NCCA

MSc Computer Animation and Visual Effects
August 2024

Unreal Engine USD Attribute Toolset

by

Jack Purkiss

Abstract

Proto Imaging is a studio that have been producing 3D animated video for over twenty years. Predominantly using Maya, Vray and a range of compositing and post-production software, they have begun exploring the area of real-time 3D animation. Using the Universal Scene Description (USD) developed by Pixar, this project aims to produce a set of pipeline tools to assist Proto Imaging in a workflow between Maya and Unreal Engine. The project is divided into three sections: the USD Attribute Function Library, which allows direct access to and manipulation of USD attributes; interactive tools for controlling USD animations within Unreal Engine; and functionality to automatically replace USD shaders with native Unreal Engine materials.

Acknowledgements

I would first like to thank James Malloch at Proto Imaging for allowing me to collaborate with him. Without his continued support, this project would not have been possible. Extending my thanks, I am incredibly grateful for Jon Macey for his frequent assistance throughout this project and the rest of year, as well as the rest of the NCCA staff.

Furthermore, I thank my fellow CAVE people, my friends, family and Laura for their encouragement (and accommodation) on this project, and degree as a whole.

Contents

1 Introduction **1**

2 Background **2**

2.1 Data Transfer Approaches 2

2.1.1 FBX 2

2.1.2 Alembic 2

2.1.3 USD 2

2.1.4 USD vs FBX 3

2.2 USD in Unreal Engine 4

2.2.1 USD Plugin 4

2.2.2 Unreal Wrapper Classes 5

2.2.3 Runtime access 5

2.3 Proto Imaging Requirements 6

2.3.1 Custom attribute access 6

2.3.2 USD animation playback controls 6

2.3.3 USD shader swap 6

3 Solution **7**

3.1 USD Attribute Blueprint Function Library 7

3.1.1 Exporting custom attributes 7

3.1.2 Blueprint accessible functions 8

3.1.3 Template functions 8

3.1.4 Value retrieval 9

3.1.5 Vector Support 9

3.2 USD Interaction 9

3.2.1 Level Sequence Player Manager 10

3.2.2 Button blueprints 10

3.2.3 Widget Button Function Library 12

3.2.4 Example Blueprint 13

3.3 USD Editor Tools 13

3.3.1 Camera Information and duplication 15

3.3.2 Attribute Export to sequence 15

3.3.3 Material Swap 16

3.3.4 Disable manual focus 16

4 Results **17**

4.1 USD Attribute Uses 17

4.1.1 Object Locations 17

4.1.2 Particle system attributes 17

4.1.3 Animated materials 18
4.2 Material swap 18
4.3 Interactive HUD 19
4.4 Testing and Design changes 19
4.4.1 Testing 19
4.4.2 Design Changes 19

5 Conclusion **20**

Appendix A Code **22**

Appendix B Blueprints **26**

List of Figures

2.1 USD Stage Editor 4
2.2 USD Generated level sequence 5

3.1 Maya Custom Attribute Exporter GUI 7
3.2 Get Translation attribute demo 8
3.3 Level Sequence Player Manager Flowchart 10
3.4 Parameters to set up Play Frame Range Button 11
3.5 Play Frame Range Function Node 12
3.6 USD Editor Tools Window 14
3.7 Camera Main Camera Number Attribute 14

4.1 Icon placed with translate vector attribute 17
4.2 Particle and Material Attributes being assigned 18
4.3 Scanner with swapped Unreal Material using custom Universal Scene Description (USD) attributes for colours and particle rate 18

B.1 Play Frame Range Button Construct 26
B.2 Play Frame Range Button Blueprint 27
B.3 Stop Sequences Button 27
B.4 Pause/Play Button 27
B.5 Play Frame Range Blueprint Function 1/2 28
B.6 Play Frame Range Blueprint Function 2/2 28
B.7 Find Camera By String Function 28
B.8 Set Sequence Player Function 29
B.9 Get Sequence Player Function 29
B.10 Stop Current Sequence Function 29
B.11 Is Current Sequence Function 29
B.12 Example Custom Loop Number Blueprint 30
B.13 Example Infinite Loop Blueprint 30
B.14 Example Stop Sequence Blueprint 30
B.15 Proto Imaging Example HUD 31

List of Algorithms

1 GetSdfPathWithName 9

List of Tables

2.1 Comparison between FBX and USD formats 3

List of Acronyms

DCC Digital Content Creation. 1, 2

FBX FilmBox. 1, 2, 3

USD Universal Scene Description. v, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20

1 Introduction

For over two decades, Proto Imaging (2024) have produced linear pre-rendered animated CGI videos for a diverse client base, from Dyson to Honda and leading aerospace companies. Utilising 'traditional' CGI workflows, they use predominantly Maya, Vray, and a range of compositing and post-production software to create pre-rendered animated CGI videos. In recent years, they have begun looking to enter the real-time 3D animation space, with a view to offer clients standalone apps for exhibitions. Unreal Engine (2024) is currently one of the leading Digital Content Creation (DCC) tools in providing real time 3D graphics, frequently being used within small and large scale productions due to its high quality rendering in a real-time context.

With the end goal of producing a real-time interactive CG product, this project aims to provide a collection of pipeline tools to assist Proto Imaging in the development of such a product. Due to Unreal Engine's limited 3D modelling, rigging and animating toolsets, and Proto's existing Maya skillset, a pipeline that enables the majority of the product to be produced in Maya is essential. After their experimentation with different data transfer methods, Proto determined that the most viable format for the immediate future is USD, as opposed to alternatives such as FilmBox (FBX) and Alembic. The benefits and fallbacks of these approaches will be explored in the next chapter. In recent years, USD has been growing in popularity. Support for it in different DCC's is in progress, with developments still frequently being made. In this project version 5.4.2 of Unreal Engine is being used, which includes a USD Importer plugin that is still in beta. Interaction with the USD file and assets are therefore not fully supported within Unreal Engine, so this project aims to bridge some of these issues to assist in a more stable Maya to Unreal Engine pipeline.

On import, Unreal Engine converts USD prims into the appropriate Unreal assets to be used, such as static mesh's and cameras. All animation from a USD is included in its generated level sequence. All of these assets are stored transiently in the project, meaning that within each session all of their references change unless imported completely into the project as Unreal assets. The process of working with a client means that animation and ideas change frequently during a project. With the major animation work being created outside of Unreal Engine, Proto requires that the production process must be flexible with this thus removing the option to import everything natively into the Unreal project. Additionally, to use USD in a packaged Unreal Engine project build the USD stage must be loaded in runtime. The set of tools built as part of this project seek to provide access to USD attributes in the editor and at runtime, as well as providing interaction with cameras, and animated frame ranges from both user and USD generated sequences.

2 Background

2.1 Data Transfer Approaches

2.1.1 FBX

FBX (FilmBox) was originally developed by Kaydara before it was acquired by Autodesk, and has since become one of the primary 3D data transfer methods used throughout most DCC tools. These files can store a wide range of data types, such as geometry, animation, textures, lighting and rigging in both binary and ASCII encoding (Ardolino et al. 2014). With a comprehensive set of features for animation and rigging, including keyframes, inverse kinematics and blend shapes, FBX is often the preferred approach in data exchange for skeletal animation between various 3D applications.

2.1.2 Alembic

Developed by Sony Pictures Imageworks and Industrial Light and Magic, alembic is an open-source software framework for efficient storage and sharing of 3D geometry and animation (SPI, ILM 2024). Alembic works by baking animated scenes into its vertices, capturing the end result of animations and simulations without having to store their underlying procedural networks or dependency graphs. This allows Alembic files to be used across multiple DCC's. However, it can become incredibly inefficient; by storing every vertex at every frame the result can be a very heavy file.

Additionally, Alembic lacks shader support required for the desired pipeline. With these issues present, it is not an appropriate data transfer approach for this project.

2.1.3 USD

Universal Scene Description (USD) is an open-source framework developed by Pixar to optimise the interchange of 3D graphics (Pixar 2023). Contrary to the previously discussed approaches, USD aims to encapsulate data for each of the areas in the pipeline. Typically, many of the cooperating applications in the pipeline have their own form of scene description designed for their specific needs. In USD, data is organised into a hierarchy of Prims. Each prim then contains child prims, but also Attribute and Relationship properties. Attributes are values of a given type, with the option to have associated time samples to provide animation to them. Example attributes would be transformations, with the option to add custom attributes as well. Relationships are multi-target "pointers" to other objects within the USD hierarchy. A typical example of this could be a material prim, with the mesh's using that material pointing to it with a relationship.

There are many benefits to using USD over other methods. Unlike other packages, a 'stage' is set up, where any number of assets can be assembled and organised into sets, scenes and shots to be used within any application. This stage is set up with composition arcs of operators (subLayers, inherits, varientSets, references, payloads, specialises). The sublayer operator provides opportunities for multiple artists to collaborate on the same assets and scenes, by allowing them to

work within their own layer. Combining this with the support across multiple applications provides substantial pipeline opportunities.

2.1.4 USD vs FBX

Feature	**FBX**	**USD**
Format	Proprietary format owned by Autodesk	Open-source format originally developed by Pixar
Age	Older but well established in the industry	Newer format that is still evolving
Referenceable	No, all scene components must be imported, requiring careful handling of animation data	Yes, scenes can be quickly updated within Unreal Engine as the USD is updated
Animation - Hierarchical	No, unsupported for Unreal Engine. All objects are combined, requiring complex process to rebuild hierarchies	Yes, as a scene descriptor all of the hierarchies remain intact
Animation (Cameras)	No, camera animation must be imported as animation curves and applied to the camera actor, with its attributes set manually	Yes, camera animation and many camera attributes are exported directly
Animation (Skeletal)	Yes skeletal animation is the primary animation format supported by FBX	Yes, skeletal animation is also established in USD
Shaders	Legacy shading models such as Phong, Blinn, and Lambert. These have very rudimentary properties, and Phong is not native to Unreal Engine	Modern PBR shaders, including usdPreviewSurface and MaterialX.
Bitmap Textures	Supported with limited attributes	Supported with extensive attributes, including individual subchannels, such as the green channel only of a bitmap
Full Scene Export	No, rebuilding a complex Maya scene is convoluted	Yes, as the main concept behind USD the entire scene can be exported in a single file

Table 2.1: Comparison between FBX and USD formats

In the context of transferring individual assets between applications, FBX has been the standard for years, and is well supported for this reason. Particularly with animated characters, it is effective and has many functional pipelines from Maya to Unreal Engine. However, Proto Imaging requires a pipeline where an entire scene within Maya can be set up with hard body animation, hierarchical relationships, locators and cameras, and continue working on it from within Unreal Engine. While FBX does offer some hierarchical features, these are not supported within Unreal Engine. Individual assets would have to be exported out and reassembled with the game engine which is not sustainable, particularly with frequent updates to the project.

