VaMEx-VTB

Modulares virtuelles Testbed für die VaMEx-Fördervorhaben

T-T-CGVR-2000-0001
Simulation- and Visualization-System

	Name and Institution	Date
Prepared by	Jörn Teuber (JT)	28.01.2019
Contributors	Jörn Teuber (JT)	
	Marc Jochens	20.10.2020
	Alexander Klier	20.10.2020
	Carlotta Herrmann	20.10.2020
	Christopher Wolff	20.10.2020
	Haya Al Maree	20.10.2020
	Julian Mehwald-Hoffman	20.10.2020
	Kai Gätjen	20.10.2020
	Lukas Gossé	20.10.2020
Checked by		
Approved by		

FKZ: 50NA1712 Fördermittelgeber

Document Change Log

Issue	Revision	Date	Changes	Name
0	0	28.01.2019	Initial Version	JT
	1	20.10.2020	VaMEx2020 Bachelorproject	

Table of Contents

List of Figures

No table of figures entries found.

List of Tables

No table of figures entries found.

1 Introduction

1.1 Scope

This document describes briefly, at a high level, the requirements, concepts, engineering and algorithms of the simulation and visualization systems used in the VaMEx-VTB.

1.2 Applicable Documents

N/A

1.3 Reference Documents

N/A

1.4 Acronyms

VTB	Virtual Test Bed
ROS	Robot Operation System

2 Requirements Analysis

At the beginning of the project all the partners were contacted to assess their requirements for the VTB. That included the requirements for the environment, the visualizations, the synthesised sensors, and the interfaces from ROS to the VTB and vice versa.
These assessments were regularly updated during the splinter meetings at the DLR synergy-meetings and incorporated into the VTB whenever possible.
VaMEx 2020:
Initially requested features:

- Sensors in VR
- Environmental Process (Methane-Sensor)
- Visualisation of all sensors
- Simulate slip on slopes
- GUI for sensor visualisation settings
- Obstacles for SUs
- Dust Devils
- SU Interaction
- Companion
- Unreal Engine upgrade to 4.25
- New Terrains

3 System Design

The system architecture was designed according to the component-based software architecture, which is also favoured by the Unreal-Engine. This makes it possible to create the swarm units as a plug-and-play system, which each sensor or even behaviour as a component that can easily be attached to it.

3.1 High-Level Architecture

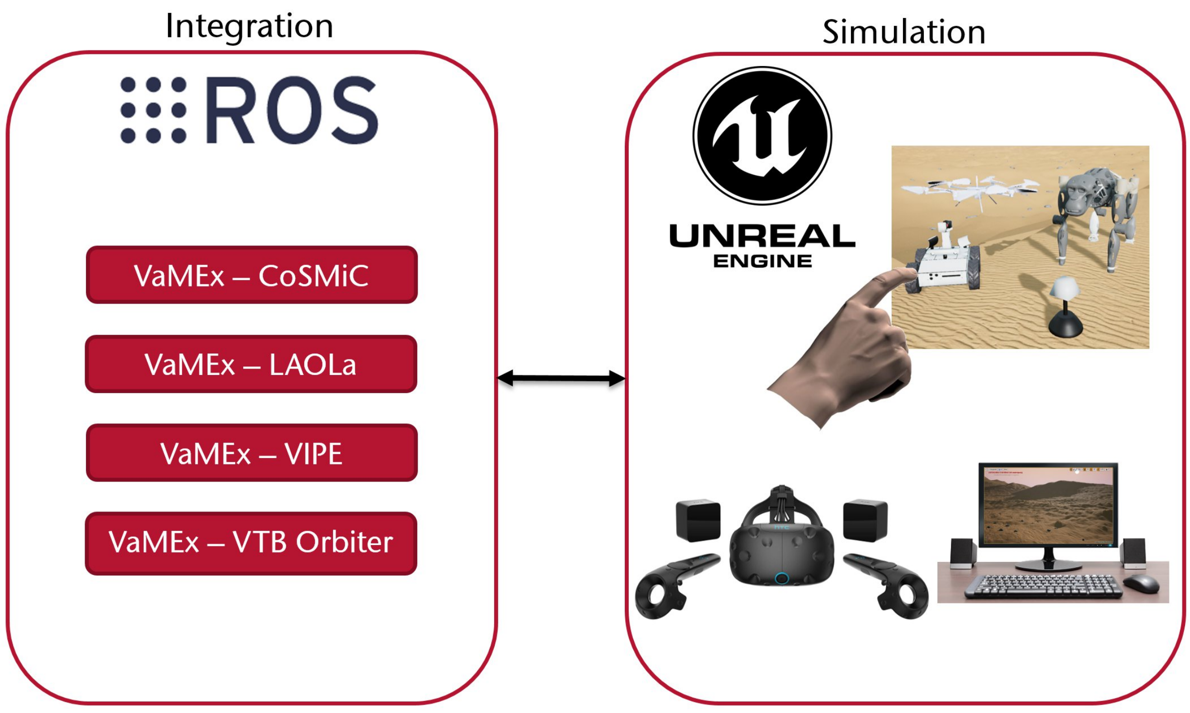


Figure 1: High-level overview of the architecture of the VTB.
As you can see in Figure 1, the VTB consists of two separate parts. The left part is an installation of ROS, containing all algorithms and software-components of the partners packaged in self-contained ROS-nodes. The

right part is an Unreal Engine-project that contains the visualization, the interaction, and the simulation of the swarm units including the virtual sensors. This part will be called the simulation from now on.
The two parts are connected by ROSbridge, which provides an interface for ROS that is accessible via a network connection. This means that the two parts of the VTB can be housed on two entirely different computer systems, for example the ROS system can be set up on a central computer accessible to all partners in the VaMEx-initiative and every partner can run the Unreal-part on their computers to visualize and interact with this central ROS system.
For development and testing purposes a setup using a virtual machine running ROS on a Windows system which is running the simulation is the easiest option though.

3.2 Components in ROS

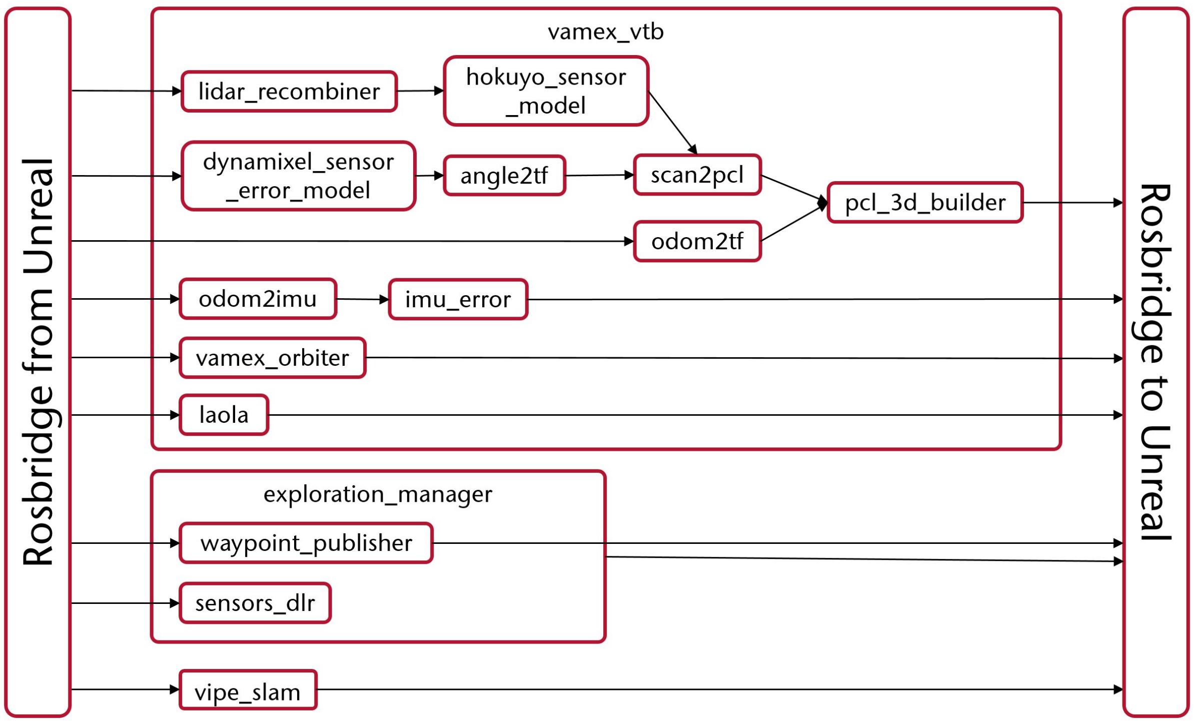


Figure 2: Components and data-flow in the ROS-system.
The above figure shows all ROS-nodes and how they interact with each other and the simulation via the ROSbridge. All rectangles in the middle represent ROS-nodes, which can also contain other ROS-nodes, like the `vamex_vtb`-node.
The `vamex_vtb`-node contains all nodes that can and should be started before the Unreal part. That includes nodes to convert the depth-measurements sent by the virtual lidar into point clouds (all nodes up to and including pcl_3d_builder), a node that converts ground-truth odometry sent by the simulation into imu-data and adds an error to it, a node containing the SPICE-kernels simulating the orbits of the orbiters, and a node containing the algorithms developed in VaMEx-LAOLa.
The `exploration_manager`-node contains the exploration strategy developed by VaMEx-CoSMiC, which has to be started after the simulation. It supplies an environmental process for the swarm-units to explore and the simulation to visualize, and goal-points for the swarm-units to make measurements at.
The bottom-most node, `vipe_slam`, contains the ORB-SLAM2 algorithm it was used in VaMEx-VIPE with a few modifications. This is not included in the `vamex_vtb`-node even though it needs to be started before the simulation as it needs to be run as its own application (it is not a ROS-node in the sense that it can't be started by ROS, but it communicates with ROS) and takes some time to start up.

