SPECIALIZATION PROJECT

An Outer Wilds Tribute

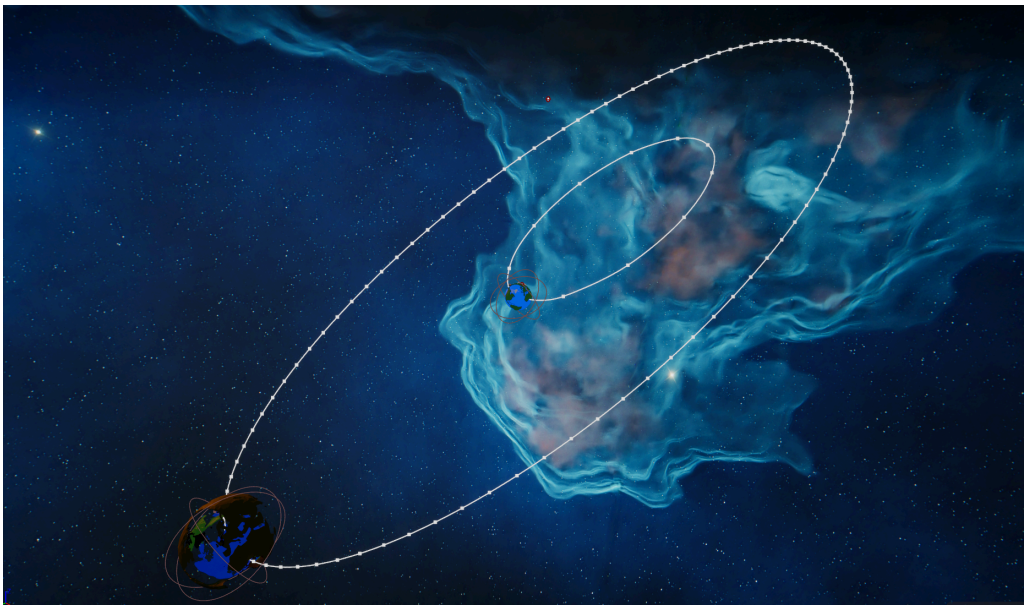
Nicolò Smaniotto
Gameplay Programming
Github

Event Horizon School 2024/25

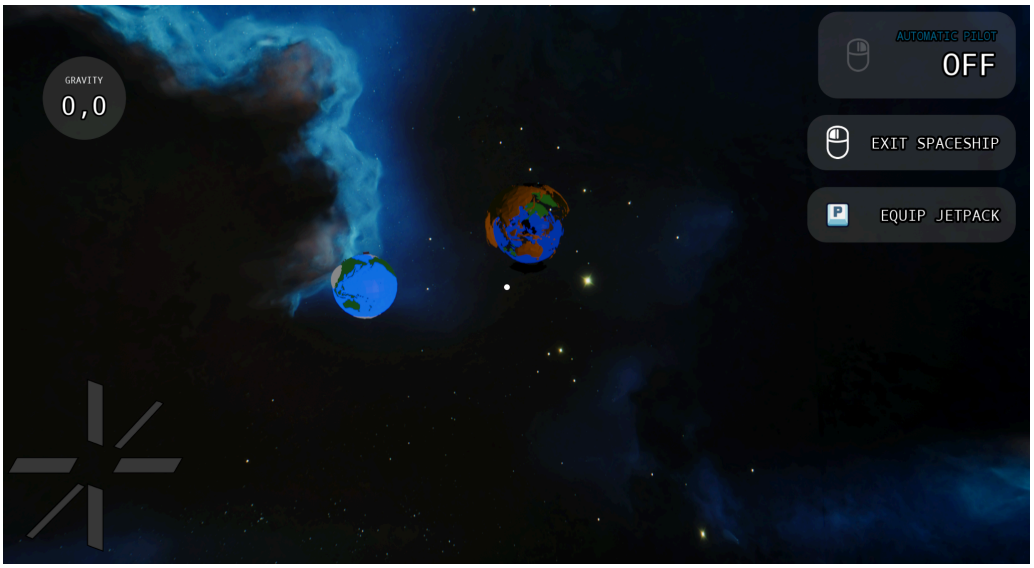
Overview

This project represents the specialization work for my final year at Event Horizon School. Developed over the course of approximately two months, it is focused on gameplay programming.

The goal was to replicate within Unreal Engine 5 the core movement mechanics of the well-known game Outer Wilds, by implementing components that leverage the engine's built-in physics system to apply forces and impulses to objects.



