Advanced Framework - Core 4.1 Documentation

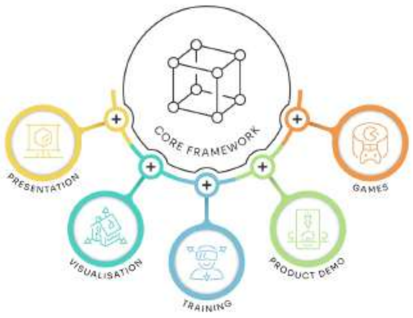
HumanCodeable

May 5, 2021

Contents

1 Introduction

Version 4.0 of the Advanced Framework brings some major innovations compared to older versions. Responding to various user issues with the convoluted version 3.1 we decided to split the Framework into a slim core and a number of extensions, that can be purchased separately. The Core Framework can be used on its own or combined with a selection of Advanced Framework Extensions which provide additional functionalities and examples tailored to a specific type of application. Up until now five AF Extensions are planned for the following use cases:



- Presentation - provides the tools and environment to hold lectures or record videos including a fully set up class room level and studio level as examples.
- Arch Viz - summarizes all functionalities to set up, show and modify architectural visualizations from a room to a whole building.
- Showcase - focuses on one complex actor, that can be equipped with a myriad of functionalities for customization and interaction.
- Training - contains tools and examples to set up training simulations.
- Games - provides a selection of useful assets to create video games and game like experiences

Additionally a number of multi-use assets like the smart watch or the keyboard were separated in form of Advanced Framework Utilities. AF Utilities can be freely combined with the Core Framework and any AF Extension. Up until now the following AF Utilities are planned:

- Smart Watch - equips the VR pawn with a watch-like UI that displays information and controls the display UI element.
- Keyboard - provides an actor that can be used as keyboard in the application
- Paintbrush - provides an actor that can create multicolored splines
- Minimaps - components, materials and UI to setup and display 2D and 3D minimaps including minimaps with multiple floors.
- Webbrowser - UI and functions to access and display webpages

Splitting the Framework into a Core, Extensions and Utilities enables everyone to purchase their own modification of the Framework that is adapted to their specific use case. Also it allows us to include many more, hand made, ready to use examples to facilitate using the Framework even for less experienced users.

1.1 The Core Framework

The Core Framework provides all basic functionalities. Neither AF Extensions nor AF Utilities can function without the AF Core. The AF Core provides:

- over 20 components to implement all kinds of functionalities.
- pawns adjusted to VR-, desktop- and mobile environments.
- various UI elements including a HUD, menus and pallets
- all basic game- and info-files, helpers, managers and widgets
- 11 example maps to showcase the basic tools of the AF Core

1.2 File Structure - Core Framework

The AF Core has a predetermined folder structure which can be the basis for any project. Every new element of your experience can be integrated unequivocally into the structure to keep even large projects neat and easy to navigate. Most of the folders are self explanatory: the folders named Animations, Materials, Meshes, Particles, etc. will contain their respective file type. Hereafter, you find a list of the folders that are most important in this documentation and what they contain.

Blueprints	contains most of the logic provided by the framework.
Components	contains the custom components of the AF Core. For further information see section 3.
Game	contains all game classes (see section 2) like the game mode or the player controller.
Helpers	contains actors that support components or other actors in their functions. Helpers are spawned automatically and do not need any setup.
Level	contains all files for a basic level setup like the MapInfo Actor or the Post Process Volume for highlighting.
Pawn	contains the pawn parent classes of the AF Core.
UI	contains all UI-actors provided by the Framework including pallets or the info window (see section 8).
VR	contains the motion controller and the motion components as described in section 5.3.3 and 5.3.2.
Widgets	contains all custom made AF Core widgets
Data Assets	contains a number of objects used to store data (section **??**).
Examples	contains all Meshes, Materials and Blueprints shown in the example maps.
Interfaces	contains all interfaces governing information transmission between components or in general.
Libraries	provide a set of universally accessible functions.
Maps	contains map templates.

```
AFCore
├─Animations
├─AnimBP
├─Blueprints
│  ├─Components
│  │  ├─Info
│  │  ├─Interaction
│  │  ├─Misc
│  │  ├─Multiplayer
│  │  ├─Pawn
│  │  ├─Snapping
│  │  ├─State
│  │  └─UI
│  ├─Game
│  ├─Helpers
│  ├─Level
│  ├─Save
│  ├─Pawn
│  ├─UI
│  ├─Utility
│  ├─VR
│  │  ├─MotionComponents
│  │  └─MotionControllers
│  └─Widgets
├─Curves
├─DataAssets
│  ├─Gesture
│  ├─Level
│  ├─Panel
│  └─Presets
├─Datatables
├─Enums
├─Examples
├─Font
├─Interface
├─Libraries
├─Maps
├─Materials
├─Meshes
├─Sound
├─Splash
├─Structs
└─Textures
```

2 Basics

2.1 Game Classes

Game classes implement all basic functions for a application to function. This includes level transitions, pawn navigation, settings and other essential data.

Game Mode

Declares all other game classes except the game instance.
Blueprint: `BP_GameMode_Main`
Settings: should not be touched

In a multiplayer experience the game mode is only accessible by the server, not the clients. Therefore, we stripped a lot of features from the game mode and distributed them to other entities.

Game Instance

The game instance has the unique ability to keep its state even after a level change which makes it useful to store persistent data. Thus, it is the best place to store settings like language, graphics or audio. Additionally the game instance stores the data assets of the current level.
Blueprint: `BP_GameInstance_Main`
Settings: all variables are automatically set when the game instance is created

Player Controller

The player controller spawns, possesses and navigates the pawn and handles the level transition logic as well as the pause.
Blueprint: `BP_Player_Controller_Main`
Settings:

- Pawn Select - determines how the pawn is selected upon starting the application
  - Dynamic - enables the pawn selection faccording to the environment check as shown in Figure 1
  - ForceVR - selects the VR pawn specified in the level data asset not regarding the environment
  - ForceDesktop - selects the desktop pawn specified in the level data asset not regarding the environment
  - ForceMobile - selects the mobile pawn specified in the level data asset not regarding the environment
  - OnlyFirstVR - defines that in multiplayer VR applications only the player with player index 0 is in VR

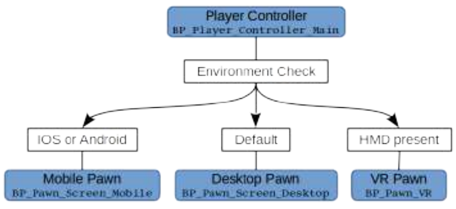


Figure 1. Startup and pawn selection.

Player State

The player state handles and provides information on each player and thus, is especially useful for multiplayer.
Blueprint: `BP_PlayerState_Main`
Settings:

- Player Index - number of the player

All other settings of the player state are outsources now to the playerinfo component which is covered in section 3.7.1

2.2 Level Setup

Each level of the Framework is encapsulated in a world map that consolidates all maps as well as the loading of the level. Additionally, the Framework relies on a number of essential actors in every level to function properly. These need to be placed and setup in every new level.

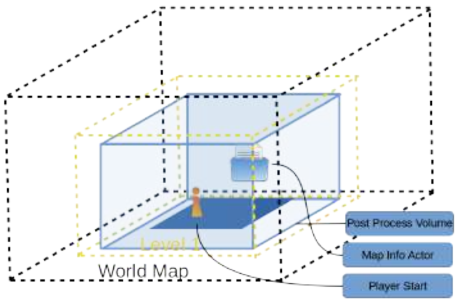


Figure 2. Minimum level setup.

MapInfo Actor

refers to the data asset (DA) containing basic information about each level like which pawns to use or which levels to load.
Blueprint: `BP_MapInfo`
Setup:

- Level DA - refers to the level data asset (see section 7.1.2)

Player Position

specifies the location and rotation where the player pawn is spawned upon starting the level.
Blueprint: `BP_PlayerPosition`
Setup

- **Player Index** - limits the player pawns spawned here to one player index
- **Default** - converts the player position in the default spawn point for all players in a multiplayer experience

Post Process Volume

handles the post process highlighting.
Blueprint: `BP_PostProcess`
Setup: no additional setup necessary

Unnecessary if no post process highlighting is used in the level.

2.3 Navigation

The navigation elements of the Framework are mainly connected to the teleport system which is very central to the VR environment, since normal movement in VR is prone to cause motion sickness. The teleport navigation elements mainly serve to restrict the areas a player can teleport to.

If you dont want to use teleportation to move the pawn, you can pass including a teleport area or teleport mesh

Teleport Area

Specifies wider areas where the player can safely teleport to. The teleport area is a plane actor with a teleport and a select component as possible interaction methods.
Blueprint: `BP_TeleportArea`
Settings:

- Grid Material - defines what the teleport area looks like
- Visibility - enables the player to perceive the teleport area in the application.
- Adjustment Search Distance - Maximum distance in which the trace of the teleport area searches for a floor actor to teleport the pawn to.

Figure 3. Teleport Area

Teleport Point

Specifies a limited number of locations that the player can teleport to. A teleport point is basically an actor containing the teleport component (see section 3.1.6) and a number of other interaction components.
Blueprint: `BP_TeleportMesh`
Settings:

- Use Rotation - forces the pawn to adopt the rotation of the teleport mesh actor if enabled

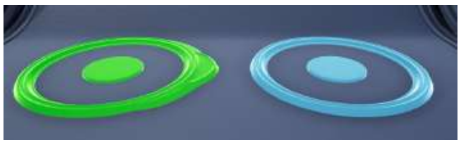


Figure 4. Teleport Point

Teleport Select vs. Normal Teleport

Both teleportation elements can be either toggled via the teleport component or via the select component.

- Teleport Select - the teleport is implemented as a custom function using the select component as interaction component, avoiding the teleport motion component and additional input
- Normal Teleport - the teleport is implemented using the teleport component and the teleport motion component, providing more elaborate controls and settings

2.4 Changing Levels

The Framework provides a number of mechanics to smoothly load or change levels. This includes the transition and intro screens but also managers orchestrating streaming levels within one map.

2.4.1 Intro Level

The first level loaded in an application is the intro level. The Framework provides a completely set up intro level which can be used as a template. It contains:

- Intro Screen - to display the intro widget and handle the transition to the next level
- Player Position - to specify where the player pawn is spawned
- Map Info Actor - containing the level data asset with the info about the intro level
- Skysphere - to provide the background for the intro level

Except the intro screen all other actors in the intro level belong to the actors necessary for a basic level set up.

Intro Screen

The intro screen is the of the intro level which handles most of the logic. It displays the intro widget and also toggles the transition to the next level
Blueprint: `BP_IntroScreen`
Settings:

- Level to Load - contains the level data asset of the next level that should be loaded

2.4.2 Transition

The transition object is automatically created and filled with content as specified in the level data asset (see section 7.1.2) whenever a new level is loaded.
Blueprint: `BP_Transition`

For customized transitions create a child of the transition class, customize it and enter it as transition class into the level DA of the corresponding levels.

2.4.3 Streaming Level Manager

This manager contains an array of all streaming levels that should be accessible in the map and the functions to exchange one streaming level for another.
Blueprint: `BP_Manager_StreamingLevel`
Settings:

- Available Stream Levels - array containing a struct for each streaming level that should be accessible
  - Name - contains the key and I18n data table for the name shown as level name
  - Image - contains the texture shown for the level
  - Level Name - full name of the map of the streaming level
  - AR - under construction
- Currently Loaded Level - is automatically set to the selected level, no setup necessary
- Currently Loaded Index - index of the current streaming level in the array of streaming levels. Also determines which streaming level is loaded upon starting the level

Overview On Setup

- In the Worldmap - add the persistent level and all streaming levels to the same worldmap
- In the Persistent Level - add an instance of the streaming level manager
- On the Streaming Level Manager Instance - enter the data for all accesible streaming levels and the index of the first streaming level to be loaded
- Access Method - prepare a functionality that calls the load streaming level function of the manager

3 Components

Components can be attached to actor classes as well as instances of actor classes providing the desired functionalities. Components are freely exchangeable and enable the customization of actors without building a complicated class hierarchy. All components we created for the AF Core fall into one of the following categories depending on their main functionality:

- Interaction Components - handle the interaction between the player and the actors in an application including selecting, grabbing and others.
- State Components - apply, initiate and replicate changes of state for the owning actor.
- Snapping Components - cover all necessary requirements to allow actors to snap into place after being released by a player.
- Multiplayer Components - consolidate a set of useful components handling the multiplayer.
- UI Components - encompass the design, display and content of UI elements.
- Pawn Components - provide the pawn with functions like controls.
- Info Components - save and provide dynamic information that can be changed during runtime.
- Miscellaneous Components - comprise a set of useful components that work in the background to enable advanced functionalities.

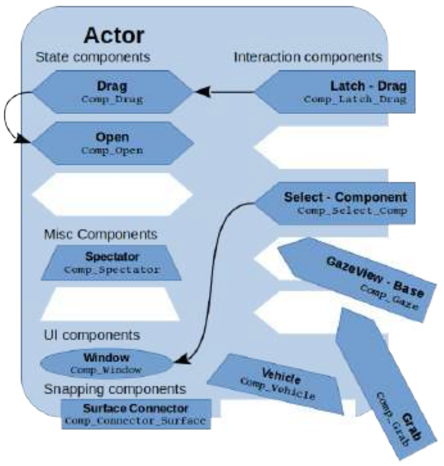


Figure 5. Component System

Info Box - Component Tags

Every component can be equipped with an array of tags that can be used to refer to this component by other components. This includes also Unreal inherent components like the static mesh component. Component tags play a vital role in coordinating component interactions. They can be used to identify a specific component on an actor (see section 3.1) or to select a suitable component/actor from a set (see section 3.3).

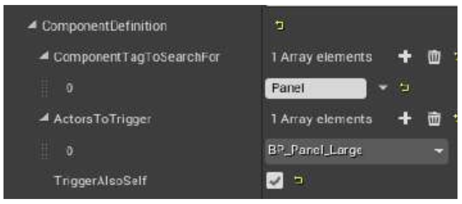
3.1 Interaction Components

All pawns and motion controllers of the Framework are designed to use interaction components to implement interactions of the player with an actor. Each set of components is adapted for a specific interaction type like select or grab. Nevertheless, all interaction components have some common settings despite their different functions. These encompass mainly the identification of the actor and component the interaction is supposed to toggle and the highlighting.

Component Identification

Most interaction components transmit the player input to the other components of the actor that implement the desired functionalities. These can be on the same actor or on another actor or on both. This facilitates the setup of triggers. Consequently, it must be possible to distinguish a single component on a single instance or a group of instances unequivocally. This is done by the Component Definition Struct which contains:

- Component Tag To Search For - array of tags that label each component on the affected actor(s) that is toggled by the interaction
- Actors To Trigger - array on each instance of the actor, that contains an element for each actor instance in the level that is affected by the interaction component
- Trigger Also Self - automatically searches the actor itself for the component tags gathered in the Component Tag To Search For array.



ComponentDefinition	
ComponentTagToSearchFor	1 Array elements
0	Panel
ActorsToTrigger	1 Array elements
0	BP_Panel_Large
TriggerAlsoSelf	✓

Figure 6. Component Definition Struct

Can only be set up on an instance of the actor.

If no other actors are affected by the interaction component no additional setup is necessary.

Highlighting

Most interaction components support a customizable highlighting function. The gazeview component is the only exception. The highlight settings on the interaction component that determine if and how the actor is highlighted consist in:

- Highlight Type - specifies if and how actor is highlighted
  - None - no highlighting
  - Custom - uses a function implemented on the actor
  - Post Process - uses a colored rim around the actor mesh for highlighting, default setting (see Figure **??**)
  - Mesh - duplicates the meshes of the actor and sets a different material to create an outline effect
  - Material Function - activates a material function changing the materials glow parameter for highlighting
- Highlight Color - determines the color for post process and material function highlighting
- Highlight Material - defines the material highlight function
- Component Tag to Highlight - specifies the mesh that should be highlighted by tag.

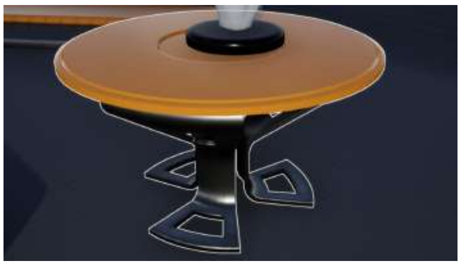


Figure 7. Post Process Highlighting

💡 Not recommended for emissive materials.

💡 Currently only one is available.

3.1.1 Select Components

```
Comp_Select
├─Comp_Select_Comp
└─Comp_Select_SelectionMenu
```

The select components implement the selection of actors in the application. Their main function however, is information transfer. In consequence, select components collaborate constantly with other components depending on what should happen upon selection.

Select Component - Base

Serves as parent class for the more specified select components and as possibility to implement custom functionalities while profiting from the logic of the select component.

Blueprint: `Comp_Select`

Settings:

- SelectEnabled - temporarily disables the select component
- Identifier - allows other components to distinguish the select component
- Component Tag to Select - refers to the mesh corresponding to the select component
- Supported Select Sources - contains an element for each input source, that can interact with the select component
  - None - No input is accepted
  - Laser - accepts the laser motion controller as input
  - Screen - accepts input from all screen-type pawns
  - Touch - accepts input from hand motion controller by touch

💡 To make all meshes selectable enter ”none”.

Sounds:

- Select Sound - non-looping sound that is played when the select is toggled

- Hover Sound - non-looping sound that is played when the select starts to be available for the player
- Unhover Sound - non-looping sound that is played when the select stops to be available for the player

Events:

- Pressed Select - toggles when the key assigned to the select interaction is pressed
- Released Select - toggles when the key assigned to the select interaction is released
- Highlight Select - toggles when the input device hovers on the actor

💡 Almost impossible to implement with a touchscreen input device.

Select Component - Selection Menu

implements the selection menu UI element (see section 8.3.1) which consists in an array of buttons that connect the select component a set of components mostly state components.

Blueprint: `Comp_Select_SelectionMenu`

Settings:

- Selection Buttons - contains an element for each button of the selection menu
  - Button Class - determines which button is spawned
  - Component Definition - specifies the state component that should be toggled by the button
- Sorting Type - determines how the buttons of the selection menu are arranged spatially
  - Circular - arranges the buttons on a circle (see Figure 8)
  - Rows - arranges the buttons in rows (see Figure 76)
  - Rows and Columns - arranges the buttons in rows and columns (see Figure 77)

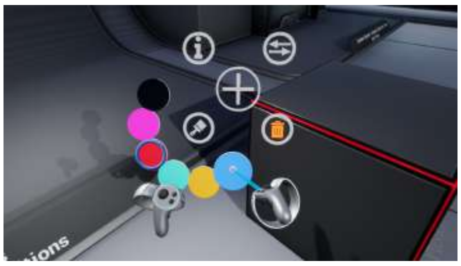


Figure 8. Circular Button Arrangement

💡 A button to close the selection menu is automatically added.

Available Buttons

- `BP_Selection_Delete`
- `BP_Selection_Bool_Info`
- `BP_Selection_Bool_Light`
- `BP_Selection_Bool_Open`
- `BP_Selection_Bool_Play`
- `BP_Selection_Spawn_Sub_Visual`

Select Component - Component Selection

transmits the select directly to **one** component. That implements the intended functionality.

Blueprint: `Comp_Select_Comp`

Settings:

- Component Definition - specifies which component is toggled by the select

3.1.2 GazeView Components

Allows the player to interact with the owning actor via sight without using a motion controller. The pawn constantly checks for actors with the gazeview component in straight line of sight from the pawns head and activates the component.

```
Comp_Gaze
├─Comp_Gaze_Trigger
└─Comp_Gaze_SelectionButton
```

Gazeview is a great tool for players that are inexperienced with VR (or handling controllers), since it enables the player to interact with actors simply by looking. It is an especially good match for the VR hand motion controller because gazeview is not dependent on a laser to interact with actors out of reach of the player.

GazeView Component - Base

serves as parent class of the more specialized gazeview components and as a basis to activate customized functions using the logic of the gazeview component.
Blueprint: `Comp_Gaze`
Settings:

- Identifier - allows other components to distinguish the gazeview component
- Component Tag to Gaze At - refers to the mesh corresponding to the gazeview component
- Component Enabled - temporarily disables the gazeview component

Sounds:

- Hover Sound - non-looping sound that is played when the gaze starts
- Unhover Sound - non-looping sound that is played when the gaze ends

Events:

- Gaze View - is toggled when the player gazes at the mesh specified by the gazeview component

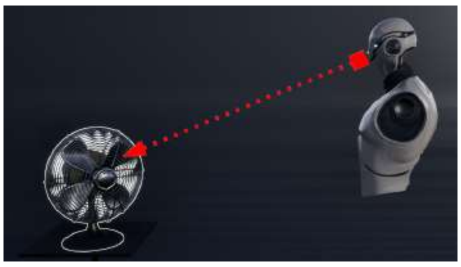


Figure 9. GazeView

To make all meshes gaze-able enter "none".

