POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Software architecture for AR art exhibitions in Unreal Engine

Relatore: **Prof. Elisabetta Di Nitto**

Tesi di Laurea Magistrale di **Mihai Barsan**, matricola **905604**

Anno Accademico 2020-2021

Thanks to all the people involved.

Sommario

Questa tesi è il resoconto di un lavoro svolto nell'ambito dei beni culturali che consiste nel sviluppare un'applicazione che consenta all'utente di ammirare un dipinto tramite l'uso di un dispositivo di realtà aumentata messo a disposizione dei visitatori durante una mostra artistica. Il contributo principale fornito da questo lavoro è un'architettura software che può essere utilizzata per implementare applicazioni simili.

Il capitolo introduttivo descrive l'ambito al quale si fa riferimento e quali potrebbero essere gli scenari di applicazione di questo lavoro. Il secondo capitolo illustra il problema che si deve affrontare e gli obiettivi da raggiungere. Viene poi raccontata l'opera artistica a cui si fa riferimento descrivendola e parlando brevemente della sua storia. Infine si introduce il dispositivo che verrà utilizzato per eseguire l'applicazione: se ne descrivono i principali pregi e caratteristiche hardware. Il terzo capitolo è dedicato ad Unreal Engine che è il motore grafico utilizzato per implementare l'applicazione. Si comincia parlando delle principali caratteristiche di questa tecnologia, successivamente vengono introdotti gli elementi fondamentali da conoscere per poter comprendere il codice che verrà mostrato nel capitolo seguente. Il quarto capitolo è il fulcro della tesi. Nella prima parte si introduce brevemente l'implementazione dell'ambientazione virtuale e del quadro in sé poi l'attenzione si sposta sulla parte software che è lo scopo principale di questo lavoro. Viene quindi spiegata l'architettura software utilizzata: si comincia esponendo gli obiettivi che si vuole raggiungere per poi discutere le sue caratteristiche nonché similarità con altri design pattern esistenti. Si fornisce quindi una descrizione delle varie parti costituenti e del loro funzionamento. L'ultimo capitolo riassume il lavoro svolto e da dei suggerimenti su come l'architettura software descritta si potrebbe utilizzare in scenari simili.

Abstract

This thesis presents a software architecture designed to implement a virtual reality application developed for an artistic exhibition. It makes use of the Unreal Engine and uses the tools and classes provided by the engine. Specifically what this architecture does is to simulate a finite state machine allowing the developer to define state classes that inherit from other state classes and execute custom code on state transitions by declaring methods that make use of the substitution principle.

Table of contents

1 Introduction

The improvement of the tools available to artists and audiovisual content creators has introduced a series of new means to experience the **art** and **cultural heritage** by enthusiasts from all over the world. In particular in recent years we have seen the introduction and rapid evolution of **virtual reality** devices. We went from being barely able to synchronize the movement of the head with the movement of a camera overlooking a simple virtual world containing a bunch of polygons, to what exists today. That is, highly reactive devices capable of showing virtual worlds that are increasingly complex and realistic in their representation, devices that offer greater interaction capabilities such as hand tracking or that allow multiple users to share the virtual experience and see the exact same dynamic scenario that changes for all of them at the same time.

Virtual reality has remained mainly confined to the world of video games. Although there have been attempts to employ virtual reality for other purposes, it is only recently that virtual elements have begun to be successfully introduced in artistic contexts, the reason being: when it comes to museums or art exhibitions, the aim is to see the peaces of art firsthand, also the staging and setting are equally important. Virtual reality headsets impose a big limitation as they confine the viewers to the virtual environment, completely isolating them from the surroundings.

The problem that one tries to solve with **augmented reality** devices is precisely this: to show virtual elements overlapping the real ones. The first iteration of this technology is the virtual reality of mobile phones. It is a simple technology that does nothing but add virtual images on top of the images coming from the camera of a

smartphone. It is not a true augmented reality device but rather an editing trick which consists of modifying a video stream in real time giving the illusion that the virtual image is somehow anchored in a real space by relying on the orientation sensors of the phone to keep the image in position. The second iteration of this technology is represented by **mixed reality devices**. These devices are able to actively map the surrounding physical space and create a 3D mesh within virtual reality. This opens up new interaction opportunities and allows virtual objects to interact with real ones. Let's imagine a virtual ball that is thrown against a real wall and bounces back realistically: this is possible if the virtual world has a 3D model of the real surfaces in the form of an invisible mesh which has the sole purpose of generating collision events. [1]

1.1 Research context

This chapter gives a clarification about the context in which the work related to this thesis was carried out, what is the nature and contribution offered by this work not only limited to the purpose it was intended for, but generalized to other scenarios with similar requirements.

This thesis presents the report of a work aimed to create an **augmented reality** experience to allow the viewer to admire the painting Sistine Madonna in the place where it should have originally been placed: that is the monastic complex of San Sisto in Piacenza (Italy). This exhibition is part of a series of cultural events that will take place in different locations in the Emilia Romagna region in 2021, in the context of a culturally-based territorial development program: "Piacenza 2020/21 Crocevia di Culture" [2]. The entire work path will be explained starting from the description of the problem to be solved and the requirements to be met, the decisions made and solutions to the various problems

will be explained and justified. Finally, a design approach will be suggested and a software architecture will be provided to solve problems similar to the one presented here.

Several people worked on this project, on different aspects. The author of this thesis has fully developed the software part, therefore the point of view and what we will focus on in the next chapters will be the **software architecture** of the application itself and the game engine used.

Before starting it must be said that the goal of this thesis is to present to the gentle reader the results of a long process of education the author has undertook. It started with C++ which is a very large language consisting of multiple paradigms and techniques (some of them very interesting and obscure to the average programmer) which take a long time to be fully understood and mastered. It then continued by breaking down the various parts of Unreal Engine, which is a difficult endeavor by itself. To further increase the size of the problem, one must consider that the application produced must run on Magic Leap: an embedded device that is rather new to the market and its development kit was, at that time of development, not perfectly stable and documented. In summary, even if this thesis does not go into the details of the various problems encountered, it is important to mention that the development process was difficult and time consuming. Both because the lapse of time between building a program and then testing it on the device is very, very long and because of the various crashes and glitches encountered during development. It happened several times that a build that behaved or looked a certain way in Unreal Engine on the PC, behaved or looked differently on the device or did not run at all because some error occurred. Errors that cannot be debugged on the device directly but rather must be figured out using logs and stack trace printings.

2 Scope

2.1 Problem description

On 6 April 2020 (the event was postponed to mid 2021 due to the COVID-19 pandemic), on the occasion of the **500 years** that have passed since the **death** of the famous Italian painter and architect Raffaele Sanzio, we want to celebrate the event by showing his most important paintings to art enthusiasts. For the celebration, some rooms of the church dedicated to San Sisto in Piacenza are set up, rooms that are normally not accessible to guests. In these rooms, visitors will be able to take an informative and entertaining tour relating to Raphael's work, which most of all is linked to the place where the exhibition is set up: the painting "Sistine Madonna".

As part of the exhibition, two totems will be set up, on them two Magic Leap augmented reality devices will be made available to the participants. The spectators must wear one of the headsets and autonomously complete the **audiovisual experience** following the instructions given to them (both from the software application and assistants supervising the exhibition). There are no precise time limits to be met but it's required that the presentation must have a limited duration to avoid creating too long queues. It is also necessary to make it clear to the spectator what the end of the show is, in order to place the headset back in its position and leave space for the next visitor.

Our aim is to bring this work to life. We have to reproduce the painting in a virtual environment and allow viewers to admire it. The painting will be slightly animated enough to make the characters appear alive but without distorting the representation, it is still a painting and must appear as such. It is also important to keep in mind that we are

reproducing subjects of a religious nature that will be exhibited in a church so we must take into account the context. The reproduction must be respectful of the reproduced subject, that is, the painting itself and the religious subjects portrayed inside. We must as much as possible reproduce (and amplify, thanks to the more powerful mean of communication we have) the sensations conveyed by the painting. It is clear that there are many ways to fail in this endeavor and only a few to successfully create a visual experience that is both spectacular and respectful of the subject matter.

Figure 1: San Sisto courtyard

Figure 2: San Sisto nave

2.2 Exhibition subject

The Sistine Madonna is an **oil painting** made by the Italian artist Raffaello Sanzio. It was commissioned in 1512 by Pope Julius II to be exposed in the church San Sisto in Piacenza. It remained there for 240 years and was then sold to August III King of Poland around the mid-eighteenth century for an unprecedented amount of money. It is estimated that the value of the transaction was equivalent in value to about 90 kilograms of gold.

2.2.1 Second WW misfortunes

Figure 3: Leonid Rabinovich

The painting was almost **lost** during the Second World War after the bombardment of the city of Dresden, which was the place the painting was located it (the city was part of **Nazi Germany** and was located near the eastern front) by the Royal Air Force. One cannot forget to mention the man who put effort to find it back: lieutenant Leonid Rabinovich, a Ukrainian Jew who was also an art student and enthusiast.

It's common knowledge that during the war the Nazis hid goods like jewels, gold ingots and pieces of art so as not to be found by the allies. Leonid Rabinovich looked for these paintings and in particular the Sistine Madonna in a post apocalyptic scenario, in the basements of the crumbling deposits and museums of Dresden: a city that was destroyed

by bombs. The painting was found in 1945 in an abandoned and sealed railway tunnel, inside a wagon along with many others. Fortunately, also in this case the Nazis put in place all their efficient organization to keep the paintings safe from humidity. They were located in a double-walled railway carriage with sawdust filled cavities, there was also a heating and ventilation system to keep the humidity level low [3]. After finding them, the Soviets took the paintings to Moscow where they were exhibited for a short period of time at the Pushkin Museum. In 1956 it was returned back to the Gemäldegalerie in Dresden where it is still exhibited today and is one of their main attractions. [4]

Figure 4: Gemäldegalerie Alte Meister

2.2.2 Portrayed characters

Figure 5: Raphael, Sistine Madonna, oil painting 1513-1514 (Gemäldegalerie Alte Meister, Dresden)

The painting depicts six subjects on four different levels of depth. On the rear level there is the Virgin holding the Christ Child in her arms. The dress and the veil on her head appear to be moved by a

breeze, giving a feeling of dynamism to the picture. Closer to the viewer, Saint Sixtus is depicted: an elderly man with thinning white hair and a long, bristly beard. His rich robe also appears to be moved by the wind and seems to bend, in the back, with the clouds. The saint seems to address to Mary directly and point with his index finger towards the viewer, rising a feeling of involvement. On a different level of perspective is Saint Barbara, portrayed as a beautiful young woman with a placid and timid attitude giving a sense of calm and peace. She does not look at the apparition of Mary but turns her gaze downwards. On the front level there are two little angels. They look like winged children and place their hands on a horizontal windowsill as if to support themselves in order to participate at the apparition.

In addition to the main characters there are some fundamental details. The clouds in the background and under the characters suggest the celestial setting of the scene. It may not be evident at first but the clouds appear as intangible angelic faces, this effect is more accentuated in the left part of the painting, near the curtains. Two other details suggest to the observer the identity of the characters. One is the papal tiara in the lower left corner to testify the presence of Saint Sixtus who was pope with the name of Sixtus II in the years 257 – 258 AD. Another is behind Saint Barbara: it is the tower where she was locked up by her father to keep her away from the suitors who have been charmed by her beauty.

Finally, on the foreground, there are the green curtains. The ripples effectively convey the idea of softness of the material they are made of and make them seem as if they were to swing gently to settle slowly in a state of rest. The curtains act as a symbolic separator between the real world in which the viewer is and the divine world to which the represented scene belongs. These, as in a theater, seem to have opened to allow the observer to get closer and assist at the scene

of the apparition and could close at any moment, bringing him back to the real world.

2.3 Hardware state of the art

The intent of this exhibition is to allow the visitors to virtually admire a work of art that is physically located elsewhere. Until some time ago, the mandatory choice as a technological means to achieve this would have been a **virtual reality** device: the spectator wears the headset and finds itself into a completely reconstructed virtual world, wherever one looks, he will see some corner of the virtual world, from the floor to ceiling, from the closest to the most distant object. Everything must be created and placed in the scene. However, **augmented reality** devices have been recently introduced. As an advantage, now the user is not completely isolated anymore but have the illusion to see the **virtual objects** placed in the **real world** as if they were part of it. This is a huge advantage because now one can enjoy not only the virtual parts of the scene that he's observing but also the real part, adding further opportunities for artistic expression. Artists can now stage their artistic work in a **plausible context**. This also has a strong symbolic meaning: it communicates to the viewer that we are bring the painting back to the place where it was in origin, albeit virtually.