These areas are precisely what USD is best for. With its composition arcs, the hierarchical ap-

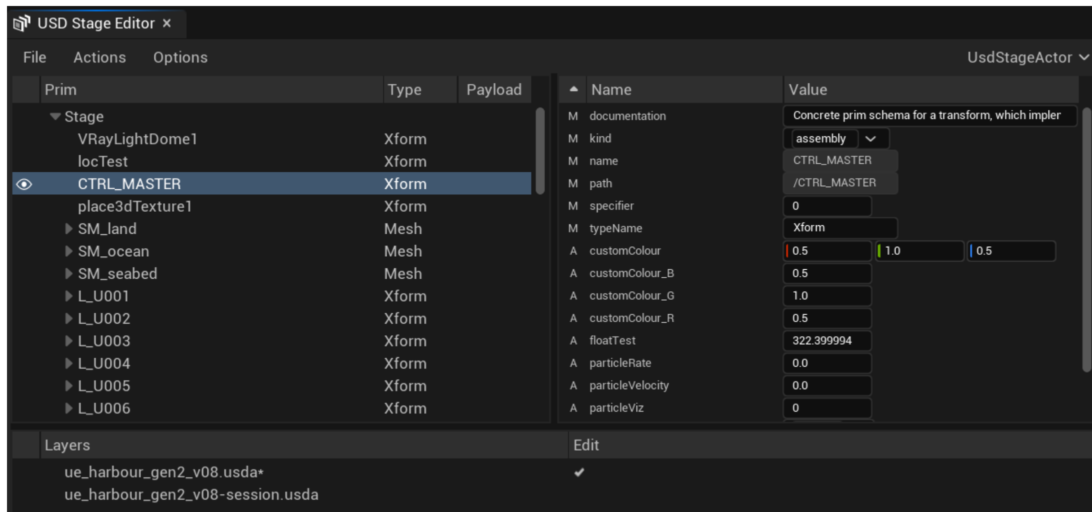

Figure 2.1: USD Stage Editor

proach combines perfectly within USD. The hard body animation is able to translate as expected, and additional areas such as shaders and cameras can be transferred to. With all the data being stored within the USD stage, it can be updated within Maya and worked on within Unreal Engine immediately, without any extra steps, in theory allowing a streamlined approach to production. Additionally, the options for custom attributes and metadata can provide a flexibility to exchange additional data that can be customised between projects.

However, with USD having been open-sourced since 2016, it is still in the process of being adopted across all DCC's. Houdini's Solaris and NVidia's Omniverse are areas that provide USD interaction properly, although Unreal Engine remains in its beta stage.

2.2 USD in Unreal Engine

2.2.1 USD Plugin

As of Unreal Engine 5.4.2, to obtain access to USD functionality, the USD Importer plugin must be installed. Within the Unreal editor, this provides accesses to the USD Stage Editor and UsdStageActor. As of this version, the USD Importer is a beta feature, so caution is encouraged when shipping with it. Version 5.5 separates the content of the plugin into an importer plugin and a runtime plugin, but at the time of writing has not been fully released. The Stage Editor, shown in Figure 2.1, is the visual interface working natively with the USD data (Epic Games 2024b). Here, the prims are displayed within the hierarchy, with non destructive attribute available, loading and unloading content with Payloads, textures and materials and more. Any changes made to the USD, either from the Stage Editor directly or through the UsdStageActor and generated assets, can then be saved back onto the USD file directly.

The UsdStageActor is present within the level, and contains the Unreal representation of the prims from the USD. When using USD, a UsdAssetCache is generated, which is where the prims are converted into their UObject equivalents. Under each child of the UsdStageActor hierarchy contains the corresponding component, such as a static mesh, or a camera. An important point to note here, is that all of these assets are stored transiently, meaning that between sessions their object references

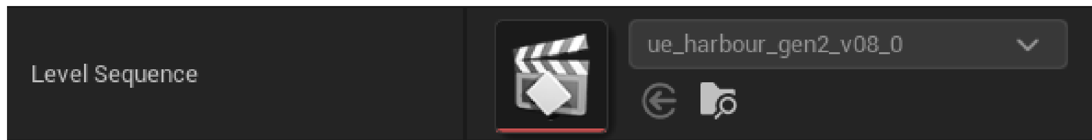

Figure 2.2: USD Generated level sequence

will change. This can lead to challenges when using the objects in blueprints, for example one might set up some logic with a camera. If this camera was used regularly in the project, the object reference would remain and will work the same each time. Using one of the USD generated assets will result in an error, due to this reference change.

Any animated features within the USD file will be present in a transient level sequence under the details of the UsdStageActor, seen in Figure 2.2. Similar to the other generated assets, any logic that references this will be lost between sessions. Changes or additions made to this level sequence can be saved back onto the USD file as timesampled data in the same way as other changed data. The animations on the USD are all set with time samples relative to the Time track on the file. Therefore, by putting a UsdStageActor onto another level sequence and setting keyframes for the time track, the animations within the USD file will play accordingly.

2.2.2 Unreal Wrapper Classes

In order to access the USD features when writing an Unreal Engine C++ module, Unreal provides the UnrealUSDWrapper module within the USD Importer plugin as a layer between the Unreal and Pixar API's. Many of the USD classes have a corresponding wrapper class used in the UE namespace, that provide access to most, but not all, of their features. An example use case of this might be starting with a UsdStageActor to access its USD Stage. With the stage object, one can access a prim, stored as a UE::UsdPrim. From here, access an attribute from the prim, stored as a UE::UsdAttribute. In both of these cases, most of the features from the original pixar classes are still available. The value from a UsdAttribute is stored as a UE::VtValue, accessing the pxr::VtValue class. Primitive types such as float, double and int can be accessed directly from here, however types such as vectors require additional conversions to be accessed in Unreal.

2.2.3 Runtime access

Unreal Engine supports the loading of USD files at runtime with the Set Root Layer blueprint node. This works similarly to the approach in the editor, by creating the required assets and components as expected. Direct access to generated components remains challenging due to the object references not being directly accessible at this stage.

Additionally for this to work in a packaged project build the Project.Target.cs file in the project's source folder must be adjusted with "`GlobalDefinitions.Add("FORCE_ANSI_ALLOCATOR=1");`". However, this is not achievable from an Unreal Engine version available from the Epic Games launcher, and requires the engine to be built from source to be adjusted. To use runtime features in C++, precautions must be taken in the module's build file and throughout the header and source files also. Any code that may be affected must be wrapped with an "`#if USE_USD_SDK`" macro to avoid compilation errors. Appropriate conditional statements must be added to the build files to ensure the Usd SDK is

only enabled at suitable times, which is highly dependent on the operating system being used.

This is relative to Unreal Engine version 5.4.2, with these features all within the USD Importer plugin. Version 5.5 is currently in development at the time of writing, and separates some features out of the USD Importer plugin and into a separate runtime USDCore plugin. Therefore, the mentioned runtime approaches are likely to change with the upcoming updates.

2.3 Proto Imaging Requirements

Having determined that USD is the data transfer approach of choice due to the quick file updates between applications, there are a few issues with it that require addressing. This project is aiming to bridge the following areas to fulfil Proto Imaging's production needs.

2.3.1 Custom attribute access

All animation edits throughout production are created in a Maya source file. This includes hard body animation, camera animation (with camera cuts) and vector positions/rotations. These could be used for a range of tasks, including setting positions of icons in a scene, particle emitters, particle emission rates, curves and any other potential custom use cases. This requires a pipeline to be able to export a custom attribute for an object out of Maya, and an approach to be able to read the data from the USD in Unreal Engine.

2.3.2 USD animation playback controls

Proto's vision is to create many thousands of frames of animation within Maya as a main timeline containing everything. Once in Unreal, the specification is to produce an interactive app that can draw on specific frame ranges to playback at will. The camera cuts could number up to 80+, with additional camera animation and static cameras also present outside of these main camera cuts. Finding all of the frame ranges within the editor is therefore important, with options available to access the cameras and attributes. Tools should also be available to interact with the desired frame ranges smoothly, with flexibility for different playback scenarios. At times, imported cameras can have issues with focus settings coming off the USD. An additional useful feature for Proto would involve automatically disabling these auto assigned properties from Unreal, for increased efficiency in production.

2.3.3 USD shader swap

USD shaders are being treated by Proto as read-only as the USD will be periodically overwritten as updates are made. In Unreal Engine, a material can be added to a generated component and when saved, will add this shader as a relationship on the USD. However, as the USD files are being overwritten often, the material will not persist when the USD is reloaded. While reasonable shaders created in Maya can be transferred on the USD, Proto have many cases where more complex Unreal materials may be required. Potentially hundreds of materials could be involved with a project, so a system is required where complex Unreal materials can be set up with names matching the materials on the USD to automatically swap them round. This should work quickly, allowing a quick swap on any new USD update. In the case where the project is production ready, the materials can be saved back onto the USD to run natively.

3 Solution

3.1 USD Attribute Blueprint Function Library

3.1.1 Exporting custom attributes

Before looking at the problem of reading attributes in Unreal Engine, they must first be exported out of Maya. With the regular USD exporter for Maya, the default attributes on an object will be exported to its corresponding prim, however any user defined attributes will not be recognised, resulting in them being ignored by the exporter. For them to be recognised, a string attribute must be added to the Maya object with the name ”`USD_UserExportedAttributesJson`”. This attribute must then contain a json as a string, in the format:

```
{"usdAttrName": attr, "usdAttrType": attr_type, "usdAttrValue" : attr_value}
```

with attr `attr` being the custom attribute name, and then the corresponding type and value of the attribute. To make this practical for an artist to use, a python script was written with pyside, allowing the user to select an object from the outliner and select any user defined attributes they wish to export. This then creates the JSON string attribute with all of the custom attributes to be recognised by the USD exporter.

From the request of Proto Imaging, this tool was kept as a script to be pasted into the script editor in Maya (Code A.1). Proto frequently use different versions of Maya, across multiple machines. Therefore, keeping the tool as lightweight as possible is valuable, with an aim to keep potential maintenance to a minimum. To write a custom exporter or a more formal plugin could present higher risk in the tool not working between versions, and may require more time spent on the installation. With the script run, the user can select which attributes they wish to choose from their current selected object from the pyside GUI, seen in Figure 3.1.