Gazeview Component - Component

transmits the gaze signal to another component (usually a state component). The Gazeview Component Component also implements the trigger with progress function, that displays a progress bar while the player continues looking on the actor and toggles a state component when the progress bar is full.
Blueprint: `Comp_Gaze_Trigger`
Settings:

- Component Definition - specifies which component the gaze toggles
- Toggle - instructs the gazeview component to act like an alteration switch

Progress Bar:

- Use Progress Bar - enables the progress bar functionality for the component
- Timeout Gaze - defines how long after the player stopped looking at the actor the gaze signal is terminated
- Progress Duration - determines the time in seconds the progress bar takes to fill completely
- Progress Bar Widget - determines what the progress bar looks like

Additional Sounds:

- Sound Progressbar - looping sound that is played during the filling of the progress bar

Events:

- Gaze Triggered - toggles when the progress bar is filled up

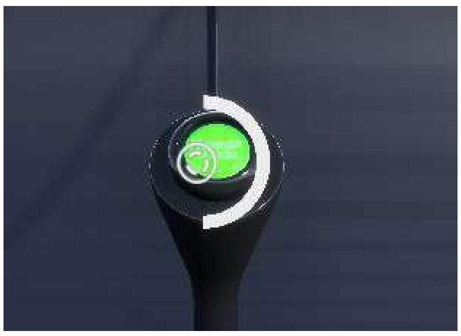


Figure 10. GazeView Trigger With Progress

Figure 11. Gazeview with selection button.

Gazeview Component - Selection Button

spawns a button that toggles **one** component or other component.
Blueprint: `Comp_Gaze_SelectionButton`
Settings:

- Timeout Gaze - defines how long after the player stopped looking at the actor the gaze signal is terminated
- Tag of Location Component - refers to the scene component that determines where the selection button spawns
- Take Root As Location - instructs the gazeview component to spawn the button at the root location instead of the impact location
- Button - states the button class of the selection button spawned upon gaze and component definition of the button to spawn

If no Location Component is present the selection button is spawned at the impact location or root location according to further settings

3.1.3 Grab Components

Grab components enable an actor to be grabbed and moved by the pawn or the motion controller.

Grab Component - Base

Right now the basic grab component is the only fully designed grab component and implements all necessary features. However, we anticipate the incorporation of other specialized grab components into the Framework as is the case for most other interaction components.
Blueprint: `Comp_Grab`
Behaviour:

- Grab Type - determines how the actor behaves during grab
  - Normal - actor is attached to the motion controller or impact location
  - Physics Handle - actor is connected to the pawn or motion controller via a physics handle
- Release Type - defines how an actor behaves upon release
  - Free Placement - after release the actor stays where ever it is currently
  - Physics - activates physics on the actor upon release.
- Should Keep Upright - automatically rotates the grabbed actor until its z-axis is parallel to the world z-axis, not relevant for hand motion controllers

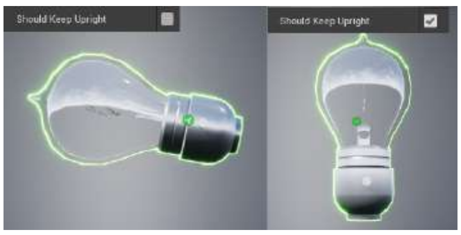


Figure 12. Should Keep Upright Examples.

Snapping components can modify the behaviour of an actor upon release. See section 3.3 for more information.

Sounds

- Hover Sound - non-looping sound played when the grab starts being available
- Unhover Sound - non-looping sound played when the grab stops being available
- Grab Sound - non-looping sound played when the grab interaction is initiated
- Release Sound - non-looping sound played when the grab interaction is ended

Haptics

- Haptics Grab - controls the haptic response of the controller from the start of the grab and during the hold
  - Haptic Curve - describes the intensity of the haptic response starting from the begin of the grab interaction
  - Duration - determines how long the haptic response is active for
  - Strength - serves as a multiplyer to finetune the haptic response

The specified haptic curve is stretched and compressed according to the specified duration.

Snap

- Snap to Controller - attaches the actor to the controller and changes the controls according to the key section below
- Relative Controller Position - defines the relative position of the actor to the controller after snap

Only relevant for laser controllers.

Conditions

- Can be Picked Up - temporarily prevents the actor from being grabbed
- Should Auto Pickup - automatically grabs actors near the palm of the hand motion controller, not relevant for any other controller or non-VR environments
- Max Distance to Socket - determines at which distance of the controller from the socket the grab is automatically released
- Grab Attached Actor Instead - prevents accidentally detaching attached actors when trying to grab the actor
- Grab Tag - specifies the mesh of the actor that should be grabable
- Allow Multigrab - allows multiple actors (pawns or motion controllers) to grab the actor
- Allow Pickup If Already Picked Up - allows another pawn or motion controller to grab the actor while it is held
- Require to Hit Grab Tag - ensures that the actor only can be grabbed at the mesh specified by the grab tag.

Multigrab and pickup if already picked up usually only make sense together

Only relevant for laser controllers.

Keys

- Occupy Keys or Functions - blocks the input of keys or functions so they can be overridden in the event graph by custom controls of the actor
  - AllKeys - automatically clears and blocks the input of all keys
  - Occupied Keys - clears and blocks the input of all entered keys
  - Occupied Functions - clears and blocks all entered controller functions

Only works in conjunction with the snap to controller function

For Clarification

- Occupied Keys serves to replace the controller function of the key with a custom function of the actor, for example the shoot function of a gun.
- Occupied Functions serves to block and clear controller functions that should not be used as long as the actor is held.

Events

- Grab Actor Picked Up - toggles when the actor is grabbed by a player
- Grab Actor Released - toggles when the actor is released by the player

- Grab Thumbstick Axis - allows the player to use the thumbstick for additional input while grabbing
- Grab Key Input - allows the player to use face keys for additional input while grabbing
- Grab Highlight - toggles if the actor is highlighted by the grab component. Use for custom highlighting.
- Grab Trigger Axis - allows the player to use the percentage the trigger button is pressed for additional input while grabbing
- Grab Grip Axis - allows the player to use the percentage the grip button is pressed for additional input while grabbing

3.1.4 Latch Components

```
Comp_Latch
 ├─Comp_Latch_Drag
 ├─Comp_Latch_Physics
 └─Comp_Latch_Simple
```

The difference between a grab and a latch is that for latching the motion controller or pawn is attached to the mesh restricting its movements to the movement possibilities of the mesh.

Latch Component - Base

serves as parent component for all latch components and for implementing custom functions.

Blueprint: `Comp_Latch`

Settings:

- Can Be Latched Onto - temporarily disables the latch component
- Max Attach Distance - defines a distance between the controller an the mesh the controller within which a socket can be found
- Auto Attach - automatically latches onto actors near the palm of the hand motion controller
- Component Tag To Attach To - specifies which mesh of the actor the controller or pawn should latch onto
- Allow Simultaneous Latching - allows multiple actors to latch onto the mesh
- Minimum Detach Distance - defines the distance between socket and controller at which the latch is automatically discontinued

Not relevant for any other controller or non-VR environments.

Sounds

- Hover Sound - non-looping sound played when the grab starts being available
- Unhover Sound - non-looping sound played when the grab stops being available
- Latch Sound - non-looping sound played when the grab interaction is initiated
- Unlatch Sound - non-looping sound played when the grab interaction is ended

Haptics

- Haptics Grab - controls the haptic response of the controller from the start of the grab and during the hold
  - Haptic Curve - describes the intensity of the haptic response starting from the begin of the grab interaction
  - Duration - determines how long the haptic response is active for

The specified haptic curve is stretched and compressed according to the specified duration.

  - Strength - serves as a multiplyer to finetune the haptic response

Events:

- Actor Got Latched - toggles when the latch is initiated
- Actor Latch Released - toggles when the latch is terminated
- Latch Highlight - toggles when the latchable mesh is hovered
- Latch Thumbstick Axis - allows the player to use the thumbstick for additional input while latching
- Latch Key Input - allows the player to use face keys for additional input while latching
- Latch Trigger Axis - allows the player to use the percentage the trigger button is pressed for additional input while latching
- Latch Grip Axis - allows the player to use the percentage the grip button is pressed for additional input while latching

Latch Component - Drag

connects the latch component to the drag component to set up sliders, buttons, levers and valves.
Blueprint: `Comp_Latch_Drag`
Settings:

- Drag Tag - specifies which drag component the latch component is supposed to interact with

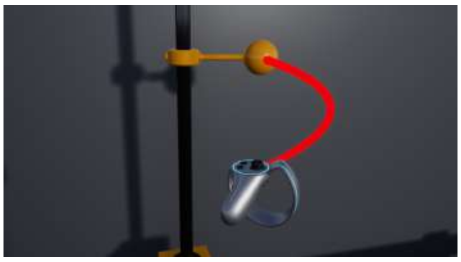


Figure 13. Drag along a spline with rotation.

Latch Component - Physics

spawns a physics handle to implement the latch
Blueprint: `Comp_Latch_Physics`
Settings: no additional settings

3.1.5 Overlap Components

The overlap components enable the implementation of triggers that react to actors that are not the pawn. The most prominent is a pressure plate.

However, the overlap component is able to handle overlap interactions with the pawn, too.

```
Comp_Overlap
 |__Comp_Overlap_Comp
 |__Comp_Overlap_Drag
 |__Comp_Overlap_Select
```

Overlap Component - Base

serves as parent class for all overlap components and as a means for implementing custom functions.
Blueprint: `Comp_Overlap`
Settings:

- Enabled - allows to temporarily disable the component
- Component Tag Of Collision - refers to the collision volume that toggles the component
- Actors To Allow - contains an array of actors that are supposed to trigger the overlap events
- Objects To Allow - contains an array of objects that are supposed to trigger the overlap events
- Component Tags To Allow - contains an array of tags that specifies which tag the actors need to have to trigger the overlap events
- Identifier - identifies the component towards others as source of the trigger

Sounds

- Sound Overlap - a non-looping sound played when an actor, object or component specified in the settings starts overlapping
- Sound Overlap - a non-looping sound played when an actor, object or component specified in the settings stops overlapping

Events

- Actor Begin Overlap - toggles when an actor, object or component specified in the settings starts overlapping
- Actor End Overlap - toggles when an actor, object or component specified in the settings stops overlapping

Overlap Component - Component

transmits the signal of the overlap component to **one** component on an actor (Figure 14).
Blueprint: `Comp_Overlap_Comp`
Settings:

- Component Definition - defines the actor and the component the overlap component should trigger
- Identifier - transmitted by the overlap component as proof of origin
- Trigger Toggle - enables the overlap component to act like an alteration switch

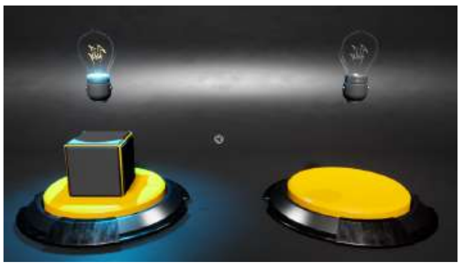


Figure 14. Overlap Component Component.

Overlap Component - Drag

connects the overlap component to a drag component (Figure 15).
Blueprint: `Comp_Overlap_Drag`
Settings:

- Tag of the Drag Component - specifies which drag component the overlap component is supposed to interact with.

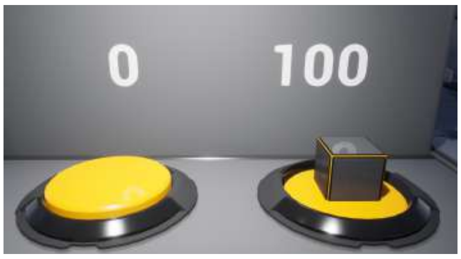


Figure 15. Overlap Drag Component.

3.1.6 *Teleport Component*

enables an actor to be the target of a teleport movement interaction.
Blueprint: `Comp_Teleport`
Settings:

- Enabled - temporarily disables the teleport component
- Teleport Duration - determines the time in seconds the teleport takes
- Teleport Fade - defines how the camera handles the teleport
  - No Fade - disables camera fades for the teleport
  - Fade In and Out - implements camera fades at the beginning and at the end of the teleport
  - Fade In - implements a camera fades only at the beginning of the teleport
  - Fade Out - implements a camera fade only at the end of the teleport
- Teleport Motion - determines how much of the movement of the pawn during teleport is shown to the player
  - None - shows no movement

To switch from normal teleport to teleport select disable the teleport component on the teleport point or area and enable the select component and vice versa.

Camera fades make no sense with the Full Motion teleport motion setting.

  - Indication - shows the first few seconds of the teleport to indicate movement
  - Full Motion - shows the full movement
- Teleport Level - adjusts which part of the pawn is teleported to exactly the chosen location
  - Ground Level - teleports the pawn so that the feet of the character pawn are at the chosen teleport location
  - Eye Level - teleport the pawn so that the camera is at the chose location.

Not suitable for non-Character Pawns.

- Rotation Type - specifies how the pawn rotation is handled during teleport.
  - Force Rotation - prevents any influence from teleport controls and overrides the pawn rotation with the rotation of the actor owning the teleport component
  - Allow Custom Rotation - enables the teleport rotation from the preset data asset (see section 7.1.1) to determine the pawn's rotation after teleport
  - Do Not Rotate - keeps the pawn´s original rotation

Sounds

- Sound Teleport - non-looping sound played when the teleport is executed
- Hover Sound - non-looping sound played when the teleport trace hits an actor with the teleport component
- Unhover Sound - non-looping sound played when the teleport trace leaves an actor with the teleport component

3.2 State Components

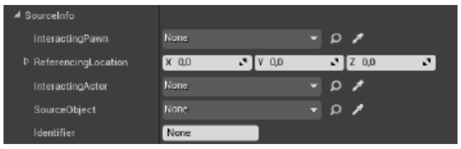
State components implement the main functionalities of an actor like toggling a light or executing a visual change. They usually are activated by interaction components or another state component and implement a plethora of common functionalities.

Interfaces

Each state component implements an interface to standardize information transfer between the state component and the actor or the interaction components (see also section 4).

Source Info

The source info is a struct identifying the interaction component which toggled the state component in question. It is transferred to the state component whenever the interaction component toggles it and can be accessed by the event dispatcher to be used in custom functions.



```
SourceInfo
  InteractingPawn        None
  ReferencingLocation    X 0,0    Y 0,0    Z 0,0
  InteractingActor       None
  SourceObject           None
  Identifier             None
```

Figure 16. Source Info.

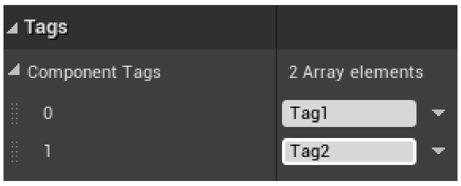
- Interacting Pawn - pawn who initiated the interaction
- Referencing Location - usually the impact location
- Interacting Actor - motion controller or pawn that initiated the interaction
- Source Object - actor owning the interaction component
- Identifier - label identifying the interaction component

The interacting pawn, actor and source object are only accessible in instances.

Component Tag

Component tags serve as references for the state component to be used by the interaction components. State components can have several tags as well as share tags to be toggled in parallel. As long as one tag fits the specification in the interaction component the state component is toggled.

3.2.1 Active Component



Tags	
Component Tags	2 Array elements
0	Tag1
1	Tag2

Figure 17. Component Tag.

Activates or deactivates a functionality of an actor, for example playing a movie on the TV screen

Blueprint: `Comp_Active`

Interface: `Interface_StateComponent_Binary`

Settings:

- Value - determines the starting value of the active-Boolean

Sounds

- Sound Deactivate - non-looping sound played when the active state changes from *true* to *FALSE*
- Sound Activate - non-looping sound played when the active state changes from *FALSE* to *true*

Events:

- Active State Changed - toggles whenever the current state of the component changes
- In Editor Active State Changed - toggles whenever the current state of the component is changed in editor

3.2.2 Color Component

initiates a color change on a given actor.

Blueprint: `Comp_Color`

Interface: `Interface_StateComponent_Color`

Settings:

- Value - determines the starting color
- Available Colors - contains an element for each color the component should support

Sounds

- Color Change Sound - non-looping sound played whenever a color change is initiated

Events:

- Color State Changed - toggles whenever the current color changes
- In Editor Color State Changed - toggles whenever the current color is changed in editor

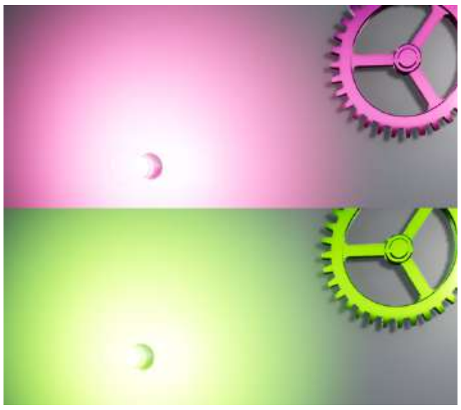


Figure 18. Color Component Example.

How The Delete Component Supports Spawning

The animation provided by the delete component can also be used to support spawning an actor. In this case the delete component only provides the animation which is toggled directly after spawning the actor.

3.2.3 Drag Component

Enables a mesh of the actor to be moved relative to all other meshes to create sliders, buttons or more complex triggers or puzzles.

Blueprint: `Comp_Drag`

Interface: `Interface_StateComponent_Trigger`

Settings:

- Drag Type - defines the spacial curve of the drag
  - Linear - the mesh is dragged along a straight line parallel to the x-, y- or z-axis like a slider

This Slider stays in the Position it is released in.
This Slider snaps to the next segment.

  - Angle - the mesh is dragged along an angle around the x-, y- or z-axis like a lever
  - Spline - the mesh is dragged along a customized spline that can describe curves and bends (see Figure)
  - Spline With Rotation - the mesh is dragged along a customized spline and can rotate around this spline (see Figure)
  - Rotation - the mesh rotates around the x-, y- or z-axis like a valve
- Snap Type - defines how the dragged mesh behaves upon release
  - Free Movement - the mesh remains at the position it is at
  - Snap To Segment - the mesh moves to the nearest section and stays there
  - Reset - the mesh moves to the start position of the drag

Figure 20. Spline Drag.

- Start Position - determines the position of the dragable mesh upon spawn or the start of the level
- Snapping Speed - defines how fast the dragged mesh snaps to its intended position after release.

Not relevant for the FreeMovement snap type.

- Follow Speed - defines how fast the dragged mesh follows the motion controller or pawn
- Select Rule - determines how the select component interacts with the drag component

If a latch component is used for interacting with the drag component this setting does not apply.

  - Toggle Extremes - select moves the dragable mesh between the highest and lowest section
  - Step Up Modulo - select moves the dragable mesh to the next section that is higher than the current

If the highest section is reached a select resets the drag

  - Step Down Modulo - select moves the dragable mesh to the next section that is lower than the current

If the lowest section is reached a select resets the drag

- Toggle - enables the component to work like an alteration switch
- Inverted Percent - commands the component to invert percentage calculation
- Identifier - identifies the component towards others as source of the trigger

Section values indicate relative position. By default the component converts these into percentage entering the lowest section value as 0% and the highest as 100%

- Component Definition - allows the drag component to toggle another state component similar to interaction components. For more information see section 3.1.
- Sections Set - contains an array of float values defining the relative position of sections of the drag.

The sections set needs at least 2 values the start and finish value.

- Looping Sound - played during the whole time the drag is active
- Component Tag to Control - specifies the mesh that should be dragged by stating its tag

Events:

- Drag State Changed Single - toggles once when the section of the drag is changed
- Drag State Changed Constant - toggles every tick as long as the dragable mesh is grabbed and/or moved
- Snapping Completed - toggles after the dragable mesh was released and repositioned according to snap type

3.2.4 *Highlight Component*

serves for highlighting actors independent of interaction components to indicate mission targets for example (see Figure 21).

Blueprint: `Comp_Highlight`

Interface: `Interface_StateComponent_Binary`

Settings:

- Type - defines what the highlight looks like
  - None - no highlighting
  - Custom - uses a function implemented on the actor
  - Post Process - uses a colored rim around the actor mesh for highlighting
  - Mesh - duplicates the meshes of the actor and sets a different material to create an outline effect
  - Material Function - activates a material function changing the materials glow parameter for highlighting

Not recommended for emissive materials.

- Highlight Tag - allows to highlight only meshes with the given tag
- Use Highlight Tag - enables highlighting of selected meshes by tag
- Highlight Material - defines the material put on the duplicate mesh created for mesh highlighting

Of no consequence for highlighting types apart from mesh highlighting.

- Color - defines the color of post process, mesh and material function highlighting

Sounds:

- Sound Start Highlight - non-looping sound played when the highlighting is activated
- Sound Stop Highlight - non-looping sound played, wehn the highlighting is deactivated
- Sound Loop - looping sound played, during highlighting

State

- Value - defines if the highlight is active or inactive at the start of the level and can be used to regulate the highlighting during runtime

Events:

- Highlight - toggles a custom highlight function

3.2.5 *Name Component*

serves to save and replicate a name variable.

