Dimension Shift

Technical Design Document

ISART Digital - Final Year Project 2025

Authors: BERTRAND Louis, BOURGOGNE Romain, DEVINE Vincent, GUTIERREZ DIAZ David, LEPINE Quentin, SEBIROT Iris

1. Summary

2. Changelog

Version	Date	Description
v1.0.0	19/05/2025	Update layout
v0.5.2	18/05/2025	Update UML diagram
v0.5.1	17/05/2025	Update diagram + Add Save system part
v0.5.0	13/05/2025	Spelling correction
v0.4.3	30/04/2025	Diagrams and text modified in line with feedback from v0.3.3
v0.4.2	29/04/2025	Updated Bark + UI Navigation, added Hint System & Level Selector
v0.4.2	25/03/2025	Update layout
v0.4.1	22/03/2025	Add Crosshair snapping
v0.4.0	15/03/2025	Add Snapping
v0.3.3	22/02/2025	Update diagram
v0.3.2	18/02/2025	Update TV, Grab
v0.3.1	25/01/2025	Update Light Through TV
v0.3.0	27/12/2024	Update Technologie Context + Game Overview
v0.2.3	05/12/2024	Update Technical Analysis, Critical Point, Diagram
v0.2.2	03/11/2024	Update Technical Analysis, Critical Point, Diagram
v0.2.1	20/11/2024	Update layout, Technical Analysis, Technological context, Critical points
v0.2.0	05/11/2024	Added Technical Analysis, Complexity and Reference
v0.1.2	26/10/2024	Update layout
v0.1.1	15/10/2024	Added team members, gameplay, technological context (platform) Updated technological context (tools), Critical points
v0.1.0	09/10/2024	Created the TDD document Added technological context (tools), Critical points, Norms

3. Team members

Game Artist 2D

BOUVIER Aurèle
PIERRET Martin

Game Artist 3D

BERTRAND Louis
LABERDURE Manon
PINHEIRO David

Game Design

BASSELIER-MARCHAL Axel
LEFEBURE Tristan
RODRIGUES Melanie
SAINTILAN Loïc

Game Design & Programming

GUTIERREZ DIAZ David
SEBIROT Iris

Game Programming

BOURGOGNE Romain
DEVINE Vincent
LEPINE Quentin

Marketing

YUEYAN Leya

Music & Sound Design

CIMIA Nathan
CLERC Marie

Producer

BRIDENNE Martin
CAPRON Alice

4. Game Overview

Dimension Shift is a **solo PC game**, **first person** and **puzzle-based**

You play as Sally as who has just received your engineering degree.
To celebrate, your grandmother, who's an eccentric inventor, has invited you to visit her strange workshop.
But when you get there she's nowhere to be found. Instead there are all of her crazy inventions. Amongst them, there are these weird TVs that can change your gravity and size.

But then, you get a memo from your grandmother. She's challenging you to find a way through all the rooms using her inventions!

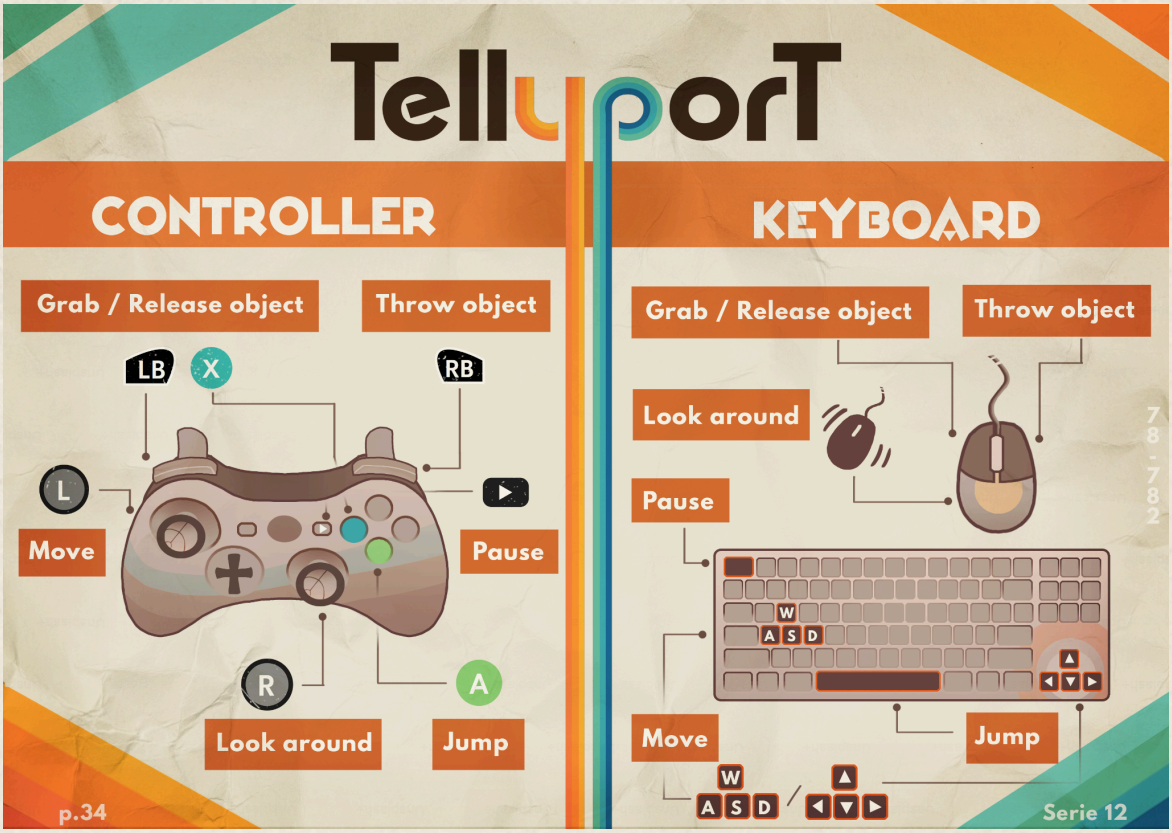
Dimension Shift introduces a variety of TVs, each with unique effects that alter the player and objects:

- **Gravity TV**
  Upon crossing a TV, players will experience a shift in gravity, the direction of which is determined by the orientation of the exit TV.

- **Size TV**
  Players will undergo a size transformation when they pass through a TV, impacting their ability to navigate and interact with the environment.

Main Inspiration:
Our game draws inspiration from several titles that have pushed the boundaries of puzzle design and player perception. **Portal 2** serves as a key reference for its masterful use of physics-based puzzles and seamless integration of mechanics into the narrative. Additionally, **Inception** influences our approach to reality-bending mechanics by reinforcing the sense of immersion and wonder as players manipulate their environment in unexpected ways.

Control Scheme: The control layout is as follows:



5. Technological context

5.1. Target

The target audience for Dimension Shift includes players who enjoy first-person puzzle games with a focus on logic and problem-solving. These players are often intellectually curious, enjoy narrative-driven experiences, and appreciate innovative game mechanics. The audience ranges from teens to adults, usually tech-savvy, and drawn to games that challenge their thinking in a fun, engaging way.

5.2. Platform

The platform target is the **PC** for easier development and better performance.
Modifying and working with the rendering system requires a solid understanding of the engine's rendering process. Doing this on another platform would require extensive research for the team, as we lack expertise in this area on other systems. Therefore, focusing on the PC will allow us to create a better and more optimized system compared to developing it for other platforms.

5.3. Tools

5.3.1. Engine

Unreal Engine 5.4 was chosen for its ability to efficiently handle the integration and optimization of our TV system. Its advanced rendering pipeline, combined with **Lumen's** real-time global illumination, ensures accurate lighting and reflections across TV, enhancing visual consistency without the need for pre-baked lighting.

The game's **special feature** requires extensive knowledge of the engine's rendering system. Unreal Engine, by providing easy **access to its source code**, enhances our understanding and allows for deeper customization. Additionally, **Blueprints** provide an intuitive scripting system, enabling non-programmers to contribute effectively, whereas alternative solutions, such as Unity's Visual Scripting, lack the same level of flexibility and functionality.

Since our TV system requires displaying real-time views of an alternate space, maintaining strong **performance** is essential. The combination of **C++** for low-level optimizations and Lumen for efficient dynamic lighting ensures a visually immersive experience while keeping the game performant.

5.3.2. Versioning

To **simplify** the use of the version control system for team members with less experience in such systems and to minimize issues related to merging binary files, Perforce was a logical choice. Its **file-locking system** prevents others from modifying a file, thus providing a **solution to the merge problem**.
Additionally, Perforce can be easily integrated into Unreal Engine, which simplifies its use for the team by allowing them to work with a single software instead of two.

5.3.3. Audio

Wwise was selected as the sound middleware for the game due to its seamless integration with Unreal Engine, ensuring a smooth development process. Wwise offers significant autonomy for Sound Designers, allowing them to create and implement audio content without the need for constant support from programmers.

Compared to other sound middleware options like FMOD, Wwise provides robust features such as advanced audio routing, real-time sound parameter control, and an efficient soundbank management system. These features enable greater flexibility and scalability in creating dynamic, interactive audio experiences, which are essential for the game's design.

6. Technical Analysis

6.1. Feature List

6.1.1. TVs

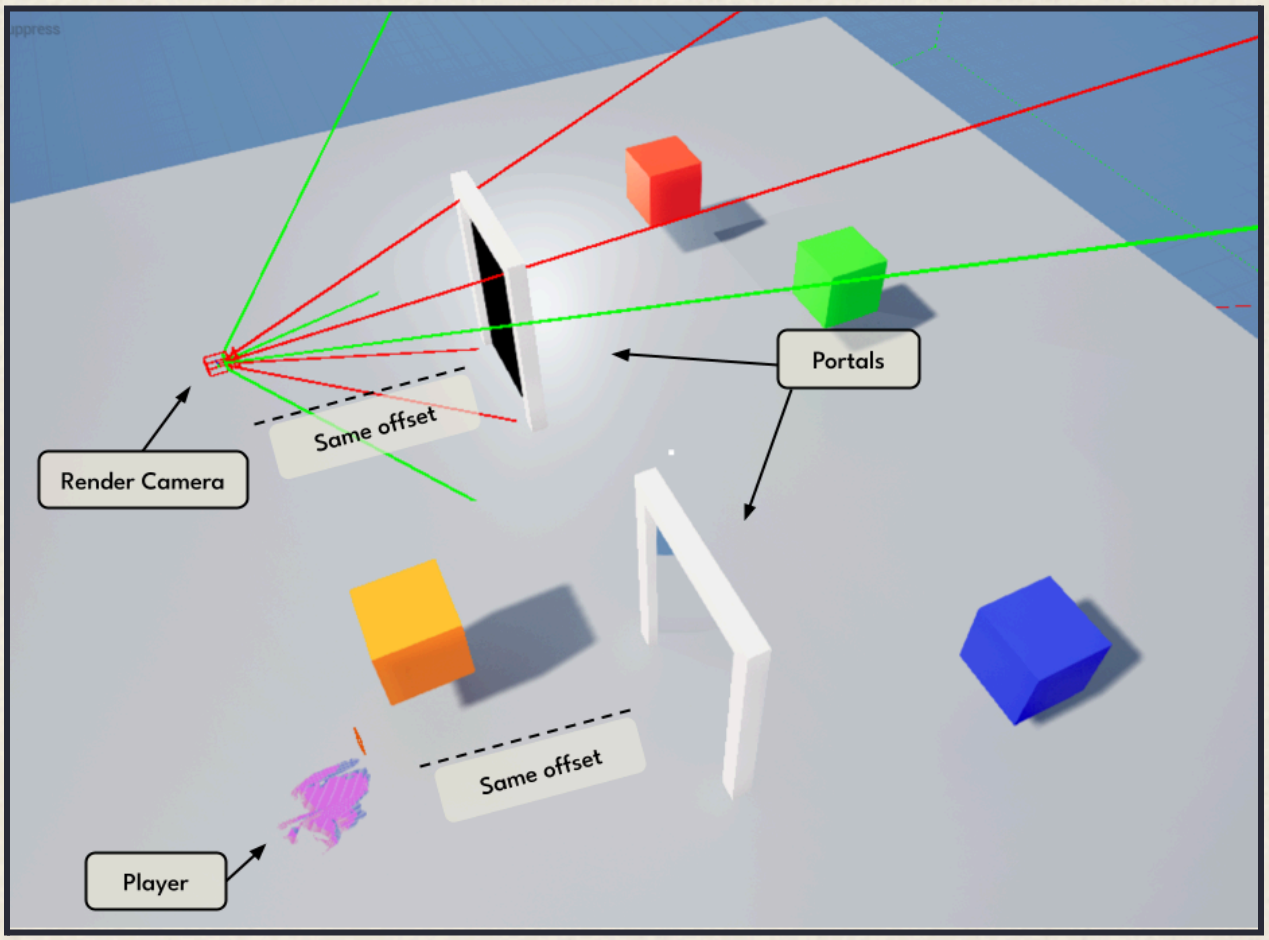
The TV system is essential and central in our game. Two approaches were possible for implementing it:

- Render Camera Methods, quick and easy to implement.
- Stencil Buffer Methods, more optimized but also much more complex.