3.3 Architecture of the Simulation

The simulation can be split into 3 parts: the swarm-units, the support-units and the environment. In the following subchapters, the architecture of the swarm-units and the support-units is shown. The environment, i.e. the terrain, rocks and other features of the landscape, and the sky, is handled completely by the Unreal Engine and is therefore not part of the architecture.

3.3.1 Architecture of the Swarm-Units

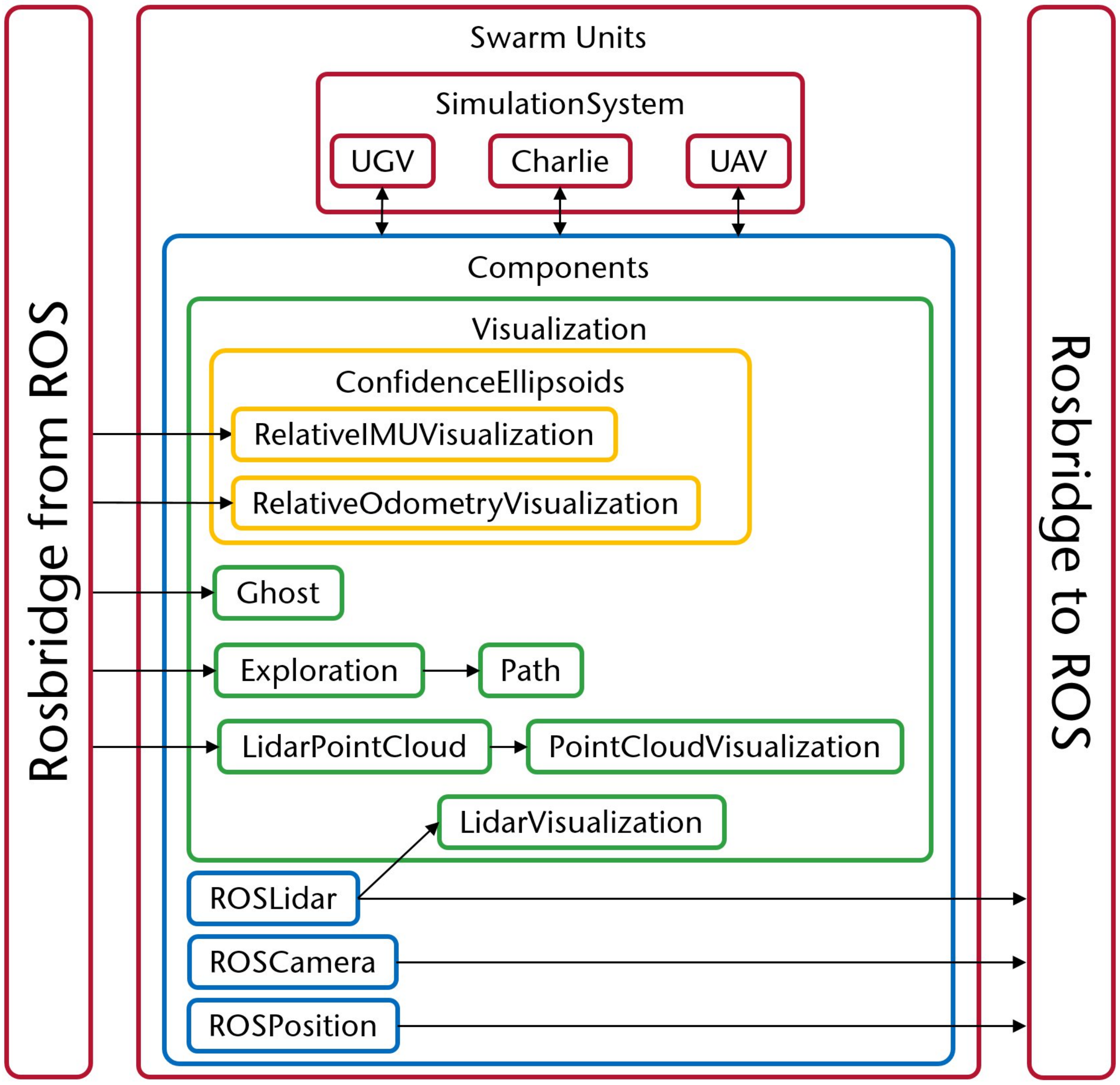


Figure 3: Actors, Components and data-flow of the swarm-units in the simulation. Data flows from left to right along the arrows. The swarm-units (UGV, Charlie and UAV) use components to add functionality.

The swarm-units and their components are designed with reusability in mind. Most of a swarm-unit's abilities are encapsulated in components which can be programmed once and then used on any swarm-unit. This results in a plug-and-play-like architecture where new functionality, like virtual sensors or visualizations, can be added, removed or just moved on a robot whenever needed.

The `SimulationSystem` is implemented as a `GameInstance`, which is a central component of the Unreal Engine. When the simulation is started, the Unreal Engine creates an instance of the `SimulationSystem`, which in turn spawns all swarm-units and arranges them around the origin of the scene. Additionally, simulation wide actors for swarm-/support-unit independent global interactions, such as the

ProcessVisualizer, are spawned. The ProcessVisualizer-Actor uses two ProcessComponents to visualize the ground-truth and estimated environmental process as sent by the ExplorationManager in ROS.
The visualization-components (in the green rectangle in Figure 3) have a common interface to make it easy to hide and show them and make them invisible to all ROSCamera-components, which simulates a Camera and shouldn't picture, for example, the point-clouds generated by the ROSLidar-component and shown by the PointCloudVisualization-component. The new MethanSensorVizComponent displays the value at the parents location within the scene of the currently tracked ProcessComponent by the SimulationSystems ProcessVisualizer actor.
The ROSLidar-, ROSCamera- and, ROSPosition and ROSMethanSensor-components synthesize ground-truth sensor outputs, specifically of a lidar, a RGB(D)-camera, and an odometry-sensor and environmental process methane measurement data respectively. The lidar can also visualize the plane it is currently scanning, which is encapsulated in the LidarVisualization-component. The ROSMethaneSensorComponent is not attached to any ROS Topics yet, due to lack of these in the vtb-ros-source project, which currently uses a static demo implementation. For now, a measurement is only signaled to the currently tracked ProcessComponent for side effects to be observed at visualization, which is required during demo-mode only.

