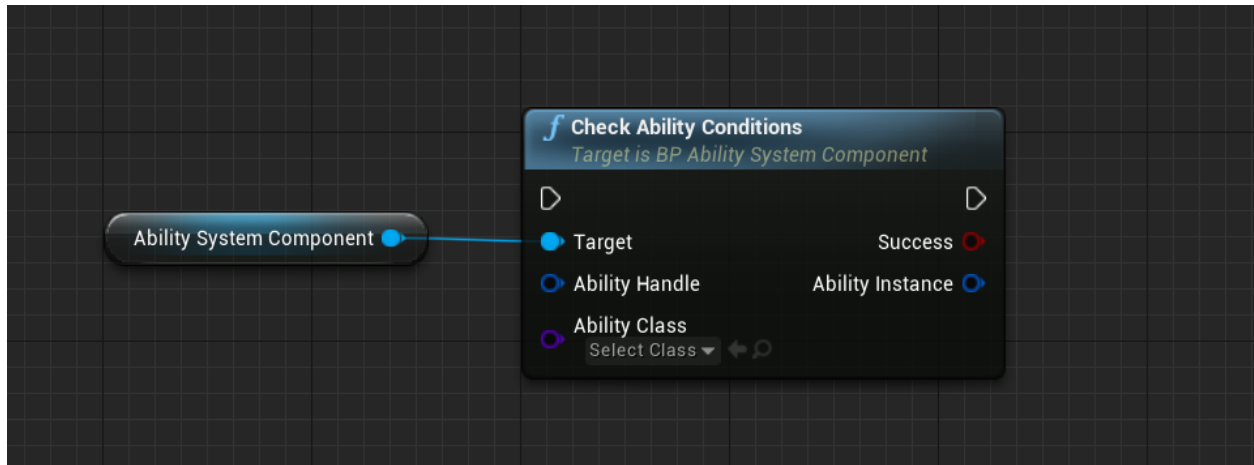
The **BP_Ability_Base** is a base ability class. Each ability should be based on it. Abilities can be configured in the **DT_Abilities** using base ability classes, ability attributes and attribute dependencies. You can create your own custom data table for abilities based on the **STR_AbilityInstance** structure.

Abilities have their own cooldown identifier. If a cooldown is set and it is run, the character cannot use ability. By default the cooldown of the ability is its handle name. It can be changed manually in the data table. Also several abilities can have the same cooldown.

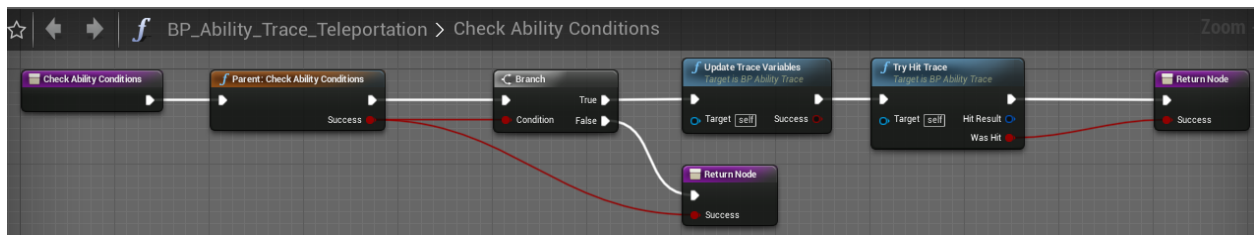
The ability can be used with the **TryToUseAbility** function of the ability system component.



The **TryToUseAbility** function also checks ability conditions before using. This check also can be used with the **CheckAbilityConditions** function. There can be different conditions such as cooldown check, manacost check and additional hard coded conditions.

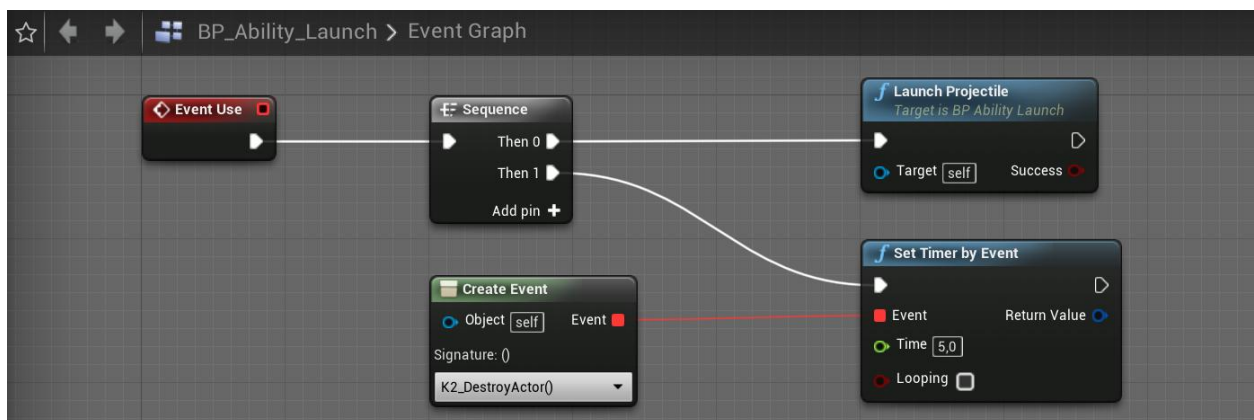


For example: the overridden **CheckAbilityConditions** function in the **BP_Ability_Trace_Teleportation** ability blueprint checks the trace hit result.



If ability conditions complete the ability system component spawns ability actor and calls the **Use** function of the spawned ability.

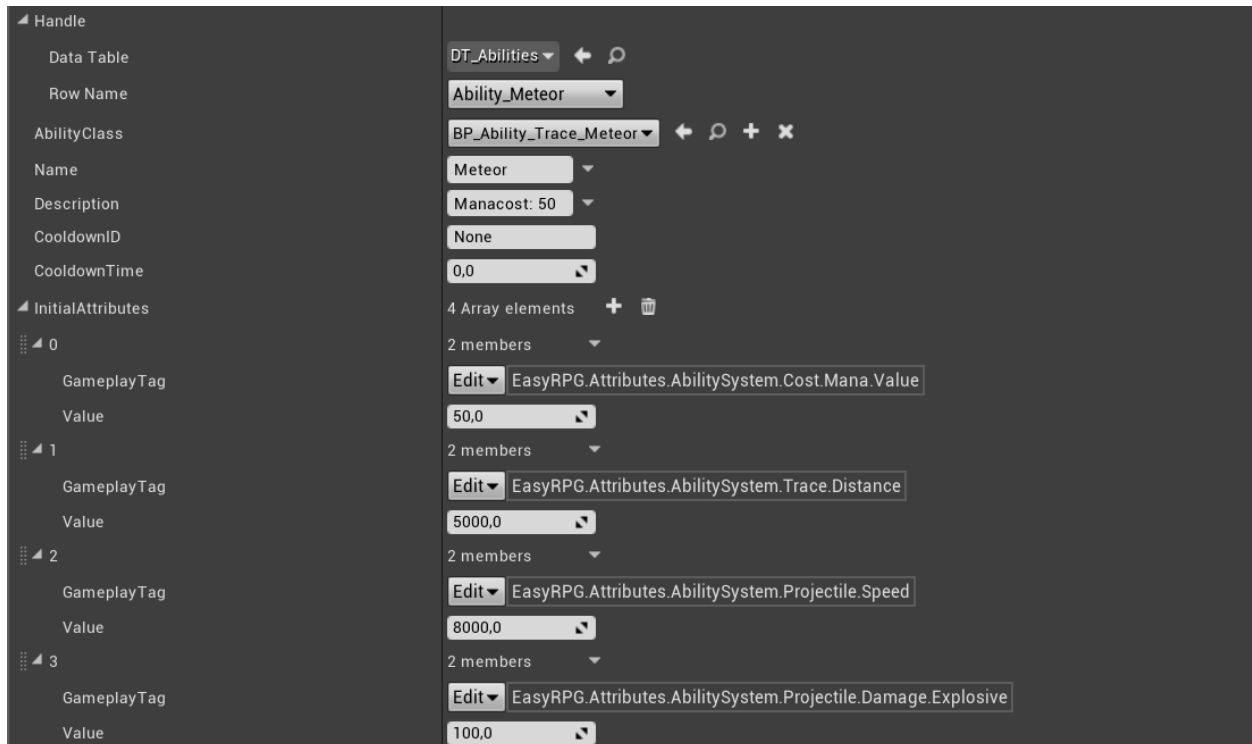
Each ability blueprint should have its own **Use** event. The ability actor should be destroyed after using it for better performance.



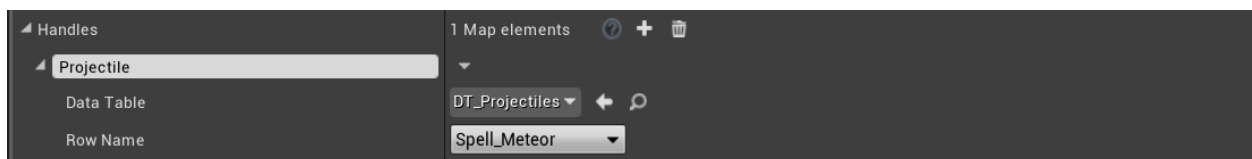
The ability blueprint can use ability attributes for updating ability variables such as mana cost, projectile speed, damage, trace distance and so on. It can be set in the ability blueprint in the InitialAttributes variable or it can be set in the data table .

The attributes which are used for abilities are in the [Ability system attributes](#) section.

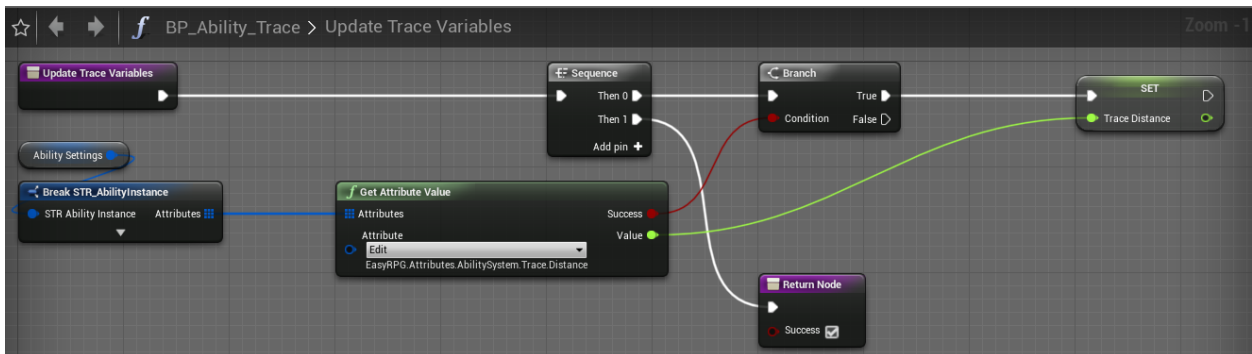
Screenshot below shows how ability attributes can be set for ability in the data table.



The handles property can be used for connection ability with other data tables.



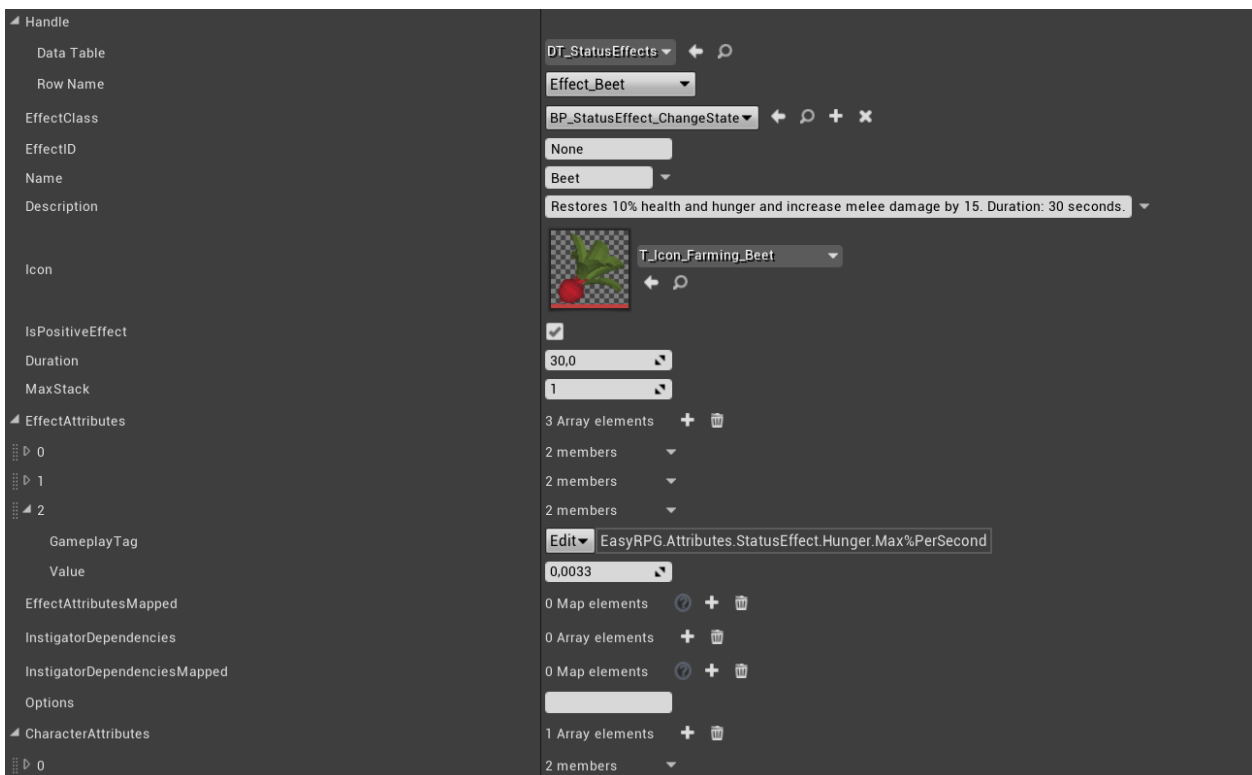
Different abilities use different attributes so each ability should implement its own update variable functions.



3.10.4. Working with status effects.

The **BP_StatusEffect_Base** is a base status effect class. Each status effect should be based on it. Status effects can be configured in the **DT_StatusEffects** using base status effect classes and status effect attributes. You can create your own custom data table for abilities based on the **STR_StatusEffectInstance** structure.

The status effect icon, name and description also can be configured in the data table.



By default the status effect actor is not replicated, but if you need to replicate visual or audio effects in the world you should turn it on.

Status effects with the same **EffectIDs** replace each other. By default the **EffectID** of status effect is equal to the **Handle** name, but it can be changed in the data table. For example, if you set the **EffectID** property to “**Food**” for the corn and the beef status effects in the data table, they will replace each other when applied to character.



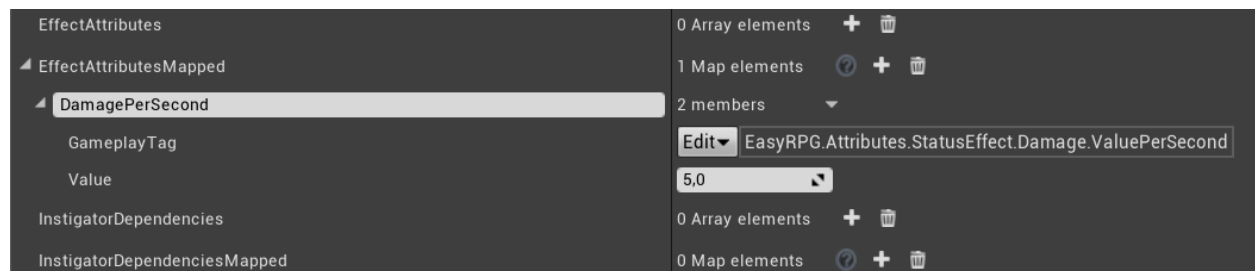
The **IsPositiveEffect** variable determines the color of the status effect tooltip.



The stack of the status effect multiply character attributes and other effects.



The status effect blueprint can use effect attributes from the status effect data table for updating status effect variables. You can configure instigator dependencies for scaling status effects attributes. For example you can scale damage per second depending on the intelligence attribute of instigator character.

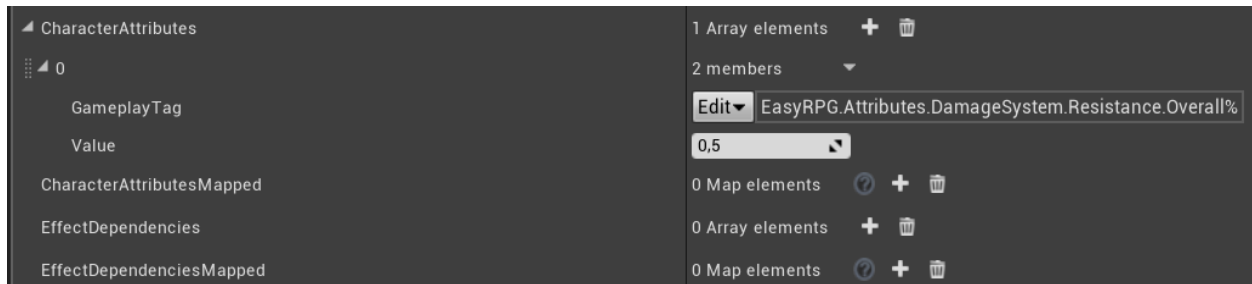


Different status effects use different attributes so each status effect should implement its own **UpdateVariables** function.

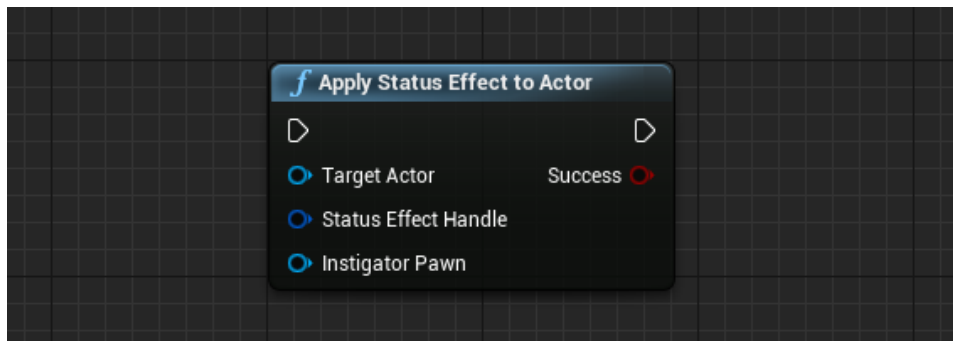
The attributes which are used for status effects are in the [Status effects attributes](#) section.

The status effect can also add additional attributes to owning character. It also can be set in the data table. You can configure effect dependencies for scaling status additional

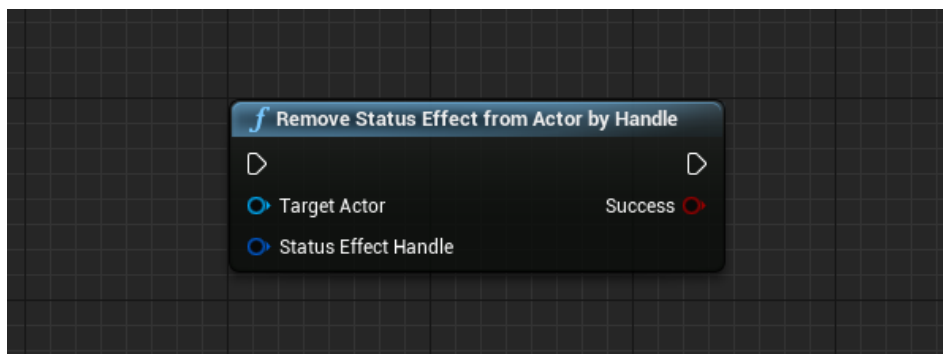
character attributes. For example you can scale resistance depending on the intelligence attribute of instigator character.



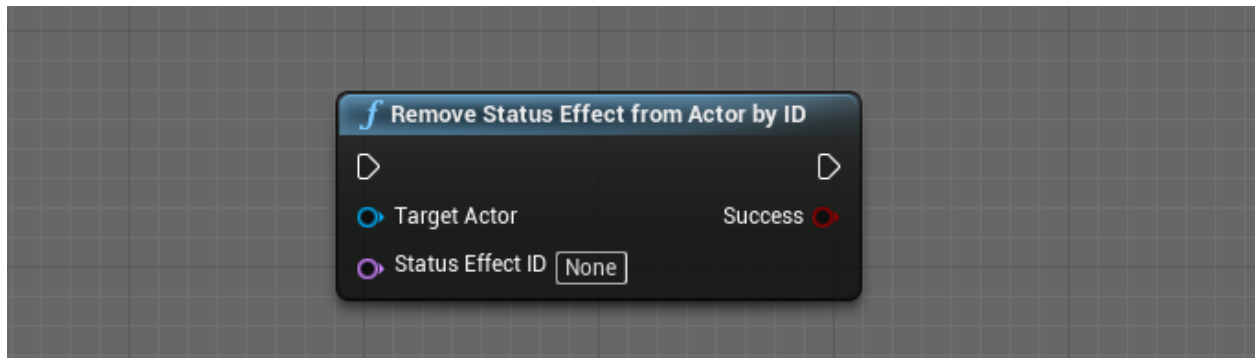
The status effect can be applied with the **ApplyStatusEffectToActor** function. If the target actor has an ability system component it will be added and registered as an active status effect.



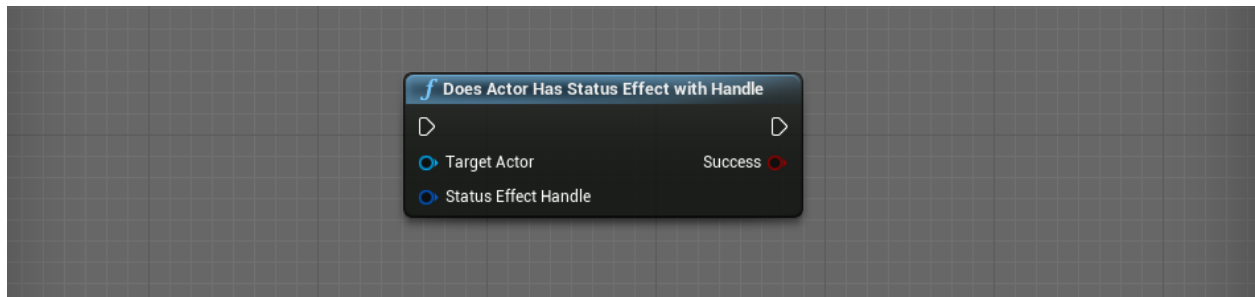
The status effect can be removed from the character with the **RemoveStatusEffectFromActorByHandle** function.



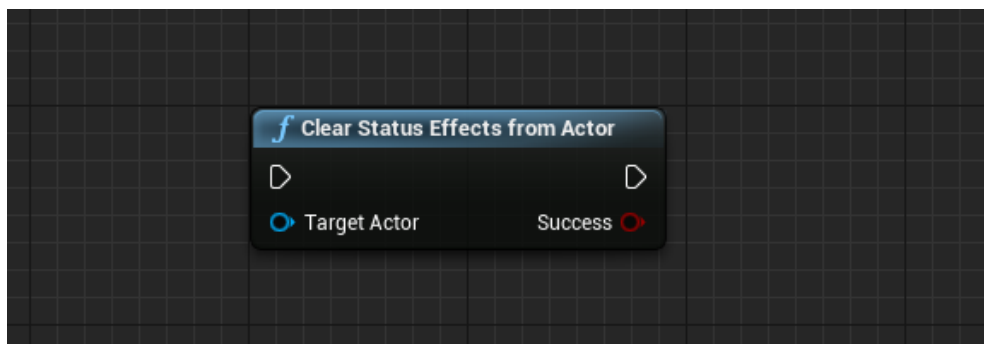
The status effect with certain **EffectID** can be removed from the character with the **RemoveStatusEffectFromActorByID** function.



The **DoesActorHasStatusEffectWithHandle** function can be used for checking a special status effect which is active for the character.

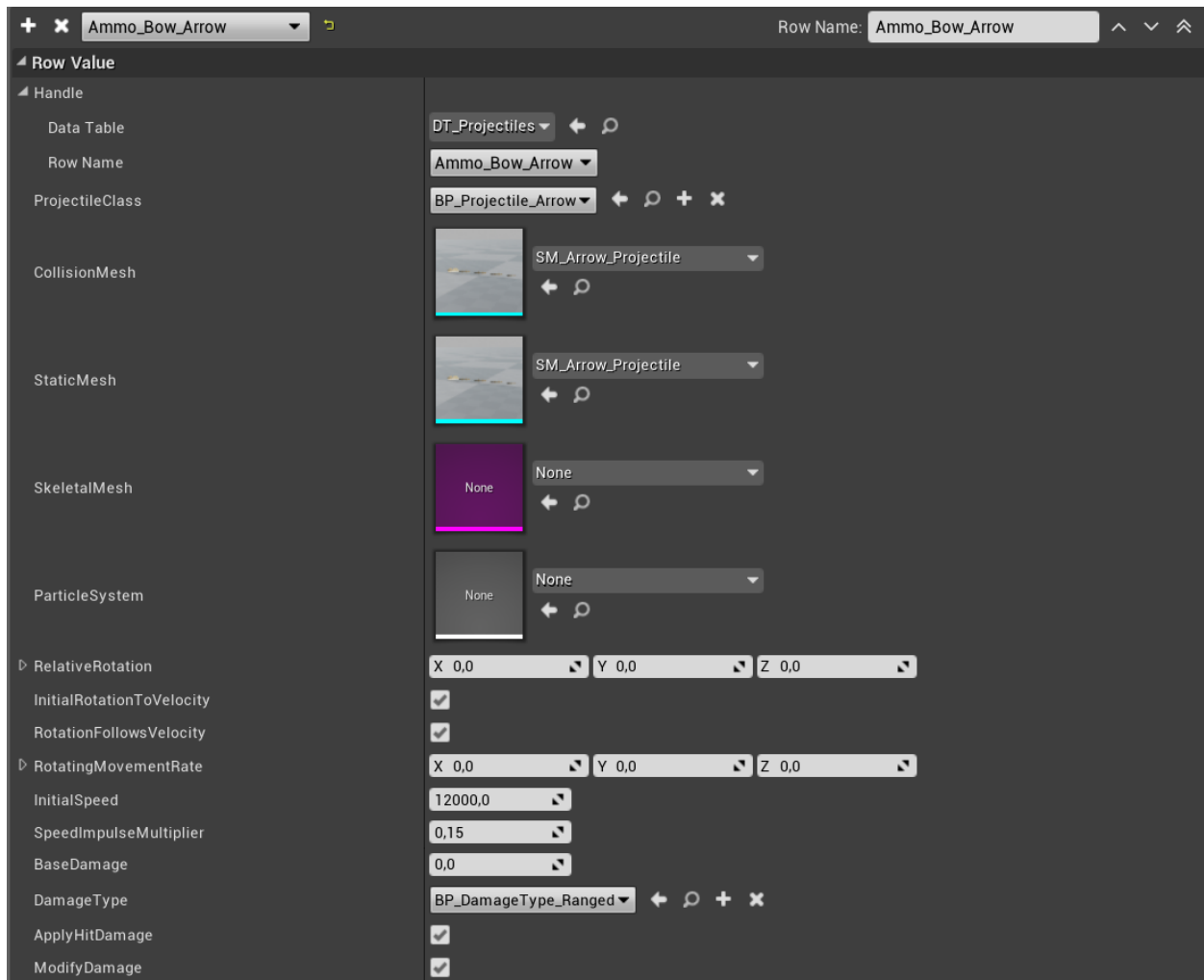


To clear each status effect from the character, you need to call the **ClearStatusEffectsFromActor** function.



3.10.5. Working with projectiles.

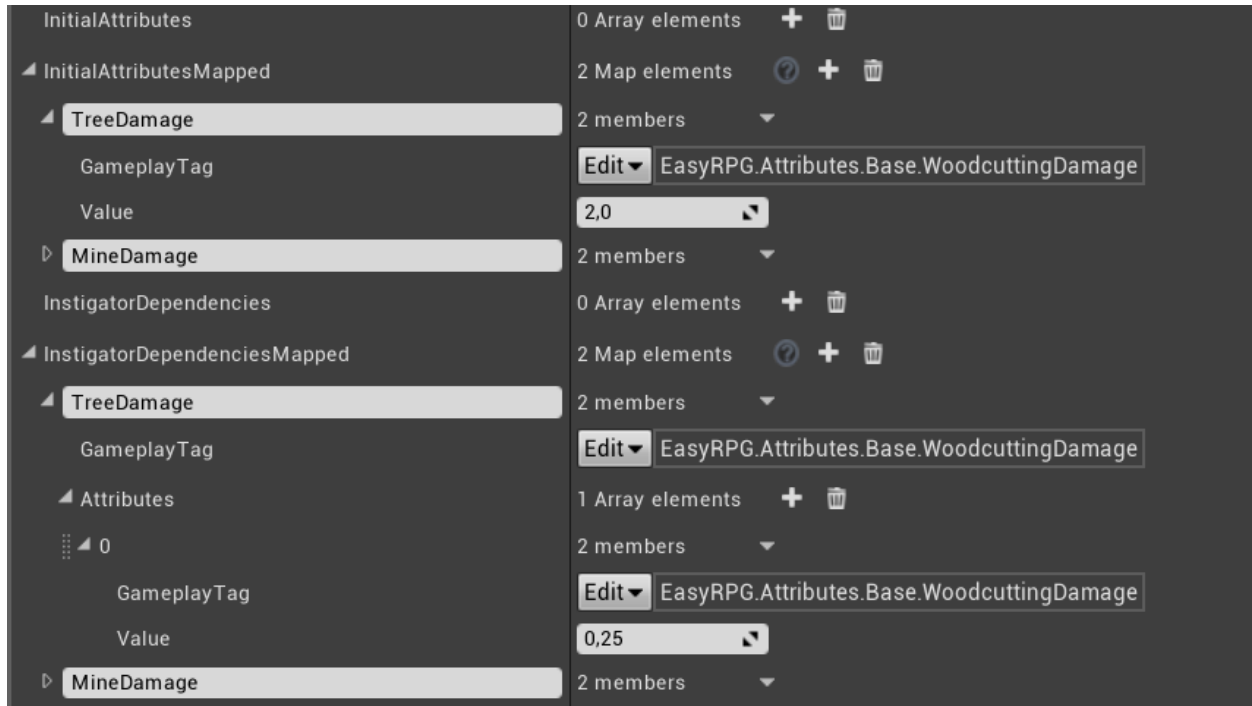
The **BP_Projectile_Base** is a base projectile class. Each projectile should be based on it. Projectiles can be configured in the **DT_Projectiles** using base projectile classes, projectile attributes and attribute dependencies. You can create your own custom data table for projectiles based on the **STR_ProjectileInstance** structure.



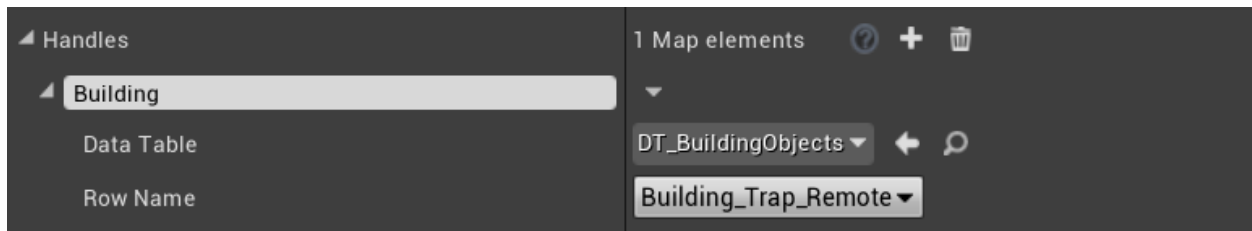
In the data table you can set projectile class, mesh, initial speed, damage and so on.

The **CollisionMesh** variable is used for collision checks while projectile moves and this mesh is always hidden. For visual display the **StaticMesh** variable is used.

The projectiles blueprint can use attributes from the projectiles data table for updating projectile variables. You can configure instigator dependencies for scaling projectile attributes. For example you can scale tree damage depending on the tree damage attribute of instigator character.



The handles property can be used for connection projectile with other data tables.

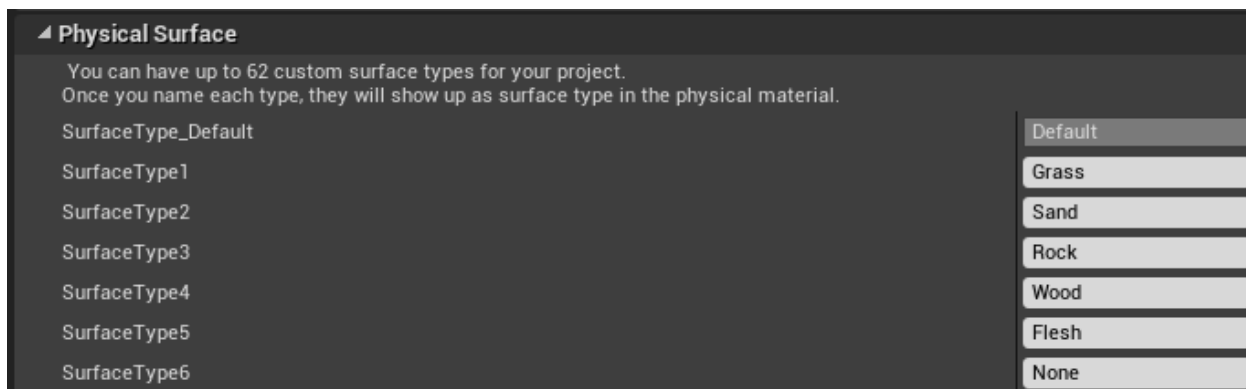


3.11. Footstep system

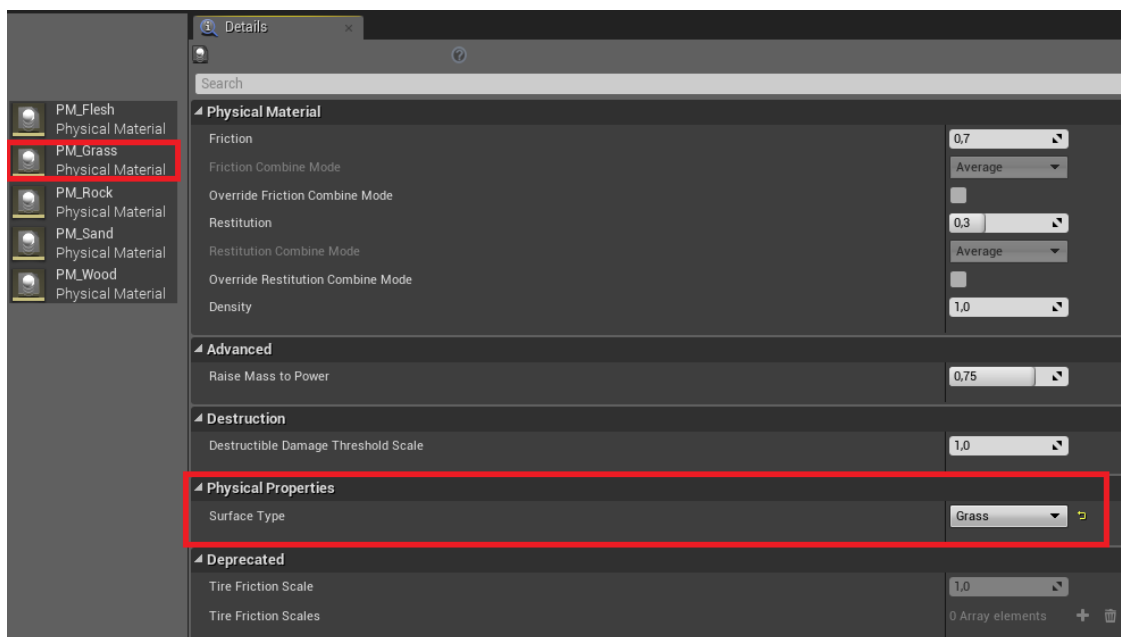
3.11.1. Description

The footstep system allows you to play sound and visual effects during the movement of the character, depending on the surface on which he steps. The system also automatically detects if the character is moving through the water and plays the appropriate sounds.

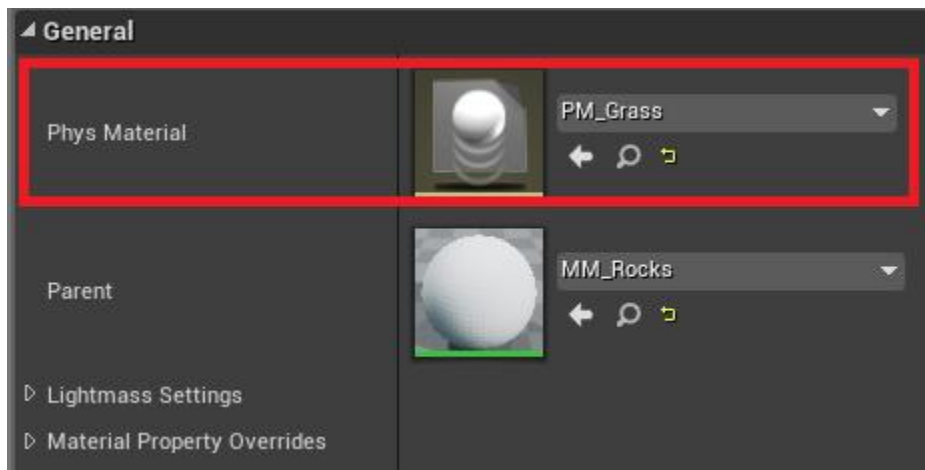
The system uses different types of physical surfaces, which are prescribed in the **Engine - Physics** section of the project settings.



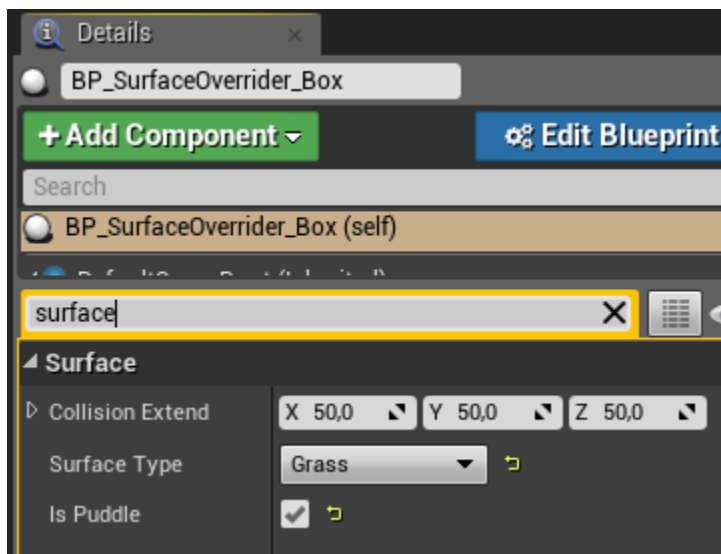
For materials that the character can move on, it is necessary to create and configure the appropriate physical materials.



In the settings for materials or layers, you can set the physical material that will be appropriate to them.



The surface on which the character moves can be overridden using the **BP_SurfaceOverrider_Base** actors by placing them in the right place.



CollisionExtend - sizes of the affected area.

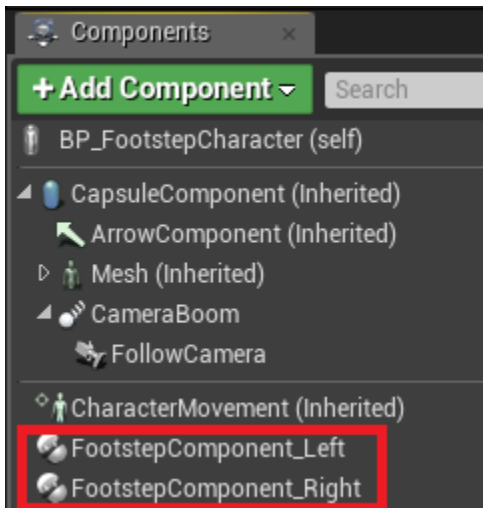
SurfaceType - surface type in the affected area.

IsPuddle - an indicator to play splash sounds in the affected area.

3.11.2. BP_FootstepComponent

3.11.2.1. Description

The main functionality of the system is located in the **BP_FootstepComponent** component. In this class, the process of finding a touching surface and reproducing sound and visual effects corresponding to this surface occurs. This component must be added to the character class, one for each leg.