Blueprint: `Comp_Name`

Interface: `Interface_StateComponent_Name`

Settings:

- Value - starting value for the name
- Available Names - defines an array of names supported by the component

Events:

- Name State Changed - toggles whenever the name variable of the component is changed
- Name State Changed In Editor - toggles whenever the name variable of the component is changed in editor

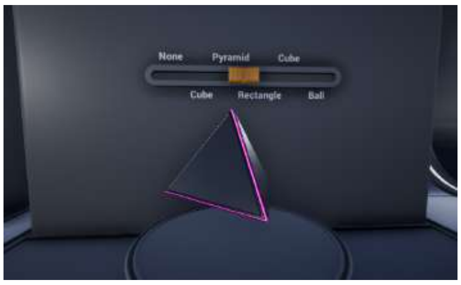


Figure 22. Spawner using the name component

3.2.6 *Open Component*

Enables the actor to open and close by transforming on of its meshes according to the settings of the component.

Blueprint: `Comp_Open`

Interface: `Interface_StateComponent_Binary`

Settings:

- Open Map - describes the open action in terms of movements or scaling
  - Text Field - contains the component tag of the mesh affected by the open action described below
  - Transform Map - describes one part of the open transformation
    - *Location* - moves the mesh linearly parallel to the corresponding axis
    - *Rotation* - rotates the mesh around the corresponding axis
    - *Scale* - scales the mesh along the corresponding axis
    - *Custom* - custom function implemented on the actor
  - Open Value - transformation value, when the actor is open
  - Close Value - transformation value, when the actor is closed
  - Curve - describes the opening or closing speed as percent (of the movement) vs time
  - Duration - defines how long the transformation takes in seconds
- Open - determines the starting value of the open component.
  - Value - set *true* to use the open values defined in the transformation maps and to *FALSE* to use the closed values

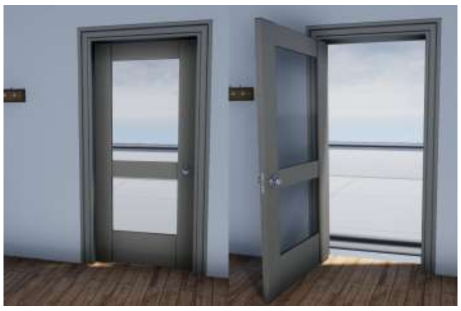


Figure 23. Angular opening.

If more than one mesh is affected by the open action add one entry in the open map for each mesh and refer to it with an individual tag

For more complex processes like the movement along multiple axis include multiple transform maps each describing a part of the transformation

The curve defined here is compressed or stretched according to the duration defined below.

Sounds

- Sound Open - non-looping sound played when the value changes from *FALSE* to *true*
- Sound Close - non-looping sound played when the value changes from *true* to *FALSE*

Events:

- Open Percent Changed - transmits the percentage of the movement every tick during the open/close process
- Open State Changed - toggles when the movement arrives at the open/closed relative transform

3.2.7 *Percent Component*

serves to receive and save a float number.

Blueprint: `Comp_Percent`

Interface: `Interface_StateComponent_Percent`

Settings:

- Value - determines the starting value of the float

Events:

- Float Changed - toggles when a new float value is transmitted to the component
- Float Changed In Editor - toggles when a new float value is transmitted to the component in editor

3.2.8 *Trigger Component*

serves to receive a signal from another component (i.e. the drag component) and process it in a custom function.

Blueprint: `Comp_Trigger`
Interface: `Comp_StateComponent_Trigger`
Settings: none
Sounds

- Sound Trigger - non-looping sound played, when the trigger signal is received

Events:

- Component Triggered - toggles when the trigger signal is received

3.2.9 *Values Component*

serves to receive and save an array of integers. The values component is specifically designed to interact with the selection menu component (see section 3.1.1) to create menus with a depth of one.

Contrary to previous versions, the values component is now equipped with a single level integer interface (see section 4) and can only receive **one** integer value

Blueprint: `Comp_Values`
Interface: `Interface_Integer`
Settings:

- Values Map - matches each supported integer to a name and button to be used in the selection menu or a similar custom made menu
  - Value - integer of the entry
  - Name - I18n struct corresponding to the name used for the current entry
  - Button - button struct allowing to define a texture or color for the selection button matching the entry
- Current Index - determines the starting from the integers defined in the values map

Events:

- Value Changed - toggles whenever a new value is selected from the array
- Value Changed In Editor - toggles whenever a new value is selected from the array in editor

3.2.10 *Velocity Component*

receives the stats of a movement

Blueprint: `Comp_Velocity`
Interface: `Interface_StateComponent_Velocity`
Settings:

- Velocity - determines the starting values of the velocity struct
  - Distance - states the current distance of the moved object
  - Percentage - states the current percentage of a predetermined movement along an angle, axis or spline.
  - Velocity - states the current velocity of the movement

Events:

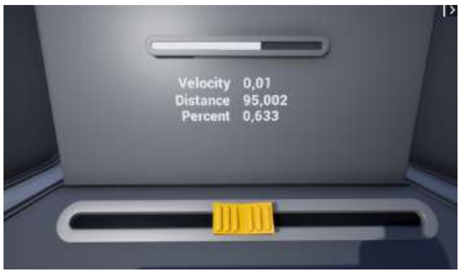


Figure 24. Using Velocity Component.

- Velocity Changed - toggles when a value of the velocity struct is changed
- Velocity Changed In Editor - toggles when a value of the velocity struct is changed in editor

3.2.11 *Visual Component*

manages the appearance of an actor including meshes, materials and the information given about the actor by UI elements. Each entry in the visual component creates a unique look that consists of one mesh and a number of material sets for that mesh. Thus, the visual component provides now an unmatched flexibility.

For more information on the structs governing the looks and materials sets of the visuals component have a look at section 7.3.

Blueprint: `Comp_Visual`
Interface: `Interface_Integer_MultiLevel`
Settings:

- Name - contains the key and I18n data table of the title of the HUD button that toggles the visuals component
- Meshes - contains an element for each look of the actor
  - Static Mesh - states the static mesh of the actor in this look.
  - Skeletal Mesh - states the skeletal mesh of the actor in this look.
  - Materials - contains an element for each set of materials this mesh can appear in.
    - *Name* - I18n struct defining the name of the material set
    - *Materials* - array contaning an entry for each material used on the mesh governing the look.
    - *Sound* - non-looping sound assigned to this particular material set
  - Button 2D - states the name, I18n data table and color or texture of the button to access this look.
  - Data Asset - specifies the data asset describing this look.
  - Sound - non-looping sound assigned to this particular look
- Component Tag to Search - refers to the tag of the mesh that is affected by the visuals component.

If no datatable is entered or the key is missing the key itself is displayed in the application.

You can insert either a static mesh or a skeletal mesh. Never both or a static mesh in one element and a skeletal mesh in another.

All materials of the mesh specified above have to be entered to create a set of materials even if they do not change.

If no datatable is entered or the key is missing the key itself is displayed in the application.

Sounds

- Sound Material Change - general non-looping sound played whenever a material set is exchanged
- Sound Mesh Change - general non-looping sound played whenever the look (including the mesh) is changed

Reminder
The visuals component can only affect one mesh at a time. In case multiple meshes should be changed or shown with different materials, one visual component must be added per affected mesh.

Events

- Visual State Changed - toggles whenever the look or material set is changed

3.3 Snapping Components

The snapping components group includes all components necessary for the snapping feature that allows a grabbed object to automatically place itself on a planar surface or attach to an anchor upon release. Both ”snap to surface” (Figure 25) and ”snap to anchor” (Figure 26) require a careful

Its not recommended to use the general sound settings in combination with the individual assigned sounds in the maps

setup to function properly.
The Framework provides two types of connector components that enable a grabbed actor to search for suitable locations to snap to.
Anchors components collaborate with the anchor connector component by providing a location and rotation for attaching as well as selecting suitable anchor connector components using tags.

3.3.1 Surface Connector Component

Determines if and how an actor snaps to planar surfaces. The forward vector of the surface connector component indicates which part of the actor is supposed to attach to the surface.
Blueprint: `Comp_Connector_Surface`
Settings:

- Rotate On Surface - enables the actor to automatically align to the surface.
- Should Be Active Even With Physics - allows the actor to snap despite physics
- Allow Physics After Snap - reactivates physics after snapping
- Requires Tag - enables the component to recognize surfaces by tag
- Wall Rotation Adjustment - compensates the deviation of the pivot of the mesh from what the surface connector expects
- Search Distance - defines the range in which you want the surface connector to search for surfaces.

Sounds:

- Sound Location Found - non-looping sound that is played whenever the connector finds a suitable surface for snapping

Events

- Connector Attached - toggles when the connector is attached to a suitable surface
- Connector Detached - toggles when the connector is detached to from a suitable surface

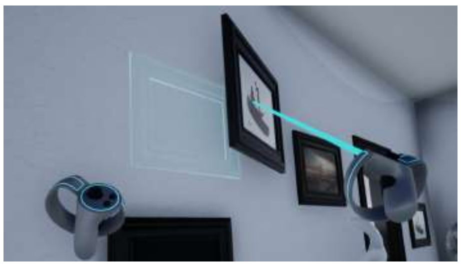


Figure 25. Snap to surface

Without a tag specified, anything with an impact angle between 120 and 60° to the search trace constitutes as planar surface.

Setup: Snap To Surface

All logic for the snap to surface functionality is encapsulated in the surface connector component. However, snapping is a complex process which needs careful setup to work properly. Here are the most important concepts to consider when setting up the snap to surface functionality:

- Positioning - the surface connector component needs to be positioned so its forward vector is orthogonal to the part of the actor that is supposed to touch the surface after the snap (a vector component can easily be used for visualization) and enter settings.
- Settings Combinations - snapping is initiated when a grab is released and therefore heavily influenced by the release type of the grab component (see section 3.1.3) The following settings combinations are recommended:
  - *Release Type: FreePlacement* - set "Should Be Active Even With Physics" and "Allow Physics After Snap" to *FALSE*

- *Release Type: Physics* - set "Should Be Active Even With Physics" to *true* and "Allow Physics After Snap" to *FALSE* to place actors on the wall or ceiling
- *Release Type: Physics* - set "Should Be Active Even With Physics" and "Allow Physics After Snap" to *true* to place actors on the floor

- Surface Tags - a consistent tag system on the surfaces in the map enables the surface connector to select suitable surfaces, but is not necessary for the surface connector to function.
- Already Attached Instances - this is only necessary if an actor should be attached to a surface upon loading the level. In this case:
  - *On the Instance* - enter the surface actor this instance of the actor is attached to
  - *In Case the Surface Actor Has Various Meshes* - enter the tag of the mesh the actor should be attached to (none means the root component)
  - *When Using a Socket* - specify the socket, too

3.3.2 Anchor Connector Component

Determines if and how an actor snaps to an anchor component on another actor. The position and rotation of the connecting actor is fully defined by the alignment of the forward vectors of the anchor connector component and the anchor component as shown in Figure 27.

Note
In case of anchor vs surface the anchor is always preferred.

Blueprint: `Comp_Connector_Anchor`
Settings:

- Search Distance - defines in which range the anchor connector searches for suitable anchor components.
- Search Radius - adjusts the width of the search trace.
- Connector ID - matches the connector to the anchor it is intended for

Don't make the search radius too small else it will be very tedious for players to attach actors to each other, since the connector will not find an anchor even in close proximity.

Sounds:

- Sound Location Found - non-looping sound that is played whenever the connector finds a suitable anchor for snapping

Events

- Connector Attached - toggles when the connector is attached to a suitable anchor
- Connector Detached - toggles when the connector is detached to from a suitable anchor

3.3.3 Anchor Component

Specifies where another actor can attach to the actor and which actors are suitable. The anchor component also indicates the position and rotation of the attaching actor as shown in Figure 27. Anchor components can also be added to character pawns creating body slots, that enable the player

to attach actors to the pawn.
Blueprint: `Comp_Anchor`
Settings:

- Sphere Radius - determines the size of the collision sphere, that allows the trace of the anchor connector component to find the anchor
- Connector IDs To Allow - contains an element for each connector ID (see section 3.3.2) that should be attachable to the anchor.
- Connector Should Attach - ensures the actors stay attached to each other after release
- Allow Physics - reactivates physics after the snapping process
- Anchor Deactivated - temporary prevents the use of this anchor

Sounds:

- Sound Attach - non-looping sound that is played whenever a connector is attached to the anchor
- Sound Detach - non-looping sound that is played whenever a connector is detached from the anchor

Events

- Connector Attached - toggles when a connector is attached to the anchor
- Connector Detached - toggles when a connector is detached from the actor

Anchors can have multiple entries for connector IDs to allow, so a variety of actors can be attached to the same anchor.

If Connector Should Attach is not enabled the connected actor may fall to the ground after snapping, when physics is activated.

If the actor is attached physics is activated for the whole construct, meaning it could tilt when the mass center was changed by attaching an actor.

3.3.4 Attach Component

Keeps record of which actor is attached to the actor
Blueprint: `Comp_Attach`
Settings: none
Events:

- Actor Attached - toggles whenever an actor is attached transmitting a reference to the attached actor
- Actor Detached - toggles whenever an actor is detached

Setup: Snap To Anchor

The snap to anchor functionality implements the careful positioning of two actors (one grabbed, one stationary) to each other, so the grabbed actor can snap (and attach) to a specific position on the stationary actor. This requires the setup of at least two elements: The *Anchor Connector Component*, which is owned by the grabbed actor that should snap upon release, and the *Anchor Component*, that is owned by the stationary actor that other actors should be able to attach to.
As with the surface connector component the anchor connector component and the anchor component implement several functions including:

- Positioning - both the anchor and the anchor connector need to be placed with their forward vector orthogonal to the surfaces where the actors should attach, since the snap position is calculated by aligning both vectors (use a vector component for visualization).
- Connector IDs - to limit which actor should attach to which each connector is assigned an ID. Only anchors with this ID in their Connector

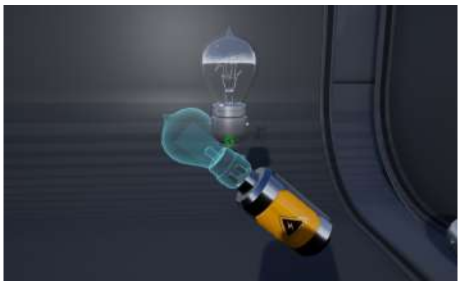


Figure 26. Snap to Anchor/Attach

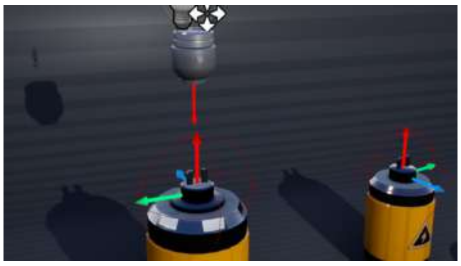


Figure 27. Anchor and connector.

IDs To Allow array will attach the actor.

- Attach - the logic for attaching an actor after snap is encapsulated in the anchor. To keep track of attached actors however, the attach component is necessary.
- Additional Setup - is only necessary if actors should be attached to a surface upon loading the level. In this case:
  - *On the Instance* - enter the actor this instance of the actor is attached to
  - *In Case the Actor Has Various Meshes* - enter the tag of the mesh the actor should be attached to (none means the root component)
  - *When Using a Socket* - specify the socket, too

3.4 Multiplayer Components

```
Replication Component
├─Actor Replication
│ Component
└─Component Replication
  Component
```

Multiplayer applications are a special challenge and the AF Core cannot provide a solution for every issue, that is connected with multiplayer. For now we concentrated our efforts to provide a consistent and performant replication system by equipping all components to replicate as well as provide a few extra components to handle movement replication.

3.4.1 Base Replication Component

serves as parent component for the more specialized replication components. The base replication component is not meant to be used in an application.

Blueprint: `Comp_Replication`

Settings:

- Enabled - can be used to temporarily disable the component
- Interp Speed - adjusts the interpolation to show a clean movement of the replicating actors to the new position forwarded by the actor replication component
- Tolerance - determines the minimum distance an actor must be moved to activate the replication process
- Tick Interval During Movement - interval in which a currently moving actor transmits its location for replication
- Tick Interval Without Movement - interval in which a currently stationary actor checks for movement

This cannot be checked every tick since that would exceed the capabilities of most hardware.

Events:

- New Transform Set - toggles whenever the replication component needs to change the transform

3.4.2 Actor Replication Component

replicates the actor's position in a multiplayer application. The actor replication component detects, if an actor is moved, and sends the new position constantly to the server and other clients.

For additional information see section 6

Blueprint: `Comp_Actor_Replication`

Settings: no additional settings compared to parent

3.4.3 *Component Replication Component*

Similar to the actor replication component the component replication component tracks the relative position of a moveable component of an actor and sends it to the server for replication.

For additional information see section 6

Blueprint: `Comp_Comp_Replication`
Settings:

- Component Tag - serves to identify the component that is subject to the replication process

Please note, that both replication components are only indispensable to replicate non-predetermined motion and physics. Consequently, it is *not* necessary to add a replication component to every moving actor or component. For more details see section 6.2.

3.5 UI Components

The AF Core already provides the basis for a great variety of UI elements. Most of them relying on components for their functions and content. As a matter of course, UI elements differ greatly for each application or environment. Consequently, each UI component has a limited set of actors they are supposed to be added to as stated in the individual description of each component.

Note
Many UI elements of the AF Core are very plain. They mainly serve as basis for more sophisticated UI elements, that can be build by the developer or purchased with the AF Extensions.

3.5.1 *Menu Pallet Component*

specifies the menu level DA (see section 7.1.2) of the menu level that can be loaded using the menu and settings pallet.

For more information see section 8.2.3.

Blueprints: `Comp_Pallet_Menu`
Suitable Actors: VR-Pawns, Board Actors
Settings:

- Menu Level - specifies the level data asset of the menu level that should be loaded with the pallet

3.5.2 *Panel-based Menu Components*

```
Panel Menu Component
 ├─Large Panel Menu
 │ Component
 └─Small Panel Menu
   Component
```

provide the content and basic settings for the panels of a panel based menu (see section 8.2.2) The `Comp_PanelMenu` serves as a parent class only. The two child classes implement the two major designs that can be chosen for the panel based menu, large, vertical panels or small, tile-like panels.
Blueprint: `Comp_PanelMenu` **Settings**:

- Currently Centered Index - allows to specify a panel that is already expanded upon creation
- Spawn Panel at Start - spawns the panel based menu, when the level is loaded

This does not mean that the panel is necessarily at the center of the menu right now.
If you dont want any panel to be expanded, set the value to -1.

Events:

- Panel Functionality - can be used to implement widget functionalities on the panel for improved replication

Large Panel Menu Component

determines the content and appearance of the panels of a large panel based menu (see Figure 28)

Blueprint: `Comp_PanelMenu_Large`

Suitable Actors: `BP_Demo_LevelMenu_Large`

Content:

- Panels - array of large panel menu data assets with an entry for each panel the menu is supposed to have

Additional Settings:

- Centered Index - defines which pallet is at the center of the menu upon creation
- Distance Between Tiles - defines the distance between the panels

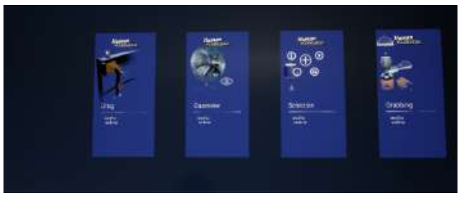


Figure 28. Large Panel Menu, center: 1.

Small Panel Menu Component

determines the content and appearance of the panels of a small panel based menu (see Figure 29)

Blueprint: `Comp_PanelMenu_Small`

Suitable Actors: `BP_Demo_LevelMenu_Small`

Content:

- Panels - array of small panel menu data assets with an entry for each panel the menu is supposed to have

Settings

- Items Per Row - defines how many small panels are shown per row
- Tag Of Custom Forward Location - determines the location and rotation of the expanded panels.
- Distance Between Rows - defines the distance between the rows of panels
- Distance Between Columns - defines the distance between the columns of panels

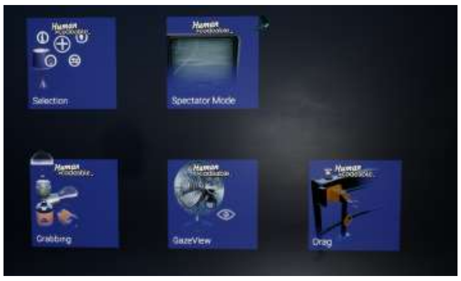


Figure 29. Small Panel Menu, 3 per row.

The small panel based menu does not support scrolling. Thus, all panels have to fit the given space.

3.5.3 Tiny Display Component

enables the actor to spawn a tiny display when grabbed by the laser motion controller describing the actor with a name or short text and an image (see Figure 30).

Blueprint: `Comp_TinyDisplay`

Suitable Actors: all actors

Settings:

- Name - contains the key and I18n data table referring to the name or a description of the actor
- Image - contains the texture corresponding to the actor
- Should Spawn - temporarily disables spawning the tiny display

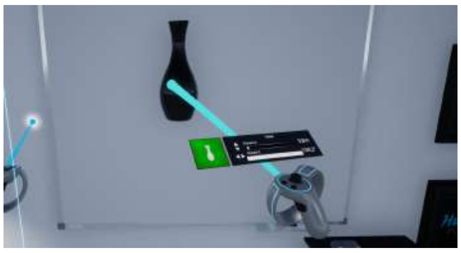


Figure 30. Tiny Display UI.