3.3.2 Architecture of Support-Units

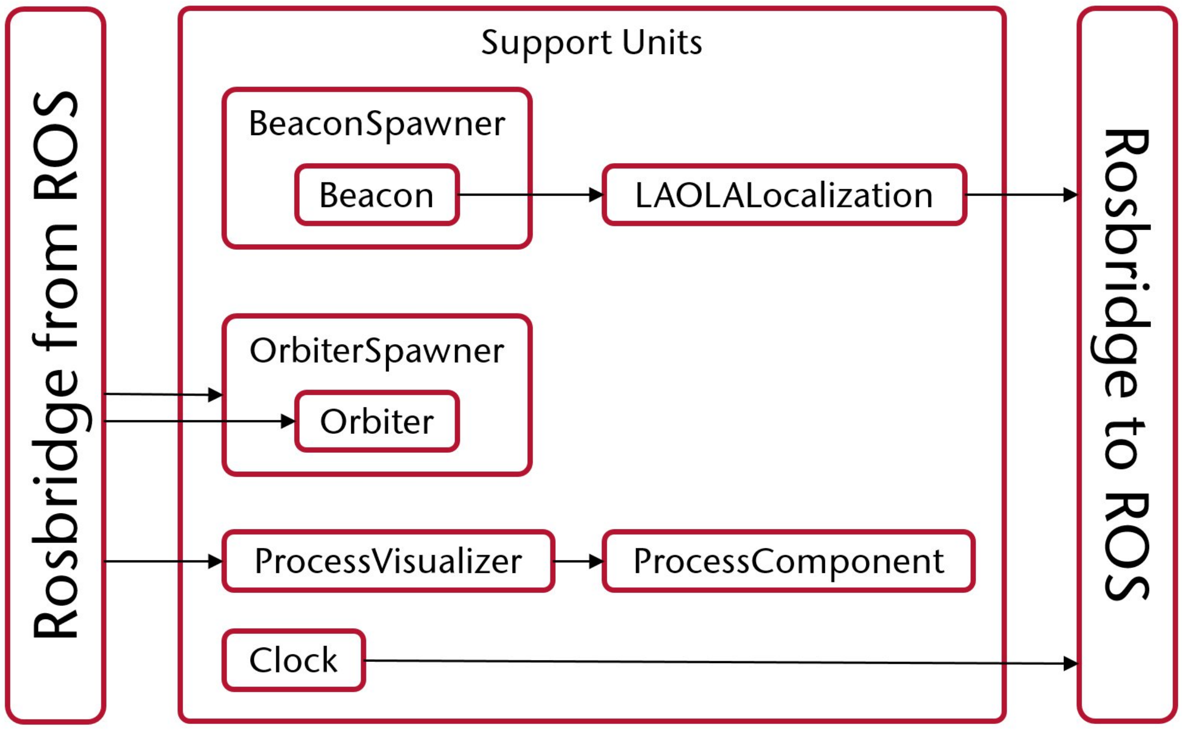
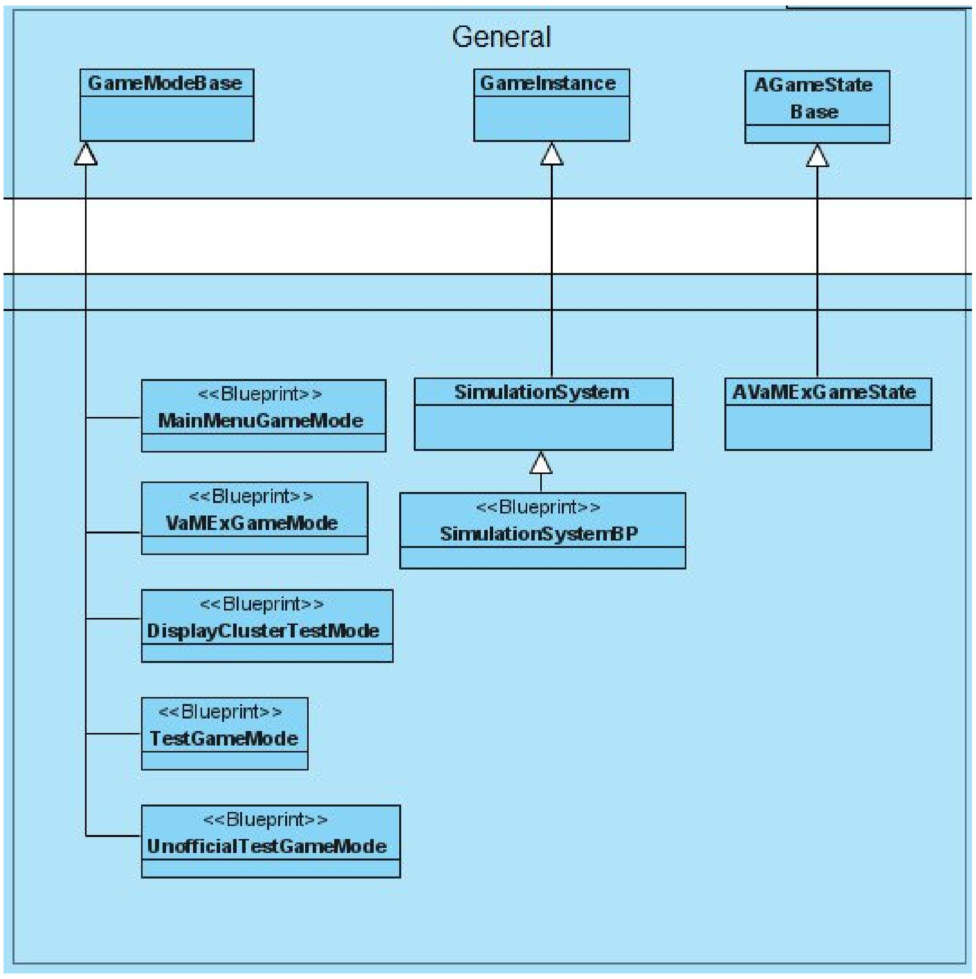


Figure 4: Actors and components of the support-units.
The support-units fulfil a variety of tasks within the VTB and the VaMEx-swarm in general. Instead of one class spawning all these like the swarm-units, each type of support-unit has its own spawner-class. This provides more flexibility and encapsulates the individual subsystems better.
The beacons, which are spawned according to the settings in the BeaconSpawner, are used by LAOLa as static landmarks for their localization algorithms. They are used by the LAOLALocalization-Actor to generate the messages that are send to the LAOLa-algorithms in ROS.
The OrbiterSpawner-actor spawns a set number of Orbiter-actors with settings that can be specified in the spawner, for example a scaling factor. The orbiters then get their position from their respective ROS-topic.
The ProcessVisualizer-Actor uses two ProcessComponents two visualize the ground-truth and estimated environmental process as sent by the ExplorationManager in ROS.
Lastly, the Clock-Actor is a very simple class that sends out the simulations "official" timestamp every frame. This timestamp needs to be used by ROS-nodes that, for their measurements, rely on the current time, for example the vamex_orbiter-node. This makes it possible to accelerate the time (time-warping) in the simulation and see the orbiters following their orbits much quicker.

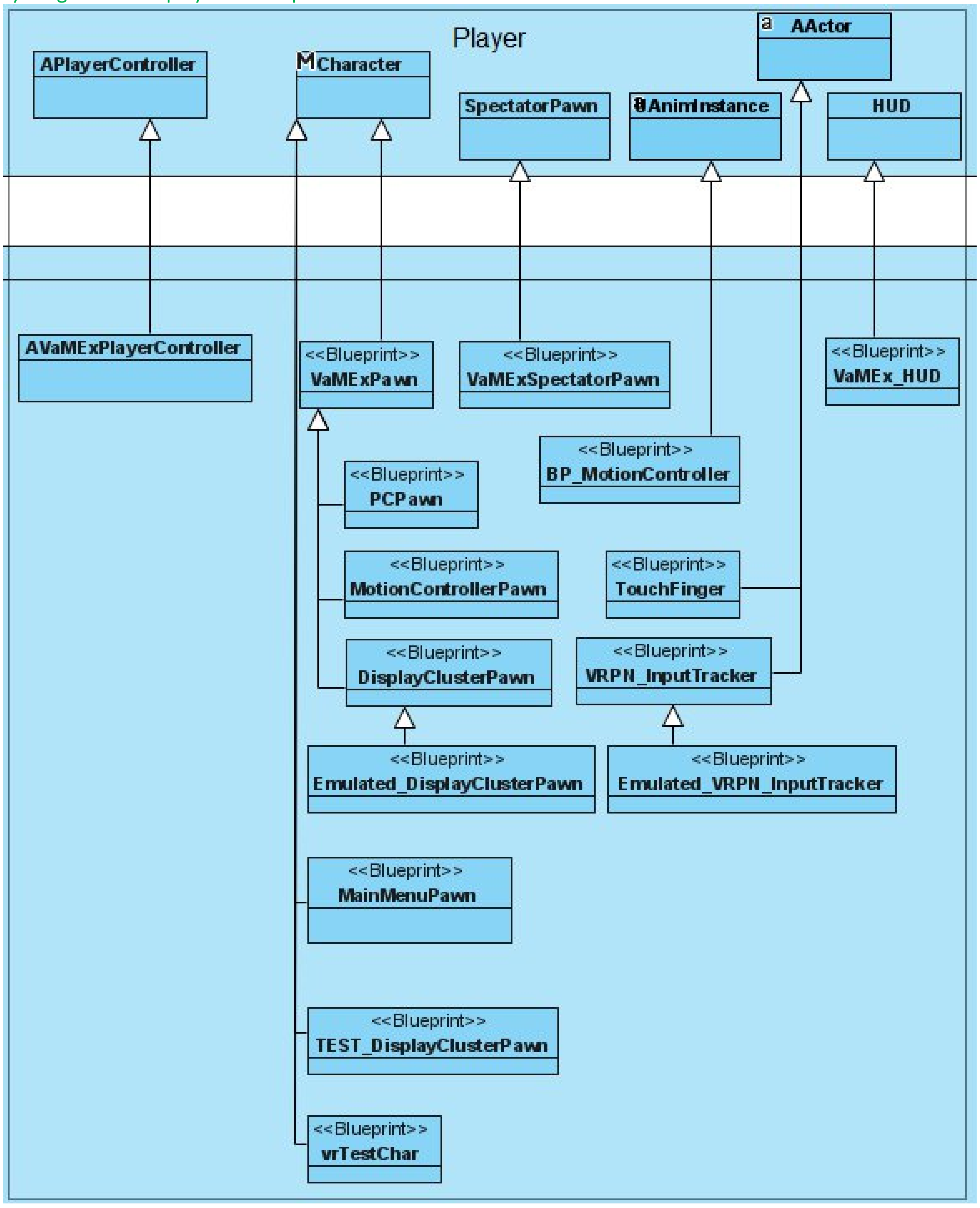
3.4 VaMEx Unreal Overview

The following UML-Class Diagram will show an overview of the whole VaMEx_Unreal Module. To see the full map, please download either the exported picture or the .vpp file. The vpp-file is made with Visual Paradigm Community Version 16.2.
In the box at the top are the Unreal-Engine Classes and below are the VaMEx_Unreal Classes. This UML Diagram does not focus on the UnrealEngine contents. For a detailed UnrealEngine Documentation, visit the official one.

