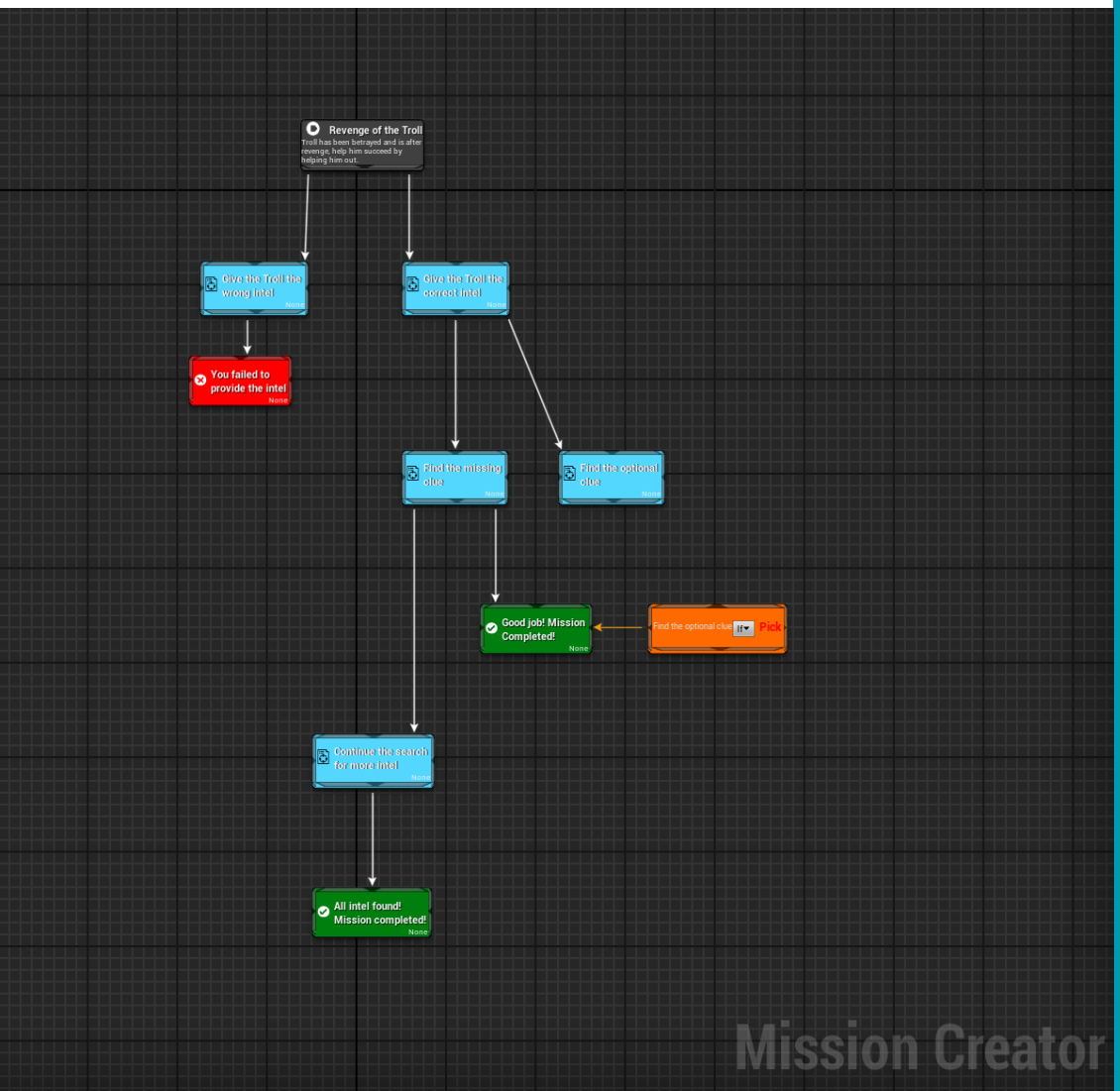
Dennis Bäckström

Mission Creator Plug-in, a Developer Tool for Unreal Engine 4



Bachelor of Engineering

Information and communications technology

Spring 2021

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Bäckström Dennis

Työn nimi: Liitännäinen tehtävien tekemiseen, kehittäjän työkalu Unreal Engine 4:lle

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: C++, UE4, liitännäinen, tehtävä

Opinnäytetyön aihe tuli saksalaiselta yritykseltä NeXR Technologies, tarkemmin VRiday:lta yrityksen VR ohjelmistokehityspuolelta. Tämän työn tarkoituksena oli ohjelmoida liitännäinen Unreal Engine 4:lle, jonka avulla pystyisi luomaan tehtäviä peleihin tai muihin UE4:llä tehtäviin projekteihin, jolloin loppukäyttäjä pystyisi näitä tehtäviä suorittamaan.

Liitännäisen vaatimuksina oli pystyä tekemään tehtäviä yksinkertaisesti käyttäen graafista editoria, tehtävillä tuli olla alku ja loppu sekä mahdollisuus haaroittamaan tehtävän kulkua. Tehtäviin tuli myös pystyä lisäämään toimintoja tehtävän eri vaiheisiin.

Työn ohjelmointiosuus toteutettiin käyttäen C++ ohjelmointikieltä, ohjelmointiosuus koostui tehtävä objektista ja siihen liittyvien objektien teosta sekä graafisesta editorista tehtävien tekoon. Ohjelmoinnin tehtyä tuli testata tämän liitännäisen kaikkia toimintoja ja korjata mahdolliset ongelmat. Työssä käytiin ensin läpi tausta tietoa liitännäisistä, Unreal Engine 4:stä ja tehtävistä, jonka jälkeen hieman työn implementaation tärkeämmistä luokista ja funktioista, viimeisenä testaus menetelmästä.

Työn parissa tuli lisää osaamista UE4:n käytöstä, editorin laajentamisesta sekä objektien sarjallistamisesta, testaamisen jälkeen muutaman toiminnon korjattua liitännäinen toimi suunnitelman mukaisesti ja liitännäisen perusvaatimukset täyttyivät.

KAMK • University of Applied Sciences

Abstract

Author(s): Bäckström Dennis

Title of the Publication: Mission Creator Plug-in, a Developer Tool for Unreal Engine 4

Degree Title: Bachelor of Engineering, information and communications technology

Keywords: C++, UE4, plug-in, mission

The subject for this thesis came from a German company called NeXR Technologies, more specifically from the VR software development department VRiday. The purpose of the thesis was to create a plug-in for Unreal Engine 4, enabling the creation of missions for games or other types of projects done with UE4, and these missions could be completed by the end user.

The plug-in requirements were to be able to create a mission in a simple way with a graph editor, and the missions would need a start and an end, and possibility to branch out the mission flow. Additional requirement was the possibility to attach events along the mission's flow.

The work was done by using C++. The programming part consisted of creating the mission and its related objects, and a graph editor for creating the missions. After the programming was done, a throughout testing of all the features in the plug-in was done. First in this thesis, some basic knowledge of what plug-ins, Unreal Engine 4 and missions are, were brought up, then some of the plug-in implementation's more important classes and functions, and lastly the testing method was explained.

During the process of making the plug-in, knowledge was gained about extending the editor in UE4 and serializing data for saving. After the testing and fixing some broken functionality, the plug-in worked as planned and the mission plug-ins basic requirements were fulfilled.

Contents

1 Introduction

The purpose of this thesis is to create a developer tool plug-in for Unreal Engine 4 (UE4). This plug-in would allow the creation of missions, that would then be completed by the end users in the final product. Extending UE4 with features such as a custom editor is not that well documented, and therefore this thesis may be of help as some guidance, while going through some of the implementation process of this mission plug-in developed for a company called NeXR Technologies.

NeXR Technologies is a technology company based in Berlin, Germany. They have expertise in areas such as motion capturing, 3D person scanners and VR software. This plug-in was for the VR software development team VRiday, to aid them easily create missions for the end user to complete. In a likely scenario these missions could be, for example, tutorials, such as learning to grab objects in VR and then interacting with them in a specific way, or how to use included locomotion systems.

With this plug-in, the developer should be able to create inside UE4 an object which includes a graph editor. In the graph editor, the developer should be able to create different types of named nodes with a description, such as begin, end, and objective nodes which could be combined to create the hierarchy of a mission. Currently, by default UE4 does not provide an efficient way to create missions, therefore a plug-in like this is beneficial to have, while very basic or short type of missions might not need a plug-in like this, but longer missions that for example could branch into multiple sub-missions could see a very good use of a plug-in like this. Similar and other types of plug-ins exist for UE4 and can be bought, for example, from the Epic Game's Unreal Engine Marketplace.

2 Plug-ins, add-ons, and extensions

Plug-ins are also often known as add-ons or extensions. Software that has plug-in support usually allows adding new functionality without modifying the underlying software. Easy extendibility is a good thing when it comes to software development, a software's lifetime might be long, and one might not know the wanted and needed features initially. Giving the internal development team or even outsiders the access and capability to extend the software in a simple way, might be a great approach depending on the product being developed. However, sometimes plug-ins might cause security issues, for example there have been cases of malicious plug-ins uploaded to the Chrome Web Store, where some Chrome users have downloaded these extensions and fallen victims of malware (1). Plug-ins can be found for many different software, both as freeware and commercial. Plug-ins are not usually interchangeable between completely different software and new updated versions of the same software might break existing plug-ins and then the creators of the plug-ins need to provide an update for those plug-ins to work again. How these plug-ins are enabled depends on the software. Some plug-ins do have an installer, when executed it installs the plug-in to the correct location and can even check if the installed software exists or if the version is suitable, whereas in other cases the plug-in must be manually copied into the correct folder often in the install location of the software or some other common folder, such as for Windows OS the My Documents folder.