Figure 6: VR/AR support

The precursors of augmented reality are **mobile phones**, which inserted in a special support, can give the illusion that one is looking at virtual objects placed in space. The sensors on board do a decent job of keeping the virtual objects anchored in a specific location. The positioning of the object in space, however,

is far from perfect: they fluctuate depending on how the phone is handled and can shift over time in a relevant meaner. Orientation sensors are calibrated to measure changes in position rather than their absolute value. In normal phone applications, it is more meaningful to know how and by how much the device has been moved rather than knowing exactly where it is at the end of that movement.

Another fundamental limitation is the quality of real-time video playback and visual quality of the screen. Everyone agrees that today's phones have very high video shooting quality. This video quality, however, is made possible in part due to the quality of the optical sensors but mainly thanks to the post processing operations of the raw images coming from the camera. These images are not directly saved in a video stream but are first cleaned up, denoised and maybe improved with the use of various software filters and artificial intelligence. Finally compressed to save space. While all of these operations take place in real time at the camera's frame rate, they introduce a time offset. This offset must clearly be kept to a minimum maybe sacrificing some of the post processing operations, otherwise the user will perceive the delay resulting in a poor AR experience and maybe causing headache.

The visual quality of the screen is an issue as well. Normally the phone screens, even though highly defined, are not intended to be held at such a close distance to the eyes. All these reasons have made it necessary to research and develop new devices capable to overcome these limits. The two main contestants that meet all our requirements are Microsoft's HoloLens 2 and Magic Leap. The choice fell on the latter, the reason being dictated by lower costs rather than technological qualities.

2.3.1 Magic Leap

Magic Leap is an augmented reality device with very sophisticated features. It was developed by the startup Magic Leap, Inc. founded in 2010 based in Florida. The first device aimed at the public was released in 2018: Magic Leap One. The company, however, has recently undergone a major restructuring, focusing for now on the business slice of the market which is more easily able to afford the investment that the purchase of these products requires. During the year 2021, they are preparing to announce the second iteration of the headset.

Figure 7: Magic Leap

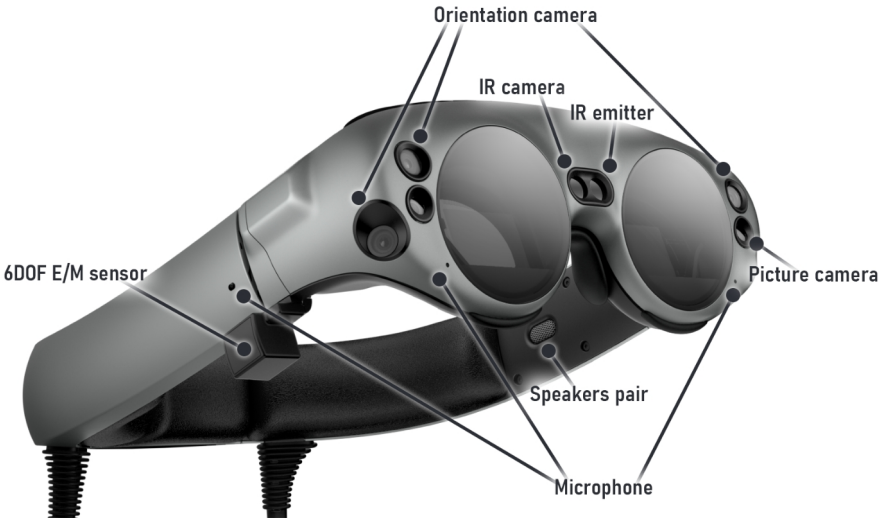


Figure 8: Magic Leap sensors

Let's see an overview: Magic Leap consists of two distinct parts connected by cable to each other (figure 7). One is the **headset** to be worn on the head. It contains the sensors, scanners, cameras and

speakers. The second is the **computation unit** which consists of all the circuitry necessary for operation and the battery. In addition to these two parts, a controller is included that communicates wirelessly with the device. The software that's running is a modified version of Android, an operating system called Lumin OS. It is designed around the concept of apps just like Android and feels similar to a certain extend. Apps can be installed, updated, uninstalled, started and stopped. The user can switch from one application to another and that application will be paused by the operating system allowing the user to resume later and continue from where it was.

We now come to the technological merits of the device. At Magic Leap they are keen to specify that this is not a VR device, nor is it the AR of smartphones, but rather it is a **spatial computer**. This definition, although not common, is actually appropriate. In fact, the device has a great ability to map the environment around it and to orient itself in the space. As one can see from the figure 8, Magic Leap has a long list of sensors that allow it to measure the space around. In fact it does not only orient itself but also creates a **virtual reconstruction** of the physical space, which will then be available to developers according to the API classes and functions of the graphic engine employed. Using the central sensors (figure 8), the headset projects a grid of points of infrared light, invisible to the human eye but visible to the nearby sensor, this information is processed by the device and used for the space reconstruction. It can also understand the space, this means that it will **recognize a location** that was previously mapped and maybe load the mapping from memory and update the map as more information is obtained. Specific locations in the space can be saved and later loaded by the application. Imagine that a user is placing a virtual object in a precise spot in the physical space, the application can decide to save that location and, when restarted, let the user find the virtual object in the exact same spot.

Magic Leap doesn't use screens like most VR devices, the first thing one notices is that the lenses are transparent. To visualize the images it uses a technology called Lightfield. Tiny image projectors point the light directly in the eyes of the user. There isn't much information on how this technology specifically works in Magic Leap, partly because it's still quite recent, partly because manufacturers keep it secret as much as possible. However, it is possible to see a general overview of the light field projector and its main parts in figure 9. Intuitively: the light coming from the projectors mixes with that coming from the outside giving a convincing sensation of looking at a virtual object physically positioned in the space.

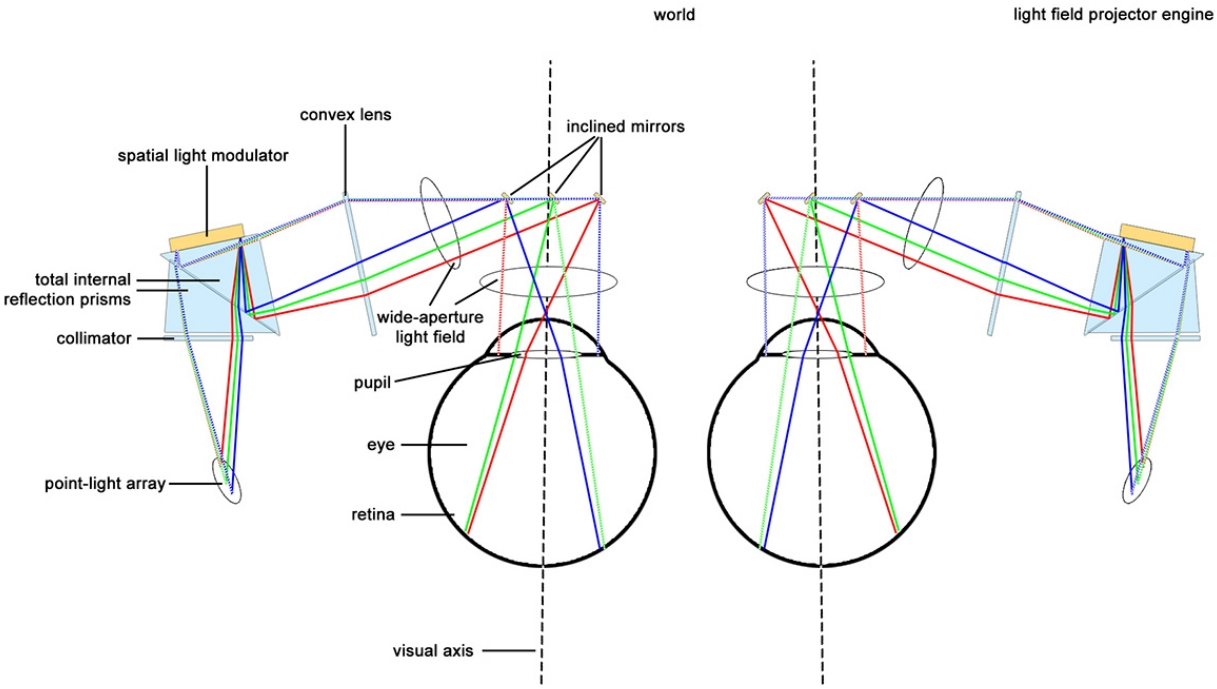


Figure 9: Lightfield schema

Let's talk now about the ways a user can interact with the application. The development kit provides the developer with components that map the position of the **hands** in the space. In fact, it not only detects the position of the hands but also provides the position of some key points of the hand (figure 10). This way it is possible to develop applications that track the hands in an extremely precise way,

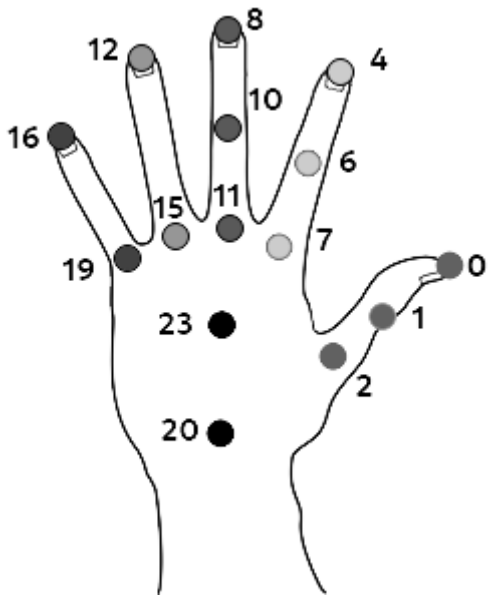


Figure 10: Magic Leap hand tracking key points

allowing the user to interact with the virtual objects in a natural and intuitive meaner. It can also recognize some specific **gestures** such as thumbs up, fist, and several others. As for the eyes, it is possible to know if the eye is opened or closed. It is possible to detect the direction of the user's gaze and to know in the application where the attention is directed. It could be useful, for example to synchronize events that must happen and must be noticed by the user telling how long to wait before showing an object in space or starting an animation and being sure that the user is observing it.

The **controller** is a more classic means of interaction, it has six degrees of freedom and its orientation is measured by the sensors on the headset. As shown in figure 8, on the device there is an electromagnetic sensor (6DOF E/M sensor) which measures the magnetic field generated by the controller. This way the positioning offset of the controller with respect to the headset can be calculated. Its absolute position in space is then calculated by summing the position of the headset. In addition to the orientation, the user can interact using a simple **button** (bumper), a **trigger** that measures a continuous value in the range 0-1 and a **touch pad**.

3 Development environment

After the long introduction and illustration of the background on which this work was carried out, as well as of the tools available, in this chapter the most interesting part of this thesis is finally discussed. Let's start by introducing the graphics engine used and the main concepts of the development kit.

3.1 Unreal Engine

Figure 11: Unreal Engine logo

Unreal Engine is a graphics engine developed by Epic Games, Inc. Like other graphics engines (CryEngine, Source Engine), UE was born from the development of a video game. This is Unreal, a video game from 1998 developed by the same company but which was called Epic MegaGames back then [5]. Over the years, the graphics engine has been developed at a steady pace and its potential applications just like its audience has increased. Now Unreal Engine is used in many fields, not only video game but also architecture visualization, film making, automotive, product design, engineering, simulation and others.

The popularity among fans has increased considerably since 2015 when Epic Game has changed its distribution policy allowing anyone to start using this engine for free and start paying 5% of the revenue when above a certain threshold (today that threshold is a million US dollars). Since then, UE's community of enthusiasts has continued to grow and today is the first alternative to Unity. Unlike Unity, Epic Games also

allows anyone to access the source code of the entire graphics engine for free. This is not an open source distribution, it is important to specify. The entire product remains under a proprietary license but is a particularly free proprietary one that places very few restrictions on the use of the product. Due to its technical qualities and permissive license, UE is a popular choice among both large game development studios and small independent developers.