Each footstep component can be configured separately or a ready-made set of **STR_FootstepSettings** can be loaded from the table.

Handling the event of touching the surface with the foot is called using animation notifications **BP_FootstepNotify**, which must be placed in the appropriate places in the animations.

BP_FootstepComponent handles the touch position and determines the type of surface, as well as the presence of a water surface at the touch point. Depending on this, the corresponding sound and visual effects are played.

The surface can be redefined using the **BP_SurfaceOverrider_Base** actor and classes inherited from it. In the settings of this class, you can select the surface that will be in the area of its action, as well as whether the sound of water will be played.

You can configure **BP_FootstepComponent** in the **Details** tab by selecting it in the list of character components.

3.11.2.2. Variables

Footsteps

PresetHandle - footstep settings preset data row handle. If selected, automatically loads footstep settings from the footstep presets data table.

FootSocket - bone or socket name of the foot for which footstep will be played.

SurfaceCheckStartOffset -

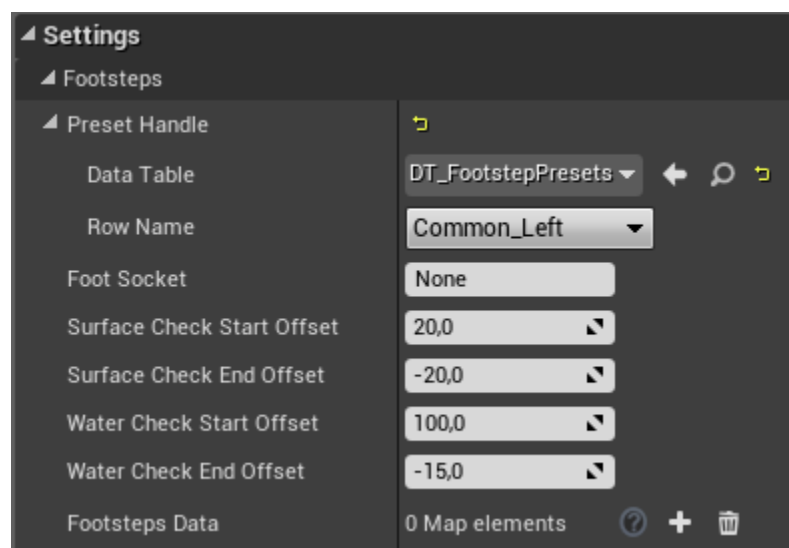
upper Z offset from **FootSocket** that is used in the check surface trace function.

SurfaceCheckEndOffset - lower Z offset from **FootSocket** that is used in the check surface trace function.

WaterCheckStartOffset - upper Z offset from **FootSocket** that is used in the check water trace function.

WaterCheckEndOffset - lower Z offset from **FootSocket** that is used in the check water trace function.

FootstepData - list of footstep type settings configured for different footstep types. Settings of sound and visual effects when the character touches various surfaces. Includes



settings for different footstep types, such as walking, running, jumping, and landing. Each footstep type includes footstep settings for different surfaces.

IK

IK_Enabled - enables simple IK calculations for the foot.

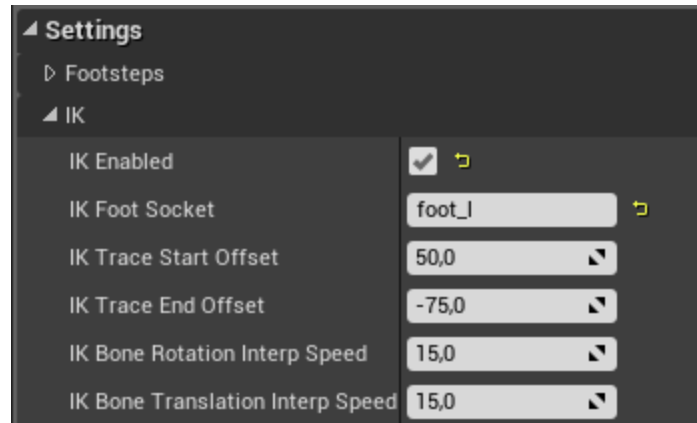
IK_FootSocket - bone or socket name of the foot for which the IK variables should be calculated.

IK_TraceStartOffset - upper Z offset from **IK_FootSocket** that is used in the simple IK calculations.

IK_TraceEndOffset - lower Z offset from **IK_FootSocket** that is used in the simple IK calculations.

IK_BoneRotationInterpSpeed - the speed of interpolation for current IK bone rotation variable.

IK_BoneTranslationInterpSpeed - the speed of interpolation for current IK bone translation variable.

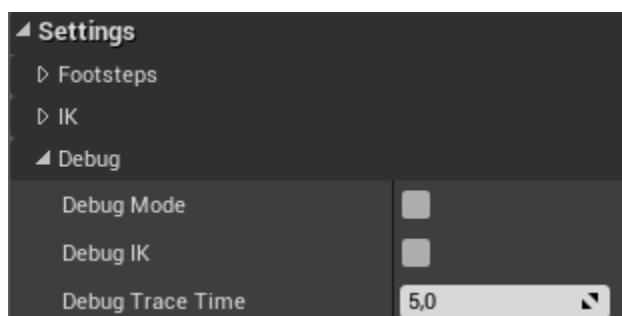


Debug

DebugMode - enables the debug mode for the footstep component.

DebugIK - enables the debug mode for IK calculations of the footstep component.

DebugTraceTime - the lifetime of debug traces.



Config

The screenshot shows a configuration window titled 'Config'. It contains several sections:

- Surface Override Object Type**: A dropdown menu set to 'SurfaceOverride'.
- Surface Override Object Type Query**: A dropdown menu set to 'SurfaceOverride'.
- Water Object Type**: A dropdown menu set to 'Water'.
- Water Object Type Query**: A dropdown menu set to 'Water'.
- Named Surface Types**: A section with a header '11 Map elements' and icons for help, add, and delete. It contains a list of 11 items, each with a text input field and a dropdown menu:
 - Default
 - Grass
 - Sand
 - Stone
 - Wood
 - Dirt
 - Gravel
 - Ice
 - Metal
 - Mud
 - Snow

SurfaceOverrideObjectType - config variable that is used for setting collision settings of the surface override type actors.

SurfaceOverrideObjectTypeQuery - config variable that is used for checking surface override type actors.

WaterObjectType - config variable that is used for setting collision settings of the water type actors.

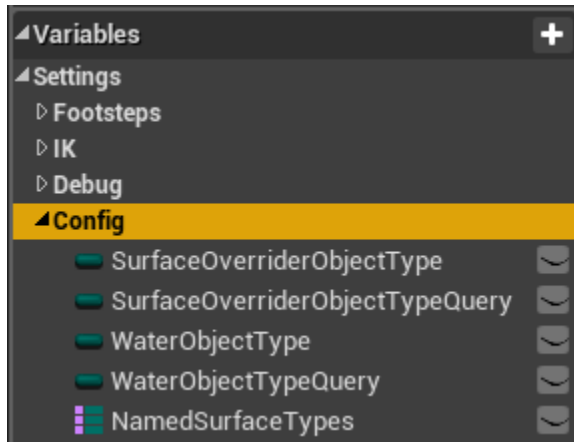
SurfaceOverrideObjectTypeQuery - config variable that is used for checking water type actors.

NamedSurfaceTypes - determines the correspondence of names - identifiers of surfaces with surface types. These identifiers can be used to set up footsteps in footstep components. This property also simplifies the setup of existing projects that already have

different types of physical surfaces and allows you to use prepared footstep settings presets from the **DT_FootstepPresets** table.

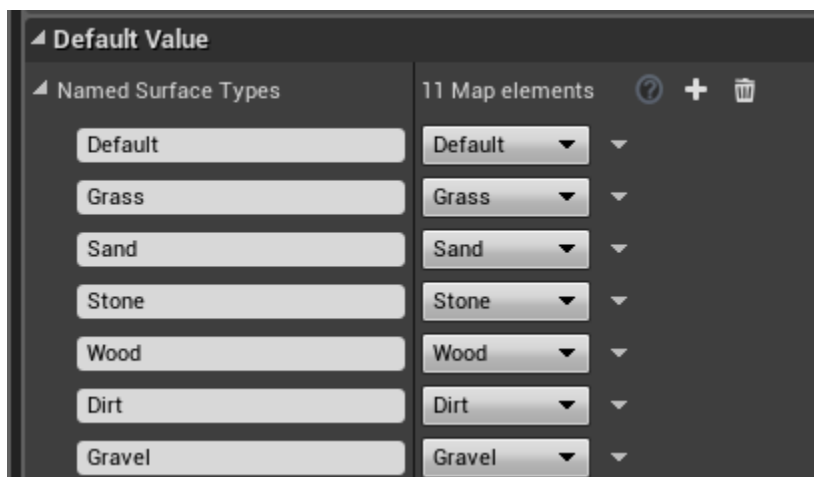
3.11.2.3. Config settings

Some variables in the **BP_FootstepComponent** must be configured for the system to work correctly. These variables are in the **Config** section in the component blueprint.

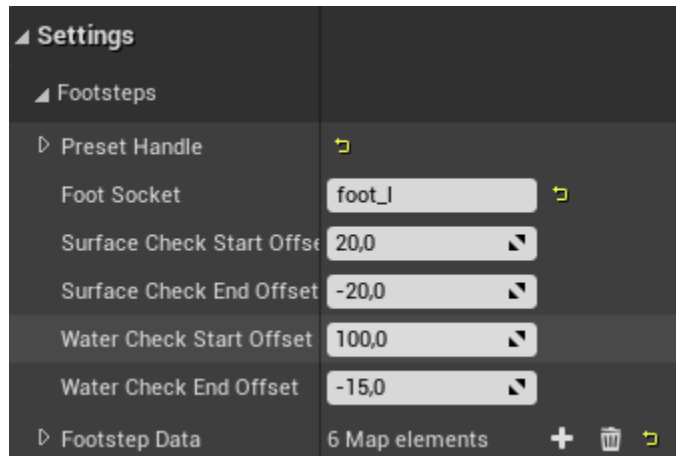


Select the appropriate collision channels for **Surface** and **Water** object types variables.

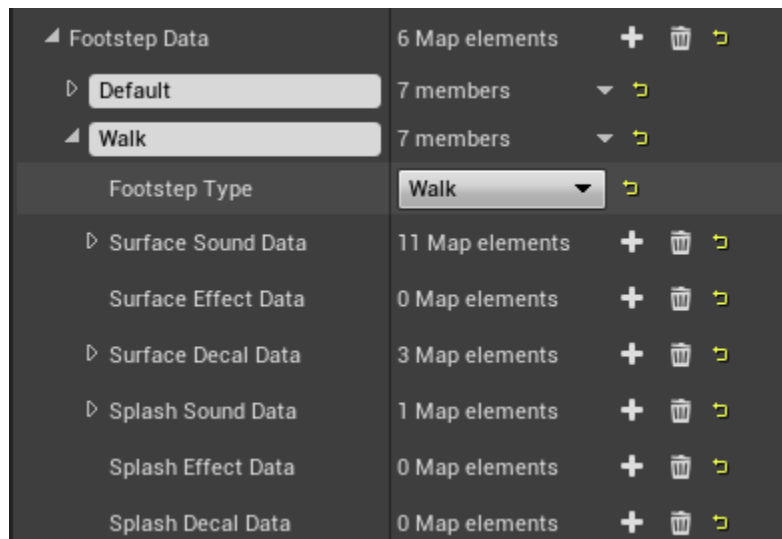
The **Named Surface Types** property determines the correspondence of names - identifiers of surfaces with surface types. These identifiers can be used to set up footsteps in footstep components. This property also simplifies the setup of existing projects that already have different types of physical surfaces and allows you to use prepared footstep settings presets from the **DT_FootstepPresets** table.



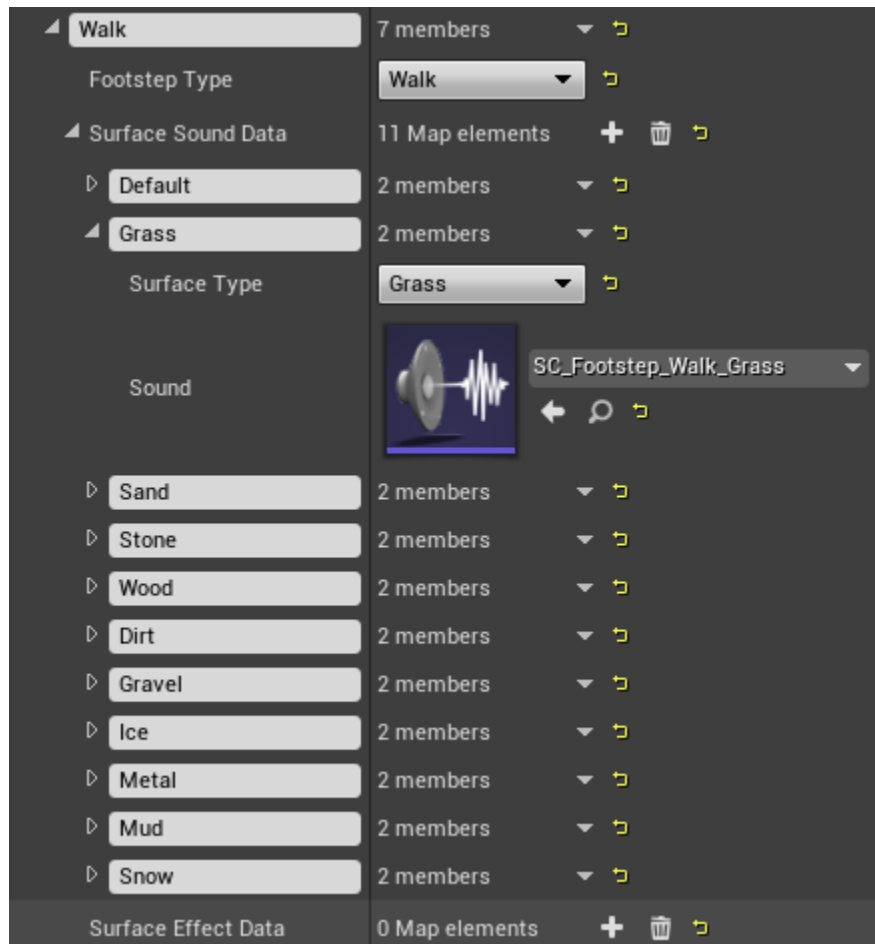
3.11.2.4. Footstep settings



The **FootstepData** property of the footstep component is used for setup footstep settings for different footstep types and different surfaces. This property can be loaded from the footstep presets data table. For this use the **PresetHandle** property of the footstep component.



Footstep settings can be configured separately for each footstep type. The footstep component tries to find settings by footstep type. Keys of the footstep type data items are not important (except **Default**) and are used only to simplify the settings. If settings for the certain footstep type are not found, then the settings with key **Default** are used.



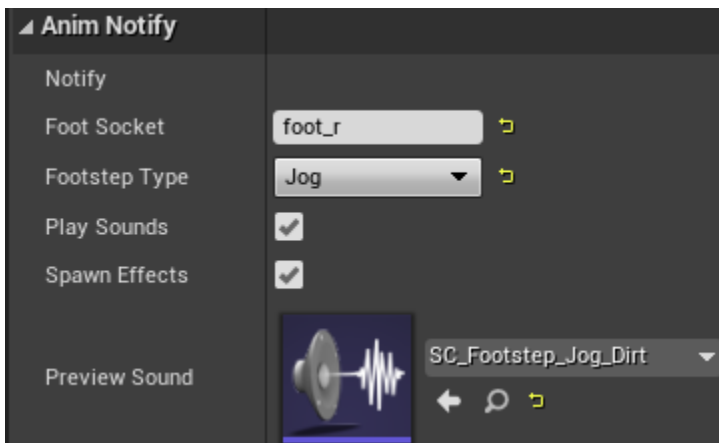
Sounds and effects for different surface types can be set here. The footstep component tries to find sounds or effects data by key from the **NamedSurfaceTypes** first. If the key is not found, then it tries to find data by surface type. If settings for the certain footstep type are not found, then the data with key **Default** are used.

3.11.4. Footstep animation settings

The process of footstep playing is invoked using animation notify **BP_FootstepNotify**.



These notifies must be placed in animations at the moment the foot touches the surface.



FootSocket - the name of the bone in which the footstep will be played. **FootSocket** must match the **FootSocket** variable in the footstep component.

FootstepType - type of the footstep.

PlaySounds - determines if the footstep component should play sounds.

SpawnEffects - determines if the footstep component should spawn effects.

PreviewSound - sound to play in the editor. Used only for preview.

3.12. Skills tree system

3.12.1. Description

The skills tree system allows you to learn skills which can add additional attributes or additional crafting blueprints. Each skill has its own attribute. The skill level affects on value of this attribute. Skills can require a certain level of the character or require certain attributes for learning. Each skill level can have its own unique requirements.

The **STR_SkillInstance** structure is used for store settings of skills in the **DT_Skills**. Each skill can be easily configured in this data table and used in skill trees.

The **STR_Skill** structure is used for storing actual data of learned skills in the save / load system.

The **STR_SkillData** structure is used for storing overall data of learned skills for calculating additional attributes and for adding additional crafting blueprints.

The **STR_SkillLevel** structure is used for storing different data of skill levels in other skill structures.

Skill attributes should be added in the **DT_GameplayTags_Attributes_Skills** data table.

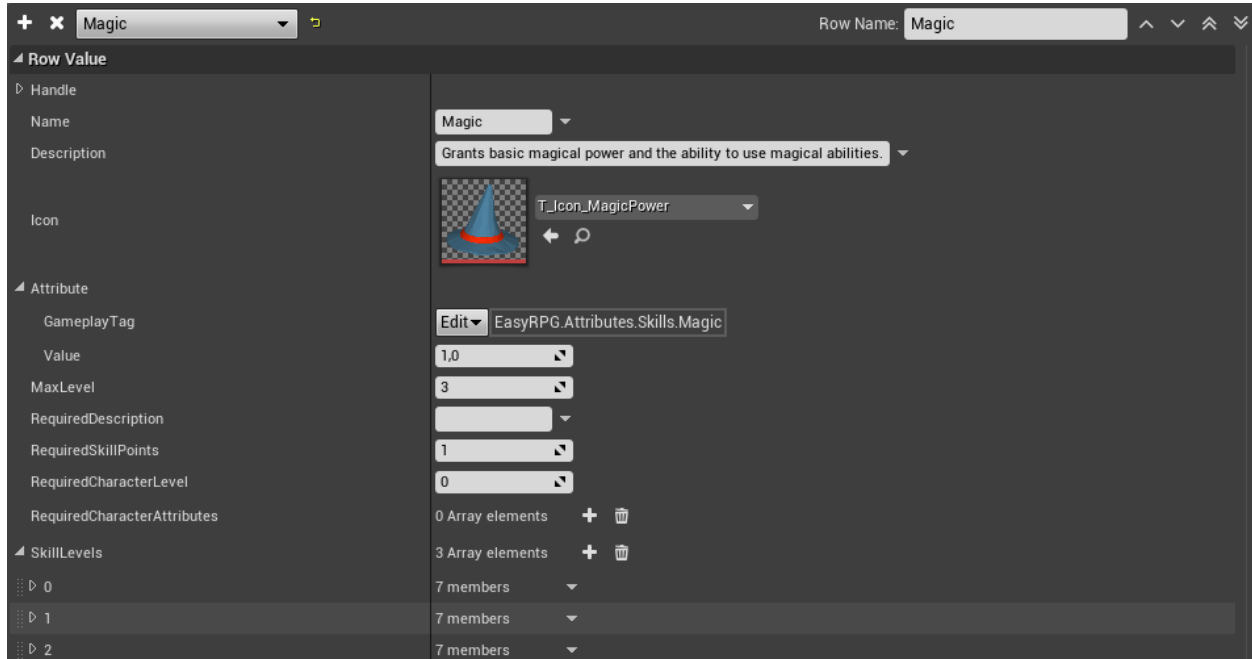
The **UI_SkillSlot** widget is used to display a skill that can be learned in the skill tree.

The **UI_SkillTree** widget is used to display available and learned skills in the skill tree.

The **UI_SkillToolTip** widget is used to display tooltip information about skill when you hover mouse on the skill slot widget.

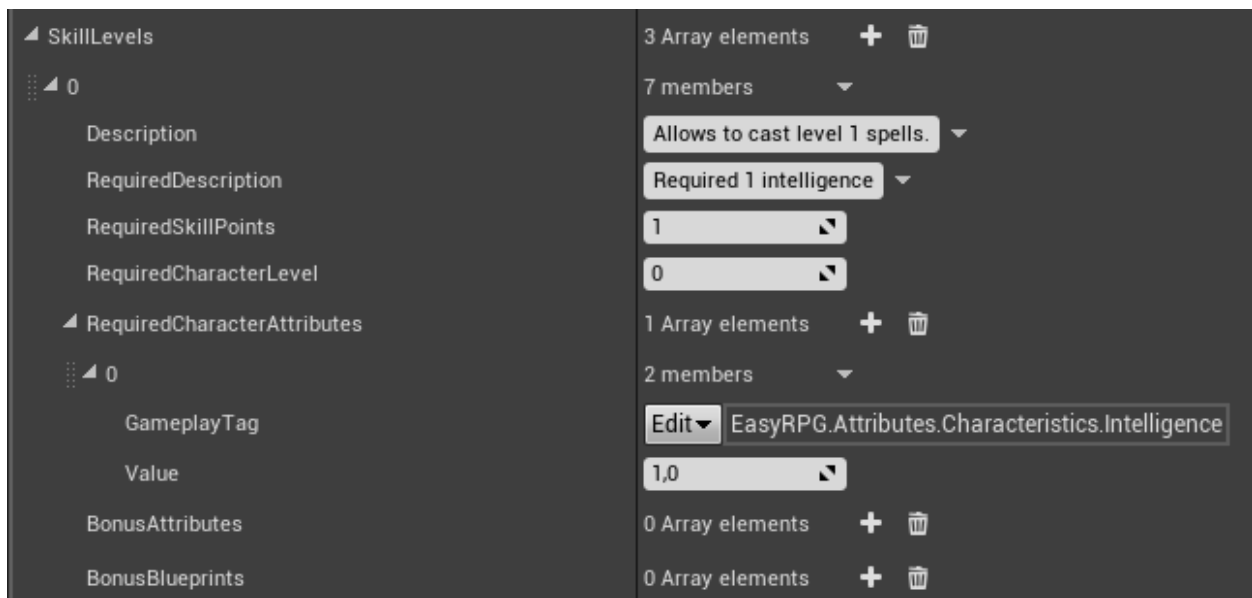
3.12.2. Working with skills

The settings of skills can be configured in the **DT_Skills**.

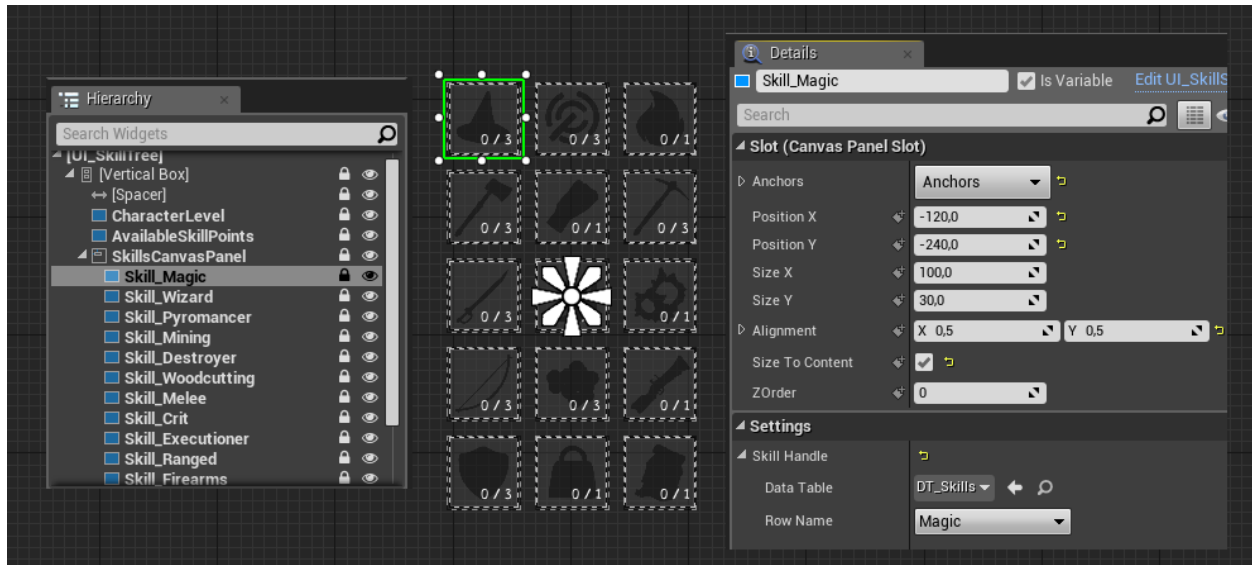


The main attribute of skill should be added in the **DT_GameplayTags_Attributes_Skills** data table and then used in settings of the skill.

Settings for certain skill levels also can be configured in the data table.

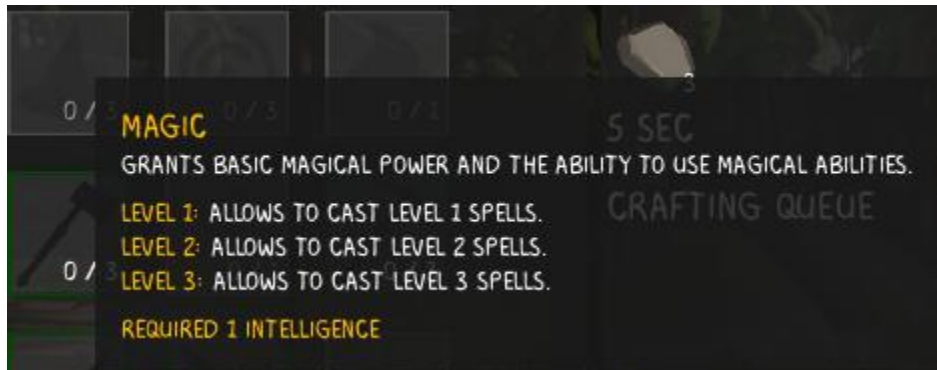


The skill tree can be configured in the **UI_SkillTree** widget.



Add the new **UI_SkillSlot** to the widget, set its offset and select the skill handle from the **DT_Skills** in details of the skill slot widget.

The design of the skill tooltip can be configured in the **UI_SkillToolTip** widget.



3.13. Spell book system

3.13.1. Description

The spell / ability book system is based on abstract ability items and the skills system. It allows you to take abstract ability items from UI and use it in the players hotbar as usable items. This ability item slots can require a certain level of the character or require certain attributes for using.

The abstract ability items can be added and configured in the **DT_Items_Abilities** data table.

The ability items should have the **abstract** item tag.



The ability items also should have its own ability which is selected in the **Handles** variable or in the **UseAbilityClass** variable.



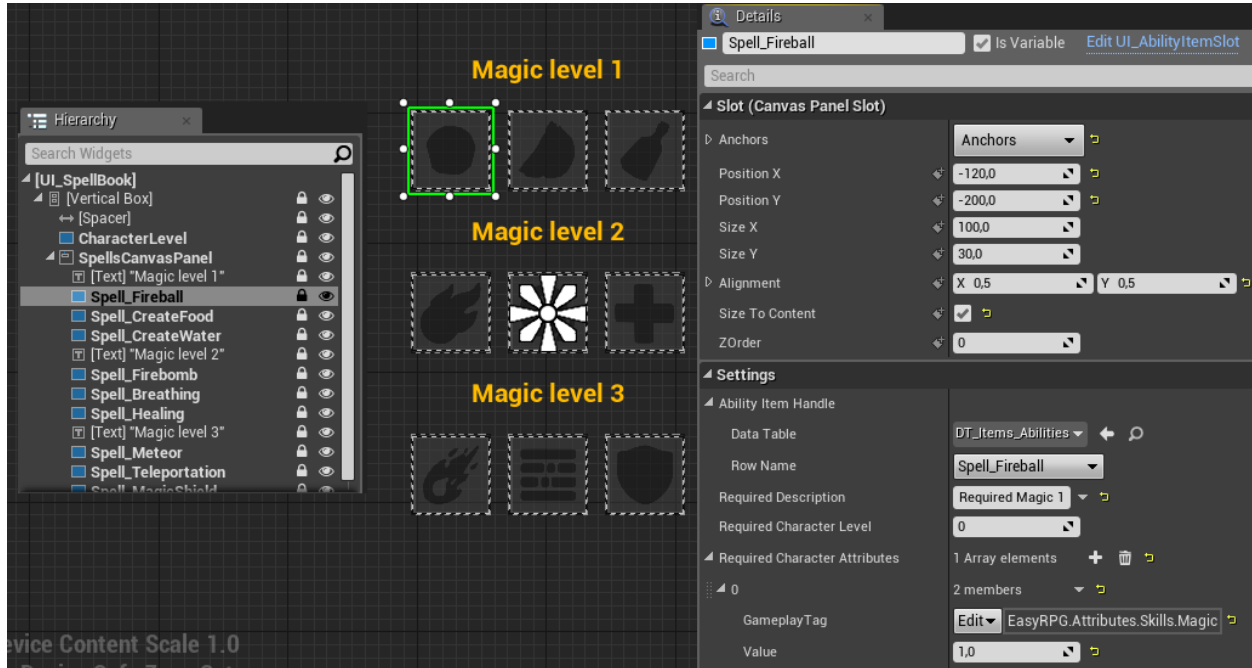
The **UI_AbilityItemSlot** widget is used for displaying spell / ability in the spell book.

The **UI_SpellBook** widget is used to display available spells in the spell book.

The **UI_AbilityItemToolTip** widget is used to display tooltip information about spell when you hover mouse on the ability item slot widget.

3.13.2. Working with spell book

The spell book can be configured in the **UI_SpellBook** widget.



Add the new **UI_AbilityItemSlot** to the widget, set its offset and select the ability item handle from the **DT_Items_Abilities** in details of the skill slot widget. The required level of the character and required attributes also can be configured in the details of the ability item slot widget.

The design of the skill tooltip can be configured in the **UI_AbilityItemToolTip** widget.

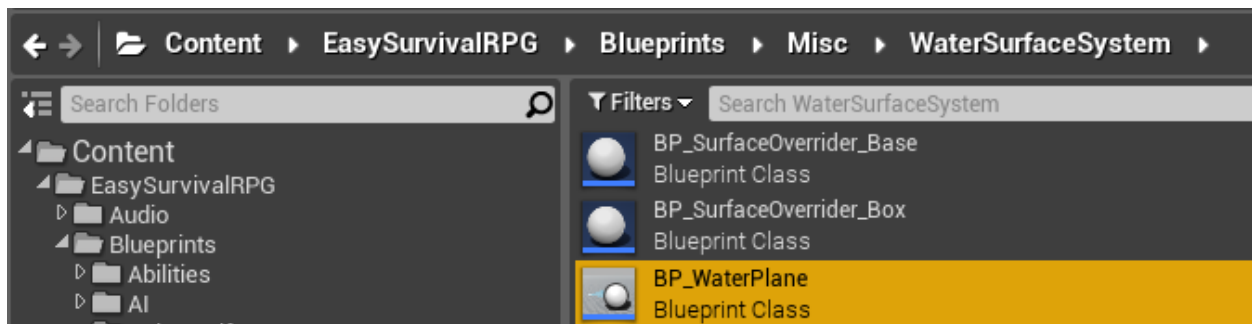


3.14. Swimming system

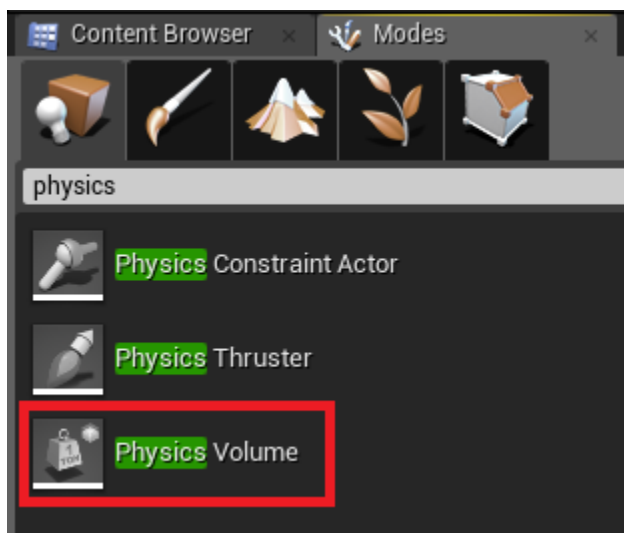
3.14.1. Description

The **ESPRG** project contains a simple swimming system which allows player characters to swim. It is based on default character movement component settings and engine physics.

The **BP_WaterPlane** can be used for placing water on the level.

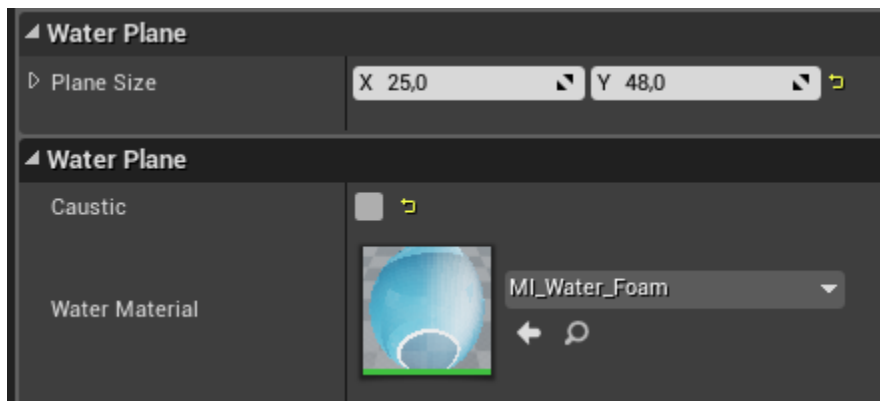


Also the **PhysicsVolume** should be placed under water for working. It will affect characters which have configured movement components. Can swim should be enabled.

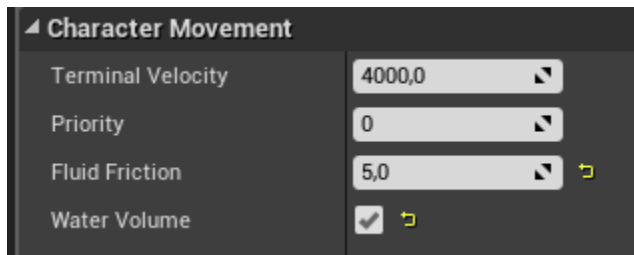


3.14.2. Working with swimming system

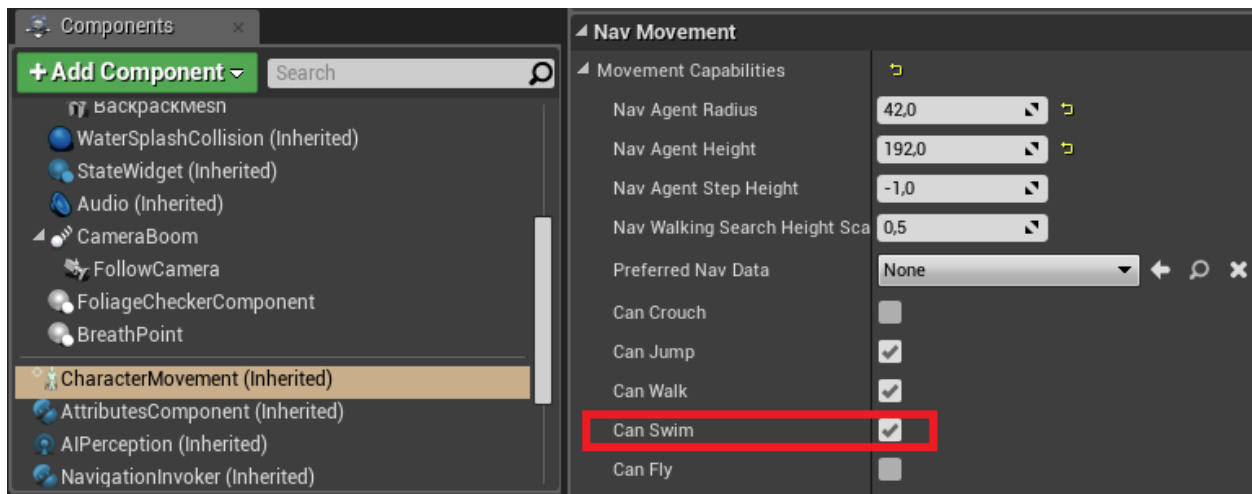
First you need to place the **BP_WaterPlane** blueprint on the level and configure it. You can change plane sizes as you want. Here can be enabled caustics decal. Also the water material can be selected here.



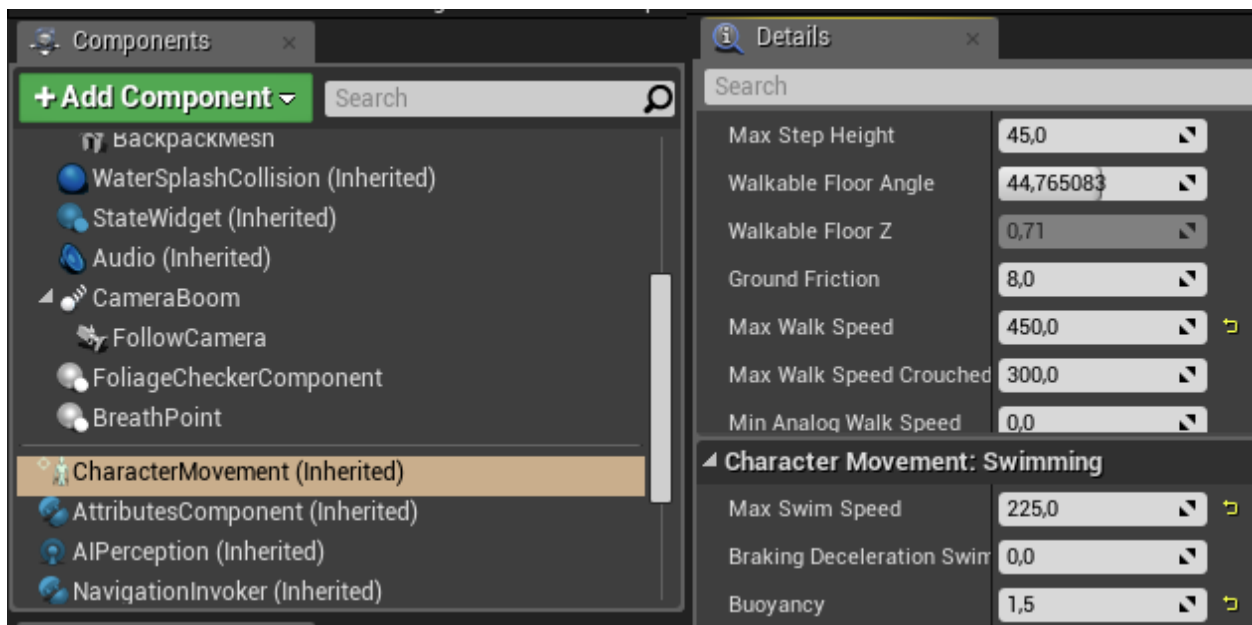
After that you need to place the **PhysicsVolume** actor under the water plane actor. Set the sizes and configure it as water volume. The water volume checkbox should be checked for affecting characters. Fluid friction also can be configtred. The higher value, the faster the character stops movement. The optimal value is 5.



The character also should be configured for swimming. The can swim variable in the **CharacterMovement** component should be enabled.



Here you also can configure max swim speed and buoyancy variables.



4. Characters

4.1. Base character class

4.1.1. Description

BP_Character_Base is the main character class that inherits from the standard **Character** class. It is the parent class of all character classes in the project.

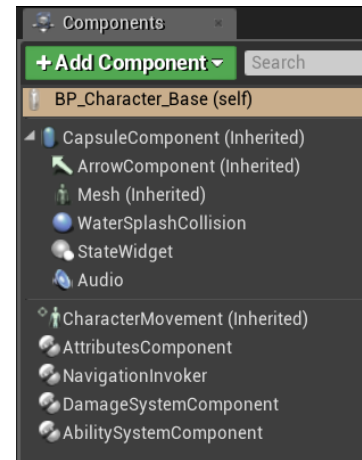
The base character class contains base character functional:

- health state
- energy state
- mana state
- character attributes
- interaction states
- dynamic change speed
- fall damage
- fall into water
- take damage and block hits
- character death
- character attacks
- start dialogue
- threat detection
- behavior tree states

BP_Character_Base class implements **BPI_Character** and **BPI_InteractionObject** interfaces. The **BPI_Character** interface contains functions for working with character states and interactions. **BPI_InteractionObject** interface is used for interaction with the character, to start a dialogue.