Plug-ins started to appear more in software in the early 1990s, and one of the early software that supported plug-ins was Adobe Photoshop in 1991 when version 2.0 was released. Photoshop the graphics and photo editing software has seen a lot of plugins since then, for a software like Photoshop this enabled it to get a lot of new features along with the version updates from Adobe themself. (2,3) Plug-ins for Photoshop can be found, for example, from Adobe's Adobe Exchange platform, where plug-ins can be bought or found for free, and from there the plug-ins directly syncs with one's Adobe account and can then be used in Photoshop. From other third-party sites plug-ins might need to be manually moved into place to work. One can still to this day develop new plug-ins for Photoshop by joining Adobe's developer program and getting access to their SDK's and tools.

Another popular appearing of plug-ins in the 1990s was within audio software. The company Steinberg had created in 1991 a software called Cubase, a digital audio workstation for short DAW. This software enabled making audio recordings directly into the computer, and Cubase then later in 1996 received an update with a feature called Virtual Studio Technology, VST for

short. In 1997, Steinberg made VST as an open standard for everyone and so it is one of the most popular plug-in formats for DAWs. DAWs allow creation and mixing of music with multiple tracks and MIDI connections, MIDI which is a sort of interface between digital instruments and software, the VST plug-ins allows for example new simulated instruments and audio processing to be included in the host DAW software. With some audio, math, and programming knowledge one can go and create VST plug-ins using for example C++ programming language. Many old DAWs such as Cubase are still up to date and have received several updates along the years, VST plug-ins can be found both newer ones but also older VST plug-ins remain available on the internet and popular to this day. (4-7)

Popular game engines such as Unity and UE4 also support plug-ins. The two companies behind these two engines both offer online stores where one can upload and sell the created plug-ins for others to use in their projects. Often games and especially game series coming from within the same game engine tend to have similarities and can assume some reasons for the similarities such as of course the game engine itself, but also possibly some plug-ins that are used in the engine and toggled on for specific projects, allowing more easily to bring over the content across several projects.

Unity game engine provides support for plug-ins called managed plug-ins and native plug-ins. Managed plug-ins in Unity are dynamically linked libraries, so these plug-ins are compiled outside Unity and the DLL files are then added to the project by dragging the files into the project, with these plug-ins the source code is not included with the plug-in. The native plug-ins in Unity can be written in languages such as C or C++, even when the language used in Unity is C#. Unity has a plug-in inspector that can be used to manage used plug-ins.

UE4 also has a few different types of plug-ins it supports, these plug-ins can be for example content only which would not have any code, only assets for example audio, art, or blueprint scripts. The plug-ins that include code consists of modules, these modules which whole UE4 is built upon enables the separation within a plug-in with module types such as editor and runtime modules, where the editor module would not be available for the end user. Plug-ins for UE4 can be installed on a per project or engine basis, by either placing the plug-in into the engine's plugin folder or the projects.

So, plug-ins vary a lot between the implementations depending on to which software they are created to. They vary a lot in complexity, used for example in game development, music production and graphic design software and more.

3 Unreal Engine

Unreal Engine is a popular game engine by the company Epic Games. The engine is far from just a game engine today. It is currently on the 4th generation of the engine and nearing the release of Unreal Engine 5. UE4 is used, for example, in the film, game, and automotive industry. Recently, UE4 has been used, for example, in the making of Westworld and The Mandalorian TV shows, upcoming Hummer EV truck by General Motors Truck Company is told to be the first out running the in-vehicle infotainment on Unreal Engine. There has been a lot of games created with UE4, for example big games such as Fortnite made by Epic Games themself, Final Fantasy VII Remake, Tekken 7, and Borderlands 3 made by other companies. (8-11)

The journey started as Unreal Engine saw its first game created on it, released in 1998 called Unreal. Unreal was a first-person shooter game that really took ground among the other famous shooters around that time like Quake and Doom, the engine provided good graphics, enabling real-time level changes, and even let access to a level editor for the players. Editor view for first Unreal Engine looked like in Figure 1. (12-14)

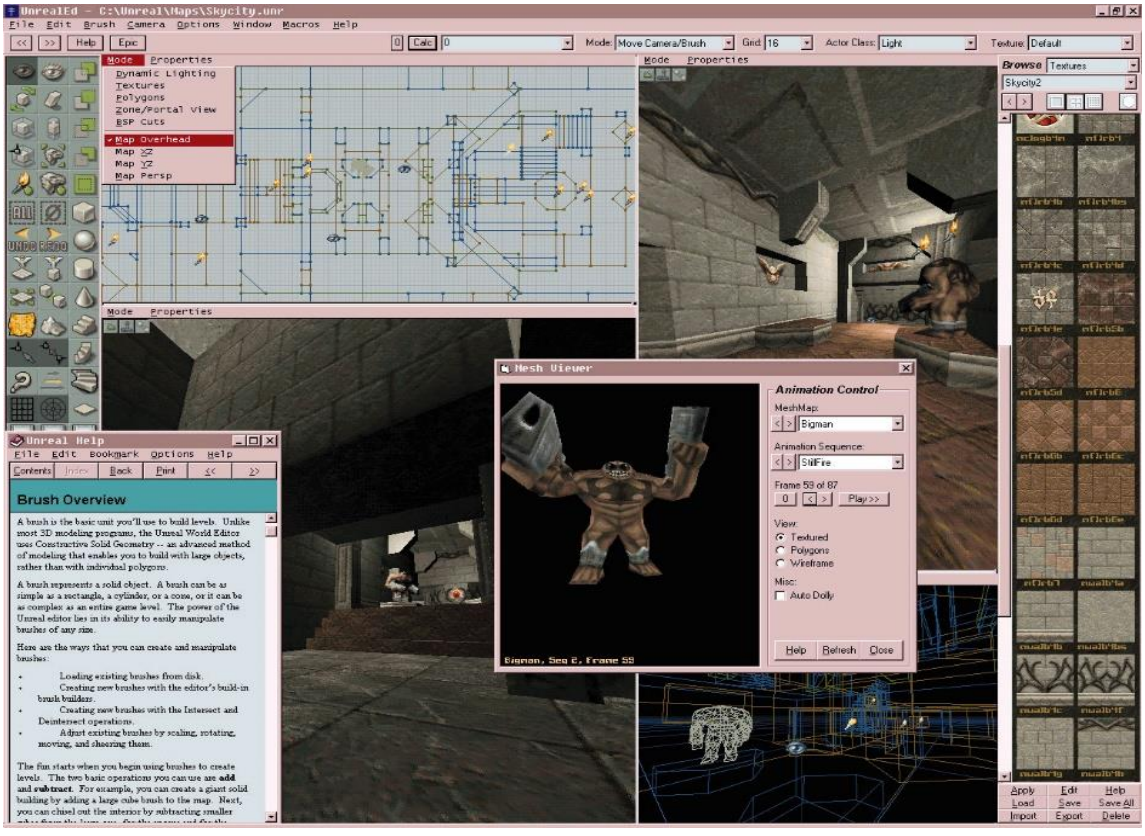


Figure 1. Unreal Engine's editor called UnrealEd. (15)

The main person behind the story of Unreal Engine is Tim Sweeney, current CEO of Epic Games. Tim Sweeney has told in an interview that he did most of the code in the first engine. Before the 4th generation of the engine, engine used a script language made by Tim Sweeney called UnrealScript, it was like Java the object-oriented language. The purpose of it was to allow a more simplified and powerful language to use for the internal and third-party Unreal Engine developers compared to the other existing programming languages. The focused feature of Unreal Engine was extendibility, shortly after the Unreal game launched the engine was licensed out to multiple companies. There was a good communication between the Unreal Engine team and companies using the engine, which allowed it to improve in the areas game companies thought it was lacking or just needed improvements. As the hardware heavily improved with things such as dedicated graphics cards, faster central processing units and more system memory the Unreal Engine received updates with the engine generations to take the most out of this more capable hardware. Over time Unreal Engine had been licensed out to companies such as Square Enix, EA, Disney, and Ubisoft. Later when Unreal Engine 4 arrived on 19th March 2014, while previous generations relied on licensing it out and having smaller teams out from considering it because of high cost, this time Unreal Engine launched as a monthly fee of $19 and royalties based on sales made by using the engine, it also had the whole source code available for developers. Being much cheaper made it easier for smaller developers to start using Unreal Engine. Around a year later Tim Sweeney announced that UE4 became free to use and only the royalties based on sales were left. (12-14,16-18)

In UE4, programming is done by C++ or Blueprint, as C++ replaced the old UnrealScript language in this generation and Blueprint visual scripting replaced the older similar Kismet. Projects can be created for many of recent devices such as PC and VR platforms, Consoles, iOS, and Android devices. Since its launch, UE4 has seen many updates with new and replacing features. Raytracing has been added after graphic cards supporting the technology started to appear, older particle and effect system has been replaced by a more powerful Niagara, and currently the physics engine is being replaced by a new one called Chaos, which is replacing the third-party PhysX implementation. UE4 has also in the recent update 4.26 included beta features such as rigging skeletal meshes directly inside UE4, which means creating animations for characters directly inside UE4 instead of software such as Blender or Maya. Also, a water plug-in that enables creating islands and river streams for example into a landscape. As UE4 is open source, with all the updates Epic Games releases they also, for example, include a wide variety of bug fixes made by the UE4 developer community.