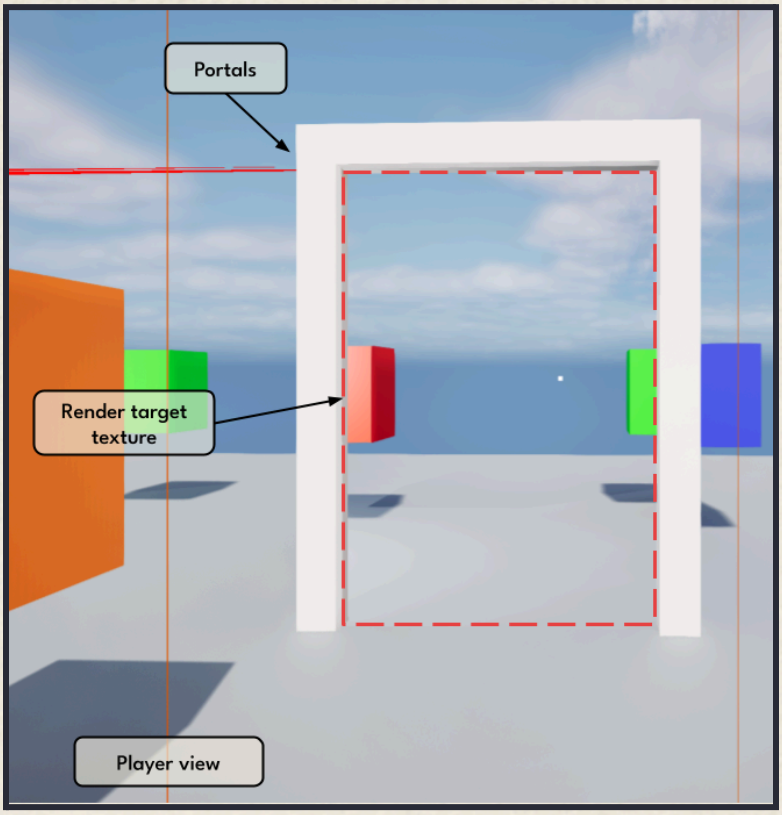
To quickly provide the team with a solution that allows them to test their ideas, the first approach is the most appealing. The second method is also much limited and is left as a technical note more than anything.

6.1.1.1. Using render cameras to draw TVs

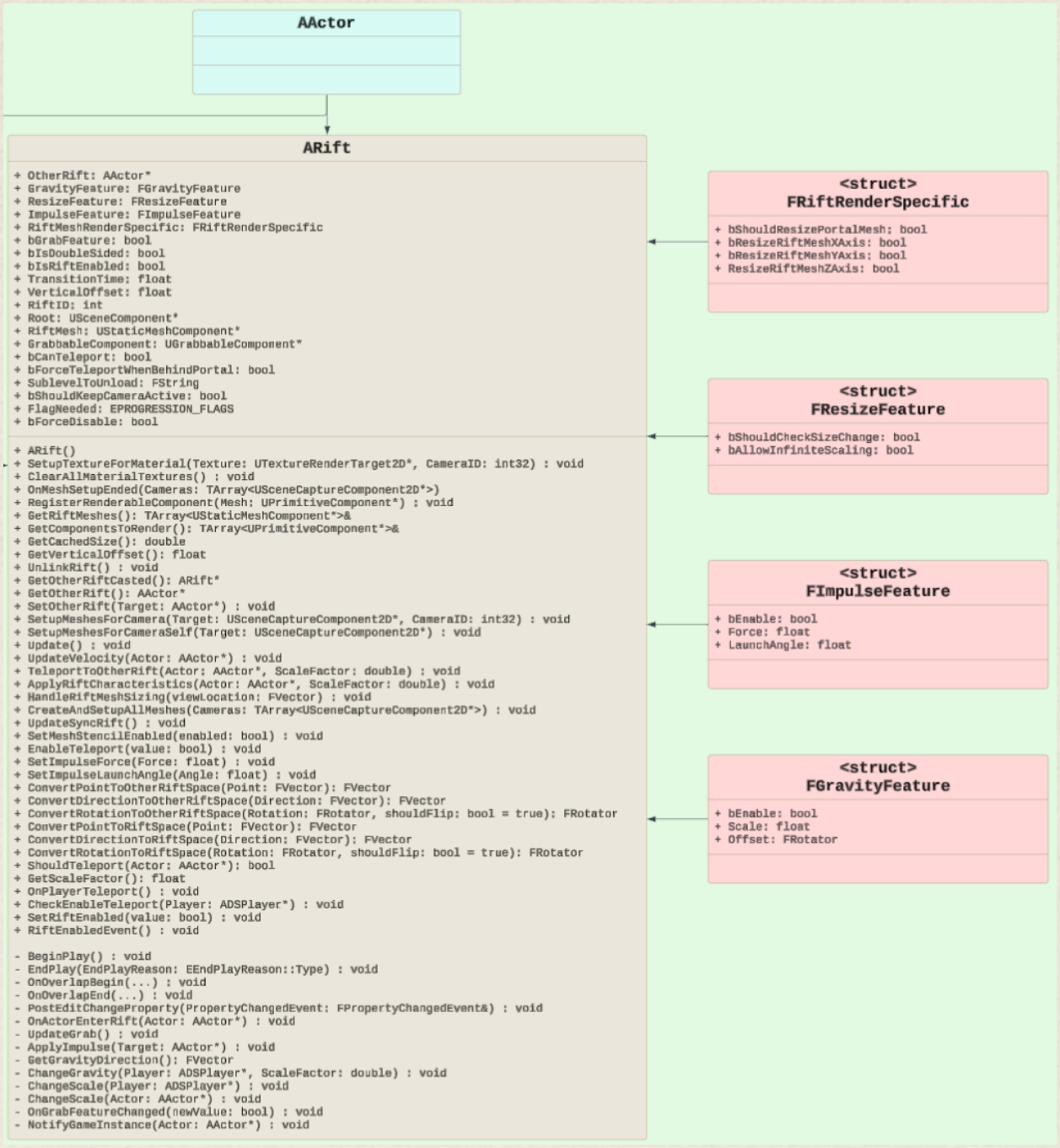
The first method makes use of render cameras in order to see through the TV. The scene is first rendered from the point of view of the other TV, with an offset applied to the render camera's position and orientation. This allows us to have an image that when cropped, the "inside" of the TV gives us the exact view we would have if we were directly at the other TV's location.



The scene is rendered from the player's point of view, while the TV is displayed as a solid object using the previously created render target texture. The TV needs to be rendered with a special shader that will sample the texture using the screen coordinates as UVs, instead of the ones provided in the model itself. This allows us to perfectly cut out the part we want and display it inside the TV.