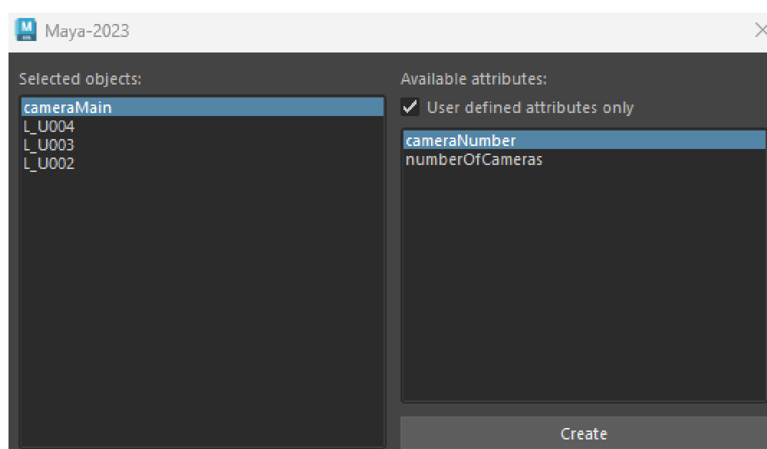

Figure 3.1: Maya Custom Attribute Exporter GUI

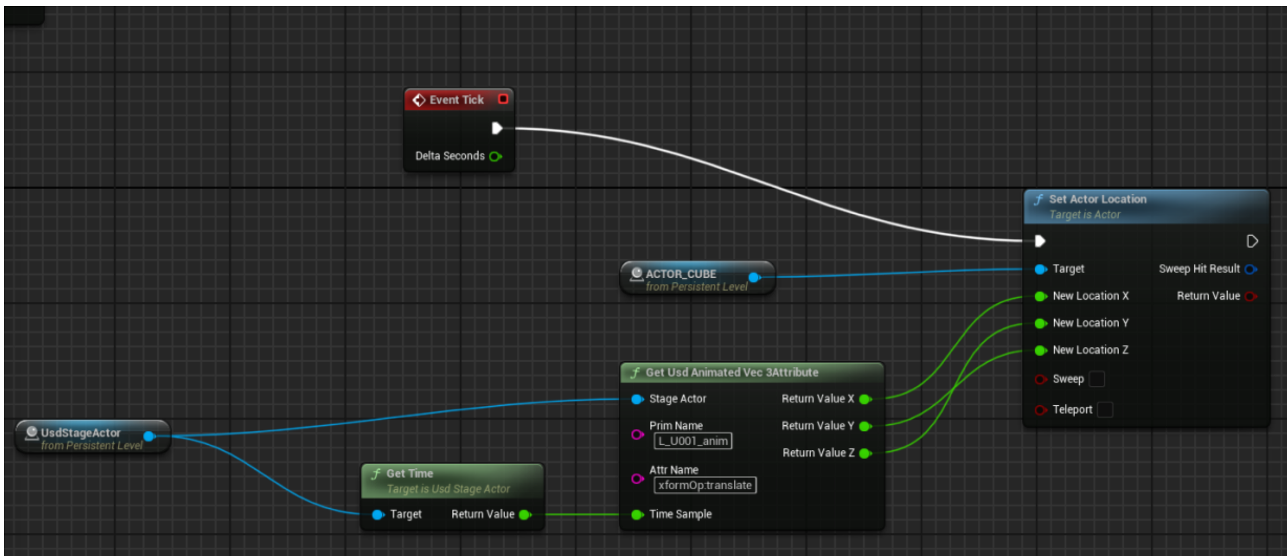

Figure 3.2: Get Translation attribute demo

3.1.2 Blueprint accessible functions

With the aim to have access to the USD prim attributes, the use cases for this scenario must first be analysed. The key examples given in the brief by Proto Imaging mainly involve finding the location of an object, or the value of a custom attribute at a given time. As this needs to be customisable to the situation and able to work in a running product, this functionality must be blueprint accessible to be available in runtime. One of the examples involves connecting an icon to a position above an object in the scene. To do this, one would create a widget blueprint of the desired icon and in the blueprint, retrieve the location of the desired object and set the widget location to that. However the issue with this remains that these object references are lost between sessions. A similar approach to the `FindCameraByString` function could be used here, but this does not address the use case for a custom attribute and is only applicable to an objects location.

To solve this, a set of functions have been written for the most common types to be used, with future opportunities to expand into more types if needed. In this case, GetUsdAttribute functions are written for 'Float', 'Int' and 'Double', with corresponding functions for 'Vec3's of each of these types. Additionally, each of these also have a matching function with a timesample input to find the attribute value when animated. The function has three parameters, with the additional timesample parameter for the animated function. First, the UsdStageActor object is referenced which allows access to the USD Stage. Then, the string name of the desired prim and attribute name are used to be able to find the value.

3.1.3 Template functions

Once the blueprint function is called for one of the given non vector type, it calls an internal template function passing the given type as typename `T`. The functionality is the same for both the animated and regular attribute retrieval, with the only difference being when the value is retrieved the animated function provides a timesample as an additional argument. This can be seen in Code A.2.

3.1.4 Value retrieval

To find the attribute object from the arguments given, a Depth-First Search is applied on the USD Stage hierarchy to find the SdfPath for the provided prim, as seen below:

Algorithm 1 GetSdfPathWithName

Require: CurrentPrim, TargetName, PathResult

```
1: function GetSdfPathWithName(CurrentPrim, TargetName, PathResult)
2: if CurrentNode.Name = TargetName then
3:     PathResult ← CurrentPrim.Path
4:     return
5: end if
6: for Child in CurrentPrim.Children do
7:     GetSdfPathWithName(Child, TargetName, PathResult)
8:     if PathResult is not empty then
9:         return
10:    end if
11: end for
```

This path is required to find the Prim object which will be used to find the desired attribute. With the prim and attribute found, value of the attribute is accessed with `bool bSuccess = Attr.Get(Value);` stored as a UE::VtValue object. The contained value still cannot be accessed from here, so the `ExtractAttributeValue` function is called with the provided typename. The pxr::VtValue is retrieved from the UE::VtValue, and checked to ensure that it is holding the correct type. Once confirmed, the value is finally returned for the appropriate type, as seen in Code A.3.

3.1.5 Vector Support

The approach is slightly altered when accessing a vector attribute. In this case, only the double, float and int Vec3's are supported. The pixar Vt types for these are `GfVec3d`, `GfVec3f` and `GfVec3i`, which don't directly work in a return statement the same way as the primitive types. In this case, an extra step is taken to check whether the held value is the specified vector type, and an additional `ConvertUsdVectorToFVector` template function is called with the desired vector type. By default, FVector's hold doubles, so for simplicity the function returns doubles for all cases. For Proto's use case, this is suitable, however could be adjusted for greater accuracy in the future.

3.2 USD Interaction

To provide interaction with the USD level sequence, and any other level sequence, a library of widget blueprints, and blueprint functions has been included in the plugin. These serve to provide functionality to play a given frame range from a level sequence, assign start and end cameras and determine whether animations are looped. With Unreal Engine being traditionally used for game development, most functionality works outside of a linear timeline with predominantly event based approaches used. Working in video, Proto Imaging are used to a linear approach to working. Due to this, the aim here is to adapt this linear approach to fit into Unreal Engine. with the vision to have one main

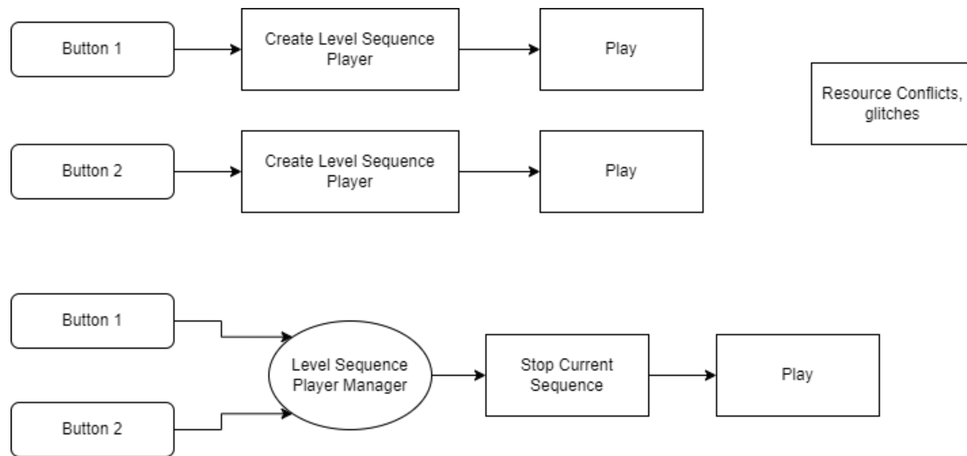

Figure 3.3: Level Sequence Player Manager Flowchart

timeline that contains all animation, this set of widgets and tools allows frame ranges of this animation to playback seamlessly, providing interaction across a linear timeline.

3.2.1 Level Sequence Player Manager

Typically to play a level sequence from a blueprint, a `Create Level Sequence Player` node is used with the desired level sequence object as an input. Then, a `Play` node is used, or other playback controls, to trigger the sequence.

However this is problematic in situations where sequences are called from different blueprints or multiple times. In these cases, a new level sequence player is generated every time, resulting in multiple instances of the same sequence, or different sequences trying to use the same asset. With these overlaps, unwanted affects can be cause, like glitches in playback and general resource conflicts.

To address this issue, a `Level Sequence Player Manager` Blueprint has been designed (Figure 3.3). This actor blueprint holds a Level Sequence Player as a variable. Alongside this, is the `WidgetButtonFunctionLibrary` which contains a set of functions that can interact with this manager blueprint.

3.2.2 Button blueprints

To assist in the USD interactivity, a set of five Widget blueprint classes have been created, each being a button with different functionalities. With the installed plugin, the user can then use these button widgets on a canvas, or anywhere it may be needed. Each button has a style variable, which allows the user to customise the style to their liking which is assigned in the pre construct phase of the blueprint. To allow the user to have full customisation of a text box, there has not been a pre built in text box, but instead an empty slot allowing the user to add one themselves, or anything else they may prefer. These buttons were created first in the design process, with their functionality then being encapsulated into the function library in the section 3.2.3 to provide more flexibility in project design. Each button has a set of variables that can be adjusted in the widget blueprint designer, which are used to determine the actions of the button.