4.1.2. Components

In addition to the main components from the **Character** class, the **BP_Character_Base** class contains the **WaterSplashCollision** and the **StateWidget** scene components and also contains the **AttributesComponent**, **NavigationInvoker**, **DamageSystemComponent** and the **AbilitySystemComponent** actor components.



The **WaterSplashCollision** component is used to detect when the character falls into the water and play the appropriate sound.

The **StateWidget** component is used to display the character's name as well as his health in the game.

The **Audio** component is used for playing dialogue sounds.

The **AttributesComponent** is a special component for working with character attributes, such as maximum health, attack range, melee damage, etc.

The **NavigationInvoker** component is updating nearby terrain for an artificial intelligence navigation system.

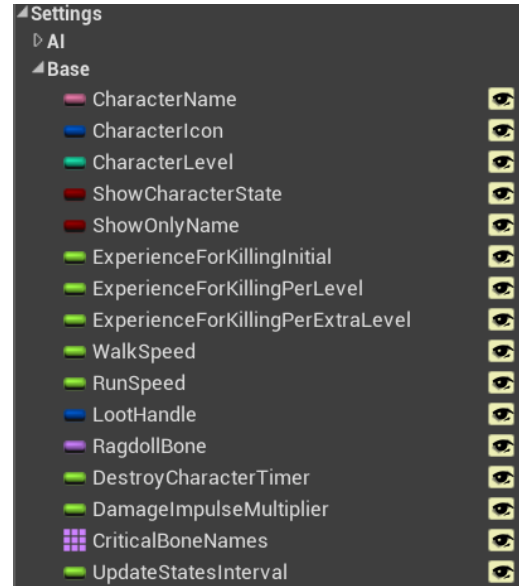
The **DamageSystemComponent** is used for taking or dealing damage.

The **AbilitySystemComponent** is used for casting abilities, checking ability conditions and for applying status effects to the character.

4.1.3. Variables

4.1.3.1. Base settings variables

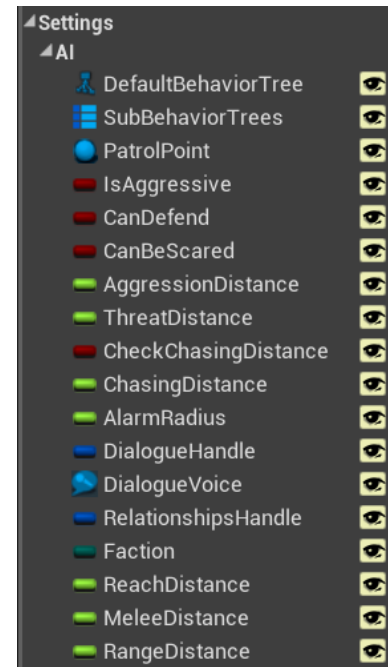
- **CharacterName** - a name of the character that is used in various widgets.
- **CharacterIcon** - an icon of the character that is used in various widgets.
- **CharacterLevel** - initial level of the character that is used in the init character function. It is also used for experience calculation.
- **ShowCharacterState** - show or hide the character state widget.
- **ShowOnlyName** - the character state widget shows only the character name.
- **ExperienceForKillingInitial** - initial amount of experience that is given for killing the character.
- **ExperienceForKillingPerLevel** - additional amount of experience per each character level that is given for killing the character.
- **ExperienceForKillingPerExtraLevel** - additional amount of experience per each extra character level (is the difference between current and initial character levels) that is given for killing the character.
- **WalkSpeed** - the character speed in the walk movement mode.
- **RunSpeed** - the character speed in the run movement mode.
- **LootHandle** - the data row handle of the loot that drops when the character dies.
- **RagdollBone** - the pelvis skeleton bone name to enable skeleton physics.
- **DestroyCharacterTimer** - the time after which the character will be removed from the game after death.
- **DamageImpulseMultiplier** - the impulse multiplier depending on damage taken.



-
- **CriticalBoneNames** - a list of bones that are critical for hits. Hits for such bones multiply incoming damage by default.
 - **UpdateStatesInterval** - an interval between calls of the update states function. If interval is less than or equal to zero then the update states function calls every frame.

4.1.3.2. AI settings variables

- **DefaultBehaviorTree** - the default behavior tree which will be runned for the character.
- **SubBehaviorTrees** - a mapped list of sub behavior trees. Dynamical parts of the character behavior can be changed runtime.
- **PatrolPoint** - the current patrol point.
- **IsAggressive** - an indicator that the character is aggressive and will attack any threat.
- **CanDefend** - an indicator that the character will defend and attack enemies, if they attack the character.
- **CanBeScared** - an indicator that the character can be scared and start running away from the threat.
- **AggressionDistance** - the distance at which the character will chase the threat.
- **ThreatDistance** - the distance for checking threats near the character.
- **CheckChasingDistance** - determines that the character should check distance to guard location while chasing the target.
- **ChasingDistance** - the distance that the character will chase the target.
- **AlarmRadius** - the distance in which the alarm will be activated for friendly characters.
- **DialogueHandle** - the data row handle of the character dialogue data.
- **DialogueVoice** - the character voice class.
- **RelationshipsHandle** - the data row handle of the character relationship with other factions.
- **Faction** - the faction of the character.
- **ReachDistance** - a distance that is used for the move to tasks in the behavior tree.
- **MeleeDistance** - a distance that is used for the move to target in a melee combat behavior tree.



-
- **RangeDistance** - a distance that is used for the move to target in a ranged combat behavior tree.

4.1.3.3. Health state variables

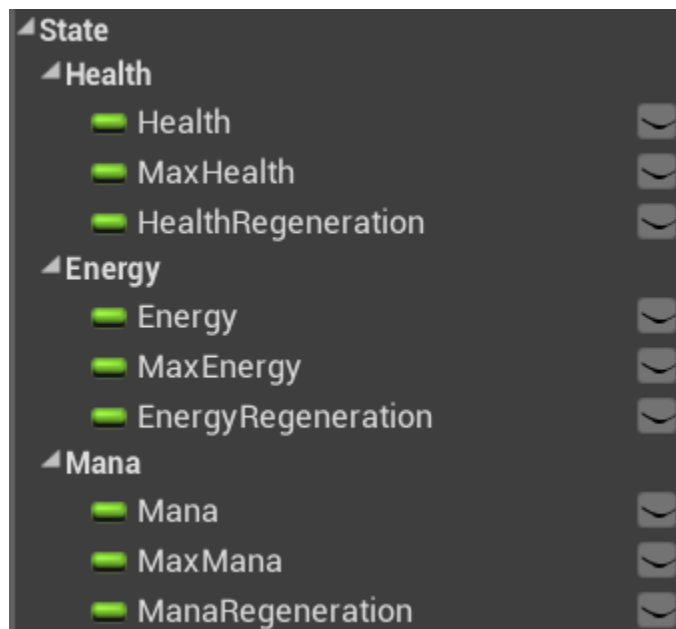
- **Health** - the current value of health.
- **MaxHealth** - the maximum value of health.
- **HealthRegeneration** - the health recovery rate.

4.1.3.4. Energy state variables

- **Energy** - the current value of energy.
- **MaxEnergy** - the maximum value of the energy.
- **EnergyRegeneration** - the energy recovery rate.

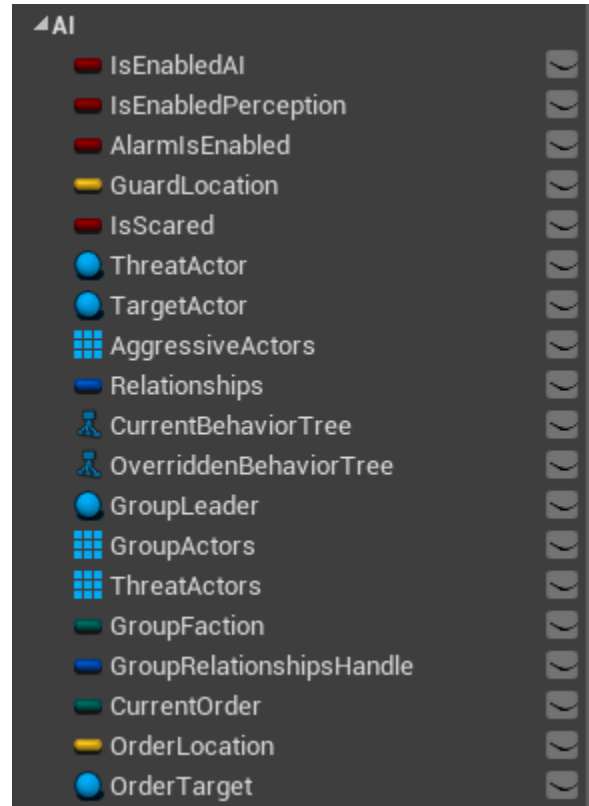
4.1.3.5. Mana state variables

- **Mana** - the current value of mana.
- **MaxMana** - the maximum value of mana.
- **ManaRegeneration** - the mana recovery rate.



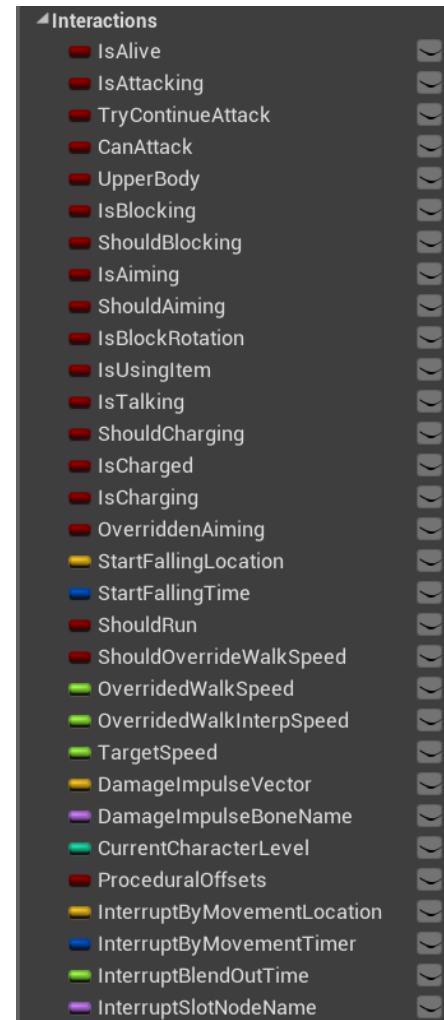
4.1.3.6. AI state variables

- **IsEnabledAI** - an indicator that the character's AI is enabled.
- **IsEnabledPerception** - if true, the perception system will identify targets.
- **AlarmIsEnabled** - if true, the character will activate the alarm for nearby friendly characters or group companions.
- **GuardLocation** - point guarded by the character.
- **IsScared** - an indicator that the character is scared.
- **ThreatActor** - the current threat actor of the character.
- **TargetActor** - the current chasing target of the character.
- **AggressiveActors** - aggressive actors which deal damage to the character.
- **Relationships** - the character's relationships with other factions.
- **CurrentBehaviorTree** - the current runned behavior tree.
- **OverriddenBehaviorTree** - the behavior tree which can override the default behavior tree.
- **GroupActors** - friendly actors which are in one group with the character.
- **ThreatActors** - actors which can be a threat for the character.
- **GroupFaction** - the faction of the group.
- **GroupRelationshipsHandle** - the group relationships with other factions.
- **CurrentOrder** - the current order of the character.
- **OrderLocation** - the target point of current order.
- **OrderTarget** - the target actor of current order.



4.1.3.7. Interactions state variables

- **IsAlive** - an indicator that the character is alive.
- **IsAttacking** - an indicator that the character is attacking.
- **TryContinueAttack** - an indicator that the character should continue to attack.
- **CanAttack** - an indicator that the character can attack.
- **UpperBody** - an indicator that the upper body animation is playing.
- **IsBlocking** - an indicator that the character is blocking.
- **ShouldBlocking** - an indicator that the character should block.
- **IsAiming** - an indicator that the character is aiming.
- **ShouldAiming** - an indicator that the character should aim.
- **IsBlockRotation** - an indicator that the character should not rotate to the controller rotation.
- **IsUsingItem** - an indication that the character is using the item.
- **IsTalking** - an indicator that the character is in dialogue.
- **IsCharged** - an indicator that the charged interaction of the character is ready.
- **IsCharging** - an indicator that the character is charging interaction.
- **OverriddenAiming** - the overridden value for **IsAiming** variable.
- **StartFallingLocation** - the location at which the character began to free fall.
- **StartFallingTime** - timestamp at which the character began to free fall.
- **ShouldRun** - an indicator that the character should run.
- **ShouldOverrideWalkSpeed** - an indicator that it is necessary to dynamically override the speed of the character's movement.
- **OverriddenWalkSpeed** - overridden value of the character's movement speed.

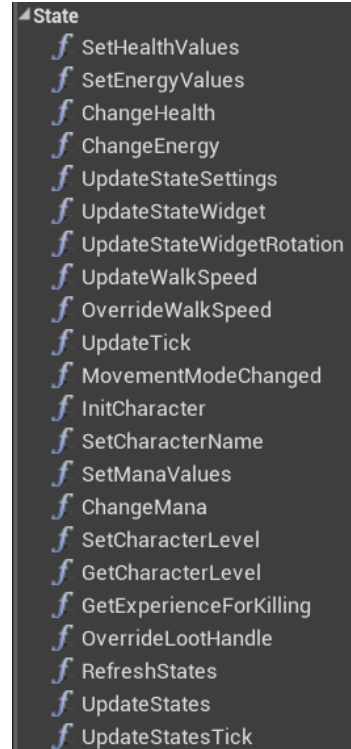


-
- **OverriddenWalkInterpSpeed** - the speed of interpolation to the overridden value of the speed of the character's movement.
 - **TargetSpeed** - the resulting movement speed of the character.
 - **DamageImpulseVector** - the current accumulated damage impulse vector value. If the character is dead, then this impulse will be released on the next tick.
 - **DamageImpulseBoneName** - the current bone name for the accumulated impulse.
 - **CurrentCharacterLevel** - current character level that is used for attribute scaling and experience calculation.
 - **ProceduralOffsets** - a state that determines that the character should use procedural offsets in animation blueprint.
 - **InterruptByMovementLocation** - a location that is used to check that character moves.
 - **InterruptByMovementTimer** - a handle to timer that checks that character moves.
 - **InterruptBlendOutTime** - a time that is used for animation interruption.
 - **InterruptSlotNodeName** - a slot name that determines which animation group should be interrupted when the character moves.

4.1.4. Functions

4.1.4.1. State functions

- **SetHealthValues** - set the values of the maximum health and the speed of its recovery.
- **SetEnergyValues** - set the values of the maximum energy and the speed of its recovery.
- **ChangeHealth** - change the current value of the health.
- **Change Energy** - change the current value of the energy.
- **UpdateStateSettings** - update character state settings, such as maximum health and energy after changing attributes.
- **UpdateStateWidget** - update the character state widget.
- **UpdateStateWidgetRotation** - update the rotation of the character state widget relative to the local player.
- **UpdateWalkSpeed** - update the current character movement speed.
- **OverrideWalkSpeed** - override the character movement speed.
- **UpdateTick** - main update by tick function, which executes all update functions.
- **MovementModeChanged** - executed when the character's movement mode changes, for example, the character starts falling.
- **InitCharacter** - initialize all character components.
- **SetCharacterName** - set the character name.
- **SetManaValues** - set the values of the maximum mana and the speed of its recovery.
- **ChangeMana** - change the current value of the mana.
- **SetCharacterLevel** - set current character level and update level attributes.
- **GetCharacterLevel** - returns current character level.
- **GetExperienceForKilling** - returns experience that is given for killing the character.

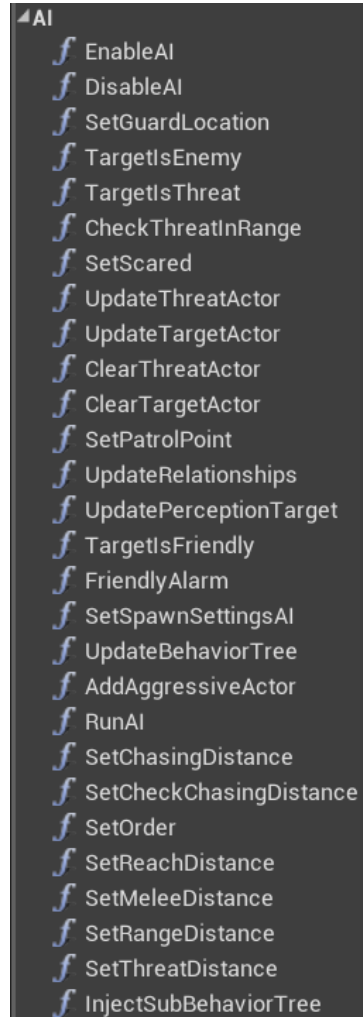


```
State
f SetHealthValues
f SetEnergyValues
f ChangeHealth
f ChangeEnergy
f UpdateStateSettings
f UpdateStateWidget
f UpdateStateWidgetRotation
f UpdateWalkSpeed
f OverrideWalkSpeed
f UpdateTick
f MovementModeChanged
f InitCharacter
f SetCharacterName
f SetManaValues
f ChangeMana
f SetCharacterLevel
f GetCharacterLevel
f GetExperienceForKilling
f OverrideLootHandle
f RefreshStates
f UpdateStates
f UpdateStatesTick
```


-
- **OverrideLootHandle** - override character loot handle.
 - **RefreshStates** - calculate character attributes and set max values for character states.
 - **UpdateStates** - update character states.
 - **UpdateStatesTick** - update states on tick.

4.1.4.2. AI functions

- **EnableAI** - enable the character AI.
- **DisableAI** - disable the character AI.
- **SetGuardLocation** - set the point guarded by the character.
- **TargetIsEnemy** - checks if the target actor is an enemy to the character.
- **TargetIsThreat** - checks if the target actor is a threat to the character.
- **CheckThreatInRange** - checks the threats to the character in a certain radius.
- **SetScared** - set the scared state of the character.
- **UpdateThreatActor** - update threat actor of the character.
- **UpdateTargetActor** - update target actor of the character.
- **ClearThreatActor** - clear threat actor of the character.
- **ClearTargetActor** - clear target actor of the character.
- **SetPatrolPoint** - set a new patrol point for the character.
- **UpdateRelationships** - update relationship data from datatable by relationship handle.
- **UpdatePerceptionTarget** - update the detected actor and determine if it is a threat actor or a target actor.
- **TargetIsFriendly** - returns true, if the target actor is friendly to the character.
- **FriendlyAlarm** - call alarm to friendly characters and update their perception target.
- **SetSpawnSettingsAI** - set spawn AI settings for the character.
- **UpdateBehaviorTree** - update behavior tree for AI controller depending on character variables.
- **AddAggressiveActor** - add aggressive actor to the array and update perception target.
- **RunAI** - run character AI and init variables for blackboard.
- **SetChasingDistance** - set chasing distance value for AI behavior.

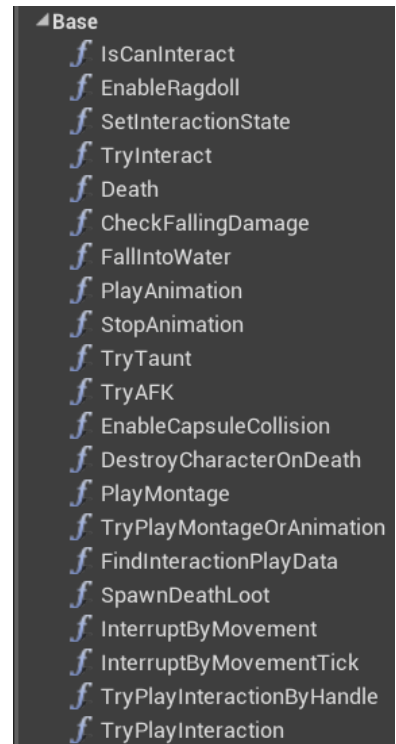


```
AI
f EnableAI
f DisableAI
f SetGuardLocation
f TargetIsEnemy
f TargetIsThreat
f CheckThreatInRange
f SetScared
f UpdateThreatActor
f UpdateTargetActor
f ClearThreatActor
f ClearTargetActor
f SetPatrolPoint
f UpdateRelationships
f UpdatePerceptionTarget
f TargetIsFriendly
f FriendlyAlarm
f SetSpawnSettingsAI
f UpdateBehaviorTree
f AddAggressiveActor
f RunAI
f SetChasingDistance
f SetCheckChasingDistance
f SetOrder
f SetReachDistance
f SetMeleeDistance
f SetRangeDistance
f SetThreatDistance
f InjectSubBehaviorTree
```

-
- **SetCheckChasingDistance** - set check the chasing distance variable for AI behavior.
 - **SetOrder** - set order for AI behavior.
 - **SetReachDistance** - set movement acceptance radius value for AI behavior.
 - **SetMeleeDistance** - set melee attack acceptance radius value for AI behavior.
 - **SetRangeDistance** - set range attack acceptance radius value for AI behavior.
 - **SetThreatDistance** - set threat acceptance radius value for AI behavior.
 - **InjectSubBehaviorTree** - try to inject sub behavior tree logic to main behavior tree logic.

4.1.4.3. Base interaction functions

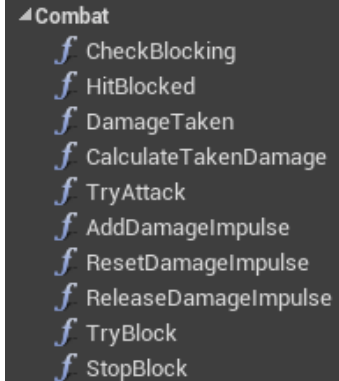
- **IsCanInteract** - check whether the character can interact at the moment.
- **EnableRagdoll** - enable physics simulation for the character's skeletal mesh.
- **SetInteractionState** - set the value of target interaction state.
- **TryInteract** - try to do interaction actions of the character for the target player controller.
- **Death** - do actions when the character dies.
- **CheckFallingDamage** - checks the damage when the character falls.
- **FallIntoWater** - do actions when the character falls into the water.
- **PlayAnimation** - play target animation as montage.
- **StopAnimation** - stop the current animation montage.
- **TryTaunt** - try to do taunt actions of the character.
- **TryAFK** - try to do afk actions of the character.
- **EnableCapsuleCollision** - enable or disable capsule collision of the character.
- **DestroyCharacterOnDeath** - remove the character after time when the character dies.
- **PlayMontage** - play montage using montage play data.
- **TryPlayMontageOrAnimation** - try to play the valid montage or animation.
- **FindInteractionPlayData** - try to find interaction play data by interaction handle.
- **SpawnDeathLoot** - spawn death loot container.
- **InterruptByMovement** - start or stop interrupt by movement process. This process interrupts the current animation if the location of the character is changed.
- **InterruptByMovementTick** - check character location by tick and interrupt interaction if location is changed.



-
- **TryPlayInteractionByHandle** - try to play interaction by specific data table row handle.
 - **TryPlayInteraction** - try to play valid montage or animation from interaction play data.

4.1.4.4. Combat interactions functions

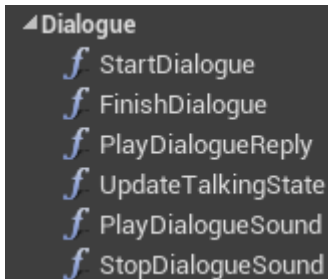
- **CheckBlocking** - checks if the character can block the attack.
- **HitBlocked** - do actions when the character blocks an attack.
- **DamageTaken** - do actions when the character takes damage.
- **CalculateTakenDamage** - calculates the damage taken by the character when hitting him.
- **TryAttack** - try to do attack actions of the character.
- **AddDamageImpulse** - add damage impulse and release it if the character is dead.
- **ResetDamageImpulse** - clear result damage impulse value.
- **ReleaseDamageImpulse** - release damage impulse for the character.
- **TryBlock** - try to start blocking.
- **StopBlock** - stop blocking.



```
Combat
  f CheckBlocking
  f HitBlocked
  f DamageTaken
  f CalculateTakenDamage
  f TryAttack
  f AddDamageImpulse
  f ResetDamageImpulse
  f ReleaseDamageImpulse
  f TryBlock
  f StopBlock
```

4.1.4.4. Dialogue interactions functions

- **StartDialogue** - do start dialogue actions of the character.
- **FinishDialogue** - do finish dialogue actions of the character.
- **PlayDialogueReply** - do dialogue reply actions of the character.
- **UpdateTalkingState** - handles the changing of the **IsTalking** variable.
- **PlayDialogueSound** - set and play dialogue sound.
- **StopDialogueSound** - stop playing dialogue sound.



```
Dialogue
  f StartDialogue
  f FinishDialogue
  f PlayDialogueReply
  f UpdateTalkingState
  f PlayDialogueSound
  f StopDialogueSound
```

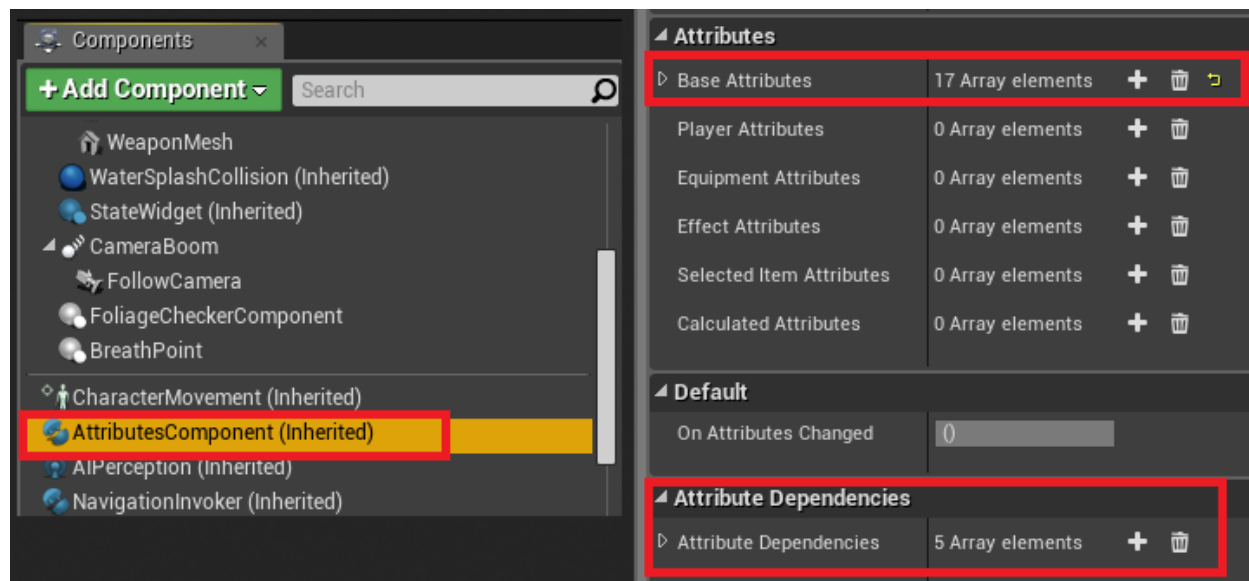
4.2. Character components

4.2.1. AttributesComponent

For the **BP_AttributesComponent**, you can configure the basic attributes that the character will have, as well as the dependencies of the attributes from each other.

You can customize the base attributes by selecting a component in the list of character components, then changing the **BaseAttributes** field in the **Details** window.

The dependencies of attributes on each other can be changed in the **AttributeDependencies** field.



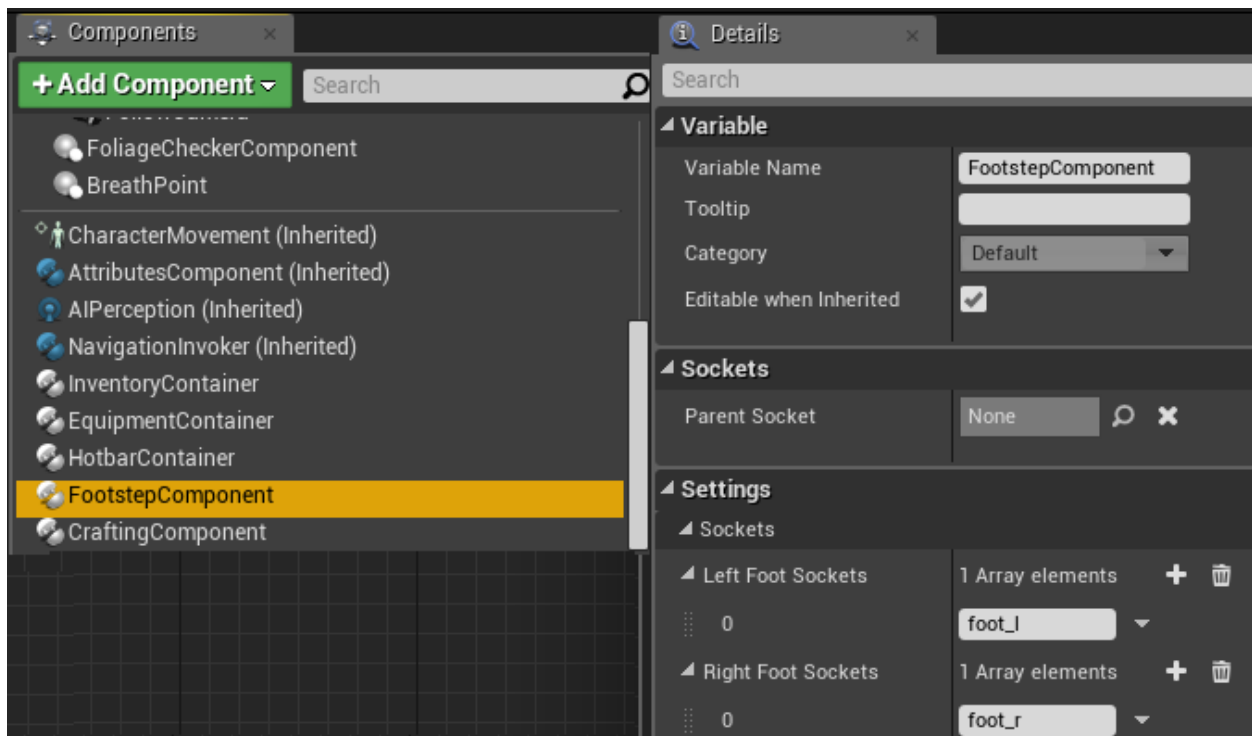
The base attributes which are used in different character systems are in the [Character attributes](#) section.

4.2.2. FootstepComponent

The **BP_FootstepComponent** allows you to play sound and visual effects during the movement of the character, depending on the surface on which he steps. The system also automatically detects if the character is moving through the water and plays the appropriate sounds.

The **BP_FootstepComponent** settings may differ for different characters. Make sure to use the correct foot bones depending on your character's skeleton. You can see the names of the leg bones in the skeleton used by the character.

You can customize the leg bones used by the component by selecting the component in the character components window and changing them in the **Settings / Sockets** section.



The footstep sounds played for each surface are configured in the **Details** of the **BP_FootstepComponent** in the **Settings / Sounds** section.

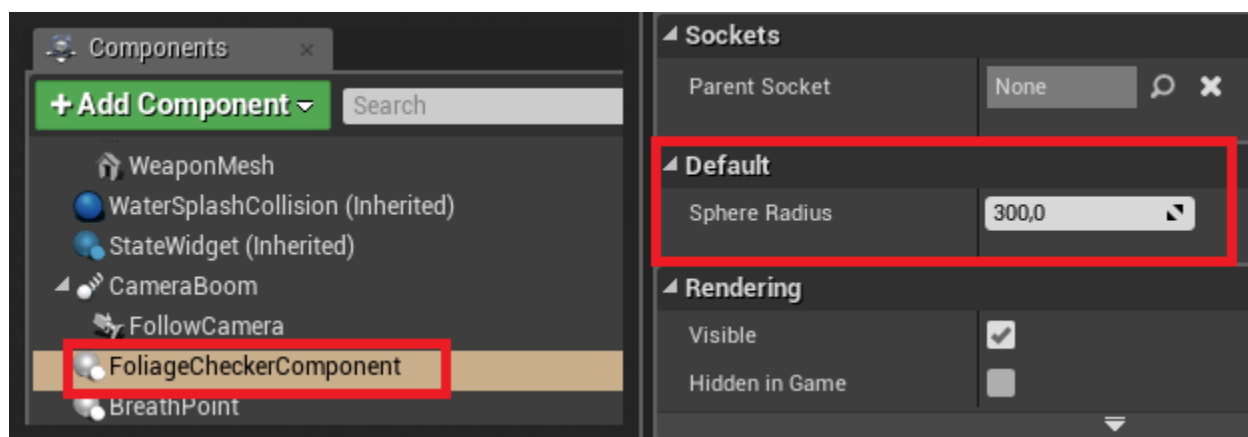
The footprint, the surfaces on which it is left and as well as the visual effects are configured in the **Settings / Effects** section.

4.2.3. FoliageCheckerComponent

The **BP_FoliageCheckerComponent** class finds and automatically replaces instances to actors. It creates an actor with a collision of a certain radius and moves it to the position of the game character by tick.

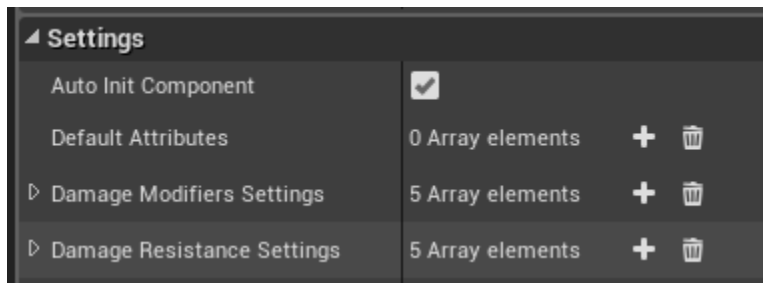
It is part of the system of interchangeable instances, which is described in detail in the corresponding section.

For the **BP_FoliageCheckerComponent**, you can configure the auto-replace radius. This can be done by selecting a component in the list of components and then changing the **SphereRadius** field in the **Details** window.

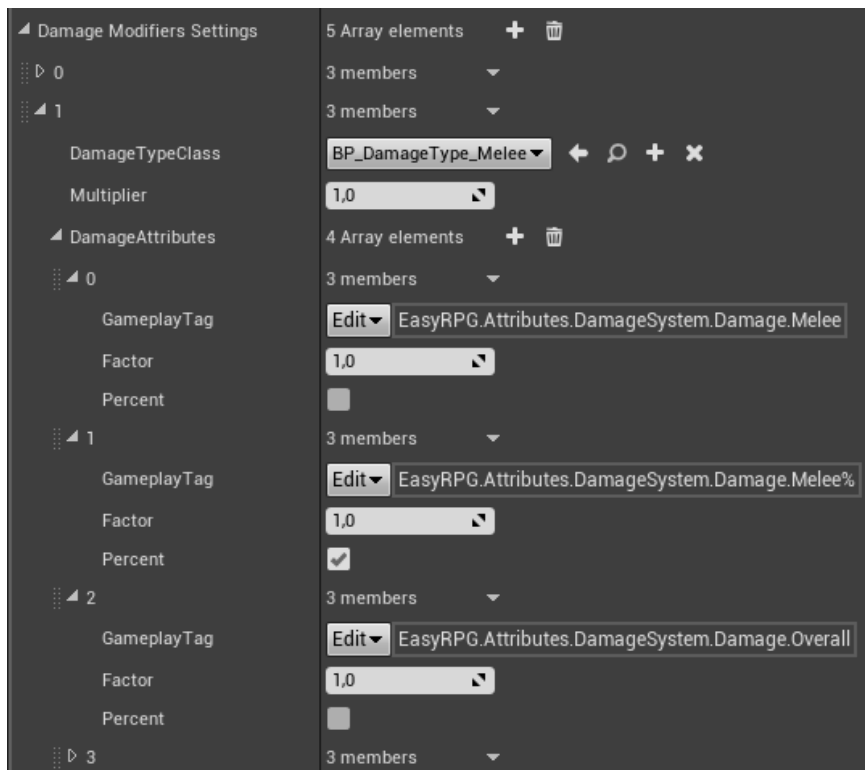


4.2.4. DamageSystemComponent

The **BP_DamageSystemComponent** allows you to use different damage types and damage modifiers in apply damage functions. Damage modifiers and resistances for the certain damage type can be easily configured in the settings section of the damage system component in the actors. By default modifiers and resistance attributes are taken from the attributes component, but it can be set manually.



Each damage type by default has few modifiers and resistance attributes. Attributes which affect on damage type can be added or changed to another. These settings can be set up individually for each actor.

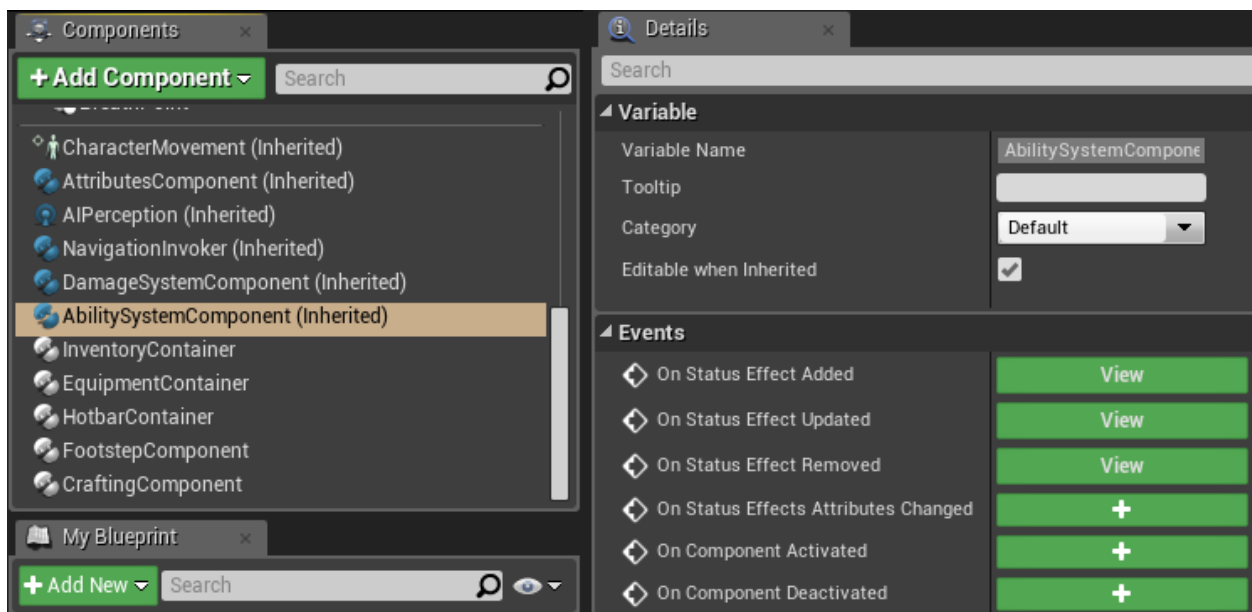


The attributes which are used in the damage system are in the [Damage system attributes](#) section.

4.2.5. AbilitySystemComponent

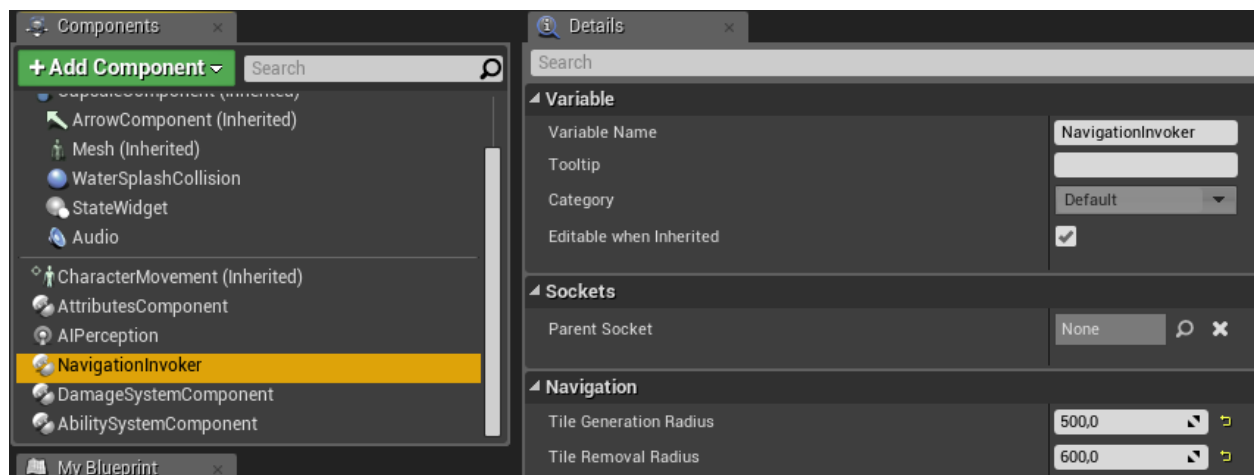
The **BP_AbilitySystemComponent** allows characters to use abilities and checks their cooldowns and other conditions before using. This component should be added to each character blueprint. Also this component allows to apply different status effects to the owning character. The active status effects save during the game save and can be loaded.