UML Class diagram



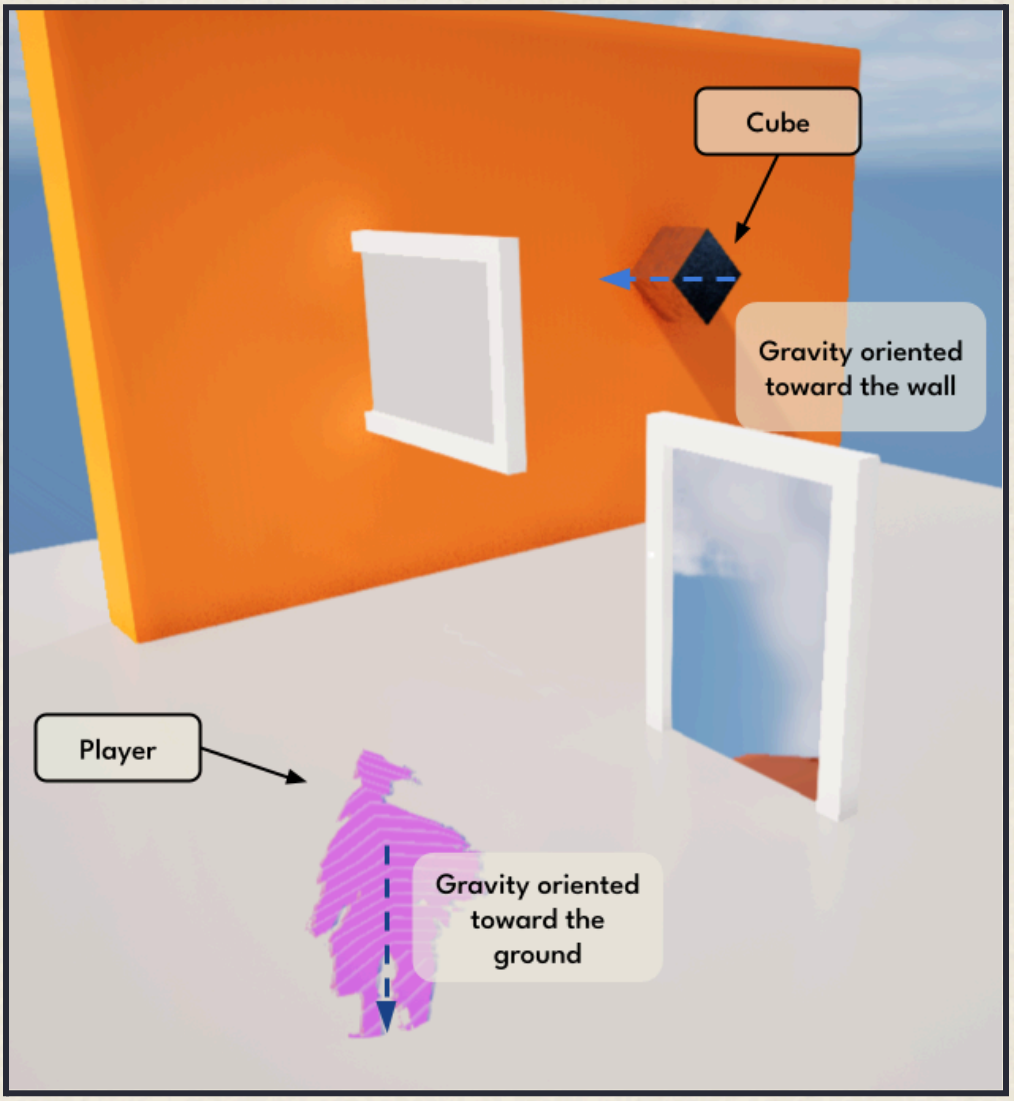

UGameInstance
URiftManager
+ MaxPortalCount: int32
+ RenderTargets: TArray<UTextureRenderTarget2D*>
+ CurrentSave: TObjectPtr<UDSSaveGame>
+ RenderTargetScaleFactor: float
+ RenderTargetSecondScaleFactor: float
+ ProgressionFlags: int32
+ bReloadScene: bool
- CachedResolution: FIntPoint
- SaveName: const FString
- SaveId: const int
- bFirstTimeGetSaved: bool
+ Initialize() : void
+ Shutdown() override : void
+ CheckViewportSize() : void
+ GetTextureTargetAt(i: int32): UTextureRenderTarget2D*
+ GetAllTextures() const: const TArray<UTextureRenderTarget2D*>&
+ GetViewSize() const: FIntPoint
+ ActiveOptionalPathEvent() [BlueprintImplementableEvent]
+ ActiveOptionalPath() : void
+ IsLevelFinish(id: int): bool
+ PlayerReturnToHub() : void
+ UseProfil(newProfilUse: int) : void
+ GetLastLevel(): int
+ GetProfilGameData(id: int): FProfilGameData&
+ GetLastTimePlayString(LastTime: FDateTime): FString
+ GetTotalTimePlayString(TotalTime: FTimespan): FString
+ FirstTimeSave() : void
+ UpdateTimeSave() : void
+ PlayerStartLevel(levelID: int) : void
+ PlayerEndLevel(levelID: int) : void
+ PlayerStartLevelEvent(levelID: int) [BlueprintImplementableEvent]
+ PlayerEndLevelEvent(levelID: int) [BlueprintImplementableEvent]
+ SetProgressionFlag(ProgressionFlag: EPROGRESSION_FLAGS) : void
+ GetProgressionFlag(ProgressionFlag: EPROGRESSION_FLAGS): bool
+ PlayerPassThroughTV(id: int)
+ SaveGameData() : void
+ GetGameData(): UDSSaveGame*
+ DeleteGameData() [Exec]
- CreateGameData() : void

6.1.2. TV Gravity

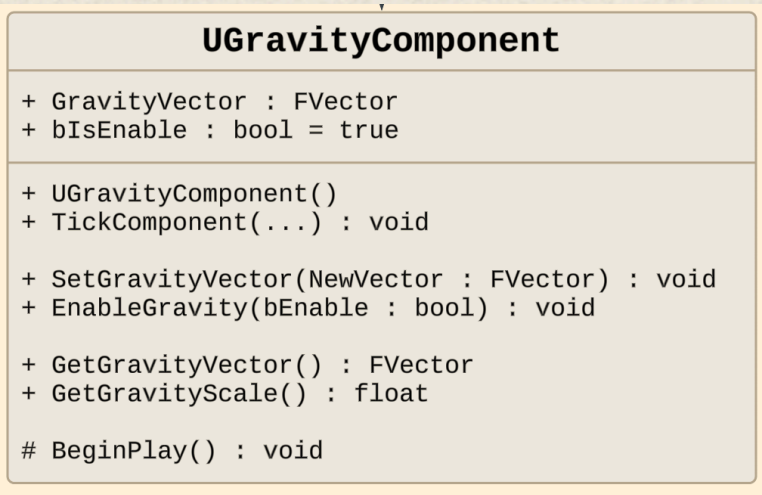
When the player or an object passes through a TV, they will experience a new gravity effect based on the orientation of the exit TV.

Player Gravity
Using Unreal Engine, the character movement component provides a gravity parameter that can be dynamically adjusted. Upon exiting a TV, the player's gravity is updated to match the TV's orientation, ensuring seamless transitions.

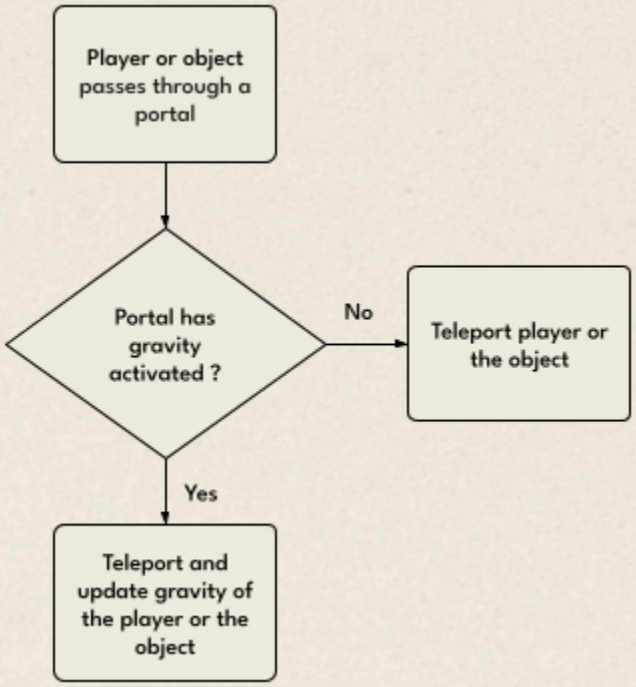
Object Gravity
For objects, managing gravity is more complex since they lack a character movement component. To address this, a custom Blueprint Component (**BPC**) is implemented to handle object gravity. This BPC dynamically updates an object's gravity vector when it passes through a TV, enabling consistent behavior across all interactable elements.



UML Class diagram

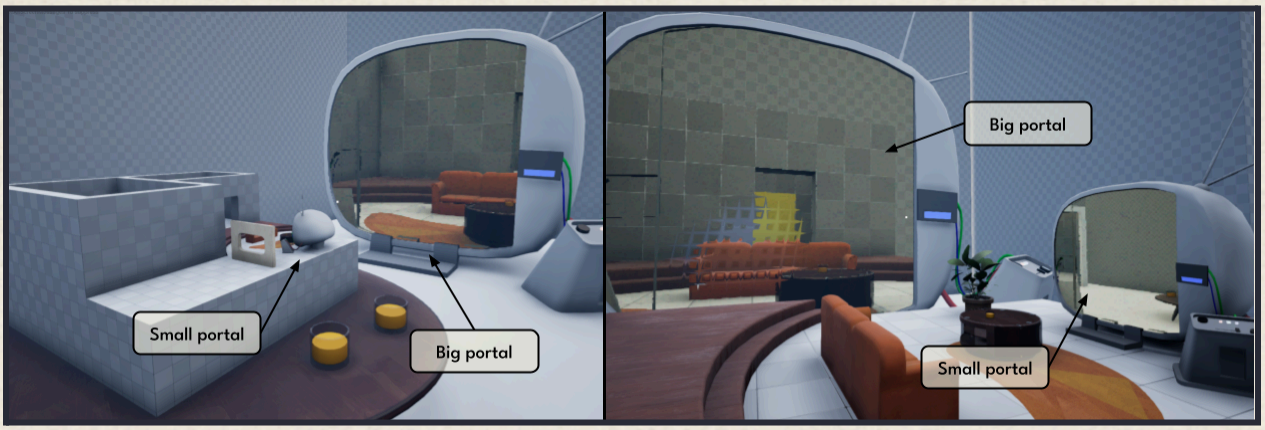


Interaction diagram



6.1.3. TV Size

Players or objects will undergo a size transformation when they pass through a TV, impacting their ability to navigate and interact with the environment.

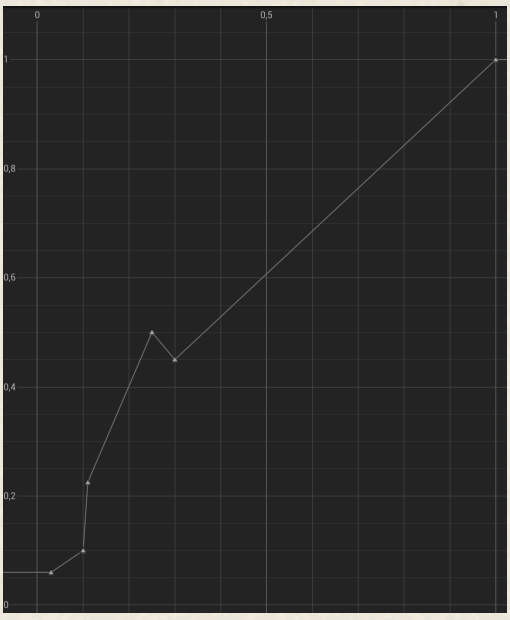


When the player or an object passes through a TV, their size is modified relative to the scale of the exit TV.

To maintain consistent physics, a lot of properties like gravity and acceleration are adjusted proportionally to the new size. All properties are scaled linearly, but the player speed is scaled using a curve for a better feeling. This ensures that the gameplay mechanics remain coherent despite the change in scale.

```
float ScaleFactor = ExitRift->GetScale() / EnterRift->GetScale();
```

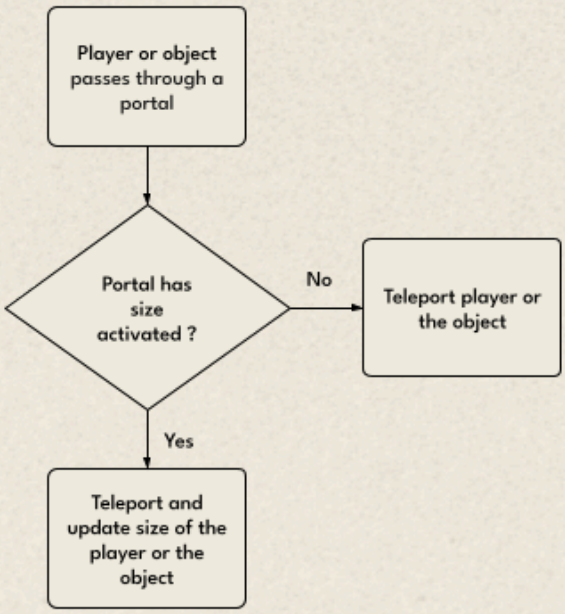
Player scale



Curve for player speed (speed multiplier/size)

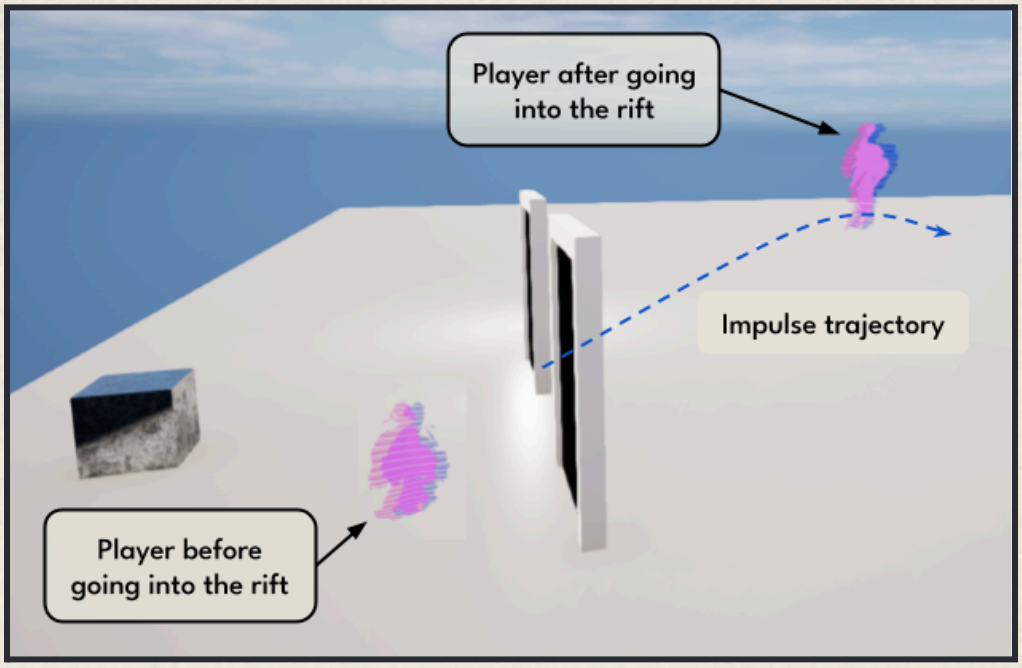
Scaling the TV's visual effect is a bit harder, but can be solved using a simple method. We can just scale the relative vector from the TV to the camera according to the TV's size. For example, a camera placed on a TV of scale two will be twice as far away from the TV.