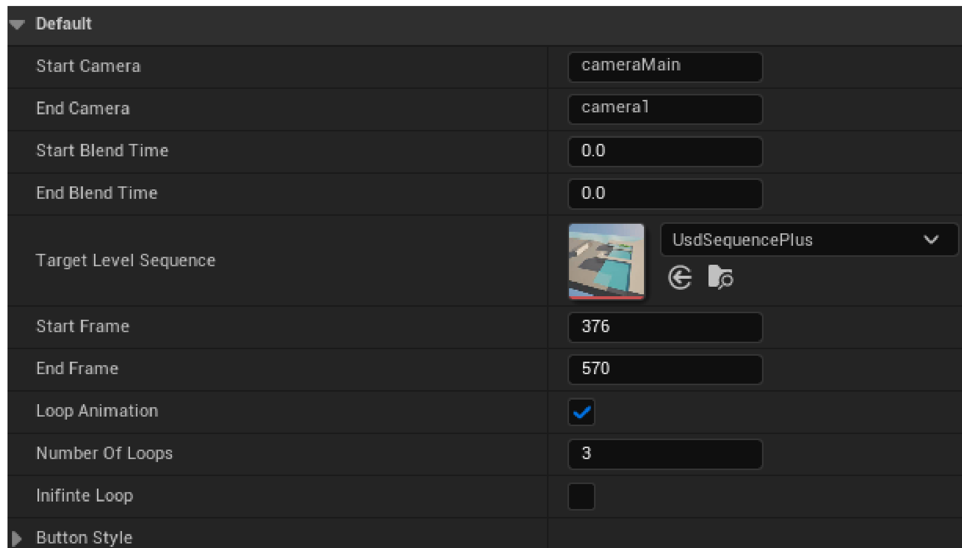

Figure 3.4: Parameters to set up Play Frame Range Button

Play Frame Range Button

To use the Play Frame Range button (Figures B.1, B.2), the user must input start and end camera, and their corresponding blend times. If no end camera is given, the start camera is used. These cameras are both found with the `FindCameraByString` function, due to USD camera references not remaining consistent. Then, the user can select the target level sequence from the content browser that will play when the button is clicked. This is for any regular level sequence, for USD level sequences the following button is to be used. A start and end frame is then given, which is the important feature for Proto's use case of importing a large project timeline. There are other options to allow looping of the frame range, with a boolean to determine whether to loop or not alongside the number of loops desired. There is also the option to loop infinitely, which will overwrite the other option.

USD Play Frame Range Button

The USD Play Frame Range button inherits the Play Frame Range button and works in the same way. The only difference is that on the construct event it will find the UsdStageActor in the level and assign the corresponding level sequence to the target level sequence. This overwrites any input target level sequence the user may add. There is not a method to remove that variable being present through the inheritance.

Stop Sequences Button

Using the level sequence manager and the widget button function library, this button (Figure B.3) checks to see whether there is currently an assigned sequence player in the level sequence player manager. If there is, it will stop that sequence playing. This is a simple button, but is particularly useful to interrupt an infinite loop and is essential in a set of interactive buttons.

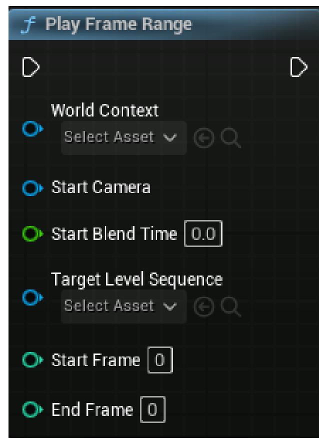

Figure 3.5: Play Frame Range Function Node

Pause Play Button

Similar to the Stop Sequences button, the pause play button (Figure B.4) uses the level sequence player manager to check that there is currently an assigned level sequence. Then, it uses the IsPaused function from the sequence player and if it is currently paused, it plays and if it is playing, it pauses.

Switch Camera Button

The switch camera button takes the camera name and blend time as inputs. Using the `FindCameraByString` function, it sets the camera with that string to the view target with the given blend time. An example use case of this could be multiple static cameras around a scene, allowing the user to change their view target to explore the scene. This would also be suitable if there are multiple animated cameras in a sequence.

3.2.3 Widget Button Function Library

Play Frame Range

The Play Frame Range function (Figures B.5, B.6) is the core to the interactivity of a level sequence in the project. Originally, the blueprint was part of one of the buttons in the next subsection, but it was since extracted into its own function. By having it in a separate function, the use cases of this widens to be used within any blueprint to be set off by more than just a widget button click. This node can be seen in Figure 3.5.The parameters are the Start Camera, blend time into the camera, the level sequence object, start frame and end frame. Using the previously mentioned level sequence player manager, it first checks to ensure there is a valid level sequence player assigned to the manager, and stops the current sequence if there is. The camera is then assigned to the view target and the input level sequence is set to the level sequence player manager. Then the sequence is played within the specified frame range.

Find Camera by String

As the camera components generated by the USD are transient, an alternative approach to referencing their object has to be made. For this reason, the `FindCameraByString` was included (Figure B.7), which is frequently used to find any camera object by a user's input string. As well as assisting with the issues with referencing the objects, it also provides greater opportunities for logic in blueprints checking camera names, and for parameters to buttons and functions.

Set Level Sequence Player

Providing a connection between a blueprint and the Level Sequence Player Manager, this function (Figure B.8) is used to assign a provided level sequence to the Level Sequence Player variable, ready to play.

Get Level Sequence Player

This function (Function B.9) is how the Level Sequence Player is accessed from its manager, allowing the blueprint to access it so that the sequence can be played.

Is Current Sequence Player

To ensure that there are no runtime errors, this function (Figure B.11) should be called in any function before attempting to stop a sequence. It checks to see whether any sequence has been assigned to the Player variable, so that there aren't any issues trying to get a null pointer which could lead to crashes.

Stop Current Sequence

After using the Is Current Sequence Player function, this can be used to stop any current playing sequence associated with the player manager (Figure B.10).

3.2.4 Example Blueprint

Included with the plugin is an example blueprint (Figures B.12, B.13, B.14), showcasing a few approaches to use the widget button function library outside of the buttons. This is largely to assist in the learning for Proto, to demonstrate how to use them within a number of loops, infinite loop or to stop, in this case with key press events. These could be copied into a level blueprint for example.

3.3 USD Editor Tools

Extending the editor standalone window plugin class, this module combines a set of editor tools relating the USD prim attributes (Figure 3.6). During tests to get a packaged project out of Unreal Engine, there was a period of time where Proto Imaging were having trouble getting a functioning build that could include the USD in runtime. This is still a potential hurdle for the future, although more recent tests have displayed promise that functionality should be available. A few of these tools come from the idea that instead of using the USD at runtime, it can be used during production alongside the frequent updates from Maya. When production ready, all of the assets could be imported directly

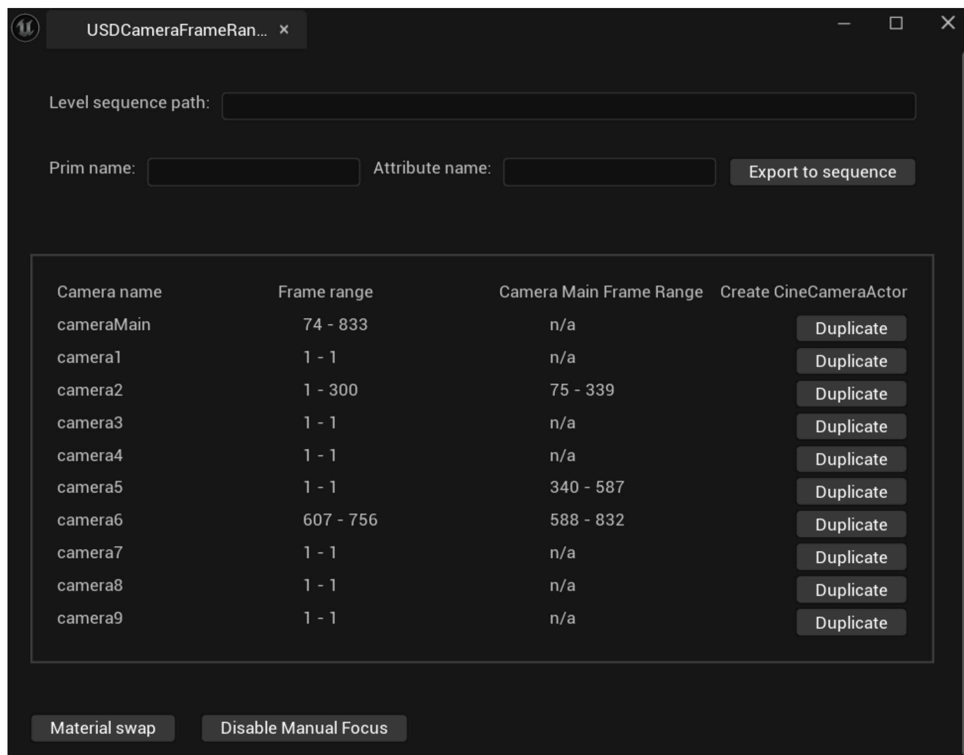

Figure 3.6: USD Editor Tools Window

into the project to be packaged without the need for the USD. In this case, there could be a need for custom attributes that wouldn't be available otherwise.

As part of Proto Imaging's Maya workflow, they use a cameraMain with a custom attribute containing camera numbers (Figure 3.7). These camera numbers are in reference to other cameras in the project. Using a script, this camera number attribute is key-framed, adjusting cameraMain to serve as a form of camera cuts in Maya. It was very important to Proto to keep this workflow in Unreal Engine, meaning some of the following tools are dedicated specifically for this pipeline.

Figure 3.7: Camera Main Camera Number Attribute

3.3.1 Camera Information and duplication

At the core of the camera functionality here, is the struct `FCameraInfo`. The class contains an array property containing all of the camera's in the USD, each one stored in this struct format. The camera prims are found with a depth first search on the USD hierarchy, finding any prims matching the type "Camera". Once they are found, the name, SdfPath, translation and rotation attributes and timesamples, start and end frame and camera properties are all stored in the struct.

Camera Frame Ranges

Once found, their name and frame ranges are displayed in the window. This is to assist the user when combining with the play frame range function and buttons. A project could contain many cameras, making it challenging to remember the originally assigned frame ranges. As Proto's vision is based around these frame ranges, this is a beneficial addition. The cameras are also checked to see if they are featured in camera main's camera number custom attribute. If they are, their camera main frame ranges are added to the struct and dislpayed in the window alongside their animation. This is a key tool, with more importance than the individual camera's animation as it affects both animated and static cameras. These frame ranges can then be referenced by the artist when determining what frame ranges to use in their buttons and functions.

Camera Duplication

Next to the frame ranges is a button to create a duplicate of that USD camera. When clicked, a CineCameraActor is spawned. The starting translation and rotation are set, as well as the camera properties including the focal length, focus distance, FStop and the horizontal and vertical aperture. If no level sequence path has been added, then a static camera with these properties will be created. If however a level sequence path has been added then the `AddCameraToLevelSequence` function is called.

To add the animated camera to a level sequence, a Guid must be found for the generated camera. If valid, a transform track is added with the Guid, and then a transform section added to the track. The section must have its frame range set, which can be calculated using the start and end frame multiplied by the ticks per frame. The ticks per frame can be calculated by the level sequence's tick resolution divided by the level sequence display rate. In this section, channels have to be added for the X, Y and Z for the translation and rotation values of the camera. As the time samples were saved onto the camera info struct, they can be iterated over, with the translate and rotate values set only on the key-frames from the timesamples array, which provides the animation of the camera on the level sequence. These translation and rotation values are found using the functions described in the `UsdAttributeFunctionLibrary` discussed in Section 3.1. The camera duplication may be used in cases where camera animations may want to be set off at different times, or to use outside of the USD without editing the USD itself.