Furthermore, the project integrates additional components for environmental interaction and for representing specific gameplay behaviors through simple and intuitive UI interfaces.



Challenges I found were mostly derived from the attempt to smoothen the unpredictable behaviors of the physics simulated objects.

Gameplay

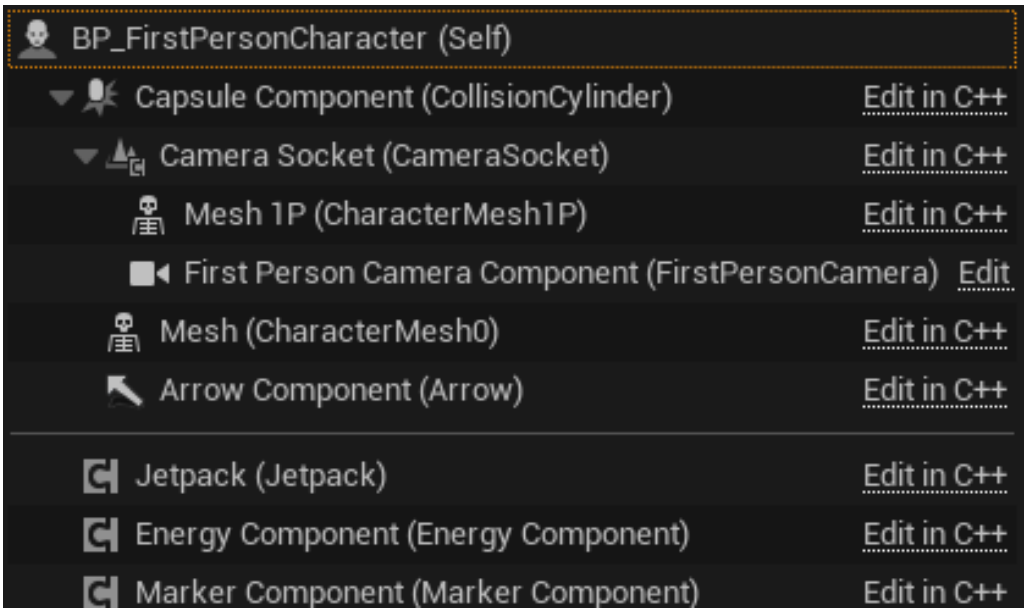
Player

Overview

The player class (ASpecializationCharacter) presents a basic behavior in it, demanding all the actual implementations to its attached components. Thus, apart from short logic implementation mostly linked to the *Look Input* and the camera rotation, its job is to redirect all the inputs it receives to the attached components.

Structure - Attached components

- Jetpack,
- Marker component,
- Energy component.



Event dispatcher

*DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(**FOnSpaceshipInteraction**, bool, OnSpaceship);*