Interaction diagram

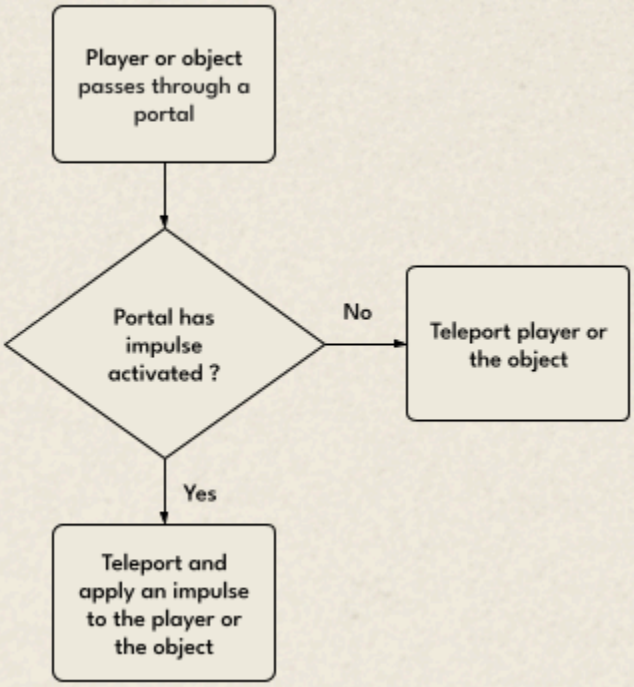


6.1.4. TV Impulse

When the player or an object passes through a TV, an impulse is applied based on the TV's orientation, along with a specified angle and force.



Interaction diagram



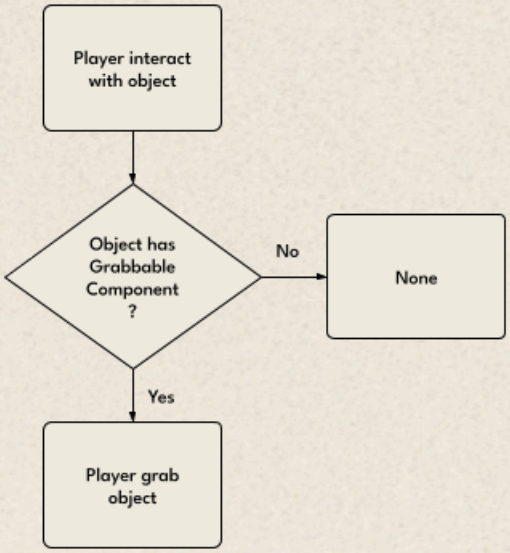
6.1.5. Grab

The game allows players to pick up, move, and throw objects while maintaining their physical properties. This feature enhances interactivity and adds depth to puzzle-solving mechanics.

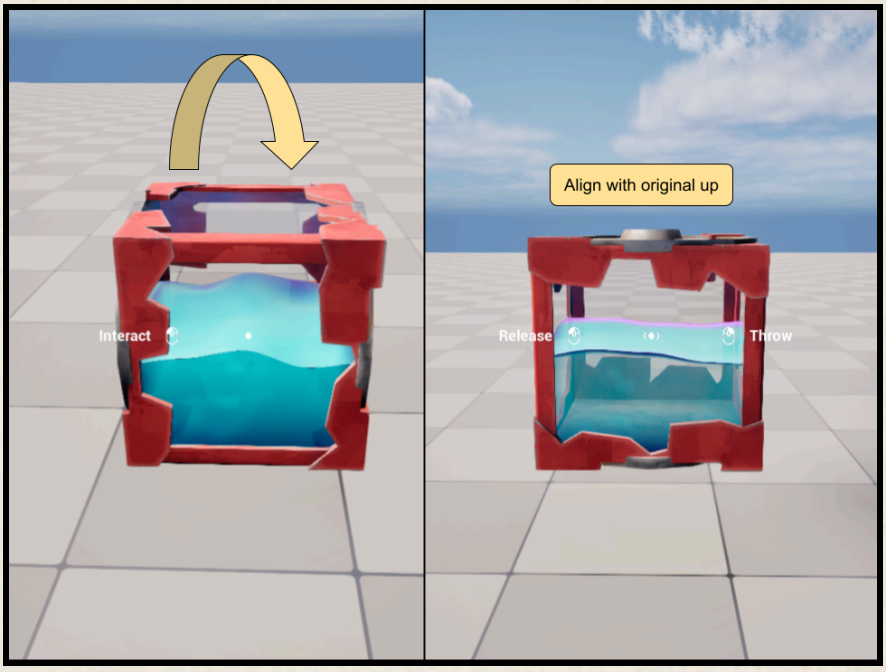
To implement this functionality, a Blueprint Component (**BPC**) was developed. This component can be added to any type of Actor, enabling it to be picked up and manipulated by the player. By leveraging this modular approach, the functionality becomes reusable across various objects, ensuring flexibility and scalability within the game's design.

UML Class diagram

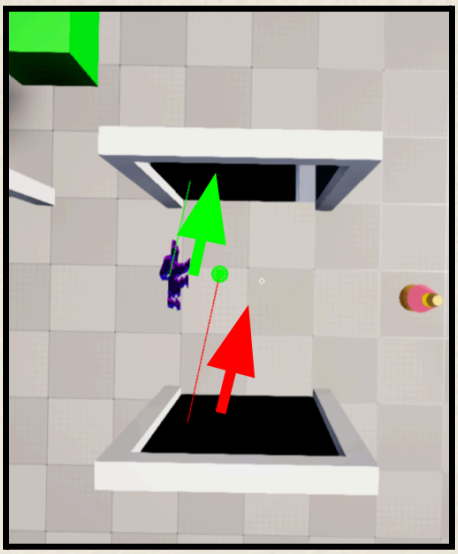
Interaction diagram with player



Lerp with Rotation on Grab When a grab is initiated (via **SetGrabbed**), the component stores the current rotation and calculates a target rotation based on the player's camera orientation. The **StartGrabLerp** method computes a smooth rotational transition using spherical linear interpolation (**Slerp**) over a defined lerp time. During each frame update, **UpdateLerpRotation** advances this rotation, ensuring that the grabbed object aligns naturally with the player's view while taking gravity's direction into account to align with the original up of the object. This approach not only provides visual polish but also maintains consistent orientation during the manipulation of objects.



The **RaycastThroughTVs** method is a critical part of the grab functionality that ensures that objects behave correctly when interacting with TVs (TVs or spatial discontinuities). The component leverages the player's **RayCastThroughTV** method to perform raycasting that "sees through" TVs. This method gathers multiple hit results including those from both the primary ray and additional offset sample points to determine the most accurate target location. The resulting data (such as raycast depth and a list of intersected TVs) is used to adjust the grab behavior, ensuring the object's position and scale are correctly updated relative to any TV transitions.



To maintain robust and error-free gameplay during object manipulation, several dynamic constraints are enforced within the **UpdatePosition** method. The code not only updates the object's position smoothly but also adapts to changes in its environment especially when interacting with TVs by performing detailed raycast checks and scale adjustments. For example, after calculating the target location via raycasting (using **DoRaycast**), the method compares the new result with the previous one (with **HasDiff** and **GetChangeState**) to determine if the object has transitioned through a TV boundary.

Overall, the object follows the player's movement fluidly (using an exponential decay function for smooth transitions) while dynamically adapting to environmental changes. This integrated approach maintains both visual consistency and gameplay integrity during complex interactions such as moving through TVs or adjusting to variable object sizes.

Throw/Release Releasing an object either as a simple drop or a deliberate throw—is handled by methods such as **ReleaseGrab** and **ReleaseThrowGrab**. When an object is released, the component:

- Restores the original collision responses (previously altered to prevent undesired interactions with player during grab).
- Re-enables gravity or delegates to a gravity component if present.
- Calculates and applies new linear and angular velocities based on the object's movement while being grabbed. For throwing, an extra force is added, ensuring the resulting velocity does not exceed pre-set maximum limits. This careful restoration of physics and collision properties ensures that, post-release, the object behaves consistently within the game world.

To maintain gameplay integrity, several constraints are enforced:

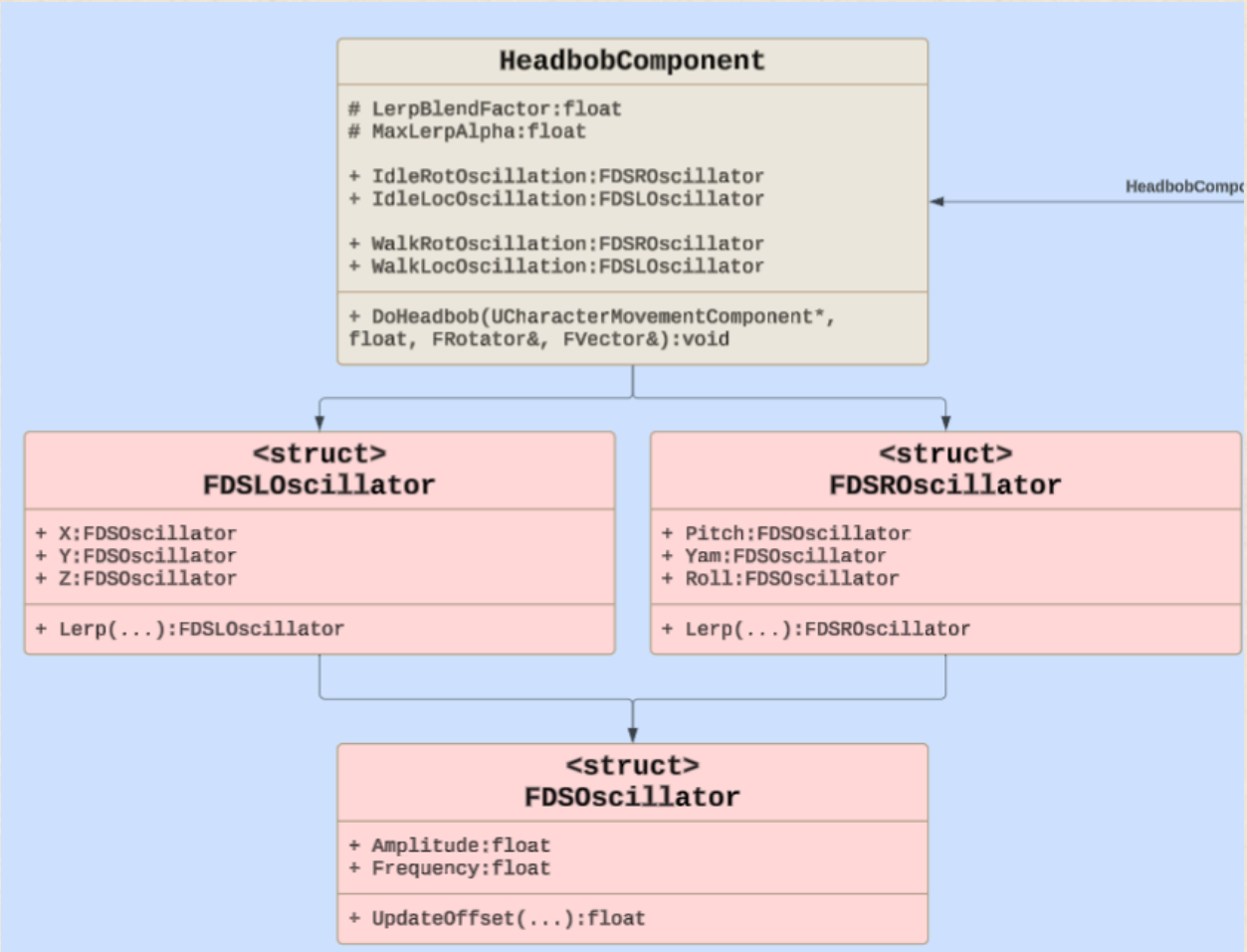
- The player cannot release an object if it remains inside the player's collision bounds. This prevents the object from unintentionally passing through walls.
- Objects that are too large are prevented from passing through a TV. Both the object and the player must be appropriately sized relative to the TV's dimensions.
- An object that exceeds the maximum allowable size for a specific TV cannot be grabbed through that TV, ensuring proper spatial behavior.
- Similarly, objects that are too big for the player cannot be grabbed. This avoids situations where the player might become embedded within the object or experience a softlock.