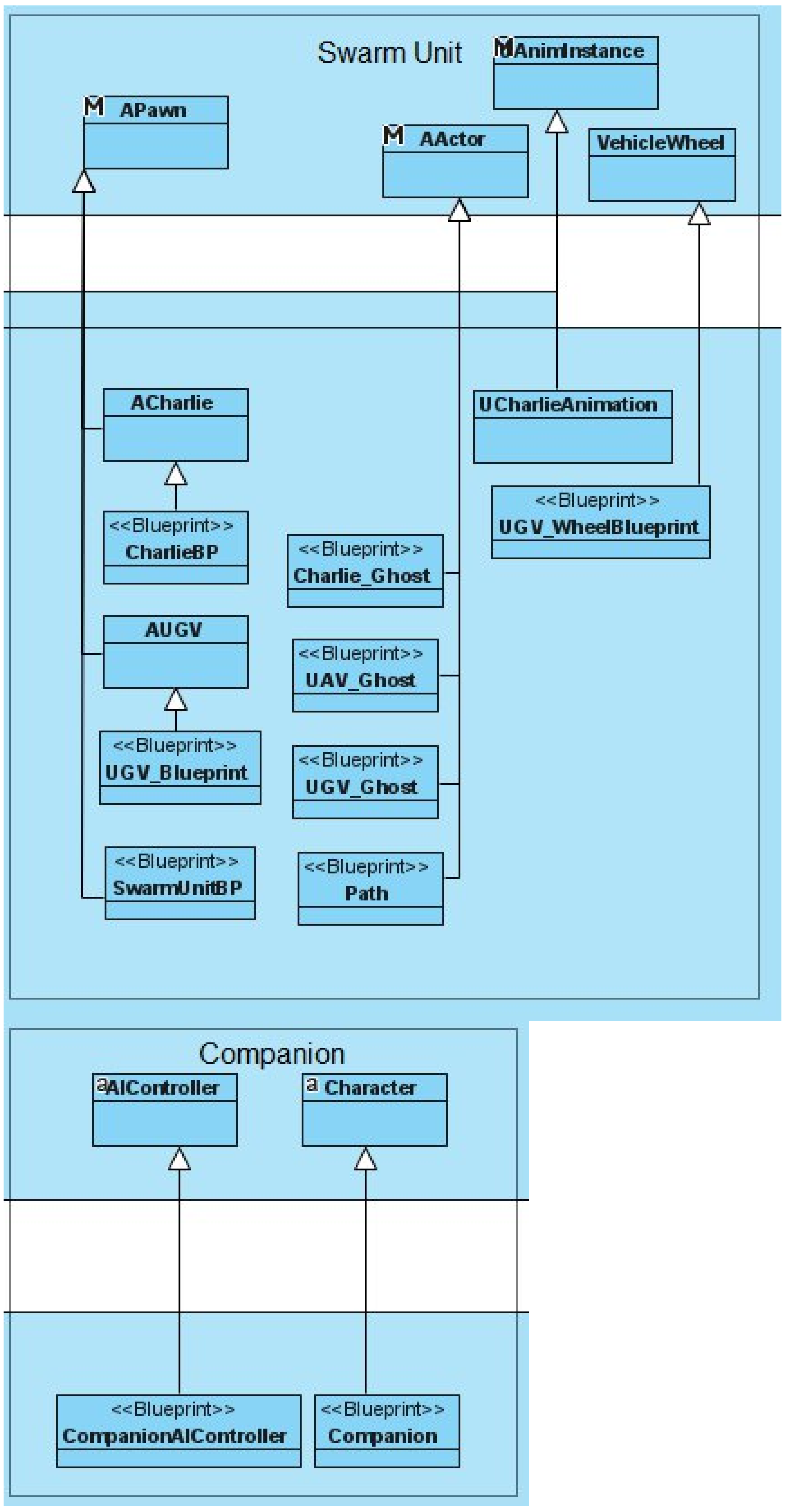

General
GameModeBase
GameInstance
AGameState Base
<<Blueprint>> MainMenuGameMode
SimulationSystem
AVaMExGameState
<<Blueprint>> SimulationSystemBP
<<Blueprint>> VaMExGameMode
<<Blueprint>> DisplayClusterTestMode
<<Blueprint>> TestGameMode
<<Blueprint>> UnofficialTestGameMode

3.4.1 Player

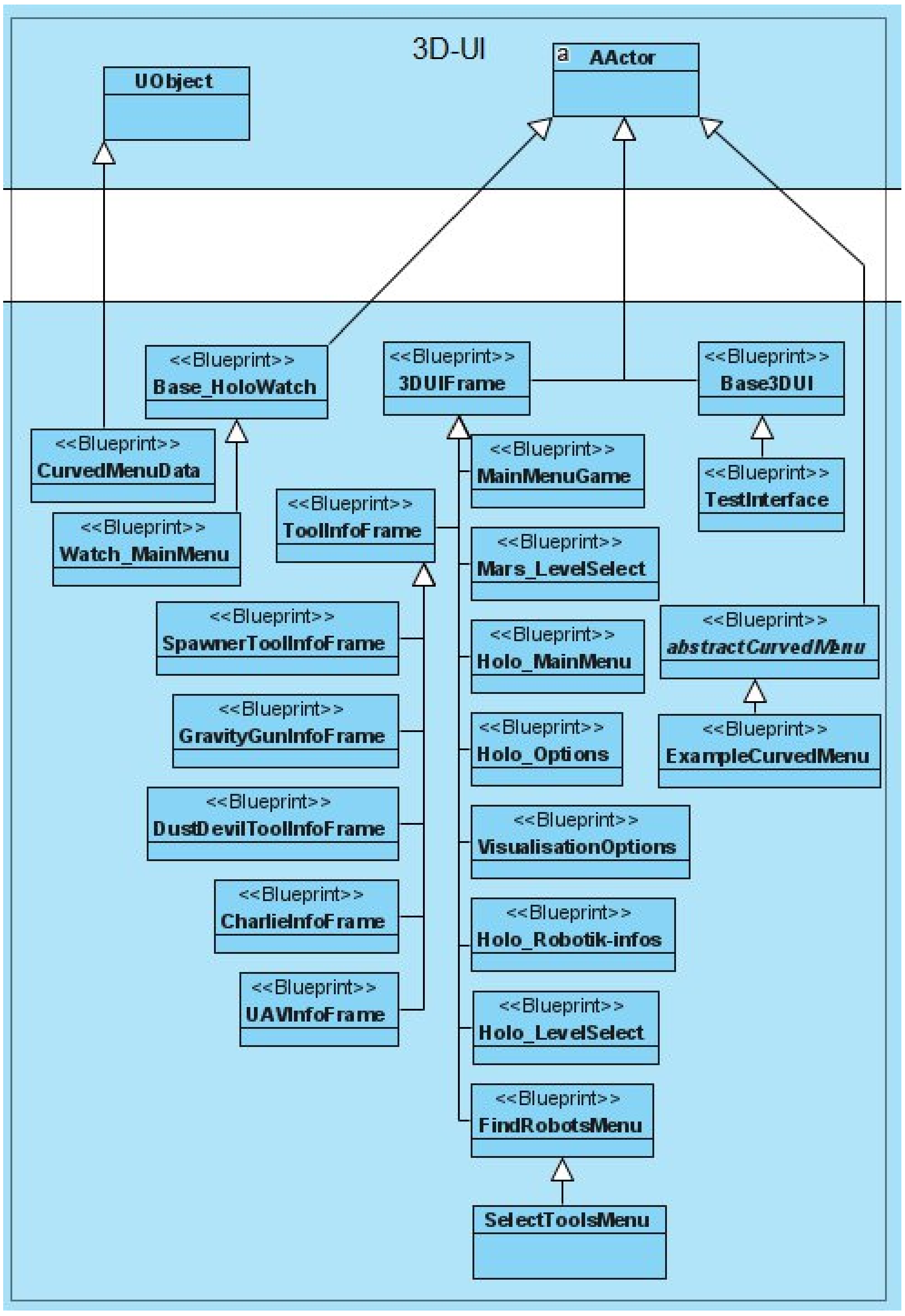
Anything related to players and inputs


Player
APlayerController
Character
SpectatorPawn
AnimInstance
AActor
HUD
AVaMExPlayerController
<<Blueprint>> VaMExPawn
<<Blueprint>> VaMExSpectatorPawn
<<Blueprint>> VaMEx_HUD
<<Blueprint>> BP_MotionController
<<Blueprint>> PCPawn
<<Blueprint>> MotionControllerPawn
<<Blueprint>> TouchFinger
<<Blueprint>> DisplayClusterPawn
<<Blueprint>> VRPN_InputTracker
<<Blueprint>> Emulated_DisplayClusterPawn
<<Blueprint>> Emulated_VRPN_InputTracker
<<Blueprint>> MainMenuPawn
<<Blueprint>> TEST_DisplayClusterPawn
<<Blueprint>> vrTestChar