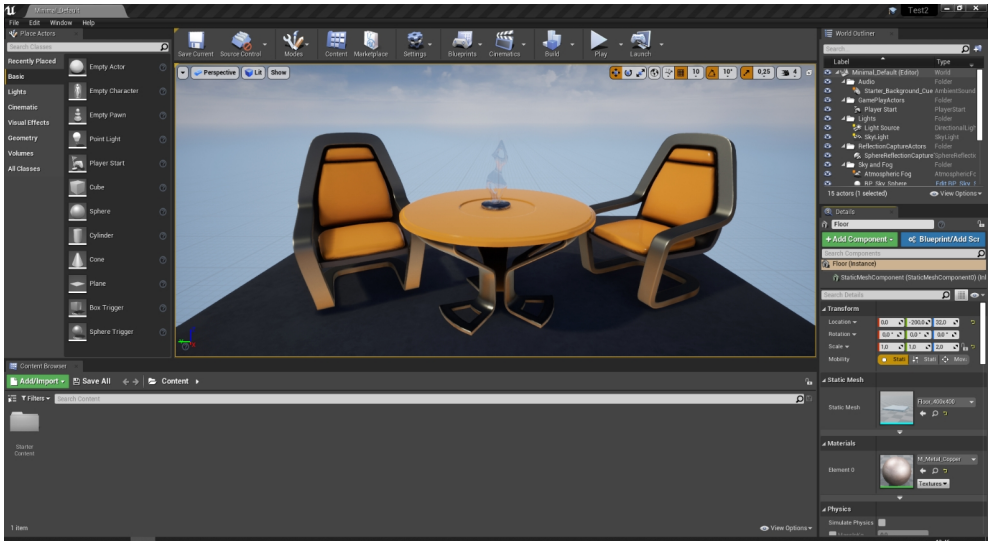


Figure 12: Unreal Engine 4.26 editor

In the course of the year 2021, a new version of Unreal Engine is expected to be released: version 5. The presentation video had great media coverage, because Epic Games announced a couple of features that could potentially be game changers for the industry: Lumen which is a new real time lighting system and Nanite which promises to be able to render meshes with an arbitrary number of polygons [6]. The latter being particularly interesting. Today's workflow of developers and 3D artists requires them to create and manage multiple versions of a certain 3D object. These versions are known as **LOD** (Level of Detail), their purpose is to optimize the size of the objects to decrease as much as possible the time required to be rendered in a frame. Showing increasingly complex scenes requires a great computational effort, an

effort that must be well directed towards the operations that can visually contribute to substantially improve the scene.

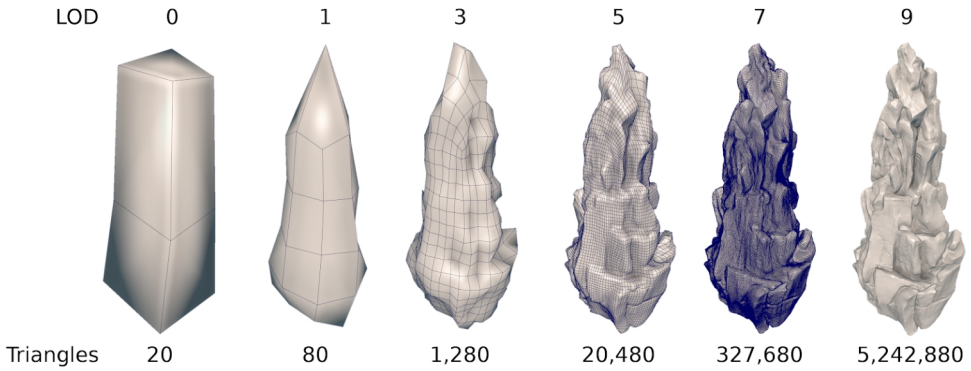


Figure 13: An object displayed at different levels of details

The way LOD works is as follows: the 3D artist prepares various versions of a mesh, those versions have an increasing degree of complexity. In fact, to reproduce an object in detail, a large number of polygons is required, this is particularly evident in the case of curved surfaces. The number of polygons per unit area must increase as the camera approaches an object to maintain the same level of visual quality (figure 13). This mesh density variation is implemented in the graphics engines as a replacement of the object with that of the next level of detail. The different levels of detail not only vary in the number of polygons but can go deeper than that and modify the structure of the object itself to make it more optimized such as eliminating some unnecessary parts, using a less detailed material or omitting shadows. Generally, creating these versions of a mesh takes time and doing it badly has a negative impact on the user experience. The gentle reader will certainly be familiar with the popping effect that some video games present especially when moving the camera at high speed: some objects seem like they appear on screen due to the replacement with more detailed versions of them. Nanite technology promises to handle this automatically, the 3D artist will simply need to load the mesh at the maximum level of detail and the graphics engine itself will

automatically, at runtime, scale the objects to the optimal level of detail with seamless transitions.

3.1.1 Basics

Here we talk about the fundamental concepts necessary to understand what will be illustrated later about software programming. Unreal Engine is written entirely in C++ but has specific **conventions** and **limitations**. Moreover it doesn't make use of the standard template library (STL). Most of the classes and templates in the standard library are implemented with similar names in Unreal hence experienced C++ developers should be able to find all the interfaces they are already familiar with.

In Unreal Engine one can use any valid C++ program but in order to interact with the editor the code has to follow some strict conventions. These conventions were necessary to allow Epic Games developers to create a massive **reflection system**, feature that is missing in C++ and that was necessary in order to implement all the functionalities offered by the editor. Unreal must in fact examine, introspect and modify the structure of the classes, this ability is called reflection in computer science. The reflection is beneficial because, on the one hand, it allows the engine to instantiate compiled C++ classes and modify the data of the objects directly from the editor, on the other hand it allows the functioning of Blueprint: a visual scripting language very well integrated within the Unreal Engine framework.

One of the main selling points of Unreal is that it has made 3D programming accessible even to non-developers. By simply using Blueprint one can do almost anything that can do with C++ and much faster because it doesn't have to compile the code (which is a very time-consuming operation). For this reason, it is much easier to prototype functionality with Blueprint. However, it should be noted that

Blueprint has generally less performance than C++, especially in CPU intensive operations. There is a trade-off to be made here. This project mainly uses C++ but still Blueprint is also used where it makes sense and is more convenient. [7]

There are two types of objects in Unreal Engine: **classes** and **structures**. These make use of the same two keywords as plain C++ but there are some considerations to be made. A class and a struct is treated the same way in plain C++ (the only difference is the default access specifier which is public in the case of struct and private in the case of class), in Unreal the two concepts are given semantic value. An object of a struct in Unreal is a simple value allocated on the stack. This can be a member variable of another struct or class, it can be the value argument of a function and in general it will normally be copied. The best way to think of a struct is as a **data container**. Conversely, a class is a more complex data type that is also handled differently by the engine. Classes, for example, often make use of **inheritance** and rely on virtual methods to implement polymorphism. The objects of classes are allocated in memory and their existence is not limited by the code scope they are in. It is responsibility of the **garbage collector** to manage objects lifetime and deallocate the memory when it is not referenced anymore by any property. In fact the garbage collectors works alongside with the reflection system to keep track of what memory is referenced by what objects. In Unreal an excellent job has been done in simplifying the memory management that is normally required to a plain C++ developer.

Here is an example of how to declare a class idiomatically in Unreal. Structures are declared in a similar way with minor differences.

```
1.  #include "CoreMinimal.h"
2.  #include "MyObject.generated.h"
3.
4.  UCLASS(Blueprintable)
5.  class TEST_API UMyObject : public UObject {
6.      GENERATED_BODY()
7.
8.  protected:
9.      UPROPERTY(EditAnywhere)
10.     int SomeInt;
11.
12.     UPROPERTY(EditAnywhere)
13.     TSubclassOf<AActor> SomeActor;
14.
15.     UPROPERTY(EditAnywhere, BlueprintSetter = SetSomeString)
16.     FString SomeString;
17.
18.     int UnknownInt;
19.
20. public:
21.     UMyObject();
22.
23.     UFUNCTION(BlueprintCallable)
24.     void SetSomeString(const FString& String);
25. };
```

Code 1: Class declaration in Unreal Engine

One detail to notice is that Unreal has a strict convention when naming classes. The class `MyObject` (code 1) is proceeded by the letter "U", similarly structures begin with the letter "F". There are additional elements to take into consideration though, they are mostly macros used to register the various parts within the reflection system of Unreal Engine. For example, the class itself is registered using the macro `UCLASS()`. The argument `Blueprintable` tells the system that this

class can be used as a parent to create Blueprint classes. There are many other specifiers that can be used as arguments (not only in `UCLASS()` but also in the other macros), they will not be further discussed though. It is of course possible to avoid registering member variables and functions. Valid plain C++ code will work correctly in the context of the engine. Not registered members, however, cannot be detected by the editor therefore it won't be possible to use then within Blueprint. See for example the member variable `UnknownInt`, not being preceded by the macro `UPROPERTY()` makes Unreal unaware of its existence therefore it will not appear and will not be available in the context of the editor (figure 14). It can still be used as usual in the context of C++ though. A noteworthy detail is the setter for the `SomeString` property. As one can see, the integration of Blueprint with C++ goes quite deep allowing the developer to specify methods defined in C++ that will be used when the property at hand will be assigned in Blueprint. The methods that can be used from Blueprint are preceded by the macro `UFUNCTION()` with the argument `BlueprintCallable` to inform the engine about that.

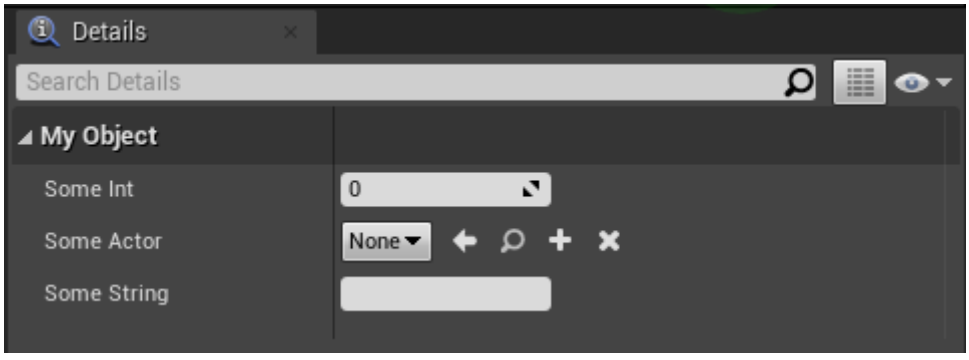


Figure 14: MyObject class properties as shown by the editor

The macros are just part of the story. The reflection system of UE is made possible thanks to the **Unread Header Tool**. UHT is invoked before invoking the normal C++ compiler. It will parse the header files (.h) to get the metadata and generate the custom code

that implements various type system related features. It is a tool integrated in the build process that generates an additional header file (ClassName.generated.h) that must be included as the last include directive in every class' header file [8].

To workaround the lack of introspection of C++, other frameworks have adopted solutions similar to the one presented here: on the one hand, macros are used to record the various parts that make up the classes, on the other hand, an additional compilation step is introduced in order to run tools specifically designed to analyze the classes and generate additional code. To make an example which is similar, the Qt framework uses macros in a similar way as seen in Unreal and a tool called MOC (Meta Object Compiler) integrated in the building system to gather information about classes and write the boilerplate code needed to implement the additional features [9].

All the classes used in Unreal inherit from the class `UObject`. The Unreal developers here wanted to conform to the trend that most programming languages follow (Java, JavaScript, C#, Kotlin and many others). In object-oriented languages usually there is a top type (also called universal base class): a class usually called Object from which all the other classes implicitly inherit methods and attributes. Although C++ is object oriented, this paradigm is only one of the possible paradigms that can be used with this language. All types in C++ are in their own right, conceptually separated from all the other classes when there's no inheritance branch in common. Any code will be valid if, given a certain value, it will happen that it has all the methods and offers all the features that will be required by the code that uses that value. This approach is known as duck typing and is used for example by C++ with templates (at compile time) or by PHP (at runtime): another language that just so happens to lack the existence of a top type. An important limitation to keep in mind when writing classes for the Unreal Engine is that a class can inherit from a single other class. In C++

there is multiple inheritance but if a class is made to be compliant with Unreal's type system then it must inherit from a single class or from the class Object itself and this behavior is enforced by the UHT. It can, however, implement multiple interfaces. There are specific conventions to declare interfaces in Unreal, they are still C++ classes properly decorated with macros. [10]

3.1.2 Building blocks

Now let's see what are the main parts to keep in mind when programming in Unreal and how they interact with each other. Although over time Unreal Engine has been constantly expanded, its origin as a game engine is evident from the terminology used in various classes and functions throughout the API. There are names like `UGameEngine`, `AGameStateBase`, `FGameplayTag`, `AActor::BeginPlay` about everywhere.

Vectors representation

In any 3D graphics software, a concern is to develop an efficient system to represent vectors because it will need to handle a very large amount of them. Any point in space is characterized by a triple to represent the location. When dealing with objects in space, besides the location, one must also take into account the rotation because, as the educated gentle readers will remember from school, an object in a three-dimensional space has six degrees of freedom. This means that 6 different numerical values are needed to precisely place it in space. Moreover an object can be scaled along the three axes. The union of the concepts listed above is called transform and it managed by the class `FTransform` in Unreal Engine. [11]

Inside a transform object, one will find a `FVector` for the location: this class is nothing more than a container of 3 float values

(one for each axis of the three dimensions XYZ). There is an `FQuat` object for the rotation. Although it would be possible to represent rotations with a vector of dimension 3 (rotations with respect to the three main axes: yaw, pitch, and roll), in order to avoid ambiguity, rotations are represented by quatermions, which use 4 values. The rotator is, in fact, affected by the gimbal lock problem which is talked about in the next paragraph. Finally, the scale operation is represented by an `FVector`, just like the location.