6.1.6. Walk

To enhance player immersion and realism, a Blueprint Component (**BPC**) was created to simulate natural head movements. This component adjusts the player's head position and orientation dynamically, whether the character is in motion or stationary.

The primary objective of this system is to provide a sense of realism without overcomplicating the movements, ensuring that the game remains accessible and easy to read. Subtle, context-appropriate head movements are used to maintain a balance between immersion and gameplay clarity.

UML Class diagram



6.1.7. Barks System

To enhance story immersion, a system dedicated to controlling the various dialogues was created. The desired behavior of this system is simple: to trigger a dialogue (visual subtitles & voice) according to specified conditions and parameters.
For the system to work we have:

- 2 UI Blueprints, responsible for displaying the dialogues.
- 5 Blueprints, responsible for managing the dialogue triggering logic.
- 1 Actor Component, who holds a dialogue reference (DataAsset), and has the "TryPlay" function that has to be called manually when we want the dialogue to play (by the designers).
- 4 Data Assets, which hold variables for the different dialogue texts, the Wwise sound events to play, and the different trigger conditions.
- 3 Data Tables, which hold a key name, the dialogue texts, and sound event references.
- 1 String Table, which holds all dialogues and facilitates localization.
- 1 Struct, for the Data Tables (texts & sound event references).
- 1 Enum, for bark types

The "BarkManager" blueprints are there so only a unique instance of a dialogue is created, and this way we can prevent dialogues from triggering repetitively.

The bark system works as following:

An AC_BarkPlayer is given to any object or blueprint in the game that should trigger a bark.

1.1. That blueprint sets up its own logic on how to trigger the bark

1.2. The AC_BarkPlayer has an instance editable DA_Bark (DataAsset) reference to pick the bark that the player should play.

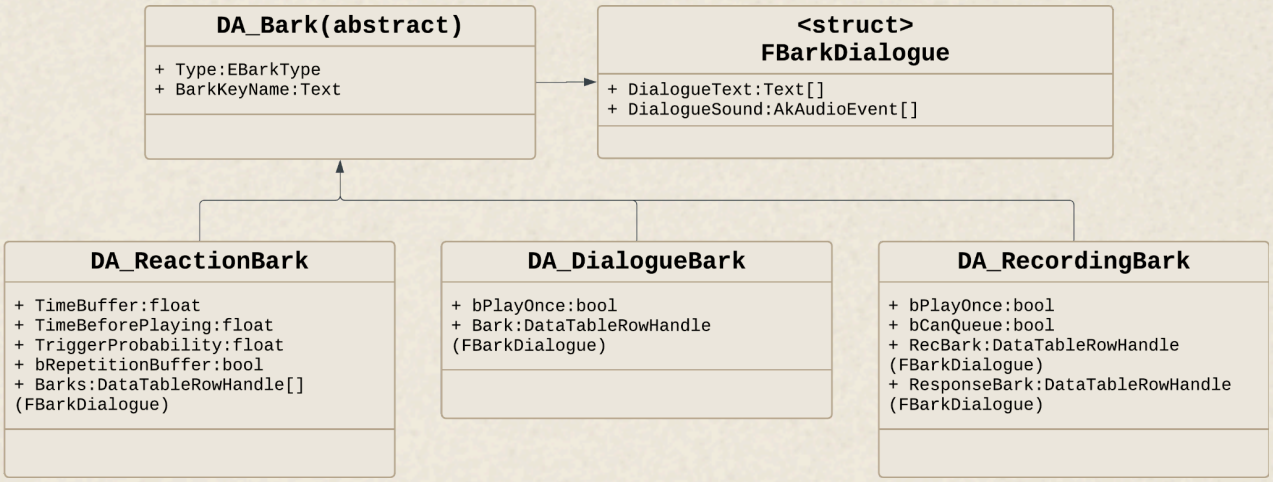
1.2.1. A DA_Bark DataAsset contains information about the dialogue texts to be shown as well as the Wwise sound events that'll be played along with the texts.

When the application starts, each AC_BarkPlayer searches for a reference to its own BP_BarkManager.
When a blueprint triggers a bark, the AC_BarkPlayer requests its manager to play a bark, the manager performs a check to see if it can play a bark and would do so if it can.
The manager holds a reference to the WBP_BarkDisplay widget, and when a bark needs to be played, it calls the WBP_BarksDisplay's "PlayBark" function, giving it information from the DA_Bark DataAsset that it holds.
Once the "PlayBark" function is called, the WBP_BarkDisplay creates a WBP_BarkText, which receives an array of the dialogue texts and Wwise events to play in a sequenced fashion. Each Wwise sound event is hooked-up with an event once the sound stops playing to determine whether a new line of dialogue should be played or if it has played all of the dialogue lines and should be destroyed.

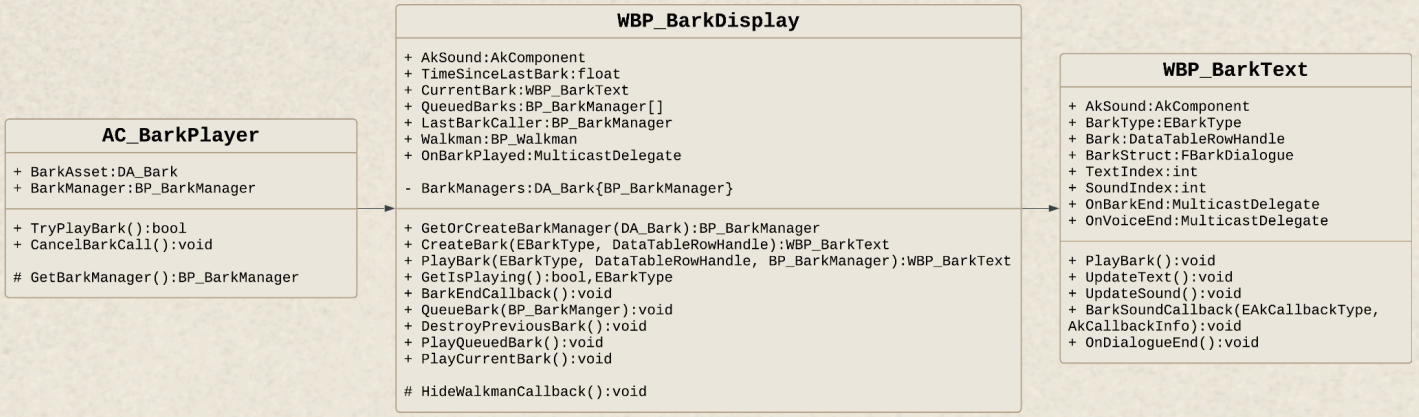
UML Class diagrams

1.0 - Bark managers

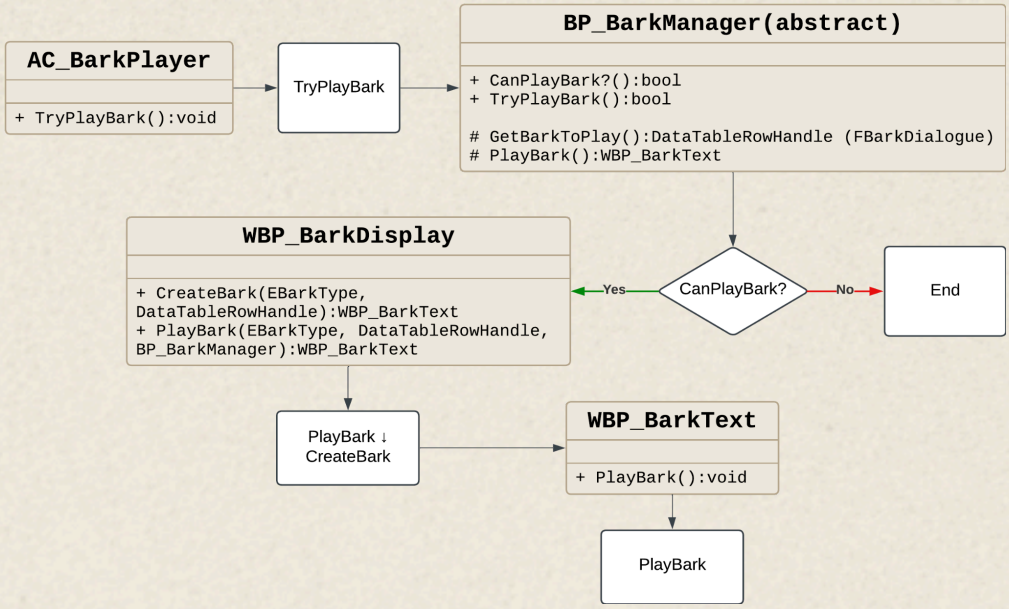
1.1 - DataAssets



1.2 - Player (caller) & displayer (UI)