3.3.2 Attribute Export to sequence

The option to export an attribute onto a sequence means that the animated attribute values can be completely imported into the project, without ties to a USD. This may be desired if there is an

animated attribute, such as a particle rate, that does not need to tie to the USD but is featured on it. In the case of this project, it is only set up for float attributes but has the potential to expand into other types.

The user enters the level sequence path, prim name and attribute name. Functions `GetUsdAttributeInternal` and `GetUsdAnimatedFloatAttribute` from the `UsdAttributeFunctionLibrary` are used to find the attribute from the input strings, and then its value. The track and section is set up in the same way as the camera duplcation, but with a float section and track instead of a transform track. Then, the attribute values are added to the keyfames using the timesamples found from the attribute.

3.3.3 Material Swap

To address Proto's need to swap USD shaders with Unreal materials with the same name, a button has been added to find all of the UMaterial objects in the specified directory. This functionality will be accessed from a button on the editor window.

When the button is clicked, the `GetAllMaterials` function is called. This looks into the content browser, specifically in the path "/Game/Materials" and searches for any UMaterials. Then the `TraverseAndCollectMaterials` is then called, performing a depth first search on the USD, checking to see if the current prim has a material binding relationship. If the relationship exists, the target paths are gathered for these bindings. For each path, material's shader prim is found, with the object name, material name and prim path stored in a `FMaterialInfo` struct. The array of collected materials are compared with the found Unreal materials to check if their names match. If they match, then the component generated by the prims with the found material relationship are assigned the Unreal material. This new binding can be saved back onto the USD, and will be used in future Unreal sessions.

3.3.4 Disable manual focus

The disable manual focus button runs through the level and changes all of the cameras' focus settings to the enum `Do Not Override` instead of `Manual`. This is a small tool to assist in early problems with development where cameras imported from the USD had issues with their focus settings and would appear blurry. However with added focus settings to the USD from Maya, this issue hasn't appeared, so isn't needed often.

4 Results

This chapter looks at the different cases that Proto have been using these tools during and since their production. At the time of writing, new uses are still being found frequently. Lots of these approaches such as the exporting of custom attributes and animated materials would not otherwise be possible from Maya to Unreal Engine.

4.1 USD Attribute Uses

4.1.1 Object Locations

One of the main uses for accessing vector attributes has been finding the location of objects. In Proto's work, this has been to include icons that can float above different objects in the scene. Helping to provide a greater interaction in the product, these icons can be clickable to serve as buttons, or display information panels about the items in the scene. These icons can appear above static objects by finding their default translation attributes (Figure 4.1), or animated objects by finding the translation values at the current time from the UsdStageActor.

4.1.2 Particle system attributes

The position of a particle system can be changed with the previously mentioned approach, using a translation vector. Proto have also used USD attributes to control particle rates and custom colours of particles by exporting these attributes and assigning them in the level blueprint. This allows them to animate these values alongside the rest of the project, as it fits, rather than having to adjust these after the import into Unreal Engine. In this case, particle system variables such as the spawn rate have to be exposed to be able to change from other blueprints, which then allows it to be set to the value as normal.

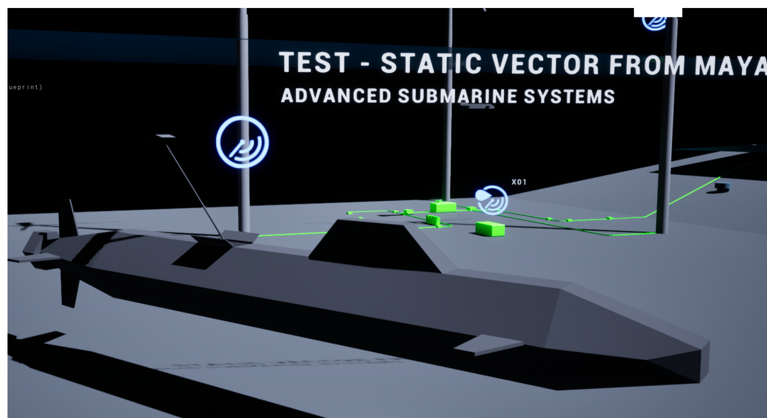

Figure 4.1: Icon placed with translate vector attribute

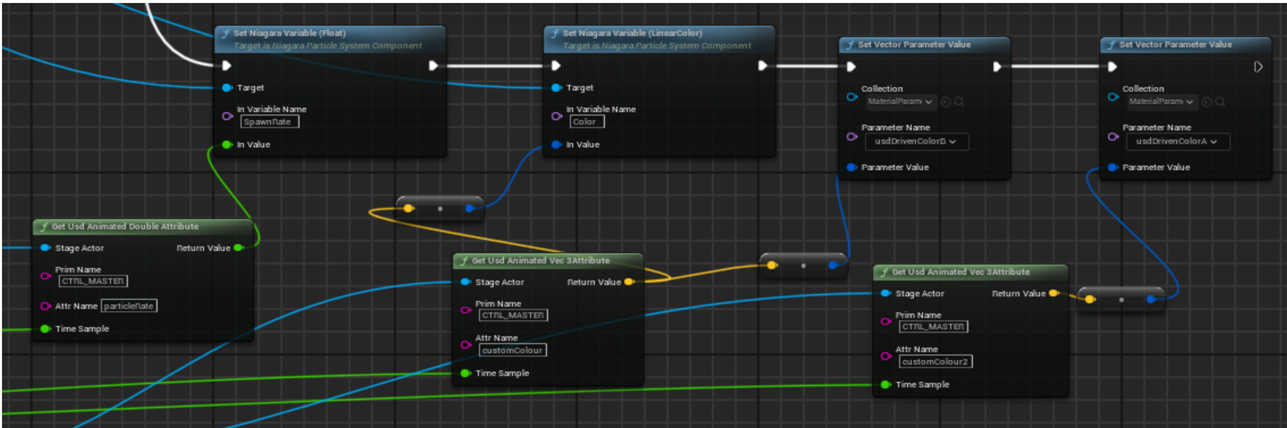

Figure 4.2: Particle and Material Attributes being assigned

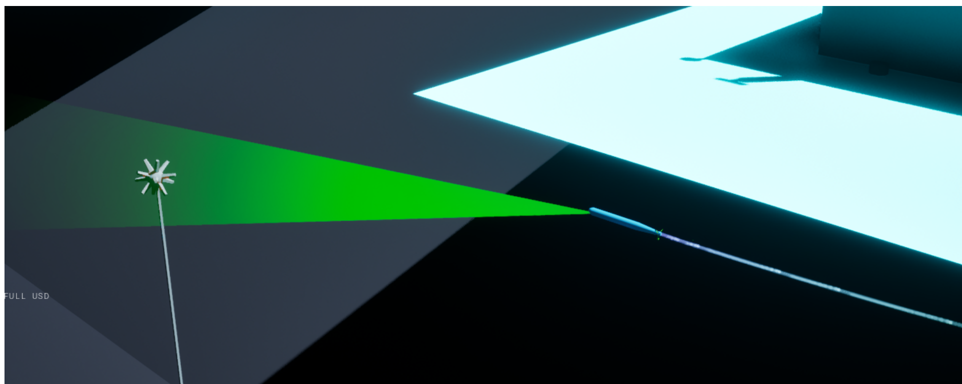

Figure 4.3: Scanner with swapped Unreal Material using custom USD attributes for colours and particle rate

4.1.3 Animated materials

After normalising colours in Maya, the RGB values can also be exported as vectors to be used in Unreal Engine. A Material Parameter Collection is used, which has its vector parameter set in the level blueprint and its parameter value read in from the material graph. Having this set up with animated vector values allows the animation of materials, reading in the attribute value at the current time sample, further providing options to import these areas from Maya.

4.2 Material swap

After setting up animated materials with a colour attribute, Proto then needed to be able to access this Unreal Material. The material swap tool has allowed this, with Proto using an animated colour material as an animated scanner in the scene, changing colour when it spots what it's looking for. Having animated this in Maya, at first it doesn't appear to have the desired affect, as the material's colour remains the same from the USD shader. Accompanied by the animated material, Proto have been able to incorporate this into the project. This is a proof of concept for one material swap, however it shows great potential to be able to swap change the USD shader to custom Unreal Engine ones automatically within a large scale project.

4.3 Interactive HUD

Using the Widget Button's made in this project, Proto have set up a base level HUD (Figure B.15) for the user to interact with the scene with. This utilises the frame ranges to play different sections of the USD animation. In addition to these buttons, Proto have set up clickable icons in the scene which utilise the functions in the Widget Blueprint Function Library to carry out frame range playback as desired.

4.4 Testing and Design changes

4.4.1 Testing

Throughout the development of this project, there was frequent testing carried out to ensure that the tools worked as expected. All updates were pushed to the github repository and downloaded by Proto. Through this back and forth, bugs and unprotected sections of code were found and fixed to prevent crashes. Unreal Engine includes an Automatic Test Framework (Epic Games 2024a) which was initially used to produce unit tests for the functions, however some issues appeared with this. As most of the functions require access to a USD stage, this is set up using the Set Root Layer function for the UsdStageActor, which works asynchonously. Due to time constraints, this was unable to be set up completely. With this, the plugin was frequently tested manually using incorrect inputs and invalid setup scenarios to ensure that it won't lead to crashes. In these situations, many conditionals are set up to write error and warning logs and to return default values to allow the project to continue.

4.4.2 Design Changes

Buttons

One of the major areas in the project that changed was the approach to buttons. Originally, the project was to revolve around the buttons due to them being the core interaction with the USD. This was extracted into the Widget Button Function Library, predominantly to a more flexible Play Frame Range function, allowing frame ranges to be played from other areas like the floating icons, or key presses on a keyboard. It also allowed more automated processes, with detailed blueprints being able to play sequences, while utilising the Level Sequence Player Manager to ensure synchronised playback.

The buttons were also set up with functionality in mind, without styling being a factor. This was apparent to be a problem for Proto as flexibility in button design is important to produce professional applications, so styling options were added to the buttons and their text.

Camera Frame Ranges

At first, the camera frame ranges window just displayed the frames of animation for each camera. However, this was far less important information to Proto than seeing the ranges used by camera-Main as this treats static cameras with equal value. Having these options appear provided greater productivity when working in Unreal, as the user will not have to look far to see what frame ranges they must set up their functions for, or which camera to set for these ranges.

5 Conclusion

This project aimed to assist Proto Imaging with integrating their Maya workflow into Unreal Engine to allow them to transition into real-time 3D animation. USD was chosen as the ideal file format after comparisons with other formats, such as FBX and Alembic, to handle the frequent changes typical of client-driven projects. Importing USD files would directly allow this, with a set of tools that could keep a project developing by interacting with the file in a one way setting.