Gimbal lock problem

Figure 15: The actor CameraActor has a locked gimbal

The problem with the **gimbal lock** is the loss of a degree of freedom if it so happens that two axes of the three rotation axis were to position themselves in a parallel configuration. Let's start with the basics. An object has 3 axes of rotation these are called yaw, pitch, and roll and in Unreal Engine they are respectively the Z (upper), Y (right) and X (front) axes. Thinking about the gimbal, these axes lie within a hierarchy. Rotating the outermost axis will also affect those at the

lower levels in the hierarchy and leave unchanged those at the higher levels. The gimbal lock problem occurs when the intermediate axis of the hierarchy is rotated 90° so that the other two axes (lower and upper) line up. Seeing the problem in action is very easy in Unreal Engine, in this context, the intermediate axis is the Y (pitch) axis, therefore it is sufficient to rotate any actor by 90° around that axis. Subsequently one can realize that varying the rotation around the X axis and around the Z axis produces the same type of rotation (figure 15). However, setting Y as an intermediate axis is a good compromise. This way one will risk having to deal with the problem only when an object is aligned along the vertical axis (looks exactly up or down) possibility that it is not very common (at least when compared with the other possible alternatives). [12]

The problem of the gimbal lock can be solved by using a 4-value system (`FQuat` in Unreal: quatermions) instead of the 3-value (`FRotator`). The 3-value rotation system is made available because it is much easier for animators to reason about. Rotators have an intuitive meaning that allows one to mentally understand what kind of pose a rotator value corresponds to. In the case of quatermions this mental operation is much more difficult, they have more of a mathematical meaning than an intuitive one.

World

In Unreal Engine, what is shown on the screen exists in the game world. This is represented by the `UWorld` class which is the top level concept representing a map or a sandbox. If one wants to perform operations such as adding a new level, or spawning a new character this is the class to refer to.

Actor

An actor is an object that can be physically placed in the world. Any actor inherits from the `AActor` class. One can recognize actors in the code because their classes names begin with the letter "A". The actors by default do not have physical qualities, the best way to think about actors is like entities that perform some actions or reveal themselves in some ways that depend on the components they have (see Components).

An actor has a precisely defined **life cycle**: it comes into existence when spawned programmatically or because it is part of the world. Later it can be destroyed programmatically or by the engine itself because the game or the current level ended. In general when programming using large frameworks like Unreal Engine that make available complex abstract environments that are managed in every single detail by the underlying software architecture (thus being opaque from the outside), a series as entry points are provided to software developers. One can attach some code to those entry points in order to perform custom actions and interact with the environment according to the needs of the specific application, in other words to script the game logic. Concerning actors, those entry points are provided as virtual methods that can be overridden by the sub classes representing the concrete actors. At the beginning, the engine calls the method `AActor::BeginPlay`. This is a very common method to carry out initialization operations. Similarly the method `AActor::EndPlay` is the opposite that carries out eventual cleanup operations.

During their lifetime, actors can be "ticked": they can run a piece of code at regular intervals, most commonly once per frame. This code is used to progressively update the 3D scene as the user interacts with the world and more actions are happening. The piece of code written inside the method `AActor::Tick` (as usual overridden by sub

classes) takes as argument a float value which is the time elapsed since the last tick (in seconds). Ticking frequency is dependent on the frame rate the rendering is running at. This frequency is obviously affected by the load the engine is sustaining in any instant: how many actors are on the field of view, how complex are their geometries, how many sources of lights that generate shadows are there and so on. One should keep in mind that the game logic is running in a single threaded fashion to avoid overloading the developer with all the problems that multi threading brings to the table. This doesn't prevent one from writing code that runs in parallel and in fact the API provides several facilities to do this easily. Still, operations carried out in the `Tick` method must be very well paid attention to because any lagging that may happen in this method has a net negative impact on the frame rate, even if it happens in the most remote and irrelevant actor.

Components

Actors are nothing more the empty shells. They perform actions or reveal themselves by using components. One should think about components as building blocks that can be added to actors in order to confer specific qualities. An actor has to produce a sound? `UAudioComponent` is what is needed. An actor has to show a cube or some other object? `UMeshComponent` will do that. The best way to think about components is in terms of reusable behaviors.

Every component is a subclass of `UActorComponent`. That class defines the main features of components that the classes inheriting will eventually override. Like the actors, the components are given the chance to attach code to the main entry points that are of interest to the application logic (like `UActorComponent::BeginPlay`, `UActorComponent::EndPlay`). Also, similarly to actors, the components can be ticked.

A fundamental component that is used extensively is `USceneComponent`. This component has a transform which means it represents a specific position into space and any component that needs the same functionality will inherit from this class. Of fundamental importance is the ability to attach scene components to other scene components. Attaching a scene component to another component means exactly what one intuitively thinks of physical object attachment: for example the neck is attached to the torso, the head is attached to the neck, the hair is attached to the head and so on. Attaching an object to another is an operation frequently done in games. Image the character grabbing a gun, that gun will temporarily be attached to the hand of the character by means of scene components attachments.

UClass

Each C++ or Blueprint class is represented by an object of type `UClass` (itself a `UObject`). This object is unique in the whole program and cannot be instantiated directly (Singleton pattern). Thanks to `UClass`, the Unreal Engine developers have aligned the capacities of C++ and Blueprint. It is in fact thanks to the information contained in this class that it is possible to access using C++ the properties and call the methods defined in Blueprint and vice versa. It is also thanks to the `UClass` that it is possible to declare Blueprint classes that inherit from C++ classes. The `UClass` has inside all the information that characterizes a class such as properties, functions, parent class, implemented interfaces, etc. Moreover this class is often used as a token in various methods, such as `UWorld::SpawnActor` this means that both using C++ and Blueprint will be possible to spawn an actor indifferently defined in C++ or Blueprint.

4 Realization

The presentation of the painting takes place in the following way. Initially, the user wears the device and finds himself inside a virtual apse. The application has to position this apse in a precise real location that must be set beforehand. To do this, the pinning function of the device will be used to recognize the precise point in which the apse must be positioned with respect to the position of the physical space.

Then the show begins. The painting initially appears with the curtains closed so it is not possible to see the characters from the position in which the viewer is (one cannot physically get behind the curtains). At this point a kinematic sequence begins that sets a high emissivity value of the material of the clouds then begins the animation of the curtains opening. As soon as the curtain is slightly moved away, a very strong ray of light dazzles the viewer. This light increases in intensity as the curtain opens, allowing one to see more of the light emitted by the clouds. This opening sequence lasts a few seconds accompanied by celestial music which together with the strong light effect is designed to create a sensation of wonder and amazement in the viewer. In front of the light, the figures appear as dark silhouettes whose specific traits are not visible. At the end of the opening sequence, the light dims slowly and the music fades away leaving the curtains open and the painting completely dark. At this point the curtains light up and the characters begin to light up one at a time, accompanied by a narrating voice that explains the role of each character within the painting and some interesting facts about them. At the end, the spectator has a few seconds to observe the painting completely illuminated and then the curtains closing sequence begins, suggesting that the show is over.

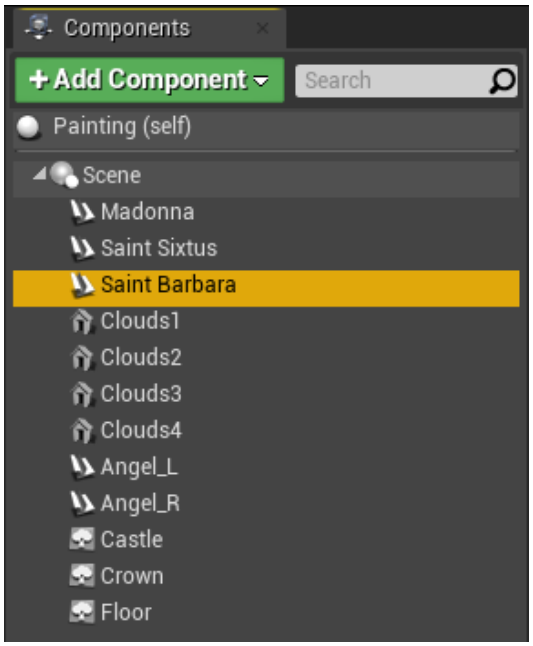


Figure 16: Painting actor

4.1 The painting

The painting is an actor which is placed in the scene in the corresponding position. This actor contains as components the various elements that make up the painting (see Actor and Components). The composition was created using Blueprint because it offers an immediate visual feedback (figure 16).

The characters are animated through a series of images to be shown in sequence. This type of animation is not a concept limited to Unreal Engine but exists in various other animation and graphics software. In this case they are managed through the tool Paper 2D Flipbooks, therefore each character is a Paper Flipbook Component.

Figure 17: Rendered painting

The additional details of the painting, namely the papal tiara, the tower of Saint Barbara and the windowsill on which the angels rest, are inserted as individual frames, exactly like the frames of the characters' animations: they are Paper Sprite Components.

Finally, the clouds are static meshes, in particular they are 2D planes arranged on different levels. A material has been applied to these planes which make them look like moving clouds. It is not possible to convey the idea of the type of animation through the images shown on these pages but the material developed (see Clouds material) does a great job to visually replicate the movement of the clouds. The gentle reader can take the words for it or can try to replicate the material within the editor (code 2).

The most careful gentle reader may have noticed that the curtains are missing from the components of the Painting. This is because the curtains are made an actor themselves so that their animation can be started and stopped independently on the animation of the characters. There are a pair of actors, one has a flip-book that contains the animation of the opening of the curtains (figure 18), the other one is a loop animation of the curtains already opened gently swinging as if moved by a soft wind.

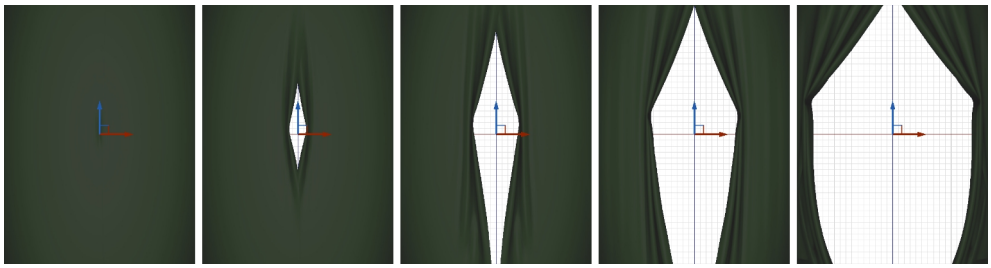
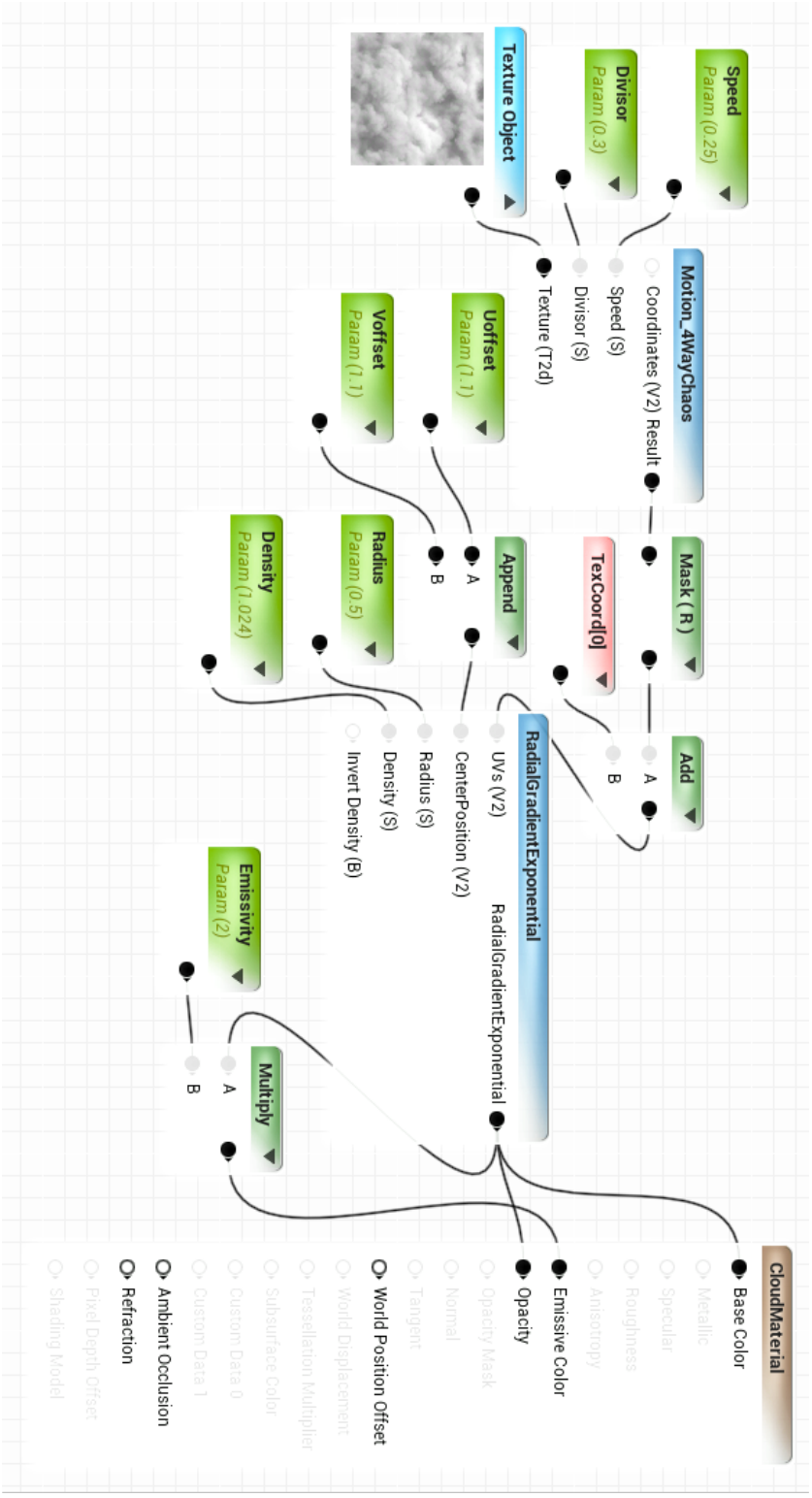