1.3 - General flow



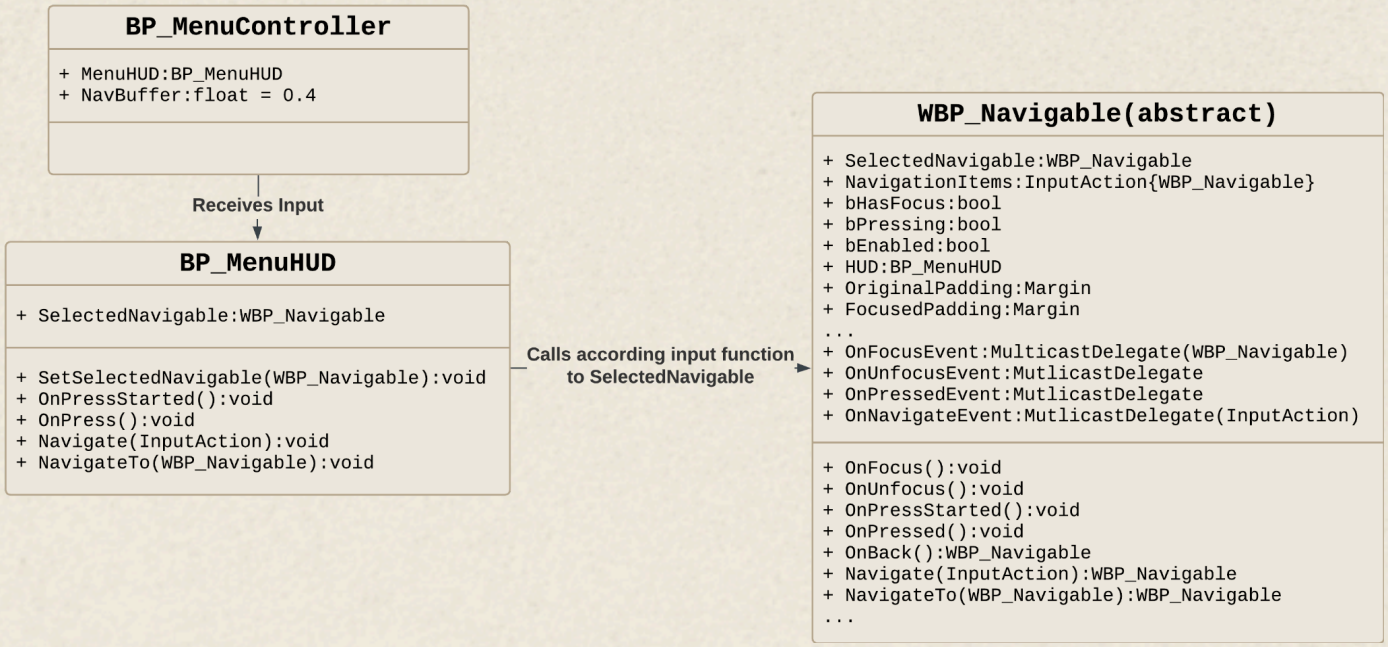
6.1.8. UI Navigation

As the game supports controllers, the UI has been adapted to work with this hardware. A completely new navigation system was made for the game, not using the native Unreal Engine UI navigation system, as it has many flaws we couldn't account for and needed more customization.
The new UI navigation system allows us to control interactions with as many inputs as we want, and to nest input & navigation.

3 classes are responsible for the system to function:

- WBP_Navigable, an abstract class that contains the navigation functionality, from which every widget that uses navigation must inherit.
- BP_MenuHUD, inherits from the base Unreal HUD class & keeps a reference to the currently selected "navigable" element, and controls their navigation.
- "Navigation Controllers" (BP_MenuController & BP_PlayerController), which read and send the input values to the NavigableHUD reference.

UML Class diagram



The BP_MenuController receives the UE5 EnhancedInput Actions, information which is then sent and treated by the BP_MenuHUD.

The BP_MenuHUD serves as a manager that keeps track of the current panel that was opened and then sends input data to that current panel.

Through the WBP_Navigable abstract class, the BP_MenuHUD can always keep track of the panels as they all inherit from this class. These panels have generic inherited functions to set up all of the UI input and navigation logic that is needed that are accessed and called by the BP_MenuHUD.

6.1.9. Snapping

This snapping system streamlines the precise alignment and assembly of cubes, allowing you to easily place them exactly where needed by automatically connecting an object to a predefined snap zone (the SnapComponent) when specific conditions are met.

Initialization & Configuration

- **Detection Sphere:**
  In the constructor, a SphereComponent is created and configured to detect overlaps (using "Overlap" mode). This sphere serves as a detection area to identify when another object enters the snap zone.

- **Snap Zone Setup:**
  The sphere is given a specific radius, and during BeginPlay, the object's dimensions (via UStaticMeshComponents) are analyzed to position the sphere correctly above the object based on its scale.

Overlap Detection & Event Handling

- **Overlap Begin :**
  When another actor enters the sphere, conditions are checked (e.g., the actor is not itself, it has a GrabbableComponent, and it is being grabbed). If these conditions are met, the SnapComponent is assigned to the GrabbableComponent.

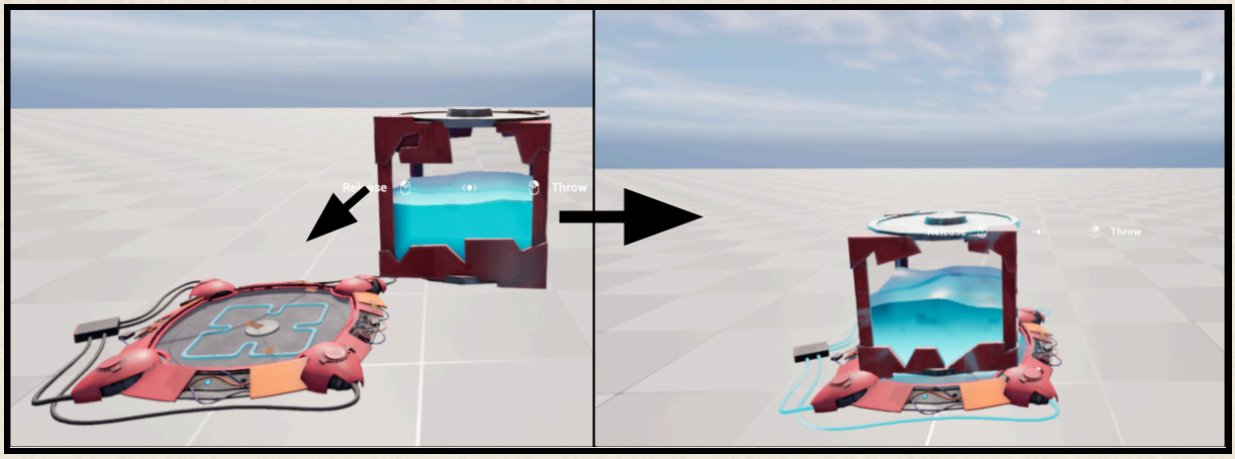
- **Overlap End :**
  When an actor leaves the snap zone, the code verifies if it corresponds to the currently snapped object. If conditions like gravity vector or orientation consistency are not met, snapping is terminated via GrabbableComponent->EndSnap().

Continuous Update & Interpolation

- **Interpolation & Animation:**
  In the update loop, if snapping is active, the object's position is smoothly interpolated towards the ideal snap location (calculated by GetSnapLocation()) using FMath::VInterpTo.

- **Orientation Verification:**
  The IsActorStillOnSnapZone function checks that the object's orientation and gravity direction remain compatible with the snap zone.

- **Cooldown Management:**
  A cooldown mechanism (CurrentSnapCooldown) prevents snapping from happening too frequently, ensuring a stable and fluid user experience.



6.1.10. Crosshair Snapping

The crosshair snapping system is designed to enhance player aiming by automatically aligning the player's crosshair with valid target zones. This is achieved through a dedicated snap component that leverages both a sphere collision detection and raycasting. When the crosshair enters a predefined snapping zone, the system smoothly interpolates the camera's orientation toward the target, ensuring accurate targeting.

Raycasting and Detection

- **Detection Logic:**
  A raycast is performed from the player's camera to determine if the crosshair is within the snap zone. The raycast checks for overlaps with objects that have an active snap component. If the crosshair falls within the snapping sphere, further calculations are performed to determine if snapping should occur.

- **Snap Decision:**
  The method *ShouldSnap* performs a series of checks:

  - It verifies whether snapping is enabled.
  - If snapping is based on the object's size, it compares the scale of the object owning the snap component with the player's scale.
  - Only if the object's scale is below a certain threshold (adjusted relative to the player's scale) will the function return true, allowing snapping to proceed.

Updating the Crosshair

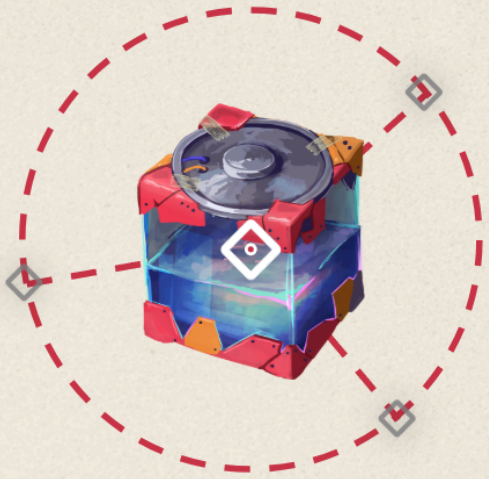
- **Snapping Update Routine:**
  In the player's update function (e.g., *ADSPlayer::UpdateCrosshairSnap*), the following steps occur:

  1. **Validation:**
     The system checks if snapping is enabled and whether the current target still qualifies based on the *ShouldSnap* method. If not, the snapping process is canceled.

  2. **Movement Threshold Check:**
     If the delta rotation (computed using a directional vector change) exceeds a predefined threshold, the snapping is canceled. This prevents abrupt or unwanted snapping when the player makes significant directional changes.

  3. **Interpolation:**
     A smooth interpolation is applied to the camera's rotation. The target rotation is calculated by aligning the camera's forward vector with the direction from the camera to the snap component. An interpolation function gradually adjusts the camera's rotation until it matches the target, at which point snapping is canceled.

- **Rift Integration:**
  In cases the player watches the SnapComponent through TV's, the component's location is converted between TV's spaces to maintain accurate snapping behavior.



6.1.11. Hint System

The Hint System provides players with guidance when they struggle to find a solution in the puzzles. It allows players to request hints at any time, ensuring a smooth and engaging experience without causing frustration.

The hint player works with using 2 components:

- AC_BarkPlayer
- AC_Interactable

The blueprint is composed of the main static mesh visual, and 2 child actor button blueprints.

The button blueprints are a generic button blueprint “BP_GameplayButtonBase” which play a button feedback animation and have an OnPressed delegate for any pressing logic that needs to be implemented.

This is how to use the hint system:

- A BP_HintPlayer is placed within the game environment.
- The panel contains two buttons, each corresponding to a different hint.
    - Designers must give a DA_Bark reference to each of the buttons, as it works by using the AC_BarkPlayer component
- Players can interact with the panel at any time to request a hint.
- Upon clicking a button, the game triggers a bark, which:
    - Plays a voiceline related to the hint. Displays a subtitle containing the hint message.

6.1.12. Level Selector

The Level Selector allows players to choose a specific level from a hub rift. This allows the player to replay levels he once did before and to keep his progression saved.

The BP_LevelSelector is a blueprint placed near each hub portal in the hub area.
It features three physical buttons which are child actors of the BP_LevelSelector (BP_LevelSelectorButton), each corresponding to a specific level.

- A level button is locked until the corresponding level is completed.
- Unlocked buttons are interactive and allow players to revisit completed levels.
- Players can press a button to change the target location of a hub rift.

The linking to a new rift works as following:

- The BP_LevelSelectorButton holds an instance editable reference for the level rift. This can then be selected manually in the editor.
- Once the button is clicked, the “SetOtherRift(Actor)” function from Rift.cpp is called to set the last linked rift “OtherRift” reference to null, and to set this new rift as the new linked rift.

6.1.13. End door

In our system, every TVs is assigned a unique identifier (ID), just as each pressure plate is. For end-of-level TVs that are meant to respond to pressure-plate activation, we reuse the same ID on both the plate and its corresponding TV.

Concretely, when a pressure plate is triggered, the runtime code reads the plate's ID and searches for any TV actors sharing that ID. It then turns on one of the TV's indicator lights to signal that the plate has been activated.

6.1.14. Save System

We store two main pieces of information: the **player's current room**, and the **rooms completed** by the player.
Rooms are all defined by unique IDs (we save the IDs).

Each time the player enters a room, we store the room they entered. Each time they exit a room, we store the fact that the room was completed, along with the new current room.

To manage different game states (mainly the hub) based on the player's progression, we infer these states from the save data.