If no datatable is entered or the key is missing the key itself is displayed in the application.

3.5.4 Widget Component

allows actors to display a widget. The widget component expands the Unreal Engine inherent widget with additional logic to handle player interaction especially with the VR pawns.

Don't confuse the widget component with the window component. The widget component only enables the display of a widget. The window component handles the spawning of an actor owning the widget component (among others).

Blueprint: `Comp_Widget`

Settings:

- Recursive Widget Selection - widget input implementation designed for the Advanced Framework
- Interaction Component Based Widget Selection - UE in-built implementation for widget input
- Initial Tab - defines which tab is open when a multi-tapped widget is spawned

All widgets used in the AF Core examples use the recursive widget selection to implement player-widget-interaction.

User Interface

- Widget Class - defines the widget displayed

Events:

- Widget Functionality - can be used to implement widget functionalities on the actor for improved replication

To use the recursive widget selection implementation all widgets need to be of the type `Widget_Base`. For additional information see section 8.1.

3.5.5 *Window Component*

spawns an info window actor that displays a widget the player can interact with.

Blueprint: `Comp_Window`

Settings:

- Window - matching the window actors to be spawned with locations referred by tags for manual placing
- Use Root Location For Positioning - uses the root location of the actor instead of the impact location for the auto-positoning feature of the window component.

For automatic placing enter *None*

Sounds:

- Sound Spawn - non-looping sound played when the info window actor is spawned
- Sound Despawn - non-looping sound played when the info window actor is deleted

Events:

- Window Butten Pressed - serves to implement widget functionalities on the actor itself for improved replication

3.5.6 *Mini Tag Component*

spawns a mini tag actor that displays a short description and is connected to an actor with a straight line.

Blueprint: `Comp_MiniTag`

Settings:

- Line - defines the design of the line connecting the mini tag to its corresponding actor
  - Has Line - determines if a line is shown

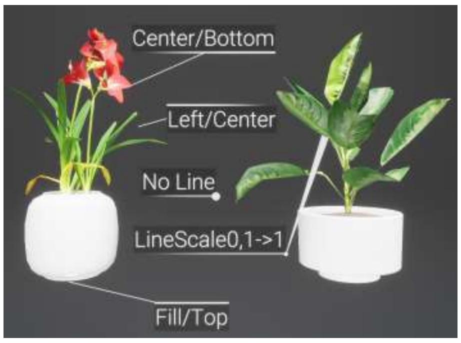


Figure 31. Mini tags with different settings.

- Tag of Component With Socket - specifies the mesh component with the line attach socket
- Socket to Attach Line - determines the socket the line is attached to
- Line Location - specifies the relative location to the actor where the line ends if no socket is specified
- Stay Attached - enables line to stay attached to the socket when the actor is moved (not relevant for line location)
- Line Start Scale - defines the width of the line at the actor
- Line End Scale - defines the width of the line at the mini tag

- Alignment - determines where the line connects to the mini tag
  - Horizontal alignment - specifies where the line connects to the tag horizontally
  - Vertical alignment - specifies where the line connects to the tag vertically

Both alignment options together describe an unequivocal connect position of the line.

- Text - specifies the content of the mini tag
  - Text Size - determines the font size of the text
  - Text - contains the key and I18n data table of the description displayed in the mini tag.

If no datatable is entered or the key is missing the key itself is displayed in the application.

- Border - designs the border of the mini tag
  - Border Height - width of the line at the edge of the mini tag
  - Draw Size - size of the widget that displays the text

3.6 Pawn Components

The AF Core contains a number of components that can only be used by pawns. They define the pawn controls or UI elements. However, since the Advanced Framework supports various environments like VR, desktop or mobile, not all pawn components can be added to every pawn. Pawn restrictions are stated in the description of every component.

3.6.1 Controls Component

The controls component contains all motion controller settings including presets, movement and teleport. In consequence, the controls component can only function on a VR pawn.

Blueprint: `Comp_Controls`
Pawn Type: VR
Controller Settings:

General Reminder

With the introduction of the controls component the controls are no longer *level* specific but *pawn* specific. Consequently you are going to need a set of different pawns if you want to change controls from level to level. For more information have a look at section 5.3.

- Controller Set - defines which motion controllers are spawned by the pawn for the left and right hand
- Hand Tracking Controller Set - defines which controllers are used when hand tracking is active
- Default Preset - specifies which preset data asset is used as fallback if the HMD could not be identified
- Controls Presets - maps a controls preset to each supported HMD (see section 7.1.1)

Upon starting handtracking the controls presets are completely overridden by the hand tracking function maps below.

Some gestures have to be defined for the left an right hand separately, since they are

- Hand Tracking Function Map - matches gesture presets (see section ?? with a controller function similar to key mapping for normal controls
  - Left Hand Gesture Function Map - assigns gesture presets for the left hand to controller functions
  - Right Hand Gesture Function Map - assigns gesture presets for the right hand to controller functions
- Handtracking Preset - preset data asset that defines the teleport mode during handtracking

Haptics Settings:

- Haptics - determines if the controller provides haptic response

Teleport Settings:

- Teleport Mode - determines which locations are available for teleport
  - TeleportFree - the teleport location can be selected freely on flat surfaces, no teleport areas necessary
  - TeleportComponent - searches for the teleport component on the teleport area or teleport point actor (see section 2.3)
  - NavMesh - uses the navigation mesh of the Unreal Engine to standard for suitable teleport locations
- Teleport Search Type - specifies how the trace searching for suitable teleport locations works
  - Straight - trace behaves like a straight line
  - Projectile - trace follows a trajectory toward the floor
- Teleport Time - adjusts how long a teleport takes
- Use Teleport Motion - determines how the movement of the pawn during the teleport is shown to the player
  - None - no motion is shown
  - Indication - activates a zoom effect during teleportation
  - FullMotion - displays the full movement of the pawn from the teleport starting location to the teleport target location

Movement Settings:

- Movement Style - specifies the settings for direkt movement of the pawn according to the button mapping in the controls component
  - Fluent - the pawn move continuously, prone to cause motion sickness
  - StopMotion - to avoid motion sickness the players vision fades in and out, while the pawn moves
  - Ghost - the player moves a ghost representation of the pawn to the desired area of the map and upon finishing the movement the pawn is moved directly to the ghost location
- Movement Direction - describes how the pawn is controlled during movement
  - Direction From Camera - the pawn takes its forwards direction from the forwards direction of the camera
  - Direction From Controller - the pawn moves according to the direction in which the controller points

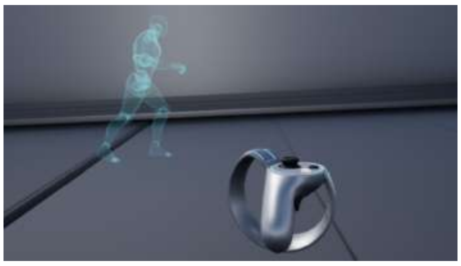
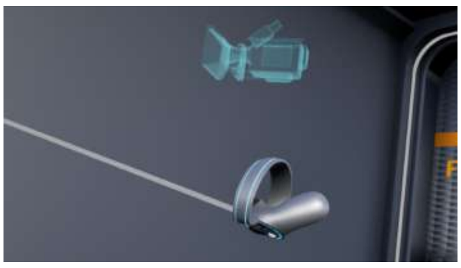


Figure 33. Ghost movement: walk.



    - Direction Relative To Controller Location - moves the pawn according to the current controller position relative to the controller position at the start of the movement
- Stop Motion Interval - specifies the time in seconds between 2 images of a stop motion movement, only relevant for stop motion movement style
- Movement Speed - determines the movement speed of the pawn
- Move With Fade - enables fade to black for movement, not recommended for fluent movement

Rotation Settings:

- Rotation Type - defines how the rotation is implemented
    - Fluent - rotates the pawn fluently as long as the rotation function is active
    - Step - rotates the pawn by a step of a given angle every time the rotation function is initiated
- Rotation Fluent Degrees - defines the velocity the pawn rotates, only relevant for the fluent rotation type
- Rotation Step Degrees - angle in degree that the pawn is rotated for each rotation step, only relevant for the step rotation type.
- Rotate Step Fade - enables camera fade during step-wise rotation
- Rotation Step Duration - determines the time in seconds each rotation step takes

Available Controller Functions

- Select
- Grab
- Ping
- Pause
- RadialMenu1
- RadialMenu2
- ToggleFly
- Teleport
- Move
- MoveForward
- MoveLeft
- MoveRight
- MoveBackward
- TurnLeft
- TurnRight
- ResetOrientation

3.6.2 Radial Menu Component

The main purpose of the radial menu component in the Advanced Framework is controlling the pallet UI element (see section 8.2.3). However, the button set spawned by the radial menu component can also be utilized for other means by designing appropriate buttons.

Blueprint: `Comp_RadialMenu`

Pawn Type: VR

Settings:

- Radial Menu 1 - contains an array of button blueprints that should be spawned upon opening the radial menu.
- Radial Menu 2 - contains an array of button blueprints that should be spawned upon opening the radial menu.

The radial menu 2 slot serves for creating different radial menus for the left hand and right hand controller for example.

Currently Implemented Button Types

- Pallet Button - spawns a pallet UI element that corresponds the button
- Spawn Button - spawns an actor like a weapon or a device.

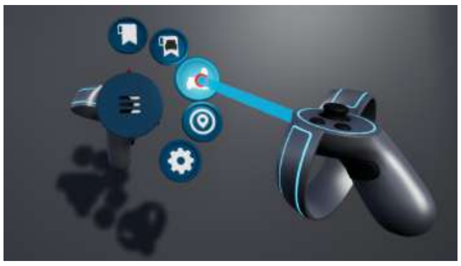


Figure 35. Radial Menu.

3.6.3 Touch Component

registers touch input and transmits it to the mobile pawn for processing.

Blueprint: `Comp_Touch`

Pawn Type: Mobile Pawn

Settings:

- Max Tap Time - maximum time in seconds that allows a touch to be registered as tap
- Max Double Tap Time - maximum time in seconds that can be between to taps if they are to be registered as double tap
- Swipe Sensitivity Threshhold - the minimum amount of pixels the finger has to move for a swipe to be recognized.
- Hold Time - time in seconds until hold is registered

3.6.4 HUD Component

defines the content of the HUD in screen applications. For more information on the HUD management in the Framework see section 8.2.6.
Blueprint: `Comp_UI_HUD`
Pawn Type: Mobile Pawn, Desktop Pawn, Asymmetric Pawn
Settings:

- HUD - contains an entry for each HUD element
  - Name - contain the key and I18n datatable that define the label of the HUD element
  - Widget - determines which widget is used for the HUD element
  - Frame - determines in which customizable slot of the HUD the widget is presented. For a list of frames have a look at the margins and for their position on-screen see Figure 65

3.7 Info Components

Info components serve to save and provide dynamic information that can be changed during runtime in contrast to data assets which are static

3.7.1 Player Info Component

is used by the player state to save individual information about one player except the player index.
Blueprint: `Bp_PlayerInfo_Basic` **Player Stats**:

- Player Color - color assigned to the player
- Player Name - supposed to save the name entered by a player in a multiplayer scenario

3.8 Miscellaneous Components

The miscellaneous components are a collection of component of various functions that cannot be sorted into one of the other groups due to their unique functionalities.

3.8.1 Delete Component

Allows the deletion of the actor.
Blueprint: `Comp_Delete`
Interface: `Interface_Deletable`
Settings:

- Handle By Interface - overrides the component's logic for a customized function on the actor using the deletable interface
- Transition Duration - adjusts how long the deletion process takes

- Curve - describes change of the scale of the actor during the delete process

The curve defined here is stretched and compressed according to the duration defined above.

Events:

- Before Delete - allows preparatory functions to execute before the delete (i.e. animations)
- Appear Finished - toggles when the animation that the delete component contributes to a spawn is finished

3.8.2 *Orbit Component*

The orbit component allows the pawn to move around an actor as if in an orbit. It does not however, move the pawn automatically. It rather changes temporarily pawn movement and navigation.

Blueprint: `Comp_Orbit`

Settings:

- Pitch Range - limits the pitch angle of the pawn position
- Zoom Range - defines the maximum and minimum radius of pawn position around the orbit component

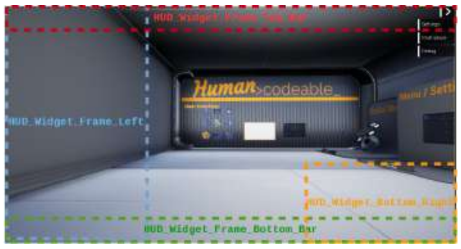


Figure 37. Position and Size of Frames.

The pawn can rotate 360° around the z-axis of the orbit component.

3.8.3 *Spawn Actor Component*

spawns the transmitted actor in a multiplayer compatible fashion.

Blueprint: `Comp_SpawnActor`

Settings: none

Sounds:

- Spawn Sound - non-looping sound that is played whenever an actor is spawned

Events:

- Actor Spawned - toggles when the actor has been spawned

Note

Both spawn components work independently and each is able to implement and replicate the spawn process.

- Spawn Actor Component - straight up spawns an actor
- Spawn Location Component - handles the complete logic of a spawner with additional effects upon spawn and a respawn when the actor has been removed

3.8.4 *Spawn Location Component*

spawns the transmitted actor in a spawner and manages the spawner functionalities.

Blueprint: `Comp_SpawnLocation`

Settings:

- Size - scales the object, so it fits inside the spawner
- Use Spawn Particle - displays a particle effect during spawn
- Use Rotation - rotates the spawned actor after spawn
- Auto Respawn - each detached actor is immediately replaced by another actor of the same type

Sounds:

- Sound Spawn - non-looping sound played whenever an actor is spawned

Events:

- New Actor Spawned - toggles whenever a new actor has been spawned
- Actor Detached - toggles when the spawned actor is taken out of the spawner

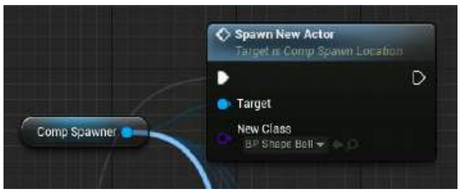


Figure 38. Code Example Spawner.

Setting up a Spawner

To create a spawner add the component to the blueprint and in the event graph call the component´s *Spawn Actor* event. (see Figure 38).

3.8.5 Spectator Component

The spectator component allows an actor to show on the computer screen, what the its camera sees. The owning actor might be a pawn like the asymmetric pawn (see section 5.3.5) for example, but also a security camera or any other actor in the application.

Blueprint: `Comp_Spectator`

Settings:

- Spectator Screen Size - resolution in ptx of the live feed the spectator component provides
- Key - provides an identifier for each spectator component in the level

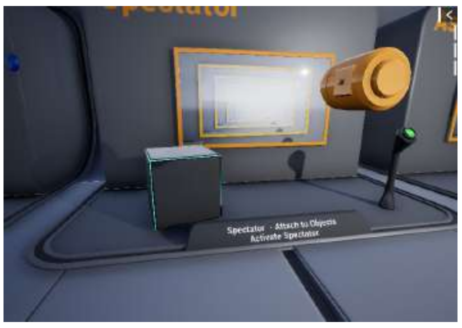


Figure 39. Spectator Component Example.

3.8.6 Vehicle Component

The vehicle component is added to vehicle-type actors to control the entering and exiting of the pawn into the vehicle. Moreover, the vehicle component handles the movement restrictions of the pawn as long as it is placed inside the vehicle.

Blueprint: `Comp_Vehicle`

Transition:

- Transition Duration - determines the time in seconds that the movement of the pawn from its current location to the in vehicle location takes
- Fade - enables the use of a camera fade during the transition
  - No Fade - disables camera fades
  - Fade In - uses camera fades for entering the vehicle only
  - Fade Out - uses camera fades for exiting the vehicle only
  - Fade In And Out -uses camera fades for entering and exiting the vehicle
- Motion - defines how the the movement of the pawn from its current location to the in vehicle location is presented to the player
  - None - shows no movement of the pawn
  - Indication - shows the beginning of the movement before cutting camera
  - Full Motion - shows the full movement of the pawn

Exit:

- Tag Of Exit Component - specifies which scene component of the actor serves as target location for exiting the vehicle (see margin)

Sounds:

- Sound Enter - non-looping sound played when the pawn is transported to the vehicle component positon
- Sound Exit - non-looping sound played when the pawn is transported from the vehicle component location to the exit location

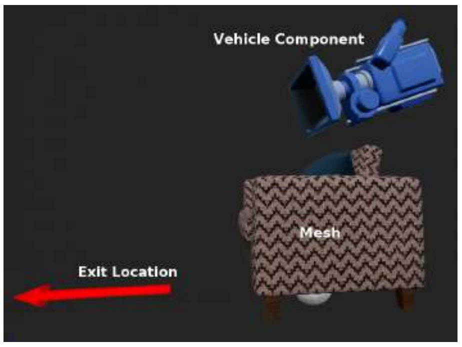


Figure 40. Vehicle Component.

💡 Can not be combined with camera fades.

💡 The vehicle component transports the pawn between the locations. It does not teleport the pawn, meaning that movable actors in the way of the pawn can be knocked over or similar.

Info Box - Vehicle Comp Setup

The vehicle component needs handles the transition of the pawn between two locations:

- In Vehicle Location - is the location of the vehicle component relative to the mesh of its owning actor and constitutes the vantage point of the pawn, when placed in the vehicle
- Exit Location - is the location of an additional scene component relative to the mesh of the actor. The exit location specifies where the pawn is placed upon exiting the actor. The exit location is indicated to the vehicle component by its individual tag.

3.8.7 *Video Component*

The video component greatly facilitates the set up for showing a video file or a streaming a video on a screen or similar in the application. In its default state the video component is set up with a plane as static mesh. However, the static mesh can be changed ad lib.

Blueprint: `Comp_Video`

Source:

- Video Source - defines which type of video source the component should use
  - Stream Media Source - streams the video from the URL registered below
  - File Media Source - uses a video file in the project as source
- Source URL - states the URL from which the video should be streamed

💡 The source URL is not relevant for the "file media source" setting.

Required:

- Media Sound Component - manages playing the sound of the video
  - Component Tag to Search For - Tag of the sound component (Unreal inherent) which should play the sound of the video
  - Trigger Also Self - informs the video component that the sound component is on the same actor
- Media Texture - renders the video file
- Material Face - Index of the material which should display the video in the material set of the static mesh of the video component

💡 Each Video needs its own Media Texture.

Info Box - Video Set Up

The video component manages most of the tedious steps that are necessary to show a video in an Unreal application. After adding it to an actor only 3 additional parts are necessary:

- Mesh and Material - the default mesh of the video component is a 16:9 plane with a dark material that is replaced by the media texture upon activating the component
- Media Texture - create an individual media texture for the actor by duplicating the existing media texture.
- Sound - add the Unreal inherent media sound component to the same actor and provide it with a tag for reference

The video component can be toggled in the same way as the active component or similar state components.

4 Interfaces

Interfaces provide a standardization for the communication between components or other functions of the Framework. Each interface defines an individual data type that can be received and transmitted by this interface.

4.1 State Component Interfaces

All state components are adjusted to work with interfaces and have one assigned interface. All interaction components use the state components' interfaces to interact with state components. State component interfaces come in two categories:

- Unleveled Interfaces - accept only one single values of their corresponding data type
- Multileveled Interfaces - accept arrays of their corresponding data type and support the construction and transfer of tree-like array structures

4.1.1 Components With Binary Interface

transmits a Boolean value.
Category: Unleveled
Blueprint: `Interface_StateComponent_Binary`
Accepted Data: 1 Boolean

Binary Interface Components

- Active Component
- Open Component
- Highlight Component
- Window Component

4.1.2 Integer Interface

transmits an array of integers
Category: Multileveled
Blueprint: `Interface_StateComponent_Integer_MultiLevel`
Accepted Data: 1 array of integers

Components With Integer Interface

- Visual Component

4.1.3 Integer Interface

transmits an integers
Category: Unleveled
Blueprint: `Interface_StateComponent_Integer`
Accepted Data: 1 integer

Components With Integer Interface

- Values Component

4.1.4 Percent Interface

transmits a float value.
Category: Unleveled
Blueprint: `Interface_StateComponent_Percent`
Accepted Data: 1 float

Components With Percent Interface

- Percentage Component

4.1.5 Color Interface

transmits a color struct consisting in 3 float values.
Category: Unleveled
Blueprint: `Interface_StateComponent_Color`
Accepted Data: 1 linear color struct

Components With Color Interface

- Color Component

4.1.6 *Name Interface*

transmits a name value.
Category: Unleveled
Blueprint: `Interface_StateComponent_Name`
Accepted Data: 1 name variable

Components With Name Interface

- Name Component

4.1.7 *Trigger Interface*

transmits a trigger signal
Category: Unleveled
Blueprint: `Interface_StateComponent_Trigger`
Accepted Data: none

Components With Trigger Interface

- Trigger Component
- Drag Component

4.1.8 *Velocity Interface*

transmits a velocity struct consinsting in 3 float values comprising the percentage, velocity and distance.
Category: Unleveled
Blueprint: `Interface_StateComponent_Velocity`
Accepted Data: 1 velocity struct

Components With Velocity Interface

- Velocity Component

4.2 Interaction Interfaces

Interaction interfaces are support the data transfer between interaction components and the actor. Added to the actor they are mainly supposed to transmit settings of the interaction component to the actor.