The **OnStatusEffectAdded**, the **OnStatusEffectUpdated** and the **OnStatusEffectRemoved** events are used for updating active status effects on the user interface. The **OnStatusEffectsAttributesChanged** event is used for updating status effects attributes in the character's attributes component.



4.2.6. Navigation Invoker

The **Navigation Invoker** component is used in the dynamic navigation system. It updates the nav mesh during the game. The **Nav Mesh Bounds** actor should also be placed in the world.



The **Tile Generation Radius** is used for checking near tiles. If this is not registered in the navigation system it adds it and updates the navigation mesh.

The **Tile Removal Radius** is used for removing unused tile from the navigation system. This value should be little greater than the **Tile Generation Radius**.

4.3. Player character class

4.3.1. Description

Functionality of the player character is in the **BP_Character_Player** class. This class includes all functionality of the **BP_Character_Base** parent class. It also contains logic for working with survival states such as hunger, thirst and oxygen, contains player controls logic, camera system, customization, equipment and weapon systems and many more.

The player character class contains variables, events and functions for working with various systems such as:

- player character controls
- hunger state logic
- thirst state logic
- oxygen state logic
- follow camera system
- character customization logic
- inventory, equipment and hotbar systems
- swimming system
- crouching system
- logic for interactions with melee weapons
- logic for interactions with ranged weapons
- logic for interactions with tools
- logic for interactions with throwable items
- logic for interactions with consumable items
- logic for interactions with spells
- logic for hit reaction and death interactions
- logic for working with custom player interactions

The **BP_Character_Player** class implements the **BPI_PlayerCharacter** interface. This interface provides functions for working with the states of the player character, for working with its customization and for calling functions for interacting with the hotbar.

4.3.2. Components

Includes all components from the **BP_Character_Base** class. These components are marked as Inherited. Their description can be found in the [Base character class](#) section.

Skeletal mesh components **HeadMesh**, **BodyMesh**, **PantsMesh**, **HandsMesh** and **FeetMesh** are used for character customization. These components are also used in the equipment system. The **BackpackMesh** static mesh is also used in the equipment system for the backpack equipment slbt.

The **CameraBoom** and **FollowCamera** components are used for the player camera system.

The **FoliageCheckerComponent** is used in the Replaceable Instance System, which is described in the Replaceable Instance System section.

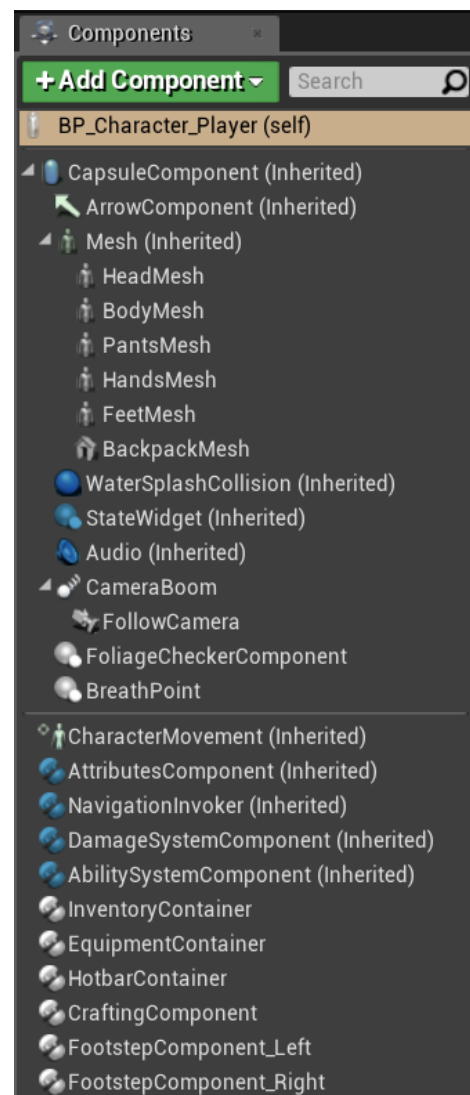
The **BreathPoint** is a scene component that is used in the swimming system. It determines whether the character is underwater or not and whether he should consume oxygen.

The **InventoryContainer** is a character's inventory container.

The **EquipmentContainer** is a character's equipment container.

The **HotbarContainer** is a character's hotbar container.

The **FootstepComponents** allows you to play footsteps that can be configured with specific sounds, effects and decals for different surfaces.

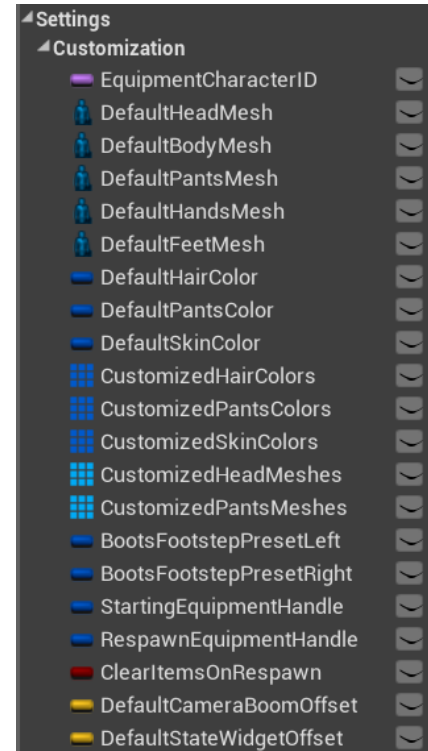


The **CraftingComponent** is a character's crafting component. Allows him to craft various items.

4.3.3. Variables

4.3.3.1. Customization settings variables

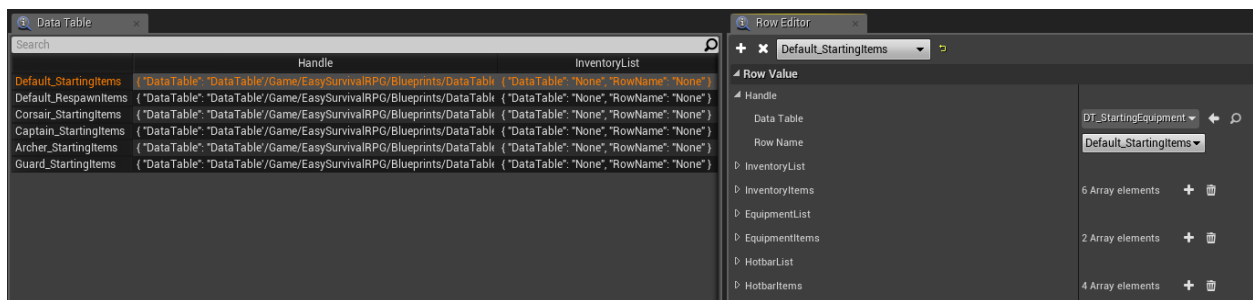
- **EquipmentCharacterID** - character ID to get the character equipment mesh from items.
- **DefaultHeadMesh** - the default skeletal mesh of the character's head for the **HeadMesh** component.
- **DefaultBodyMesh** - the default skeletal mesh of the character's torso for the **BodyMesh** component.
- **DefaultPantsMesh** - the default skeletal mesh of the character's pants for the **PantsMesh** component.
- **DefaultHandsMesh** - the default skeletal mesh of the character's hands for the **HandsMesh** component.
- **DefaultFeetMesh** - the default skeletal mesh of the character's hands for the **FeetMesh** component.
- **DefaultHairColor** - the default hair color of the character.
- **DefaultPantsColor** - the default pants color of the character.
- **DefaultSkinColor** - the default skin color of the character.
- **CustomizedHairColors** - a list of possible colors for the character's hair.
- **CustomizedPantsColors** - a list of possible colors for the character's paints.
- **CustomizedSkinColors** - a list of possible colors for the character's skin.
- **CustomizedHeadMeshes** - a list of possible skeletal meshes for the character's head.
- **CustomizedPantsMeshes** - a list of possible skeletal meshes for the character's pants.
- **BootsFootstepPresetLeft** - a handle to footstep configuration preset for left foot.
- **BootsFootstepPresetRight** - a handle to footstep configuration preset for left foot.



- **StartingEquipmentHandle** - a handle to initial equipment for the character. This handle will be used if the **UseStartingEquipmentFromInstance** variable is disabled in the **Game Instance** class.
- **RespawnEquipmentHandle** - a handle to respawn equipment for the character. This handle will be used if the **UseRespawnEquipmentFromInstance** variable is disabled in the **Game Instance** class.
- **ClearItemsOnRespawn** - if true, all items will be dropped when the character dies and cleared after respawn of the player.
- **DefaultCameraBoomOffset** - offset that is used for the camera boom component by default when the character is in standing state.
- **DefaultStateWidgetOffset** - offset that is used for the state widget component by default when the character is in standing state.

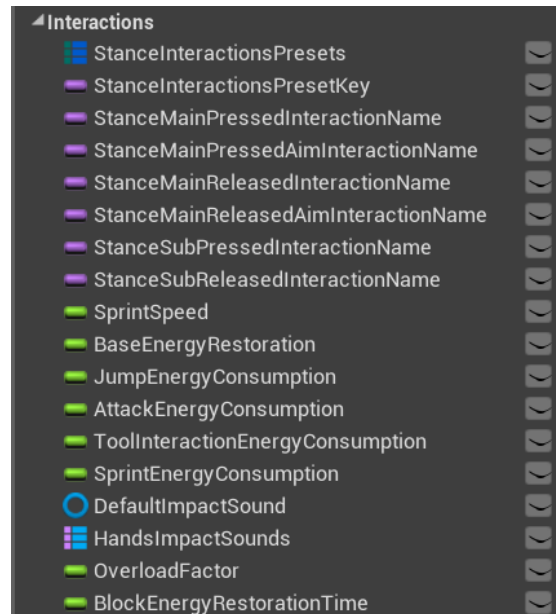
NOTE: The **Construction script** function of the character sets the default meshes of the character depending on mesh component settings.

NOTE: Starting and respawn equipment and items can be set in the **DT_StartingEquipment**. Different playable characters can be set with different equipment.



4.3.3.2. Interactions settings variables

- **StanceInteractionsPresets** - an array of handles, that is used for advanced player interactions and for specific stances.
- **StanceInteractionsPresetKey** - name variable that is used to get a specific handle from the selected item. This handle connects item structure with advanced player interactions structure that allows the user to use custom interactions for inputs.



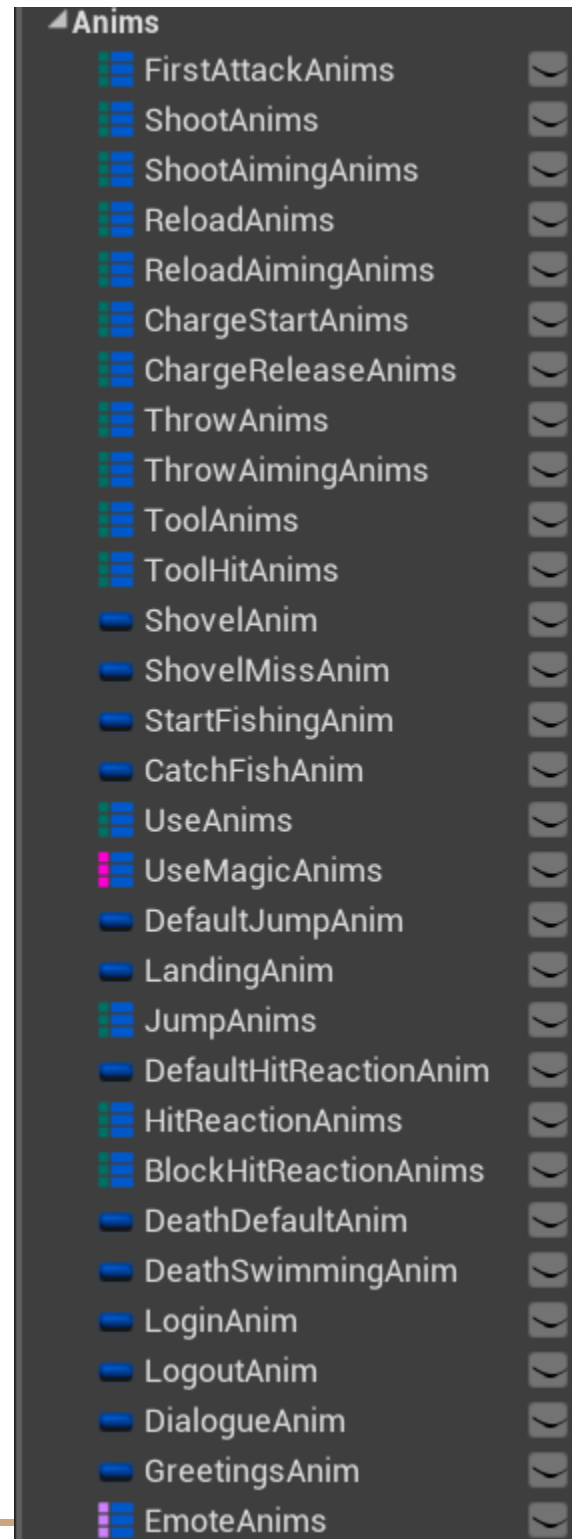
NOTE: Detailed information about stance variables and the custom player interaction system is in the [Advanced player interaction system](#) section.

- **SprintSpeed** - maximum character speed during sprint.
- **BaseEnergyRestoration** - base rate of energy recovery per second.
- **JumpEnergyConsumption** - the amount of energy spent by the character when he jumps.
- **AttackEnergyConsumption** - the amount of energy that a character consumes when he attacks.
- **ToolInteractionEnergyConsumption** - the amount of energy that a character spends on interacting with tools.
- **SprintEnergyConsumption** - the amount of energy consumed by the character during the sprint per second.
- **DefaultImpactSound** - the sound that is played when the character hits a surface without configured physical material.
- **HandsImpactSounds** - the sounds that are played when the character hits surfaces with hands.

- **OverloadFactor** - a value which is used for the weight overload system.
- **BlockEnergyRestorationTime** - a duration during which the energy restoration will be blocked after interactions.

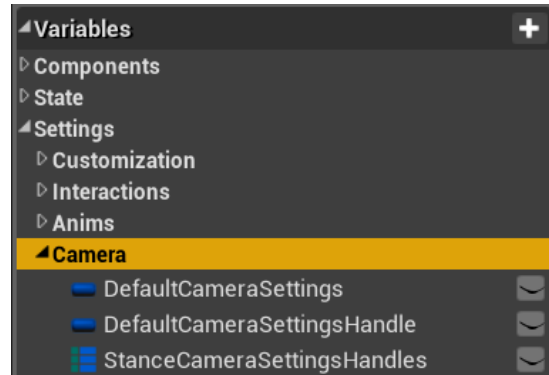
4.3.3.3. Anims settings variables

This category contains variables that are used for different player character interactions. Each variable is a handle to the specific interaction data in the data table. Animations and montages of all player interactions are in the **DT_Animations_Player**.



4.3.3.4. Camera settings variables

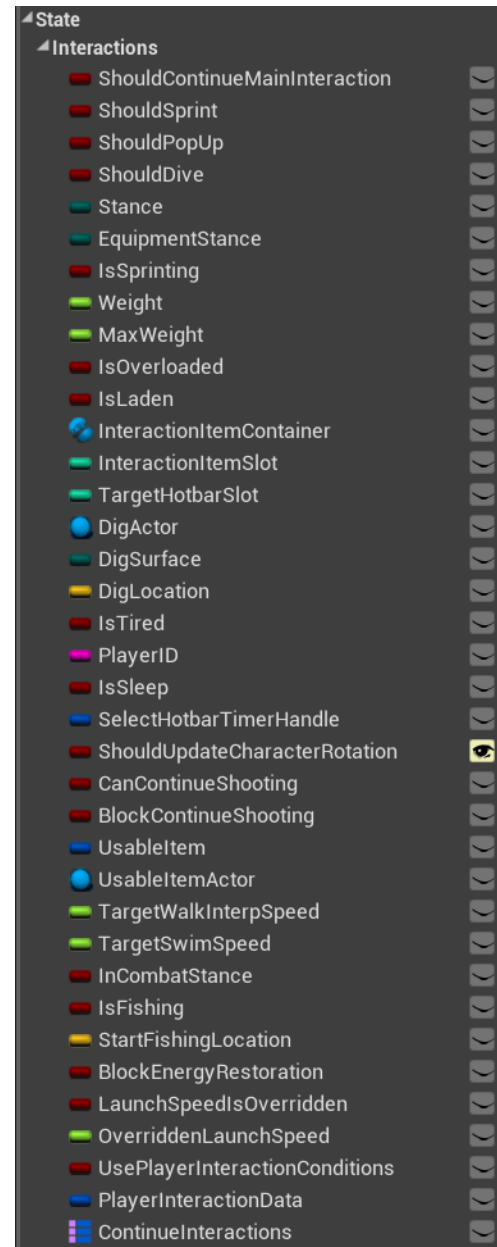
- **DefaultCameraSettings** - a structure that is used as camera settings by default if stance camera settings are not set. Contains camera offset, camera arm length, camera follow lag and player pitch limits for common and aiming states.
- **DefaultCameraSettingsHandle** - a data row handle that is used for loading default camera settings from the data table.
- **StanceCameraSettings** - a mapped list of handles that is used for loading specific camera settings from the data table for different stances.



NOTE: The **Construction script** function of the character sets the default camera settings variable depending on camera component properties. It also tries to init settings from the data table.

4.3.3.5. Interactions state variables

- **ShouldContinueMainInteraction** - an indicator that the character should continue the main interaction.
- **ShouldSprint** - an indicator that the character should sprint.
- **ShouldPopUp** - an indicator that the character should pop up.
- **ShouldDive** - an indicator that the character should dive.
- **Stance** - determines the final animation stance of the character.
- **EquipmentStance** - determines the animation stance of the character depending on the character's equipment.
- **IsSprinting** - an indicator that the character is sprinting.
- **Weight** - the current overall weight of items in containers of the character.
- **MaxWeight** - the maximum allowable weight of items that a character can carry in containers.
- **IsOverloaded** - an indicator that the character is overloaded by weight. The character can't move.
- **IsLaden** - an indicator that the character is laden. Movement speed of the character is decreased.
- **InteractionItemContainer** - the reference to the container that is used in current interaction.
- **InteractionItemSlot** - an index of the item slot that is used in current interaction.
- **TargetHotbarSlot** - the target hotbar slot that the character should switch to.

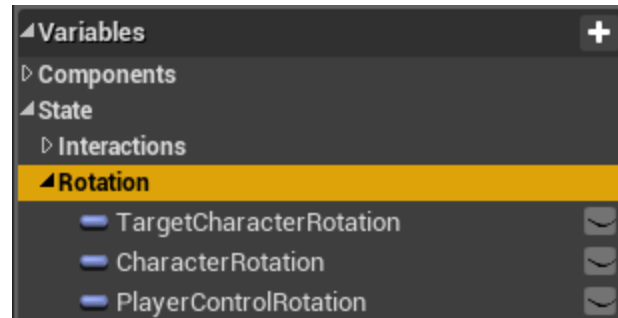


-
- **DigActor** - reference to the actor that will be used in the successful digging function.
 - **DigSurface** - the type of surface that will be used in the successful dig function.
 - **DigLocation** - the dig hit location that will be used in the successful dig function.
 - **IsTired** - an indicator that the character is tired. The character can't sprint and jump.
 - **PlayerID** - an identifier that is associated with a certain player.
 - **IsSleep** - an indicator that the character is sleeping.
 - **SelectedHotbarTimerHandle** - selecting a hotbar slot timer identifier.
 - **ShouldUpdateCharacterRotation** - an indicator that rotates character to the controller rotation.
 - **CanContinueShooting** - an indicator that the character can continue shooting with their current weapon.
 - **BlockContinueShooting** - an indicator that the character can not continue shooting with their current weapon.
 - **UsableItem** - an usable item data that is currently used by the usable item actor of the character.
 - **UsableItemActor** - a reference to the usable item actor that is currently used for the character.
 - **TargetWalkInterpSpeed** - the current interpolation speed value that is used for changing the walking speed of the character.
 - **TargetSwimSpeed** - the current swimming speed value that is used for changing the swimming speed of the character.
 - **InCombatStance** - an indicator that the character is in combat.
 - **IsFishing** - an indicator that the character is fishing.
 - **StartFishingLocation** - a location that is used for checking and stopping fishing, if the character starts moving.
 - **BlockEnergyRestoration** - an indicator that energy restoration is blocked.
 - **LaunchSpeedIsOverridden** - an indicator that launch speed is overridden.
 - **OverriddenLaunchSpeed** - an overridden launch speed value that is used in the launch projectile abilities.

-
- **UsePlayerInteractionConditions** - an indicator that player interaction conditions should be checked.
 - **PlayerInteractionData** - a variable that contains data about player interaction. It includes animation, montage play data and condition flags that should be checked before interaction.
 - **ContinueInteractions** - a list of mapped interactions that is used in the continue interaction function.

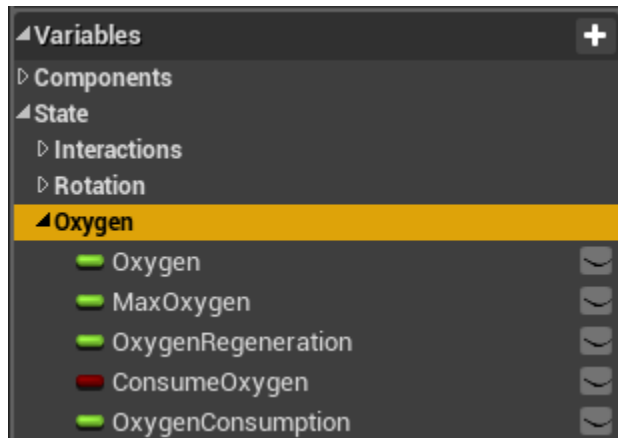
4.3.3.5. Rotation state variables

- **TargetCharacterRotation** - the required character rotation.
- **CharacterRotation** - the current character rotation.
- **PlayerControlRotation** - the current player controller rotation.



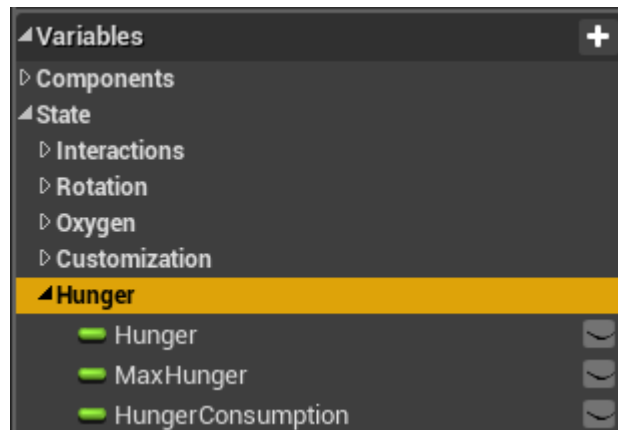
4.3.3.6. Oxygen state variables

- **Oxygen** - the current value of oxygen.
- **MaxOxygen** - the maximum value of oxygen.
- **OxygenRegeneration** - oxygen recovery rate per second.
- **ConsumeOxygen** - an indicator that the character should consume oxygen.
- **OxygenConsumption** - the rate of oxygen consumption under water per second.



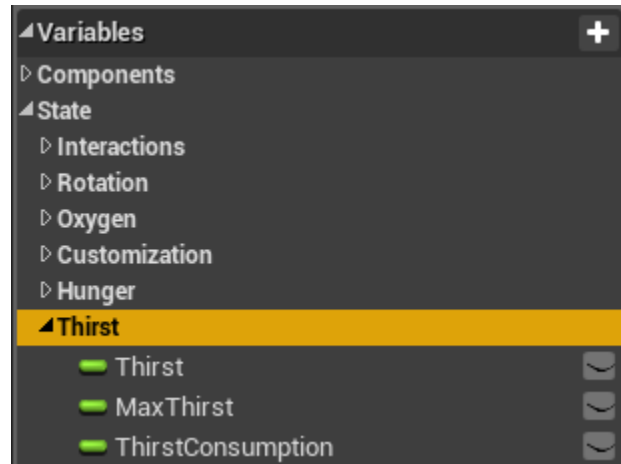
4.3.3.7. Hunger state variables

- **Hunger** - the current value of hunger.
- **MaxHunger** - the maximum value of hunger.
- **HungerConsumption** - the rate of hunger consumption per second.



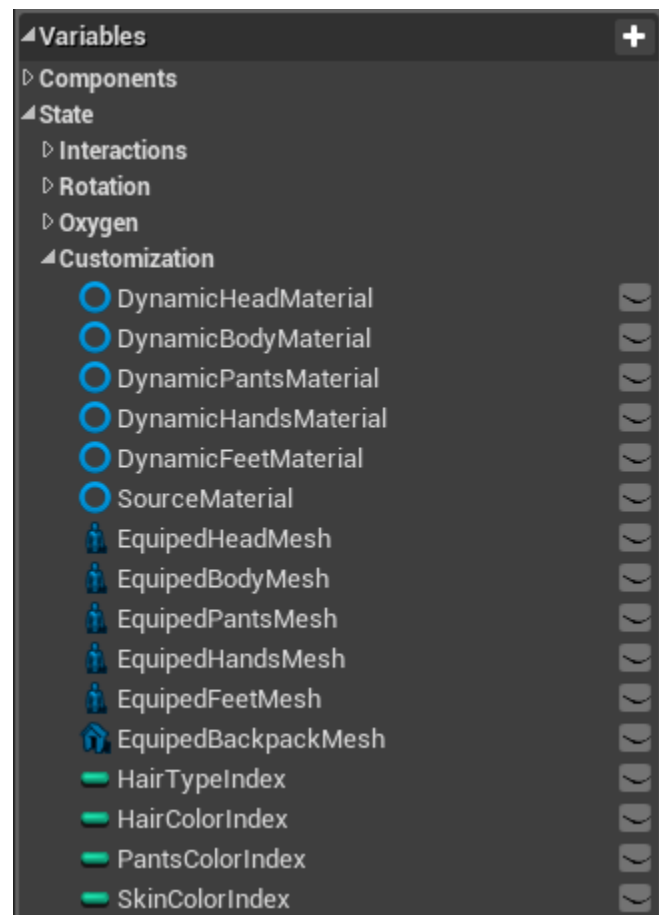
4.3.3.8. Thirst state variables

- **Thirst** - the current value of thirst.
- **MaxThirst** - the maximum value of thirst.
- **ThirstConsumption** - the rate of thirst consumption per second.



4.3.3.9. Customisation state variables

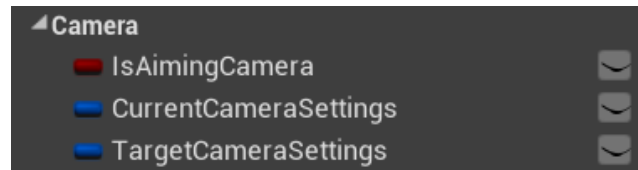
- **DynamicHeadMaterial** - dynamic material for the character's head mesh.
- **DynamicBodyMaterial** - dynamic material for the character's body mesh.
- **DynamicPantsMaterial** - dynamic material for the character's pants mesh.
- **DynamicHandsMaterial** - dynamic material for the character's hands mesh.
- **DynamicFeetMaterial** - dynamic material for the character's feet mesh.
- **SourceMaterial** - the character's source material.
- **EquippedHeadMesh** - the head skeletal mesh in equipment.



- **EquippedBodyMesh** - the equipped body skeletal mesh.
- **EquippedPantsMesh** - the equipped pants skeletal mesh.
- **EquippedHandsMesh** - the equipped hands skeletal mesh.
- **EquippedFeetMesh** - the equipped feet skeletal mesh.
- **EquippedBackpackMesh** - the backpack item static mesh.
- **HairTypeIndex** - the current hair customisation index.
- **HairColorIndex** - the current hair color customisation index.
- **PantsColorIndex** - the current pants color customisation index.
- **SkinColorIndex** - the current skin color customisation index.

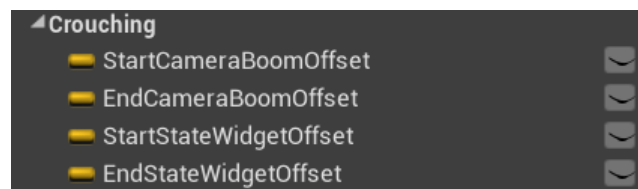
4.3.3.10. Camera state variables

- **IsAimingCamera** - an indicator that the player camera is in aiming mode.
- **CurrentCameraSettings** - a variable that is used for updating camera offset and camera arm length properties in the update camera function.
- **TargetCameraSettings** a variable that is used as target in the interpolation function for current camera settings in the update camera function.



4.3.3.11. Crouching state variables

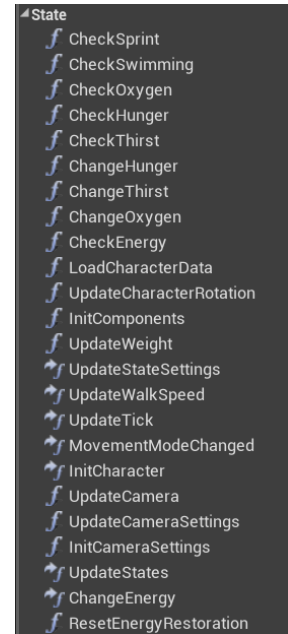
- **StartCameraBoomOffset** - start camera boom relative offset that is used for offset interpolation in crouching transitions.
- **EndCameraBoomOffset** - end camera boom relative offset that is used for offset interpolation in crouching transitions.
- **StartStateWidgetOffset** - start state widget relative offset that is used for offset interpolation in crouching transitions.
- **EndStateWidgetOffset** - end state widget relative offset that is used for offset interpolation in crouching transitions.



4.3.4. Functions

4.3.4.1. State functions

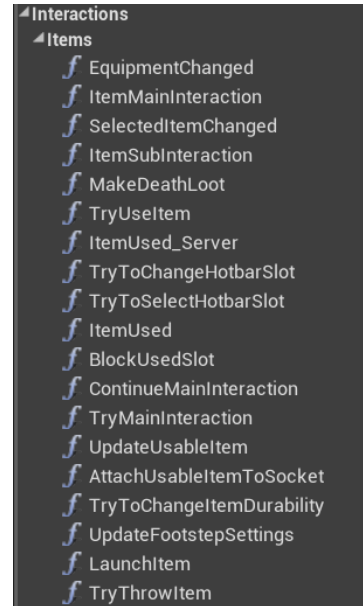
- **CheckSprint** - checks if the character is in sprint mode and updates the dependent variables.
- **CheckSwimming** - checks if the character is in swimming mode and updates the dependent variables.
- **CheckOxygen** - checks the current value of the character's oxygen state and updates the dependent variables. If this value is at zero, the character's health is spent.
- **CheckHunger** - checks the current value of the character's hunger state and updates the dependent variables. If this value is at zero, the character's health is spent.
- **CheckThirst** - checks the current value of the character's thirst state and updates the dependent variables. If this value is at zero, the character's health is spent.
- **ChangeHunger** - change the current value of the hunger state of the character.
- **ChangeThirst** - change the current value of the thirst state of the character.
- **ChangeOxygen** - change the current value of the oxygen state of the character.
- **CheckEnergy** - checks the current energy value and updates the dependent variables.
- **LoadCharacterData** - loads character data and updates character name, customization, attributes and items in containers.
- **UpdateCharacterRotation** - updates the rotation of the character relative to the player's camera.
- **InitComponents** - initializes the components of the player character.
- **UpdateWeight** - updates the overall weight of items in containers.
- **UpdateStateSettings (overridden)** - update additional settings for the states of the player character.
- **UpdateWalkSpeed (overridden)** - update the current movement speed of the character.
- **UpdateTick (overridden)** - execute all check and update functions.
- **MovementModeChanged (overridden)** - called when the character's movement mode changes, for example, the character starts to swim.
- **InitCharacter (overridden)** - initializes all components of the character.
- **UpdateCamera** - update camera variables depending on aiming states.
- **UpdateCameraSettings** - try to update current camera settings by target settings handle or stance.
- **InitCameraSettings** - init default camera settings depending on components details.



-
- **UpdateStates (overridden)** - update states (health, energy, mana, thirst, hunger and oxygen) of the player character.
 - **ChangeEnergy (overridden)** - change energy and block energy restoration.
 - **ResetEnergyRestoration** - resume energy restoration.

4.3.4.2. Items interaction functions

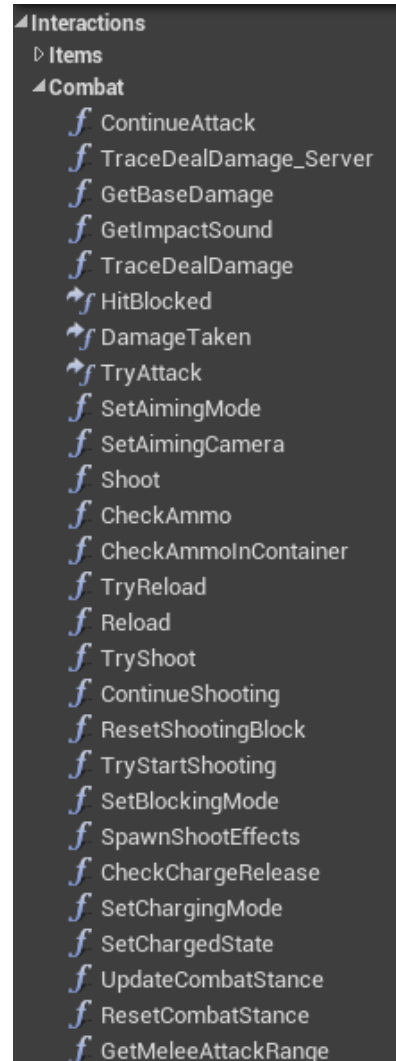
- **EquipmentChanged** - called when a character's equipment item changes.
- **ItemMainInteraction** - try to start main interaction actions for the selected item in the hotbar slot.
- **SelectedItemChanged** - called when an item in the selected hotbar slot changes.
- **ItemSubInteraction** - try to start sub interaction actions for the selected item in the hotbar slot.
- **MakeDeathLoot** - creates a container with the character's loot when the character dies.
- **TryUseItem** - try to start using actions for the selected item in the hotbar slot.
- **ItemUsed_Server** - consumes a used item and executes its ability.
- **TryToChangeHotbarSlot** - try to change the selected hotbar slot.
- **TryToSelectHotbarSlot** - try to select the hotbar slot.
- **ItemUsed** - calls the **ItemUsed_Server** function on the server. Called by the **BP_Notify_ItemUsed** notify at the moment when the item should be used when playing the animation.
- **BlockUsedSlot** - blocks the selected hotbar slot while animation is playing.
- **ContinueMainInteraction** - try to do main item interaction and repeat it after a certain time.
- **TryMainInteraction** - try to do main item interaction depending on current item type.
- **UpdateUsableItem** - update usable item actor for the character. Also update selected item attributes and equipment stance.
- **AttachUsableItemToSocket** - attach usable item actor to target character mesh socket.
- **TryChargeInteraction** - try to do charging interaction with the current selected item.
- **TryToChangeItemDurability** - try to change container item durability.
- **UpdateFootstepSettings** - update footstep settings depending on equipped boots.



-
- **LaunchItem** - try to launch a selected item.
 - **TryThrowItem** - try to throw selected item and play throw item animation.

4.3.4.3. Combat interaction functions

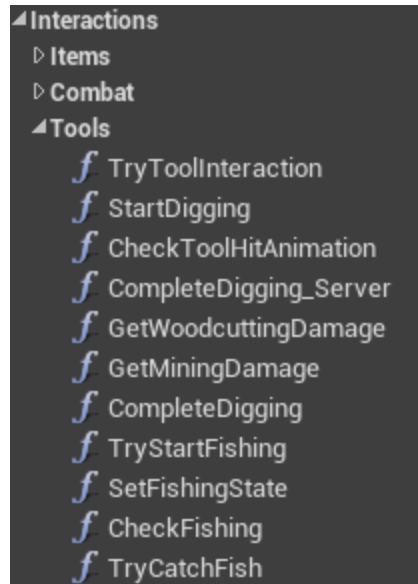
- **ContinueAttack** - try to continue attack.
- **TraceDealDamage_Server** - finds possible targets of attack using traces on the server and deals appropriate damage on them.
- **GetBaseDamage** - returns the base damage that the character can do.
- **GetImpactSound** - returns the impact sound depending on the impact surface type.
- **TraceDealDamage** - calls the **TraceDealDamage_Server** function on the server. Called by the **BP_Notify_TraceDamage** notify at the time the damage is dealt while the attack animation is playing.
- **HitBlocked (overridden)** - plays a blocking hit animation depending on the current animation stance.
- **DamageTaken (overridden)** - plays the animation of taking damage.
- **TryAttack (overridden)** - try to start attacking and play the appropriate animation depending on the current animation stance.
- **SetAimingMode** - set aiming mode and set camera setting for the player.
- **SetAimingCamera** - set aiming camera settings for the player.
- **Shoot** - shoot with the current ranged weapon, consume ammo and use its ability.
- **CheckAmmo** - check ammo in the character containers.
- **CheckAmmoInContainer** - check ammo in the target container.
- **TryReload** - try to reload current rechargeable item using certain reload animation.
- **Reload** - complete reload current rechargeable item, change its charges and consume ammo from inventory.



-
- **TryShoot** - try to shoot with the current ranged weapon using certain shoot animation.
 - **ContinueShooting** - try to continue shooting with the current ranged weapon using certain shoot animation.
 - **ResetShootingBlock** - reset the block shooting variable.
 - **TryStartShooting** - try to start shooting with the current ranged weapon using shoot animation which depends on current stance.
 - **SetBlockingMode** - set blocking mode.
 - **SpawnShootEffects** - spawn shoot particle effects for the current usable item actor.
 - **CheckChargeRelease** - check charging conditions for charge release and play certain release animation.
 - **SetChargingMode** - set charging mode state.
 - **SetChargedState** - set charged state for character and for usable item actor.
 - **UpdateCombatStance** - start combat stance for character. Also start the timer and reset the combat stance when it expires.
 - **ResetCombatStance** - reset combat stance for character.
 - **GetMeleeAttackRange** - returns melee attack range value.

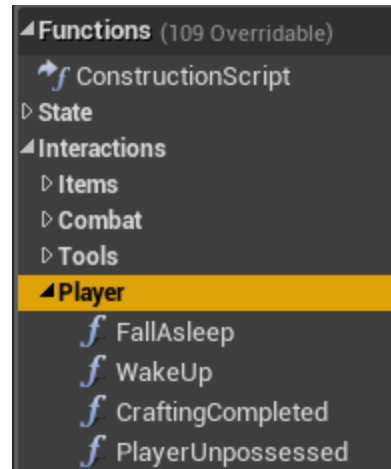
4.3.4.4. Tools interaction functions

- **TryToolInteraction** - try to start interaction with the selected tool in the hotbar and play appropriate animation.
- **StartDigging** - check the surface when the character tries to dig and play the appropriate animation.
- **CheckToolHitAnimation** - check and play the animation of the hit by the tool.
- **CompleteDigging_Server** - successfully completes the digging process on the server.
- **GetWoodcuttingDamage** - returns the damage that the character can do when hitting a tree.
- **GetMiningDamage** - returns the damage that a character can do when hitting an ore vein.
- **CompleteDigging** - calls the **CompleteDigging_Server** function on the server. Called by the **BP_Notify_CompleteDigging** notify when the excavation is complete while playing the animation of digging with a shovel.
- **TryStartFishing** - try to play the start fishing animation. Check required ammo in the inventory.
- **SetFishingState** - set fishing interaction state.
- **CheckFishing** - check that the player moves and reset the fishing state.
- **TryCatchFish** - try to play catch a fish animation.