The visuals and overall features in the editor view have also changed a lot since the first generation of the engine. UE4's editor view can be seen in Figure 2 below. The only thing that still looks a bit like the first engine's editor is the viewport, which is quite standard looking when it comes to software with editing viewports, such as Unity, Maya or Blender has viewports which looks similar and can be divided to four different camera views.

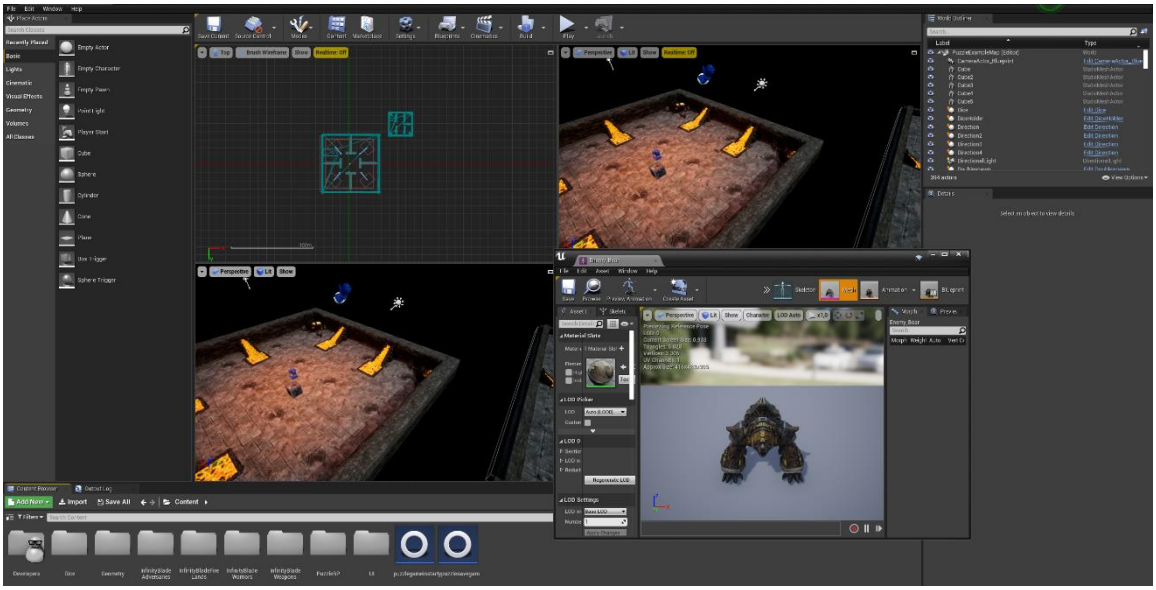


Figure 2. Editor View in Unreal Engine 4.

UE4 provides short ready-made templates for different categories when choosing to create a new project as seen in Figure 3 below. These categories have inside furthermore specific templates for more specific use cases, such as for games there are templates for First Person, Puzzle, Vehicle and Virtual Reality.

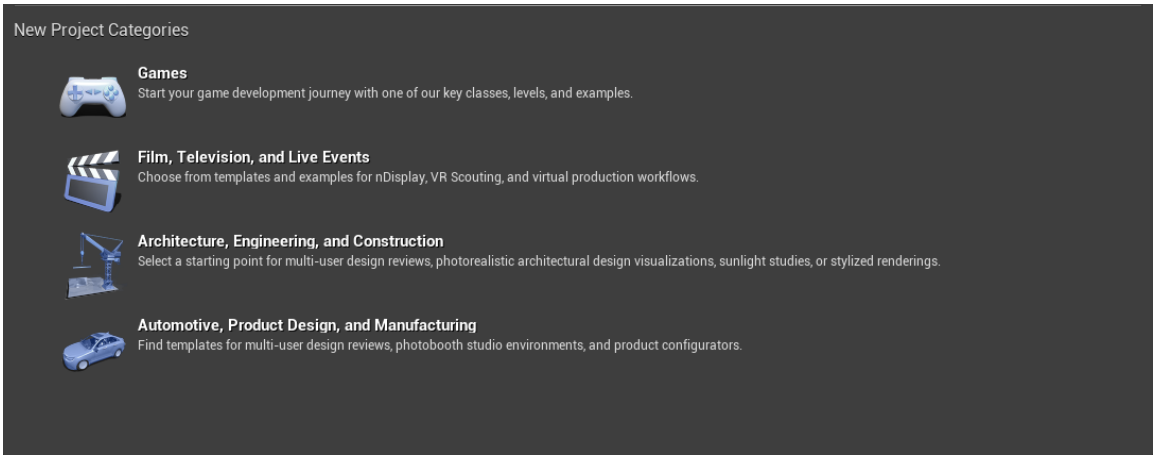


Figure 3. UE4 template categories in a new project creation window.

4 Missions, quests, and objectives

In some video games and interactive experiences, one can come across missions also often called quests. It usually depends on the type of game, more often in role-playing games one gets quests, while for example in some first-person shooter game one would more likely get missions. These then might have some objectives that tell more specifically what one is supposed to do to complete the mission or quest.

For the end user, these missions provide a way to have something they strive to complete, be it a mission aiding to know the user interface and controls, or just a mission related to a story. How the user knows about these missions, there is usually a UI that provides the required information. It can vary a lot, as an example below in Figure 4 shows how it is in a third-person shooter game called Mafia: Definitive Edition and in an RPG game called Divinity: Original Sin 2. In the Mafia game, the player only has one mission at a time, and it shows the current objective of it on the upper left corner, while for the RPG game the player has a journal that they open to look through several quests and can choose which ones they make as active and track.

Figure 4. Objective examples: left one Mafia: Definitive Edition, right one Divinity: Original Sin 2.

From the development side there is quite a variety of ways to enable the creation of these missions, as it is for almost everything programming related there is almost always another way to implement. In this thesis, the mission creator is made as a plug-in and capable of using it in any UE4 project, it will feature executable events, in which different functionality would be attached to. But a larger mission creation tool could for example feature built in tools for creating dialogue, rewards, tracking missions, NPC AI routing and more.

For basic mission functionality, there needs to be a manager object that can take care of checking current missions, giving out, cancelling missions, and updating objectives. Missions and their objectives would need a name and description so that the UI can include these and show to the end user, just like in the Figure 4. The progress of the mission needs to be able to be saved in its current state, which includes already done objectives and current objective progress, one would not want to have the end user go through the same things again because he had to close the game and it did not save the progress. Missions need a starting and an end point, on these points one should be able to connect functionality that would execute on triggering the point. This functionality could be for example to create an item in the game world or when ending the mission it would give a reward or clean up mission related things.

Some companies provide tools to modify their game with additional content, most likely slightly modified of what they use within the company, Ubisoft a large game company provides for example for their Assassin's Creed Odyssey (2018) game a story creator tool. With this tool anyone owning the game can create story content for the game. It has features such as choosing the start location of the mission and how it should trigger, creating a name and thumbnail for the mission, modifying the NPC characters in the mission, with things like character customization, choosing factions, where some factions react differently to other faction's characters, or what quest items should the NPC carry and can the character die. Quest objectives can be set, for example, as talk, destroy, escort, free prisoners. The quests are created by a node system as shown in Figure 5. This mission creator has advanced features such as travel behaviour where one can make an NPC go somewhere after certain activity is completed. It also includes a dialogue system to create one's own dialogue. (19)

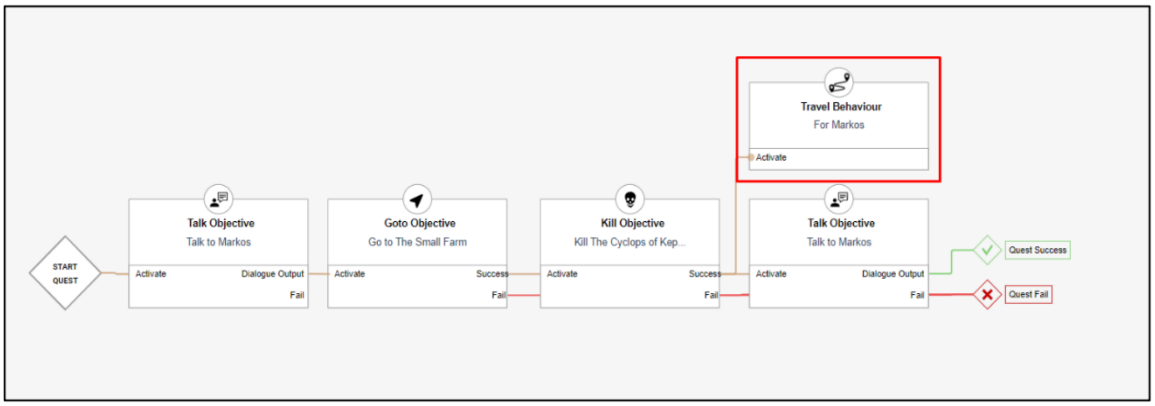


Figure 5. Assassin's Creed Odyssey's story creator quest example. (19)