Figure 18: Curtain opening animation

4.1.1 Clouds material



Code 2: Clouds material Blueprint

The clouds are made using a material designed to be applied to flat 2D surfaces and viewed from the front. This material was made in Blueprint because it really makes much more sense and it's much more comfortable to use Blueprint to make materials rather than C++. The entire program can be seen on the previous page. The main function here is that of the node `Motion_4WayChaos` which does what it promises: given a static texture (`Texture Object`), it chaotically moves its various parts resembling the movement of a cloud. The second relevant function in this material is the `RadialGradientExponential` node which exponentially decreases the density of the cloud when moving away from the center. All the parameters that can be adjusted are self explanatory. There are have values like `Speed` (which adjusts the speed of the internal movement of the cloud), `Uoffset` and `Voffset` (which move the center of the cloud along the two directions) and others (code 2).

4.2 The virtual environment

The user is inside a 3D modeled **virtual apse** (figure 20), itself an actor with a static mesh component having materials that, together with the soft lighting, give it a discreet appearance so as not to distract the user from the focal point that should be the painting. The floor has been voluntarily left empty

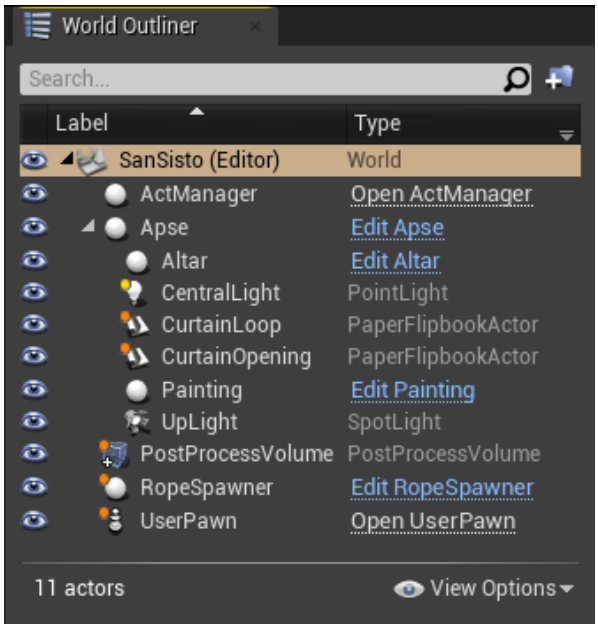


Figure 19: Elements of the 3D scene

as to allow the user to see its feet and where it steps. The user's window on the virtual world is a camera that moves in the virtual space following the movement of the headset in the physical space. This camera is a component of the actor called `UserPawn`. Besides the camera, this actor also manages the controller and shows a red laser beam as a virtual pointer. The user pawn's class is a subclass of `APawn` (itself a subclass of `AActor`) which is nothing more than a type of actor that can be possessed wither by a human user or by the AI. An overview of the elements present in the world can be seen in figure 19 while the scene rendered is shown in figure 20. The names are self-explanatory: there is the picture actor, the marble altar, two flip-books which are respectively the opening/closing animation of the curtains and movement loop of the opened curtains. There are some lights, a post-processing volume which adds some visual settings and the act manager that will be explained afterwards.

Figure 20: 3D Apse, the painting and a marble altar

4.3 Architectural design

Now let's talk about software **architecture**. It has been developed not only to implement this project but more widely for research purposes to have a versatile tool that can be used in the case of similar projects for artistic exhibitions or maybe even video games or others purposes.

4.3.1 Goals

The aim to achieve is the realization of a **finite state machine** that uses elements understood by Unreal Engine (thus allowing one to modify these elements directly from the editor). We want to promote code reuse as much as possible by specializing only the parts that differ and allowing equal parts to be kept in common.

Finite state machine

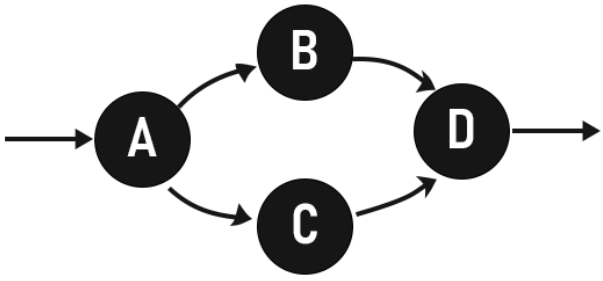


Figure 21: Finite-state machine example

A finite state machine is a very general **mathematical model** that consist of **states** and **transitions**. It can be used to study the evolution of a system. Graphically a state is represented by a circle (figure 21) and identifies a particular configuration of the system, configuration that can be distinguished by all the others, a specific arrangement of the founding elements, a state precisely. In this case, a state will be an

exact moment of the show. For example, while trying to locate the apse in the physical space, that will be a state. While the curtains are opening, that is a state in which a cinematic sequence is playing. It will become clearer to the gentle reader what a state should represent, in the following paragraphs. Systems evolution is modeled as transitions between states. The system can progress from one state to another according to its internal rules of operation. This evolution is graphically an arrow (figure 22).

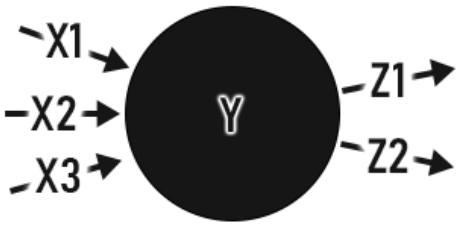


Figure 22: State transitions

Although the model just described is very simple, it is also very powerful. It allows the adoption of a common language to describe very different systems and is useful to represent the type of actions that are commonly performed in game engines. [13]

What is of interest to the developer is to intercept these state transitions in order to express the logic of the application. In practice, this means that one is interested in being able to attach code in the main parts of these transitions, that is, when passing from one state to another (distinguishing according to the source state), inside the state itself and when moving to a destination state (again, distinguishing according to the specific state). With reference to the figure 22, if one calls Xi the transition from a previous state i to the current one, Y the current state, and Zj the transition from the current state to the next state j, then one should be able to execute specific code in each of these parts.

The architecture must be designed in such a way as to allow high **reuse of the code**, after all this is a purpose of the software architectures: to define very precise rules to add new classes and functions without duplicating or further tangling the code. To

implement the finite state machine model, C++ will be used, which is an object language therefore it is natural to implement states as classes. An important feature of object-oriented languages is the **inheritance** between classes hence this powerful mechanism should be made use of. Having the ability to inherit between states somehow goes beyond the narrow finite state machine model introduced above but it proves to be very useful and convenient. A state should be able to inherit from another state and decide to rewrite only the behaviors that differ while keeping all the remaining behaviors unchanged. In relation to the figure 22, imagine having a class `Y` available, someone wants to create another class (type of state) that is very similar to `Y` but has a minor requirement that makes it incompatible. For example, the code in point `X1` must be change while the others remain unchanged. The new state can very well inherit from `Y` and override the method where that code for `X1` is.

As an additional requirement, the developer should be able to define transition methods between states that take into account the hierarchy. Given a couple of states whose specific type is unknown at compile time, the correct method between the overloads available (the one having the argument that most closely matches the actual type) will be called depending on the **dynamic types** of the states.

4.3.2 Implementation

Now let's see what an architecture that meets the requirements above could look like. In this context, a state is a game act. One wants stick with the terminology familiar to the developers who use Unreal Engine which was initially intended for the development of games so this is reflected in the names of many classes, therefore the "game" part. Borrowing from the theatrical terminology it was decided to divide the show into acts, therefore the "act" part of the name.

Game act

A game act is an actor thus it can do everything that actors can do: it can be spawned in a level and begin play, it can be ticked, etc (see Actor for additional information). Furthermore a game act, and here again a theatrical terminology is used, can be staged. When staged it effectively starts to perform its actions and reveal itself. Just like a pair of virtual methods remark the begin and end play of an actor, similarly, a pair of methods remark the begin and end of a game act: `AGameAct::GameActBegin`, `AGameAct::GameActEnd` (remember that actors names begin with the letter "A") (code 3).

The game acts are managed by the act manager. This is the class to refer to if one wants to stage a new act. There are methods like `AActManager::StageGameActClass` (which takes a `UClass` object as argument and stages an instance the corresponding game act) or `AActManager::StageGameActObject` which directly takes an already instantiated game act object. To cause a transition from one act to another, the developer can use `AActManager::MoveToGameActFollowerOf` which, given the current game act, makes this one finish and move on to the next one.

```
1.  UCLASS(Config = Game)
2.  class SANSISTO_API AGameAct : public AActor {
3.      GENERATED_BODY()
4.      GENERATED_ACT_BODY()
5.
6.      protected:
7.      UPROPERTY(EditAnywhere, Config)
8.      TSubclassOf<AGameAct> NextGameActClass;
9.
10.     protected:
11.     virtual void BeginPlay() override;
12.     virtual void EndPlay(const EEndPlayReason::Type
          EndPlayReason) override;
13.
14.     virtual void GameActBegin();
15.     virtual void GameActEnd();
16.
17.     public:
18.     UFUNCTION(BlueprintCallable)
19.     virtual void DispatchTransitionFromGameAct(AGameAct* Act);
20.     UFUNCTION(BlueprintCallable)
21.     virtual void DispatchTransitionToGameAct(AGameAct* Act);
22.
23.     void TransitionFromGameAct(AGameAct* Previous);
24.     void TransitionToGameAct(AGameAct* NextAct);
25. };
```

Code 3: AGameAct class

Each game act knows the class of its successor. When an act ends it will call the act manager to move on to the next act, providing as argument the `UClass` object of the next one. Of course the act that follows can be modified by the components of the act. The act manager takes care to initiate a transition between acts. Each act involved in a transitions has the ability to execute specific code when moving from another act towards itself and when it has finished and moves on, to the next act. These pair of methods are respectively: `AGameAct::TransitionFromGameAct` and

`AGameAct::TransitionToGameAct`. Upon seeing the `AGameAct` class (code 3) declaration, the careful gentle reader will have noticed that these methods are not declared virtual. One might wonder how subclasses can rewrite such methods. Well here is exactly where the most creative and interesting part of this research lies.

```
1.  UCLASS(Config = Game)
2.  class SANSISTO_API AActManager : public AActor {
3.      GENERATED_BODY()
4.
5.      protected:
6.      virtual void BeginPlay() override;
7.
8.      public:
9.      UFUNCTION(BlueprintCallable)
10.     void StageInitialGameAct();
11.
12.     UFUNCTION(BlueprintCallable)
13.     AGameAct* StageGameActClass(UClass* ActClass);
14.
15.     UFUNCTION(BlueprintCallable)
16.     void StageGameActObject(AGameAct* GameAct);
17.
18.     UFUNCTION(BlueprintCallable)
19.     void UnstageGameActObject(AGameAct* GameAct);
20.
21.     UFUNCTION(BlueprintCallable)
22.     AGameAct* MoveToGameActFollowerOf(AGameAct* OldGameAct);
23. };
```

Code 4: AActManager class

Let's start with order. Each subclass of `AGameAct` can **intercept** the transition in the two methods (transition to and from another game act). The developer must be able to capture the most specific transition possible by declaring an overload of such methods that takes as argument a game act of some type, relying on the fact that the this method will be called for all the transitions that involves that exact type

or a subclass type when no more specific method overload exist. It's easier to show this concept using a practical example.