4.2.1 *GazeView Interface*

Components: GazeView Components
Blueprint: `Interface_InteractComponent_Gazeview`

4.2.2 *Latchable Interface*

Components: Latch Components
Blueprint: `Interface_InteractComponent_Latch`

4.2.3 *Overlap Interface*

Components: Overlap Components
Blueprint: `Interface_InteractComponent_Overlap`

4.2.4 *Grabable Interface*

Components: Grab Component
Blueprint: `Interface_InteractComponent_Grabbable`

4.2.5 *Selectable Interface*

Components: Select Components
Blueprint: `Interface_InteractComponent_Selectable`

4.3 Miscellaneous Interfaces

4.3.1 *Deletable Interface*

Components: Delete Component
Blueprint: `Interface_Deletable`

4.3.2 Highlightable Interface

Components: Highlight Component
Blueprint: `Interface_Highlightable`

4.3.3 Grabbing Actor Interface

Enables an actor to grab another actor. Each actor that should be able to grab another act needs a grabbing actor interface including pawns or motion controllers.
Blueprint: `Interface_GrabbingActor`

4.3.4 I18n Interface

The I18n interface enables an actor displaying text to implement the update language function. Upon changing language settings each actor with the I18n interface (mostly widgets) automatically changes the text to the corresponding translation according to its I18n data table. For more information see section 7.2.
Blueprint: `Interface_I18n`

5 Environments

5.1 Pawn Hierarchy

The Framework relies on a quite intricate pawn class hierarchy (see scheme in the margins) which provides customized pawns for the different environments the Framework supports including desktop, VR and mobile. The most basic pawn is the `BP_Pawn_Base` which serves as parent class for all pawns. The VR pawn is the most singular of the pawn classes, since the VR environment requires vastly different features than the other environments.
Many features of desktop, mobile and asymmetric game play environments however, are very similar and were therefore condensed into the screen pawn (`BP_Pawn_Screen`) which serves as parent class for the desktop, mobile and asymmetric pawn. Further sub-classes expand the pawns features:

- Camera Pawns - do not have a virtual body
- Character Pawns - are equipped with a virtual body

```
Base Pawn
├─VR Pawn
│  └─Character VR Pawn
└─Screen Pawn
   ├─Assymetric Pawn
   │  ├─Assymetric Camera
   │  │  Pawn
   │  └─Assymetric Character
   │     Pawn
   ├─Desktop Pawn
   │  ├─Character Desktop
   │  │  Pawn
   │  └─Camera Desktop Pawn
   └─Mobile Pawn
      ├─Character Mobile
      │  Pawn
      └─Camera Mobile Pawn
```

Base Pawn

The base pawn is the most basic pawn class of the Framework and consolidates the common features of all pawns of the hierarchy. The base pawn is not meant to be used in any application.
Blueprint: `BP_Pawn_Base`
Features:

- Camera - places the perspective to the player position
- Gazeview - establishes the camera trace for interaction with the gazeview component (see section 3.1.2)
- Teleportation - allows the pawn to change location and rotation and establishes a number of variables to adjust teleportation style

```
Base Pawn
├─Screen Pawn
├─VR Pawn
└─...
```

Screen Pawn

The screen pawn consolidates the mutual features of the mobile, desktop and asymmetric pawn establishing controls without relying on the motion controllers.
Blueprint: `BP_Pawn_Screen`
Features:

- Movement - enables the pawn to navigate the experience without relying on the controls component
- Interaction - establishes selecting, grabbing and latching without a motion controller
- Navigation Modes - offer different navigation styles for mouse or touchscreen
  – Crosshair - uses the mouse to navigate the camera as in FPS-games
  – Free Mouse - shows and allows free movement of a cursor that is either controlled by the mouse or touchscreen
  – Touch - specifically adapted to navigate experiences on mobile devices

```
Base Pawn
└─Screen Pawn
   ├─Desktop Pawn
   └─...
```

The screen pawn like the base pawn is not intended for use in an experience, but merely serves as summary of the features shared by the desktop, mobile and asymmetric pawn.

The crosshair navigation mode is only available for the desktop pawn or the asymmetric pawn

The touch navigation mode can only be used for the mobile pawn

5.2 Startup

Upon starting the application the player controller checks the environment of the experience and chooses the pawn that is specified in the info level file for this environment (see Figure 41) using the following criteria:

- VR - the VR pawn is automatically chosen when a HMD is present upon the start of the experience and spawns motion controllers to move and interact with the experience in VR (see section 5.3)
- Desktop - the desktop pawn serves as default environment and provides functionalities to move and interact with the experience using mouse and keyboard (see section 5.4)
- Mobile - the mobile pawn is automatically chosen when the operating system is IOS or Android and provides functionalities to move and interact with the experience via touchscreen (see section 5.5)

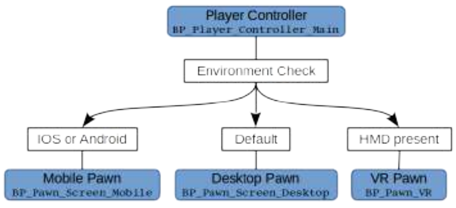


Figure 41. Startup and pawn selection.

5.3 Virtual Reality

The VR environment presents the developer with very specific challenges that are seldom encountered in other environments. This includes first and foremost the representation of the real live controllers in the application, but also movement, the implementation of UI elements without a HUD and simulation sickness.
The Advanced Framework uses motion controllers to represent the real-life controllers and motion components to implement most of their functionalities. This requires a way to connect the button input of a real-life controller to the functions of the motion components. This information flow is shown in Figure 42. Each button input on the real-life controller is transmitted through the pawn to its controls component (see section 3.6.1) which is the key element to design the VR pawn's controls. The controls component references a number of preset data assets that connect the button input of the real-life controller to a controller function, which the controls component transmits back to the VR pawn. Subsequently, the VR pawn passes on the controller function to the motion controllers which in turn pass it down to all motion components. Finally, the motion component matching the controller function implements the requested functionality.

Reminder

All pawn classes described in the environment sections are supposed to be used as parent classes for the pawns of an application.

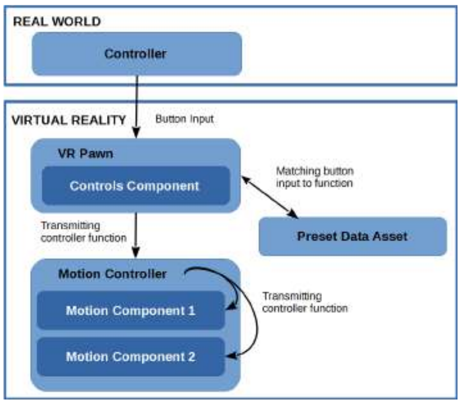


Figure 42. Information Flow.

5.3.1 VR Pawn

The VR pawn is adjusted to the requirements of VR applications. In contrast to the other pawns most features of the VR pawn are outsourced to motion controllers and motion components (see sections 5.3.2 and 5.3.3). The pawn mainly serves as vantage point of the player and to transmit the input signals from the real-life controller to the motion controllers.

```
Base Pawn
└VR Pawn
  ├Character VR Pawn
  └...
```

Blueprint: `BP_Pawn_VR`
Features:

- Motion Controllers - the VR pawn spawns a pair of motion controllers upon start of the experience that implement movement and interaction with the experience (see section 5.3.2)

- Input Forwarding - the VR pawn forwards all key inputs of the player to the motion controllers

5.3.2 Motion Controllers

Motion controllers are designed to represent the physical controllers in VR experiences. Each controller is composed of a motion controller and number of motion components. Since version 3.1 the motion controllers have evolved a more complex hierarchy. Especially the hand motion controllers are more complex now providing the possibility to design different controllers for each hand. Moreover, hand tracking is implemented in two additional classes inheriting from the hands motion controller.

In all other environments the controls are implemented by the pawn directly and motion controllers are obsolete.

```
Base Motion Controller
 ├─Controller Motion
 │  Controller
 │   ├─Laser Motion Controller
 │   └─Pause Motion Controller
 └─Hands Motion Controller
     ├─Left Hand Motion
     │ Controller
     ├─Right Hand Motion
     │ Controller
     └─Handtracking Motion
        Controller
         ├─Left Handtracking
         │ Motion Controller
         └─Right Handtracking
           Motion Controller
```

Base Motion Controller

serves as parent class for all motion controllers and is not supposed to be used as motion controller in an application. It mainly manages the motion components and transmission of controller functions or key inputs.
Blueprint: `BP_MotionController`
Included Motion Components: none

Motion Component Sockets

Since the controllers of various HMD are constructed very differently each motion controller is equipped with a number of sockets to ensure that motion components like the laser, teleport or radial menu are attached at the correct position relative to the controller mesh.

Controller Motion Controller

equips the base motion controller with the mesh of a controller matching the HMD and implements mini tags that display the button mapping.
Blueprint: `BP_MotionController_Controller`
Included Motion Components: none

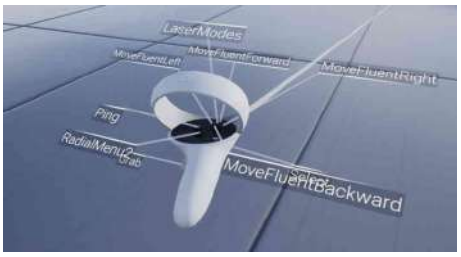


Figure 43. Controller Mini Tags.

Laser Motion Controller

equips the motion controller with a laser (implemented as motion component) as main means of interacting with other actors (see Figure 47).
Blueprint: `BP_MotionController_Controller_Laser`
Included Motion Components:

- Laser Pointer Motion Component - see page 5.3.3
- Radial Menu Motion Component - see page 5.3.3
- Movement Motion Component - see page 5.3.3
- Teleport Motion Component - see page 5.3.3

Pause Motion Controller

The pause pawn spawns its own motion controller with minimal functionalities.
Blueprint: `BP_MotionController_Controller_Pause`
Motion Components included: none

This means during the pause the controller will always match the HMD regardless of the appearance of the controller within the level.

Hands Motion Controller

equips the motion controller with the possibility to appear with a skeletal mesh of a hand. However, the hands motion controller mainly serves as parent class for all hands motion controllers.
Blueprints: `MC_Hands`
Included Motion Components: none

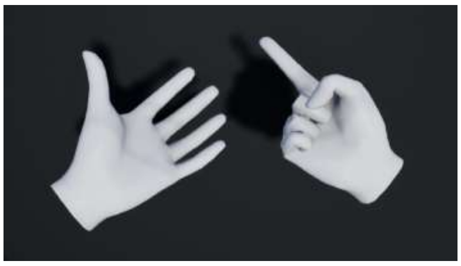


Figure 44. Hands Motion Controller

Flip the right hand mesh along the z-axis to create the left hand mesh.

Left and Right Hand Motion Controller

provide the motion controller with the skeletal mesh of a left or right hand and the corresponding animation blueprint.
Blueprints:

- `MC_Hands_Left`
- `MC_Hands_Right`

Included Motion Components:

- Grab Motion Component - see page 50
- Laser Motion Component - see page 49
- Ping Motion Component - see page 53
- Radial Menu Motion Component - see page 51
- Movement Motion Component - see page 51
- Teleport Motion Component - see page 51

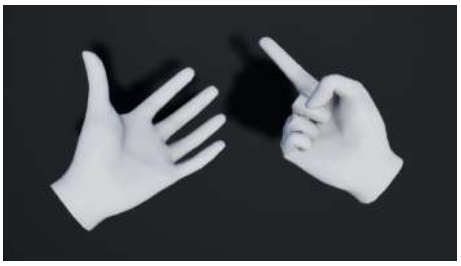


Figure 45. Meshes for Hand-tracking

Hand-Tracking Motion Controller

Upon start of hand-tracking the hand-tracking motion controllers substitute the original motion controllers. Hand-tracking motion controllers can recognize gestures defined by a gesture data asset (see section **??**) and match them with controller functions as defined in the controls component (see section 3.6.1) similar to button mapping for normal motion controllers.
Blueprint: `BP_MotionController_Hands_Tracking`
Included Motion Components: none

How To Set Up Hand-Tracking

Hand-tracking works completely different from the normal VR controls. The controls component on the pawn has its own section controlling it. Here are the basic steps to set everything up:

- Motion Controllers - Hand-tracking has is own motion controllers. Enter them under Hand-tracking motion controllers on the controls component
- Gestures - For the Hand-tracking motion controller to recognize a gesture must be defined in a gesture data asset.
- Function Mapping - the controls component contains a map which assigns a controller function to each gesture DA for each hand
- Preset DA - Hand-tracking still needs a preset to define teleport controls

Left and Right Handtracking Motion Controller

provides the hand tracking controller with a left or right hand skeletal mesh that is matched to the human hands, so they can dynamically follow the players hand and finger positions during runtime.
Blueprints:

- `BP_Controller_Hands_Tracking_Left`
- `BP_Controller_Hands_Tracking_Right`

Included Motion Components:

- Grab Motion Component - see page 50
- Ping Motion Component - see page 53
- Movement Motion Component - see page 51
- Teleport Motion Component - see page 51

5.3.3 Motion Components

Motion components apply the component system to the motion controllers. Consequently, custom motion controllers can be designed in a similar fashion as new actor classes. Motion components frequently interact with other components to perform complex functionalities like grabbing or selecting just like actor components cooperate among each other, see Figure 46 for a schematic example. The AF Core contains overall 8 custom made motion components.

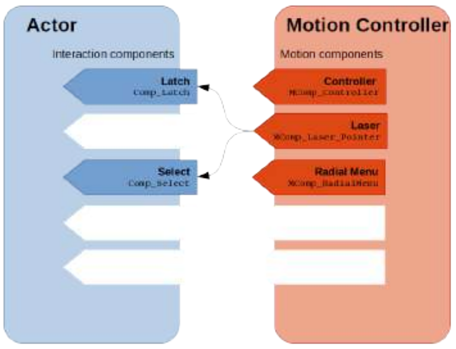


Figure 46. Motion Component System.

- Laser Motion Component - attaches a laser pointer to the controller
- Finger Motion Component - enables selection with the index finger
- Grab Motion Component - manages grabbing with the hands motion controller
- Hand Laser Motion Component - more simple laser motion component adjusted for the hand motion controller
- Radial Menu Motion Component - spawns a set of radial menu buttons
- Ping Motion Component - spawns a visual effect to indicate actors, direction etc.
- Movement Motion Component - allows the player to navigate the application by moving the pawn
- Teleport Motion Component - enables the player to navigate the application by teleporting the pawn

Laser Motion Component

The laser motion component attaches a laser pointer to the controller that emits a laser trace and provides the the player with a possibility of remote interaction.

Blueprint: `BP_MComp_Controller_Laser`
Socket Preference: MComp_Laser
Suitable Motion Controllers: Controller Motion Controllers
Controller Functions:

- Select - toggles the select interaction component (see section 3.1.1) on an actor
- Grab - toggles the grab component (see section 3.1.3) and allows the motion controller to attach the actor to itself
- LaserMode - switches laser mode (see info box below)
- Ping - spawns a visual effect at the laser impact location

Settings:

- Max. Laser Length - maximum length of the laser trace
- Max. Interpolation Distance - define from which distance between the original trace impact location and the new trace impact location (when the controller is moved) the laser movement is no longer interpolated and the new location is just set instead
- Pointer Transform Interaction - sets the default laser mode
- Interpolations Speed - determines how fast the laser trace follows the controller movement

Controller Functions

Most motion components implement one or more controller functions. Upon selecting motion components for a controller it is necessary to ensure, that each controller function is only implemented on *one* motion component to avoid incompatibilities.

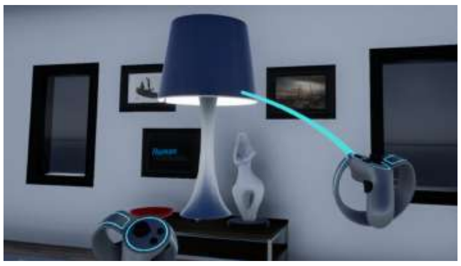


Figure 47. Laser Motion Controller.

- Max Grab Distance Offset - maximum distance you can move a grabbed actor away from the pawn
- Preferred Socket to Attach to - defines the socket on the mesh of the motion controller, where the laser should be attached

Laser Modes

Actors grabbed by the laser motion controller do not rotate with the controller. Instead the player can access the transform of the actor according to the preset data asset. The Framework provides 3 modes:

- distance + rotation: the thumbstick´s y-axis moves the actor nearer and farther from the pawn and the thumbstick´s x-axis rotates the actor around its z-axis.
- two axis rotation: the thumbstick´s x-axis rotates the actor around its z-axis. and the thumbstick´s y-axis rotates the actor around its y-axis.
- distance + scale: the thumbstick´s y-axis moves the actor nearer and farther from the pawn and the thumbstick´s x-axis scales the actor.

Grab Motion Component

enables the motion controller to grab or latch onto actors.
Blueprint: `BP_MComp_Hand_Grab`
Socket Preference: none
Suitable Motion Controllers: hands motion controller
Controller Functions:

- Grab - toggles the grab component (see section 3.1.3) and latch component (see section 3.1.5

Settings:

- Search Distance - distance from the GrabSearch Socket location on the hand motion controller in which the motion component searches for actors
- Search Radius - determines the radius of the search trace
- Pull Speed - defines how fast the object moves towards the hand motion controller upon grabbing
- Preferred Socket to Attach to - defines the socket on the mesh of the motion controller, where the laser should be attached

Additional Setup for Grab and Latch

When the VR hands try to grab or latch onto an actor, the hand searches the mesh of the actor for sockets and relates the most suitable socket to the motion controller. Generally, this will be the closest compatible socket. This implementation allows the possibility for the actor to snap onto preferred locations without having to make specific meshes with weird pivot locations. For the VR hands to look naturally while grabbing an actor the first 4 characters of the name of each socket must follow this convention:

X_NN

In this X is either the letter `L` or `R` assigning the socket to the left VR

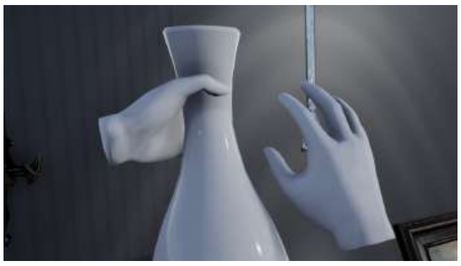


Figure 48. Grabbing a Vase with `L_00`

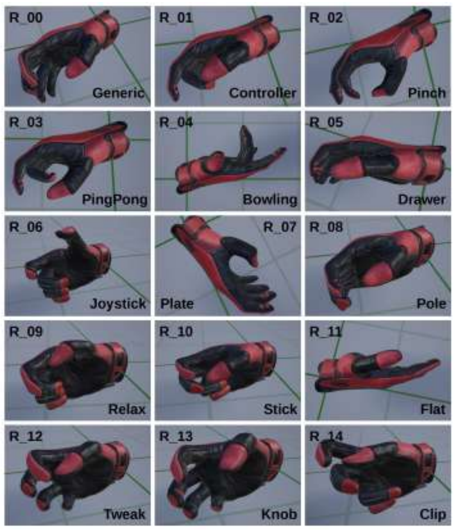


Figure 49. Hand Grab Positions.

Hand (L) or the right VR Hand (R). And NN a number between 00 and 13 assigning the posture of the VR Hand as shown in Figure 49.

Radial Menu Motion Component

The radial menu motion component spawns a circular set of buttons that can be selected either by the thumbstick at the same hand or the laser of the other hand. For more information see section 8.3.2.
Blueprint: `BP_MComp_RadialMenu`
Socket Preference: MComp_RadialMenu
Suitable Motion Controllers: all
Controller Functions:

- RadialMenu1 - spawns a radial menu according to the radial menu 1 settings of the radial menu component on the pawn
- RadialMenu2 - spawns a radial menu according to the radial menu 2 settings of the radial menu component on the pawn

Settings:

- Preferred Socket to Attach to - defines the socket on the mesh of the motion controller, where the laser should be attached

Teleport Motion Component

The teleport motion component enables and controls teleportation
Blueprint: `BP_MComp_Teleport`
Socket Preference: MComp_Teleport
Suitable Motion Controllers: all
Controller Functions:

- Teleport - executes the teleport of the pawn according to the settings on the controls component (see section 3.6.1)

Settings:

- Max Trace Length - determines the maximum distance of the teleport search trace from the pawn
- Max. Interpolation Distance - define from which distance between the original trace impact location and the new trace impact location (when the controller is moved) the laser movement is no longer interpolated and the new location is just set instead
- Interpolations Speed - determines how fast the laser trace follows the controller movement
- Preferred Socket to Attach to - defines the socket on the mesh of the motion controller, where the laser should be attached

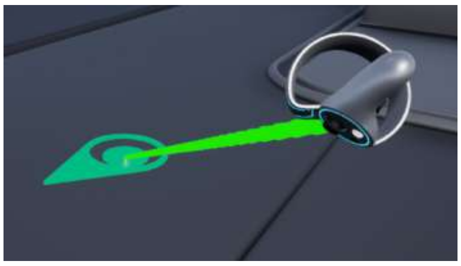


Figure 50. Free teleport with rotation by hand.