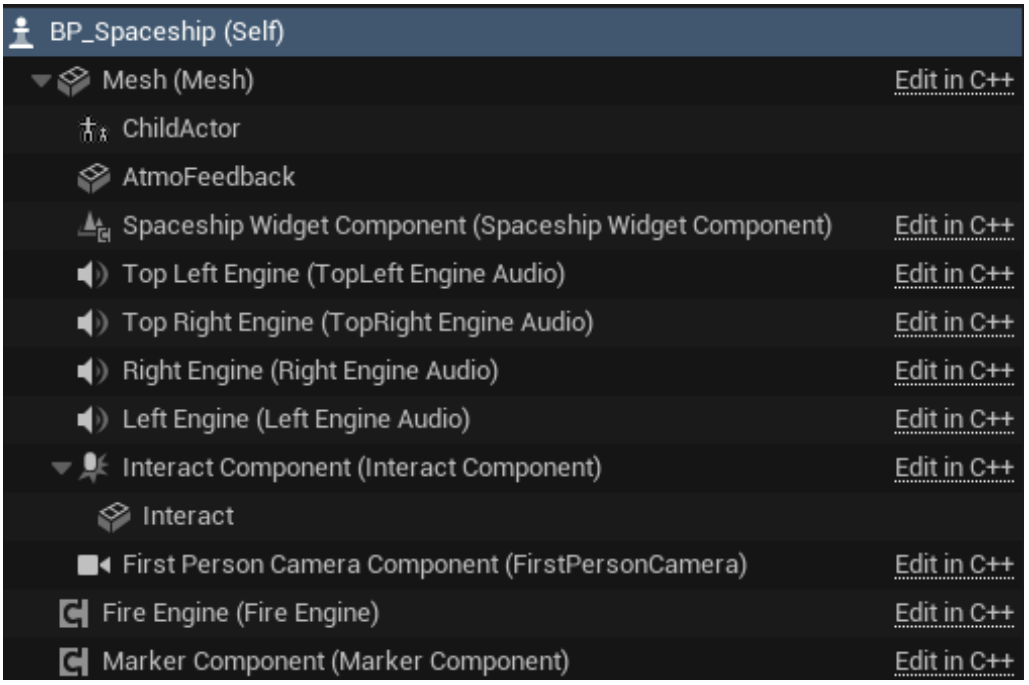
Spaceship

Overview

The spaceship class (ASpaceship) is a pawn derived class which, as for the player, redirects its input to the attached components.
It can be possessed to be driven by the user. The possession function is an override of the IPossessable interface function Possess. Linked to the spaceship, there is an Interaction Component that the player can interact with in order to take possession of it.

Structure - Attached components

- Fire engine,
- Marker component,
- Energy component,
- Interaction component,
- Jetpack interaction,
- SpaceshipWidget component.



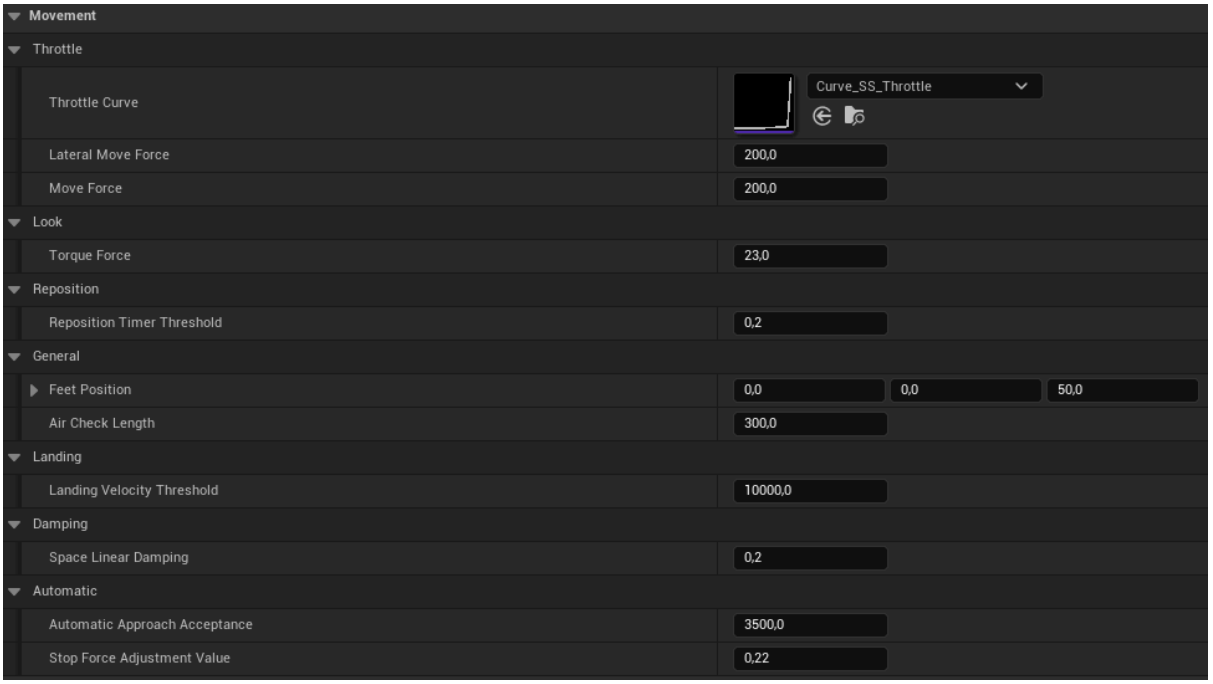
Fire Engine

Overview

The Fire Engine component is an actor component that manages all the on air movement capabilities of an actor.
This component:

- provides several public functions that the owner can use to navigate through the air;
- stores important information about the atmosphere it is in (if any) (*i.e.* gravity force, planet ref, hit surface).
- Provides public functions that the atmosphere itself can call to notify updates to calculations regarding forces affecting the owner physics component.
- Grants the possibility to activate an automatic pilot to reach a certain marked object (if a marker is given as parameter for the *SetDependencyComponents* function).