Let's imagine that the game acts are geographic locations. There are acts such as Italy (code 5), from which acts such as Piedmont inherit. An act like Turin (code 6) will inherit from Piedmont. Now imagine that a global pandemic has broken out and that travel between geographical locations must be limited. Let's create the act France (code 7) and the game act Lyon (code 8) which inherits from France. Imagine that at this moment to limit the infections, travels from Piedmont to France are forbidden but one wants a strategic exception for those who go from Turin to Lyon due to high economical value of trade on a high speed railway that connects those two locations. To implement those rules the developer must declare two methods: one in France that takes as argument a state Piedmont and implements the travel ban logic (code 7, line 13), the second one in Lyon that takes as argument a state Turin and implements the travel permission logic (code 8, line 13).

There is some additional code to add to a game act class. Unlike a plain actor, a game act has a pair of macros `GENERATED_ACT_BODY()` and `GENERATED_ACT_DISPATCH_DECLARE()`. The use of macros is generally an inelegant practice but in Unreal Engine, those are usually employed extensively. The first one defines the method `AGameAct::GetIndex` which is used to enumerate all the game acts. The second one must be used in conjunction with the macro `GENERATED_ACT_DISPATCH_DEFINE()` in a code (.cpp) file. More details about those macros will follow in the news paragraphs.

```
#include "GameActs/France.h"
#include "GameActs/GameAct.h"
#include "Italy.generated.h"

UCLASS()
class SANSISTO_API AItaly : public AGameAct {
    GENERATED_BODY()
    GENERATED_ACT_BODY()
    GENERATED_ACT_DISPATCH_DECLARE()

    protected:
    virtual void GameActBegin() override {
        // Moving from "Italy" to "France" immediately
        this->NextGameActClass = AFrance::StaticClass();
        this->ScheduleMoveToFollowingGameAct();
    }
};
```

Code 5: Italy game act

```
#include "GameACts/Lyon.h"
#include "GameActs/Piedmont.h"
#include "Turin.generated.h"

UCLASS()
class SANSISTO_API ATurin : public APiedmont {
    GENERATED_BODY()
    GENERATED_ACT_BODY()
    GENERATED_ACT_DISPATCH_DECLARE()

    protected:
    virtual void GameActBegin() override {
        // Moving from "Turin" to "Lyon" immediately
        this->NextGameActClass = ALyon::StaticClass();
        this->ScheduleMoveToFollowingGameAct();
    }
};
```

Code 6: Turin game act

```
#include "GameActs/GameAct.h"
#include "France.generated.h"

class APiedmont;
UCLASS()
class SANSISTO_API AFrance : public AGameAct {
    GENERATED_BODY()
    GENERATED_ACT_BODY()
    GENERATED_ACT_DISPATCH_DECLARE()

    public:
    using AGameAct::TransitionFromGameAct;
    void TransitionFromGameAct(APiedmont* Previous) {
        UE_LOG(LogTemp, Display, TEXT("Piedmont -> France:
         DISALLOW!"));
    }
};
```

Code 7: France game act

```
#include "GameActs/France.h"
#include "Lyon.generated.h"

class ATurin;
UCLASS()
class SANSISTO_API ALyon : public AFrance {
    GENERATED_BODY()
    GENERATED_ACT_BODY()
    GENERATED_ACT_DISPATCH_DECLARE()

    public:
    using AFrance::TransitionFromGameAct;
    void TransitionFromGameAct(ATurin* Previous) {
        UE_LOG(LogTemp, Display, TEXT("Turin -> Lyon: ALLOW"));
    }
};
```

Code 8: Lyon game act

Dispatching mechanism

Summarizing the previous paragraphs, we want to be able to create an architecture that allows one to intercept a transition between two game acts by declaring methods that are at the same time overridable by subclasses, overloadable for different subclasses of `AGameAct` and that also support type substitution.

Let's see what happens when a state transition is requested. The `AActorManager::MoveToGameActFollowerOf` method is called and an argument of type `AGameAct` is provided, this is the origin game act. Within this method, the game act following the origin one is fetched. At this point there are two objects of type `AGameAct`: the origin and destination one. The task is to decide what is the (dynamic) type of these two objects, down cast to the exact type, and then call the transition methods providing the exact types in order to execute the appropriate overload among those available. This is in fact a common requirement for many software architectures that have to perform different operations on multiple types of objects, a topos of computer science literature, in a manner of speaking. It is called **multiple dynamic dispatch**, in this case it is a **double dispatch** (because there are two objects involved). Some languages solve this problem at a design level (the feature is sometimes called multi-methods), examples of those being: C#, Common Lisp, Julia. Unfortunately C++ is not one of them. For those languages that do not support this features, a good design pattern is usually Visitor. [14]

Visitor pattern

Let's see how visitor pattern achieves the double dynamic dispatching. There are two types of concepts in this design pattern: **visitors** and **elements**. The former are algorithms or operations, which must act on the latter. For different elements, the same visitor will

perform different operations as if it was an optimized version of some algorithm. Let's think for example of data compression. When compressing an audio file, one will use a different algorithm than when compressing an image file. In this case the concept of visitor is a compression algorithm while an element is a binary file. The visitor subclasses will be different algorithms and elements will be specific types of file (text, image, audio, etc) (figure 23). A visitor can also decide to not implement the algorithm for some types (for example `CompressionB` does not deal with Text files, just Image and Audio). Given a pair of `Visitor* visitor` and `Element* element`, one wants the right algorithm to run without having to worry about the specific type of visitor and element.

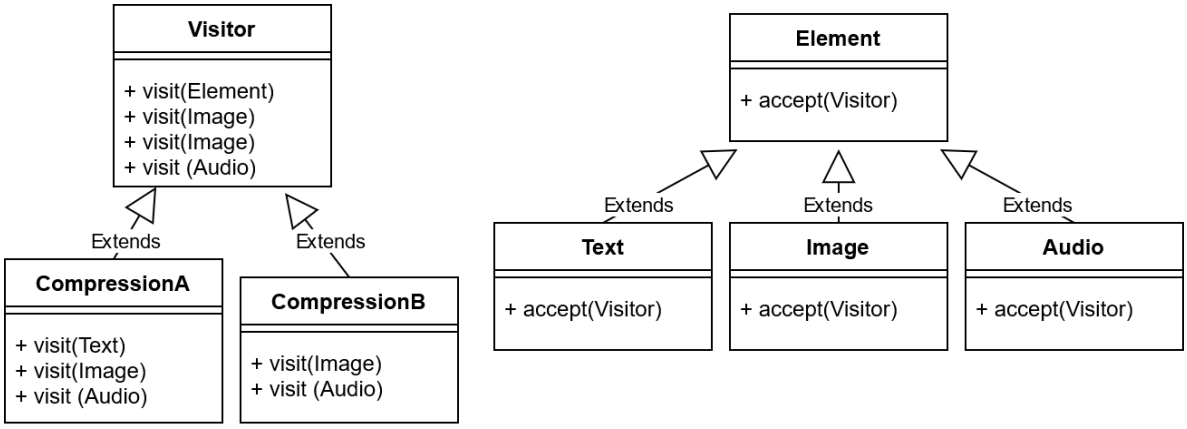


Figure 23: Visitor design pattern UML diagram

To achieve the dispatching the first method to be called is `Element::accept` (`element->accept(visitor)`). Each subclass of `Element` will override that method and define it as shown in code 9: the appropriate `Visitor::visit` method is called. This is the first step of the double dispatch, the precise element was identified because the method is overridden in each (concrete) subclass of class `Element`. The visitors can decide which methods to override from the parent class as to perform the specific operations associated with that element. This is the second step of dispatch where the visitor is identified (code 10). [14]

```
class Element {
    public:
    virtual void accept(Visitor* dispatcher) = 0;
};
class Image : public Element {
    public:
    void accept(Visitor* visitor) override {
        visitor->visit(this);
    }
};
```

Code 9: Visitor's pattern element class

```
class Visitor {
    public:
    virtual void visit(Text* file) {
        std::cout << "no action performed" << std::endl;
    }
    virtual void visit(Image* file) {
        std::cout << "no action performed" << std::endl;
    }
    virtual void visit(Audio* file) {
        std::cout << "no action performed" << std::endl;
    }
};
class CompressionB : public Visitor {
    public:
    void visit(Image* file) override {
        std::cout << "B: compressing an Image file" << std::endl;
    }
    void visit(Audio* file) override {
        std::cout << "B: compressing an Audio file" << std::endl;
    }
};
```

Code 10: Visitor's pattern visitor class

Solution attempt

Concerning game acts transitions, the exact same mechanism can be exploited, except in this case both visitors and elements are game acts, the interest here is not towards semantics but more towards the dispatching mechanism. The way visitor pattern is defined however makes it hard to add a new element because adding one (in this case a new game act), a lot of visitors must be modified (in this case a lot of game acts) to take into consideration the new class. Given that C++ has strong code generation capabilities, one can explore a way to get rid of the boilerplate code and let the language generate all those "visit" methods on behalf of the developer. The first implementation that comes to mind is to keep the same structure as in the visitor pattern but instead of overloading the methods, just use a template to generate them and template specialization to implement the logic for specific elements. An example follows.

```
1.  class Visitor {
2.      public:
3.      template<typename F>
4.      virtual void visit(F* file) {
5.          std::cout << "no action performed" << std::endl;
6.      }
7.  };
8.  class CompressionB : public Visitor {
9.      public:
10.     template<>
11.     virtual void visit<Image>(Image* file) override {
12.         std::cout << "B: compressing an Image file" << std::endl;
13.     }
14.     template<>
15.     virtual void visit<Audio>(Audio* file) override {
16.         std::cout << "B: compressing an Audio file" << std::endl;
17.     }
18. };
```

Code 11: Template Visitor pattern invalid implementation

The code above, may appear to be syntactically correct but in fact it is invalid, specifically prohibited by the standard. In C++ it is forbidden for a method to be declared template and virtual at the same time. In fact, compile-time and runtime polymorphism cannot be mixed together.

Polymorphism workaround

Let's see how the proposed solution operates and how to avoid the problem of code repetition that would derive from the use of the visitor pattern. To work around the inability to have virtual and template methods, one has to manually implement a similar functionality. A sort of poor man's polymorphism. The first step is to enumerate all the game acts (it's important to do this in a centralized manner). One can declare a template class that acts as a type list (this is a basic technique from C++ meta-programming). The actual argument list of this template is declared in a macro in the file AllActsDeclare.h, which is included in the header of each game act. Another header file includes all game act headers and must be included in each game act code (.cpp) file, that file is AllActsInclude.h. At this point it is possible to identify a game act by means of an index number which represents its position in the aforementioned list. The classes are those of the geographic locations example (see Game act).

```
1.  class AGameAct;
2.  class AItaly;
3.  class APiedmont;
4.  class ATurin;
5.  class AFrance;
6.  class ALyon;
7.
8.  #define TYPES AGameAct, AItaly, APiedmont, ATurin, AFrance, ALyon
```

Code 12: AllActsDeclare.h

```
#include "GameActs/GameAct.h"
#include "GameActs/Italy.h"
#include "GameActs/Piedmont.h"
#include "GameActs/Turin.h"
#include "GameActs/France.h"
#include "GameActs/Lyon.h"
```

Code 13: AllActsInclude.h

The macro `TYPES` (code 12) contains the list of all the game act types. Now there are two problems to be solved: given a specific type, how to get its position in that list (as an integer value) and given an integer value how to get the type it corresponds to?

The first problem is solved easily:

```
template <typename Target, typename Types>
struct TFindIndex {
    static_assert(
        TIsSame<Target, void>::Value,
        "Types must be registered before getting their index.");
    static constexpr uint32 Value = 1;
};
template <typename T, typename... Ts>
struct TFindIndex<T, TTuple<T, Ts...>> {
    static constexpr uint32 Value = 0;
};
template <typename T, typename U, typename... Us>
struct TFindIndex<T, TTuple<U, Us...>> {
    static constexpr uint32 Value
        = 1 + TFindIndex<T, TTuple<Us...>>::Value;
};
```

Code 14: TFindIndex class definition

The template class `TFindIndex` has two template arguments: the first is the target type to look for, the second is a list of types. In this case, `TTuple` is used as a list, which is the Unreal equivalent of

`std::tuple`. The member variable `Value` keeps track of the index value, it is calculated at compile time (and declared as `constexpr`). The basic version of the template is never instantiated (if the code works correctly) so there is a `static_assert` which shows a message informing the developer that the searched type (`Target`) was not found in the list and must be "registered" as shown previously. This class has two template partial specializations. The first (code 14, line 8) is the base case that occurs when the target type is the first argument of the tuple, in other words we are looking for a type `T` and provide a tuple that happens to start with `T`. In that case the index is `0` because the value is in the first position. The second specialization is used to increment the index and unpack the tuple. The index is calculated as `1` plus the value of the index that the type `T` would have in a tuple that starts from the second position of the tuple provided (code 14, line 15).