3.4.2 Swarm Unit & Companion

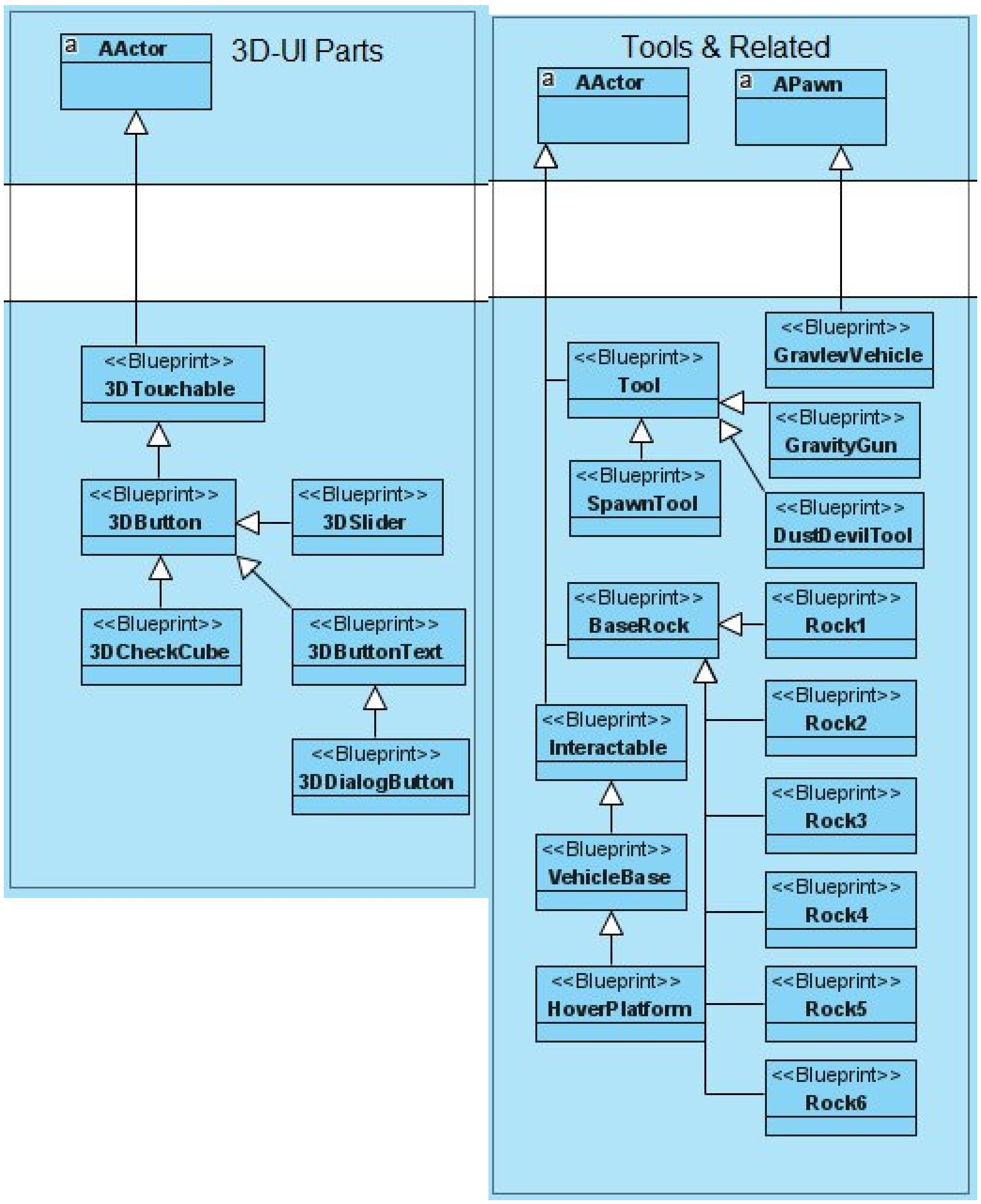


3.4.3 3D-UI

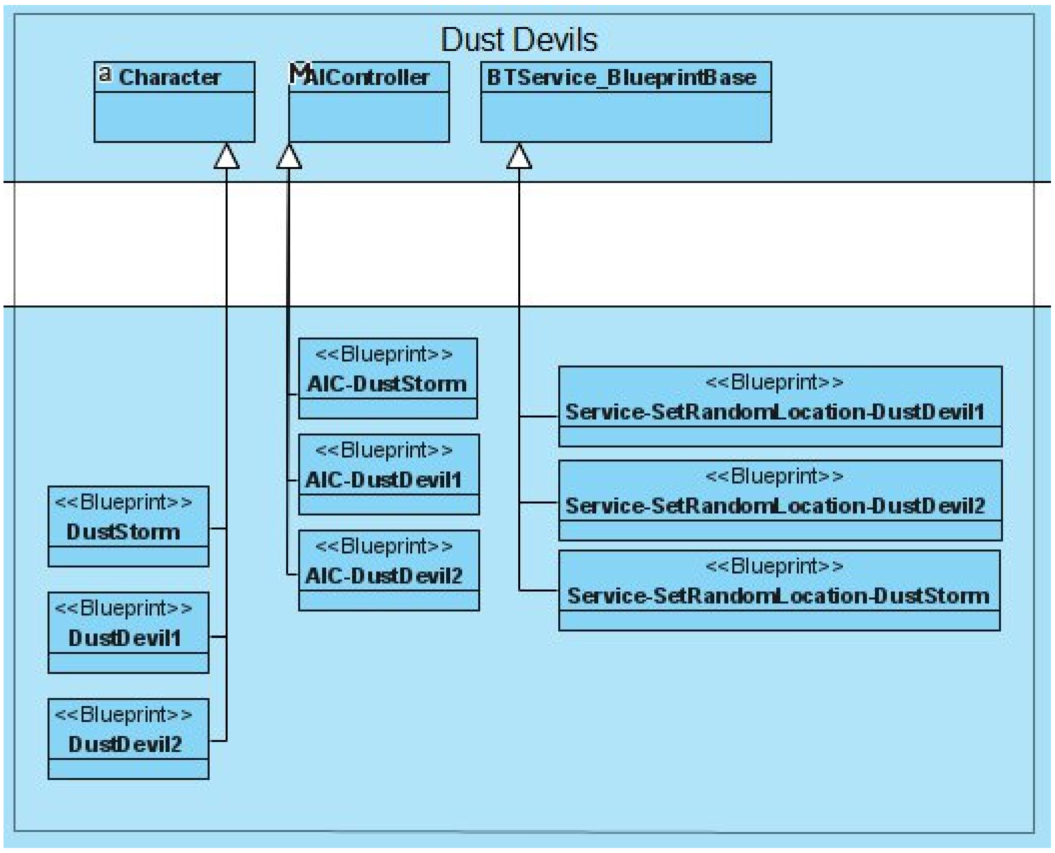

3D-UI
a AActor
UObject
<<Blueprint>> Base_HoloWatch
<<Blueprint>> 3DUIFrame
<<Blueprint>> Base3DUI
<<Blueprint>> CurvedMenuData
<<Blueprint>> MainMenuGame
<<Blueprint>> TestInterface
<<Blueprint>> Watch_MainMenu
<<Blueprint>> ToolInfoFrame
<<Blueprint>> Mars_LevelSelect
<<Blueprint>> SpawnerToolInfoFrame
<<Blueprint>> Holo_MainMenu
<<Blueprint>> abstractCurvedMenu
<<Blueprint>> GravityGunInfoFrame
<<Blueprint>> Holo_Options
<<Blueprint>> ExampleCurvedMenu
<<Blueprint>> DustDevilToolInfoFrame
<<Blueprint>> VisualisationOptions
<<Blueprint>> CharlieInfoFrame
<<Blueprint>> Holo_Robotik-infos
<<Blueprint>> UAVInfoFrame
<<Blueprint>> Holo_LevelSelect
<<Blueprint>> FindRobotsMenu
SelectToolsMenu

3.4.4 3D-UI Parts, Tools & Related to Tools

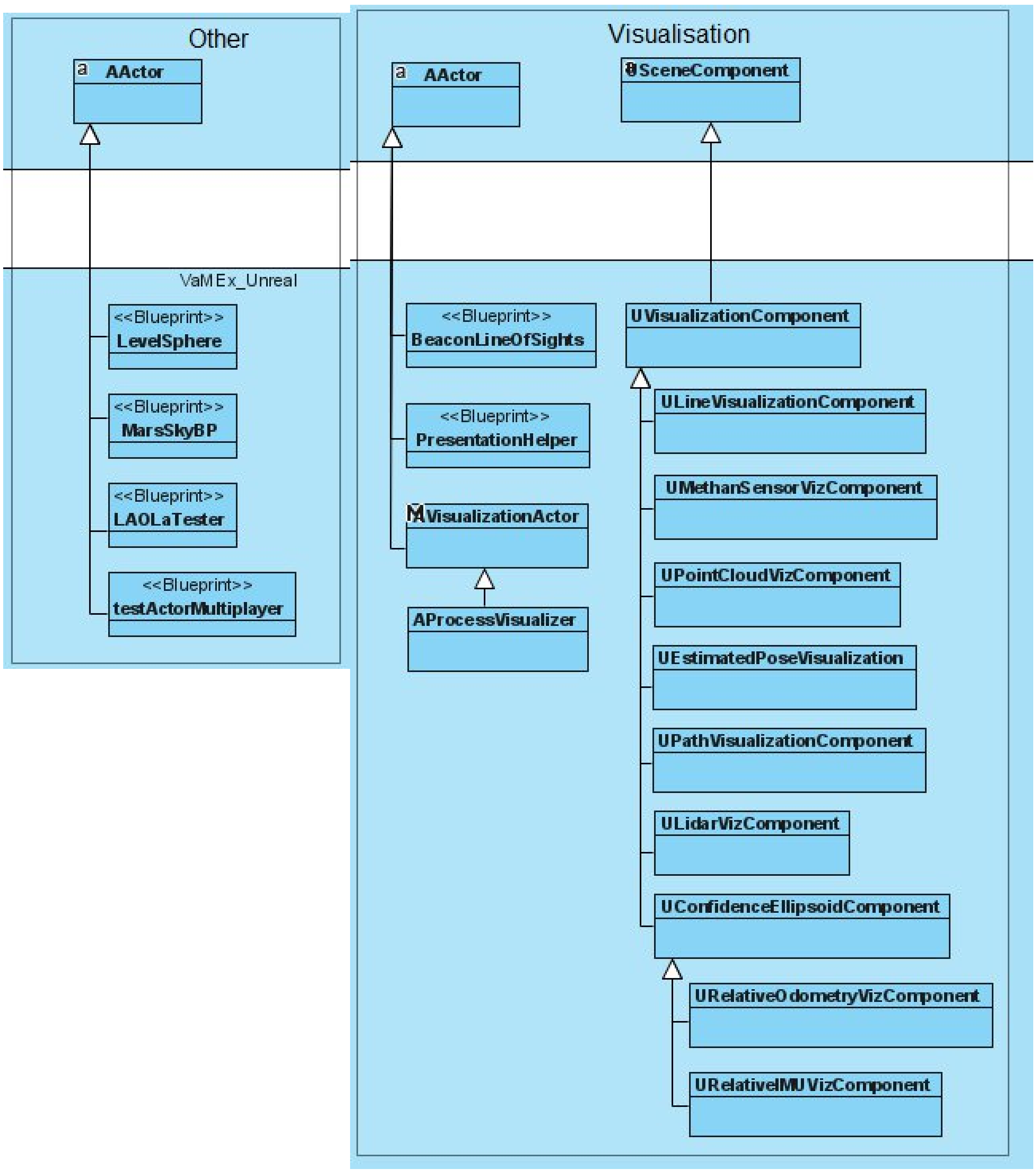
The UI Parts are used as building blocks in assembling the 3D-UI pages (frames).