For this to work by default, the owner must be a physically simulated object as movement is achieved through the use of linear and torque forces.



Event Dispatcher

*DECLARE_DYNAMIC_MULTICAST_DELEGATE_TwoParams(**FOnAtmoForce**, FVector, AtmoDir, float, Magnitude);*
*DECLARE_DYNAMIC_MULTICAST_DELEGATE_TwoParams(**FOnGravityUpdate**, FVector, OldGForce, FVector, NewGForce);*
*DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(**FOnAutomaticPilot**, bool, Active);*
*DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(**FOnAtmosphereUpdate**, APlanet*, Planet);*
*DECLARE_DYNAMIC_MULTICAST_DELEGATE_ThreeParams(**FOnVerticalMovement**, FVector, ForceDir, float, Magnitude, FTransform, Transform);*
*DECLARE_DYNAMIC_MULTICAST_DELEGATE_ThreeParams(**FOnLateralMovement**, FVector, ForceDir, float, Magnitude, FTransform, Transform);*

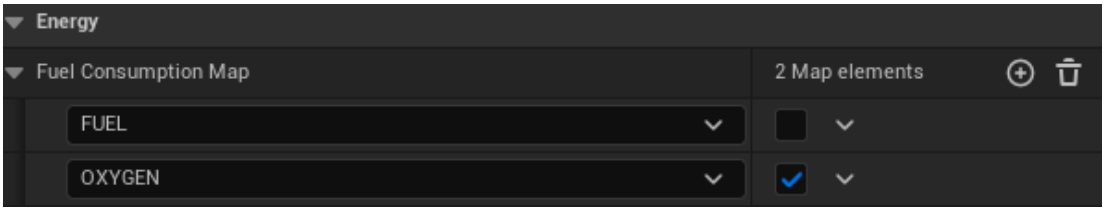
Jetpack

The jetpack is a fire engine derived class that, in addition to the super class behavior, allows the on-land movement. As everything is physics based, characters are not capable of moving making use of the built-in *character movement* component: the jetpack provides this type of movement, calculating and applying compensations for the planet movement shiftness on characters.

Additionally, it grants an overridable custom alignment reposition function which operates unbound to physics, in order to preserve an almost always consistent rotation of a specified object (see IRepositionable interface).

The jetpack also presents an energy consumption and a propulsion behavior.

- **Energy consumption behavior:** in order to move it consumes some energy through the use of the energy component. The energy types must be defined in the *FuelConsumptionMap* variable and the energy component must be passed in the overload of the *SetDependencyComponents* function.



- **Propulsion behavior:** when the gravity force applied to the owner is not null, the flying movement is subject to a limitation due to propulsion which is a value that decays when movement is actuated. Movement is only possible when this value is greater than zero.

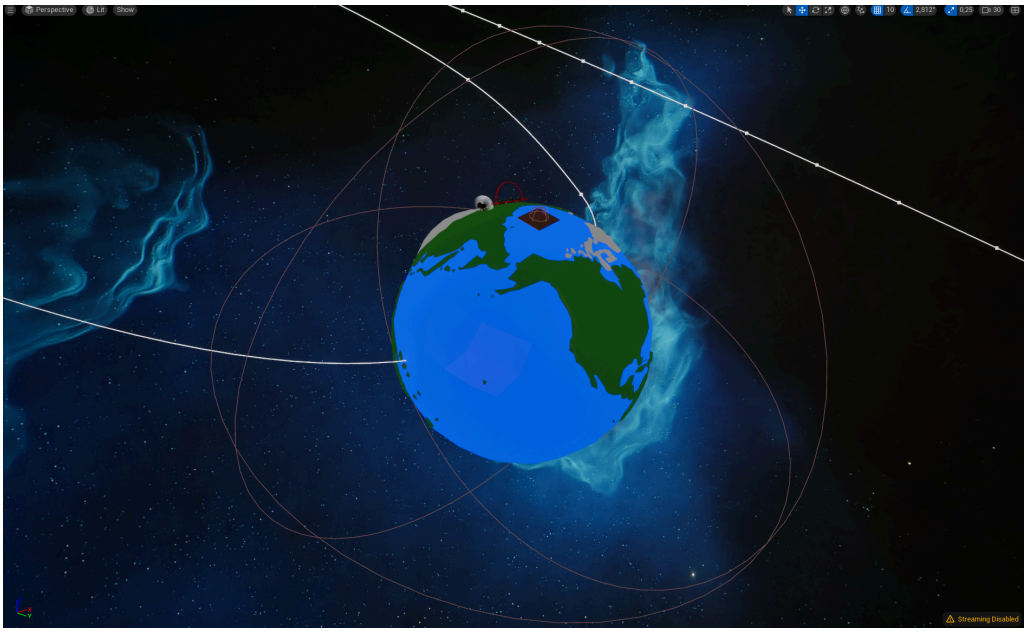
Event Dispatcher

DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FOnJetpackEquip, bool, IsEquipped);
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FOnMovement, EMovType, MovType);