Another game company called CD Project Red provides a tool for the public called REDkit, with which the user can create new content for The Witcher 2 (2012) game if one owns the game. This tool also includes capability to create quests among other features. Similarly, to the Ubisoft's story creator tool, this one provides a graph-based editor to create the quest hierarchy as seen in Figure 6. It can be used together with the other tools in the kit to create quests with cutscenes, dialogues and different kinds of objectives with rewards and NPC characters doing things such as working, fighting, and walking. (20)

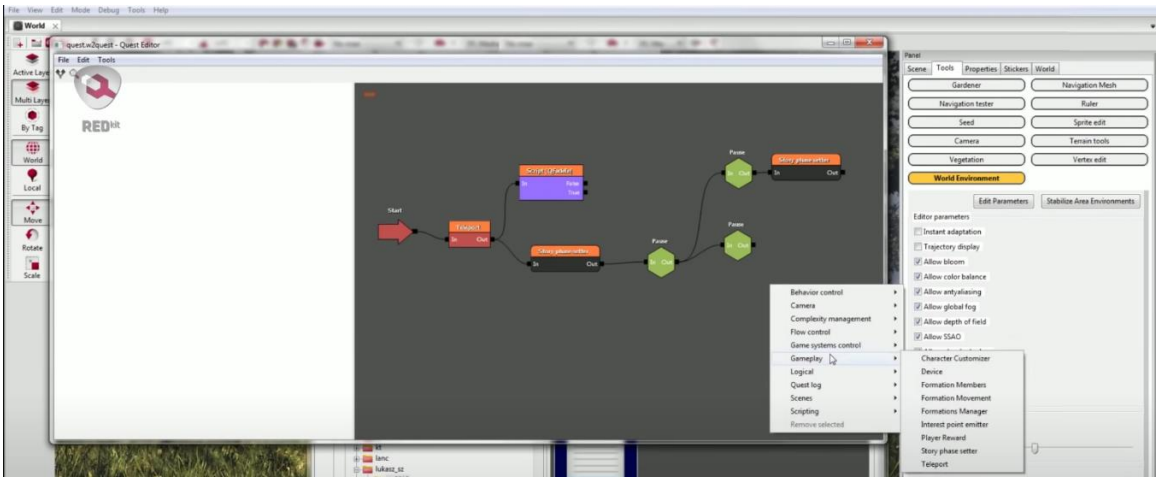
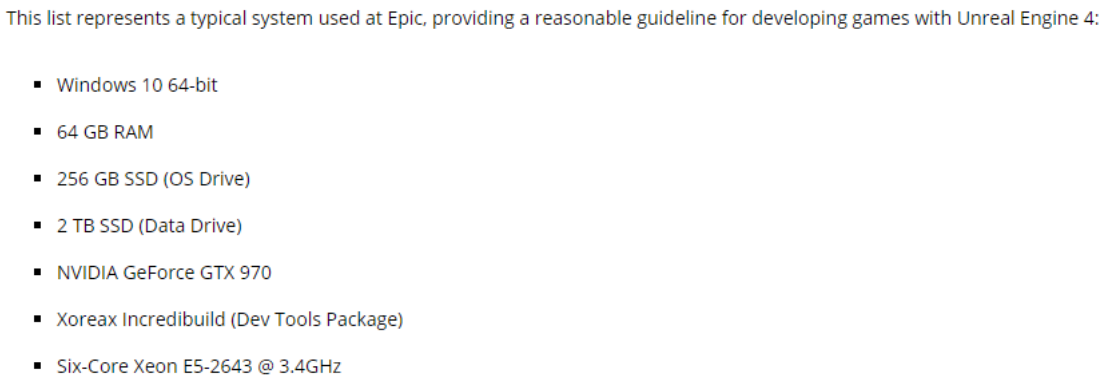


Figure 6. REDkit quest editor view. (21)

5 Development tools and methods

UE4 can be run on Windows, Linux and MacOS. This plug-in was created on a PC with Windows OS, with Visual studio 2019 as the IDE, 2017 version is still also supported. UE4 can be downloaded from Epic Games Launcher or the source release from GitHub, used version was the binary 4.25.4 downloaded from the launcher, for making this plug-in there is no need for the things that the source release provides, compared to what the binary release provides. Using the binary release, it will automatically install the required DirectX components and Visual C++ redistributables. If using the source, it might not install automatically so one would need to run an executable found inside the engine install location in Extras folder which would install them, along the additional other steps for compiling and running the source release. When using the binary release, it will not by default include the symbols for debugging editor so this must be enabled for it to download them, it can be found from the downloaded engine options menu in Epic Games Launcher, this will be necessary to debug any problems appearing with the created custom editor that this plug-in will feature. In UE4 a plug-in can be created in any project, existing or new, this plug-in was started from an empty blank project without additional starting content, the quality settings did not either matter when creating this plug-in.

The hardware for just creating a plug-in does not necessarily need to be like computers that Epic Games uses or used previously as listed in Figure 7. But compile times are kept at minimum when combined with strong performing high core count CPU's and fast SSD's.


This list represents a typical system used at Epic, providing a reasonable guideline for developing games with Unreal Engine 4:

- Windows 10 64-bit
- 64 GB RAM
- 256 GB SSD (OS Drive)
- 2 TB SSD (Data Drive)
- NVIDIA GeForce GTX 970
- Xoreax Incredibuild (Dev Tools Package)
- Six-Core Xeon E5-2643 @ 3.4GHz

Figure 7. Typical system used at Epic Games for development. (22)

6 Implementation

First step was creating the basic structure for the plug-in, in UE4 to create plug-ins one can either create them manually or use the included creator which is found in the plug-ins list menu with all existing plug-ins. The tool has a few different types of plug-in templates one can use, some of them seen in Figure 8. Choosing the blank template for this plug-in, a blank plug-in when created will create a folder with given plug-in name into the project's plugins folder, in this newly created folder it also creates the uplugin file, resources folder and a source folder with one included module named as the created plug-in.

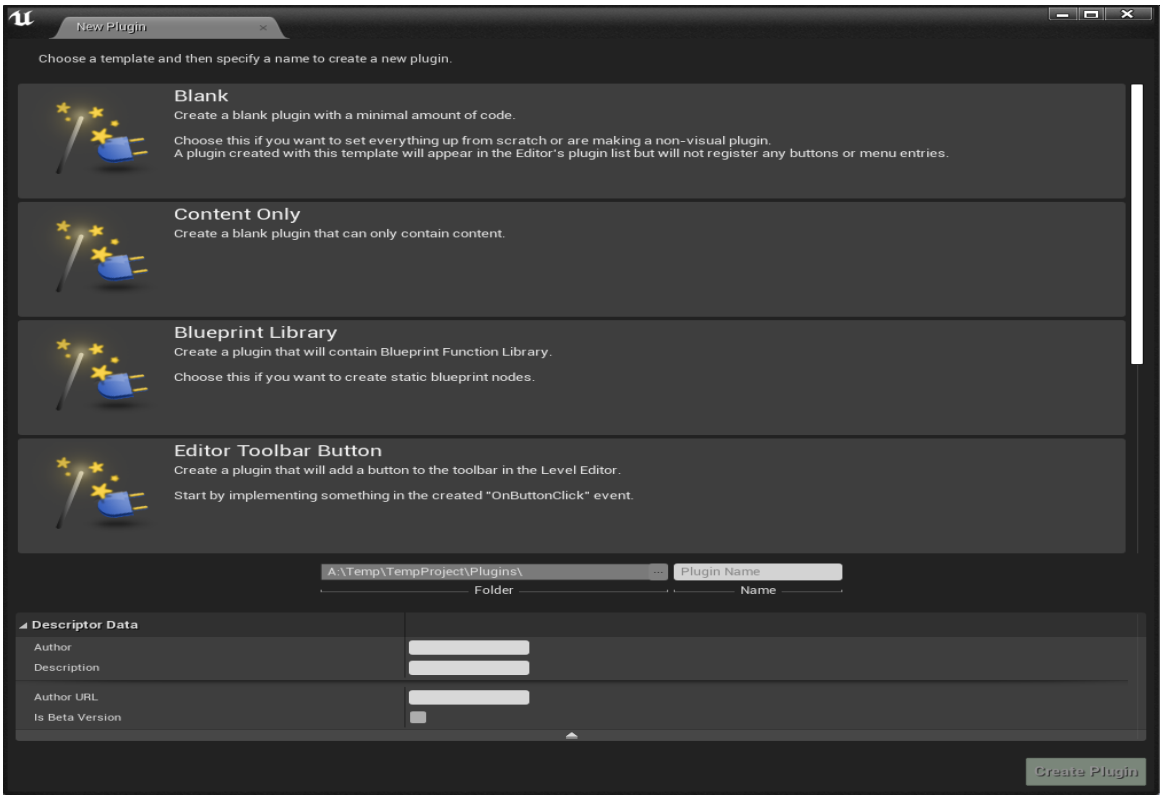


Figure 8. Plug-in creation window in UE4.