When the player continues a game, we load the save, determine the saved states, and all elements reacting to the player's progression update accordingly. The player is then teleported to their current room (based on the save).

6.2. Critical points & Risks

6.2.1. Recursivity

Implementing TV recursivity requires improving the basic implementation of TVs.

The base implementation described earlier can be easily made with a TV blueprint having a camera attached to it. This camera is then moved to the corresponding TV and its framebuffer is used as a render target for the TV itself.
The main problem with this implementation is that it does not handle the case where the TV can see itself. A “trivial” fix would be to add another camera to the blueprint and use that second camera when the TV can see itself.
The problem with that is that the recursivity problem would not be solved, and simple situations like having two TVs seeing the same TV would not work either. Plus, having so many cameras to handle and update (even if they are not rendering anything) will impact performances in a meaningful way.

A better implementation would be to have a **TV manager** object with a set amount of cameras. Each frame the TVs will then use the amount of camera they need. This system would let us handle cases where a lot of TVs are present on screen, either via a recurrence or simply via a lot of them being placed in the world. This method even lets us implement a priority system in order to make the TVs closer to the screen able to use more cameras.

6.2.2. Optimizations

TVs are a critical feature of the game, and optimizing their performance is even more crucial. Rendering the contents of a TV is a computationally expensive operation since it involves re-rendering a significant portion of the scene. Therefore, every possible optimization must be implemented to minimize the number of TVs rendered at any given time.

6.2.2.1. Frustum Culling

The primary optimization implemented is *frustum culling*. This technique determines whether an object is within the camera's field of view and avoids rendering it if it is not. This method is applied for both TVs and objects in the scene. To determine if a TV is within the camera's view, the convex volume of the main camera can be computed from its *view-projection matrix*. By performing an overlap test between this volume and the objects in the scene, it becomes possible to identify visible objects. Objects outside the volume are excluded from rendering. This method is also applied to the cameras used for rendering TV views. This allows to greatly reduce the actual field of view of the render camera, thus limiting the amount of objects drawn.

6.2.2.2. Screen Space Optimization

An improvement to the basic frustum culling approach involves considering the proportion of the TV's view that is actually visible on the screen. TVs typically occupy only a small portion of the screen, meaning objects rendered for the TV but outside its visible screen area are unnecessarily drawn. This optimization reduces the camera's field of view to match the screen space occupied by the TV, minimizing wasted rendering effort. This technique has proven particularly effective in reducing the number of TVs drawn in areas with heavy recursion.

6.2.2.3. Upscaling via FSR

After careful analysis using the profiler, the main bottleneck of our game was the rendering time. Lumen and the many render cameras simply take too much time to compute, and it was a very important point to focus on. A great way to reduce the frame time was to implement upscaling via FSR, which will render the game at a lower resolution and then upscale it to the full screen resolution.

6.2.2.4. Other Implemented Optimisations

Additional methods to reduce the number of rendered TVs include:

- **Backface Culling:** TVs that face away from the player can be hidden.
- **Distance Culling**: TVs that are too far from the player or too small on the screen can be deactivated.
- **Quality level for portails**: The closest visible portal will always have a better quality camera with lumen enabled than the others. This lets us prioritize quality for the TV the player is most likely to go in while still having the other TVs being Visible.
- **Resolution for portails**: Using the same principle, the first render camera has a better resolution than the other ones. The resolution of all cameras can be controlled by the in game setting dedicated to TV quality.
- **Fake portail preview**: Some far away portals that needed to be always rendered use a special material made using a cubemap of the target visible area, allowing for a relatively good portal effect from far away. This method is only implemented for some TVs in the hub as it is tedious to set up.

6.2.3. Lights through TVs

Extensive graphical testing has revealed a critical issue: managing lighting effects through TVs.

- When looking through a TV, light-emitting elements appear excessively bright, as if a flash were directed at the player.
- Upon passing through a TV, the recalculation of lighting by Unreal Engine 5 and Lumen becomes noticeable, breaking immersion.

The main lighting issue can be handled by adapting the exposure to match the main camera's exposure, and making sure that the TV closest to the player matches as much as possible the view of the player.
That second part can be achieved by having the first render camera use lumen, and using a post process material that will replace the content of the TV screen closest to the player with the actual camera render target.

6.2.4. Audio according to the object's size

The distance at which a sound can be heard should scale with the size of the emitting object or player. To achieve this, time must be allocated to automate sound attenuation levels based on the emitter's size, ensuring a consistent and immersive audio experience.

6.2.5. Oil Painting

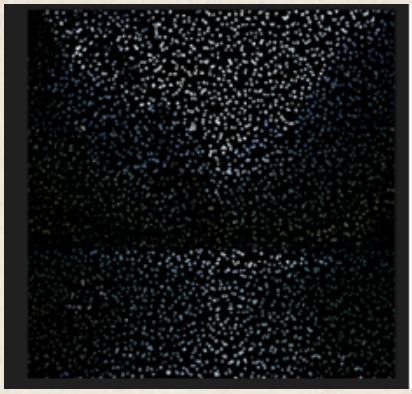
By BERTRAND Louis (Game Artiste 3D)

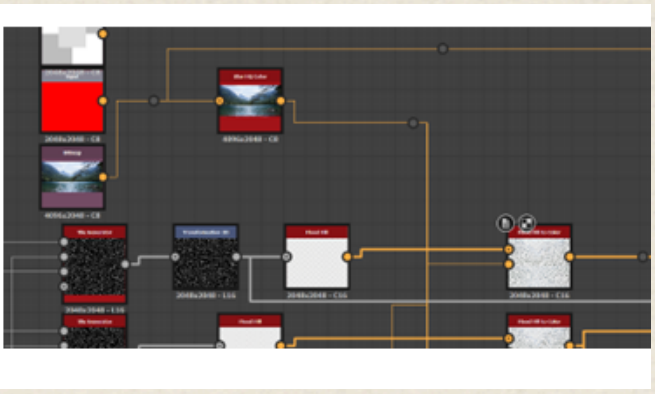
6.2.5.1. Introduction

The goal of this POC is to demonstrate the application of substance designer tools in order to apply an Oil Painting effect to PBR Materials and standard pictures. The wanted result is a Substance Designer graph that can be used to output PBR textures with the effect as well as giving the artists freedom over the effect parameters.

6.2.5.2. Basic Principle

This effect is based upon a Substance Designer feature called Flood Fill, which allows to convert shapes into usable data for various effects. This is then used to sample the color of the given images at each paint brush stroke.

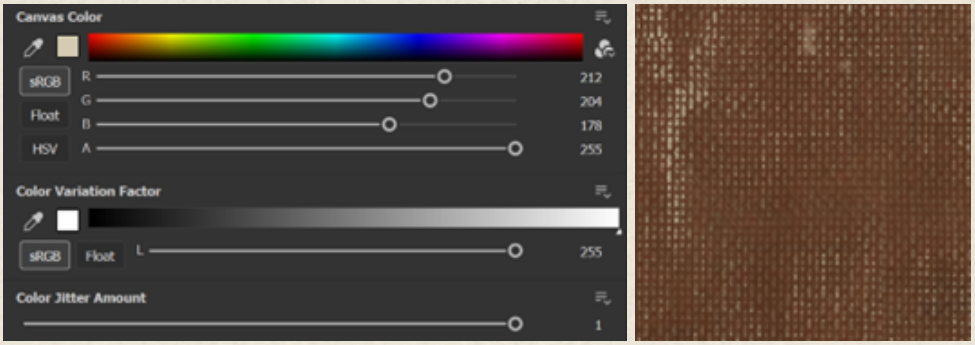




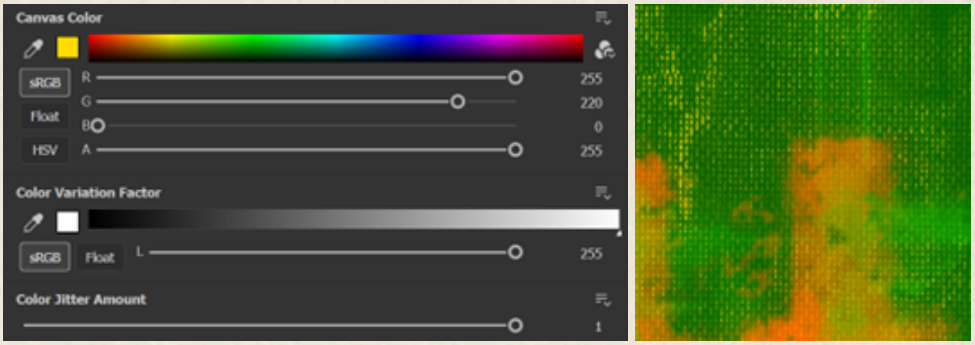
The brush strokes are sampled in a tile generator which are then sent to the flood fill node and used to sample the color of the picture. The effect is repeated multiple times and blended together to create the final image.

6.2.5.3. Applying this effect to PBR Materials

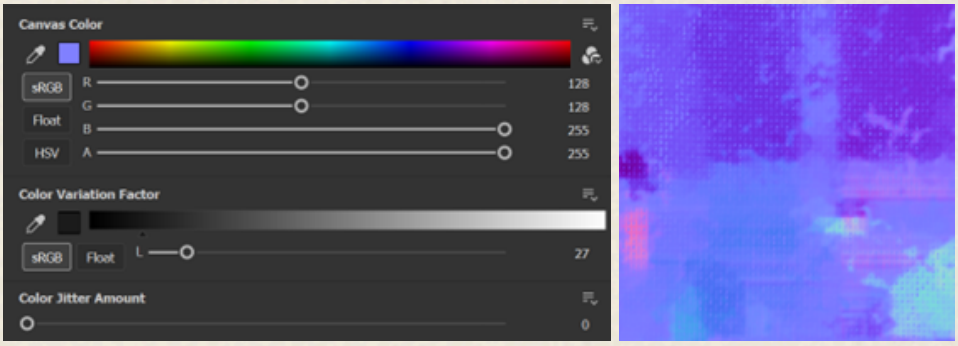
This effect was made to be as easy to use and flexible as possible. This means that ORM and Normal Maps can be generated via this tool. Changing the value of the graph params will come very handy to generate precise values for the data related maps. Here, these parameters are for the Base Color map. The canvas color is beige.



Changing it to orange (255 Red, 200 Green, 0 Blue) makes the map a valid standard ORM Map with this high roughness value where the canvas is visible.



Setting the value to the default normal value can be used to generate the normal map.

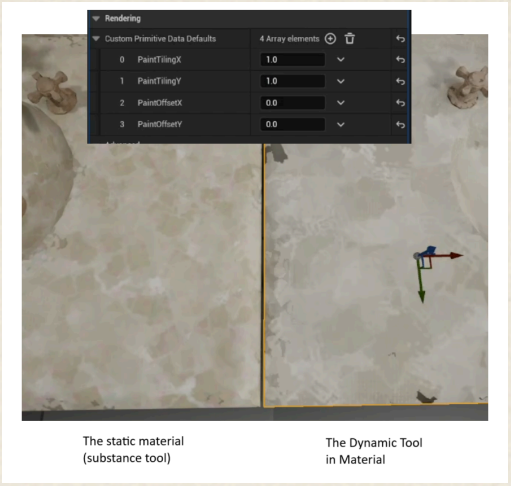


Finally, the tool is used in the PaintPassMaster Subgraph where the artists import their texture maps and apply the shader on them.