These developed tools have addressed many of the challenges associated with using USD in Unreal Engine. The USD Attribute Function Library provided direct access to previously inaccessible values, assisting with actor locations, particle rates, and animated materials in a runtime setting. Alongside this, the Widget Button Function Library provided a selection of functions enabling interaction with USD animation, by setting off animations within given frame ranges. Sequences were synchronised with a manager class so that they do not cause any issues overlapping with one another. The functions written for this have also extended out of the buttons, providing Proto with a flexible set of tools for animation playback anywhere in the project. Within the editor, material swap functionality was developed to allow a user to swap their USD shaders with materials created within Unreal Engine, assisting in an adaptable workflow.

Going forwards, the main development is to rebuild this plugin to work in version 5.5 of Unreal Engine. Currently built for version 5.4.2, Proto were unable to get a successful project build with runtime USD features in any 5.4 build of Unreal Engine. At the time of writing, Unreal 5.5 is currently being developed but has not been released. Using the Unreal Engine github, Proto were able to produce a successful build of a USD based project. However due to the changes of the USD plugins in this new version, many features of the plugin will need rebuilding to be able to package in this later version.

Additionally, the Automation Test Framework for Unreal Engine will be incorporated in order for the plugin to be a better developed piece of software. Further functionality can be added to the types of attributes that can be accessed as part of the USD Attribute Function Library. Currently only available for float, double and int types, this is a large area to expand as required for a wider range of attribute transfer.

From a user interaction perspective, the display window containing the frame ranges could be improved. one area here could be to improve expand the material swap into its own section displaying materials found to allow the user to select which materials to swap, rather than an automated approach across all of them. The same concept can be applied to the disable manual focus, although that remains a relatively minor feature. While the import attributes to a level sequence isn't required with runtime USD access, this area in the UI could also be expanded to work alongside the USD Stage Editor for a quicker experience selecting attributes.

In conclusion, this project has successfully addressed several of the limitations associated with the current USD functionality in Unreal Engine, while also directly resolving specific pipeline challenges faced by Proto Imaging. By developing tailored tools and workflows, the project has laid the groundwork for a more flexible and efficient real-time animation pipeline.

Bibliography

Ardolino, A., Arnaud, R., Berinstein, P., Franco, S., Herubel, A., McCutchan, J., Nedelcu, N., Nitschke, B., Robinet, F., Ronchi, C., Samour, G., Turkowski, R. and Walter, R., 2014. *Geometry and Models: 3D Format Conversion (FBX, COLLADA)*. 19–37.

Epic Games, 2024a. Automation system in unreal engine. `https://dev.epicgames.com/documentation/en-us/unreal-engine/automation-system-in-unreal-engine`. Accessed: 2024-08-11.

Epic Games, 2024b. Universal scene description in unreal engine. URL `https://dev.epicgames.com/documentation/en-us/unreal-engine/universal-scene-description-in-unreal-engine`, accessed: 2024-08-05.

Pixar, 2023. Universal scene description. `https://openusd.org/docs/`. Accessed: 2024-08-03.

Proto Imaging, 2024. Profile - proto imaging. `https://www.protoimaging.com/profile.shtml`. Accessed: 2024-08-08.

SPI, ILM, 2024. Alembic: Interchange framework for computer graphics. URL `https://www.alembic.io/`, accessed: 2024-08-08.

Unreal Engine, 2024. Unreal engine 5.4. URL `https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5.4-release-notes`, accessed: 2024-08-08.

Appendix A - Code

```python
class AttributeSelectionDialog(QtWidgets.QDialog):

    def __init__(self, parent=get_main_window()):

        # Sets up the GUI

    def update_selected_objects(self, *args):
        self.selected_objects_list.clear()
        selected_objects = cmds.ls(selection=True)
        if selected_objects:
            for obj in selected_objects:
                self.selected_objects_list.addItem(obj)

    def update_attributes(self, item):
        self.attributes_list.clear()
        user_defined = self.user_defined_checkbox.isChecked()
        attributes = cmds.listAttr(item.text(), userDefined=user_defined)

        if attributes:
            for attr in attributes:
                if attr == "USD_UserExportedAttributesJson":
                    continue
                self.attributes_list.addItem(attr)

    def create_usd_attributes(self):
        selected_object = self.selected_objects_list.selectedItems()[0].text()
        selected_attributes = [item.text() for item in self.attributes_list.selectedItems()]

        attr_json = {}
        usd_attr = "USD_UserExportedAttributesJson"
        if cmds.attributeQuery(usd_attr, node=selected_object, exists=True):
            cmds.deleteAttr(selected_object, attribute=usd_attr)

        for attr in selected_attributes:
            attr_value = cmds.getAttr(f"{selected_object}.{attr}")
            attr_type = cmds.attributeQuery(attr, node=selected_object, attributeType=True)

            attr_json[attr] = {"usdAttrName": attr, "usdAttrType": attr_type, "usdAttrValue" : attr_value}

        cmds.addAttr(selected_object, longName=usd_attr, dataType="string")
        cmds.setAttr(f"{selected_object}.{usd_attr}", json.dumps(attr_json), type="string")
```

Code A.1: Maya Custom Attribute Exporter

```cpp
    template <typename T>
T UUsdAttributeFunctionLibraryBPLibrary::GetUsdAttributeValueInternal(
    AUsdStageActor* StageActor, FString PrimName, FString AttrName)
{
    UE::FUsdAttribute Attr = GetUsdAttributeInternal(StageActor, PrimName, AttrName);

  // Check that an attribute has been found from the given inputs
    if (!Attr)
    {
        UE_LOG(LogTemp, Warning, TEXT("Specified attribute is not holding any value")
    )
        return T();
    }

  // Using the Unreal wrapper of the pxr type VtValue
    UE::FVtValue Value;
    bool bSuccess = Attr.Get(Value);

    if (!bSuccess)
    {
        UE_LOG(LogTemp, Warning, TEXT("Failed to get value for Attribute: %s"), *Attr
    .GetName().ToString());
        return T();
    }

  // Required to return the useable type within Unreal
    return ExtractAttributeValue<T>(Value);
}
```

Code A.2: GetUsdAttributeValue Template Function

```cpp
    UE::FUsdAttribute UUsdAttributeFunctionLibraryBPLibrary::GetUsdAttributeInternal(
    AUsdStageActor* StageActor, FString PrimName, FString AttrName)
{
    // Check if the StageActor is valid
    if (!StageActor)
    {
        UE_LOG(LogTemp, Error, TEXT("StageActor is null"));
        return UE::FUsdAttribute();
    }

    // Retrieve the Usd stage from the actor
    UE::FUsdStage StageBase = StageActor->GetUsdStage();
    if (!StageBase)
    {
        UE_LOG(LogTemp, Warning, TEXT("No Usd Stage found"));
        return UE::FUsdAttribute();
    }

    UE_LOG(LogTemp, Log, TEXT("Found stage"));

    UE::FSdfPath PrimPath;
    UE::FUsdPrim root = StageBase.GetPseudoRoot();
    if (!root)
```

```
23     {
24         UE_LOG(LogTemp, Warning, TEXT("Failed to get PseudoRoot"));
25         return UE::FUsdAttribute();
26     }
27 
28     // Retrieve the path of the specified prim
29     GetSdfPathWithName(root, PrimName, PrimPath);
30     if (PrimPath.IsEmpty())
31     {
32         UE_LOG(LogTemp, Warning, TEXT("PrimPath is empty for PrimName: %s"), *
    PrimName);
33         return UE::FUsdAttribute();
34     }
35 
36     // Get the prim at the specified path
37     UE::FUsdPrim CurrentPrim = StageBase.GetPrimAtPath(PrimPath);
38     if (!CurrentPrim)
39     {
40         UE_LOG(LogTemp, Warning, TEXT("No Prim found at path: %s"), *PrimPath.
    GetString());
41         return UE::FUsdAttribute();
42     }
43 
44     // Get the attribute from the prim
45     const TCHAR* AttrNameTChar = *AttrName;
46     UE::FUsdAttribute Attr = CurrentPrim.GetAttribute(AttrNameTChar);
47     if (!Attr)
48     {
49         UE_LOG(LogTemp, Warning, TEXT("No Attribute found with name: %s"),
    AttrNameTChar);
50         return UE::FUsdAttribute();
51     }
52 
53     return Attr;
54 }
55 
56 T UUsdAttributeFunctionLibraryBPLibrary::ExtractAttributeValue(UE::FVtValue& Value)
57 {
58     // Access the pxr VtValue from the Unreal wrapped FVtValue
59     pxr::VtValue& PxrValue = Value.GetUsdValue();
60 
61   // Check to ensure the value is of the specified type
62     if (PxrValue.IsHolding<T>())
63     {
64         // Access the value of the specified type from the VtValue
65         T AttrValue = PxrValue.Get<T>();
66         UE_LOG(LogTemp, Log, TEXT("Successfully retrieved attribute"));
67         return AttrValue;
68     }
69     else
70     {
71         UE_LOG(LogTemp, Warning, TEXT("Attribute is not holding a value of specified
    type"));
```

```
72      }
73      return T();
74  }
```