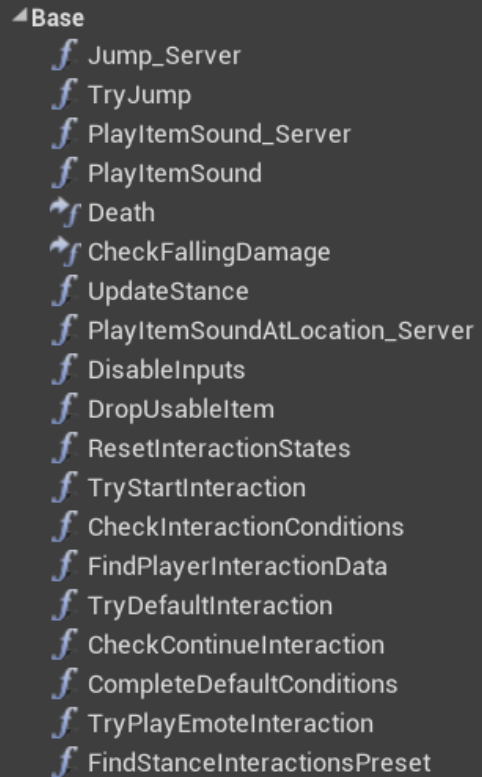
4.3.4.5. Player interaction functions

- **FallAsleep** - plays an animation in which the character goes to sleep. Also disables collision of the character capsule.
- **WakeUp** - plays the wake up animation of the character. Enables the character's collision capsule.
- **CraftingCompleted** - called when a character completes crafting an item.
- **PlayerUnpossessed** - called when the player loses control of the character and leaves the game.



4.3.4.6. Base interaction functions

- **Jump_Server** - play animation of a jump depending on the animation stance and consume energy for this jump.
- **TryJump** - try to jump.
- **PlayItemSound_Server** - play the sound of the item on the server. Multicast it to all clients.
- **PlayItemSound** - calls the **PlayItemSound_Server** function on the server. Called by **BP_Notify_PlayItemSound** notify from animation.
- **Death (overridden)** - do death actions when the player character dies.
- **CheckFallingDamage (overridden)** - checks and deals damage when the character falls. Also plays the appropriate animation.
- **UpdateStance** - checks and updates the final animation stance.
- **PlayItemSoundAtLocation_Server** - play item sound on the server.
- **DisableInputs** - disable character inputs. Reset interaction variables.
- **DropUsableItem** - drop usable item if it is valid.
- **ResetInteractionStates** - reset interaction states.
- **TryStartInteraction** - try to start or continue advanced interaction from a specific interactions preset by specific interaction name.
- **CheckInteractionConditions** - check player interaction conditions.
- **FindPlayerInteractionData** - try to find player interaction data in specific player interaction preset by name.
- **TryDefaultInteraction** - try to play montage or animation for default interaction.
- **CheckContinueInteraction** - try to find continue interaction data.
- **CompleteDefaultConditions** - complete default player interaction conditions.

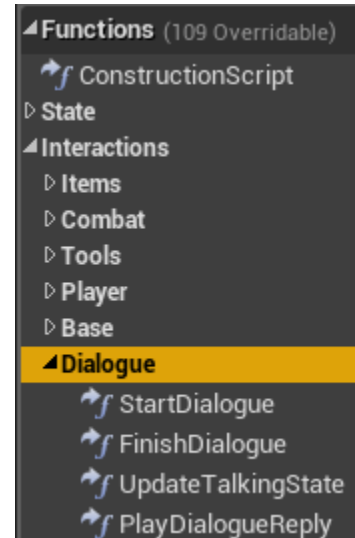


```
Base
  f Jump_Server
  f TryJump
  f PlayItemSound_Server
  f PlayItemSound
  *f Death
  *f CheckFallingDamage
  f UpdateStance
  f PlayItemSoundAtLocation_Server
  f DisableInputs
  f DropUsableItem
  f ResetInteractionStates
  f TryStartInteraction
  f CheckInteractionConditions
  f FindPlayerInteractionData
  f TryDefaultInteraction
  f CheckContinueInteraction
  f CompleteDefaultConditions
  f TryPlayEmoteInteraction
  f FindStanceInteractionsPreset
```

-
- **TryPlayEmoteInteraction** - try to play emote interaction.
 - **FindStanceInteractionsPreset** - try to find specific interactions preset handle from specific item or from presets variable depending on current stance.

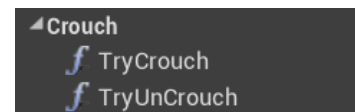
4.3.4.7. Dialogue interaction functions

- **StartDialogue (overridden)** - do start dialogue actions of the player character.
- **FinishDialogue (overridden)** - do finish dialogue actions of the player character.
- **UpdateTalkingState (overridden)** - hide selected usable item of the character.
- **PlayDialogueReply (overridden)** - do dialogue reply actions of the player character.



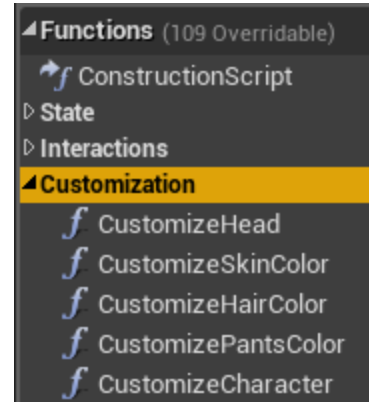
4.3.4.8. Crouch interaction functions

- **TryCrouch** - try to crouch.
- **TryUncrouch** - try to uncrouch.

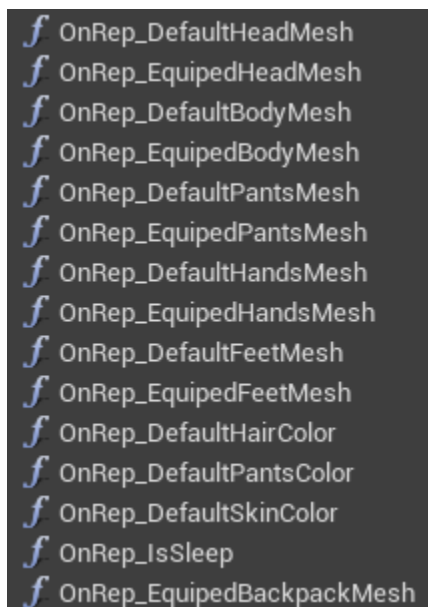


4.3.4.9. Customization functions

- **CustomizeHead** - change the head mesh to an array mesh depending on the index.
- **CustomizeSkinColor** - change the skin color to an array color depending on the index.
- **CustomizeHairColor** - change the hair color to an array color depending on the index.
- **CustomizePantsColor** - change the pants color to an array color depending on the index.
- **CustomizeCharacter** - customize character using target customisation indexes.



NOTE: OnRep functions are used for replicating equipment meshes for the character during the game when it equips or unequips items. They also are used for customisation of the character.



4.3.5. Player interaction system

4.3.5.1. Base player interactions

Interactions which the player character can do are depending on the selected item. For example if the sword item is selected the character can attack and block attacks, and if the pistol item is selected then the character can shoot and reload the pistol. There are special functions for each interaction type.

CanInteract - a function that checks that the player character can interact. The character can interact if it is on the ground and it does not play root or upper body animation.

TryAttack - a function that checks that the player character can interact and if he can then the character plays attack animation depending on the selected melee weapon type. If the character is attacking then it tries to play the next attack animation. The next attack animation can be configured using the continue attack notify in the attack animation.

TryToolInteraction - a function that checks that the player character can interact and if he can then the character plays tool interaction animation depending on selected tool item type.

TryStartShooting - a function that checks that the player character can interact and it has ammo in selected weapon or in the inventory and if conditions are completed then the character plays shoot animation depending on selected weapon type and current aiming state. To continue shooting use the continue shoot notify. Add it to the shoot animation and configure the details.

TryReload - a function that checks that the player character can interact and can reload the selected weapon and if conditions are completed then the character plays reload animation depending on selected weapon type.

TryUseItem - a function that checks that the player character can interact and can use selected item ability and if conditions are completed then the character plays use or magic animation depending on selected item or spell.

TryThrowItem - a function that checks that the player character can interact and can use selected item ability and if conditions are completed then the character plays throw item animation depending on selected item and current aiming state.

TryChargeInteraction - a function that checks that the player character can interact and has ammo in selected weapon or in the inventory and if conditions are completed then the character plays charge animation depending on selected item type.

TryMainInteraction - a function that selects specific interaction depending on selected item type.

ItemMainInteraction - the main interaction function that processes the main input button key (left mouse button key) and selects specific main interaction depending on selected item settings. This function also checks the advanced interaction handle of the selected item and processes the custom interaction for the main interaction button with specific conditions and actions in the **TryStartInteraction** function.

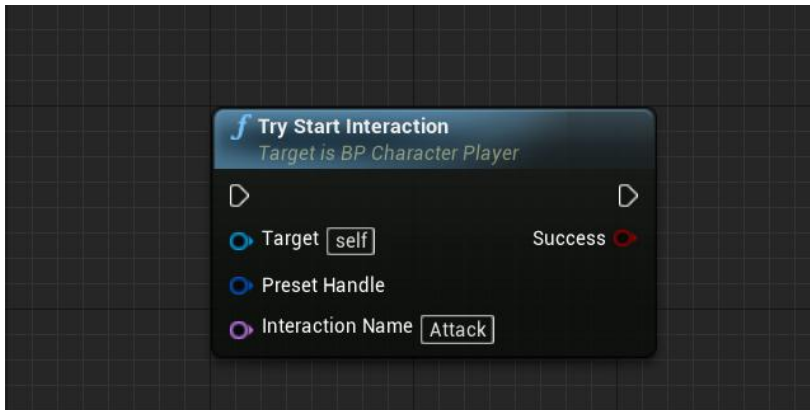
ItemSubInteraction - the main interaction function that processes the sub input button key (right mouse button key) and selects specific sub interaction depending on selected item settings. This function also checks the advanced interaction handle of the selected item and processes the custom interaction for the sub interaction button with specific conditions and actions in the **TryStartInteraction** function.

4.3.5.2. Advanced player interaction system

Advanced player interaction system allows users to configure specific interactions with specific conditions for specific items. For example you can set the attack interaction to work in the air or in the water. It also allows users to set up attack combinations with different input buttons (left / right mouse buttons combinations for example).

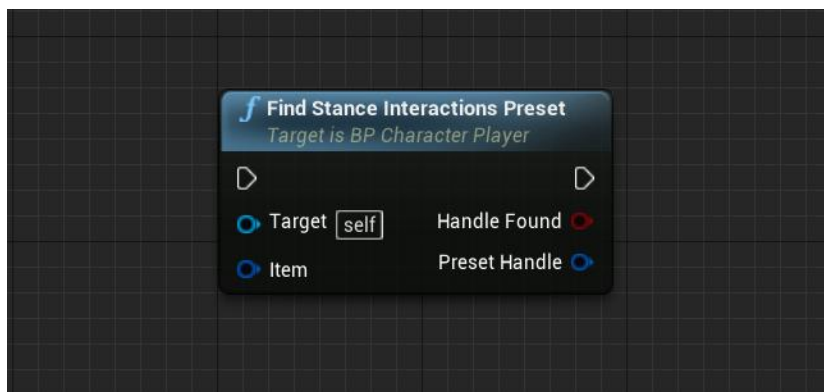
To play advanced interaction use the **TryStartInteraction** function. This function gets the player stance interactions preset data (**STR_PlayerInteractionsPreset**) from the data table by the preset handle and then tries to find player interaction data

(**STR_PlayerInteractionData**) by the interaction name. If the player interaction data has been found and conditions are complete this function plays the interaction depending on interaction settings.



The preset handle can be found using the **FindStanceInteractionsPreset** function. This function tries to get the stance interactions preset handle from the item using the

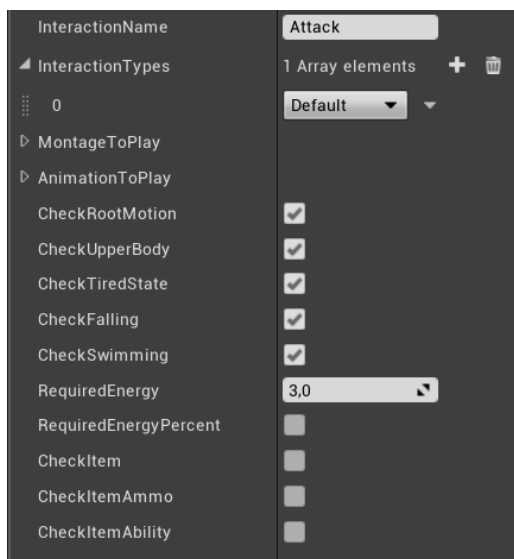
StanceInteractionsPresetKey or from the **StanceInteractionsPresets** variable.



STR_PlayerInteractionData

The **STR_PlayerInteractionData** structure is used for advanced player interaction settings. This structure includes interaction type, montage and animation play settings and various conditions.

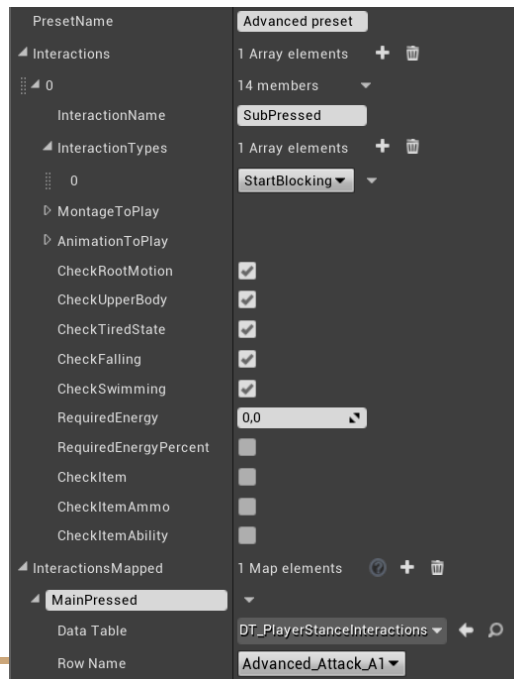
Advanced stance interaction settings can be added and configured in a custom data table based on the **STR_PlayerInteractionData** or in the **DT_PlayerStanceInteractions** data table.



STR_PlayerStancePreset

The **STR_PlayerStancePreset** structure is used for advanced stance interaction preset settings. This structure contains a list of advanced interaction settings and a mapped list of handles that can be used for connection with the advanced player interactions data table.

Advanced stance interaction presets can be added and configured in a custom data table based on the **STR_PlayerStancePreset** or in the **DT_PlayerStancePresets** data table.



Item interactions

The **ItemMainInteraction** and the **ItemSubInteraction** functions try to find the advanced stance interactions preset data to play first. If advanced stance interactions preset data is not found then the function plays base interactions.

The specific name variables are used to get advanced stance interaction data from the preset in the main and sub item interaction functions.

Stance Main Pressed Interaction Name	MainPressed
Stance Main Pressed Aim Interaction Name	MainPressedAim
Stance Main Released Interaction Name	MainReleased
Stance Main Released Aim Interaction Name	MainReleasedAim
Stance Sub Pressed Interaction Name	SubPressed
Stance Sub Released Interaction Name	SubReleased

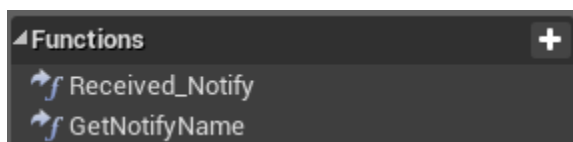
4.4. Work with animations

4.4.1. Anim notifies

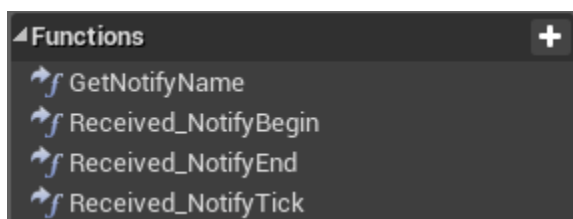
Animation notifies are used to call various functions during animations. These notifies can be used to set special states for a character during animations, to call functions for deal damage, call functions for footsteps, etc.

There are two standard notify classes:

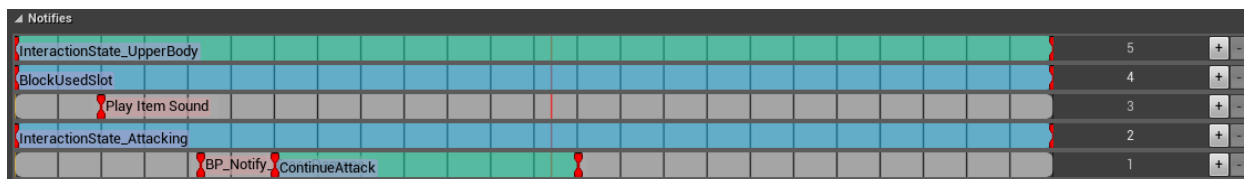
- **AnimNotify** is a notify can be set at a specific moment in the animation. The class has a function for getting the name of the notification, as well as a function for triggering the notify when it is called. These functions can be overridden.



- **AnimNotifyState** is a notify which can be set for a specific animation time. The class has a function for getting the name of the notify, functions that are triggered at the beginning and at the ending of a notify call, and a function that is triggered by a tick during the whole animation. These functions can be overridden.



The anim notify can be placed in any moment of the animation.



4.4.2. Project notifies

BP_Notify_CompleteDigging

This notify is used to complete the digging interaction for the character. To be triggered, the character must implement the **CompleteDigging_BPI** function from the **BPI_PlayerCharacter** interface.

BP_Notify_Footstep

This notify is used to play footstep effects for the characters. To be triggered, the character blueprint should have a configured **BP_FootstepComponent**.

FootSocket - a socket name that is used for play footstep effects.

FootstepType - footstep touching type.

PlaySound - an indicator that footstep sound should be played.

SpawnEffects - an indicator that footstep visual effects should be spawned.

Preview variables - variables that are used for playing footstep sounds in the editor.



BP_Notify_ItemUsed

This notify is used to complete the use item interaction, cast the item ability and the consume the item if needed. To be triggered, the character must implement the **ItemUsed_BPI** function from the **BPI_PlayerCharacter** interface.

BP_Notify_LaunchItem

This notify is used to complete the throw item interaction, cast the item launch ability with specific launch speed and then consume the item if needed. To be triggered, the character must implement the **LaunchItem_BPI** function from the **BPI_Character** interface.

BP_Notify_PlaySound

This notify is used to play a specific sound depending on the specific name for the character. To be triggered, the character must implement the **PlaySound_BPI** function from the **BPI_Character** interface.

SoundName - an identifier that determines which sound will be played.



BP_Notify_Reload

This notify is used to complete the reload interaction of the character. To be triggered, the character must implement the **Reload_BPI** function from the **BPI_Character** interface.

BP_Notify_SetChargedState

This notify is used to update the charged state of the character. To be triggered, the character must implement the **SetChargedState_BPI** function from the **BPI_Character** interface.

NewCharged - the new charged state of the character to be set.



BP_Notify_Shoot

This notify is used to complete the shooting interaction, cast item ability and consume ammo of the weapon. To be triggered, the character must implement the **Shoot_BPI** function from the **BPI_Character** interface.

AimedShoot - an indicator that the shot should be aimed.



BP_Notify_SpawnActor

This notify is used to spawn a specific actor in front of the character.

SpawnClass - a class of the actor that should be spawned.

CheckSpawnedActors - if true prevents spawning more than one actor of the same class in specific range.

CheckSpawnedActorsRange - the range that is used in the **CheckSpawnedActor** function.

CheckSpawnedActorsTypes - a collision channel that is used in trace function for finding the same actors.

MinPitchAngle - a minimum pitch angle that determines the vector for the trace function.

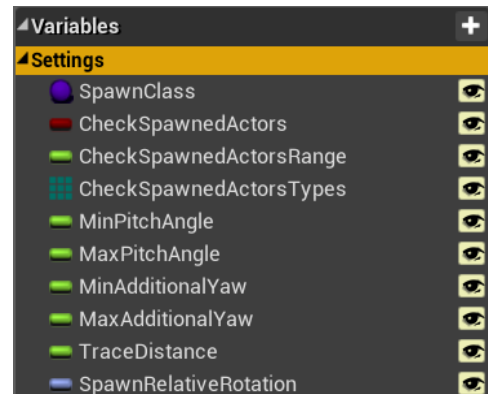
MaxPitchAngle - a maximum pitch angle that determines the vector for the trace function.

MinAdditionalYaw - a left angle limit that determines rotation for the trace vector.

MaxAdditionalYaw - a right angle limit that determines rotation for the trace vector.

TraceDistance - a distance that is used in the trace function.

SpawnRelativeRotation - additional rotation for spawned actors.



BP_Notify_TraceDamage

Calls the function for dealing damage by the character using traces. To be triggered, the character must implement the **TraceDealDamage_BPI** function from the **BPI_Character** interface.

BP_NotifyState_BlockUsedSlot

Blocks an item in the active hotbar slot for the duration of the notify. For the character class to work, it must implement the **BlockUsedSlot_BPI** function from the **BPI_PlayerCharacter** interface.

BP_NotifyState_CheckChargeReleasing

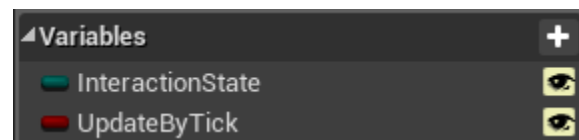
Checks and releases charged interaction for the duration of the notify. To work, the character class must implement the **CheckChargeRelease_BPI** function from the **BPI_Character** interface.

BP_NotifyState_InteractionState

Sets interaction states for the character for the duration of the notify. To work, the character class must implement the **SetInteractionState_BPI** function from the **BPI_Character** interface.

InteractionState - the interaction state of the character to be set.

UpdateByTick - an indicator that the state will be updated every tick while the notify is working.



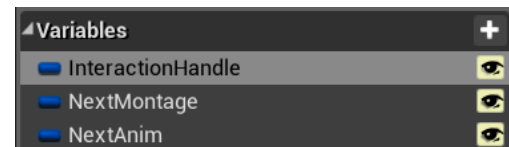
BP_NotifyState_ContinueAttack

Checks and tries to continue attack for the character for the duration of the notify. To work, the character class must implement the **ContinueAttack_BPI** function from the **BPI_Character** interface.

InteractionHandle - a handle that is used to get next attack interaction data, which contains animation or montage settings.

NextMontage - montage settings for the next attack.

NextAnim - animation settings for the next attack.

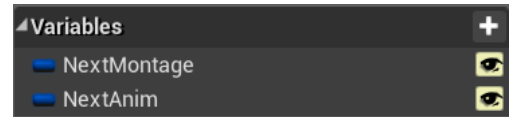


BP_NotifyState_ContinueShooting

Checks and tries to continue attack for the character for the duration of the notify. To work, the character class must implement the **ContinueShooting_BPI** function from the **BPI_Character** interface.

NextMontage - montage settings for the next shooting animation.

NextAnim - animation settings for the next shooting animation.

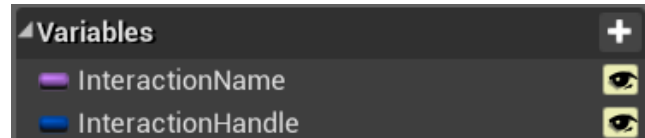


BP_NotifyState_ContinueInteraction

The notify checks and tries to continue interaction for the character. To work, the character class must implement the **ContinueInteraction_BPI** function from the **BPI_PlayerCharacter** interface.

InteractionName - an identifier of interaction that will be added to the continue interactions list. If the character tries to start interaction with this name the notify will continue interaction using specific player interaction data.

InteractionHandle - a handle that is used to get player interaction data from the data table which contains animation and montage settings and play conditions of the next interaction.

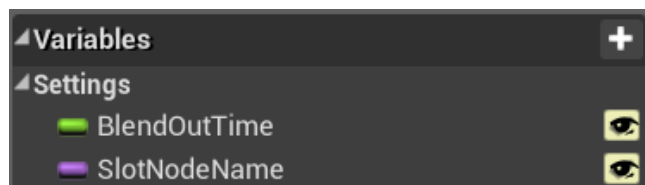


BP_NotifyState_InterruptByMovement

The notify checks and tries to interrupt animation if the character starts to move. To work, the character class must implement the **InterruptByMovement_BPI** function from the **BPI_Character** interface.

BlendOutTime - a blend time that is used to stop animation.

SlotNodeName - a slot name that determines which animation group will be stopped.

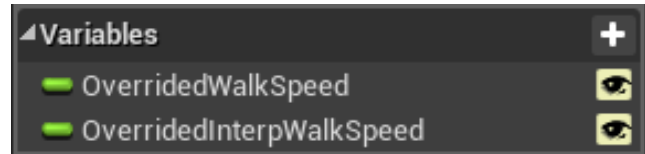


BP_NotifyState_OverrideWalkSpeed

The notify overrides the movement speed of the character. To work, the character class must implement the **OverrideWalkSpeed_BPI** function from the **BPI_Character** interface.

OverriddenWalkSpeed - overridden movement speed value.

OverriddenInterpWalkSpeed - the rate of change of the movement speed.

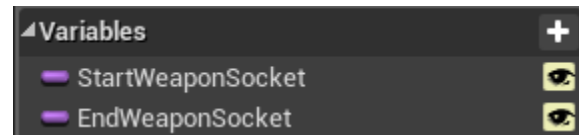


BP_NotifyState_SetWeaponSocket

The notify changes the attachment socket of the active character item or weapon. To work, the character class must implement the **SetWeaponSocket_BPI** function from the **BPI_Character** interface.

StartWeaponSocket - the socket on which the active item is attached at the notify begin.

EndWeaponSocket - the socket on which the active item is attached at the notify end.



BP_NotifyState_SetWeaponHidden

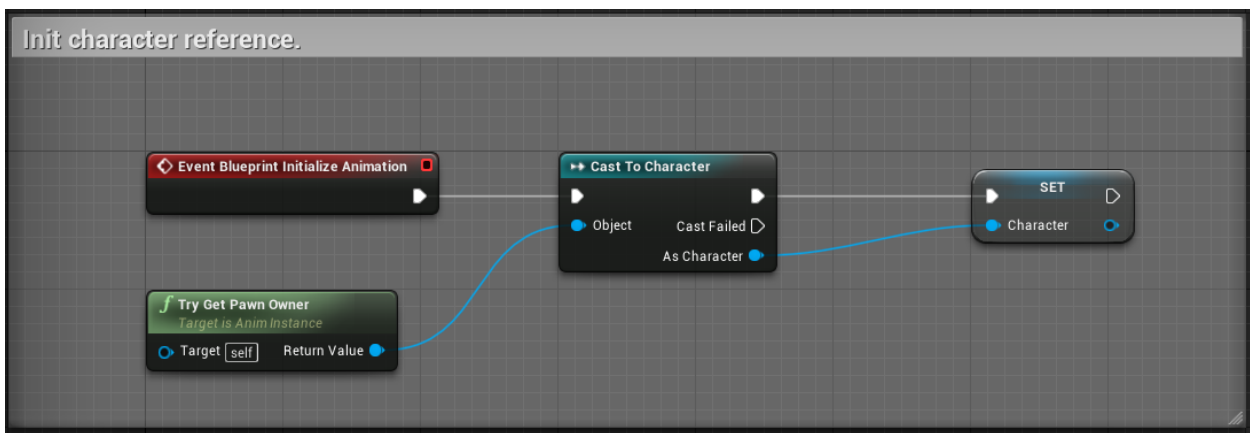
The notify hides the active character item or weapon. To work, the character class must implement the **SetWeaponHidden_BPI** function from the **BPI_Character** interface.

4.4.3. Anim Blueprint

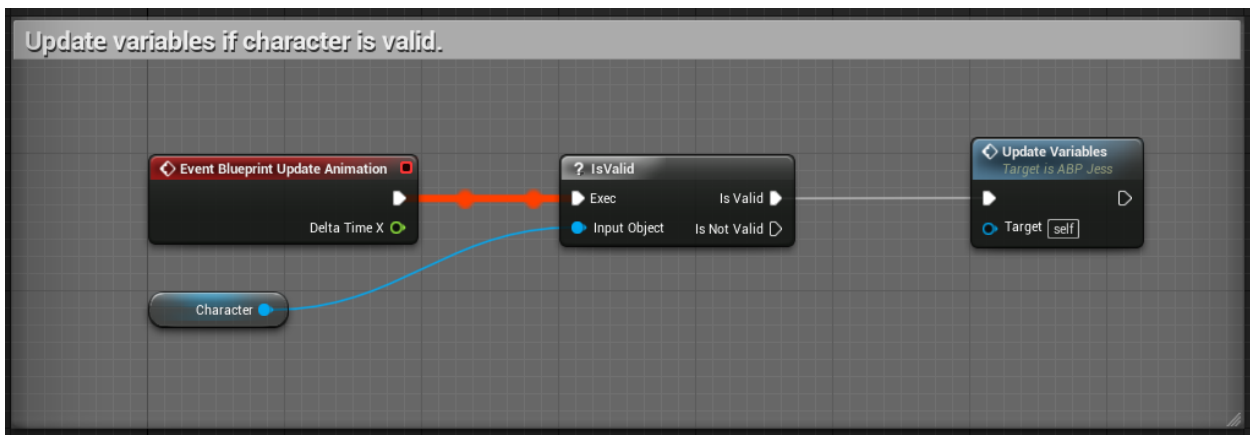
To work with an animation blueprint, you need to know the state of the character. In order to get and update the states of the character, you need to override two functions:

BlueprintInitializeAnimation and **BlueprintUpdateAnimation** in the **Event Graph**.

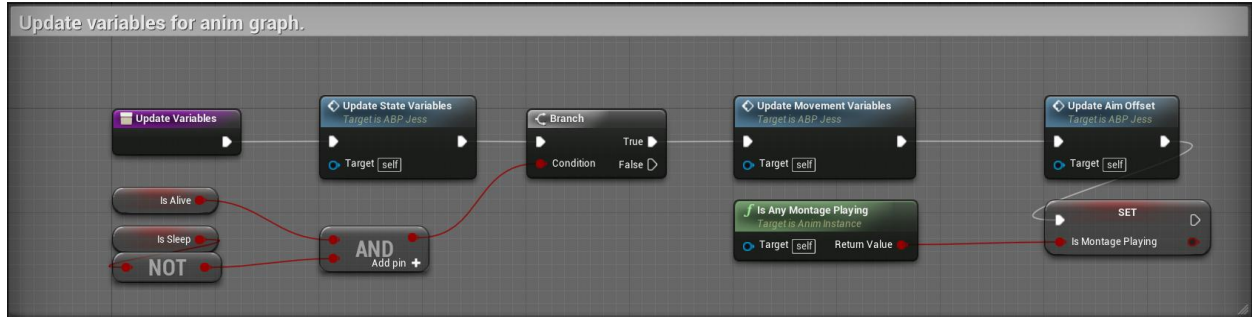
The **BlueprintInitializeAnimation** function is called when the animation blueprint is initialized. In this function, you need to initialize the link of the active character in order to access its state variables.



The **BlueprintUpdateAnimation** function is called every tick before the animations start blending. In this function, you need to determine and get the values of all the state variables of the character before blending and selecting the appropriate animations.



In the **UpdateVariables** function, all states of the character are determined and values are set to the appropriate variables. These variables can include the character's speed, the current movement mode, the indicator that the character is alive or is dead, etc.



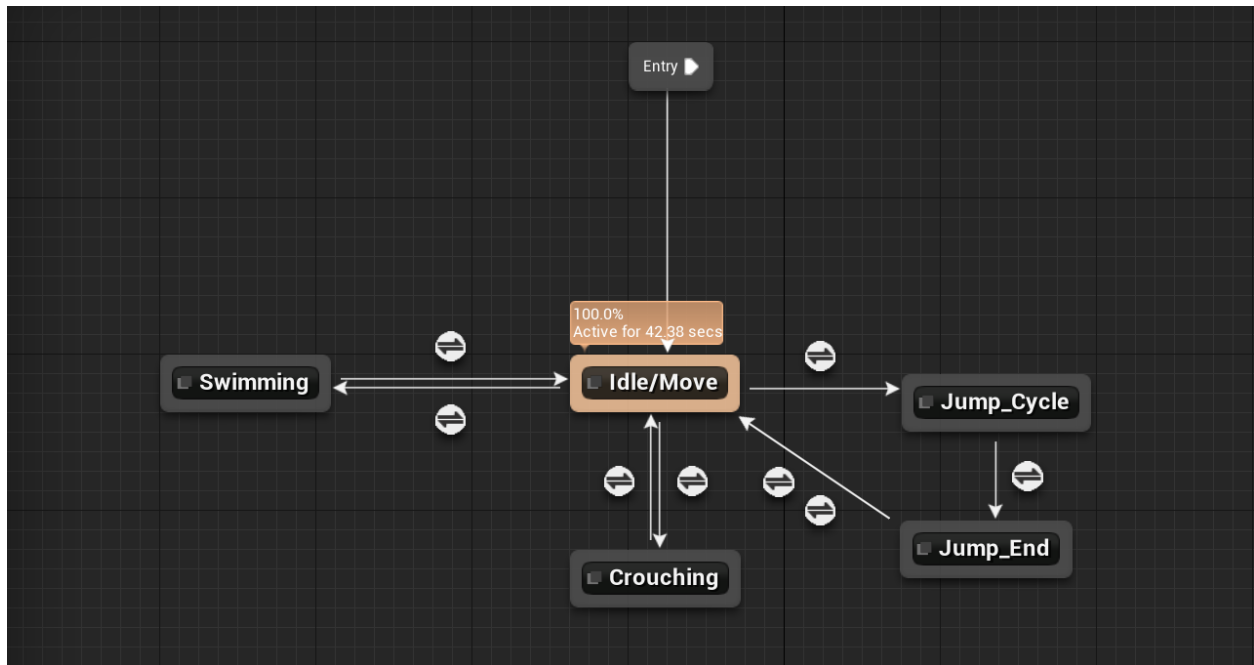
In the **UpdateVariables** function for the game character, three functions are called to determine all the states required for the blend animations.

UpdateStateVariables - gets the variables needed to determine the state of the character. Here it is determined whether the character is alive or dead, in which stance he is depending on the selected weapon, etc.

UpdateMovementVariables - gets the variables needed to determine the current mode of movement, the speed of movement of the character, his direction, etc.

UpdateAimOffset - gets the variables needed to turn the character's head in the direction the player is facing.

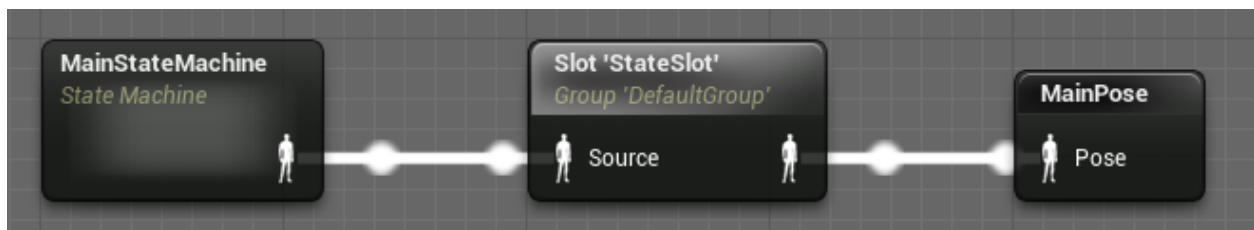
The animation blending for the character depending on the character's movement mode. The main transitions between character animations are in the **MainStateMachine**.



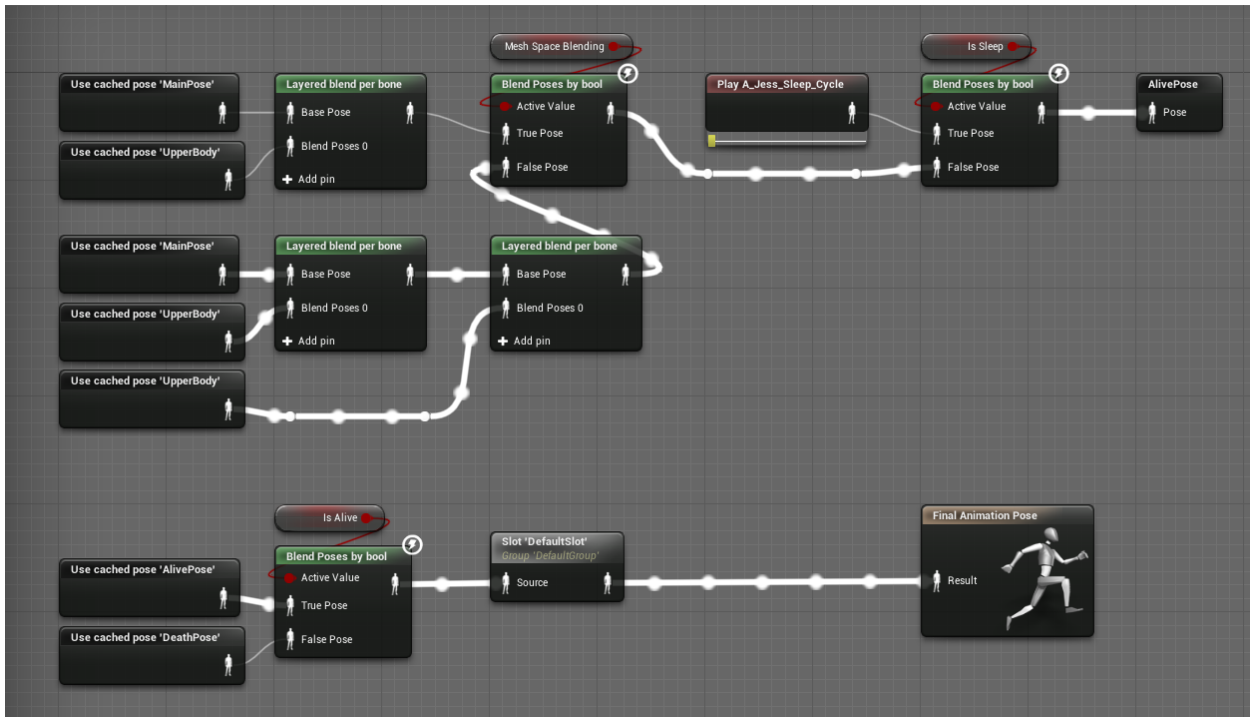
The transitions between the main modes of movement are implemented here. The character is in the **Idle / Move** state if he is on the ground. After the jump, it goes into the **Jump_Cycle** state until it lands.

On landing, it goes into the **Jump_End** state and then returns to **Idle / Move**. If a character enters the water, he goes into the **Swimming** state until he gets on land.

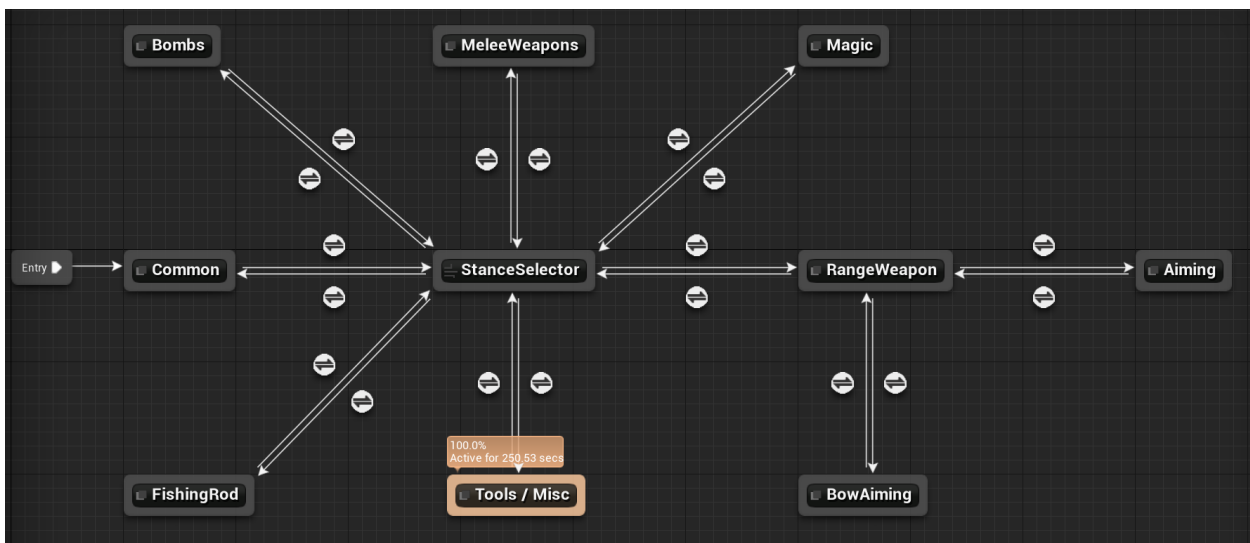
The main pose of the character is caught in **MainPose**.