Planet

The planet is the actor which owns the actual planet mesh (and all of the actors attached to the planet) and its relative gravity bound.
Its job is to give at demand information about the current state of the planet (location, linear velocity, angular velocity direction) and to execute the planet rotation and revolution movements.

It is worth noticing that the planet actor itself remains immobile while the mesh is the actual moving object.
The revolution movement is made by a spline that ideally would revolve around an object (as the orbits are more elliptical than circular, I decided to use a spline).

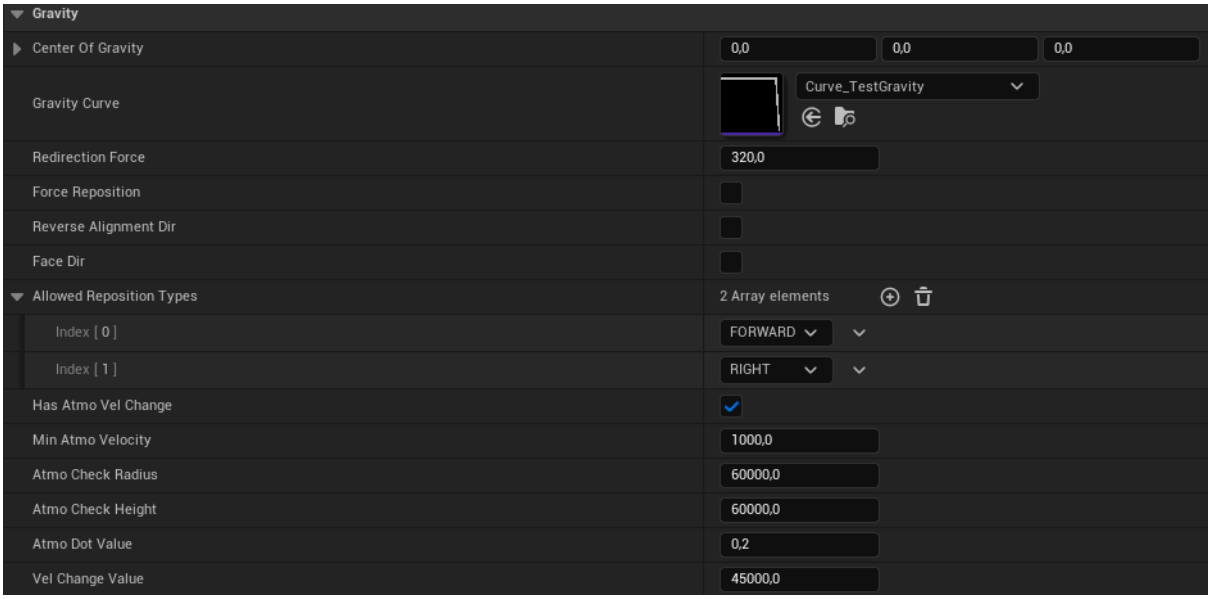


Gravity Bound

This capsule collision derived component manages and calculates all the forces that should be applied to a physic object. Different tasks are executed on the objects inside its area according to their nature. Tasks are:

- Gravity: applies gravity to objects according to a gravity curve.
- Alignment: calculates torque forces to align objects to a custom direction (default: object up vector aligned with reversed gravity direction).
- Atmosphere Velocity Change: applies a tangential force to the object if in a “liminal” space of the atmosphere. The space size can be edited.