Movement Motion Component

The movement motion component enables and controls all movement options except teleportation
Blueprint: `BP_MComp_Movement`
Socket Preference: none

Suitable Motion Controllers: all

Controller Functions:

- Walk - moves the pawn along the x- and y-axis
- Fly - moves the pawn freely along all 3 axis
- TurnLeftStep - turns the pawn to the left in steps of a given angle
- TurnRightStep - turns the pawn to the right in steps of a given angle
- TurnLeftFluent - turns the pawn to the left fluently
- TurnRightFluent - turns the pawn to the right fluently
- MoveFluentForward - moves the pawn forward fluently
- MoveFluentBackward - moves the pawn backward fluently
- MoveFluentRight - moves the pawn to the right fluently
- MoveFluentLeft - moves the pawn to the left fluently

Settings:

- Preferred Socket to Attach to - defines the socket on the mesh of the motion controller, where the laser should be attached

Ping Motion Component

enables the motion controller to spawn a visual effect to indicate actors, targets or directions.

Blueprint: `BP_MComp_Ping`

Socket Preference: none

Suitable Motion Controllers: all

Incompatible with laser pointer motion component.

Controller Functions:

- Ping - spawns the visual effect

Settings:

- Preferred Socket to Attach to - defines the socket on the mesh of the motion controller, where the laser should be attached

Hand Laser Motion Component

The hand laser motion component is a down-graded version of the laser pointer adjusting it for use on the hands motion controller.

The laser pointer implements a number of controller functions, that may interfere with other controller functions designed for the hands motion controllers.

Blueprint: `BP_MComp_Hand_Laser`

Socket Preference: none

Suitable Motion Controllers: hands motion controller

Controller Functions:

- Select - interacts with the select component (see section 3.1.1) on an actor

Settings:

- Max Laser Length - determines the maximum distance laser trace from the pawn
- Laser Start Distance - distance from the hand motion controller from which the laser starts to search for impact
- Preferred Socket to Attach to - defines the socket on the mesh of the motion controller, where the laser should be attached

Finger Motion Component

positioned at the tip of the index finger the finger motion component allow the hand motion controller to detect forward movement of the outstretched index finger and interact with the select component on actors.

Blueprint: `BP_MComp_Hand_Finger`
Socket Preference: none
Suitable Motion Controllers: hand motion controller
Controller Functions: none
Settings:

- Preferred Socket to Attach to - defines the socket on the mesh of the motion controller, where the laser should be attached

5.3.4 Presets

Preset data assets provide the button mapping that connects a key input to a functionality of the motion controller. The framework already provides a number of different presets for the controllers of different HMD. For more information see section 7.1.1.
Content: see section 7.1.1.

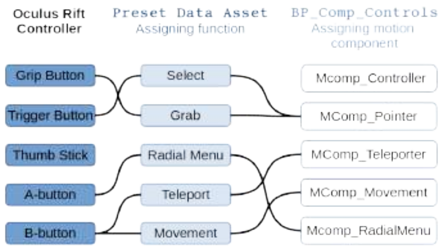


Figure 51. Button Mapping Example

5.3.5 Asymmetric Game Play

Asymmetric game play mixes the VR and desktop environment to allow two players to experience a VR application when only one HMD is available. The AF Core provides two methods to implement asymmetric game play:

- Spectator Mode - allows the implementation of vantage points from which a second player can monitor the player using the HMD navigate the application on the desktop.
- Asymmetric Pawn - spawns a complete screen pawn, that can move and interact with the level independently controlled by mouse and keyboard.

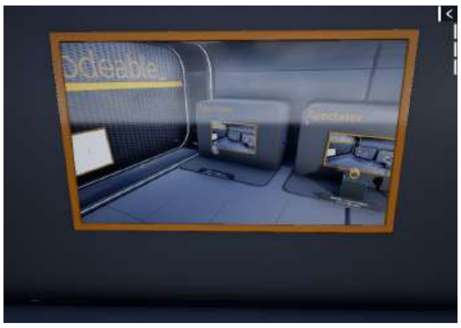


Figure 52. Spectator Mode.

Spectator Mode

Spectator mode is based on the spectator component which can be added to any actor providing additional vantage points. The camera feed of the spectator component can either be accessed directly on the desktop or transmitted to a in-application screen.

Only one spectator component can be active per level at a given time. Upon activating another spectator component the previous one is deactivated automatically.

Overview On Setup

- On the actor - add the spectator component and set it up according to section 3.8.5. The spectator component itself includes a camera, so it is not necessary to add a camera to the actor
- Activating the Camera Feed - the spectator component relies on an event or key input for activation.
- Showing the Camera Feed - the camera feed of the spectator component is show on the real life monitor automatically upon when the

component is activated. For showing camera feeds within the VR experience create a 2D render target to create a dynamic material.

Asymmetric Pawn

serves as a additional pawn, that is controlled by keyboard and mouse and can navigate the experience.
Blueprint: `Pawn_Screen_Asymmetric`
Controls: see desktop pawn

```
Base Pawn
└Screen Pawn
  ├...
  └Asymmetric Pawn
    └Character Asymmetric
     Pawn
```

Overview On Setup

- On the VR Pawn - the VR pawn provides the "spawn asymmetric pawn" function that only needs to be called and provided with an position and pawn class
- On the Asymmetric Pawn - the asymmetric pawn is a screen type pawn and can be provided with all components and functions that are compatible with the screen environment
- Asymmetric Pawn Camera Feed - the asymmetric pawns vantage point is automatically shown on the real life monitor. It can be displayed in the VR application by using a render target as briefly described above.

5.4 Desktop

The desktop environment is the most widespread and variable of all environments supported by the Framework including (in theory) a number of input devices like mouse, keyboard or game pad.

Desktop Pawn

The desktop pawn serves as default pawn for all screen applications. The desktop pawn (or its child classes) by now provide all functionalities to navigate and interact with experiences using mouse and keyboard.
Blueprint: `BP_Pawn_Screen_Desktop`

```
Base Pawn
└Screen Pawn
  └Desktop Pawn
    ├Character Desktop
    │Pawn
    └Camera Desktop Pawn
```

Controls

- Locomotion - the WASD keys of the keyboard move the pawn according to the specifications of the navigation mode
- Locomotion Modes - the desktop pawn support the movement modes sprint, crouch, jump and fly assigning the following keys
  - Space - jump
  - Shift - sprint (to keep sprinting hold the key)
  - c - crouch (hit the key to enter/exit crouch mode)
  - y - fly (hit the key to enter/exit fly mode)
- Change Navigation Mode - the alt key changes the navigation mode between crosshair and free mouse
- Select - the pawn selects an actor by clicking the left mouse button.
- Grab/Latch - the pawn grabs an actor by holding the left mouse button or tapping the right mouse button

Game pad controls are still under construction.

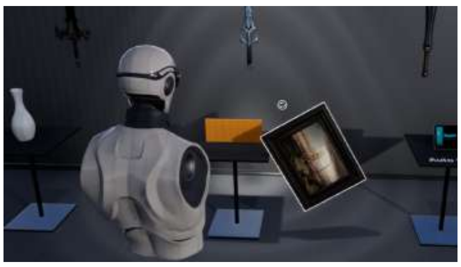


Figure 53. Character Desktop Pawn.

- Rotate grabbed actor - holding the right mouse button while moving the mouse rotates a grabbed actor.

5.5 Mobile Devices

The most notable difference between mobile and desktop environments are the limited capacities of mobile devices and their reliance on touch screen as main input source. Additionally the small screen can only support a minimum of HUD elements.

Mobile Pawn

```
Base Pawn
└─Screen Pawn
   ├─...
   └─Mobile Pawn
      ├─Character Mobile
      │ Pawn
      └─Camera Mobile Pawn
```

The mobile pawn is a screen-type pawn with controls adjusted for input by touchscreen.
Blueprint: `BP_Pawn_Screen_Mobile`
Controls:

- Movement - a double tap on the floor teleports the pawn to the selected location
- Camera Movement - swiping moves the camera of the mobile pawn
- Select - tab on a selectable actor
- Grab/Latch - tab and hold on a grabable/latchable actor

6 Multiplayer

Creating a multiplayer application is an intricate task and includes lots of issues that are outside the scope of the Advanced Framework. Thus, this chapter is by no means a comprehensive discussion of the topic. It mainly presents the modifications we made in the components and other features of the AF Core to allow you to set up a multiplayer application. This includes first and foremost replication and ownership and also a basic lobby to set up multiplayer games.

6.1 Client Server Models and Ownership

The Framework supports at the moment 2 client server models:

- Dedicated Server - does not have an assigned player controller and only hosts the game (see Figure 54, top)
- Listening Server - has an assigned player controller and hosts while playing (see Figure 54, bottom)

Ownership basically constitutes which client if not the server can execute replicable events including changing states, location and more on an actor. Therefore specific interactions especially grab and latch require a change of ownership to be executed.

Generally it is beneficial if most actors are owned by the server and changes of ownership are kept to the absolute minimum.

The client generally owns:

- the pawn
- actors permanently attached to the pawn like the motion controllers
- grabbed and latched actors
- all actors spawned by the client

The server generally owns:

- all actors in the level (exceptions: see above)
- all actors spawned by the server

Changing Ownership

Actor ownership changes mainly when a player grabs or latches onto an actor. In the AF Core a grabbed/latched actor is always owned by the client of the player who executed the grab or latch. Upon release the ownership is not restored to the previous owner to limit ownership changes.

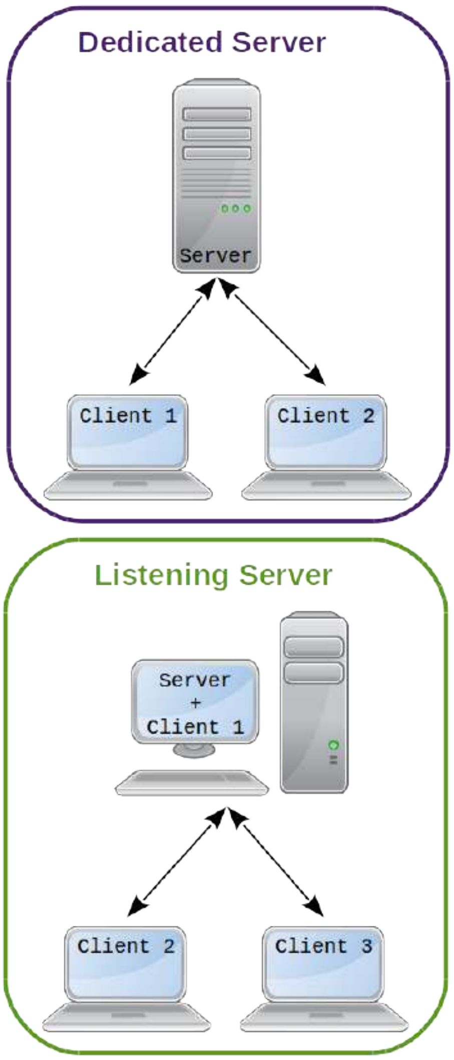


Figure 54. Client Server Models.

6.2 Replication

All actors whose state, position or other properties are relevant in multiplayer must replicate to ensure synchronisation between the clients. This includes spawning and destroying actors. The AF Core provides a number of solutions to deal with replication in multiplayer applications.

Specifically the extensive use of components is intended to facilitate replication.

Regardless of the client server model and instigator all replication processes go via the server and the amount of replication processes is strictly limited by the network bandwidth.

6.2.1 State Components and Interaction Components

Both state components and interaction components like the select component or the active component implement the ability to replicate providing the owning actor is set to replicate. The component system has been particularly adjusted to provide the basis for multiplayer applications. This concerns especially the controls component on the pawns which was designed to enable the clients to access the controls settings.

6.2.2 Non-predetermined Movement

The movement of actors and components is replicated by the replication components described in section 3.4. These components were specifically designed to replicate non-predetermined movement caused by pawns grabbing actors or physics. They are also meant to replicate the movement of the pawn through the level.

There is no need to replicate predetermined movement like the hands of a clock or the swinging of a pendulum for example, since predetermined movement will automatically be synchronous on all clients and the server provided it is started at the same moment.

The replication of location and rotation by the replication components is a very resource intensive process and should be used as seldom as possible.

6.2.3 Spawning and Deleting of Actors

Specifically the spawn process interlinks heavily with the question of ownership, since each actor is owned by the entity instigating the spawn process which might either be the server or a client. This has serious consequences on the ability of the actors to replicate, since only spawn processes instigated by the server are replicated. This is implemented in the spawn components (see sections 3.8.3 and 3.8.4) which should be exclusively used to spawn actors in multiplayer applications, since actors spawned by a client only exist in this client's instance of the application. The deletion of an actor should be exclusively handled by the delete component that is equipped to replicate the process.

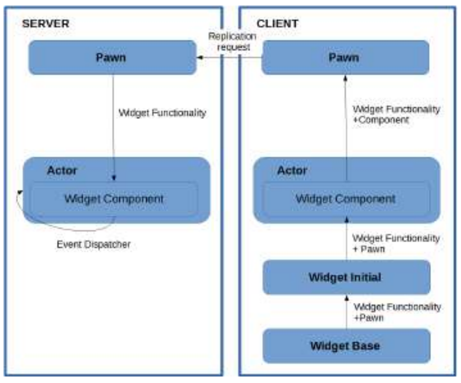


Figure 55. Replication of Widget Functionalities.

6.2.4 User Interfaces

Regarding multiplayer applications, user interfaces present a number of issues especially in VR. UI elements like the pallets or menu buttons must be spawned and also frequently incorporate widgets to display text, buttons and images. The AF Core provides a number of UI elements, however only a few of them are compatible with multiplayer.

For more details on UI see section 8.

- Full Replication - spawned by the server, no widgets

  - Selection Menu - all buttons are spawned by the server and as actors themselves contain a select component that replicates normally
  - Radial Menu - all buttons are spawned by the server and as actors themselves contain a select component that replicates normally
- Partial Replication - spawned and interacted with via the server, contains widgets
  - Pallet - spawned by the server on request of the instigating client. Furthermore, all interactions with the pallet are redirected through the server, too, to enable all players to interact with the pallet regardless who spawned it
  - Info Window - spawned by the server on request of the instigating client, so all players can share the information provided
- No Replication - spawned by the client, contains widgets
  - HUD - cannot be shared with other players

Widgets

Widgets have never been programmed to function in a multiplayer environment. Widgets can only used on clients and are unable to replicate within themselves. Only widget functionalities that trigger functions on the actor can be replicated via the information flow shown in Figure 55. This is an intrinsic property of widgets the Advanced Framework **cannot** change.

7 Data

The Advanced Framework supports a number of methods to manage and provide static data as well as orderly data transmission. Not all data entities especially in the categories of structs and enums are essential for developers. Some are only used in a very specific way and better described with the component or other entity which uses them. Others are only of internal use. Therefore, we will limit the description of individual data entities to the strictly necessary.

7.1 Primary Data Assets (PDA)

Primary Data Assets (PDA) are basically a prearranged set of variables to be used to store, arrange and access a variety of information. As a consequence, PDA fill a similar role as data tables with the added advantage of being more intuitive to manage. The AF Core 4.1 provides the following types of PDA:

- Preset PDA - provide the button mapping for the VR motion controllers (see also sections 3.6.1 and 5.3.2)
- Panel PDA - determines the content and appearance of the panels in the panel based menu (see also section 8.2.2)
- Level PDA - store basic information on each level like the pawns (see also section 2.2)
- Gesture PDA - store the finger and hand positions that describe a gesture to be used for handtracking

Data Assets follow a hierarchy with the data asset class on the top parenting the primary data assets which parent PDA instances in turn. The difference between a PDA and a PDA instance you can imagine like the difference between an empty form and a form that has been filled in by a person. However, for simplicity's sake we will use the term data asset roughly for PDA as well as PDA instances within the documentation.

How To Use PDA

Primary Data Assets are basically templates which consolidate a number of variables of any kind. To create a data asset for a specific use case, generate a PDA instance using the corresponding PDA as reference and fill out all variables.

```
Data Assets
└─Primary Data Assets
   └─PDA-Instance
```

Building PDA

To take PDA along into a build they have to be entered into the asset manager in the project settings. This is a vital step. Any PDA (not PDA instance) even if its the child of a registered PDA needs its own entry. Any obmitted PDA will not make it into the build application causing all kinds of problems.

7.1.1 *Preset Data Assets*

Preset DA provide the button mapping that connects a key input to a controller function (see margin) of the motion controller. Consequently presets are only of importance for VR applications.

Primary Data Asset (PDA): `PDA_ControllerVR`
Used in: controls component (see section 3.6.1)
Controller Settings:

- Trigger Grip - grab is triggered by the assigned and held by the assigned button. To release the assigned button must be pressed again.
- Holding Grip - to hold an item continuously the assigned button must be held. Upon release of the button the grabbed actor is released, too.
- Key Mapping - contains one element for the button mapping of each controller.
  - Key - determines if the button mapping is for the left hand motion controller or the right hand motion controller

List of Controller Functions

- Select
- Grab
- LaserMode
- Ping
- Pause
- RadialMenu1
- RadialMenu2
- ToggleFly
- Teleport
- Move
- MoveForward
- MoveLeft
- MoveRight
- MoveBackward
- TurnLeft
- TurnRight
- ResetOrientation

  - Value - contains an element that connects each key of the HMD controller to a controller function (by name) provided by the corresponding motion component.

Haptics And Teleport Settings:

- Haptics Multiplier - serves to adjust the intensity of haptic response of the controller

The haptics intensity changes a lot with different HMD. Use this to keep your response consistent for each HMD.

- Teleport Execute - specifies which button initiates the teleport to the impact location of the trace
  - OnTrigger - the trigger button initiates the teleport
  - NeutralThumbstick - teleport is initiated automatically when the thumbstick returns to its origin position
  - OnTeleportButtonRelease - teleport is initiated automatically when the button assigned to change to teleport mode is released again

Cannot be used in combination with the NeutralThumbstick Teleport Deactivate setting

- Teleport Deactivate - specifies how the teleport mode can be exited
  - OnlyOnExecute - excludes the possibility to deactivate teleport mode
  - NeutralThumbstick - deactivates teleport mode when the thumbstick returns to its origin position.

This comprises that the button which activates teleport mode must be held while searching for the desired location.

Cannot be used in combination with the OnThumbstickRelease Teleport Execute setting

- Teleport Rotation Type - specifies if and how the rotation of the pawn during teleportation is implemented
  - None - the pawn will not rotate upon teleportation
  - RotateByHand - the pawn will rotate upon teleportation according to the rotation of the motion controller towards its upright position, see Figure 50
  - RotateByThumbstick - the pawn will rotate according to the position of the thumbstick.

Currently Available Preset DA

Preset DA can be used to adapt an application for different HMD. The AF Core provides the following presets:

Oculus Preset	button mapping for the oculus touch controller
Vive Preset	button mapping for the vive wand controller
Knuckles Preset	button mapping for the knuckles index controller
MR Preset	button mapping for the windows mixed reality controller
Cosmos Preset	button mapping for the Vive Cosmos controller

7.1.2 Level Data Asset

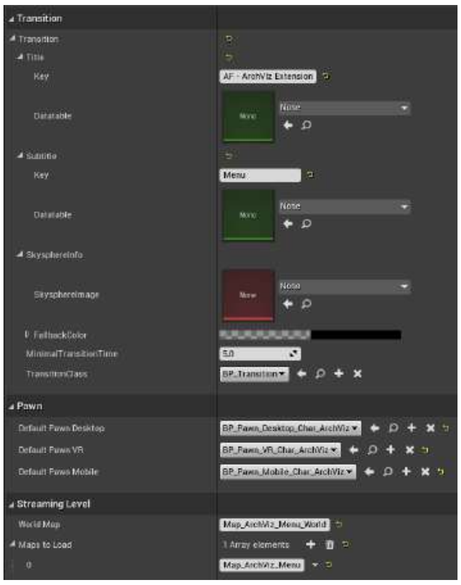
The level DA replaced the former info level files completely containing all essential information on a level. Each level needs its own level DA, that is referred to by the map info actor in level map.

Primary Data Asset (PDA): `PDA_Level`

Used In: `BP_MapInfo` (section 2.2)

Settings:

- Transition - determines the look of the transition screen while the level is loaded



```
Transition
  Transition
    Title
      Key                     AF - ArchViz Extension
      Datatable               None
    Subtitle
      Key                     Menu
      Datatable               None
    SkysphereInfo
      SkysphereImage          None
    FallbackColor
    MinimalTransitionTime     5.0
    TransitionClass           BP_Transition
Pawn
  Default Pawn Desktop        BP_Pawn_Desktop_Char_ArchViz
  Default Pawn VR             BP_Pawn_VR_Char_ArchViz
  Default Pawn Mobile         BP_Pawn_Mobile_Char_ArchViz
Streaming Level
  World Map                   Map_ArchViz_Menu_World
  Maps to Load                1 Array elements
    0                         Map_ArchViz_Menu
```

Figure 56. Level Data Asset.