The final pose of the character is obtained from the upper body pose and his main pose.



The upper body pose consists of the stance pose and additional blendings such as aim offsets and arms blends. The stance pose depends on the current equipment stance. The main transitions for the upper body are in the **StanceStateMachine**.



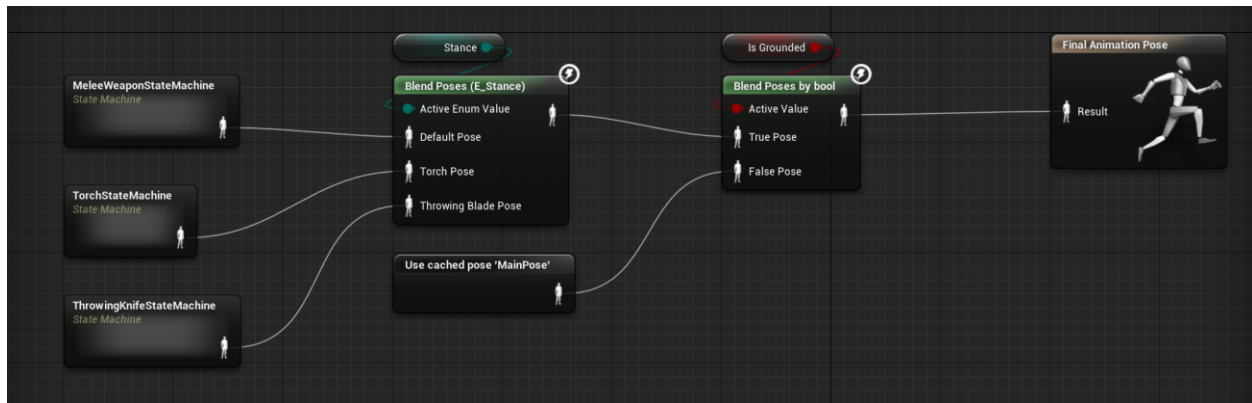
The character is in the **Common** state if he does not have a single item in hands. The current stance type is determined by the stance variable of the character. As soon as this

variable changes to a different state from the current state, there is an instant transition to **StanceSelector**, which determines the further transition depending on the current stance.

For example, if a character equips a melee weapon in the active slot, the transition

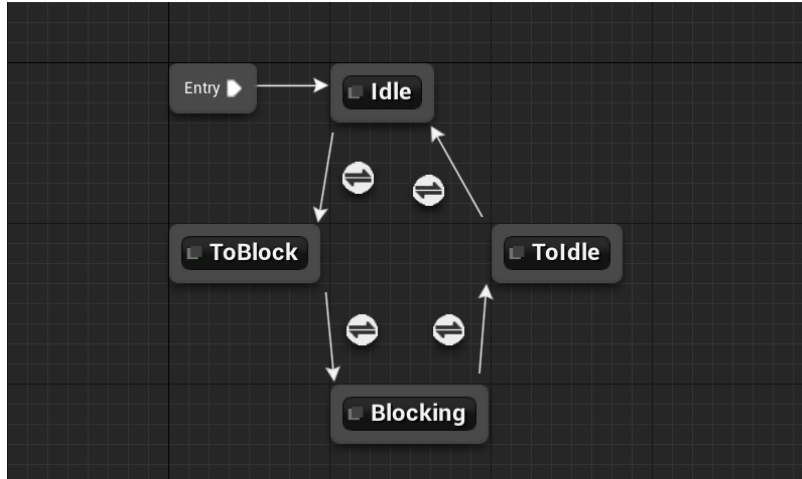
Common > StanceSelector > MeleeWeapons will occur.

The state from the stance state machine can contain blends for familiar stances.

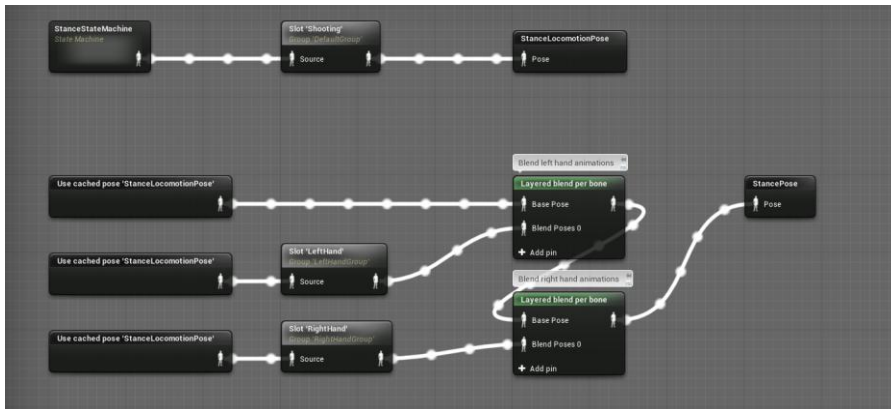


Specific stance can have a specific state machine. For example the

MeleeWeaponStateMachine contains transitions for the block interaction.

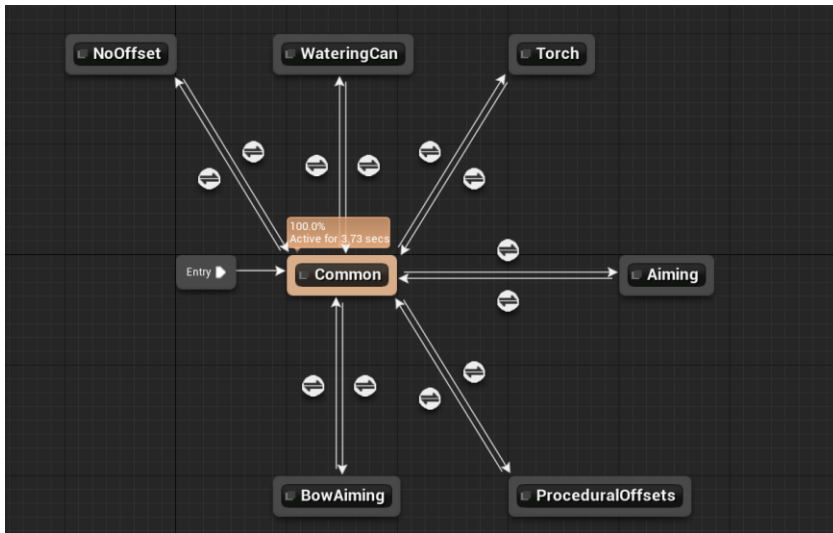


The final stance pose is caught in **StancePose**.

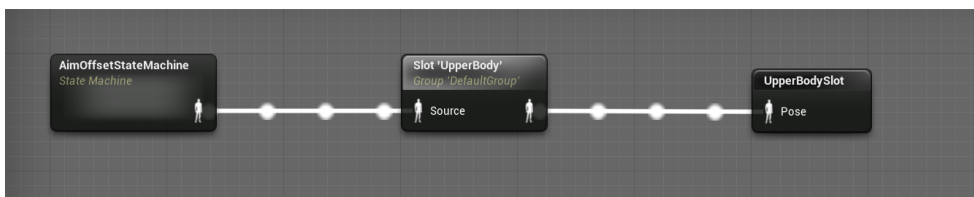


The **Shooting** slot is used for upper body animations such as attack or shoots. The **LeftHand** and the **RightHand** slots are used animations where you need to use only a specific hand.

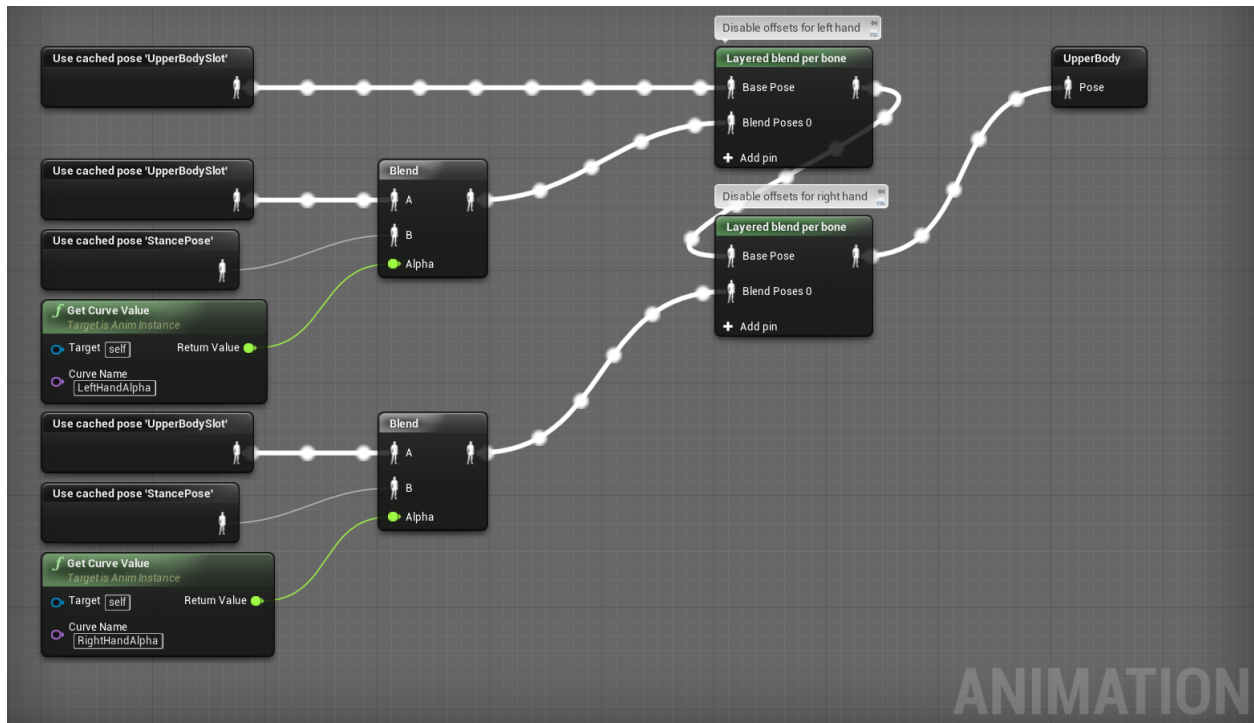
The **AimOffsetsStateMachine** determines offsets for spine and arm bones when the character uses a specific item. There can be configured aim offsets for specific stances.



The **UpperBody** slot is used for upper body animations where you need to play animation without aim offsets.



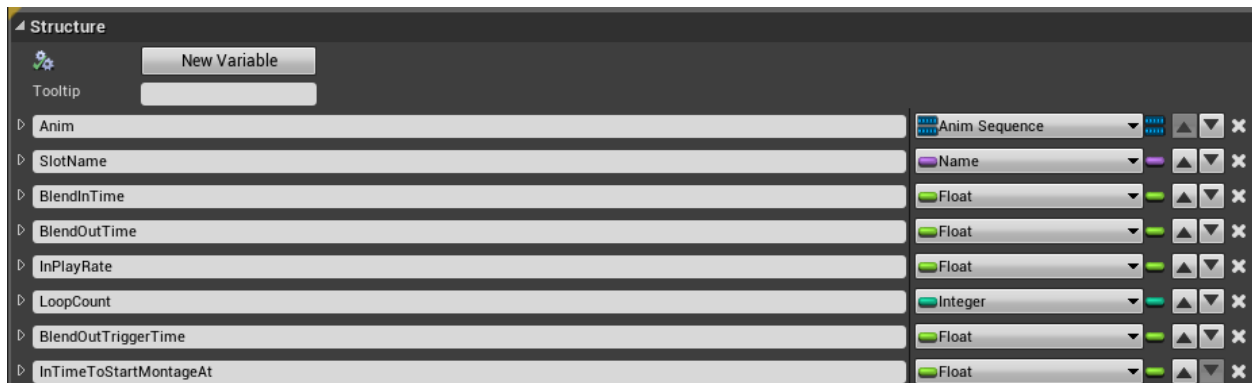
The final upper body pose contains blends which allow the user to disable a specific hand using the **LeftHandAlpha** or the **RightHandAlpha** curves.



Various interaction animations, such as attacking with a weapon or using items, are called from the character's blueprints.

Each animation must be configured to play in a dedicated slot. The animation settings are stored in the **STR_AnimationPlayData** in the **STR_MontagePlayData** structures.

STR_AnimationPlayData



Anim - an animation sequence that should be played.

SlotName - animation slot for which the animation will be called.

BlendInTime - blending time when animation starts playing.

BlendOutTime - blending time when animation ends playing.

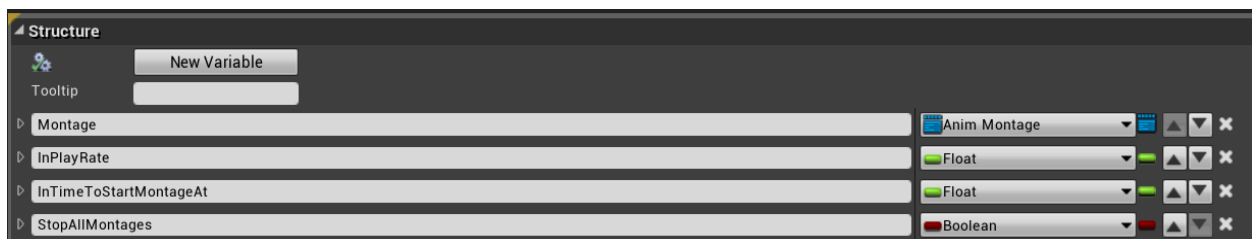
InPlayRate - the animation play rate.

LoopCount - the count of animation plays in a loop.

BlendOutTriggerTime - time relative to the end of the animation, from which the animation blend starts.

InTimeToStartMontageAt - time from which the animation will start playing.

STR_MontagePlayData



Montage - a montage that should be played.

InPlayRate - the animation play rate.

InTimeToStartMontageAt - time from which the animation will start playing.

StopAllMontages - determines that all active montages should be stopped.

The interaction animations and montages can be played for different animation slots.

The **DefaultSlot** is used when you need to play animation without any blends.

The **StateSlot** is used when you need to override the main pose of the character.

The **UpperBody** is used when you need to play upper body animation without aim offsets.

The **Blocking** is used when you need to play block hit reaction animation.

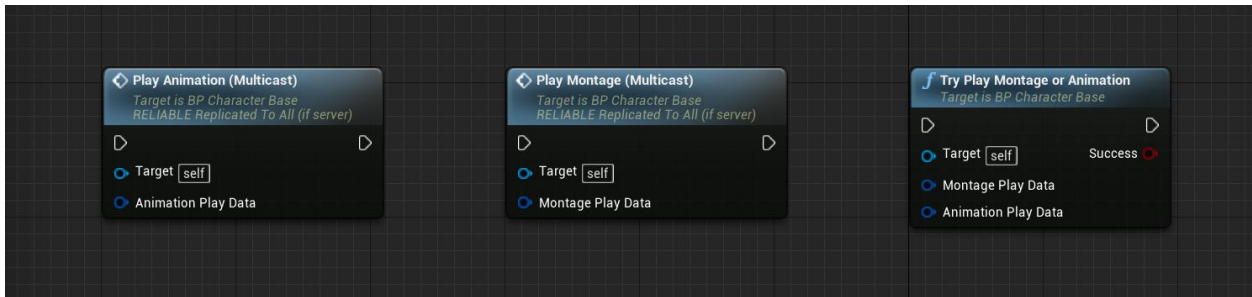
The **Shooting** is used when you need to play upper body animation with aim offsets.

The **LeftHand** is used when you need to play animation only for the left hand.

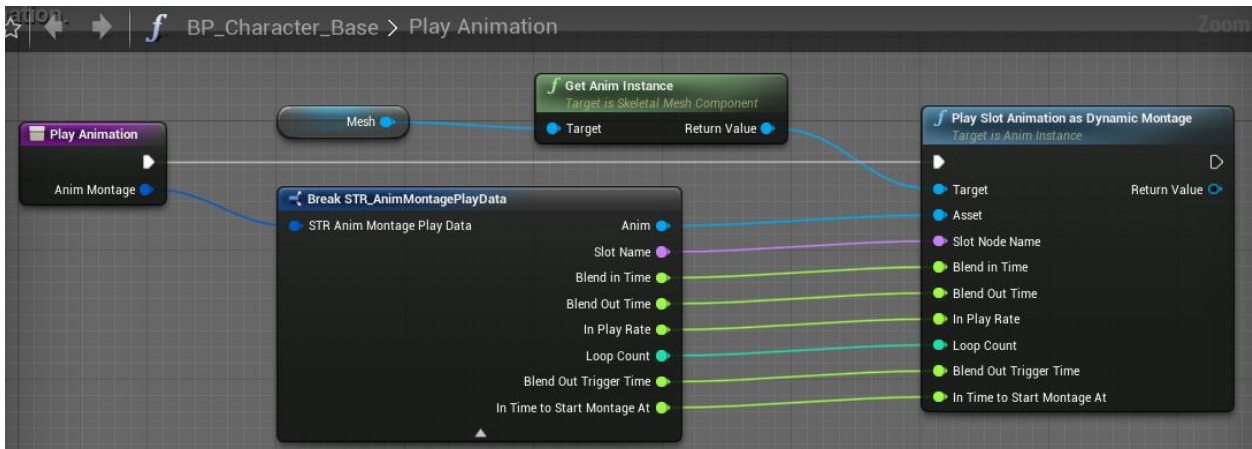
The **RightHand** is used when you need to play animation only for the right hand.

4.4.4. Character animations

The **PlayAnimation (Multicast)**, the **PlayMontage(Multicast)** and the **TryPlayMontageOrAnimation** events are used for playing different interaction animations for the characters. These events replicate animation for all clients.

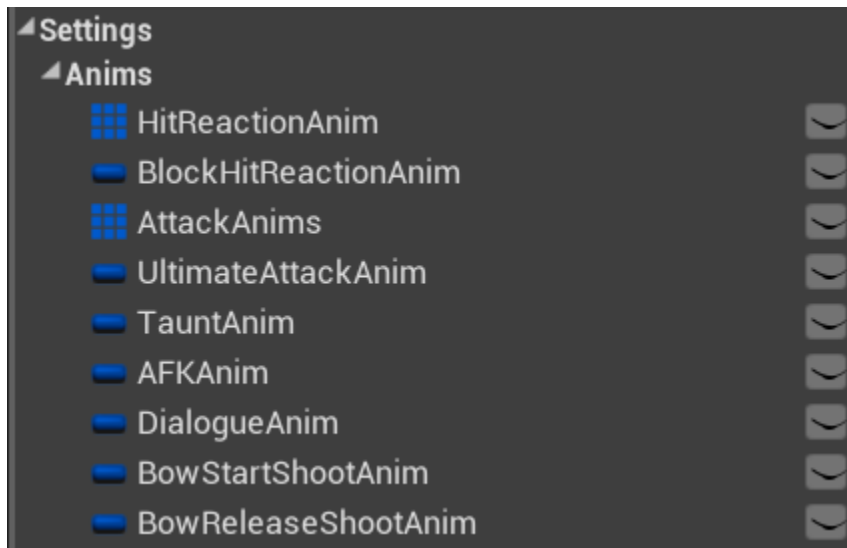


The **STR_AnimationPlayData** and the **STR_MontagePlayData** structures are used for sending data about animation which should be played. This data is used for playing animation with specific settings as dynamic montage for the certain animation slot.



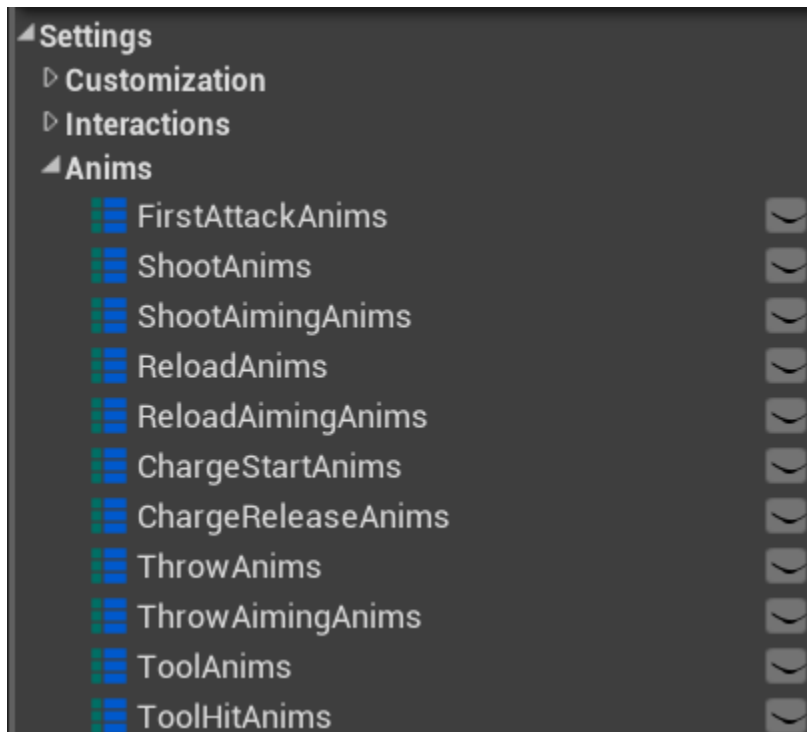
NOTE: Always use montage and the **STR_MontagePlayData** structure when you need to play animation with root motion.

The interaction animations can be configured in the settings section of the character. For simple characters like undead or animals the animation or montage structures are used.



The player interactions are set as handles that are used for loading specific data from the animations data table. The interaction data contains animation and montage play data in a

single structure.



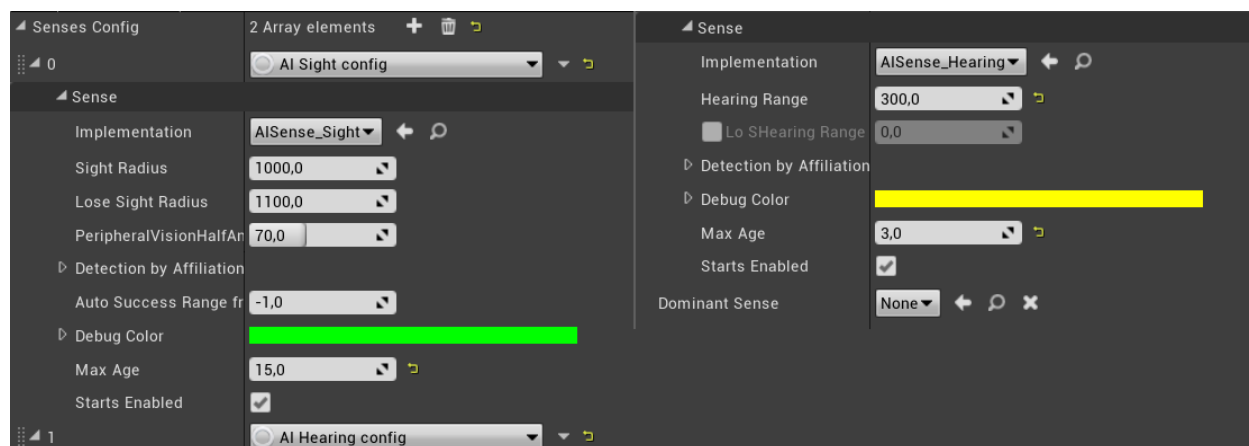
4.5. Artificial Intelligence

4.5.1. Description

Artificial intelligence is based on a behavior tree. It determines the state of the character and assigns him certain tasks that the character must complete.

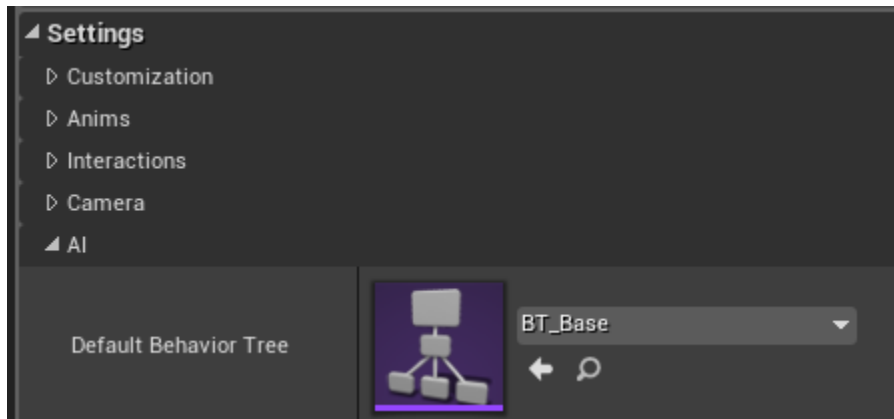
All NPCs are controlled by the **BP_AIController_Base** or **BP_AIController_Detour** classes. In these classes, the **BB_Base** blackboard is initialized, which contains variables for working with the character's artificial intelligence. Then the controller initializes the behavior tree for the controlled character in the **RunAI** function of the character.

These classes also have a **Perception Component** that is used for detecting actors by different senses. You can set sight and hear radius for detecting threats and enemies.



NOTE: The character which uses the **BP_AIController_Detour** class can get stuck in place, because it has inner functionality to detour obstacles that is limited in the **Project Settings** (**Max Agents** property). If you don't need such functionality you can select the base class for the character in the class defaults.

The default behavior tree can be set in the **DefaultBehaviorTree** variable of the character. The behavior tree also can be overridden from the spawn point blueprint if the behavior tree is set in the spawn settings.

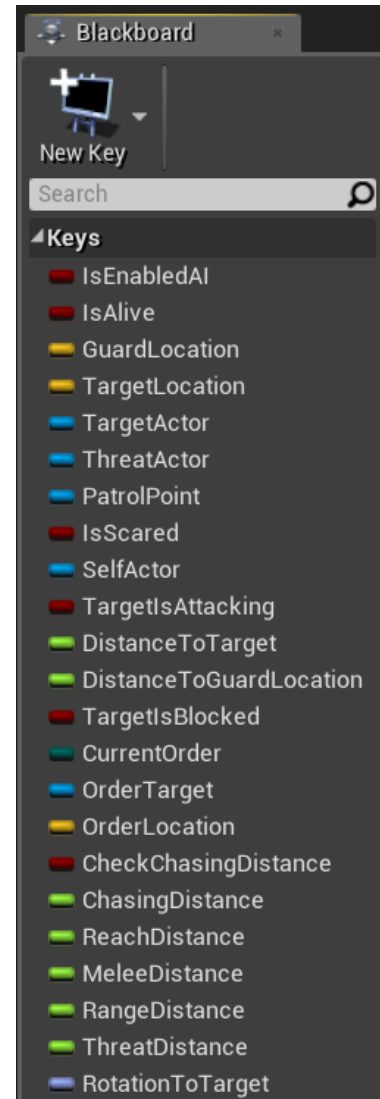


The base character class **BP_Character_Base** implements the main logic for artificial intelligence. The character can identify other characters as neutral, as a threat, or as an enemy. If the character is aggressive (**IsAgressive**) and can attack (**CanAttack**), he will automatically attack the threat when the character detects it. If the character cannot attack and can be scared (**CanBeScared**), he will run away from the threat. The first patrol point can be set in the character to which he will go at the beginning of the game or at the moment of spawn.

4.5.2. Blackboard

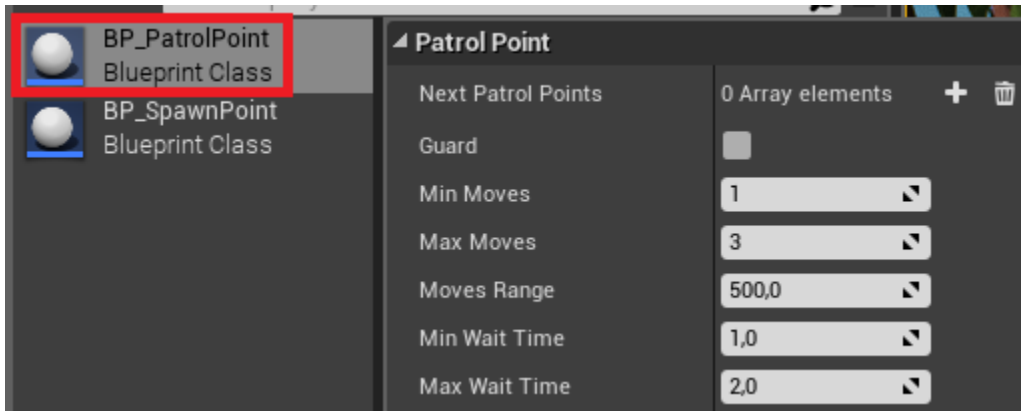
The **BB_Base** blackboard is used to define variables for working with artificial intelligence in the behavior tree.

- **IsEnabledAI** - an indicator that the behavior tree is active for the character.
- **IsAlive** - an indicator that the character is alive.
- **GuardLocation** - point guarded by the character.
- **TargetLocation** - the point to which the character should move.
- **TargetActor** - a chased actor reference.
- **ThreatActor** - a main threat reference.
- **PatrolPoint** - a current character patrol point.
- **IsScared** - an indicator that the character is scared.
- **SelfActor** - reference to the character itself.
- **TargetIsAttacking** - an indicator that the target is attacking.
- **DistanceToTarget** - distance to the target.
- **DistanceToGuardLocation** - distance to the guard location.
- **TargetIsBlocked** - an indicator that the target of the character is blocked by an obstacle.
- **CurrentOrder** - current order for the character.
- **OrderTarget** - a target actor of the current order for the character.
- **OrderLocation** - a target location of the current order for the character.
- **CheckChasingDistance** - an indicator that the character should check distance to guard point while chasing target.
- **ChasingDistance** - chasing distance value.
- **ReachDistance** - acceptance distance for move to tasks.
- **MeleeDistance** - acceptance distance for melee attacks and actions.
- **RangeDistance** - acceptance distance for range attacks and actions.
- **ThreatDistance** - threat distance value.
- **RotationToTarget** - rotation to the target.



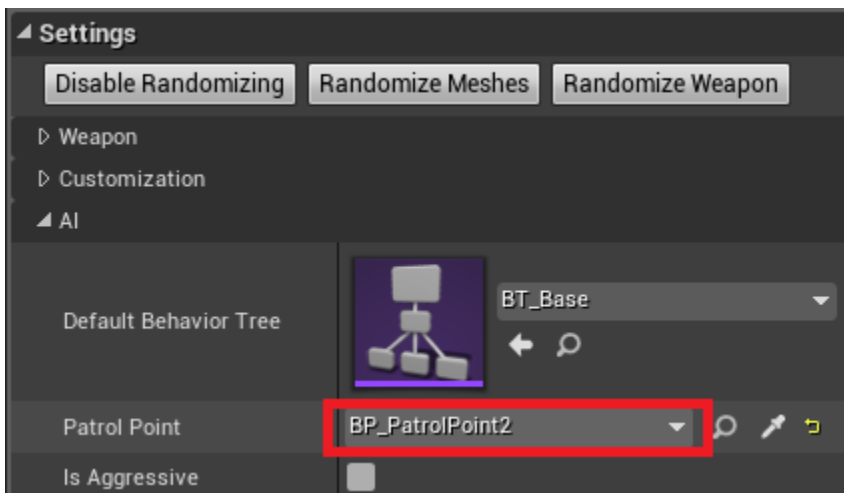
4.5.3. Patrol points

Patrol points allow characters to move between different points. They also allow characters to guard locations. Place the **BP_PatrolPoint** on the level and configure it. You can add next patrol points and characters will select one of it randomly as the next patrol point.



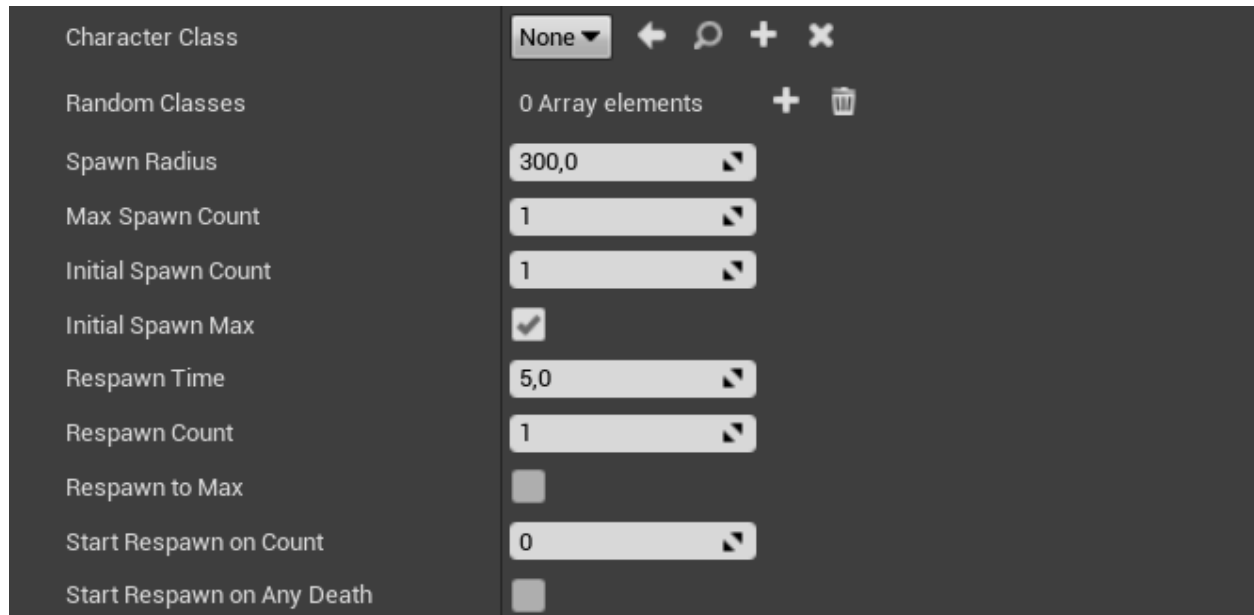
Characters also do some moves around patrol points. But if the **Guard** variable is checked then they just hold position. After some moves or delay in guard position they move to the next patrol point if it is available.

The first patrol point can be selected for each character in the AI settings.



4.5.4. Spawn points

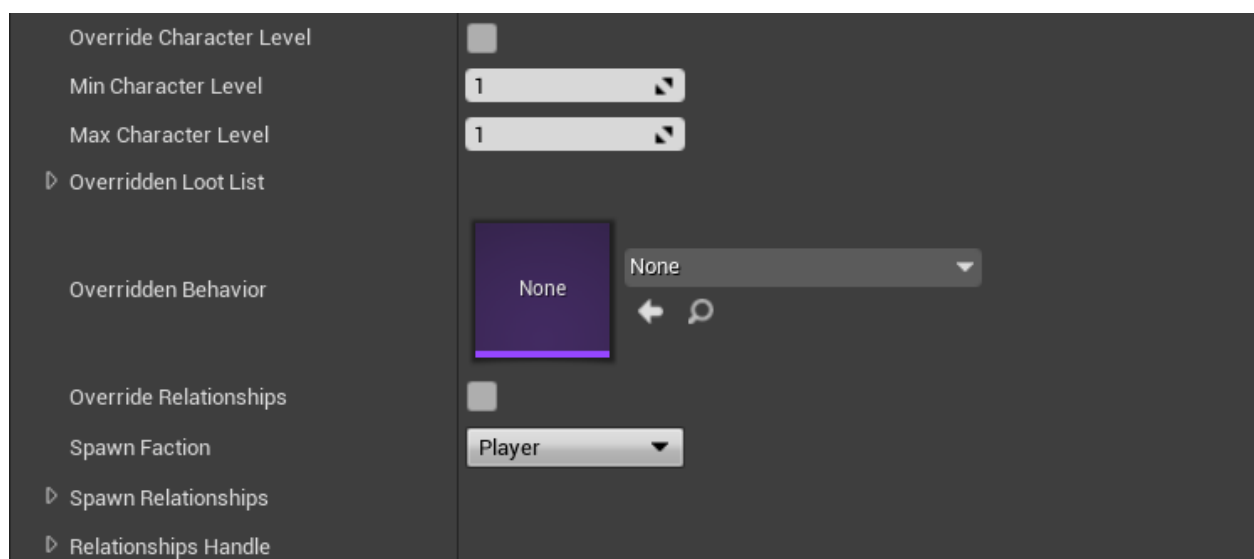
BP_SpawnPoint - is an actor that allows you to spawn non-playable characters on the level. You can configure the character class, amount of characters, respawn time, spawn radius and so on. Spawn point is also the patrol point and has patrol point settings and functionality. The spawned characters select the spawn point as the first patrol point.



The screenshot displays the configuration panel for the BP_SpawnPoint actor. The panel is divided into two columns. The left column lists the settings, and the right column shows their current values and controls.

Setting	Value/Control
Character Class	None (dropdown menu with back, search, add, and delete icons)
Random Classes	0 Array elements (with add and delete icons)
Spawn Radius	300,0 (text input with a right arrow icon)
Max Spawn Count	1 (text input with a right arrow icon)
Initial Spawn Count	1 (text input with a right arrow icon)
Initial Spawn Max	<input checked="" type="checkbox"/>
Respawn Time	5,0 (text input with a right arrow icon)
Respawn Count	1 (text input with a right arrow icon)
Respawn to Max	<input type="checkbox"/>
Start Respawn on Count	0 (text input with a right arrow icon)
Start Respawn on Any Death	<input type="checkbox"/>

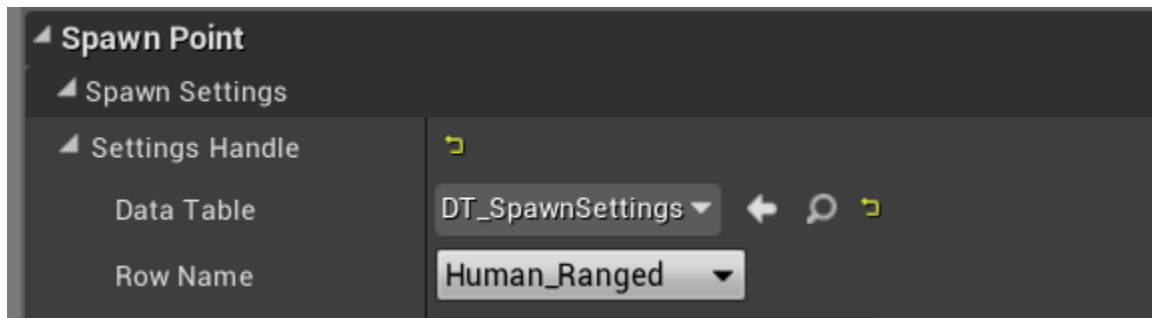
You also can override the character level, death loot, behavior and relationships.



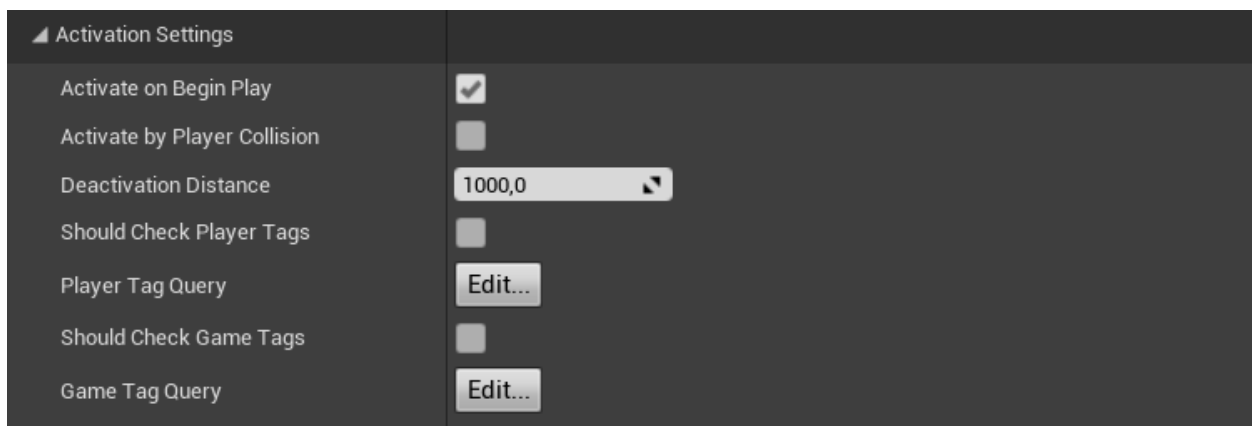
The screenshot displays the configuration panel for the BP_SpawnPoint actor, focusing on the override settings. The panel is divided into two columns. The left column lists the settings, and the right column shows their current values and controls.