TASK \ NATURE	No-physics object	Physics object	Physics object with Fire Engine
Gravity	NO	YES	YES
Alignment	NO	NO	YES
Atmosphere Velocity Change	NO	NO	YES

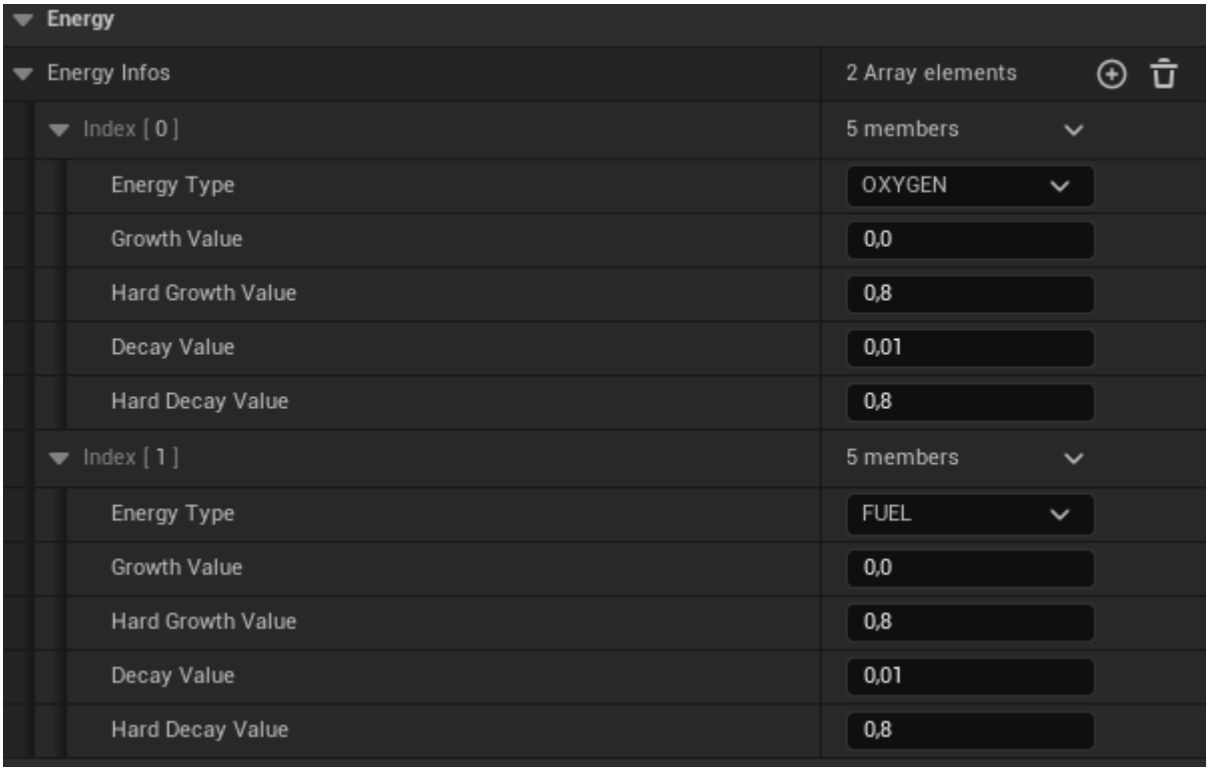


Forced Gravity Bound

Child class of Gravity Bound. Should be placed inside an outer gravity bound and applies its gravity bound logic to objects without letting the outer gravity bound affect the object. When the object leaves this boundary, it restores old behavior of the outer.
Examples of usage: spaceship landing platform, player moving tunnels.

Energy Component

Actor component that holds and manages the available energy types an object can use to exploit its operations. For every ***FEnergyInfo*** passed in construction, it is created and stored an object of ***UEnergyStructure*** class which manages the single energy type data.



Event Dispatcher

DECLARE_DYNAMIC_MULTICAST_DELEGATE_ThreeParams(**FOnEnergyUpdate**, EEnergyType, EnergyType, float, OldValue, float, NewValue);

Energy Structure

UObject derived class which manages a single energy type according to its passive usage or direct consumption by an energy component. A hard behavior is possible and it overrides the “normal” behavior. If a hard consumption is requested in conjunction with a hard restore, the hard restore takes precedence.

Event Dispatcher

DECLARE_DYNAMIC_DELEGATE_OneParam(**FOnEnergyEnd**, EEnergyType, EnergyType);
DECLARE_DYNAMIC_DELEGATE_ThreeParams(**FOnUpdate**, EEnergyType, EnergyType, float, OldValue, float, NewValue);

Interactable Component

Child class of Sphere Component. In order to work, it is necessary that at least one among the owner and the owner components implements the *IInteractable* interface. When the player interacts with this component, the interaction is redirected to every found *IInteractable* implementing object.
It also asks the HUD to toggle the interaction widget for visual feedback.

Marker

This component is used to check marking components along the way. Once a marking component is found, it can be locked and used for automatic piloting if possible.