3.4.5 Dust Devils

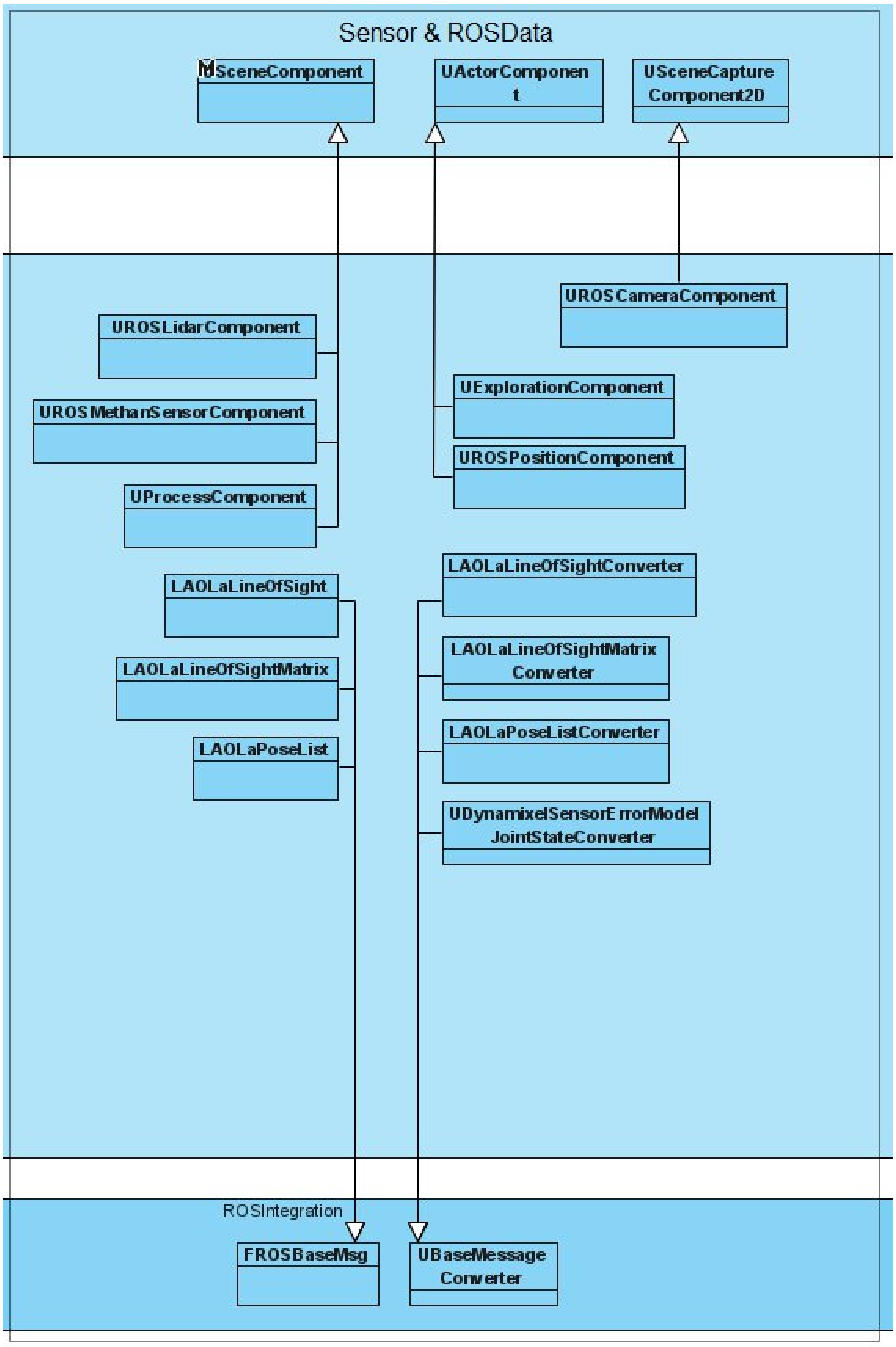

Dust Devils
Character
AIController
BTService_BlueprintBase
<<Blueprint>> AIC-DustStorm
<<Blueprint>> AIC-DustDevil1
<<Blueprint>> AIC-DustDevil2
<<Blueprint>> Service-SetRandomLocation-DustDevil1
<<Blueprint>> Service-SetRandomLocation-DustDevil2
<<Blueprint>> Service-SetRandomLocation-DustStorm
<<Blueprint>> DustStorm
<<Blueprint>> DustDevil1
<<Blueprint>> DustDevil2

3.4.6 Other & Visualisation

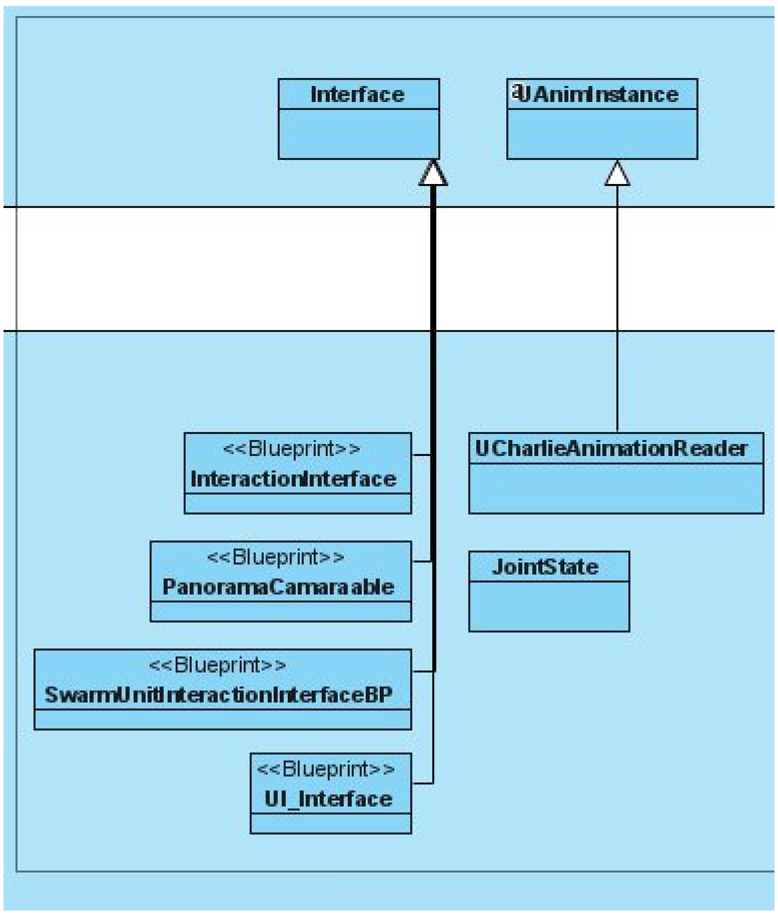

Other
AActor
VaMEx_Unreal
<<Blueprint>> LevelSphere
<<Blueprint>> MarsSkyBP
<<Blueprint>> LAOLaTester
<<Blueprint>> testActorMultiplayer
Visualisation
AActor
USceneComponent
<<Blueprint>> BeaconLineOfSights
<<Blueprint>> PresentationHelper
AVisualizationActor
AProcessVisualizer
UVisualizationComponent
ULineVisualizationComponent
UMethanSensorVizComponent
UPointCloudVizComponent
UEstimatedPoseVisualization
UPathVisualizationComponent
ULidarVizComponent
UConfidenceEllipsoidComponent
URelativeOdometryVizComponent
URelativeIMUVizComponent