  - Title - contains the key and I18n data table for the level title
  - Subtitle - contains the key and I18n data table for the level subtitle

If no datatable is entered or the key is missing the key itself is displayed in the application.

  - Skysphereinfo - specifies the texture used for the skysphere and the fallback color if no image was found
  - MinimalTransitionTime - states the time in seconds the transition screen is shown at minimum

To progress to the level as soon as it is loaded enter 0.

  - Transition Class - specifies which `BP_Transition` actor is spawned upon loading a level (see section 2.4.2).

If no transition blueprint is specified the `BP_Transition_Default` is spawned.

- Pawn - assigns the pawns for the level depending on environment
  - Default Pawn Desktop - defines which pawn is used in a desktop environment

If the environment could not be determined the Default Desktop Pawn is chosen

  - Default Pawn VR - specifies which pawn is used, when a HMD is present
  - Default Pawn Mobile - states which pawn is used when the operating system is Android
- Streaming Level - consolidates the maps that need to be loaded for the level
  - World Map - refers to the world map of the level

Refer to each map by the full file name

  - Maps To Load - contains an element for each map file, that must be loaded for this level

7.1.3 Panel Data Asset

Panels DA consolidate the content of panel type widgets in the panel based menu. The Panel PDA form a hierarchy as shown in the margins of which only the Level Panel PDA should be used as templates for DA. Each Level Panel DA specifies the contents of one panel of the panel based menu (see section 8.2.2). Consequently, each level that should be loadable from the panel based menu needs its own panel level DA. The AF Core provides two panel designs, the small level panel and the large level panel, which mainly differ in the collocation of its content.

```
Panel PDA
├─Large Panel PDA
│ └─Large Panel Level PDA
└─Small Panel PDA
  └─Small Panel Level PDA
```

Small Level Panels

consist of one main panel and 2 side panels, that expand upon select. An expanded small level panel is shown in Figure 57 with main panel content in violet and side panel content in green.

Settings:

- Image Large01 - texture for the image on the left side panel
- Image Small01 - texture for the left image on the right side panel
- Image Small02 - texture for the right image on the right side panel
- Level Info - refers the level DA of the level that can be accessed by this panel
- Slogan - specifies the level title on the main panel
- Details - content of the text field on the right side panel
- Logo - texture for the small image on the main panel
- Cover - texture for the large image on the main panel
- Color - background color of the panel

Recommended Resolutions

- Large: 1600x975
- Small: 800x585
- Logo: 250x115
- Cover: 650x650

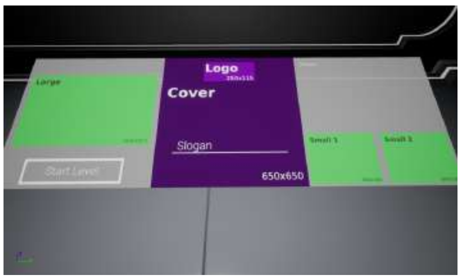


Figure 57. Small Panel Example.

- Left Content Widget - widget displayed in the left side panel
- Right Content Widget - widget displayed inf the right side panel

Large Level Panels

consist of one main panel and one side panel, that expands upon select. An expanded large level panel is shown in Figure 58 with main panel content in violet and side panel content in green.
Settings:

- Image Large01 - texture of the top window in the side panel
- Image Small01 - texture of the bottom left window in the side panel
- Image Small02 - texture of the bottom right window in the side panel
- Level Info - refers the level DA of the level that can be accessed by this panel
- Slogan - specifies the level title on the main panel
- Details - specifies the left text field in the main panel
- Additional Text - specifies the right text field in the main panel
- Logo - texture for the small image on the main panel
- Cover - texture for the large image on the main panel
- Color - background color of the panel
- Content Widget - specifies the widget shown in the side panel

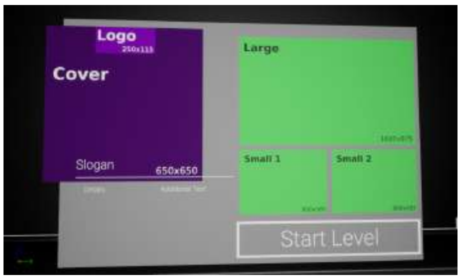


Figure 58. Large Panel Example.

7.1.4 Gesture Data Assets

Each Gesture PDA instance defines a hand gesture via its finger and hand position toward one of 3 points of reference. In the controls component of the pawn (see section 3.6.1) gesture DA can be assigned to controller functions enabling the player to perform functionalities like select, grab and many others in hand-tracking mode.
Primary Data Asset (PDA) : `PDA_Gesture`
Used In: `Comp_Controls` (section 3.6.1)
Settings:

- Finger Requirements - array containing an element describing the position of each finger
  - Finger - defines the finger this requirement is assigned to
  - Position - defines the position of the finger
    - *Extended180* - extended finger at 180° angle to the palm
    - *Extended90* - extended finger at 90° angle to the palm
    - *ContractedInward* - finger contracted and touching the palm
    - *ContractedOutward* - finger contracted without touching the palm
    - *Pinch* - touch the tip of the thumb of the same hand
  - Toleance - deviation in cm from the standard position used for handtracking.
  - Not - converts the requirement to a taboo, meaning all positions except the one specified are accepted to create the gesture
- Hand Requirements - array containing an element describing the hand position towards a point of reference

NOTE

1) Gesture PDA are limited to single hand gestures although you can define the position relative to the other hand.
2) Some gestures are non-overlapping mirror images and **cannot** described in one gesture PDA instance which works for both hands. In these cases we recommend creating a separate gesture DA for each hand.

This covers for different hand sizes and bad hand coordination.

- Hand Direction - defines the direction of the hand which is used for comparison
- Other Component - determines which entity serves as point of reference for the hand position
  - *Pawn* - uses the pivot of the pawn as reference
  - *Camera* - uses the direction the pawn looks in as reference
  - *OtherHand* - uses the pivot of the other hand as reference
- Relative To - defines the direction of the point of reference to which the hand direction needs to be parallel
- Angle Tolerance - deviation in degree from the specified position
- Not - converts the requirement to a taboo, meaning all positions except the one specified are accepted to create the gesture

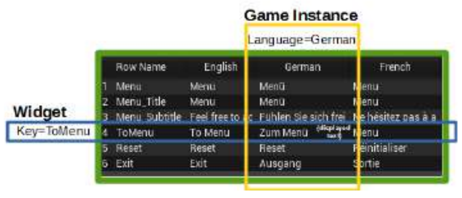
This covers for different hand sizes and bad hand coordination.

7.2 Data Tables

The AF Core uses data tables mainly for providing the update language function of its widgets with translations. The data tables provided for that are called I18n data tables.

I18n Data Tables

I18n data tables are separated into rows. Each row is assigned a key which together with the current language setting unequivocally points to one cell of the data table which provides the text which is finally displayed by the widget.
Basis Struct: `Struct_I18n_Key`



	Row Name	English	German	French
1	Menu	Menu	Menü	Menu
2	Menu_Title	Menu	Menü	Menu
3	Menu_Subtitle	Feel free to c	Fühlen Sie sich frei	Ne hésitez pas à a
4	ToMenu	To Menu	Zum Menü	Menu
5	Reset	Reset	Reset	Réinitialiser
6	Exit	Exit	Ausgang	Sortie

Figure 59. I18n Data Table.

Note
While it may seem unnecessary we recommend the use of multiple I18n data tables organized by usage of the texts tabulated, since huge data tables are very difficult to maintain.

7.3 Structs

Structs combine a number of variables that are usually transmitted or used together to a set. The Framework makes use of this concept to standardize translatable texts, communication between components and other functions. Many structs of the AF Core are only used internally and are generated automatically. However, others are used especially in the component settings. So we will provide selection of AF Core structs here.

I18n Key Struct

Used for practically all text displays in the AF Core, the I18n key struct enables the data table based update language function implemented in most widgets.
Full Designation: `Struct_I18n_Key`
Content:

- Key - defines one row of a I18n data table which is used for translating the text
- Data Table - determines in which data table the function searches for the key

For more information on the I18n data tables used for translation see section 7.2.

Graphic Button Struct

Used in UI elements like the selection menu (see section 8.3.1) to define the appearance of a button.

Full Designation: `Struct_Button_Graphic_2D`

Content:

- **MainTexture** - specifies a texture to be shown on the button
- **Material** - specifies a material for the button
- **Color** - defines a color for the button using the linear color struct of the Unreal Engine

Not recommended for 3D buttons.

The different parts of the struct are used in succession with the material replacing the texture if it was not found. And the color acting as ultimate fallback in case neither a texture nor a material are defined or are found.

Text Button Struct

Used in most widgets the text button struct provides a button with a name and texture.

Full Designation: `Struct_Button_Text`

Content:

- **ButtonText** - contains a I18n key struct to display a translatable button name
- **ButtonImage** - determines the texture of the button

Visual Mesh Struct

The visuals component (see section 3.2.11) allows to provide an actor with different looks and material sets which are defined via a set of interlocking structs. The most basic of these structs is the visual mesh struct which contains all necessary information on one look of the actor.

Full Designation: `Struct_Visual_Mesh`

Content:

- **Name** - I18n Struct providing a translatable name for the look
- **Static Mesh** - static mesh corresponding to the look
- **Skeletal Mesh** - skeletal mesh corresponding to the look
- **Materials** - array of visual material structs describing the different material sets available for the look
- **Button2D** - struct defining the look of the button accessing the look
- **DataAsset** - data asset corresponding to the look

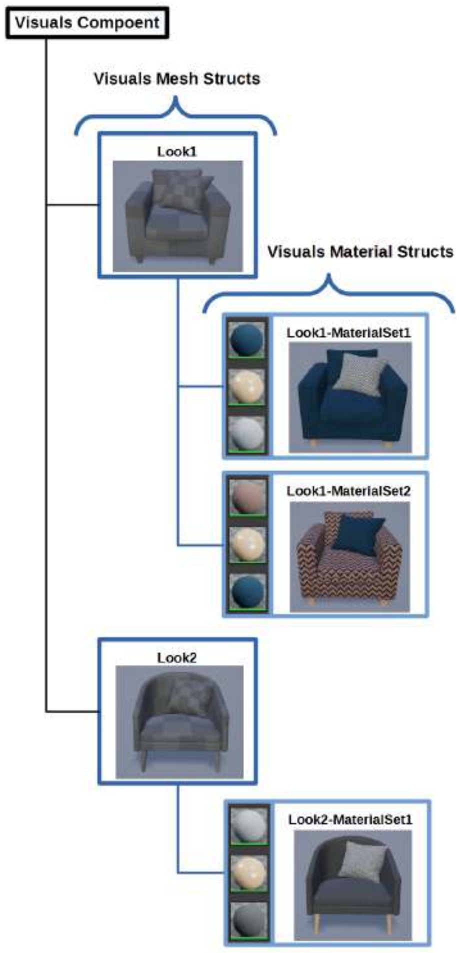


Figure 60. Visual Structs Scheme.

Each look must either have a corresponding static or skeletal mesh. It is *not* possible to mix or enter both a skeletal and static mesh.

Visual Material Struct

Referenced in the visual mesh struct, the visual material struct serves to describe one of various material sets belonging to one look. Each material set is composed of all materials used for the mesh even if they do not change between different material sets.

Full Designation: `Struct_Visual_Material`

Content:

- **Name** - I18n Struct providing a translatable name for the materials set

- ButtonContent - Button struct determining how the button accessing the material set looks like
- Materials - array of materials comprising the material set

7.4 Enums

Enums provide a modifiable list of keys that can be used to switch functions or be displayed in multiple parts of the application. Most enums used by the AF Core are best explained in their inherent environment meaning the component or UI or other element they are used in. However, there a small number of enums that need to be explained here.

I18n Enum

The I18n enum is used in the game instance and settings widgets to determine which languages are supported by the experience
Full Designation: `Enum_I18n`
Content: one entry for each supported language

Currently supported languages

- English
- German
- French
- Spanish
- Italian

HMD Enum

The HMD enum is used to adjust controller meshes and function mappings to the detected HMD in a VR application.
Full Designation: `Enum_HMD`
Content: one entry for each supported HMD

Currently supported HMD

- NoHMD
- SteamVR_Vive
- SteamVR_Index
- OculusHMD
- OculusQuest
- PSVR
- WindowsMixedRealityHMD
- Cosmos

Controller Buttons Enum

The controller buttons enum is used on the motion controllers, motion components and most importantly in the function mapping of the preset data asset (see section 7.1.1) to manage the allocation of the button input to the controller functions.
Full Designation: `Enum_Controller_Buttons`
Content: one entry for each controller button incorporated in the controls of the application

Available Controller Buttons

- None
- Trigger
- Grip
- Face01
- Face02
- Face03
- Face04
- ThumbstickPress
- Shoulder
- ThumbStickUp
- ThumbStickDown
- ThumbStickLeft
- ThumbStickRight
- Menu

Finger Enum

The finger enum is used to define the finger requirements in the gesture DA (see section 7.1.4) used for hand-tracking.
Full Designation: `Enum_HandTracking_Finger`
Content: one entry for each finger of a normal hand.

Fingers

- Thumb
- Index
- Middle
- Ring
- Pinky

3D Directions Enum

The 3D direction enum is used to refer to a direction
Full Designation: `Enum_Directions_3D`
Content:

- Up - usually parallel to z+

- Down - usually parallel to z-
- Forward - usually parallel to x+
- Backward - usually parallel to x-
- Right - usually parallel to y+
- Left - usually parallel to y-

8 User Interfaces

The AF Core provides several user interfaces to choose from for information display as well as a plethora of possibilities for the player to interact with the experience. This includes widget based UI like the HUD, Info Window or Pallets as well as widget-less UI like the Selection Menu. However, with the variety of environments the Framework supports by now it was not possible to provide all of these options for every environment, especially since the VR environment does not support traditional HUD elements. The following table will provide you with a guideline of which UI element can be used in which environments.

Please note that screen here encompasses desktop as well as mobile environments.

	VR Only	Screen Only	All Environments
Widget Based	• pallets • tiny display • mini tags	• HUD	• info window • panel based menu
Widget-less	• radial menu		• selection menu

8.1 Widgets

The AF Core provides several custom made widgets that are ready to use, but also a number of parent classes, that can be used to create individual widgets. The main challenge while creating widgets for the Framework was implementing reliable interactivity especially in VR. The Unreal inherent implementation of widget interactivity (via the widget interaction component) did not prove suitable for all functionalities of the Framework. Thus, we created our own algorithm, the recursive widget selection. When working with the Advanced Framework both implementations can be used in parallel.

```
Widget_Base
├─Widget_Initial
└─Widget_Button
  └─...
```

- Widget Interaction Component - Unreal inherent implementation for widget interactions. It is based on the unreal widget interaction component.
- Recursive Widget Selection - Advanced Framework inherent widget interaction implementation which is used with for all widget based UI elements of the AF Core. For recursive widget selection to work each widget must be supplied with a function that indicate all sub-widgets of the widget that contain buttons.

As To The Widget Component

Die Widget component of the Framework has been tailored to the AF inherent widgets and will only work smoothly with widgets of the class `Widget_Initial` or their child classes. Still it supports the Unreal widget interaction implementation with the right settings. For more information see section 3.5.4.

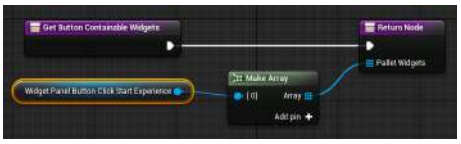

Get Button Containable Widgets
Return Node
Pallet Widgets
Make Array
Widget Panel Button Click Start Experience
[0]
Array
Add pin

Figure 61. Code Example: Get Buttons Function.

Base Widget

serves as a parent class of all interactive widgets of the AF Core and implements a limited amount of replication using the widget component (see section 6.2.4).
Blueprint: `Widget_Base`

Initial Widget

serves as a box for all sub-widgets and as a connection between the widget component (see section 3.5.4) and all other widgets.
Blueprint: `Widget_Initial`

8.1.1 *Widget Buttons*

The widget buttons represent a class specifically designed to be used in the Framework's widgets and widget based UI elements. It is particularly adjusted to allow the motion controllers in VR applications interact with widgets.
Each button implements four basically independent states:

- Press - practically the off-state of the button
- Hovered - when the input device or controller is on the button
- Selected - basically the on-state of the button
- Disabled - suspends the buttons functionality

```
Widget_Button
 ├─Widget_Button_Arrow
 ├─Widget_Button_Standard
 └─...
```

Widget Button - Base

The widget button base class serves as the parent class of all widget buttons. implements all basic functions of widget buttons. However, the widget button does not have a visual appearance and is serves as a parent class only.
Blueprint: `Widget_Button`

8.1.2 *Pallet Widgets*

Pallet widgets are specifically designed for the pallet UI which displays them. Most of them are custom designed for their application, highly specialized and have numerous sub-widgets.

Multiplayer Pallet Widget

provides all information and functions to host or join multiplayer sessions (Figure 62).
Blueprint: `Widget_Initial_Pallet_Multiplayer`
Content:

- Title Box - shows the widget title
- Client Field - contains an additional widget that provides an overview on available servers
  - Title - widget name and a button to refresh the server list
  - Server List - contains either a search sign, a list of available servers or an error message if no servers were found
- Search Bar and Keyboard - allows to search for servers by IP
- Player Field - contains an additional widget giving an overview on the players in a session
  - Title - widget name and a button to refresh the player list
  - Player Field - adds a player field widget for each player in session. This widget contains the player name, ping and a mute button.
- Host Button - allows to host a session

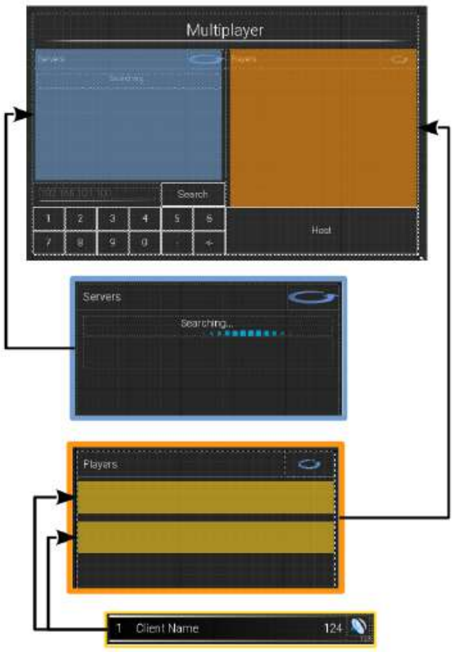


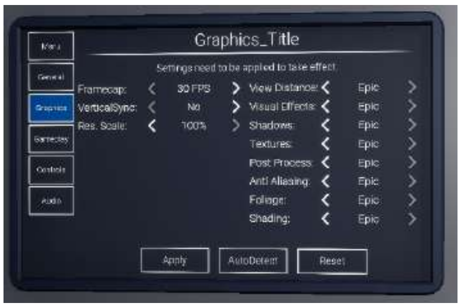
Figure 62. Multiplayer Pallet Widget.

Menu and Settings Pallet Widget

the menu and settings pallet encompasses a number of widgets that are all custom designed for their purpose (Figure 63).

Blueprint: `Widget_Initial_Pallet_Menu`
Content:

- Menu Widget - contains a simple menu that allows the player to reset the level, switch to the menu level and exit the application
- General Settings Widget - contains a list of general settings like language as well as buttons to apply new settings and reset to default.
- Audio Settings Widget - contains a list of audio settings as well as buttons to apply new settings and reset to default
- Graphics Settings Widget - contains a list of graphics settings as well as buttons to apply new settings, auto-detect optimal settings and reset to default.



Graphics_Title

Settings need to be applied to take effect.

```
Framecap:      <  30 FPS  >    View Distance:  <  Epic  >
VerticalSync:  <  No      >    Visual Effects: <  Epic  >
Res. Scale:    <  100%    >    Shadows:        <  Epic  >
                               Textures:       <  Epic  >
                               Post Process:   <  Epic  >
                               Anti Aliasing:  <  Epic  >
                               Foliage:        <  Epic  >
                               Shading:        <  Epic  >
```

Figure 63. Settings Pallet Widget.

Debug Pallet Widget

serves as an additional tool to test your application (Figure 64).
Blueprint: `Widget_Initial_Pallet_Debug`
Content:

- Unreal Classes Field - checks if the AF Game classes are set up correctly
- Pawn Field - states the current pawn and motion controllers in case of a VR pawn
- Trace Buttons - activate optical traces for the gazeview, select and grab interactions.
- General Field - shows the FPS and custom debug functionalities

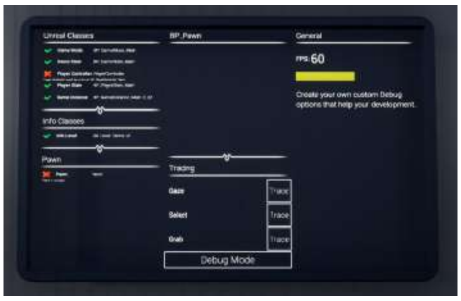


Figure 64. Debug Pallet Widget.