Event Dispatcher

DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(**FOnLock**, bool, IsLocking);

Marking Component

Child class of widget component which:

- visualizes the actual marker in scene,
- changes linked widget behavior according to locking/unlocking,
- dynamically calculates all the necessary forces in order to align the asker linear movement with the owner's linear movement.

In order to work, it is necessary that at least one among the owner and the owner components implements the *IMarkable* interface.

UI

My HUD

HUD child class that sets up all the most commonly used screen widgets. It manages the transition between them and the priority of the widgets. It exposes a function to display a certain widget depending on the type requested.
The transition widget can be displayed with the *ExecuteTransition* function that accepts two delegates to be executed (mid-transition, end-transition).

Event Dispatcher

DECLARE_DELEGATE(**FOnTransition**);
DECLARE_DELEGATE(**FOnTransitionEnd**);

Base Menu

HUD base class. It owns an instance of the Base HUD widget in it.

Jetpack Menu

It derives from the Base Menu. It displays info about the jetpack:

- oxygen,
- fuel,
- propulsion.

Spaceship HUD

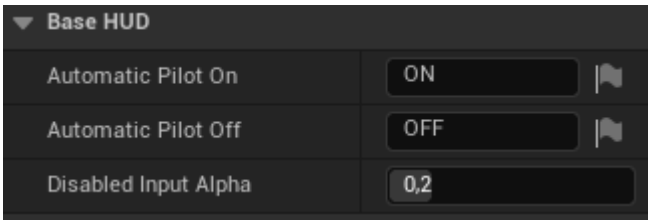
It derives from the Base Menu. It displays info about the spaceship:

- Spaceship Mov UI,
- jetpack equipping/unequipping.

Base HUD

Widget presenting base info an HUD has:

- gravity,
- automatic piloting.

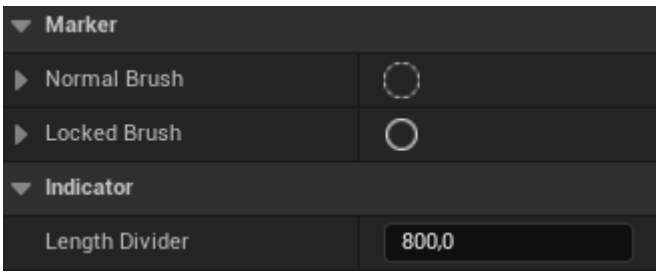


Spaceship Mov UI

A widget representing the actual movement in the 3D axis through the use of progress bars.

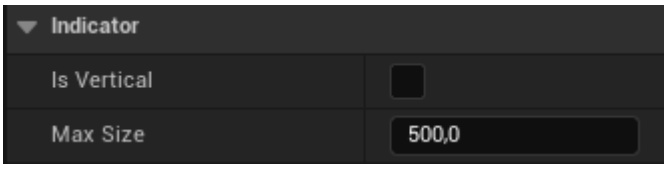
Marker Widget

Widget that is applied to a marking component to visualize the actual marked object. The approach forces are adjusted by length divider and then dispatched to four children indicator widgets.



Indicator Widget

Visual indicators of approach forces used by a fire engine automatic pilot.



Transition Widget

Widget that executes a simple fade-in and fade-out with two different animations. An event is fired at the end of the animations so that the My HUD instance can execute custom events depending on the transition state.

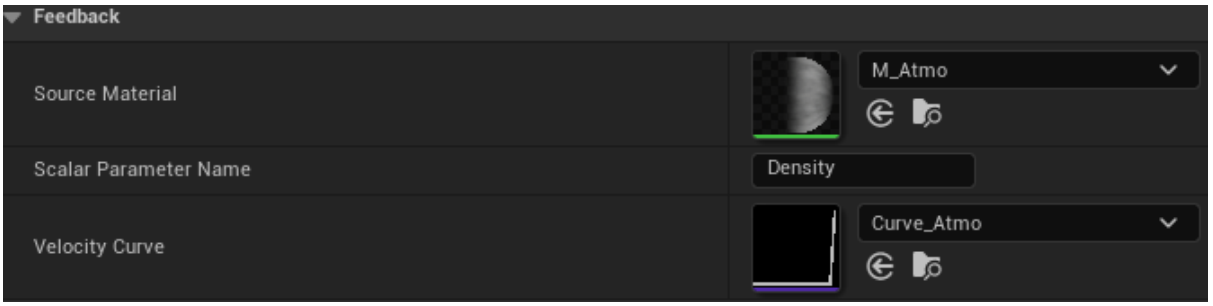
Event Dispatcher

DECLARE_DYNAMIC_MULTICAST_DELEGATE(**FOnMidtimeTransition**);
DECLARE_DYNAMIC_MULTICAST_DELEGATE(**FOnCompletedTransition**);

Other classes

Atmo Feedback

Static mesh component attached to elements for atmosphere feedback. A curve defines the intensity of the material according to the object current velocity.