3.4.7 Sensor & ROSData

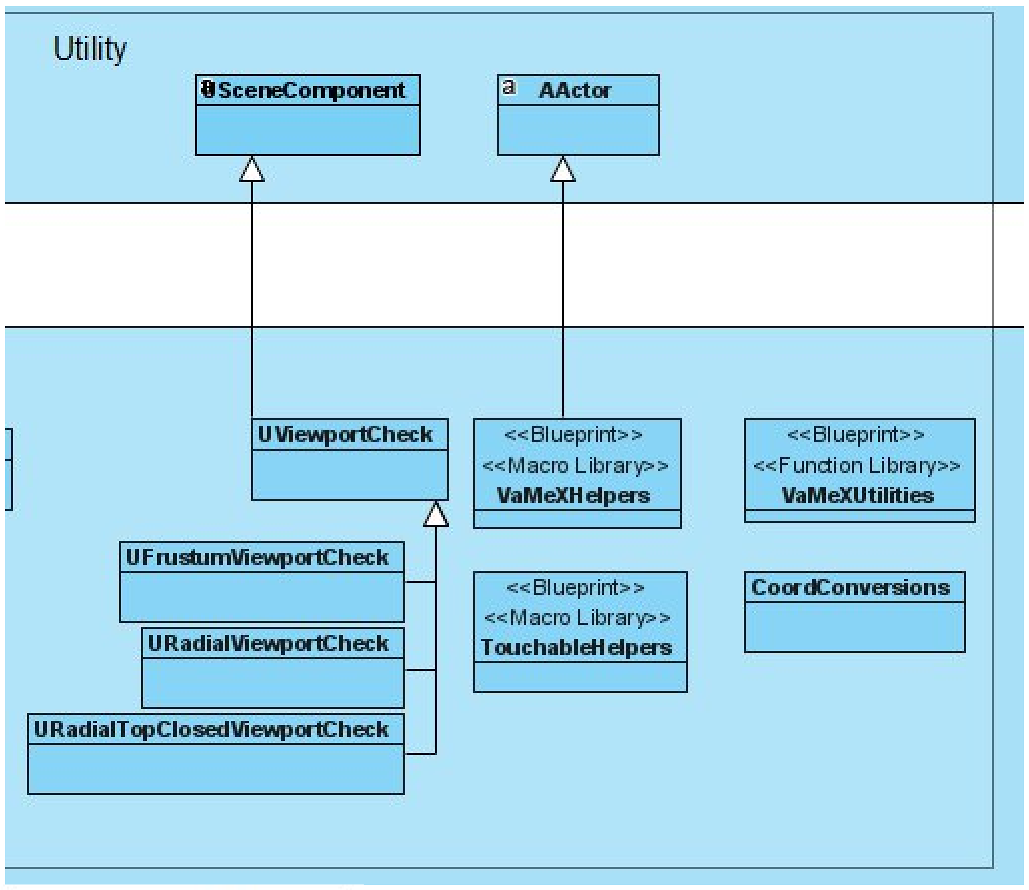
Anything that is in the chain from collecting sensor data until sending it over.

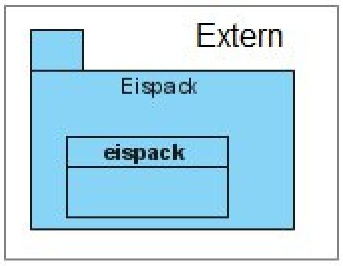


3.4.8 Utility (1/2)



3.4.9 Utility (2/2)





4 Implementation

4.1 C++ Dokumentation

4.1.1 ROSMethanSensorComponent

If attached to a parent component, this component will perform and publish measurements of the specified `measuredProcess` to ROS (currently not implemented due to lack of associated topics on the ros implementation side) as well as the visualizing `ProcessComponent` for demo mode filtering purposes.

4.1.1.1 *Fields*

Signature	Description
`float UpdateIntervall`	Time intervall on which to perform new measurements. (default: `1.0f`)
`EProcessVizEnum measuredProcess`	Enum specifying the currently tracked process. (default: `EProcessVizEnum::None`)

4.1.2 MethanSensorVizComponent

This component displays the value of the `ProcessComponent` which is currently tracked by the `ProcesVisualizer` at the location of the parent it is attached to.

4.1.2.1 Fields

Signature	Description
`FColor Color`	The color of the displayed measured value. (default: `FColor(0, 255, 0, 125)`)
`UTextRenderComponent* DisplayLabel`	Pointer to the `UTextRenderComponent` used to visualize the process value at the attached parents location. (default: `nullptr`) [set during initialization]
`float UpdateIntervall`	Time intervall [s] on which to update the displayed value of the process at the attached parents location in seconds. (default: `1.0f`)

4.1.2.2 Methods

Signature	Description
`void SetColor(FColor color)`	Set the color of the displayed value of the process at the attached parents location.

4.1.3 ProcessVisualizer

This actor gets spawned during initialization of the `USimulationSystem` and is responsible for controlling the display of the environmental process data ground truth and estimate by usage of two `UProcessComponent`.

4.1.3.1 Fields

Signature	Description
`EProcessVizEnum VisualizedProcess`	Enum specifying the currently visualized process. (default: `EProcessVizEnum::None`)
`UMaterial* DecalMaterial`	Pointer to the material to be used for the `UProcessComponents` managed. (default: `nullptr`) [fetched during setup]
`UProcessComponent* GroundTruth`	Pointer to the `ProcessComponent` handling the environmental process data ground truth received from ROS. (default: `nullptr`) [created during initialization]
`UProcessComponent* Estimate`	Pointer to the `ProcessComponent` handling the environmental process data estimate received from ROS. (default: `nullptr`) [created during initialization]

4.1.3.2 Methods

Signature	Description
`virtual void SetVisualizedProcess(EProcessVizEnum process)`	Set which `UProcessComponents` data shall be displayed, if visualized.
`UProcessComponent* GetVisualizedProcess()`	Returns a pointer to the tracked `UProcessComponent` whose data is currently being visualized, if `UProcessVisualizer` itself is being visualized.
`UProcessComponent* GetTrackedProcessByType(const EProcessVizEnum type)`	Returns a pointer to the tracked `UProcessComponent` for the passed type of environmental process data received from ROS.

4.1.4 ProcessComponent

This component processes environmental process data received from ROS for display. Instances of this are used in `UProcessVisualizer`.

4.1.4.1 Fields

Signature	Description
`FString TopicName`	The name of the topic used to receive environmental process data from ROS (default: “/process”)
`UMaterial* DecalMaterial`	Pointer to a `UMaterial` instance to be used for visualization. (default: `nullptr`)
`UDecalComponent* Decal`	Pointer to the `UDecalComponent` instance to be used to generate the visualization of the received environmental process data. (default: `nullptr`)
`UMaterialInstanceDynamic* OverlayMaterialDynamic`	Pointer to the `UMaterialInstanceDynamic` set for the material of field `Decal` derived from field `DecalMaterial` (default: `nullptr`) [set during startup]

4.1.4.2 Methods

Signature	Description
`bool isValidLocation(const`	Returns `true`, if the passed location in unreal units is covered by the

`FVector& location)`	environmental process data received from ROS, `false` otherwise. In demo mode always returns `true`.
`uint8 valueAtLocation(const FVector& location)`	returns the value of the visualized environmental process at the passed location in unreal units.
`void measurementAtLocation( const FVector& location)`	Notifies the `UProcessComponent` about a measurement at a passed location in unreal units to enable filtering of visualized environmental process data received from ROS in demo mode only.

4.1.5 SimulationSystem

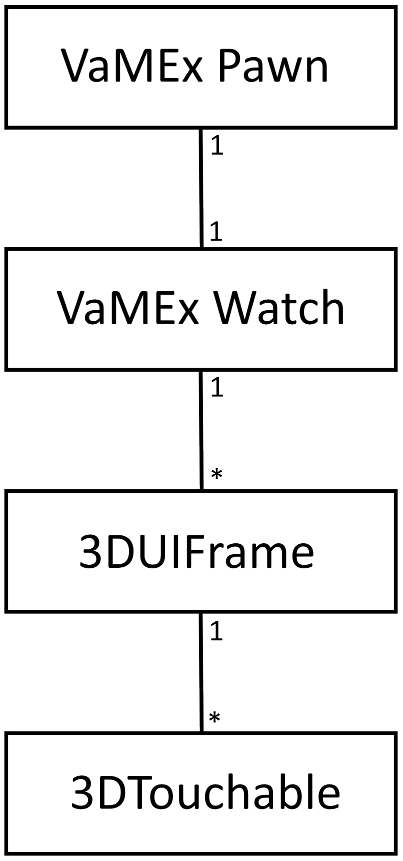
This class is a GameInstance for the Unreal-Engine and the inheriting SimulationSystemBP Blueprint class is set up for that purpose. This is where most of the ROS startup is implemented but also where the start parameters are parsed. Those help configuring the VTB to the users wishes. Valid start parameters from the SimulationSystem side are:

-vr	Used to start in vr mode. Not compatible with -pw, -dc_cluster or -vr-emulate-pw.
-dc_cluster	Used to start in powerwall mode. Not compatible with -vr or -vr-emulate-pw.
-pw	Same as -dc_cluster, used for debugging purposes.
-vr-emulate-pw	Used to start in emulated powerwall mode. Not compatible with -vr, -pw or -dc_cluster
-ovrw	Overwrites the following settings in the SimulationSystemBP unless *OverwriteStartParams* is true. It is usually recommended to give -ovrw as parameter when starting the simulation.
-demo	Starts the simulation in demo mode which effectively disables the exit buttin within the simulation currently.
-visualizationsFiltered	Wether to use filtered visualisations for the methane visualisation or not.
-ezaf	Enables or disables Easy and Fast Mode. Recommended to be used for beginners.
-companion	Enables or disables the companion.

To start the VTB with these they can be appended in a cmd or powershell command this way:
Unreal-Engine_path .uproject_file_path -game append_parameters_here

4.2 Unreal Engine Blueprints Dokumentation

4.2.1 VaMEx Menu System



All menus used in the simulation are bound to a VaMEx Watch which is bound to a players left arm. One Watch has different Submenus which are technically all on the same level in the outliner even though some are practically “deeper” (submenus of submenus) than others. Every Menu derives from the 3DUIFrame which comes with helping functions and a background so all menus have the same size. Each menu can contain multiple 3DTouchable which can come in different flavours depending on the need. Every Menu contains a Back or Close button for convenience. The VaMEx Watch holds functions to switch synchronously (client-server) between the menus. The client-server functionality however is implemented in the VaMEx Pawn.