The uplugin file contains things such as name and version of the plug-in and contains what modules are part of the plug-in, it also has more additional options such as blacklisting or whitelisting certain OS platforms or programs. The uplugin file is a json file so it can be edited by any text editor. The created plug-in has only one module by default and if more modules are needed, they need to be created manually. The plug-in was created with the name QMissionCreator, so that is also the name of the first module, and second module had to be created. Second one was done

by copying the original module and renaming everything with the QMissionCreator to QMission-CreatorEditor in the new copy, it is good practice to name the editor type modules with the name editor in the end. In the uplugin file the modules needs to be included with the Name and Type and the LoadingPhase which was left at Default. The Type for modules can be Runtime, RuntimeNoCommandlet, Developer, Editor, EditorNoCommandlet, and Program. The original module was left as a Runtime, it contains code that runs for the end user, while the new module was included and had the type set as Editor as seen in Figure 9, an editor type module only runs together with the editor, and this module was for the custom editor related objects and classes. The other types such as Developer would only run for development runtime or editor builds.

Figure 9. The uplugin descriptor file for QMissionCreator plug-in.

The modules each have inside their folder a build.cs file, while UE4 is based on C++ it has some non-runtime features built on C#. These build files mainly include which other modules this module depends on, these modules are added either to the PublicDependencyModuleNames or PrivateDependencyModuleNames. The ones added to the public ones will make that module's classes available for this module's public classes and private one will only be available for the private classes.

With the two modules in place the structure for the plug-in looked like in Figure 10. The following steps could have either been starting to implement the editor or runtime module. The runtime objects in QMissionCreator were selected to be implemented first, because the editor module relies on the runtime module a lot.

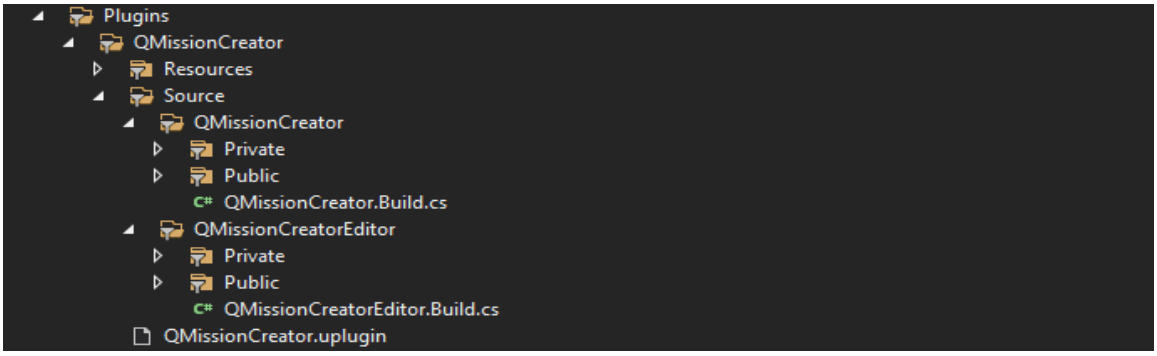


Figure 10. QMissionCreator's folder structure with two modules.

6.1 Mission object

First the main mission object was created and parented from UObject. When creating new classes it is good to make sure that the new classes were created to the correct module, which in this case for the plug-in was the runtime module QMissionCreator as seen in Figure 11.

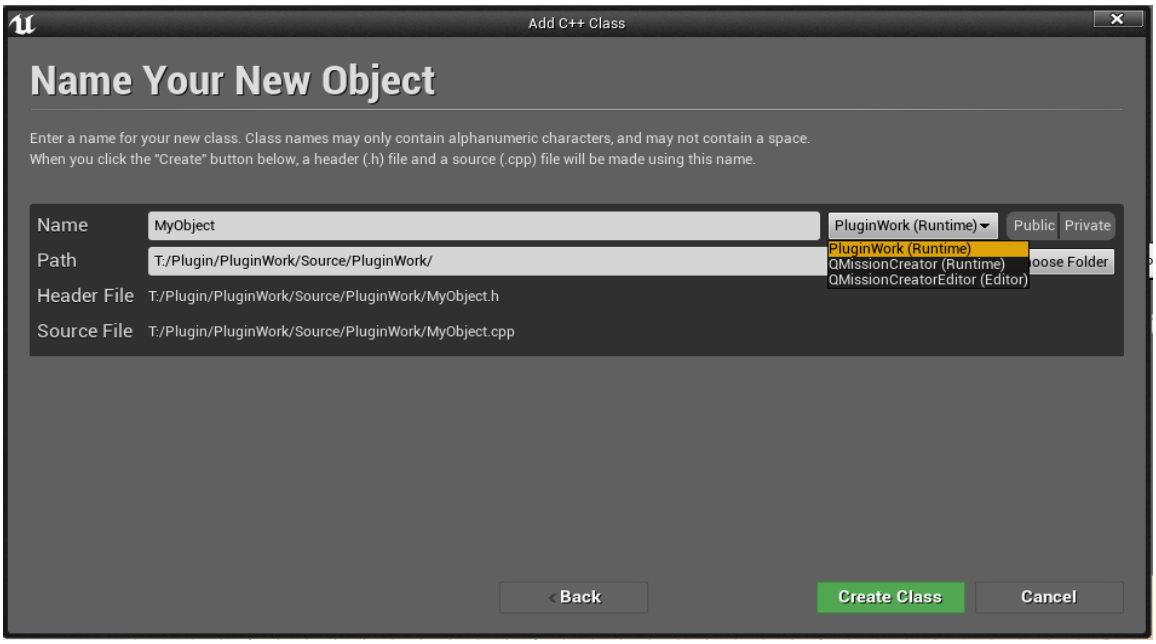


Figure 11. Creating a new C++ class in UE4.

UE4 takes use of multiple macros, three common macros used for a class are the UClass, UPROPERTY and UFUNCTION macros. Macros are something that takes a piece of code one has defined as macro with a name and then when used somewhere with the macro name the compiler replaces the named piece with the code, similarly how a keyboard can have macro keys which can be programmed to do a combination of key presses with just one key. These three macros each have several metadata and variable specifiers that can be used. One commonly used for this plug-in is the DocumentationPolicy metadata used in the bigger classes' UClass macro, when set as Strict this makes the compilation fail whenever there is a missing or duplicate comment in any functions or variables using UFUNCTION or UPROPERTY. The UFUNCTION macro is used on functions in cases such as wanting to use the function with UE4 delegates or wanting to call the function in the Blueprints by using the specifier BlueprintCallable. The UPROPERTY also has multiple use cases, to show a variable and letting access to it in a Blueprint with specifiers such as BlueprintReadOnly, BlueprintReadWrite, VisibleAnywhere, EditAnywhere and multiple more can be used for different purposes. UPROPERTY makes it also possible for serialization for saving, loading and garbage collection. The BlueprintProtected metadata with the URPOPERTY was often used, which makes it possible to have the variable as protected in C++ and still make it available in Blueprints. A protected variable or function makes it so that they are only available for another class directly if it is a child of the parent with the function or variable set as protected. Also, the Category specifier is something commonly used which makes it show up categorized with chosen category name in property details panel for example. Some examples how these were declared are shown in Figure 12.

```
/**
* This is the class for Missions, has a custom graph to easily add the objectives for mission.
*/
UCLASS(Blueprintable, meta = (DocumentationPolicy = "Strict"))
class QMISSIONCREATOR_API UQMission : public UObject

    /** The given name of this mission */
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "MissionType", meta = (BlueprintProtected, MultiLine = "true"))
        FName MissionName = "NameMission";

    /** The description for this mission */
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "MissionType", meta = (BlueprintProtected, MultiLine = "true"))
        FName MissionDescription = "";


    /** Get the mission status */
    UFUNCTION(BlueprintCallable, Category = "QMission")
    const EQMissionStatus GetMissionStatus() const;

    /** Get the mission name */
    UFUNCTION(BlueprintCallable, Category = "QMission")
    const FName GetMissionName() const;
```

Figure 12. Some use cases of the UE4 macros in the QMission class.

QMission class keeps a pointer to base class QMissionBaseNode of the start node, for this pointer to stay saved even after closing the engine or custom graph editor it needs the UPROPERTY macro even if it is just a private variable, and as it is a private variable which means it cannot be accessed from any outside class, it needed a set function SetStartNode. Now the graph editor creating the start node in the graph could set the start node for the mission. When the mission gets first time created, the custom editor creates the UEdGraph object if the mission does not already have one. UEdGraph which is the UE4 graph object itself which keeps all the graph object nodes in the graph saved. Some structs were defined in the header file for the mission class for the save data for mission and nodes, a good place to have the structs in, because the mission header already also has itself but also the node class known. A separate header file was created for some commonly used enums, one of which is the EQMissionStatus enum. This is used both for the objective and mission status, to know status such as is it completed or is it still in progress. This enum declaration has the macro UENUM and uses the BlueprintType specifier, to be able to use it in Blueprints. EQMissionStatus enum is declared as seen in Figure 13.

```
UENUM(BlueprintType)
enum class EQMissionStatus : uint8
{
    NotStarted,
    Completed,
    Failed,
    InProgress,
    InActive,
};
```

Figure 13. Enum for the mission and objective status.