Setting	Value/Control
Override Character Level	<input type="checkbox"/>
Min Character Level	1 (text input with a right arrow icon)
Max Character Level	1 (text input with a right arrow icon)
Overridden Loot List	(collapsed)
Overridden Behavior	None (dropdown menu with a right arrow icon)
Override Relationships	<input type="checkbox"/>
Spawn Faction	Player (dropdown menu)
Spawn Relationships	(collapsed)
Relationships Handle	(collapsed)

Spawn settings also can be loaded from the data table using the **SettingsHandle** variable.



The spawn point actor has activation settings that allows you to configure activation mode.



By default spawn points spawn characters on the begin play event, and it can be disabled with the **ActivateOnBeginPlay** variable.

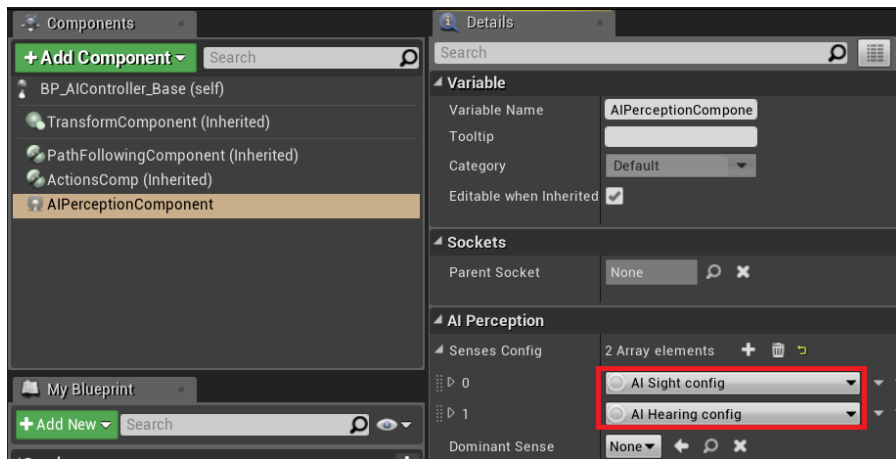
The **ActivateByPlayerCollision** variable allows you to activate and deactivate spawn points automatically using the **SpawnCheckerCollision** of the **BP_PlayerController**. In this activation mode the spawn point also checks specific player tags and specific game tags. This can stop activation if conditions are not complete. Deactivation happens when the player collision leaves activation zone and all spawned characters are in the **DeactivationDistance** radius of spawn point.

To activate or deactivate spawn points manually use the **ActivateSpawnPoint** and the **DeactivateSpawnPoint** functions.

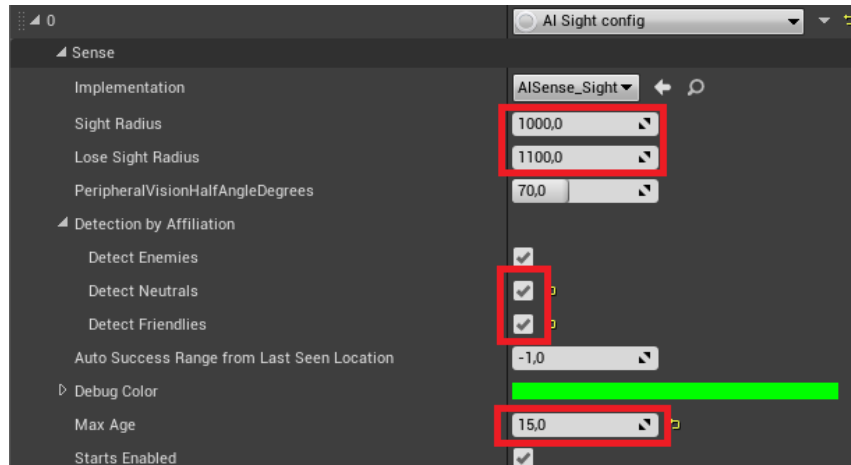
4.5.5. AI Perception

The **AI Perception** component allows the AI controllers to use different senses to detect friends and enemies. The **AI Perception** component should be used only in AI controller classes.

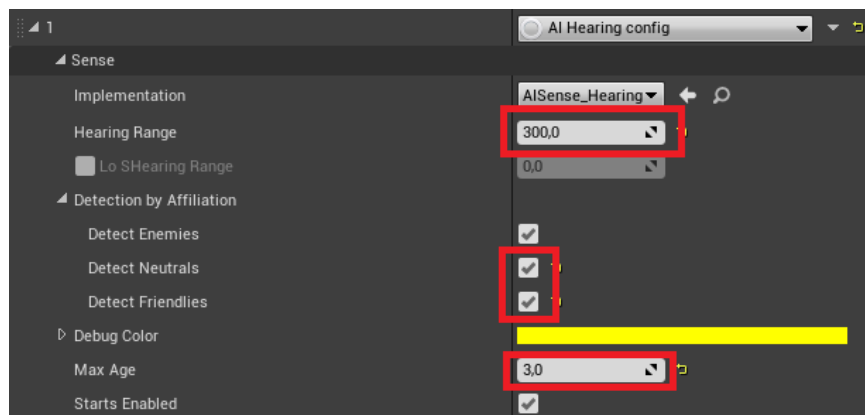
The **Sight** and the **Hearing** senses are the most important senses in the AI perception system.



The **Sight** sense can be configured in the **AI Sight config** section. The **Sight Radius** is the maximum sight distance to notice a target. The **Lost Sight Radius** should be little greater than the **Sight Radius**. The **Detect Neutrals** and the **Detect Friendlies** variables should be always enabled for correct working of the perception system. Also the **Max Age** should be greater than 0.



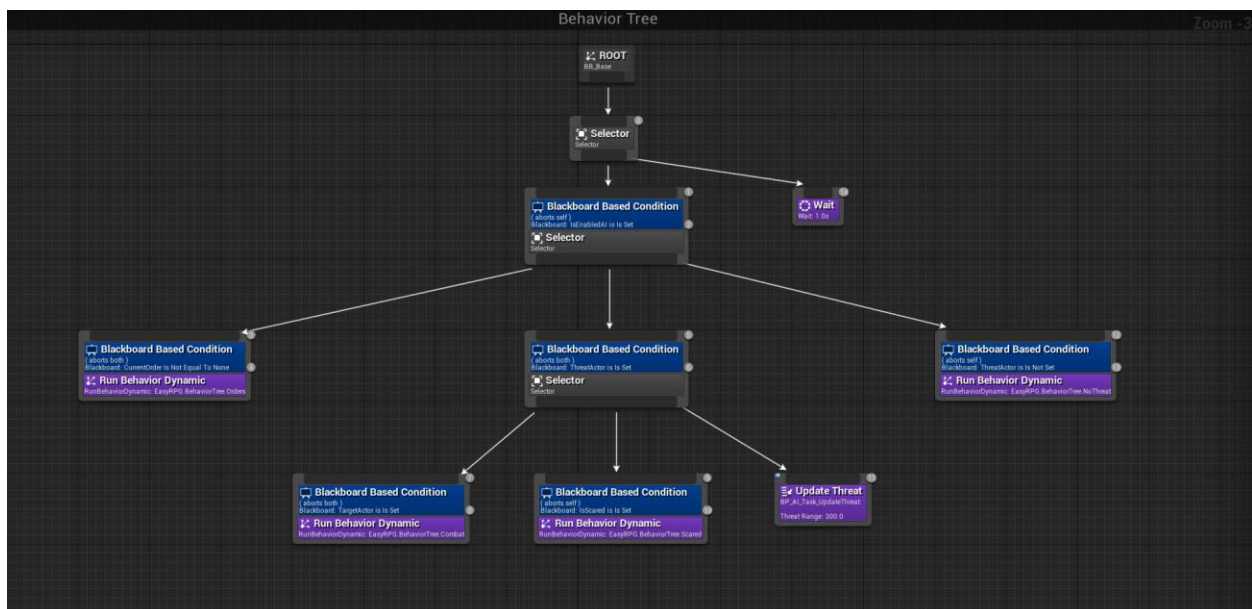
The **Hearing** sense can be configured in the **AI Hearing config** section. The **Hearing range** is the maximum hearing distance to notice a target. The **Detect Neutrals** and the **Detect Friendlies** variables should be always enabled for correct working of the perception system. Also the **Max Age** should be greater than 0.



4.5.6. Behavior trees

Behavior trees are used by AI controllers to control non-playable characters. The base **BT_Base** behavior tree can be used for simple characters like crab or rats, and also for simple melee attack enemies like skeletons, wolves, bears which can chase and attack targets.

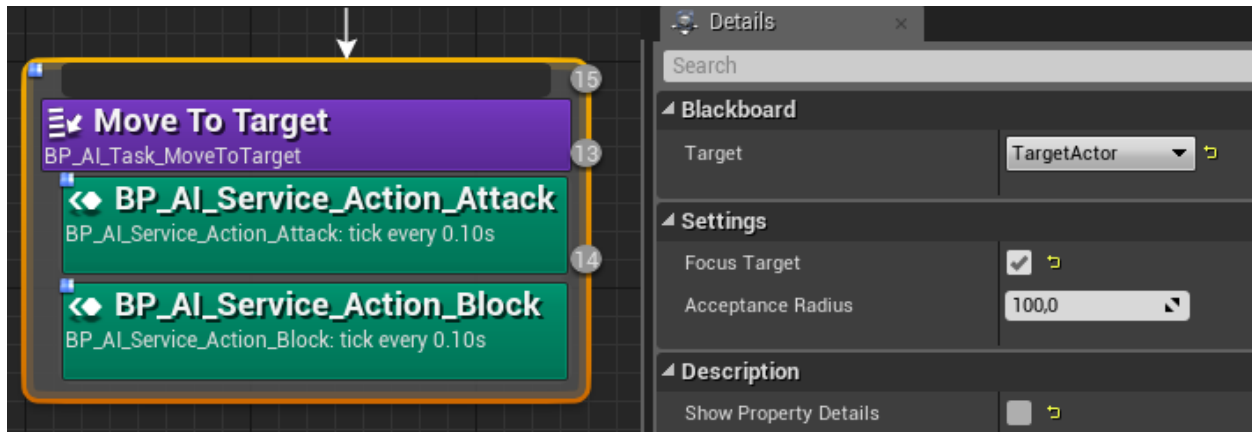
Depending on the state of the **BB_Base** blackboard, the character's behavior is determined.



The character's behavior is chosen depending on whether there is a threat to the character or not. It also checks the orders for the character.

4.5.6.1. Tasks

Tasks are used for different actions that run until they are completed or aborted.



- **BP_AI_Task_MoveToTarget** - the task that makes the character move to target location or to target actor.
- **BP_AI_Task_MoveToRandomLocation** - the task that makes the character move to a random reachable location in a specific radius around the character.
- **BP_AI_Task_MoveStrafe** - the task that makes the character move around the target actor.
- **BP_AI_Task_MoveBack** - the task that makes the character move back from the target actor.
- **BP_AI_Task_PatrolPointMoves** - the task that makes the character move around the patrol point in a specific radius.
- **BP_AI_Task_RunAway** - the task that makes the character move away from the threat when the character **IsScared**.
- **BP_AI_Task_ClearOrder** - the task that clears the current order of the character.
- **BP_AI_Task_UpdateThreat** - the task that checks threats near the character and updates the **ThreatActor** for the character and for the blackboard.
- **BP_AI_Task_TryTaunt** - the task that makes the character play the taunt animation.
- **BP_AI_Task_Chicken_FindFood** - the task that makes the chicken (hen, rooster, chick) character move to the chicken feeder and eat.
- **BP_AI_Task_Rooster_FindHen** - the task that makes the rooster character move to the hen for mating.
- **BP_AI_Task_Pig_FindFood** - the task that makes the pig (pig, swine, piglet) character move to the pig feeder and eat.

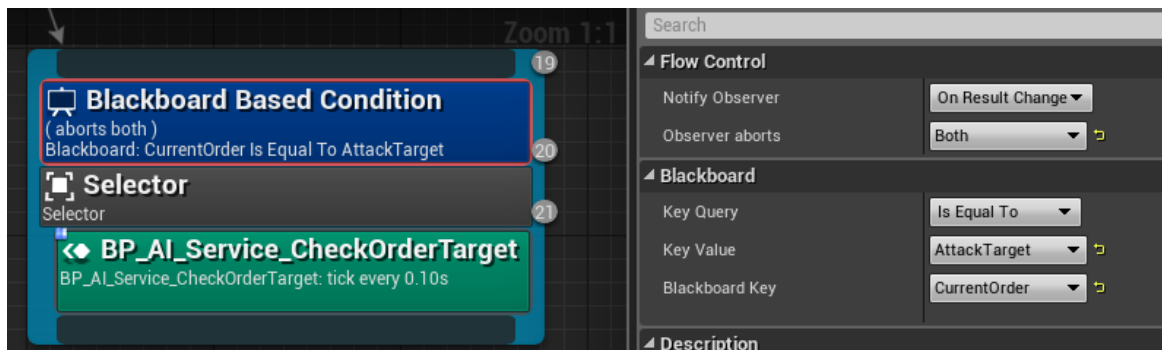
-
- **BP_AI_Task_Pig_FindMud** - the task that makes the pig (pig, swine, piglet) character move to the mud pile and roll in the mud.
 - **BP_AI_Task_Pig_FindCoop** - the task that makes the pig (or swine) character move to the pig coop for mating with swine (or pig).
 - **BP_AI_Task_Swine_FindPig** - the task that makes the swine character move to the pig for mating.
 - **BP_AI_Task_ShouldWalk** - the task that makes the character use the **WalkSpeed** as the maximal speed of the character for movement.
This task is deprecated. The **BP_AI_Service_ShouldWalk** is used instead.
 - **BP_AI_Task_ShouldRun** - the task that makes the character use the **RunSpeed** as the maximal speed of the character for movement.
This task is deprecated. The **BP_AI_Service_ShouldRun** is used instead.
 - **BP_AI_Task_TryAttack** - the task that makes the character try to play attack and continue attack animation.
This task is deprecated. The **BP_AI_Service_Action_Attack** is used instead.
 - **BP_AI_Task_TryBlock** - the task that makes the character try to block enemy attacks. This task is deprecated. The **BP_AI_Service_Action_Block** is used instead.
 - **BP_AI_Task_TryCharge** - the task that makes the character try to charge the bow.
This task is deprecated. The **BP_AI_Service_Action_BowCharge** is used instead.
 - **BP_AI_Task_TryReleaseCharge** - the task that makes the character try to release an arrow from the bow. This task is deprecated.
The **BP_AI_Service_Action_BowRelease** is used instead.
 - **BP_AI_Task_TryShoot** - the task that makes the character try to shoot with current ranged weapon. This task is deprecated.
The **BP_AI_Service_Action_Shoot** is used instead.
 - **BP_AI_Task_TryTaunt** - the task that makes the character try to play taunt animation. This task is deprecated.
 - **BP_AI_Task_PlayAFK** - the task that makes the character try to play AFK animation.
This task is deprecated. The **BP_AI_Service_PlayAFK** is used instead.
 - **BP_AI_Task_Aiming** - the task that makes the character aim to target.
 - **BP_AI_Task_InjectSubBehavior** - the task that injects dynamic sub behavior tree in the main behavior by specific tag.

4.5.6.2. Decorators

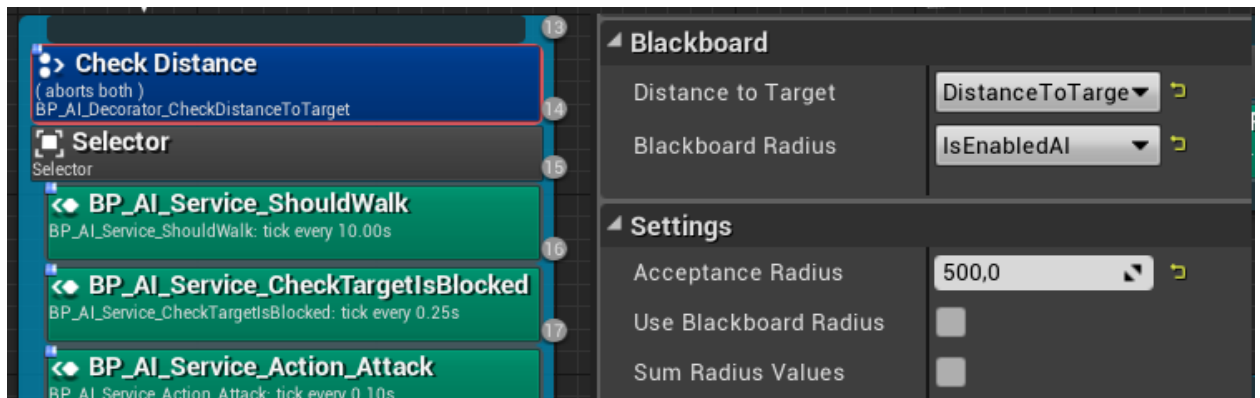
Decorators are used to make specific checks for behavior tree blocks and tasks.

They also can automatically abort decorated blocks or tasks when the decorator result or state is changed.

- **Blackboard Based Condition** - this default decorator is used when needed to check the value of a certain blackboard variable.



- **BP_AI_Decorator_CheckDistanceToTarget** - this decorator is used when needed to check that the blackboard distance to target property is in a specific radius.



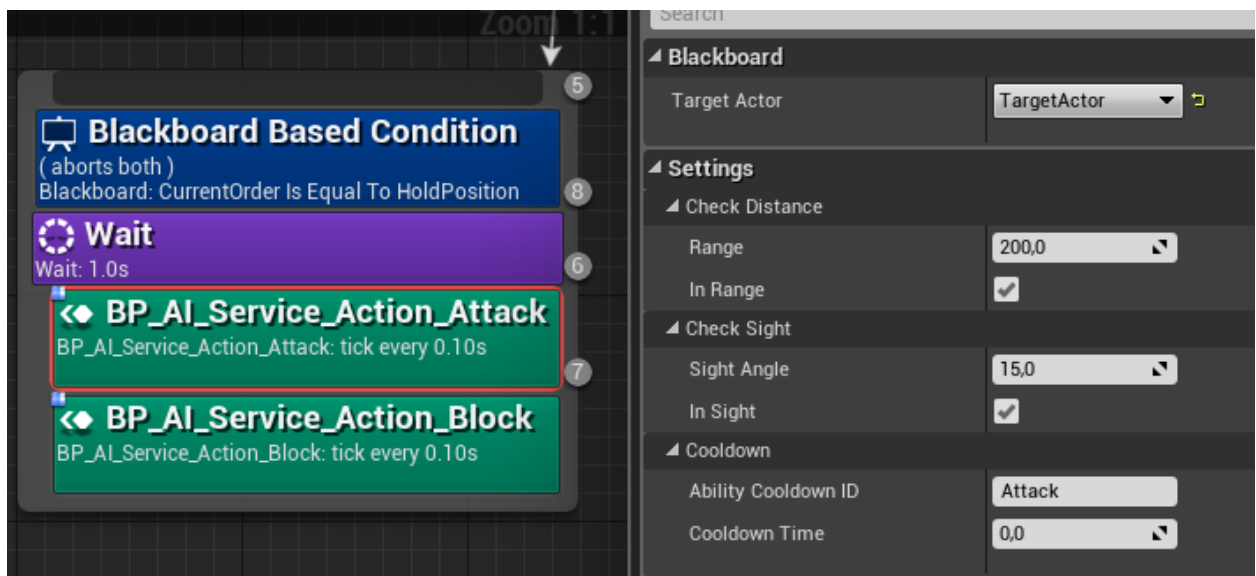
- **BP_AI_Decorator_InRange** - this custom decorator is used for checking range (or out range) to the target actor or to the target location.
- **BP_AI_Decorator_InSight** - this custom decorator is used for checking that the target actor or location is in sight (or out sight) of the character.
- **BP_AI_Decorator_Chicken_FindFood** - the decorator checks that the **NeedToFindFood** variable of the chicken (hen, rooster, chick) character is **True**. This decorator is used only for the **BP_AI_Task_Chicken_FindFood**.

-
- **BP_AI_Decorator_Rooster_FindHen** - the decorator checks that the **NeedToFindHen** variable of the rooster character is **True**.
This decorator is used only for the **BP_AI_Task_Rooster_FindHen**.
 - **BP_AI_Decorator_Pig_FindFood** - the decorator checks that the **NeedToFindFood** variable of the pig (pig, swine, piglet) character is **True**.
This decorator is used only for the **BP_AI_Task_Pig_FindFood**.
 - **BP_AI_Decorator_Pig_FindCoop** - the decorator checks that the **NeedToFindCoop** variable of the pig (pig, swine) character is **True**.
This decorator is used only for the **BP_AI_Task_Pig_FindCoop**.
 - **BP_AI_Decorator_Pig_FindMud** - the decorator checks that the **NeedToFindMud** variable of the pig (pig, swine, piglet) character is **True**.
This decorator is used only for the **BP_AI_Task_Pig_FindMud**.
 - **BP_AI_Decorator_Swine_FindPig** - the decorator checks that the **NeedToFindPig** variable of the swine character is **True**.
This decorator is used only for the **BP_AI_Task_Swine_FindPig**.
 - **BP_AI_Decorator_CheckChase** - the decorator checks the distance between the target actor and the guard location. If the distance is greater than a specific value, then it clears the target. This decorator is deprecated. The **BP_AI_Service_CheckTarget** is used instead.
 - **BP_AI_Decorator_CheckTarget** - the decorator checks that the target actor is alive. If the target is dead, then it clears the target actor. This decorator is deprecated. The **BP_AI_Service_CheckTarget** is used instead.
 - **BP_AI_Decorator_CheckInteractionState** - the decorator checks the interaction state of the target actor.

4.5.6.3. Services

Services are used to make specific checks or actions while behavior tree blocks are active. They are also used for updating blackboard variables on tick.

- **BP_AI_Service_Action_Base** - the base action service that can be used as the parent class for different actions like attacks, blocks, shoots, reloads, super attacks, etc. This service has range, sight and cooldown checks.

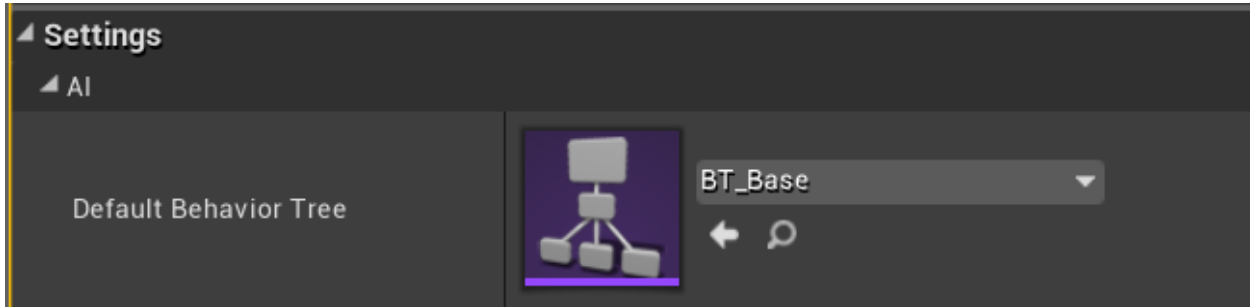


- **BP_AI_Service_Action_Attack** - this service checks the distance to the target actor and checks that the target actor is in sight. If conditions complete then it runs the **TryAttack** function for the character.
- **BP_AI_Service_Action_Block** - this service checks the distance to the target actor and checks that the target actor is in sight. If conditions complete and the target attacks the character then it runs the **TryBlock** function for the character.
- **BP_AI_Service_Action_BowCharge** - this service checks the distance to the target actor and checks that the target actor is in sight. If conditions complete then it runs the **TryCharge** function for the character.
- **BP_AI_Service_Action_BowRelease** - this service checks the distance to the target actor and checks that the target actor is in sight. If conditions complete then it runs the **TryRelease** function for the character.

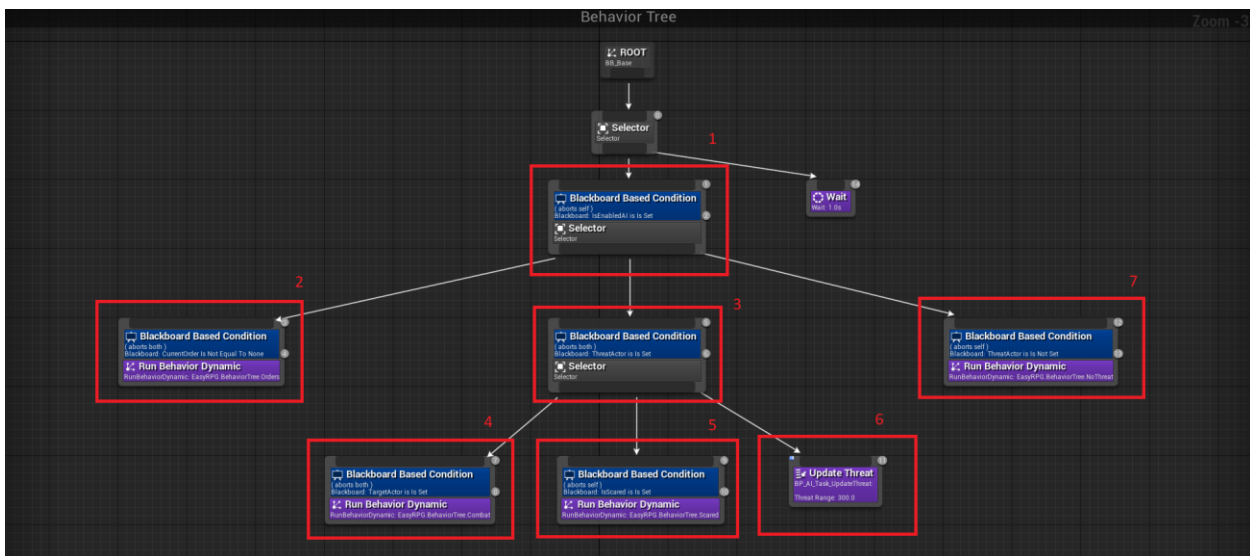
-
- **BP_AI_Service_Action_Shoot** - this service checks the distance to the target actor and checks that the target actor is in sight. If conditions complete then it runs the **TryShoot** function for the character.
 - **BP_AI_Service_Action_Reload** - this service checks that the current ranged weapon of the character is empty . If conditions complete then it runs the **TryReload** function for the character.
 - **BP_AI_Service_CheckOrderTarget** - this service checks that the order target actor is alive. If the target is dead, it clears the current order.
 - **BP_AI_Service_CheckTarget** - this service checks that the target is alive and in chasing distance. If the target is dead or out of distance, it clears the current target.
 - **BP_AI_Service_CheckTargetIsBlocked** -this service checks that the target is not blocked by obstacles. It updates the **TargetIsBlocked** blackboard variable.
 - **BP_AI_Service_ChickWalk** - this service is used only in the chick behavior tree and plays special root motion chick walk animation.
 - **BP_AI_Service_PlayAFK** - this service plays the AFK animation for the character.
 - **BP_AI_Service_ShouldRun** - this service calls the **ShouldRun** function for the character. This function sets the max walk speed of the movement component to the **RunSpeed** variable value.
 - **BP_AI_Service_ShouldWalk** - this service calls the **ShouldWalk** function for the character. This function sets the max walk speed of the movement component to the **WalkSpeed** variable value.
 - **BP_AI_Service_UpdateGuardLocation** - this service updates the **GuardLocation** blackboard variable in the blackboard.
 - **BP_AI_Service_UpdateTargetIsAttacking** - this service updates the **TargetIsAttacking** blackboard variable in the blackboard.
 - **BP_AI_Service_UpdateDistanceToTarget** - this service updates the **DistanceToTarget** blackboard variable in the blackboard.

4.5.6.4. Main character behavior

The main character behavior can be set in the **DefaultBehaviorTree** variable of the character.



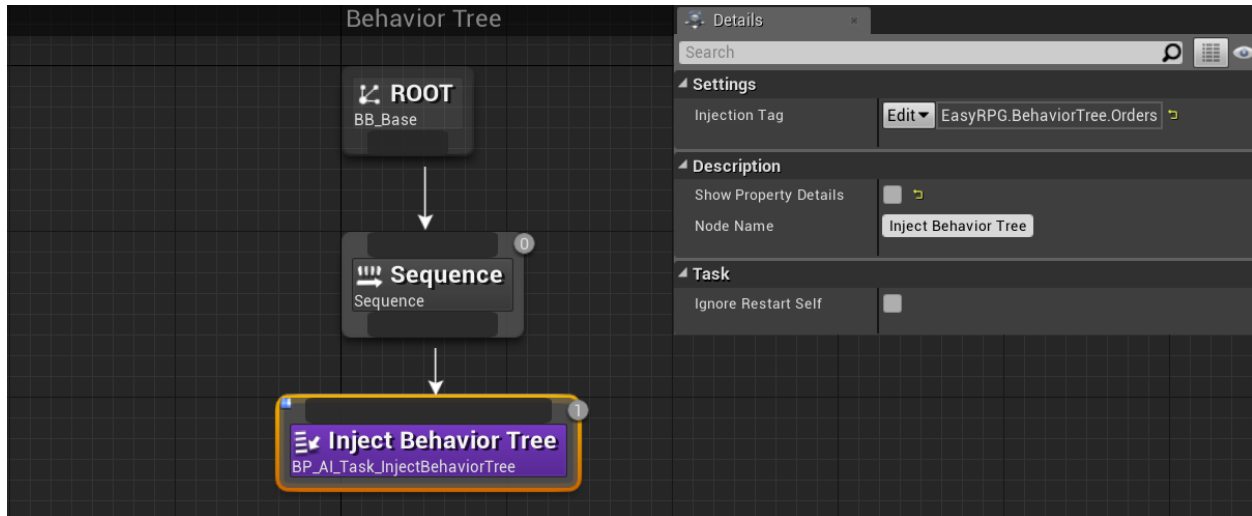
The **BT_Base** behavior tree is used as base character behavior for common characters. It contains dynamic trees which can be replaced with sub behavior trees from the character class at runtime.



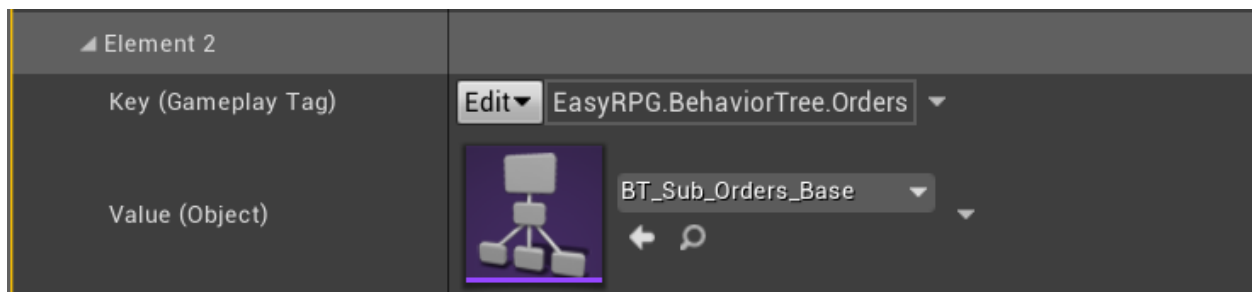
1. Checks that AI enabled and selects one of the behaviors (**2,3,7**) if so.
2. Checks that the character has active order and runs an **Orders** dynamic behavior tree if so.
3. Checks that the character has **Threat** and selects one of the behaviors (**4,5,6**) if so.
4. Checks that **Threat** is **Target** and runs a **Combat** dynamic behavior tree if so.
5. Checks that the character is scared and runs a **IsScared** dynamic behavior tree if so.
6. Checks threats and aborts current behavior if there are no threats nearby.
7. Check that the character has no **Threat** and runs **NoThreat** dynamic behavior if so.

4.5.6.5. Dynamic orders behavior

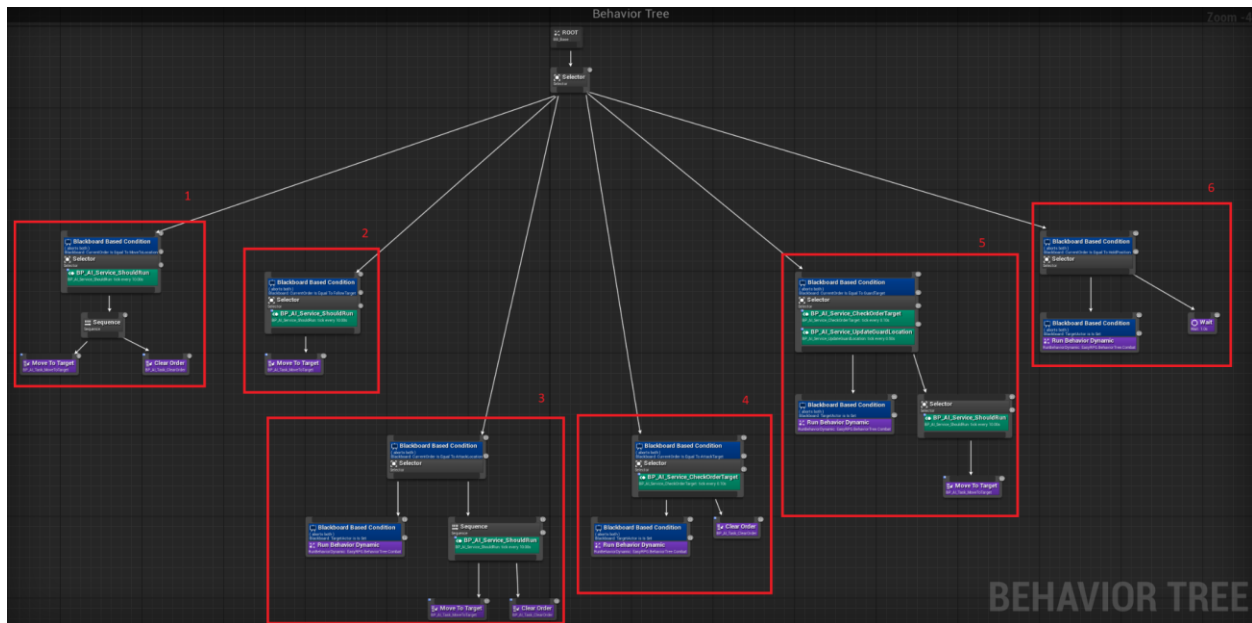
The **BT_Sub_Dynamic_Orders** behavior tree is used to inject another behavior tree into the main behavior tree.



The dynamic behavior tree tries to inject behavior by the **Orders** injection tag. The appropriate sub behavior should be configured for the tag in the **SubBehaviorTrees** variable of the character.



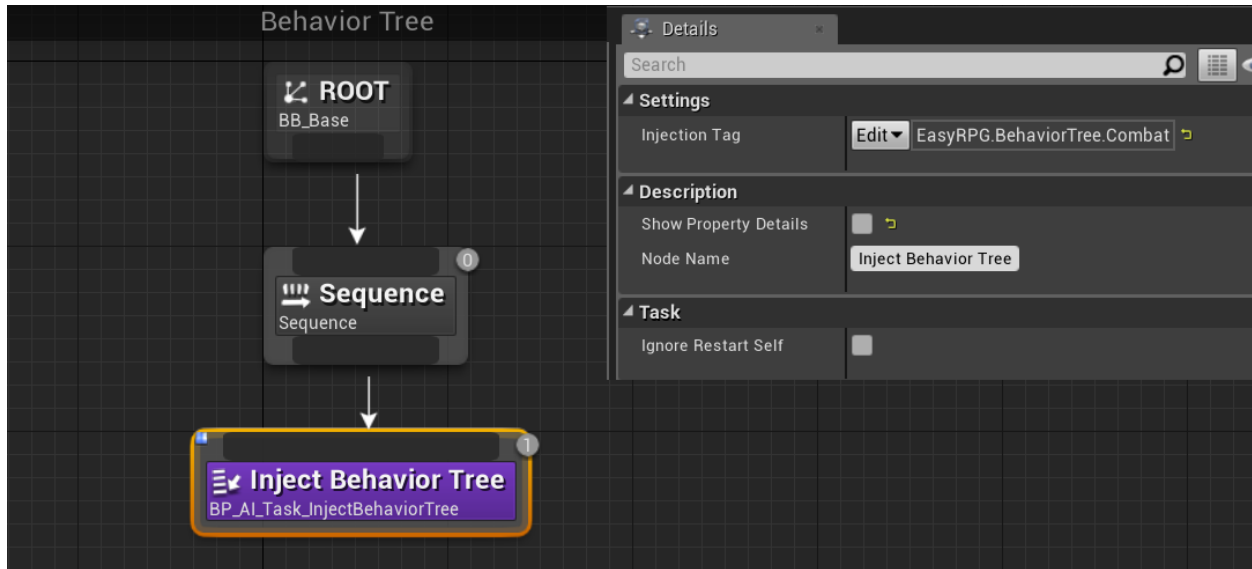
The **BT_Sub_Orders_Base** behavior tree is used as a base orders behavior tree.



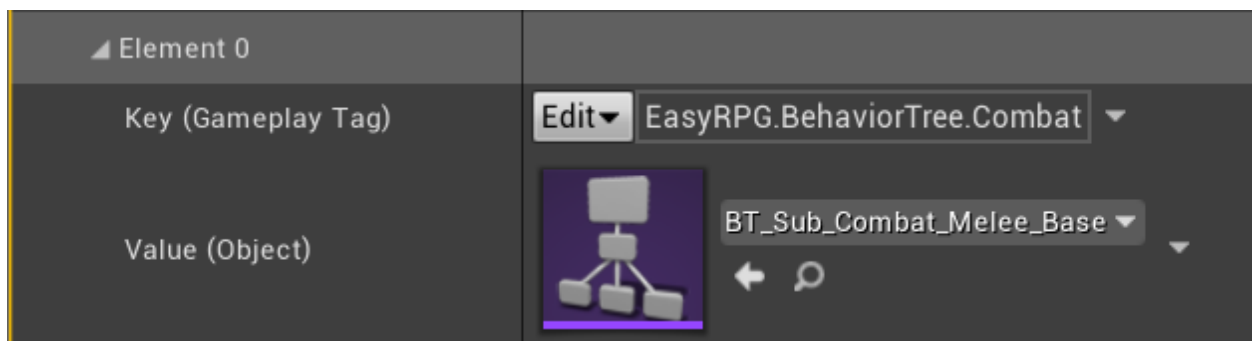
1. Checks that the active order is **MoveToLocation**. Makes the character move to the target location.
2. Checks that the active order is **FollowTarget**. Makes the character follow the target actor.
3. Checks that the active order is **AttackLocation**. Makes the character move to the target location. If the character detects enemy targets it runs a **Combat** dynamic behavior tree.
4. Checks that the active order is **AttackTarget**. Makes the character move to the target actor. If the character detects the target it runs a **Combat** dynamic behavior tree. Other threats will be ignored.
5. Checks that the active order is **GuardTarget**. Makes the character follow the target actor. If the character detects enemy targets it runs a **Combat** dynamic behavior tree.
6. Checks that the active order is **HoldPosition**. Makes the character hold position. If the character detects enemy targets it runs a **Combat** dynamic behavior tree. The character won't chase targets.

4.5.6.6. Dynamic combat behavior

The **BT_Sub_Dynamic_Combat** behavior tree is used to inject another behavior tree into the main behavior tree.



The dynamic behavior tree tries to inject behavior by the **Combat** injection tag. The appropriate sub behavior should be configured for the tag in the **SubBehaviorTrees** variable of the character.