8.1.3 HUD Widgets

In general, HUD widgets implement similar functionalities as pallet widgets. However, since the HUD only provides limited space we created new, adjusted HUD widgets.

HUD Initial Widget

The HUD initial widget consists in a list of buttons which grant access to the other HUD widgets.
Blueprint: `Widget_HUD_Initial`

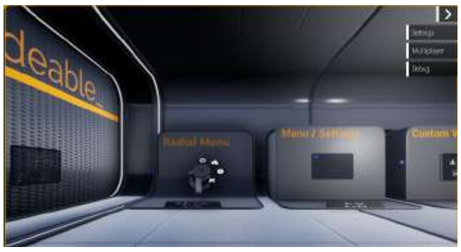


Figure 65. HUD Initial

Multiplayer HUD Widget

provides all information and functions to host or join multiplayer sessions.
Blueprint: `Widget_HUD_Multiplayer`
Intended Frame: Left
Content:

- Host Button - allows to host a session
- Client Field - contains an additional widget that provides an overview on available servers
  - Title - widget name and a button to refresh the server list
  - Server List - contains either a search sign, a list of available servers or an error message, if no servers were found

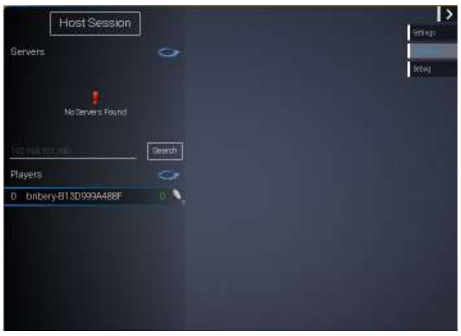


Figure 66. HUD Multiplayer.

- Search Bar - allows to search for servers by IP
- Player Field - contains an additional widget giving an overview on the players in a session
  - Title - widget name and a button to refresh the player list
  - Player Field - adds a player field widget for each player in session. This widget contains the player name, ping and a mute button.

Settings HUD Widgets

the menu and settings pallet encompasses a number of widgets that are all custom designed for their purpose (Figure 67).
Blueprint: `Widget_HUD_SettingsPage`
Intended Frame: `HUD_Widget_Frame_Left`
Content:

- Menu Widget - contains a simple menu that allows the player to reset the level, switch to the menu level and exit the application
- General Settings Widget - contains a list of general settings like language as well as buttons to apply new settings and reset to default.
- Audio Settings Widget - contains a list of audio settings as well as buttons to apply new settings and reset to default
- Graphics Settings Widget - contains a list of graphics settings as well as buttons to apply new settings, auto-detect optimal settings and reset to default.

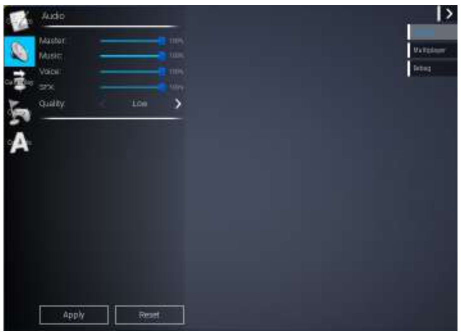


Figure 67. HUD Initial

Debug HUD Widget

serves as an additional tool to test your application. The HUD widget can switch between three sub-widgets of different debug tools (Figure 68).
Blueprint: `Widget_HUD_Debug`
Intended Frame: Left
Content:

- Unreal Classes Field - checks if the AF Game classes are set up correctly
- Pawn Field - states the current pawn and motion controllers in case of a VR pawn and provides three buttons to activate optical traces for the gazeview, select and grab interactions.
- General Field - shows the FPS and custom debug functionalities

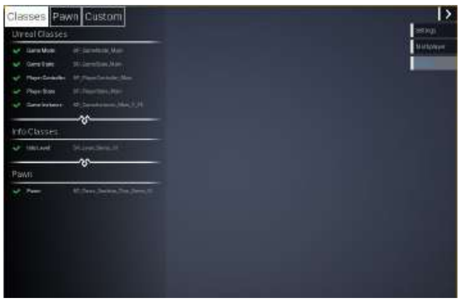


Figure 68. HUD Multiplayer.

8.2 Widget Based UI

The basic method of widget based UI elements comprises the spawning or placement of an actor that displays a tailored widget as well as some basic functions like a delete function. Overall the AF Core implements the essentials for 6 different UI elements.

- Pallet - spawnable, grabbable actor displaying an interactible wigdet
- Panel-based Menu - auto-generated, interactive menu
- Info Window - spawnable actor displaying an interactible widget
- Tiny Display - small display describing the grabbed actor
- Mini Tags - small widget showing limited amount of information

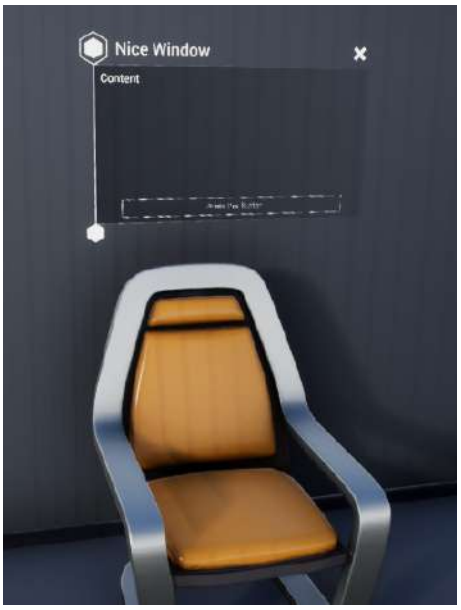


Figure 69. Info Window

8.2.1 Info Window

provides a customizable environment to display additional information on an actor. The info window is spawned using select the component at suitable locations near the actor and displays one or multiple widgets encapsulated by a blueprint that handles the delete and grab logic for example. (Figure 69)

As many UI elements, the info window presents itself in the AF Core in its most basic form. For more elaborate info windows check the AF Extensions.

Blueprints: `BP_InfoWindow`

Settings: see section 3.5.5

Overview on Setup

The info window UI relies on an interaction component to spawn it and a UI component to specify its content.

- Interaction Component - a select type component or gazeview component with setup according to section 3.1.1 or 3.1.2
- UI Component - window component with setup as described in section 3.5.5

8.2.2 Panel Based Menu

Allows the player to enter all or selected levels of an application via an auto-generated menu. In the panel based menu each level is represented by a selectable panel. The content of each panel is summarized by its Panel Level Data Asset (see section 7.1.3). Upon selection the panel expands revealing among others a start button, which grants access to the level. The panel based menu comes in two designs:

- Large Panel Menu - the main panels are arranged in a row that the player can scroll along. Upon select the main panel expands one side panel to its right as shown in Figure 70.
- Small Panel Menu - the main panels are arranged in rows and columns like tiles (no scrolling). Upon select the main panel moves in front of the others expands two side panels as shown in Figure 71.

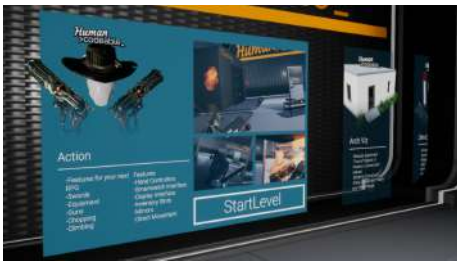


Figure 70. Large Panel Menu

Both designs of the panel based menu are set up in the same way, but they cannot be mixed.

Blueprints: `BP_Demo_LevelMenu_Large`, `BP_Demo_LevelMenu_Small`

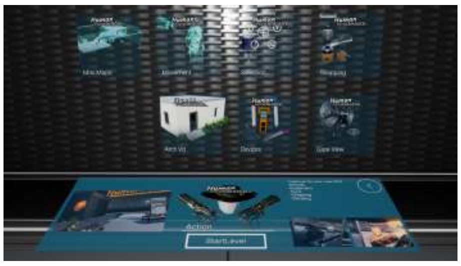


Figure 71. Small Panel Menu

Overview on Setup

The panel based menu relies on the collaboration of the panel menu actor which displays the menu in the application and the panel menu data assets which provide the content for each level.

- Data Assets - create a panel level DA for each level, that should appear in the panel based menu using as template *either*
  - `PDA_Panel_Small_Level`

  or
  - `PDA_Panel_Large_Level`

  and enter settings according to section 7.1.3.
- In the Map - add an instance of the panel menu actor corresponding

to your panel level DA to the menu map.

- On the Panel Menu Actor - access the panel menu component on the instance of the panel menu actor in the menu level and enter all panel menu DA.

8.2.3 Pallets

Pallets are actors that display an interactive widget and can be grabbed, spawned and deleted. Pallets are usually spawned by the radial menu UI (see section 8.3.2) and are dedicated to a specific function like displaying settings or a set of functions. As UI elements pallets mainly substitute for the HUD of non-VR applications. The pallet UI element obtains its functionality from the widget it displays. The AF Core provides a number of pallet widgets that mainly serve as examples for the power of this UI element.

Available Pallet Widgets

- Widget_Initial_Pallet_Debug
- Widget_Initial_Pallet_Multiplayer
- Widget_Pallet_Settings_Audio
- Widget_Pallet_Settings_General
- Widget_Pallet_Settings_Graphics
- Widget_Pallet_Menu

Overview on Setup

Pallets are actors displaying the pallet widget. Most of the widgets contain all necessary functions already. Only the menu widget needs the menu widget component which specifies the info DA of the menu level. For further information look at section 8.3.2 which is about the radial menu.

8.2.4 Tiny Display

The tiny display is a special function of the laser motion controller. A tiny display is spawned when the laser motion controller grabs an actor giving a name and an image as well as the distance of the actor from the motion controller and rotation angle around the z-axis.

Blueprint: `BP_ControllerDisplay`

Settings: see section 3.5.3

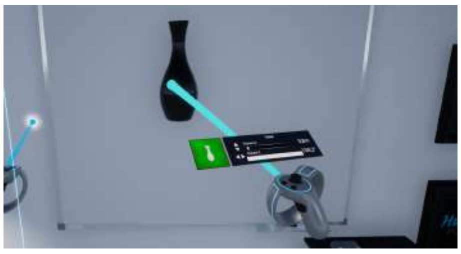


Figure 72. Laser controller with tiny display

Overview On Setup

The tiny display is a pretty simple device, that only works on the laser motion controller. All logic for spawning and deleting the display is inserted in the motion controller. Consequently, the only setup necessary is adding the tiny display component to the participating actors and entering the desired information.

8.2.5 Mini Tags

Mini tags are spawnable actors for minimalist information display. Mini tags are encapsulated in the mini tag component (see section 3.5.6). Each mini tag consist of a widget showing the text element and a line connecting the mini tag to its actor.

Blueprint: `BP_MiniTag`

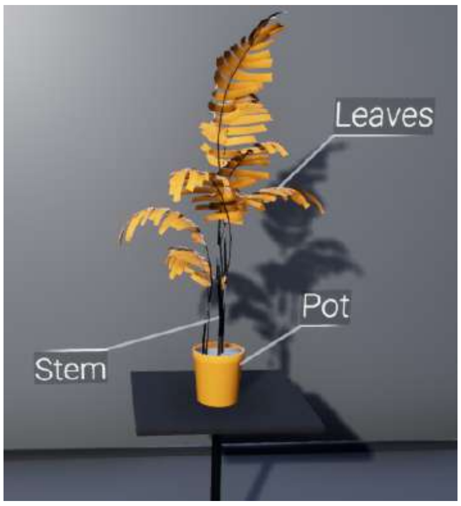


Figure 73. Actor with mini tags

Settings: see section 3.5.6

Overview on Setup

All settings for the mini tags are encapsulated in the mini tag component, which must be added to every actor that is supposed to display a mini tag.

8.2.6 HUD

The HUD of the Advanced Framework consists of three cooperating widget types allowing for interactivity and interchangeable content.

- Menu - the menu is created automatically according to the settings of the HUD component (see section 3.6.4) and grants access to the interchangeable widgets of the HUD
- Frame - frames are specialized widgets that act as slots on the screen and manage interchangeable content widgets. Whenever a new widget is opened its assigned frame checks if its occupied and removes the widget occupying it if necessary.
- HUD Content - the HUD widgets provide the actual content to the HUD. They are accessed by the menu and occupy frames on the screen.

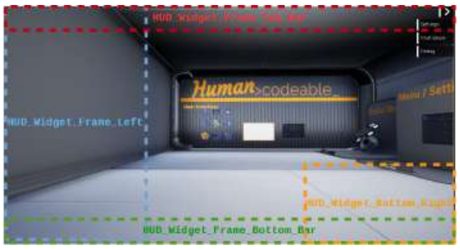


Figure 74. Frames Positioning

Overview on Setup

The HUD is automatically created upon the start of a level. It consists of the menu and a number of other widgets like the cross in the middle of the crosshair navigation mode. The content of the menu widget in turn is determined in the HUD component, which assigns each HUD element with a name, a widget and a frame (see section 3.6.4).

8.3 Widget-less UI

Besides the conventional widget based UI elements the AF Core provides a number of UI elements especially for VR environments, that forgo widgets. These widget-less user interfaces are especially convenient for multiplayer application, since widgets imply a number of issues regarding replication (see section 6.2).

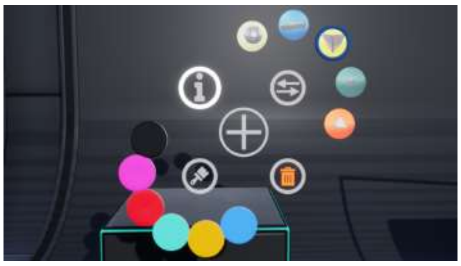


Figure 75. Button Arrangement Circular

8.3.1 Selection Menu

The selection menu is implemented by the `Comp_Select_SelectionMenu` (see section 3.1.1). It consists in a pre-defined set of buttons that are spawned in convenient assembly and can be interacted with using the motion controller or other input methods. Sub-sets for additional choice will be spawned automatically if necessary. Thus, the selection menu provides an intuitive way to consolidate multiple functionalities of one actor which is especially convenient for customization purposes.

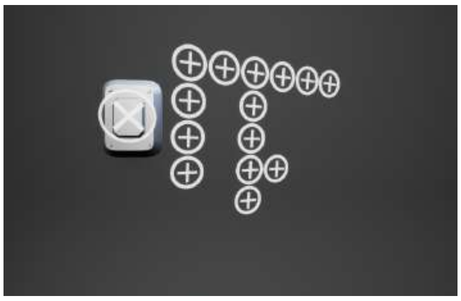


Figure 76. Button Arrangement Rows & Columns

Implemented on: selection menu select component (section 3.1.1)
Possible Functionalities:

- change the visual appearance of an actor
- toggle animation, video or light
- open or close an actor
- toggle an info window about the actor
- delete an actor

Overview on Setup

The selection menu relies on the cooperation of a number of different components, that need to be added to the actor and set up accordingly for the selection menu to function properly. Here is a short summary:

- Interaction Component - selection menu select component with settings according to page 13
- State Components - one state component for each desired button with setup according to descriptions in section 3.2 and an individual tag to be referred by on the select component.

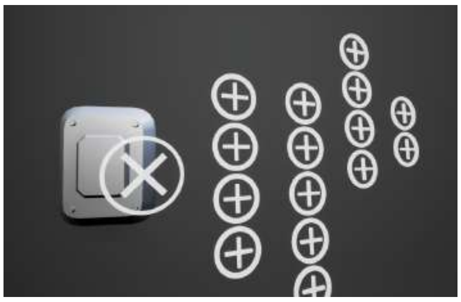


Figure 77. Selection menu.

8.3.2 Radial Menu

The radial menu is encapsulated in a motion component on the motion controllers of the VR pawn (see section 5.3.3). Consequently, the radial menu is a VR-only UI. It consists in a circular array of buttons that are spawned upon opening the radial menu and deleted when the radial menu is closed or a button is selected. Each radial menu button is a child of on of the following classes:

- `BP_Radial_Button_Pallet` - spawning a pallet UI elements (see section 8.2.3)
- `BP_Radial_Button_Quick` - serve as templates for customizing the radial menu

All buttons implement the select functionality and can be selected by the other motion controller as well as using the thumbstick and trigger button. The radial menu automatically closes when a button has been selected.

```
Radial Button
└─Radial Button - Pallet
│  ├─Radial Button -
│  │  Functions Pallet
│  ├─Radial Button -
│  │  Multiplayer Pallet
│  └─Radial Button -
│     Settings Pallet
└─Radial Button - Quick
   └─Radial Button - Quick
      Spawn
```

Overview on Setup

The radial menu is the main connection between the VR motion controllers and the main VR UI element, the pallets, as well as an UI in itself. Consequently, it needs a bit of care arranging the necessary components on both the pawn and the motion controller.

- On the VR Pawn - add the radial menu component and enter settings as described in section 3.6.2
- On the Motion Controller - set up the radial menu motion component according to section 5.3.3
- In the Preset DA - add a key to open the radial menu to the key mapping as described in section 7.1.1 and 3.6.1

Currently Implemented Radial Buttons

- `BP_Radial_Button_Pallet_Custom`
- `BP_Radial_Button_Pallet_Multiplayer`
- `BP_Radial_Button_Pallet_Settings`
- `BP_Radial_Button_Pallet_Debug`
- `BP_Radial_Button_Quick`
- `BP_Radial_Button_Quick_Spawn`

💡 The navigation of the radial menu using thumbstick and trigger button is already built-in.

9 Glossary

A

absolute position: location of an item in relation to the origin of the map.
anchor: component you add to the accepting actor of a snap to anchor process to determine location and direction of the attached actor (section 3.3.3).
asymmetric game play - means here specifically a VR application that combines a *VR pawn* controlled by the player with a HMD and and an *asymmetric pawn* controlled by another player using desktop and keyboard.
asymmetric pawn - additional pawn in a VR application that is spawned by the *VR pawn* to facilitate *asymmetric game play* (section 5.3.5).

B

Base Pawn: parent class of all customized pawns in the framework (section 5)
body slot: anchor that has been placed on a character pawn.
button mapping: assignment of buttons of the physical controllers to functionalities of the *motion controller* via a *preset*.

C

character pawn: pawn class that provides the player with a virtual body
component: exchangeable functional element implementing a functionality (section 3)

D

datatables: datatables are Unreal inherent tables that function like a database in its simplest form. Each entry is identified by a key which allows functions to return the entry of the corresponding cell. The framework uses datatables mainly for the internationalization (I18n, section 4)
desktop pawn: child class of the *screen pawn* that is adjusted to navigate an application using keyboard an mouse (see section 5.4)

E

event dispatcher: notification function to enable actors to react to changes in other actors.

F

G

game instance: manages the information transfer (most importantly of the level key) when the player changes the level
game mode: spawns the *player controller* and the *game state* upon starting

an application.
game state: manages the transition maps when a level is loaded.
gazeview: interaction method that uses the forward trace of the player camera to highlight and select actors.
grab: moves an actor within the application by attaching it to the *motion controller* (VR) or pawn (other environments)

H

helper: mostly automatically created actors to support other classes in executing their functionalities.

I

impact location: location where the actor is grabbed or latched onto in screen environments
interfaces: standards for information transfer. Do not confuse with *structs*.
intro: first thing that happens when the application is started. Usually it shows a skysphere and a logo.

J

K

L

latch: moves an actor or mesh component within an application by attaching the motion controller (VR) or pawn to the actor.
level: an individually designed map the player visits during the application or a combination of them. Practically each construct you assign with a level data asset.
level data asset: data asset collecting all essential information on a level including world map, transition screen content and pawns.

M

menu level: map of the first level loaded in a play-through of your application, most probably containing a menu.
mobile pawn: child class of the *screen pawn* that is adapted to navigate an application using a touchscreen (section 5.5)
motion component: component implementing a controller functionality in VR (section 5.3.3)
motion controller: container for *motion components* in VR applications (section 5.3.2)

N

O

P

pallet: freely placable UI element that displays information or spawns actors (section 8.2.3)
panel data asset: consolidates the content of a panel in the panel based menu
panel based menu: auto-generated menu **pause map**: default map that is loaded, when the player presses pause.
player controller: spawns and controls the *pawns*, manages the transition to the *pause map* and back
preset: data asset that defines the *button mapping* for a motion controller (section 7.1.1)

Q

R

relative position: position of an item relative to a selected point that does *not* coincide with the origin of the map.
replicating actor: actor that replicates its position or state to keep track of other players actions in multiplayer

S

screen pawn: parent class of all pawns adjusted for applications that are not in VR (section 5.1)
structs: container holding selected variables that are usually transferred together.

T

transition map: default map that is loaded and displayed before and while levels are loaded.
timeout (highlighting): time until the highlighting is removed from an item.
trigger: actor that can be manipulated to activate a functionality on a *receiving actor*

U

UI: UI or user interfaces allow the player access to information and selected functionalities

V

visuals component: allows the change of the appearance of an instance of an actor.

W

widget: system to create 2D elements to be displayed in the application

worldmap: practically an empty container for a level

X

Y

Z