Delegates in UE4 are basically events that one can attach functions to, and those functions will then be run whenever the delegate gets broadcasted, the delegates can have different amount of parameters if wanted, and can also support multiple functions bound. In this plug-in delegates were used both in the runtime and editor module and they were of type multicast, which makes binding multiple functions possible. In the mission class these delegates were used to broadcast when for example the mission was set as inactive or active and when objective was completed or failed. The delegates used the BlueprintAssignable specifier to be able to use these within Blueprints as well, as seen in Figure 14.

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_TwoParams(FQMissionTwoParamSignature, UQMission*, Mission, UQMissionBaseNode*, RequestingNode);

// Delegate signature for when an objective has been completed
UPROPERTY(BlueprintAssignable)
    FQMissionTwoParamSignature OnObjectiveCompleted;

// Delegate called when objective has failed, failure of objective happens when dependency has been added as close that objective
UPROPERTY(BlueprintAssignable)
    FQMissionTwoParamSignature OnObjectiveFailed;

// Delegate signature for when an objective has been started
UPROPERTY(BlueprintAssignable)
    FQMissionTwoParamSignature OnObjectiveStarted;

// On objective become active delegate, called if putting mission as active
UPROPERTY(BlueprintAssignable)
    FQMissionTwoParamSignature OnObjectiveBecomeActive;

// On objective become inactive delegate, called if putting mission as inactive
UPROPERTY(BlueprintAssignable)
    FQMissionTwoParamSignature OnObjectiveBecomeInActive;
```

Figure 14. Delegate declaration with two parameters and five use cases.

The mission class' most essential functions are the start and complete objective, these functions get called from the manager class when starting or completing an objective. The start function takes the manager as an input parameter, this is so that the mission can give the player controller to attached mission events. Both the manager and player controller exists in a world, so they work as the world context object for an UObject which by default does not have any world context. While the mission class also needs a SetMissionManager function to give the manager to the mission in some other way, because the start function is not run when an in-progress mission gets loaded from a save file. Start function also takes in a bool parameter which is for if the mission should be set as inactive when started, this is one feature of the mission, so that a mission can be in an inactive state. Related to this the mission class has another bool variable that decides if the mission and objectives can be completed even if it is in an inactive state or not. In the start function the mission gets initialized to make sure it has default values, and an additional mission data object gets created if one is provided to be used, which can be used for storing and tracking variables belonging to the mission. The rest what the start function does is execute included start events, included inactive events if it was set as inactive and loops through all the child nodes of start node and calls for them to start as seen in Figure 15.

```
// If initally not as active so run the inactivate event scripts, later for each objective too.
if (!bShouldSetActive)
{
    for (const auto InActivateEvent : InActivateEvents)
    {
        if (InActivateEvent)
        {
            InActivateEvent->ExecuteMissionEventNative(this, PlayerController, EQMissionEventCallReason::InActivated);
        }
    }
}



if (StartNode && MissionManager.IsValid())
{
    StartNode->GiveMissionReferencesToAllNodes(this);

    for (const auto Node : StartNode->GetConnectedChildNodes())
    {
        if (UQMissionObjectiveNode* ObjectiveNode = Cast<UQMissionObjectiveNode>(Node))
        {
            ObjectiveNode->StartObjective(MissionManager.Get(), bShouldSetActive);
            NodesInProgress.Add(ObjectiveNode);
        }

    }
}
```

Figure 15. Start function's inactive event execution and objective starting.

The complete objective function takes in a FName as input parameter, which is for the mission's objective name. This objective is looked up from the mission's in-progress nodes and tries complete it if it is possible to be completed, it does nothing if no such objective is found by the given name. If matching objective was found, then it calls the node's end objective function with a completed status. When an objective is completed, the OnObjectiveCompleted delegate gets broadcasted and it has been bound to a function in the mission class, which is the RequestObjectiveChange. This function then gets called, and it checks the dependencies for the children of the completed node and adds the new found child nodes to arrays, and if some nodes were set by some dependency to be closed, those are then added to an array for closing and failing those objectives. If some end node was part of the possible new nodes, then the end node gets priority, and if multiple end nodes were possible, it selects the one with the best priority to end the mission with. If only objectives were found, then those objectives will be added to the in-progress array and the objectives will be started. RequestObjectiveChange function can be seen in Figure 16. Additionally, the mission can have force end events, while usually the mission ends by going to an end node, but the mission can be forced to end from the mission manager, and while doing so the attached end events to an end node would not run, but the force end events will instead run.

```
// store but not start objectives which could be started before confirming no success or fail node that could be started and mission ended which is higher prio
// than starting new objectives when mission would be done anyway

// Requesting node goes to -> parent -> check parents all children again and check for bShouldCloseIfUndone, then close those. But only if some new objective was and could be started, if no new objective could be started
// probably has the IF specific objectives done first then able to start objective, so keep open and close this requesting node as done only.

TArray<UQMissionObjectiveNode*> NewObjectives;
TArray<UQMissionEndNode*> ValidEndNodes;
TArray<UQMissionObjectiveNode*> ToBeClosedNodes;


CollectNodeProcedures(RequestingNode->GetConnectedChildNodes(), NewObjectives, ValidEndNodes, ToBeClosedNodes);

if (ValidEndNodes.Num() > 0)
{
    // Sort end nodes based on priority, lower the higher prio
    ValidEndNodes.Sort([](const UQMissionEndNode& LEndNode, const UQMissionEndNode& REndNode)
        {

            return LEndNode.GetPriority() < REndNode.GetPriority();

        });

    // Try take first one which is lowest prio if not nullptr and prepare ending the mission
    if (UQMissionEndNode* ChosenEndNode = ValidEndNodes[0])
    {
        PrepareMissionEnd(ChosenEndNode);
        return;
    }
}


// No end nodes found, need to check the objective nodes
if (NewObjectives.Num() > 0)
{
    // Close as failed those objectives that had close dependency
    for (const auto CloseNode : ToBeClosedNodes)
    {
        CloseNode->EndObjective(MissionManager.Get(), EQMissionStatus::Failed);
        NodesInProgress.Remove(CloseNode);
        FailedNodes.Add(CloseNode);
    }

    for (const auto NodeInProgress : NodesInProgress)
    {
        if (UQMissionObjectiveNode* ObjectiveNodeInProg = Cast<UQMissionObjectiveNode>(NodeInProgress))
        {
            if (ObjectiveNodeInProg->GetShouldCloseIfUndone())
            {
                ObjectiveNodeInProg->EndObjective(MissionManager.Get(), EQMissionStatus::Failed);
                NodesInProgress.Remove(ObjectiveNodeInProg);
                FailedNodes.Add(ObjectiveNodeInProg);

            }
        }

    }

    for (const auto NewObjective : NewObjectives)
    {
        NewObjective->StartObjective(MissionManager.Get(), !IsInActive());
        NodesInProgress.Add(NewObjective);
    }

    return;
}
```

Figure 16. RequestObjectiveChange function procedure.

6.2 Mission node objects

As the mission works by a node tree, the node which is the root of the tree is saved and held by the mission object, the root in this case is of type QMissionStartNode which is a child class of the QMissionBaseNode. The start node class ended up being empty, so it only functions as cast checking a base node to start node making sure a found node is of type QMissionStartNode. The base node class QMissionBaseNode has variables for storing pointers to parent and child nodes, this is what makes it possible to go back and forward with pointers from the start node. The base class has functions both for removing and adding parent and child nodes, but not only that it also needed functions to clear this node from child and parent nodes because they are also keeping this node in arrays. Like the mission object the nodes have a name and description, the name must be unique in the mission because the completion and saving works based on the node names. In the base node's class header, there is also declared a struct for the dependencies. FQDependencyNode which stores a pointer to type QMissionBaseNode which is the node with a dependency on when this node is hit, and then an enum type, which is what kind of dependency is there on that node, the dependency types can be If, IfNot and Close, as seen in Figure 17.

The If makes so that if the node pointed by DependOnNode is done, it can go to this new being checked node, while the IfNot is opposite, if the DependOnNode is not done it can go to this node and lastly the Close type is that it will close as failed the DependOnNode if it is still not done. The EQMissionDependencyNodeType enum is declared in the earlier mentioned separate header file for commonly used enums. These dependencies are then saved in an array in the node and has set and remove functions for the editor to access and add or remove.