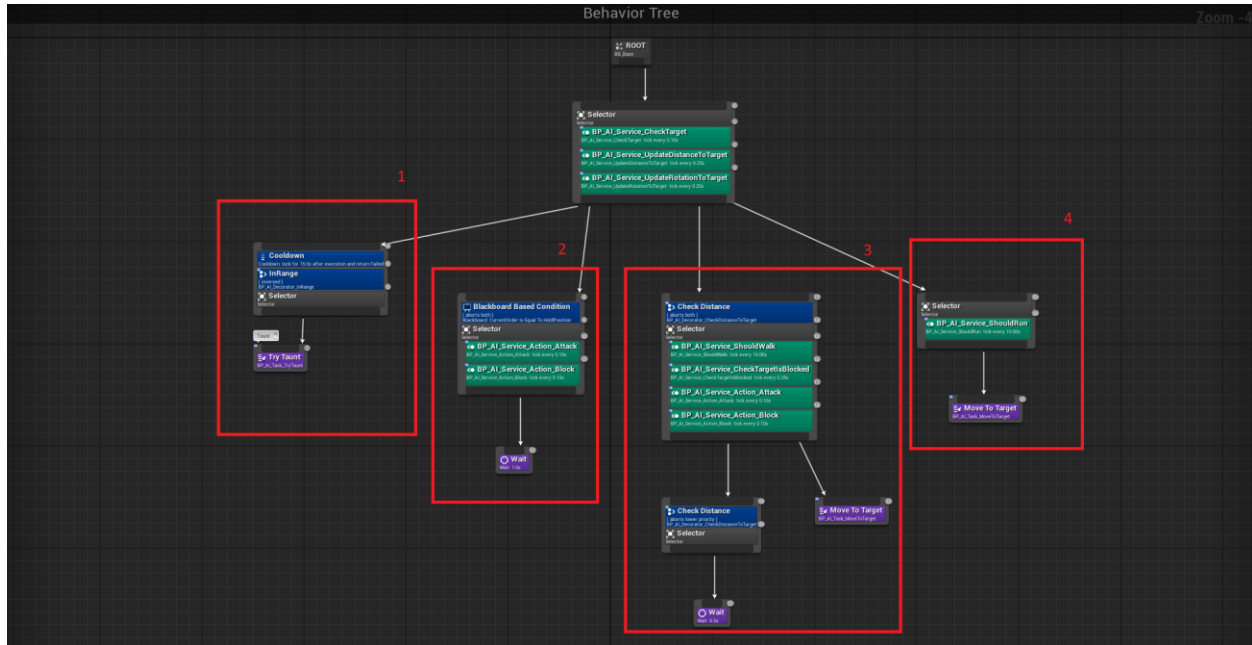


The **BT_Sub_Combat_Melee_Base** behavior tree is used as a base combat behavior tree for characters which use melee attacks and actions.

The **BT_Sub_Combat_Ranged_Base** behavior tree is used as a base combat behavior tree for characters which use firearm ranged attacks and actions.

The **BT_Sub_Combat_Bow_Base** behavior tree is used as a base combat behavior tree for characters which use bow ranged attacks and actions.

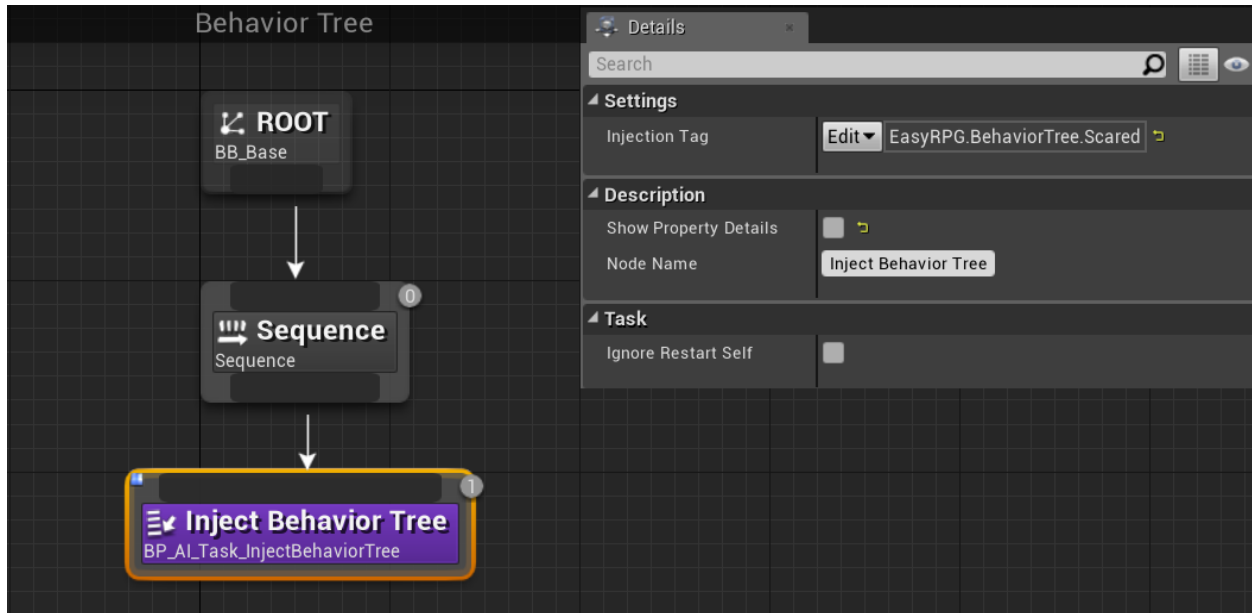
If the character **IsAggressive** and **CanAttack**, the threat becomes a target and the character starts chasing it until it dies. The chase stops if the target runs far away from the character at a certain distance or he leaves the **GuardLocation** too far.



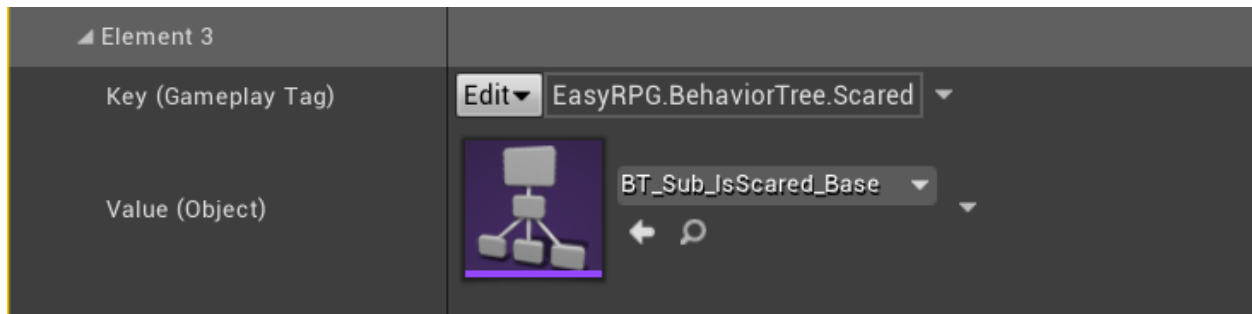
1. Checks cooldown and plays the **Taunt** animation if the target is in a specific range.
2. Checks that the character has **HoldPosition** order. Makes the character to hold position and attack targets whenever it is possible.
3. Checks that target in a specific range and turns the character to walking from running. The character attacks enemy targets whenever it is possible.
4. Makes the character move to the target actor. The character is running in this state.

4.5.6.7. Dynamic scared behavior

The **BT_Sub_Dynamic_IsScared** behavior tree is used to inject another behavior tree into the main behavior tree.

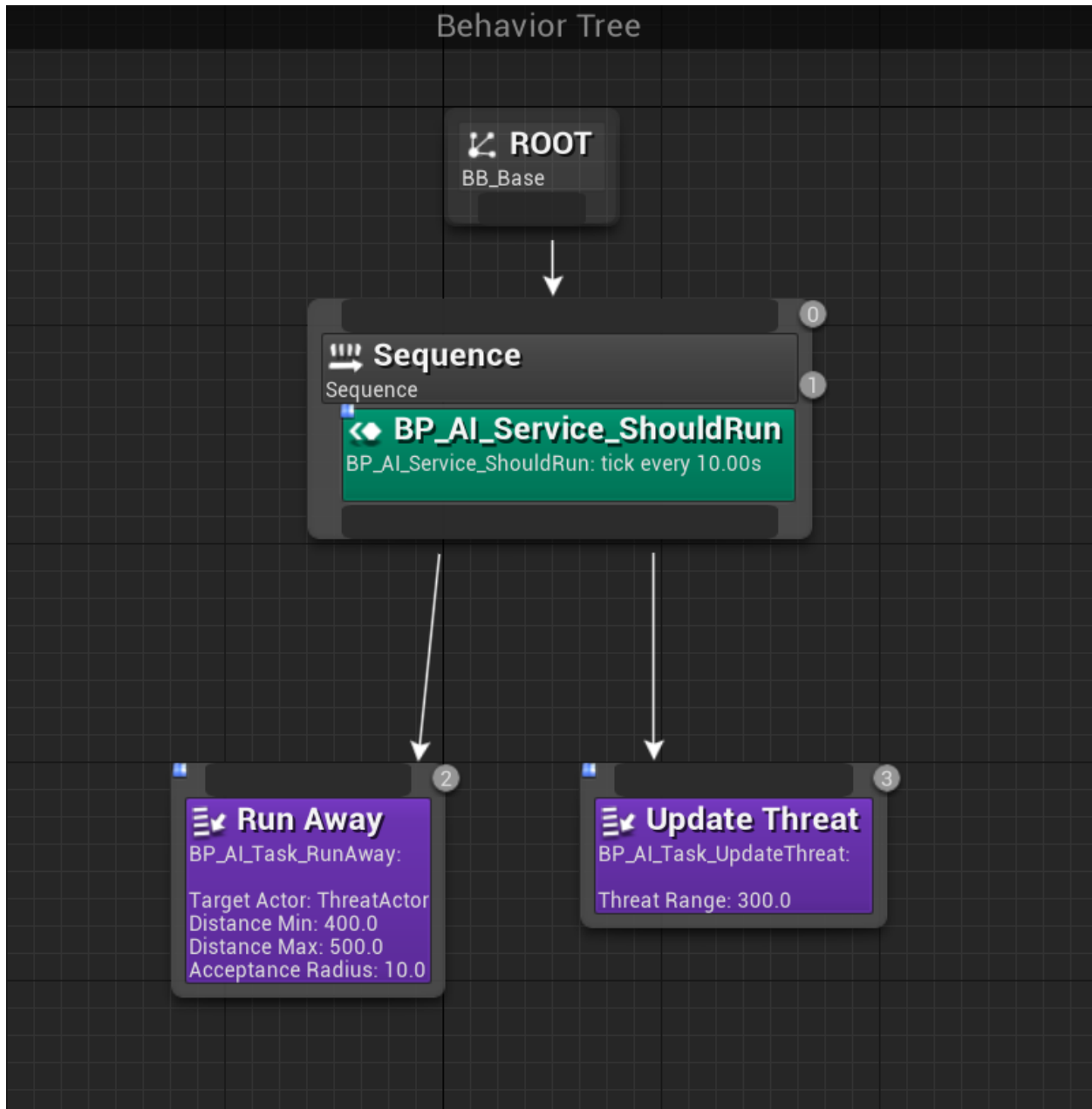


The dynamic behavior tree tries to inject behavior by the **Scared** injection tag. The appropriate sub behavior should be configured for the tag in the **SubBehaviorTrees** variable of the character.



The **BT_Sub_IsOrders_Base** behavior tree is used as a base orders behavior tree.

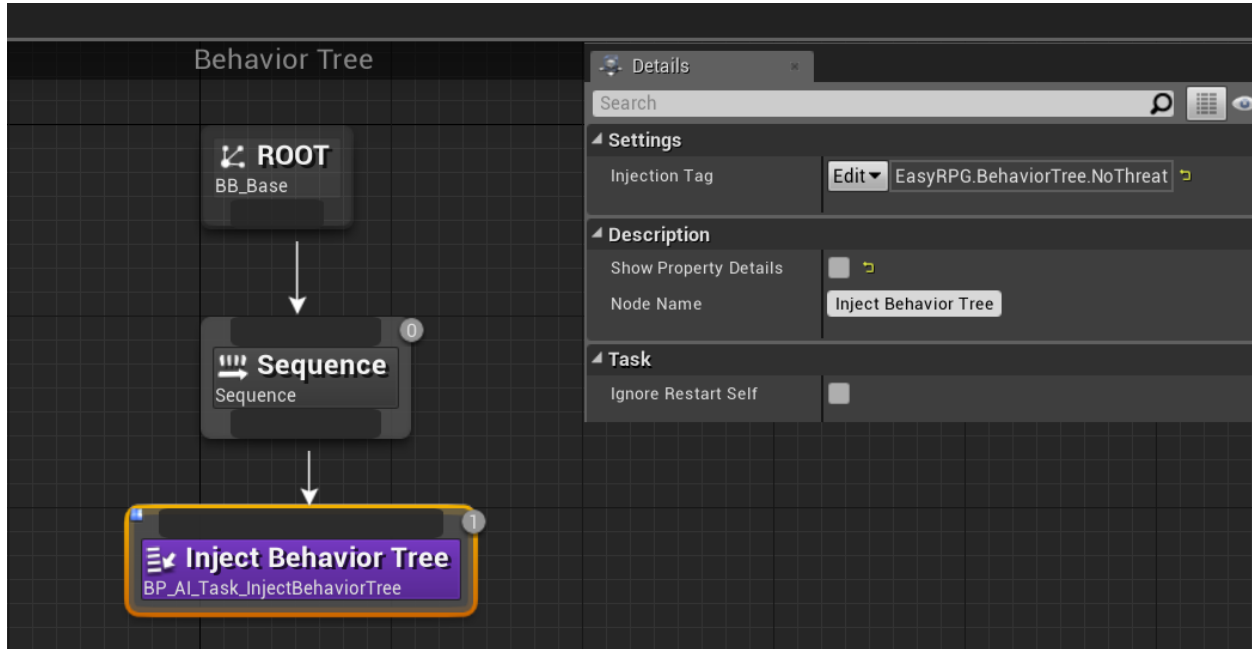
If the character **CanNotAttack** and **CanBeScared**, he starts to run away from the threat for a certain distance.



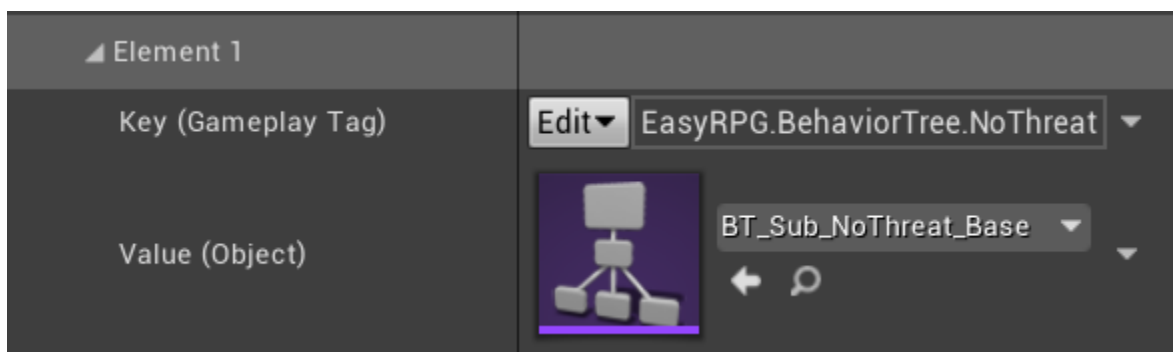
The behavior makes the character run away from threat. Then checks threats and aborts current behavior if there are no threats nearby.

4.5.6.8. Dynamic calm behavior

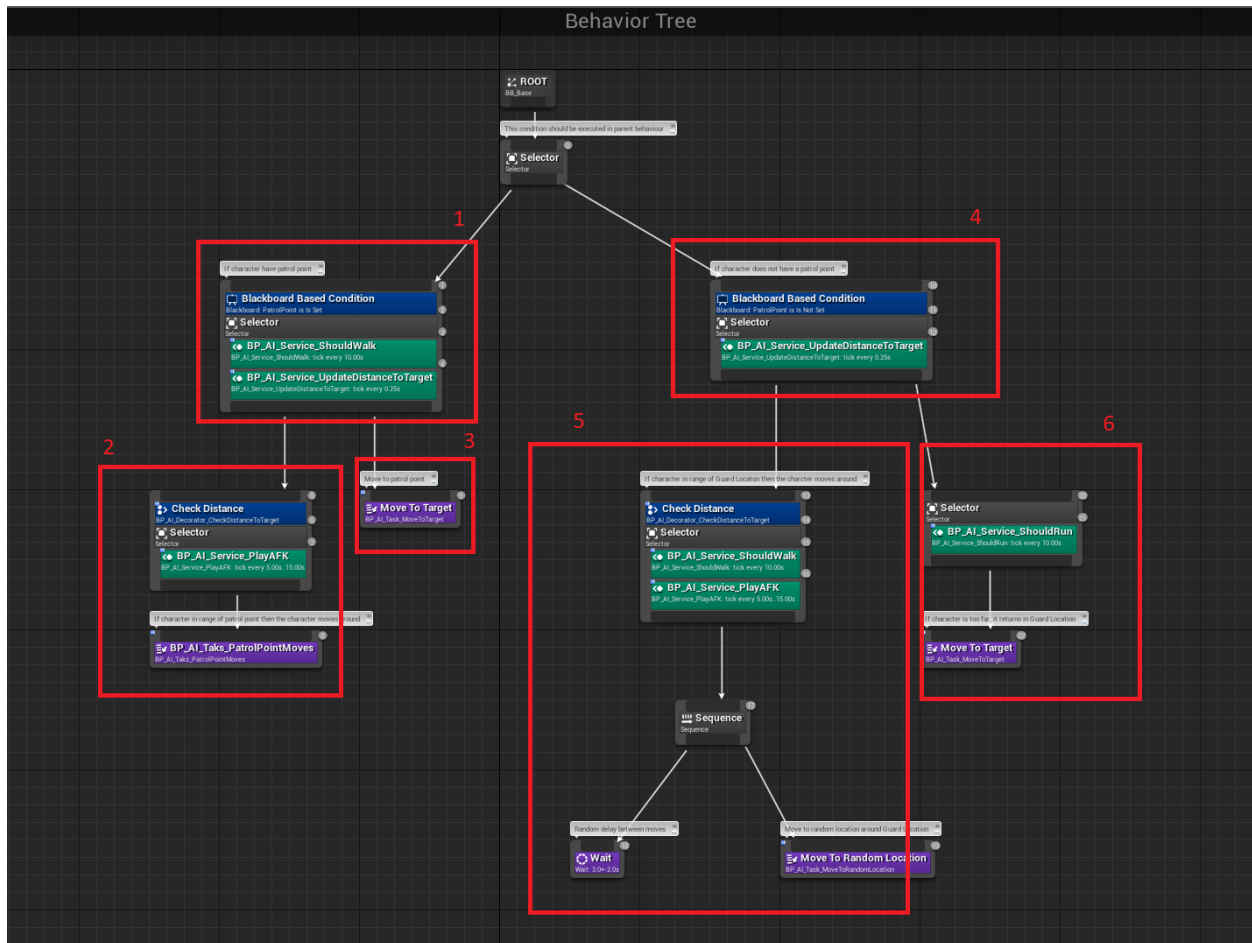
The **BT_Sub_Dynamic_NoThreat** behavior tree is used to inject another behavior tree into the main behavior tree.



The dynamic behavior tree tries to inject behavior by the **NoThreat** injection tag. The appropriate sub behavior should be configured for the tag in the **SubBehaviorTrees** variable of the character.



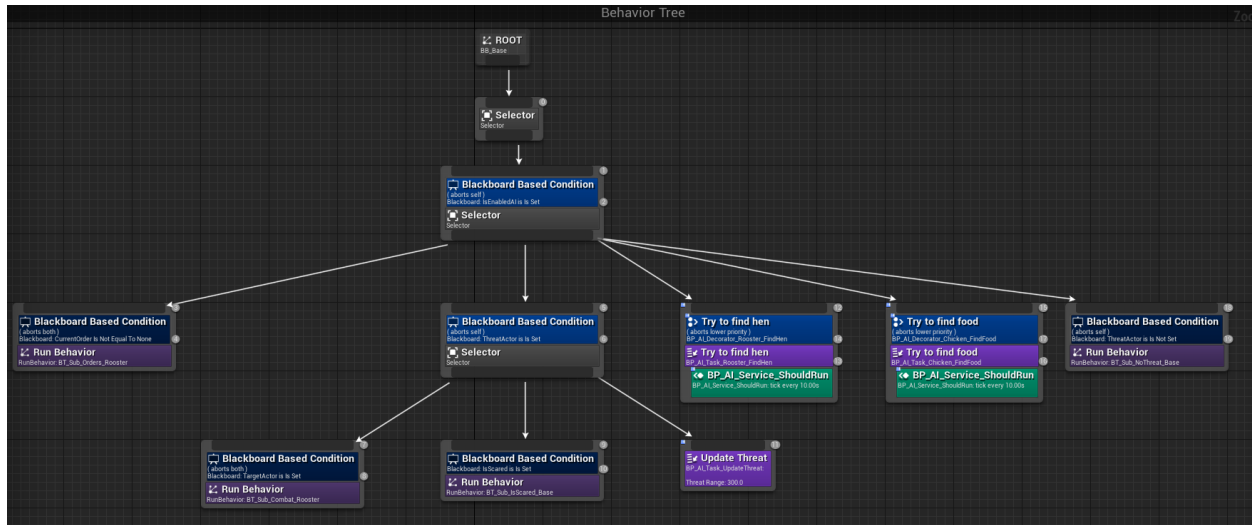
The **BT_Sub_NoThreat_Base** behavior tree is used as base calm behavior.



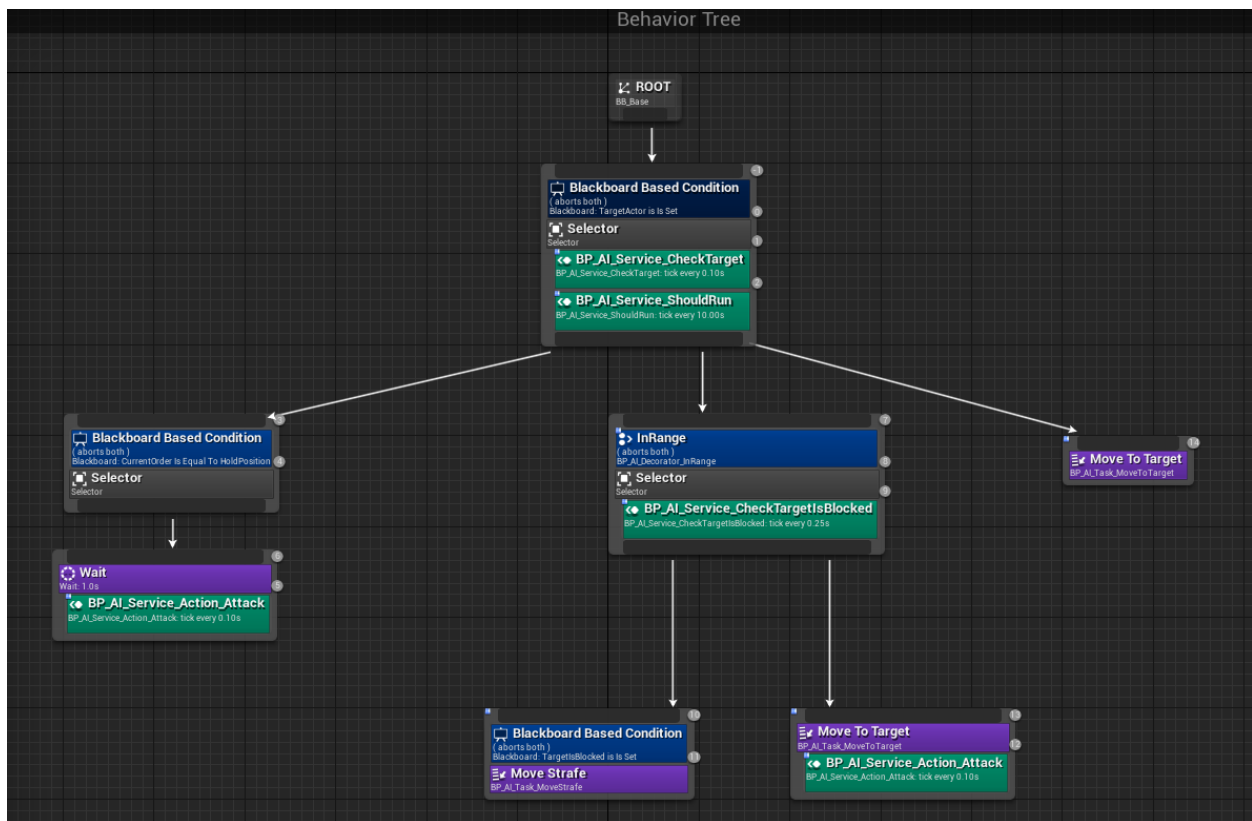
1. Check that the character has **PatrolPoint** and selects one of the behaviors (2,3).
2. Check that character in range of the **PatrolPoint**.
Makes the character move around.
3. Makes the character move to the **PatrolPoint** if the character is too far away from it.
4. Checks that the character has no **PatrolPoint** and selects one of the behaviors (5,6).
5. Checks that the character is in range of the **GuardLocation** and makes the character move around it.
6. Makes the character move to the **GuardLocation** if the character is too far away from it.

4.5.6.9. Unique behaviors

Complex characters can use advanced behavior trees. Such trees use modified base sub trees. For example, the rooster behavior tree has blocks for finding food and for matting.



The **BT_Rooster** also has modified combat and orders trees.



5. Project settings

5.1. Collision settings

Several types of collisions have been added for different project systems.

Collision settings can be found in **Project Settings** under **Collision**.

Engine - Collision

Set up and modify collision settings.

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

Object Channels

You can have up to 18 custom channels including object and trace channels. This is list of object type for your project. If you delete the object type that has been used by game, it will go back to WorldStatic.

Name	Default Response
StaticFoliage	Ignore
FoliageChecker	Ignore
DynamicFoliage	Ignore
SurfaceOverride	Ignore
Water	Ignore
Projectile	Ignore
SpawnPoint	Ignore
SpawnChecker	Ignore

Engine - Collision

Set up and modify collision settings.

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

Object Channels

Trace Channels

You can have up to 18 custom channels including object and trace channels. This is list of trace channel for your project. If you delete the trace channel that has been used by game, the behavior of trace is undefined.

Name	Default Response
WaterTrace	Ignore

Engine - Collision

Set up and modify collision settings.

Set as DefaultExportImportReset to Defaults

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

Object Channels

Trace Channels

Preset

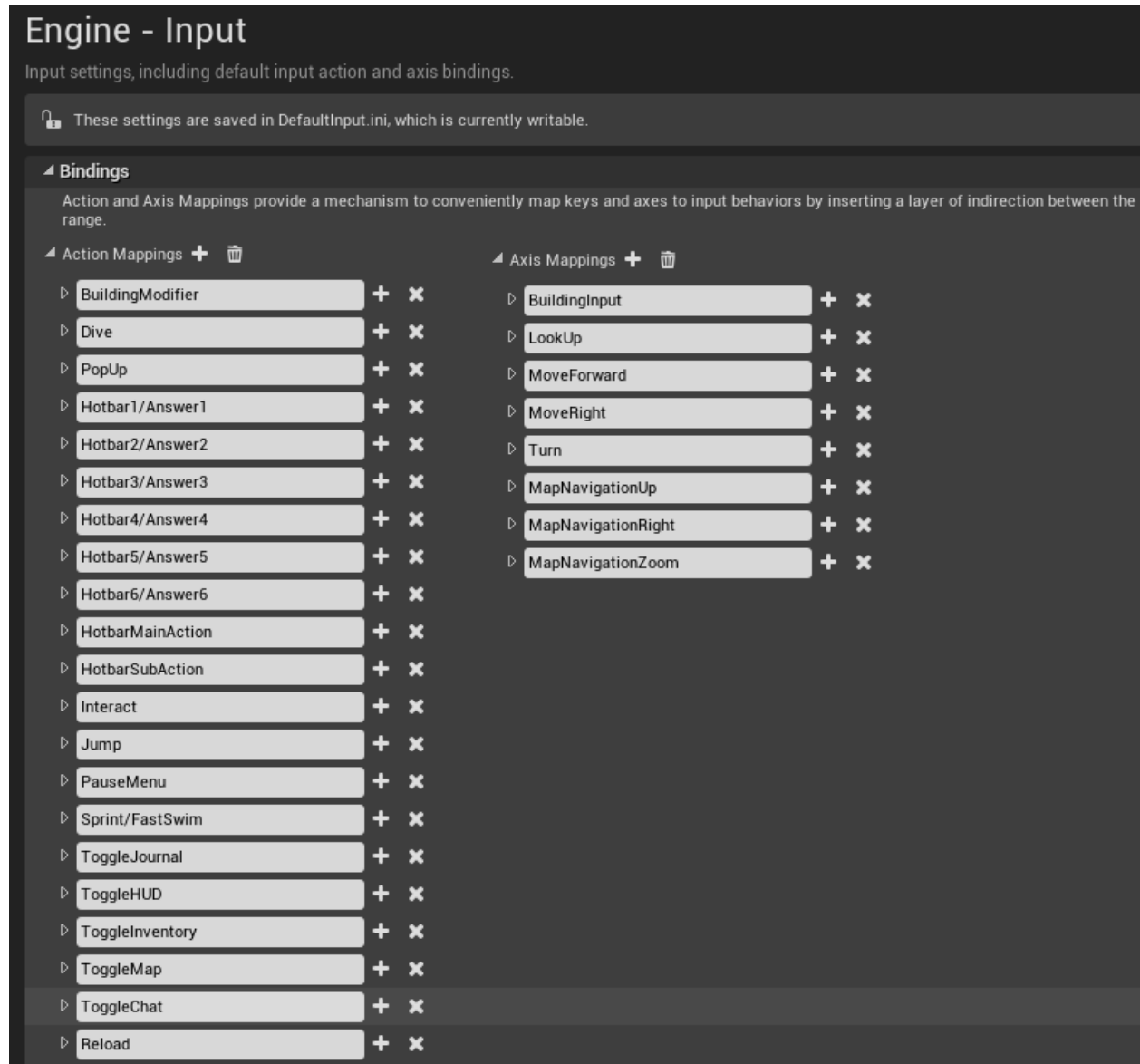
You can modify any of your project profiles. Please note that if you modify profile, it can change collision behavior. Please be careful when you change currently existing (used) collision profiles.

NewEditDelete

Name	Collision	Object Type	Description
NoCollision	No Collision	WorldStatic	No collision
BlockAll	Collision Enabled (Query and Physics)	WorldStatic	WorldStatic object that blocks all actors by default. All new custom channels will use its own default response.
OverlapAll	Query Only (No Physics Collision)	WorldStatic	WorldStatic object that overlaps all actors by default. All new custom channels will use its own default response.
BlockAllDynamic	Collision Enabled (Query and Physics)	WorldDynamic	WorldDynamic object that blocks all actors by default. All new custom channels will use its own default response.
OverlapAllDynamic	Query Only (No Physics Collision)	WorldDynamic	WorldDynamic object that overlaps all actors by default. All new custom channels will use its own default response.
IgnoreOnlyPawn	Query Only (No Physics Collision)	WorldDynamic	WorldDynamic object that ignores Pawn and Vehicle. All other channels will be set to default.
OverlapOnlyPawn	Query Only (No Physics Collision)	WorldDynamic	WorldDynamic object that overlaps Pawn, Camera, and Vehicle. All other channels will be set to default.
Pawn	Collision Enabled (Query and Physics)	Pawn	Pawn object. Can be used for capsule of any playable character or AI.
Spectator	Query Only (No Physics Collision)	Pawn	Pawn object that ignores all other actors except WorldStatic.
CharacterMesh	Collision Enabled (Query and Physics)	Pawn	Pawn object that is used for Character Mesh. All other channels will be set to default.
PhysicsActor	Collision Enabled (Query and Physics)	PhysicsBody	Simulating actors
Destructible	Collision Enabled (Query and Physics)	Destructible	Destructible actors
InvisibleWall	Collision Enabled (Query and Physics)	WorldStatic	WorldStatic object that is invisible.
InvisibleWallDynamic	Collision Enabled (Query and Physics)	WorldDynamic	WorldDynamic object that is invisible.
Trigger	Query Only (No Physics Collision)	WorldDynamic	WorldDynamic object that is used for trigger. All other channels will be set to default.
RigidBody	Collision Enabled (Query and Physics)	PhysicsBody	Simulating Skeletal Mesh Component. All other channels will be set to default.
Vehicle	Collision Enabled (Query and Physics)	Vehicle	Vehicle object that blocks Vehicle, WorldStatic, and WorldDynamic. All other channels will be set to default.
UI	Query Only (No Physics Collision)	WorldDynamic	WorldStatic object that overlaps all actors by default. All new custom channels will use its own default response.
StaticFoliage	Collision Enabled (Query and Physics)	StaticFoliage	Needs description
FoliageChecker	Collision Enabled (Query and Physics)	FoliageChecker	Needs description
DynamicFoliage	Collision Enabled (Query and Physics)	DynamicFoliage	Needs description
SurfaceOverride	Query Only (No Physics Collision)	SurfaceOverride	Needs description
Projectile	Collision Enabled (Query and Physics)	Projectile	Needs description
SpawnPoint	Query Only (No Physics Collision)	SpawnPoint	Needs description
SpawnChecker	Query Only (No Physics Collision)	SpawnChecker	Needs description

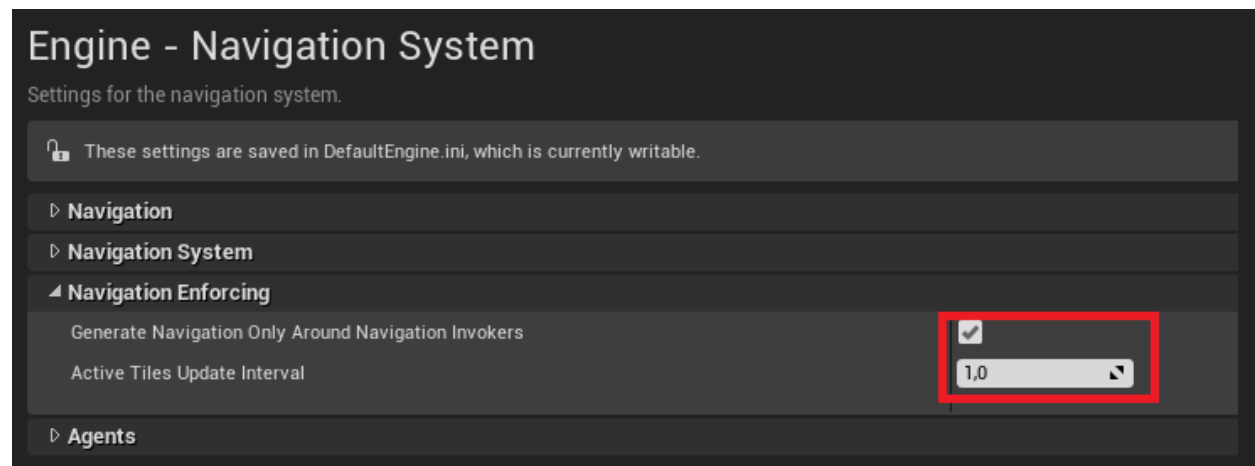
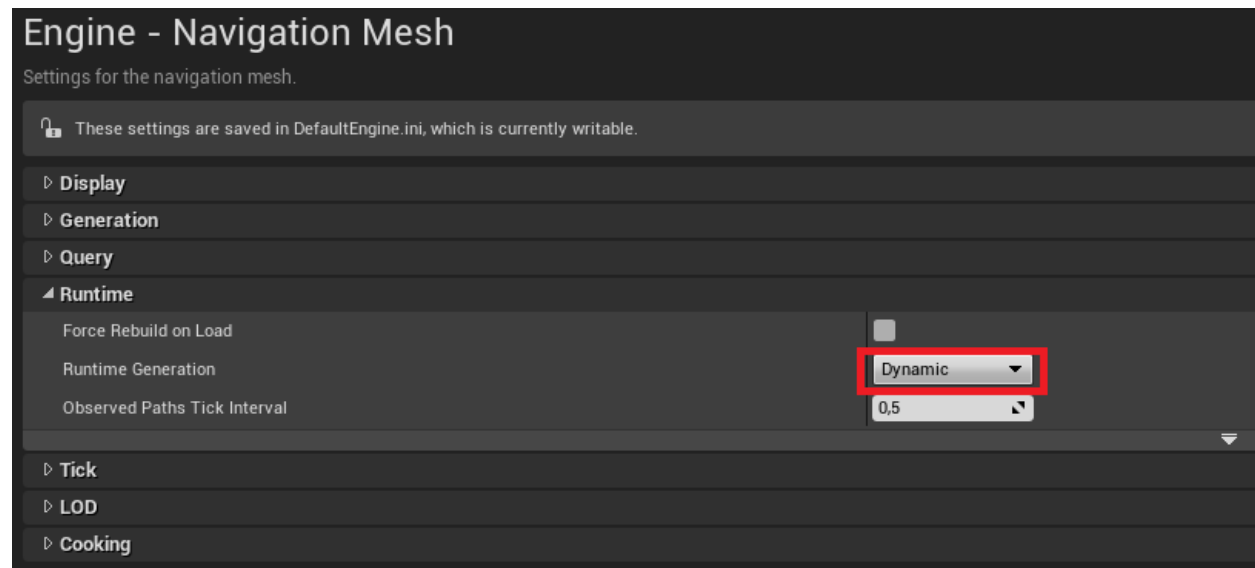
5.2. Control settings

Control button settings can be found in **Project Settings** under **Input**.



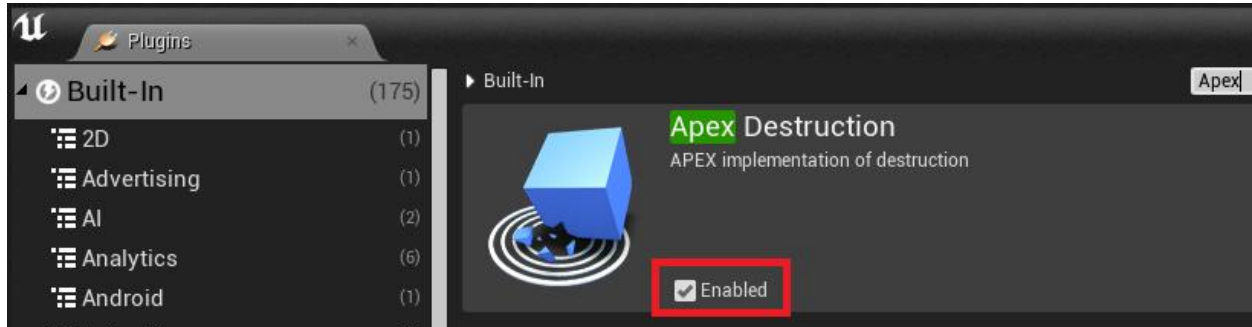
5.3. AI navigation settings

Settings for dynamically updating the navigation mesh can be found in the **Project Settings** under the **Navigation Mesh** and under the **Navigation System**.



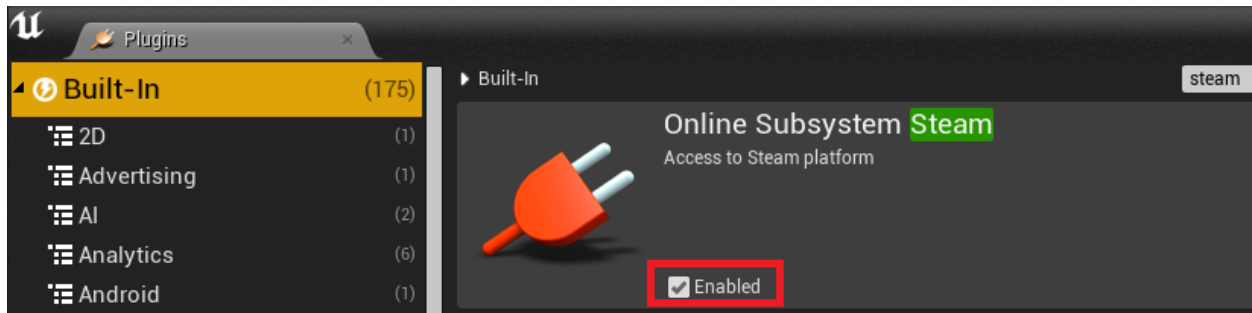
5.4. Plugin settings

For the project to work, the **Apex Destruction** plugin must be enabled.



5.5. STEAM settings

To search for sessions through the **STEAM** platform, you must enable the **Online Subsystem Steam** plugin.



You also need to insert the following lines into the **Config / DefaultEngine.ini** file:

```
[/Script/Engine.GameEngine]
```

```
+NetDriverDefinitions=(DefName="GameNetDriver",DriverClassName="OnlineSubsystemSteam.SteamNetDriver",DriverClassNameFallback="OnlineSubsystemUtils.IpNetDriver")
```

```
[OnlineSubsystem]
```

```
DefaultPlatformService=Steam
```

```
[OnlineSubsystemSteam]
```

```
bEnabled=true
```

```
SteamDevAppId=480
```

```
[/Script/OnlineSubsystemSteam.SteamNetDriver]
```

```
NetConnectionClassName="OnlineSubsystemSteam.SteamNetConnection"
```