The second problem may be less easily solved. If the index is known at compile-time then it is possible to use the `TTupleElement<Index, TupleType>::Type` template class (Unreal's equivalent of `std::tuple_element<Index, TupleType>::type`). Unfortunately `Index` doesn't accept a value known at runtime (such as the value of a variable). A possible solution is to declare an array that has as many elements as there are elements of `TupleType` and that each element is a function that performs the operations associated with the type that appears in the tuple in the position where the function appears in the array. Using a runtime index then that array can be accessed and the related function executed.

Remember that each game act can provide its own index via the `AGameAct::GetIndex()` method. This virtual method is overwritten by each game act through the use of the `GENERATED_ACT_BODY()` macro that the gentle reader has already seen in code 3. Its content is simply a method (code 15) that allows each game act to provide its own index to

allow the dispatching mechanism to happen. The details and additional types sanitization in code 15 can be ignored. What is interesting though is how the class `TFindIndex` was used (`MethodDispatcher_Private` is a namespace). It has generality and works for each class because the specific type doesn't need to be explicitly stated, it can be obtained using the specifier `decltype` (introduced in C++11). Also, why `private:` at the end? This macro will be used inside the class body (normally at the beginning), it needs to declare a public method thus altering the class' default access specifier, with the final `private:` it will restore the default one.

```
1.  #define GENERATED_ACT_BODY()                                \
2.      public:                                                 \
3.      virtual uint32 GetIndex() const {                       \
4.          return MethodDispatcher_Private::TFindIndex<        \
5.              typename TRemoveCV<                             \
6.                  typename TRemovePointer<                    \
7.                      decltype(this)>::Type>::Type,           \
8.              TTuple<TYPES>>::Value;                          \
9.      }                                                       \
10.                                                             \
11.     private:
```

Code 15: GENERATED_ACT_BODY macro definition

The other two macros used by game acts can be seen in code 16. Unlike `GENERATED_ACT_BODY()`, these macros are optional and must appear only within the classes that need to make use of the dispatching mechanism. In other words, only within classes that need to execute specific code in transition from or to other game acts. Those are `GENERATED_ACT_DISPATCH_DECLARE()` and `GENERATED_ACT_DISPATCH_DEFINE(ActClassName)`.

```
#define GENERATED_ACT_DISPATCH_DECLARE()                                    \
    public:                                                                 \
    virtual void DispatchTransitionFromGameAct(AGameAct* Previous) override; \
    virtual void DispatchTransitionToGameAct(AGameAct* Next) override;      \
                                                                            \
    private:

#define GENERATED_ACT_DISPATCH_DEFINE(ActClassName)                         \
    void ActClassName::DispatchTransitionFromGameAct(AGameAct* Previous) { \
        constexpr uint32 TypesCount = TTupleArity<TTuple<TYPES>>::Value;    \
        return MethodDispatcher_Private::DispatchFunction(                  \
            TMakeIntegerSequence<uint32, TypesCount>(),                     \
            [this](auto* ActualPrevious) {                                  \
                this->TransitionFromGameAct(ActualPrevious);                \
            },                                                              \
            Previous)(Previous);                                            \
    }                                                                       \
                                                                            \
    void ActClassName::DispatchTransitionToGameAct(AGameAct* Next) {        \
        constexpr uint32 TypesCount = TTupleArity<TTuple<TYPES>>::Value;    \
        return MethodDispatcher_Private::DispatchFunction(                  \
            TMakeIntegerSequence<uint32, TypesCount>(),                     \
            [this](auto* ActualNext) {                                      \
                this->TransitionToGameAct(ActualNext);                      \
            },                                                              \
            Next)(Next);                                                    \
    }
```

Code 16: Additional macros definition

The first macro contains the declaration of the two virtual methods `AGameAct::DispatchTransitionFromGameAct` and `AGameAct::DispatchTransitionToGameAct`. Those represent the first step in the double dispatching process. Since they are declared as virtual and overridden in the subclasses, the type of the first of the two game acts is known inside their body.

The second macro (code 16, line 8) calls the function `DispatchFunction` (code 16, line 11, line 21) providing, in order, an integer sequence of values that represents all the valid indexes of the tuple `TTuple<TYPES>`, a lambda function capturing `this` that must be executed (the argument of the function will be the precise type of game act) and the game act object (whose exact type is not yet known here). Inside the lambda, it just calls the most appropriate method of the overloads and overrides of `AGameAct::TransitionFromGameAct` (code 16, line 14) or `AGameAct::TransitionToGameAct` (code 16, line 24) providing as argument the game act converted to its dynamic type.

The final piece of the puzzle is the `DispatchFunction` function. What this function has to do has already been anticipated. Given an index which is the position in the list of a game act, at runtime, it must cast to the corresponding type which, however, can only be obtained using a compile time value therefore a conversion mechanism between the two must be implemented. The integer sequence passed as the first argument of the function is instrumental in the creation of a parameter pack (`Indexes`). That parameter pack is expanded to populate the array `Functions[]`. More specifically, that array is populated by assigning to each element, the result of calling the outer lambda (`[&](auto* DummyTarget) { /*...*/ }(TypeElement)`). The outer lambda is given as argument a value of the type of the current index in the types list (`TTuple<TYPES>`). The purpose of that argument is just to contain the information about type, has no other purpose. That information is then used on the line 16 (code 17) to produce another (inner) lambda (`[Function](TargetType* ActualTarget, ArgsTypes&&... ActualArgs) {}()`) that calls the function provided by the user (captured by copy). That lambda will cast the argument to the correct type before calling the function provided by the user.

```
1.  template <uint32... Indexes, typename FunctionType, typename TargetType,
        typename... ArgsTypes>
2.  auto DispatchFunction(TIntegerSequence<uint32, Indexes...>, const FunctionType&
        Function, TargetType* Target, ArgsTypes&&... Args) {
3.      using ReturnValueType
4.          = decltype(Function(Target, Forward<ArgsTypes>(Args)...));
5.      // We are declaring a static array of TFunction<...> called Functions.
6.      static const TFunction<ReturnValueType(TargetType*, ArgsTypes && ...)>
        Functions[]
7.          // The first time this function is called, Functions will be populated
        with the result returned from the OuterLambda(DummyTarget).
8.          = {
9.              // The argument has no benefit except provideding type information
        for the static_cast below.
10.             [&](auto* DummyTarget) {
11.                 // The OuterLambda(DummyTarget) returns
        InnerLambda(ActualTarget, ActualArgs...)
12.                 return [Function](TargetType* ActualTarget, ArgsTypes&&...
        ActualArgs) {
13.                     // InnerLambda(ActualTarget, ActualArgs...) returns the
        result of calling Function(ActualTarget, ActualArgs...)
14.                     return Function(
15.                         // Get information about the type from above
16.                         static_cast<decltype(DummyTarget)>(ActualTarget),
17.                         Forward<decltype(ActualArgs)>(ActualArgs)...);
18.                 };
19.             }
20.             // Calling OuterLambda(DummyTarget) for each expansion of
        TTupleElement<Indexes, TTuple<TYPES>>... (Remember Indexes is a parameter
        pack)
21.             (static_cast<typename TTupleElement<Indexes,
        TTuple<TYPES>>::Type*>(Target))...
22.         };
23.     return Functions[Target->GetIndex()];
24. }
```

Code 17: DispatchFunction definition

Game act components

Game acts themselves do not perform any action. To assign them abilities, one has to add components, as for actors. In this case `UGameActComponent`. All components inherit from this class. The components also have the opportunity to execute specific code at the beginning and at the end of a game act (as well as on `BeginPlay` and `EndPlay` being components that are assigned to the actors), however, they have no way to intercept the transitions of states and should not even be interested in doing so. Game act components must be thought of as pieces of reusable code that add general functionality and must be able to be reused in various game acts.

Cinematic sequence act component

There are various components and the developer can declare others as needed. For example, a component widely used in this project is the one that allows to start a kinematic sequence: `UCinematicSequenceActComponent`. Let's see its main features as an example of component implementation. To play a sequence three elements are needed:

1. `LevelSequence`: an object which is the sequence itself,
2. `LevelSequenceActor`: an actor spawned in the world that physically represents the sequence as a location in space,
3. `LevelSequencePlayer`: an object to control the sequence play, player is used here in the multimedia sense of the term, remember start, stop, pause?

```
1.  UCLASS(ClassGroup = (GameAct), meta =
        (BlueprintSpawnableComponent))
2.  class SANSISTO_API UCinematicSequenceActComponent : public
        UGameActComponent {
3.      GENERATED_BODY()
4.
5.      protected:
6.      UPROPERTY(EditAnywhere)
7.      ULevelSequence* LevelSequence;
8.      UPROPERTY(VisibleAnywhere)
9.      ALevelSequenceActor* LevelSequenceActor;
10.     UPROPERTY(BlueprintGetter = GetSequencePlayer)
11.     ULevelSequencePlayer* LevelSequencePlayer;
12.
13.     public:
14.     UCinematicSequenceActComponent();
15.
16.     protected:
17.     virtual void BeginPlay() override;
18.     virtual void EndPlay(const EEndPlayReason::Type
          EndPlayReason) override;
19.
20.     public:
21.     virtual void GameActBegin() override;
22.     virtual void GameActEnd() override;
23. };
```

Code 18: UCinematicSequenceActComponent class definition

Let's see some examples of methods. In general, the game act components should perform the setup operations in the `AActor::BeginPlay` method and the corresponding cleanup operations in `AActor::EndPlay`. This is exactly what `UCinematicSequenceActComponent` does as well, as one can see, in `BeginPlay` it performs checks to verify that the necessary details (`LevelSequence`) are available and if not, logs the error. Then it spawns the actor who has to manage the ciinematic sequence (and initializes it by setting the `LevelSequence` object). In `EndPlay` it

eliminates the spawned actor to leave the game world as clean as it was at the beginning.

```
1.  void UCinematicSequenceActComponent::BeginPlay() {
2.      Super::BeginPlay();
3.      if (!this->LevelSequence) {
4.          UE_LOG(LogGameLogic, Error, TEXT("%s: No LevelSequence
          set."), *this->GetPathName());
5.          return;
6.      }
7.      this->LevelSequenceActor
8.          = this
9.                ->GetWorld()
10.               ->SpawnActor<ALevelSequenceActor>(
11.                   ALevelSequenceActor::StaticClass(),
12.                   this->SequenceActorTransform);
13.     this->LevelSequenceActor->SetSequence(this->LevelSequence);
14.     this->LevelSequencePlayer =
          this->LevelSequenceActor->GetSequencePlayer();
15. }
16.
17. void UCinematicSequenceActComponent::EndPlay(const
          EEndPlayReason::Type EndPlayReason) {
18.     Super::EndPlay(EndPlayReason);
19.     if (this->LevelSequence) {
20.         this->GetWorld()->DestroyActor(this->LevelSequenceActor);
21.     }
22. }
```

Code 19: Cinematic sequence act component methods 1

The `GameActBegin` and `GameActEnd` methods, as one can see from code 20, take care of starting the revelation of the game act. In this case it simply starts the cinematic sequence using the object `LevelSequencePlayer`. One can also observe how the game act components take the initiative to end the current game and move on to the next act. When the sequence ends, the current game act has fulfilled its purpose and has nothing more to show, so the transition to the next game act is scheduled by this component.

```
1.  void UCinematicSequenceActComponent::GameActBegin() {
2.      Super::GameActBegin();
3.      if (this->LevelSequence) {
4.          this->LevelSequencePlayer->OnFinished
5.              .AddDynamic(this, &ThisClass::ScheduleMoveToFollowing
        GameAct);
6.          this->LevelSequencePlayer->Play();
7.      }
8.  }
9.
10. void UCinematicSequenceActComponent::GameActEnd() {
11.     Super::GameActEnd();
12.     if (this->LevelSequence) {
13.         this->LevelSequencePlayer->OnFinished
14.             .RemoveDynamic(this, &ThisClass::ScheduleMoveToFollow
        ingGameAct);
15.
        this->DoSequencePlayerAction(this->PlayerActionGameActEnd);
16.     }
17. }
```

Code 20: Cinematic sequence act component methods 2

A noteworthy detail in the previous code snippet is the check that variables that may be uninitialized (`nullptr`) are actually initialized. This is because in the unfortunate event that the game code should crash, then the entire Unreal Engine editor will crash and possibly even corrupt the current project. The author learned this lesson the hard way. Looking at the code of the engine itself (and one can do it because it is source available as explained in 3.1 Unreal Engine), one can certify that this sort of defensive programming is widely used.