```
/**
 * struct for dependency node, which allows to branch with if and if not the mission structure
 */
USTRUCT(Blueprintable)
struct FQDependencyNode
{
    GENERATED_BODY()

    /** The type of dependency on a objective if completed or if not completed */
    UPROPERTY()
    EQMissionDependencyNodeType Type = EQMissionDependencyNodeType::If;

    /** The pointer to the objective that needs to be checked if undone or done */
    UPROPERTY()
        UQMissionBaseNode* DependOnNode = nullptr;


    bool operator==(const FQDependencyNode& other) const
    {
        return (other.DependOnNode == DependOnNode);
    }
```

Figure 17. The FQDependencyNode struct declaration in QMissionBaseNode.h header file.

Like the mission object the QMissionObjectiveNode and QMissionEndNode were made to have events executable. For the objective, one can add events for on starting the objective and ending, but also in active and become active events. While the objective node has also functions for starting and ending the objective, which basically goes and sets the status of the objective to either failed or completed and executes earlier mentioned events, if events were included. The end node class only has ending events. It also has an ending status and a priority variable, the ending status gets set from the custom editor and depending on if the editor node is a failed or completed node gets set to either completed or failed status. The priority variable is the value used for sorting priority between multiple end nodes as back in Figure 16. The name and description work a bit differently for the end node, the name and description are taken when ending from this node and given to the mission where this can be used for user information such as name of failing or completing the mission can be shown and a description can be shown for why it either failed or completed.

6.3 Mission manager and mission save object

The QMissionManager class which is the mission manager was made inheriting from UActorComponent, this component makes it possible to attach the manager to actors such as the character or pawn actor or player controller. The mission manager's purpose is to manage the missions, this includes loading and saving the in progress, failed and completed missions. Other functions such as starting missions, completing objectives and force ending missions are also part of the mission manager. The mission manager has a variable MaxActiveMissionCount, this is related to the missions being able to be in an in active state, this integer value makes it possible to regulate a max limit of how many missions could be active at once if wanted. One other optionally changed variable is the UsedSaveClass, this can be set to point to another class which inherits from the base mission save class UQMissionSaveGame. In case something else would be wanted to be added additionally to the base save class, with a new child class that can be had.

Like the mission object, the manager was made to have a few delegates, as the manager is basically the owner of a started mission it knows what is going on with the missions. The delegates made for the manager are for, when a mission is started, mission is made in active or active, and when a mission is done or when a mission was loaded from a save. And all the delegates has the mission as parameter, as seen in Figure 18.

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FQMissionManagerOneParamSignature, UQMission*, Mission);

/** The delegate signature for a mission started*/
UPROPERTY(BlueprintAssignable)
    FQMissionManagerOneParamSignature OnMissionStarted;

/** The delegate signature for a mission set as inactive*/
UPROPERTY(BlueprintAssignable)
    FQMissionManagerOneParamSignature OnMissionSetInActive;

/** The delegate signature for a mission set as active*/
UPROPERTY(BlueprintAssignable)
    FQMissionManagerOneParamSignature OnMissionSetActive;

/** The delegate signature for a mission is over */
UPROPERTY(BlueprintAssignable)
    FQMissionManagerOneParamSignature OnMissionDone;

/** The delegate signature for a mission is loaded from save file */
UPROPERTY(BlueprintAssignable)
    FQMissionManagerOneParamSignature OnMissionLoaded;
```

Figure 18. QMissionManager's delegate declarations.

The more important functions such as StartMission, has input parameters for the mission object which is supposed to be started, and a bool value for active state. The bool value is true by default if not giving any value for it. The received mission pointer was duplicated with the DuplicateObject function as not to save any values in the original object, between for example plays in editor. Then some of the mission's objective related delegates are bound to functions in the manager for

some empty functions that can be overridden in a child C++ class or a Blueprint. When OnMissionStarted delegate is broadcasted, mission is added to ActiveMissions array if it is starting as active. The OnMissionEnding is bound in the BeginPlay function to function MissionEnding, this function takes care of removing the completed mission from the arrays and moving it to either the failed or completed array. The manager for the save and load functions uses the UE4's UGameplayStatics class, which is a static class, which means no object needs to be created before accessing the functions. This class has several helper functions, and in this case the SaveGameToSlot and LoadGameFromSlot were used. This saves a USaveGame object to a given slot name and id. But before calling the save function the pointers need to be serialized in to the savegame object, else it will not save everything needed. One way to serialize is to use the FObjectAndNameAsStringProxyArchive struct class and creating a child of it and setting the ArIsSaveGame variable inside it to true, making the archive for saving as seen in Figure 19.

```
struct QMissionSaveGameArchive : public FObjectAndNameAsStringProxyArchive
{
    QMissionSaveGameArchive(FArchive& InInnerArchive)
        : FObjectAndNameAsStringProxyArchive(InInnerArchive, true)
    {
        ArIsSaveGame = true;
    }
};
```

Figure 19. Archive used for serializing declared in the QMissionSaveGame.h header file.

The serialized data in UE4 is stored into TArrays containing uint8 and using FMemoryWrite and the struct from Figure 19. Then by overriding the Serialize function in the class that has things to be serialized and making sure those variables are marked with UPROPERTY and savegame specification, Serialize is then called and given the archive as the argument, as seen in Figure 20.

```
FMemoryWriter MemoryWriter(MissionRecordData.MissionData, true);
QMissionSaveGameArchive SaveAr(MemoryWriter);
MissionToSerialize->Serialize(SaveAr);
```

Figure 20. Process of serializing data from the mission into the MissionData.

This is similarly repeated for each node part of the mission and the additional save game data object if one is used within the mission. And the order when saving and loading the different objects needs to be same, because they are in a specific order within the data array.

The loading function first needs to create back the mission objects before serializing the data back into the objects from the saved data, this can be done in multiple ways. In this case, it was done by keeping stored the original mission object which was received when starting a mission in the manager before duplicating it. When the object is created back instead of the FMemoryWrite

class used when saving, now the FMemoryReader is used as seen in Figure 21. Both the FMemoryReader and FMemoryWrite is available because of the FObjectAndNameAsStringProxy-Archive. The node objects already exist in the original mission object, so they do not have to be recreated except the saved data loaded similarly with the FMemoryReader.

```
FMemoryReader MemoryReader(LoadData->MissionData, true);
QMissionSaveGameArchive Ar(MemoryReader);
MissionToRestore->Serialize(Ar);
```

Figure 21. Process of deserializing back the data from MissionData into the mission.

6.4 Editor module setup, style, and asset type actions

Each module needs to be able to be started and shutdown. When the plug-in was created it also received a header and source file named the same as the original module, this is good to rename to have module in the end of the name, so it is easier to know this is the file that implements the IModuleInterface. The IModuleInterface contains the StartupModule and ShutdownModule functions that when overridden can be used to initialize needed things, in this case the StartupModule function was set to initialize the editor style and register the asset type actions as seen in Figure 22. And on the ShutdownModule function the asset type actions had to be unregistered when the module is shutting down.

```
void QMissionCreatorEditor::StartupModule()
{

    // Initialize the editor style, which loads the textures from resource folder and set fonts and so on.
    QMissionCreatorEditorStyle::Initialize();
    QMissionCreatorEditorStyle::ReloadTextures();

    IAssetTools& AssetTools = FModuleManager::LoadModuleChecked<FAssetToolsModule>("AssetTools").Get();

    // Register asset types, which are actions such as opening an asset or right clicking and selecting some
    RegisterAssetTypeAction(AssetTools, MakeShareable(new QAssetTypeActions_QMission(MissionAssetCategoryBit)));
    RegisterAssetTypeAction(AssetTools, MakeShareable(new QAssetTypeActions_QMissionEvent(MissionAssetCategoryBit)));
    RegisterAssetTypeAction(AssetTools, MakeShareable(new QAssetTypeActions_QAdditionalMissionData(MissionAssetCategoryBit)));


}
```

Figure 22. StartupModule function, editor style initialized, and asset type actions registered.

The editor module required access to additional modules, such as the mission's runtime module QMissionCreator, UnrealEd, AssetTools, GraphEditor and PropertyEditor so these were added to the module's build.cs file as seen in Figure 23, to get access to their classes.

```
PublicDependencyModuleNames.AddRange(
    new string[]
    {
        "Core",
        "CoreUObject",
        "Engine",
        "UnrealEd",
        // ... add other public dependencies that you statically link with here ...
    }
    );


PrivateDependencyModuleNames.AddRange(
    new string[]
    {

        "ContentBrowser",
        "QMissionCreator",
        "KismetWidgets",
        "InputCore",
        "Projects",
        "AssetTools",
        "Slate",
        "SlateCore",
        "GraphEditor",
        "PropertyEditor",
        "EditorStyle",
        "Kismet",
        "KismetWidgets",
        "ApplicationCore",
        "ToolMenus",
        "EditorWidgets",
        "DMXEditor",

    }
    );
```

Figure 23. QMissionCreatorEditor.build.cs additional public and private dependency modules.