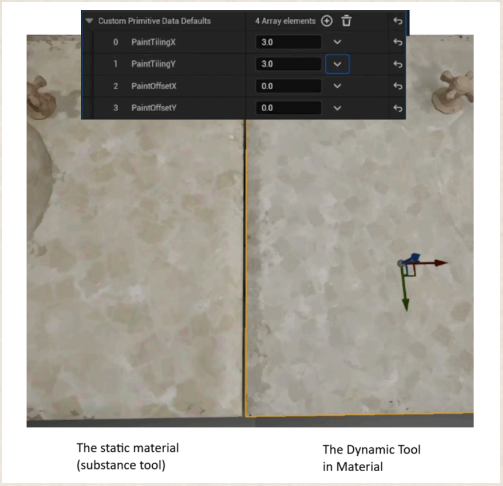
6.2.5.4. Replicating the effect in real-time in Unreal Engine

The previous tool has been repurposed to generate a custom data texture map (UVCV map) holding the sampling UV Coordinates in the R and G channels, the Canvas Mask for the effect in the B Channel, and the color value variation mask in the alpha channel. These masks can be separated into different textures or included in other data structures, but we choose this 4-channel texture mask to optimize the amount of loaded textures in the GPU and ease the sampling methods.

Using “Custom Primitive Data” on the current mesh or instance parameters, the UVCV map is tiled and remapped according to the parameters. Allowing this per mesh can help with the scale of the effect and could even be used as a way to change the “Level of Detail” of the texture. This is useful for distant objects.

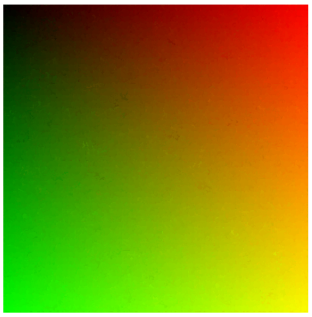


The static material (substance tool) — The Dynamic Tool in Material

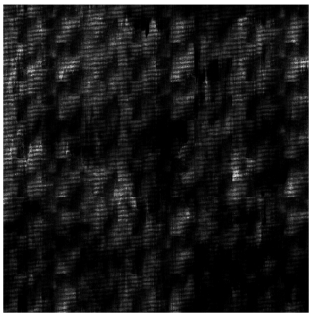


The static material (substance tool) — The Dynamic Tool in Material

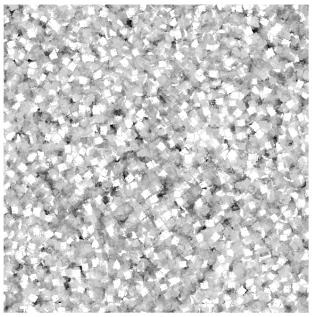
The UVCV map (UV Canvas Value) is packed as followed:



UV (R+G)



Canvas (B Channel)

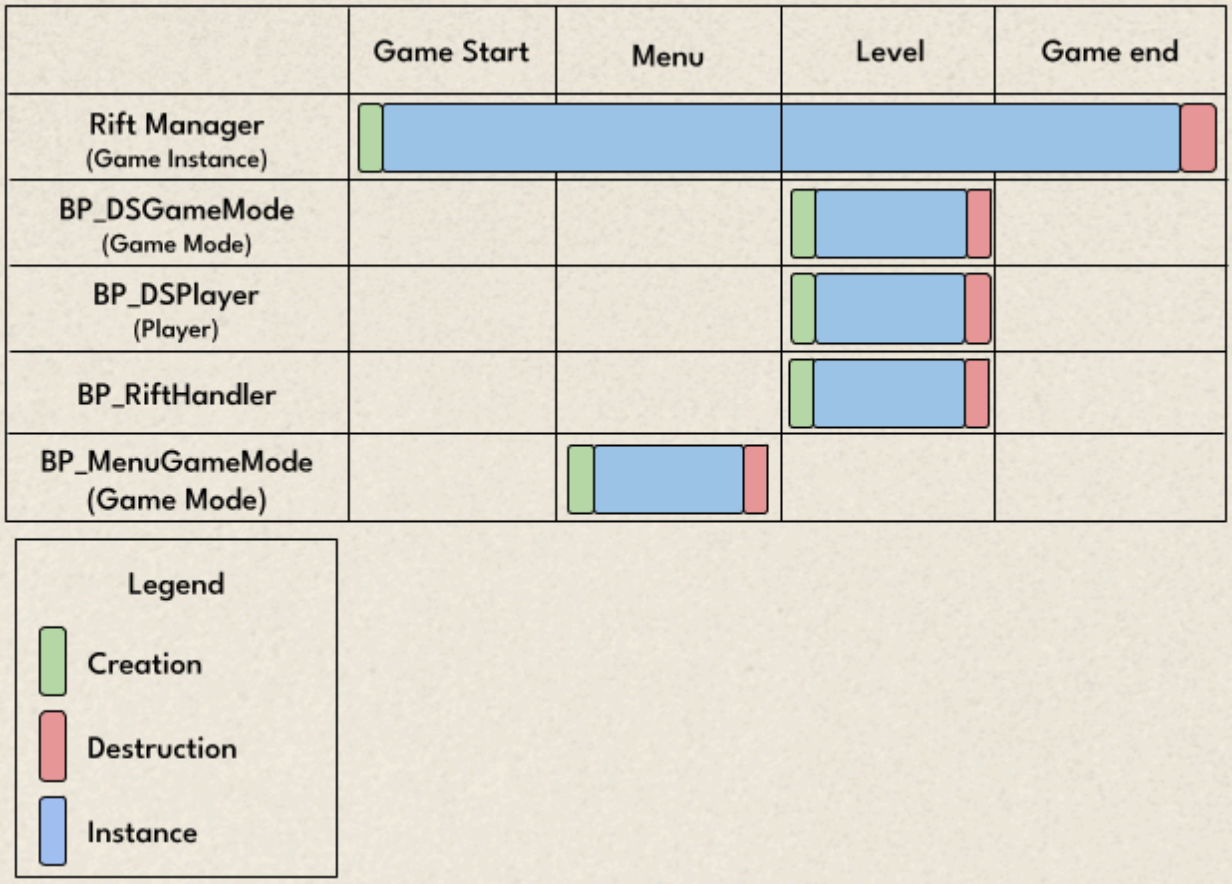


Value Variation (A Channel)

The UVs are baked with the paintbrush strokes effect.

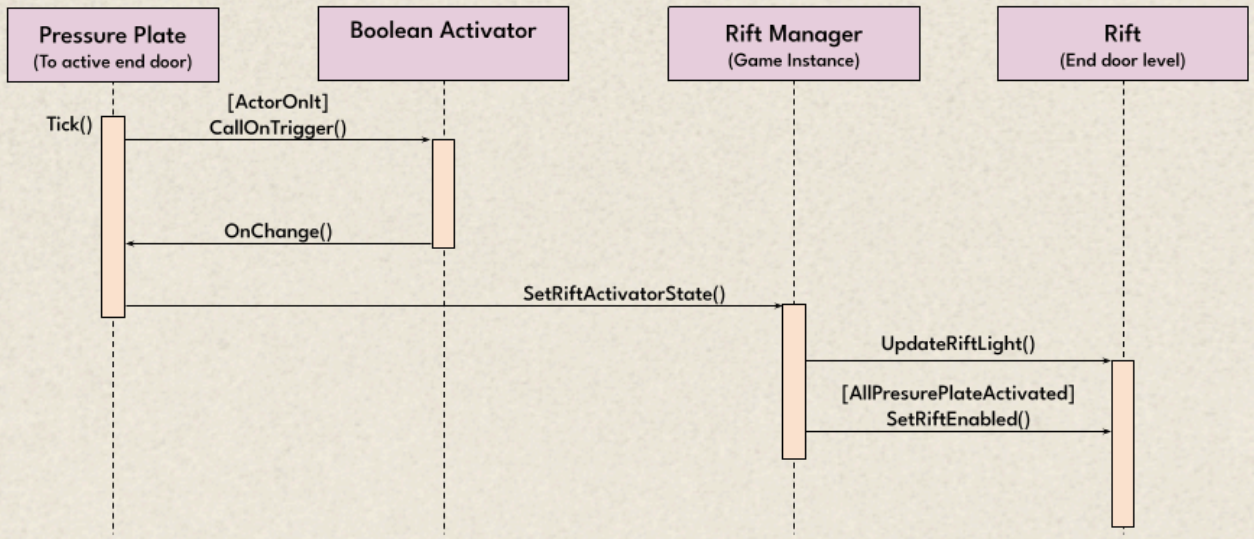
7. Diagram

7.1. Static modules diagram



7.2. UML Interaction Diagram

UML Interaction Diagram for level game loop (pressure plate to TV end door opening)



7.3. UML Class Diagram

[UML Class Diagram link (lucidchart)]

During the development of our TV system, we opted for a Rift-based approach, which is why the term "Rift" appears in our code.

8. Time estimation

8.1. Gold

Title	Risk	Priority	Estimation (Day)
Portal Optimisation	Medium	Low	7
Debug Cube / Pressure Plate	High	High	21
Debug Save system	Medium	High	21
Debug TV	High	High	21
Debug UI	Medium	High	14

8.2. Beta

Title	Risk	Priority	Estimation (Day)
Portal Optimisation	High	High	20
Grab Debug	High	High	20
Save system	Low	Medium	7
Music and Sound Integration	Low	High	7
Debug	Medium	Medium	14
UI	Low	High	7
Level Selector	Medium	Medium	7

8.3. Alpha

Title	Risk	Priority	Estimation (Day)
Portal Optimisation	High	Medium	20
Grab	Medium	High	20
Lights Through TVs	High	High	14
Music and Sound Integration	Low	High	2
Sublevel system	Low	Medium	2
UI	Low	High	7
Bark System	Medium	Medium	5

8.4. 3C

Title	Risk	Priority	Estimation (Day)
Portal	High	High	5
Portal Size	High	High	2
Portal Gravity	High	High	2
Grab	Medium	High	2
Movement Character	Medium	High	3
Wwise Implementation	Low	High	0.5
Portal Optimisation	High	Medium	5
Portal Impulse	Medium	Medium	1
Button	Low	Medium	1
Preasure Plate	Low	Medium	1
Portal Recursivity	High	Low	5

9. References and sources

1. **TV System**

A tutorial was used to validate the logic of the TV system. While the structure is custom-built in C++, the tutorial ensured the approach was robust and aligned with best practices, focusing on spatial logic and rendering principles.
How to Create TVs in Unreal Engine by Dev Squared

2. **FSR**

A guide was followed to implement AMD FidelityFX Super Resolution 3.1.3 in Unreal Engine. The integration process involved setting up the plugin and adjusting the settings for optimal performance and quality, ensuring compatibility with Unreal Engine's rendering pipeline.
AMD FilelityFX Super Resolution 3.1.3 Unreal Engine plugin guide

3. **DLSS**

NVIDIA's DLSS (Deep Learning Super Sampling) was utilized to enhance rendering performance in Unreal Engine. Following NVIDIA's official developer documentation, the integration leveraged AI-based upscaling to boost frame rates without sacrificing visual fidelity, ensuring smooth performance in resource-intensive scenes.
NVIDIA DLSS | NVIDIA Developer

10. Norms

10.1. Asset Name

Utilization of the recommended asset naming conventions: documentation.

```
[AssetTypePrefix]_[AssetName]_[Descriptor]_[OptionalVariantLetterOrNumber]
```

Ex: M_Player

10.2. Submit

```
[Job] Descriptor
```

Ex: [GP] TV feature

10.3. C++

Utilization of the coding standard conventions: documentation

Certain files, such as *Player* and *GameMode*, cannot follow Unreal's naming convention because files and classes with the same names already exist. Therefore, we are prefixing these names with "DS" (for Dimension Shift), ex: *DSPlayer*, and *DSGameMode*.