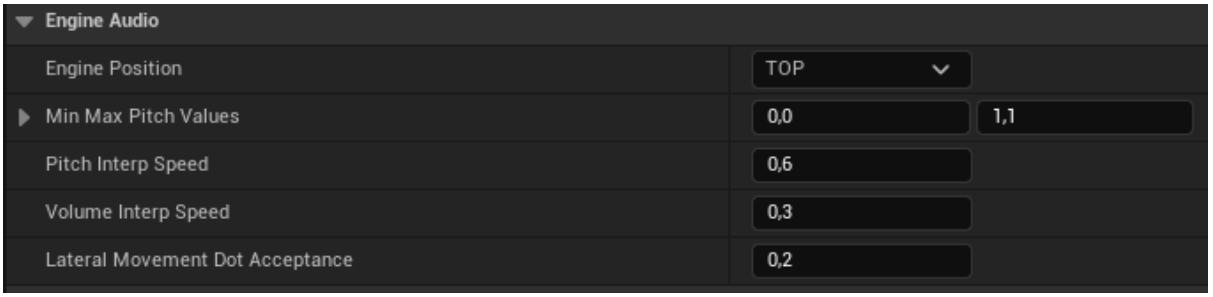


Energy Restorer Interactable

Zone where an Energy Component with a specific Restorer Type will be refilled upon interacting.

Engine Audio Component

Audio component for an actor that presents a fire engine.
It is necessary to position the component according to the engine position variable for spatial audio.

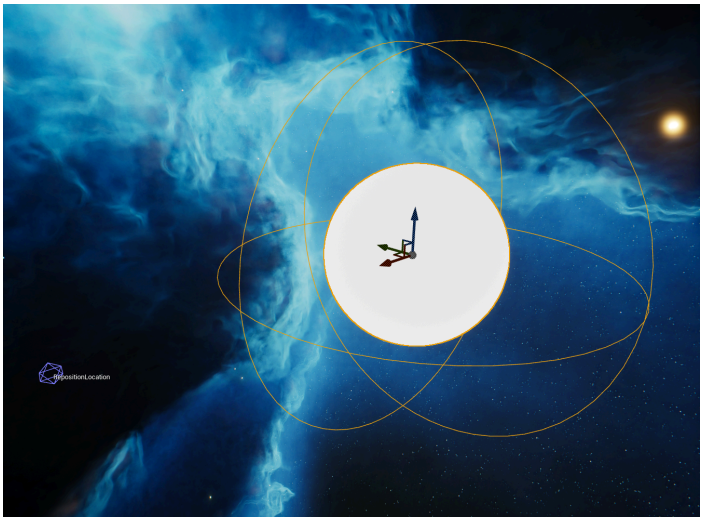


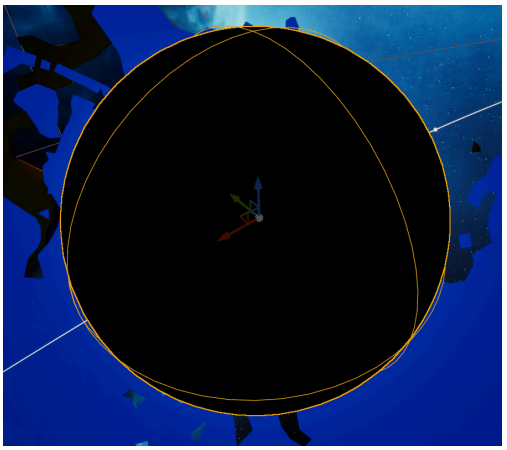
Hole

Actor representing a base structure of a hole. It presents a *static mesh component* and a *sphere component* for hole collision. The hole collision begin/end overlaps are overridable by sub-classes such as *White Hole* and *Black Hole*.

Black Hole

A white hole ref is needed to teleport to the linked location on overlap.





White Hole

Repulse every actor that enters its collision.

Restorer Zone

Zone where an EnergyComponent with a specific RestorerType will be refilled when entered.