4.4 Computational complexity

Concerning the use of **memory**, the complexity is quadratic ($O(n^2)$) in the number of game acts classes. However, also using the visitor pattern there is a quadratic complexity because a visitor must be able to manage every single element type. If, as stated previously, both the visitors and the elements are game acts, then one has that each game act class must be able to manage every single other game act class. Here it is, the quadratic complexity. There is however a possible improvement: only the game acts that need it can make use of the macro `GENERATED_ACT_DISPATCH_DEFINE()` which will instantiate the template function `DispatchFunction` and allocate memory for the array `Functions[]` which contains an element for each game act. From that point of view it could be said that the memory utilization is less than quadratic if only a small part of the game acts have to distinguish the transitions from and to different types of game acts.

Considering the **time** complexity, it is constant ($O(1)$) from the second function call onward. The first time the function `AGameAct::DispatchTransitionFromGameAct` (or its dual "transition to") is called, in turn it calls `DispatchFunction` and initializes an array of functions (as one can see in code 17, line 6) therefore the time complexity will be linear ($O(n)$) because that array contains an entry for each game act class. However, since this array is declared static, in case of subsequent calls of the `DispatchFunction`, the initialization operation will be skipped and the processor will proceed directly to line 23. This is a relevant difference with the visitor pattern. In the case of visitor, the operation requiring linear time takes the form of compiling a number of methods, one for each game act class, that is done at compile time rather than runtime like in our case.

4.5 Overview

At this stage the gentle reader has understood what the aims to be achieved with this application consist of and what tools are available to implement it. The finite state machine that manages the succession of actions and events in the application is shown in figure 24. This diagram is encoded by symbols and colors. Each oval shape represents a game act, black ones are implemented in C++ while blue ones are Blueprint. In this case, Blueprint has been used, not to implement the operational logic but rather to populate the default properties of the class with the specific assets (cinematic sequences, audio files, graphic widgets to be displayed, etc.) that are needed by the object representing the specific game act. Similarly the most relevant game act components are shown using icons nearby, please refer to the legend to understand what each icon is about.

The finite state machine is logically divided into six parts. The first one (violet square) is an introduction to allow the user to become familiar with the device and the controller, there is a start menu displayed as a widget in the space showing some logos of sponsors of the exhibition as well as basic information (`Home` game act). This is followed by the interaction with the rope (see 4.5.2 Rope Interaction). Once the introduction part is over, the curtains opening sequence begins (see 4 Realization). The third part of the show (green square) starts when the curtains are completely opened and the light of the clouds is already dimmed out, it plays an audio description of the various elements of the painting: the characters, initially darkened, light up one at a time and the narrator explains their role in the painting. The fourth part (blue square) is the conclusive one: the user is given some time to admire the painting, then the curtains begin to close, once closed the user is invited to put the device back on the stand. The

part that follows is a connection step to allow the application to start over again, for a new user. As one can see in the diagram, the application is cyclical, it starts from the game act `Home` and continues until the end of the show at the game act `WaitEyeTrackingLost`. At that point the user is expected to take off the headset so eye tracking is lost. Another user will wear the device then the eye tracking will resume. The application restarts from the beginning and the cycle continues until the application is closed or the device is turned off. The last part (see 4.5.1 World positioning) in the red square is functional to the positioning of the virtual scene in the real space, it does not directly concern the show so it takes place in parallel to the rest.

4.5.1 World positioning

The main game act of the virtual scene positioning functionality is `WorldPositioning`. This game act will spawn in the world origin position (zero transform: location (0,0,0), rotation (0,0,0), scale (1,1,1)) an actor called `WorldOrigin` that contains a `UMagicLeapARPinComponent`. Initially this AR pin will not be associated with any real position, so the user will be asked via a widget to position the scene into space then, the game act `SceneSetup` will be staged. This game act allows the user to move its point of view in the virtual space using the controller, from the user point of view it feels like the entire scene is being moved while in reality just `UserPawn` is moving. Once positioning is finished, the user can save the new location, at that point the actor `WorldOrigin` spawned before will be pinned and associated with a precise physical position, saved in the device's memory. The application can now be closed and the device turned off.

By restarting the device and relaunching the application the game act `WorldPositioning` is staged again, the actor `WorldOrigin`

will be spawned again in the transform zero. A difference this time is that the `UMagicLeapARPinComponent` remembers being previously saved as pinned, so it will automatically position itself in the virtual space as to lay in the same position of the physical space. At this point we have a reference of the physical world in the virtual world. We know that this reference in the real world should be in the zero transform position. To make the two points overlap, the actor `UserPawn` is moved to the correct position giving to the user the illusion that the scene automatically goes to orient itself in the real space in its correct position. The gentle reader might wonder why instead of moving the user pawn into the virtual world, aren't all the actors attached to a root pin that automatically positions itself in the space. The problem is that doing this way all the actors would have to be set movable and one would lose some optimizations that apply only to the static actors. Leaving the scene untouched and moving only the user's perspective moreover simplifies the job of the artists that don't need anymore to be aware of those implementation details.

4.5.2 Rope interaction

To engage the user a little bit, an interaction element has been added, it consists of touching a rope in order to start the show. This interaction is used also as an artifice to make sure the user is prepared to observe the scene in order to not miss the curtains opening animation. The action takes place as follows: the precise position of the `RopeSpawner` (figure 19) actor is calculated based on the user's position in the scene. Once the placeholder is located, the actual rope actor is spawned. The rope is being lowered from above through an animation (`RopeDescending` game act), the narrating voice asks the user to touch the rope with the hand, at which point the user has a few seconds to carry out this action (game act `PullRope`), after that, the rope is being taken away and the sequence of the curtains opening sequence starts.

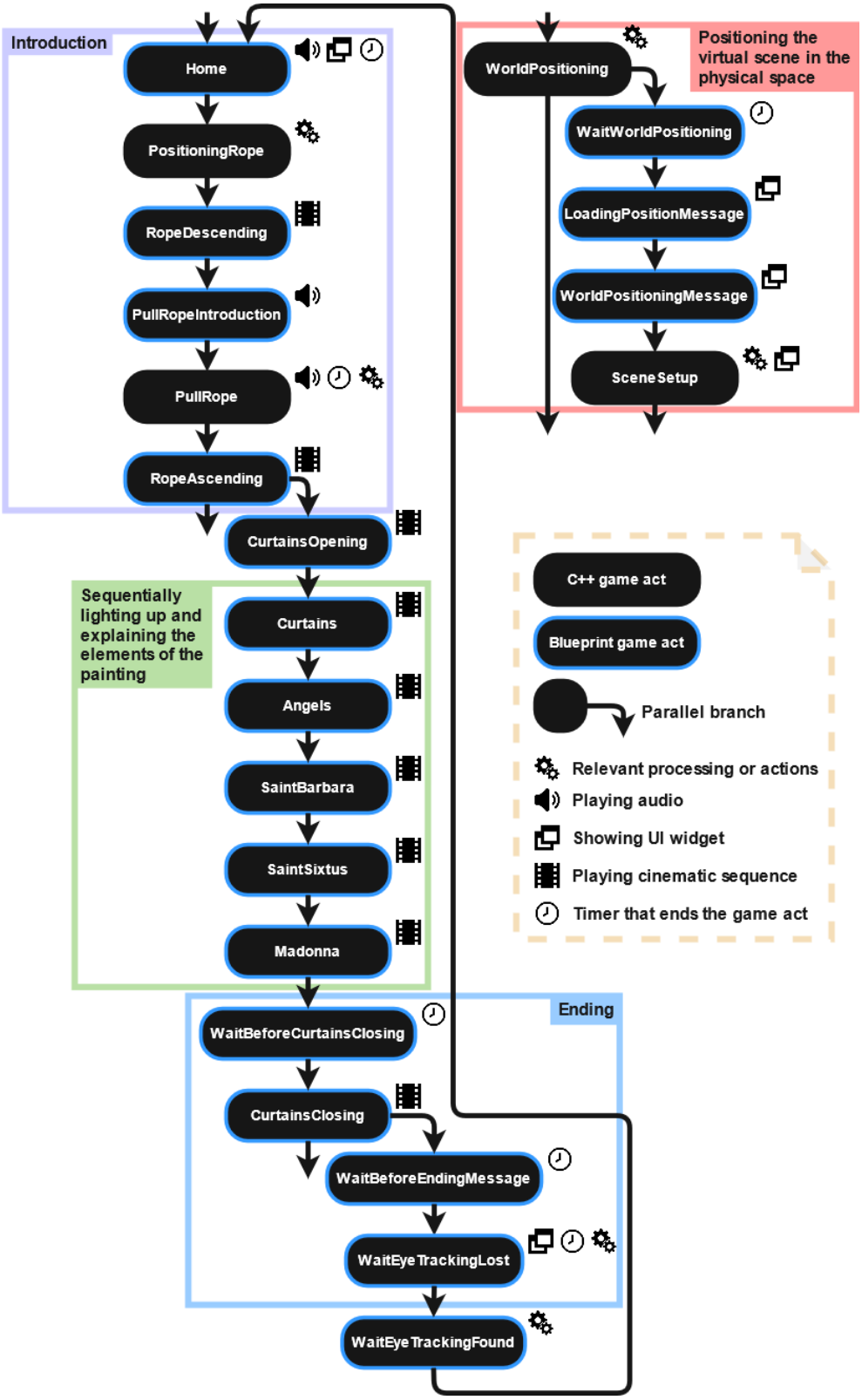


Figure 24: Full game acts diagram

5 Conclusions

This study carried out in the context of the development of the project for the church of San Sisto to show the painting Sistine Madonna can prove useful and provide inspiration for the gentle reader. An example of software architecture was shown that allows the implementation of a finite state machine: a paradigm in which it's easier to express the logic of the application. It is convenient less confusing for the developer to divide the logic of the game into small steps to be made one after the other or quests to be completed in succession. Having a precise set of tools provides a common guideline and language to use within a development team. This is essential to keep the code as tidy as possible and make it maintainable even when the elements involved begin to grow in number.

This type of approach applies very well to scenarios similar to the one presented, that is, where one wants to implement applications related to cultural heritage or in artistic contexts in general. In fact, by their nature these applications consist of a series of actions to be performed in sequence to show the user some type of content related to the show or in a series of objectives to be completed. Let's imagine that we are perhaps inside a museum with multiple rooms. The viewer will have to wear an augmented reality device and walk through the halls of this museum by completing a series of quests. The quest may consist, for example, of observing a certain virtual (or real) sculpture or a painting, or physically reaching a certain place to trigger certain events. These can very well be game acts that are activated automatically when the user walk nearby or queued to other game acts.

Whatever the specific need is, this architecture that mimics a state machine is general enough to be adapted to meet those

requirements. It is up to the developer to identify in the context of the specific application the appropriate and meaningful game acts. To these game acts the developer will create and add the game act components necessary to carry out the specific actions attached to them. We have seen as an example a game act component that starts a kinematic sequence but the possible operations can be the most disparate, examples being: spawning an actor, splitting the quest line by staging another game act and starting a parallel path, playing a sound, waiting for an input from the user, waiting the gaze of the user to be directed towards a specific point, etc. The limit is imposed only by creativity not by technology.

Bibliography

[1]: How VR and AR Will Change How Art is Experienced https://www.invaluable.com/blog/how-vr-is-changing-the-art-experience/

[2]: Piacenza 2021/21 https://www.piacenza2020.it/en/project/

[3]: Madonna Sistina in guerra https://st.ilsole24ore.com/art/arteconomy/2013-12-03/madonna-sistina-guerra-073140.shtml

[4]: La Madonna Sistina di Raffaello, in un doc la storia del capolavoro ritrovato https://www.repubblica.it/spettacoli/cinema/2020/12/16/news/la_madonna_sistina_di_raffaello_in_un_doc_la_storia_del_capolavoro_ritrovato-278486285/

[5]: History of the Unreal Engine https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine

[6]: A first look at Unreal Engine 5 https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5

[7]: Introduction to C++ Programming in UE4 https://docs.unrealengine.com/en-US/ProgrammingAndScripting/ProgrammingWithCPP/IntroductionToCPP/index.html

[8]: UnrealHeaderTool https://docs.unrealengine.com/en-US/ProductionPipelines/BuildTools/UnrealHeaderTool/index.html

[9]: Using the Meta-Object Compiler (moc) https://doc.qt.io/qt-6/moc.html

[10]: Explanations of the basic gameplay elements, Actors and Objects https://docs.unrealengine.com/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/index.html

[11]: Transforming Actors https://docs.unrealengine.com/en-US/Basics/Actors/Transform/index.html

[12]: What is a gimbal and what does it have to do with NASA? https://science.howstuffworks.com/gimbal.htm

[13]: What is a Finite State Machine?
https://medium.com/@mlbors/what-is-a-finite-state-machine-6d8dec727e2c

[14]: A polyglot's guide to multiple dispatch
https://eli.thegreenplace.net/2016/a-polyglots-guide-to-multiple-dispatch/

List of figures

List of code snippets