Code A.3: USD Attribute Value Extraction

Appendix B - Blueprints

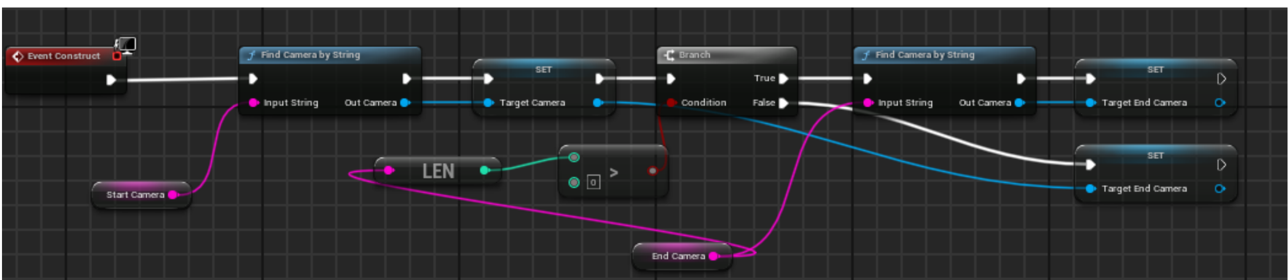

Figure B.1: Play Frame Range Button Construct

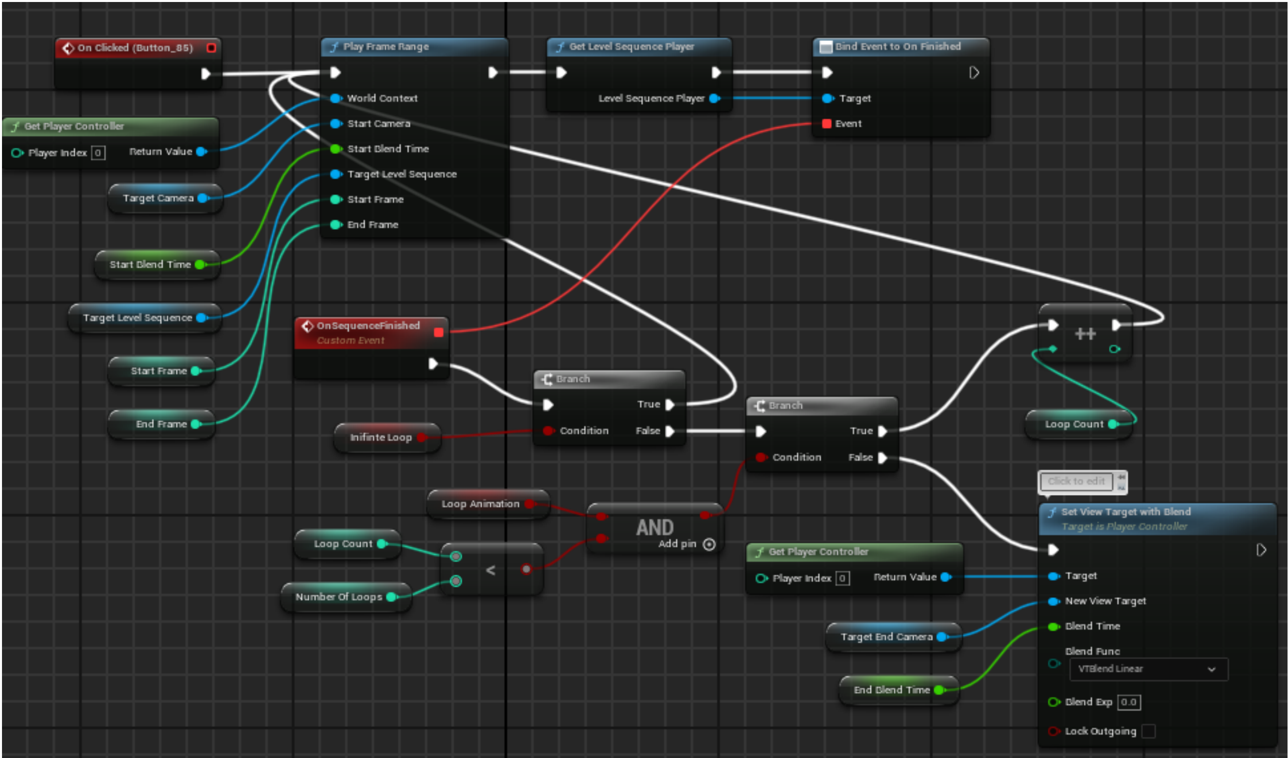

Figure B.2: Play Frame Range Button Blueprint

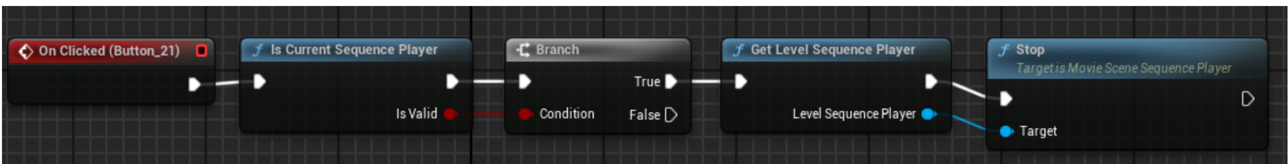

Figure B.3: Stop Sequences Button

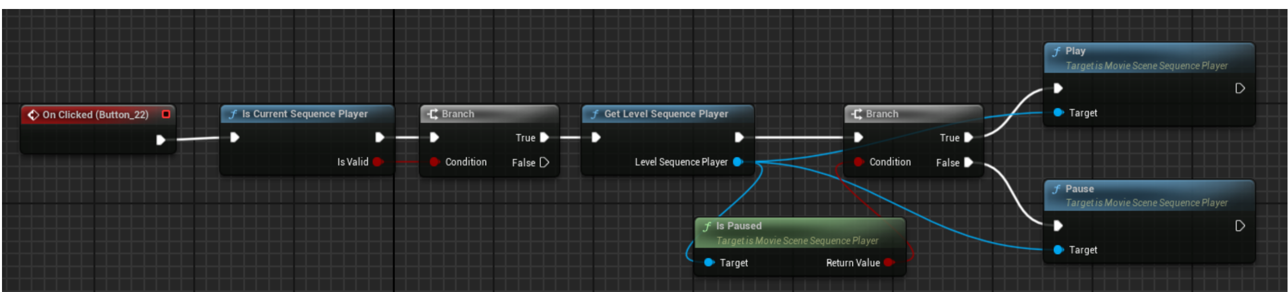

Figure B.4: Pause/Play Button

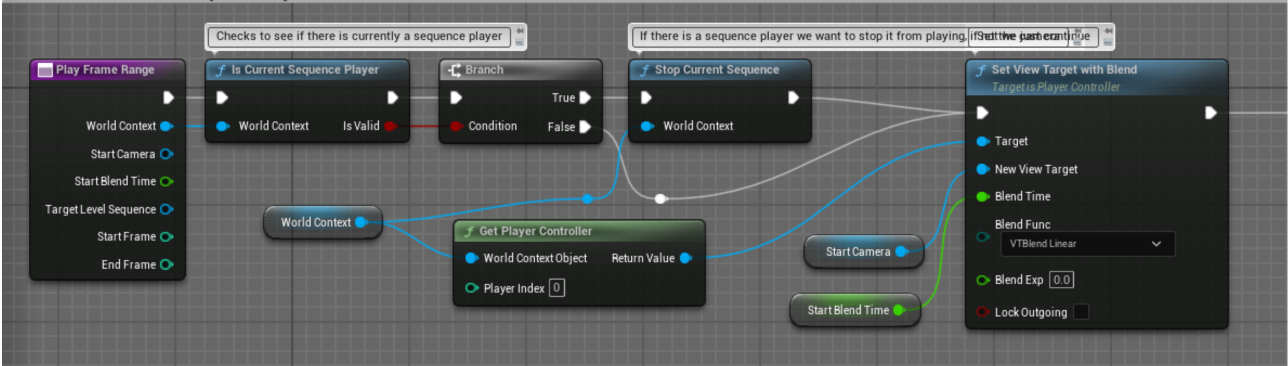

Figure B.5: Play Frame Range Blueprint Function 1/2

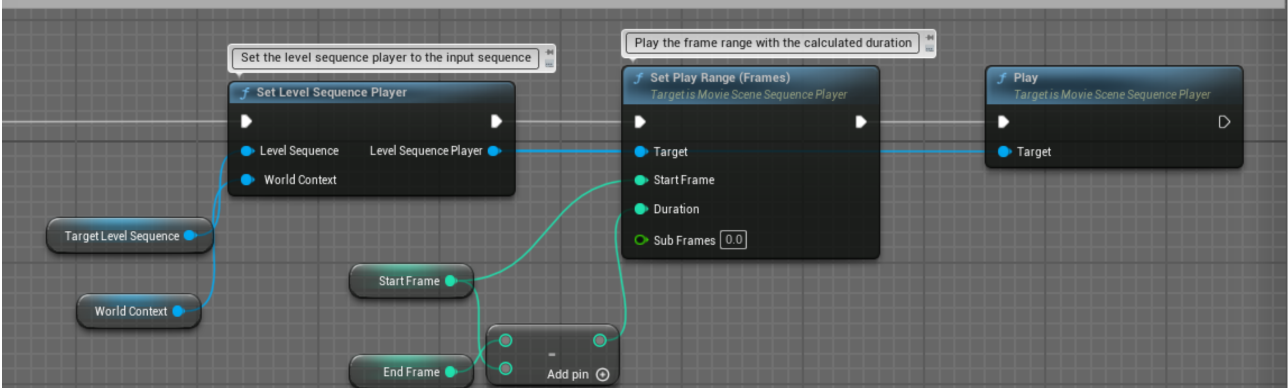

Figure B.6: Play Frame Range Blueprint Function 2/2

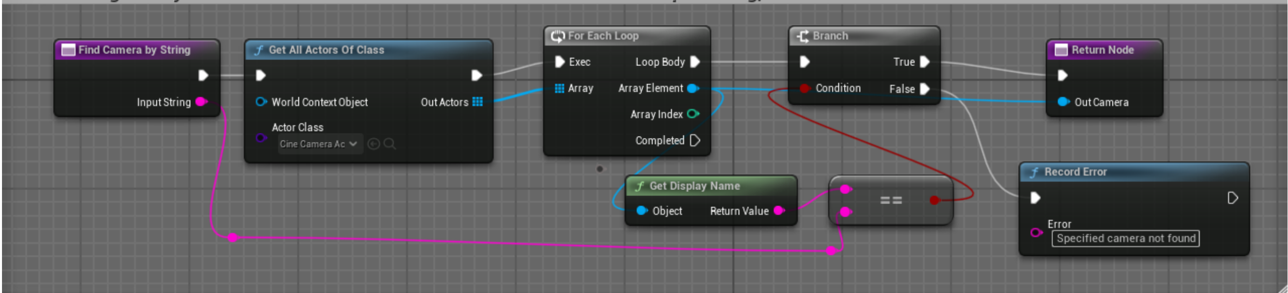

Figure B.7: Find Camera By String Function