4.2.2 3DTouchable

The 3DTouchable is the base class for all 3D Buttons used in VaMEx. The derivates such as buttons, sliders or checkbuttons (buttons that stay pressed for true and not pressed for false) hold event dispatchers for buttonDown, buttonUp and their appropriate Highlight colliders to support being pressed by a TouchFinger.

The Basic 3DTouchable reacts to being pressed and released. It has no animation properties besides changing it’s material when being pressed or highlighted.

3DButtons give user feedback by scaling down the button to give a impression of being pressed down.

3DButtonText is a 3DButton with a label for text being attached.

3DChechCubes Stay pressed in when pressed once and release on the second press action making them ideal for boolean decisions in menus.

Sliders can be pressed and moved to trigger an OnSliderMoved event.

4.2.3 VaMExPawn

The VaMEx Pawn inherits from the Unreal base Character and is used in all simulation maps. It's children reflect the different input and output devices. The model including hands is the same for all pawns and implemented here. Also replication for multiplayer is set up in this Pawn.
There are currently 4 Pawns:
- PCPawn:

  For use with mouse and keyboard and regular display device such as a monitor. Has the ability to click on Menus, Robots and the Companion by using the mouse cursor. Can also fly freely around in the level and teleports to robots automatically when clicking on them.

- MotionControllerPawn:

  For use with an HMD and appropriate Controllers. Uses the Motion Controllers as virtual Hands to touch but also click on Buttons, Robots or the Companion. Hands Location will also be used for gestures for other players on the server.
- DisplayClusterPawn:

  For use with a Powerwall and an Optitrack tracking System. Requires the powerwall.cfg to be setup correcty. This can be found in *VaMEx_Unreal\Content\Config.* Menu Controls work the same as seen in the MotionControllerPawn however without the clicking actions. To call the companion a waving gesture is used. Movement is done streching the arms forward for going forward, up for upwards, down for downwards and T-Pose for backwards. The player can turn by holding either the right or the left hand to his right or left.

- EmulatedDisplayClusterPawn:

  For use with an HMD and appropriate Controllers but only exists for testing purposes and can only be selected within the engine. Has the same functions as the DisplayClusterPawn but does not require a powerwall or Optitrack system to work. All input is emulated with motion controllers and output is shown on the HMD.

There is also another Pawn used in the Main Menu. It provides similar controls to the menu as the VaMExPawns do depending on the selected output/input methods. It is not able to move and only required for changing simulation options and starting a map.

4.2.4 Tools

For the user to change the way the simulation behaves 3 tools have been implemented. These can be selected in the SelectToolsMenu. Depending on the mode and tool they receive different inputs from mouse, motion controllers or 3DButtons. All tools however can purely controlled using the 3DButtons.
There are currently 3 Tools implemented which all inherit from the Tools class. They are: Spawning-Tool, Gravity-Gun and Dust Devil-Tool. Further explanation of these Tools is given in chapter **5.4 Interacting in the VTB**.

4.3 ROS Interface

5 User Manual

5.1 Dependencies

- Windows 10
- Unreal Engine 4.25
- Substance in UE4 Plugin (Unreal Marketplace)
- ROSIntegration Plugin (included)
- ROS (only for use with ROS)
- VaMEx Launcher (optional but highly recommended)

5.2 Configuring the VTB

Some of the configuration of the simulation can be changed with the launcher or in the SystemSimulationBP within in the engine. Also it can be overwritten for easier testing.

5.2.1 Configuration in Unreal

The SystemSimulationBP contains all variables passed to it during standalone start which can also be overwritten. These variables include the amount of robots, HoverUI, ConnectToROS, easyAFMode, demoMode and Companion. When Overwrite is enabled passed standalone parameters will be ignored.

5.2.2 Configuration in the simulation

In the Main Menu ROS can be enabled or disabled and another type of UI the HoverUI can be enabled. Also the Amount of Robots can be set.

5.2.3 Configuration of the Components in ROS

5.3 Starting the VTB and ROS

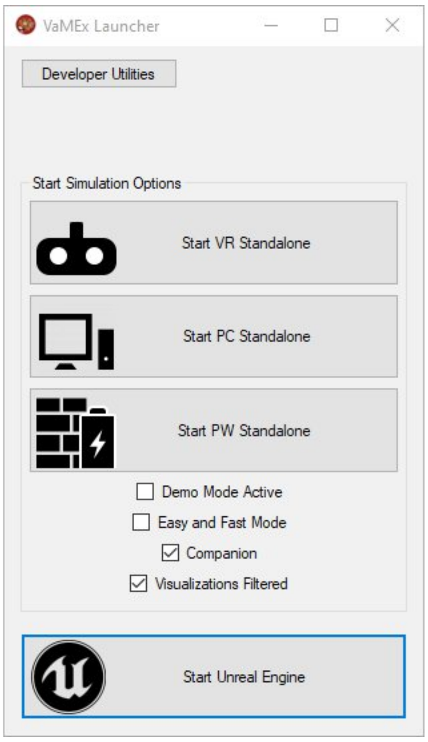
For this Chapter read the Quick Start guide. It provides all information required to start the simulation and ROS.

5.4 Interacting in the VTB

Menu interaction can happen by touching the buttons or clicking them with the trigger button (in VR). To open the menu the watch needs to be activated.
Robots can only be clicked on but they can be tipped over when touched.
In the 'Tools' section of the menu 3 different tools can be equipped. A tool can be unequipped by holding it behind the users back/camera or by pressing the tool toggle button (Left Motion Controller Menu Button or T for PC). The tools are:
- Spawning-Tool: Used to spawn different sized and shaped Rocks
- Gravity-Gun: Used to move Objects around. Without EasyAF it has the ability to activate a second mode by clicking the button on the top right corner of the tool which allows holding Objects. This mode is default in EasyAF Mode. By moving the lever on the side or the mouse wheel (PC) the Object in front of the gun can be moved closer or further away.
- Dust Devil-Tool: Used to spawn a dust devil in front of the user. When pointed at the ground the Dust Devil will go towards the pointed location. To move around on PC use WASD Space and Ctrl. For VR use the Trackpad to teleport around. The trackpad finger position is used to determine the rotation after the teleport when easyAF mode is off.

5.5 The VaMEx Launcher

The VaMEx Launcher can be used to easily start the VTB. It also has some built in utilities to clean up the Binaries and the Intermediate folder of the Unreal-Engine project and can repair Symlinks if their destination does not exist anymore. This was used to externalize the Binaries and Intermediate folder to a RAM drive due to the Unreal-Engines behaviour of writing and reading very often from these locations which can heavily decrease the write cycles of an SSD and speed up the building and hot reload process rapidly especially when using slower storage media.
It works by locating the Unreal-Engine installation folder automatically and starting it with some the given parameters. In case the simulation is started *-game* is being used to start the engine for directly starting a standalone game.



☐ Demo Mode Active
☐ Easy and Fast Mode
☑ Companion
☑ Visualizations Filtered

The Launcher is the recommended way to start the simulation as the editor does not show the variables that can be chosen here with user friendly buttons. Also using the Launcher two instances of the same simulation can be started by staring first the engine and starting an instance using the launcher afterwards.
It is written in C# and can be built using the csproj file located in: *VaMEx_StandaloneLauchner\VaMEx-StandaloneLauchner\VaMEx-StandaloneLauchner\VaMEx-StandaloneLauchner.csproj*.
If Visual Studio says the dotNET Framework version is not the right one, also 6.1 can be use and others may work too. Build the Launcher as Release and copy the resulting .exe from *VaMEx_StandaloneLauchner\VaMEx-StandaloneLauchner\VaMEx-StandaloneLauchner\bin\Release\StandaloneLauchner.exe*
to the projects directory.

6 Expanding the VTB

6.1 Adding another Type of Swarm-Unit

To add another Swarm-Unit, there is a series of things that need to be done to provide the full intended functionality:

- A ROS Interface that supports the desired number of the new Swarm-Unit is needed.
- The Behaviour, Appearance and Components need to be defined by a custom Blueprint.
- For full functionality as well as data consistency a few changes in the USimulationSystem are needed
  - All Swarm-Units are saved in the array SwarmUnits, the new ones should be added
  - numSwarmUnits should be updated to equal the sum of all Swarm-Units now including the new one
  - A new variable for the new Swarm-Unit amount (numberOf<name>s) should be added
- The spawn script should be updated to also spawn the new Swarm-Unit
- On spawn, Swarm-Units of the new type should register to LAOLA.
- The GUI should give control over the new Swarm Units sensor visualisation, offer an info page similar to the other ones and also implement the search feature.
- To ensure proper physical interactions, the collision should be set to match the other Swarm-Units.
- The new Swarm-Unit needs to be replicated
  - To properly replicate and set up the Actor, make sure to factor in network latency in combination with the fact that actor-initialisation is not an atomic operation.

6.2 Adding a new Visualization

If the new visualisation is supposed to be attached to individual actors, such as swarm- or support-units, it should consist of a reusable component implementing its behaviour. To use it it should then get added to the desired Unit. If the visualisation is global, this restriction does not apply and it can be placed in the world using a custom Actor. Currently the visualisations are set for each individual player and done client-side, so no replication is desired.

6.3 Setting up a new Map

When a new Map is used, it requires a valid NavMesh in the middle because of automatic robot spawns. Also a LAOLa Localisation Actor is required for the visualisations to work and a Clock-Actor for the ROS timings to work. For the Beacons there is the BeaconsSpawner and for Orbiters to work the Orbiters Spawner is required.

References

Im aktuellen Dokument sind keine Quellen vorhanden.