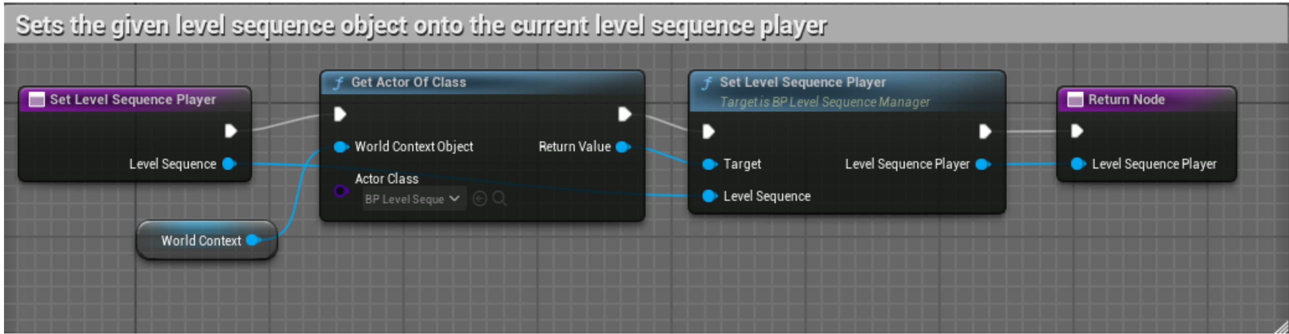

Figure B.8: Set Sequence Player Function

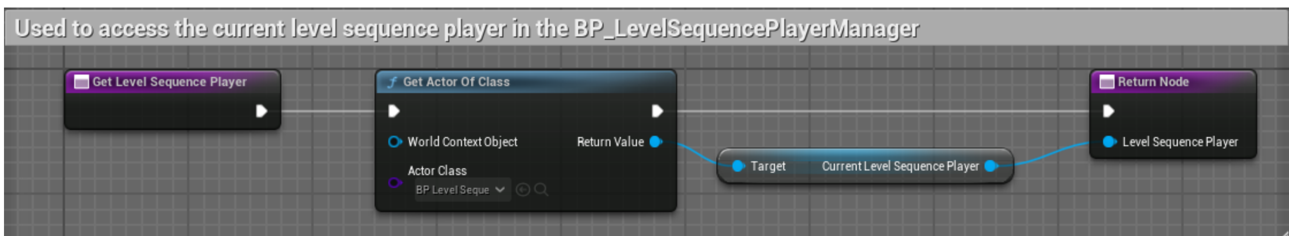

Figure B.9: Get Sequence Player Function

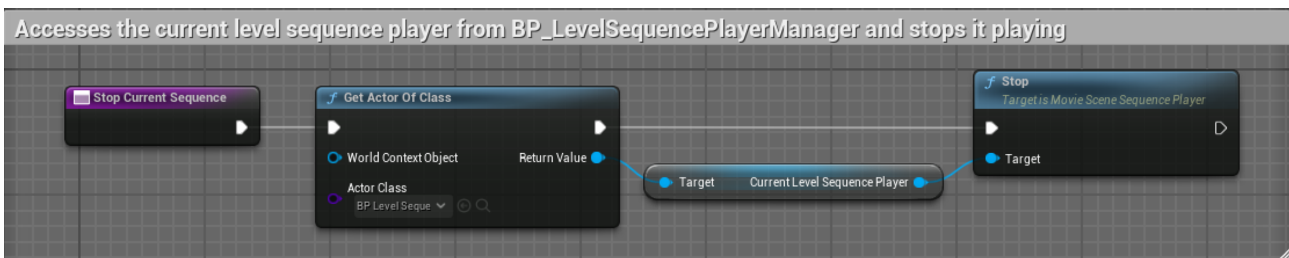

Figure B.10: Stop Current Sequence Function

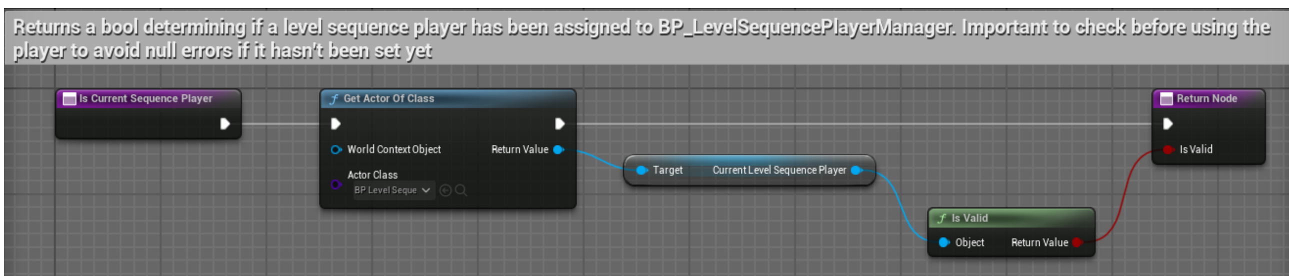

Figure B.11: Is Current Sequence Function

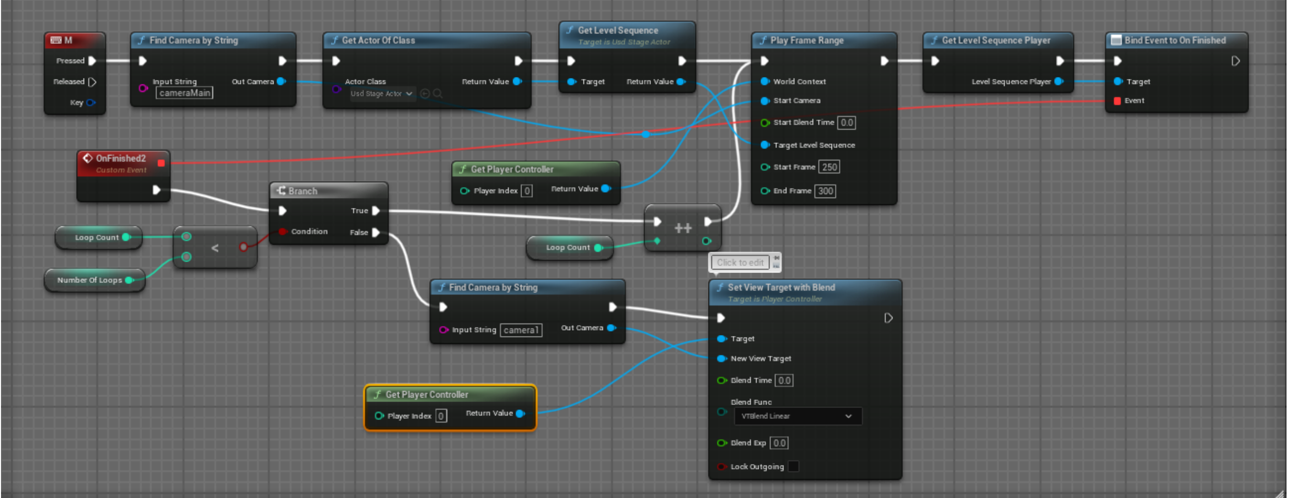

Figure B.12: Example Custom Loop Number Blueprint

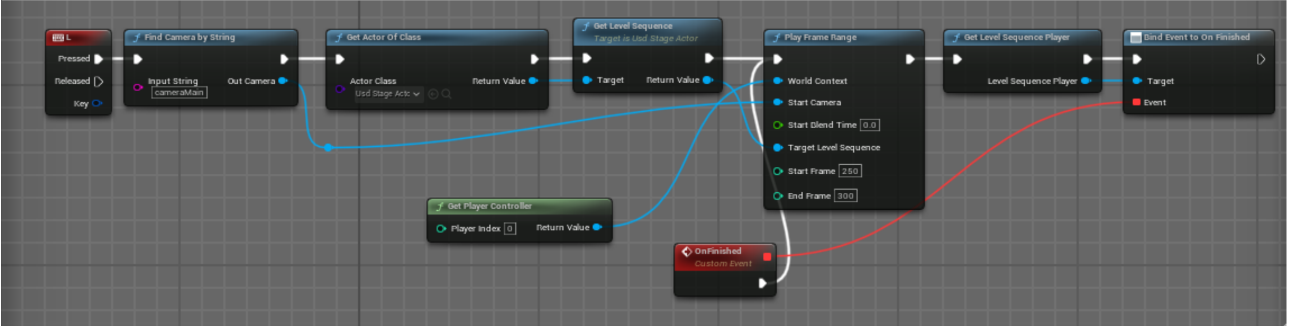

Figure B.13: Example Infinite Loop Blueprint

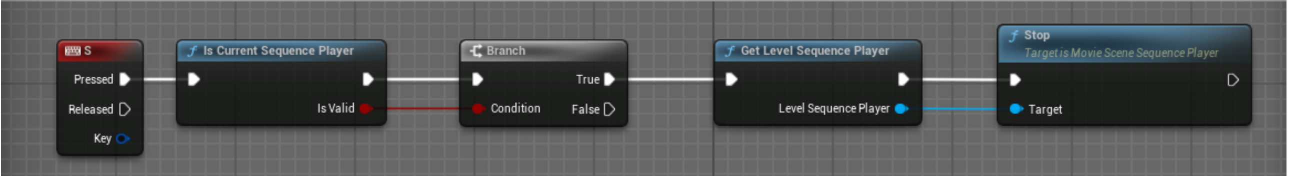

Figure B.14: Example Stop Sequence Blueprint

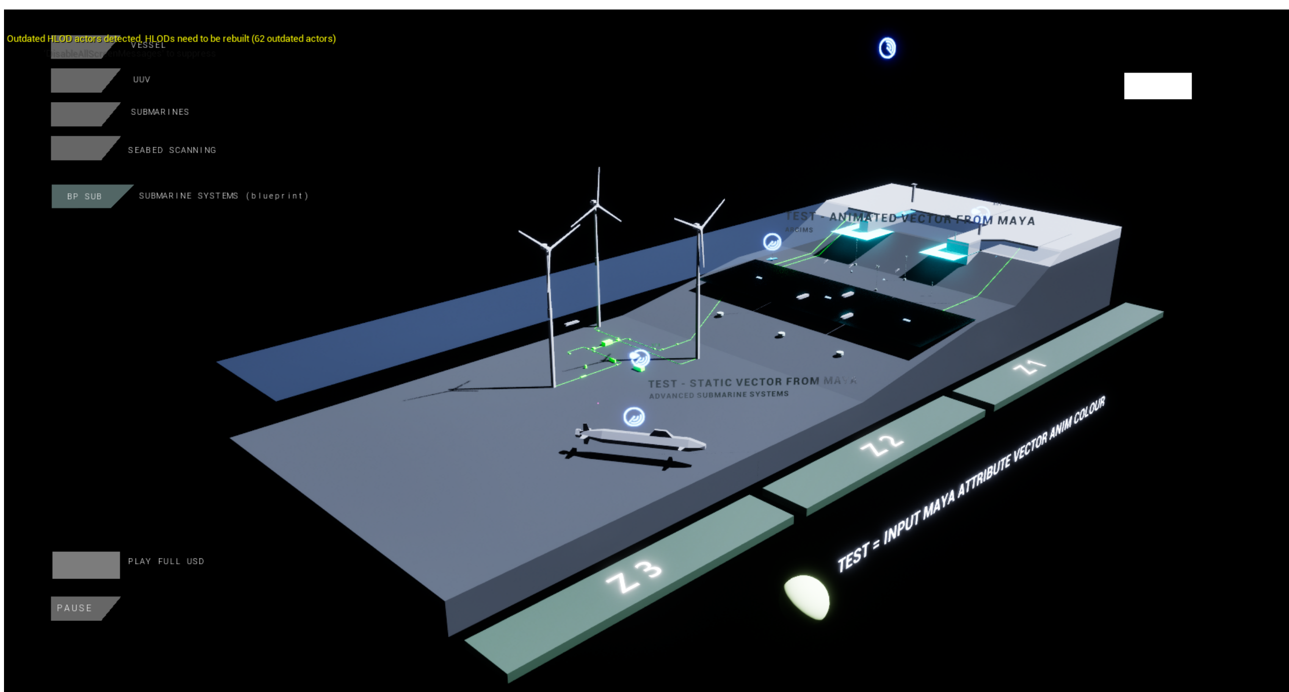

Figure B.15: Proto Imaging Example HUD