Unlike the runtime module's inside UE4 created classes, the header and source files for the classes in the editor module were created directly from Visual Studio. The style class is created so that it has static functions. The style instance is created from Initialize function which is seen called in Figure 22. After the style instance is created a FSlateStyleSet object is also created, and that object can then be pointed to the Resources folder in the plug-in directory. The object has set functions for setting different styles into the style set, for this plug-in some textures for the nodes were set, such as for the background of the nodes, and pin input or outputs as seen in Figure 24.

```
TSharedRef< FSlateStyleSet > QMissionCreatorEditorStyle::Create()
{
    TSharedRef< FSlateStyleSet > Style = MakeShareable(new FSlateStyleSet("QMissionCreatorStyle"));

    Style->SetContentRoot(IPluginManager::Get().FindPlugin("QMissionCreator")->GetBaseDir() / TEXT("Resources"));

    const FVector2D Icon16x16(16.0f, 16.0f);
    const FVector2D Icon20x20(20.0f, 20.0f);
    const FVector2D Icon32x32(32.0f, 32.0f);
    const FVector2D Icon40x40(40.0f, 40.0f);
    const FVector2D Icon64x64(64.0f, 64.0f);
    const FVector2D Icon64x6(64.0f, 6.0f);
    const FVector2D Icon6x64(6.0f, 64.0f);


    // The mission icon on the created asset in content browser and in corner when opened
    Style->Set("ClassThumbnail.QMission", new IMAGE_BRUSH(TEXT("QMissionClassThumbnail"), Icon64x64));
    Style->Set("ClassIcon.QMission", new IMAGE_BRUSH(TEXT("QMissionClassIcon"), Icon16x16));

    // The mission event icon on the created asset in content browser and in corner when opened
    Style->Set("ClassThumbnail.QMissionEvent", new IMAGE_BRUSH(TEXT("QMissionEventClassThumbnail"), Icon64x64));
    Style->Set("ClassIcon.QMissionEvent", new IMAGE_BRUSH(TEXT("QMissionEventClassIcon"), Icon16x16));

    // The additional mission data icon on the created asset in content browser and in corner when opened
    Style->Set("ClassThumbnail.QAdditionalMissionData", new IMAGE_BRUSH(TEXT("QAdditionalMissionDataClassThumbnail"), Icon64x64));
    Style->Set("ClassIcon.QAdditionalMissionData", new IMAGE_BRUSH(TEXT("QAdditionalMissionDataClassIcon"), Icon16x16));

    // Node background colors
    Style->Set("QMissionEditor.NodeBackground_Blue", new BOX_BRUSH(TEXT("NodeBackgroundBlue"), FMargin(4.f / 16.f)));
    Style->Set("QMissionEditor.NodeBackground_DarkGreen", new BOX_BRUSH(TEXT("NodeBackgroundDarkGreen"), FMargin(4.f / 16.f)));
    Style->Set("QMissionEditor.NodeBackground_Red", new BOX_BRUSH(TEXT("NodeBackgroundRed"), FMargin(4.f / 16.f)));
    Style->Set("QMissionEditor.NodeBackground_Orange", new BOX_BRUSH(TEXT("NodeBackgroundOrange"), FMargin(4.f / 16.f)));
    Style->Set("QMissionEditor.NodeBackground_Green", new BOX_BRUSH(TEXT("NodeBackgroundGreen"), FMargin(4.f / 16.f)));
    Style->Set("QMissionEditor.NodeBackground_Grey", new BOX_BRUSH(TEXT("NodeBackgroundGrey"), FMargin(4.f / 16.f)));

    // Node background arrows input areas
    Style->Set("QMissionEditor.ArrowTopInput", new IMAGE_BRUSH(TEXT("NodeArrowTopInput"), Icon64x6));
    Style->Set("QMissionEditor.ArrowLeftInput", new IMAGE_BRUSH(TEXT("NodeArrowLeftInput"), Icon6x64));
    Style->Set("QMissionEditor.ArrowRightInput", new IMAGE_BRUSH(TEXT("NodeArrowRightInput"), Icon6x64));
    Style->Set("QMissionEditor.ArrowBottomOutput", new IMAGE_BRUSH(TEXT("NodeArrowBottomOutput"), Icon64x6));
    Style->Set("QMissionEditor.ArrowLeftOutput", new IMAGE_BRUSH(TEXT("NodeArrowLeftOutput"), Icon6x64));
    Style->Set("QMissionEditor.ArrowTopOutput", new IMAGE_BRUSH(TEXT("NodeArrowTopOutput"), Icon64x6));
    Style->Set("QMissionEditor.ArrowRightOutput", new IMAGE_BRUSH(TEXT("NodeArrowRightOutput"), Icon6x64));

    // Icons for nodes to recognize node types
    Style->Set("QMissionEditor.NodeIcon.Start", new IMAGE_BRUSH(TEXT("NodeIconStart"), Icon20x20));
    Style->Set("QMissionEditor.NodeIcon.Objective", new IMAGE_BRUSH(TEXT("NodeIconObjective"), Icon20x20));
    Style->Set("QMissionEditor.NodeIcon.Completed", new IMAGE_BRUSH(TEXT("NodeIconCompleted"), Icon20x20));
    Style->Set("QMissionEditor.NodeIcon.Failed", new IMAGE_BRUSH(TEXT("NodeIconFailed"), Icon20x20));
```

Figure 24. Style set creation and setting brushes.

The actions registered in Figure 22, are based on the FAssetTypeActions_Base class. Only the mission has a custom editor, but the event and additional data classes still both have classes that inherits the FAssetTypeActions_Base and a factory class inheriting from UFactory, to get them all to show up in miscellaneous category as seen in Figure 25.

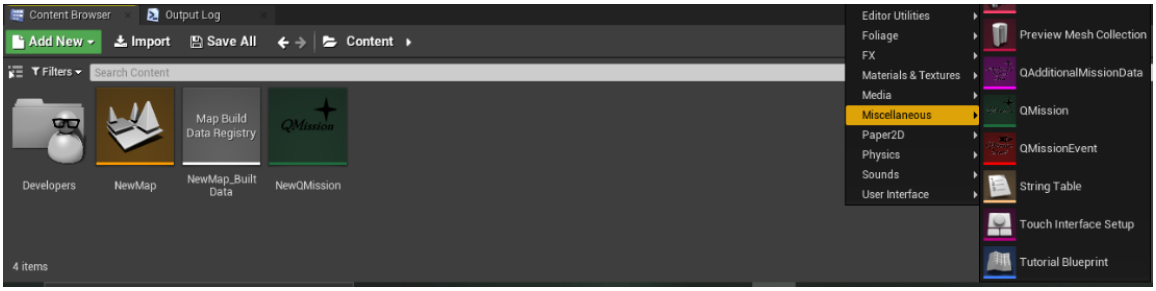


Figure 25. This plug-in's classes shown up in the create advanced asset miscellaneous category.

The UFactory has multiple functions that can be overridden, in this case in the constructor, the SupportedClass variable is set to the object that it supports, and bCreateNew and bEditNew are set to true, so new ones can be created and new on is opened. The FactoryCreateNew function is the only one that needs to be overridden. It is made to call from the static helper utility class

FKismetEditorUtilities the function CreateBlueprint, which takes some input parameters and basically creates then a Blueprint for the asset. This is used for the event and additional data class but for the mission which does not use a normal Blueprint, it will just instead be creating a new object using the NewObject function as seen in Figure 26.

```
UObject* UQMissionEvent_DataFactory::FactoryCreateNew(UClass* InClass, UObject* InParent, FName InName, EObjectFlags Flags, UObject* Context, FFeedbackContext* Warn, FName CallingContext)
{
    return FKismetEditorUtilities::CreateBlueprint(InClass, InParent, InName, BPTYPE_Normal, UBlueprint::StaticClass(), UBlueprintGeneratedClass::StaticClass(), CallingContext);
}

UObject* UQMission_DataFactory::FactoryCreateNew(UClass* Class, UObject* InParent, FName Name, EObjectFlags Flags, UObject* Context, FFeedbackContext* Warn)
{
    return NewObject<UQMission>(InParent, Class, Name, Flags | RF_Transactional);
}
```

Figure 26. FactoryCreateNew function, mission's factory below and mission event use the above.

The AssetTypeAction classes has more functions overridden than the UFactory based classes, the GetCategories function is overridden to return the category variable, which was received from the constructor, seen as being given when registering action in Figure 22. This category is an enum variable and the type being set as Misc therefore the asset is then visible in the miscellaneous. GetSupportedClass is overridden and made to return the respective class for each asset type. The thing that differs from the mission asset and the rest is that the AssetTypeAction for the mission also overrides the OpenAssetEditor function, this is where class created for the custom editor gets created as seen in Figure 27. The object is tried to cast to the mission object, and if valid the FQMissionCreatorEditor class is created, and its initialized function is called, the important part is to give the mission object that is being opened to the editor as argument.

```
void QAssetTypeActions_QMission::OpenAssetEditor(const TArray<UObject*>& InObjects, TSharedPtr<class IToolkitHost> EditWithinLevelEditor)
{
    // This is where the QMission objects gets instead of default blueprint editor, opens the custom graph editor instead
    const EToolkitMode::Type Mode = EditWithinLevelEditor.IsValid() ? EToolkitMode::WorldCentric : EToolkitMode::Standalone;
    for (const auto Object : InObjects)
    {
        if (UQMission* Mission = Cast<UQMission>(Object))
        {
            TSharedRef<FQMissionCreatorEditor> NewQMissionCreatorEditor(new FQMissionCreatorEditor());
            NewQMissionCreatorEditor->InitializeMissionCreator(Mode, EditWithinLevelEditor, Mission);
        }
    }
}
```

Figure 27. AssetTypeAction's function for mission object creating custom editor and calling initialize.

6.5 Mission's custom editor

The editor class was done to inherit from UE4's FAssetEditorToolkit class, this class has several functions related to creating the editor, it has for example the registering and unregistering of

tab spawners, which are overridden in this mission editor, to create the graph viewport and property editor widgets. Also, the class is made to inherit from the FNotifyHook, and as the name implies it has functions that can be overridden such as NotifyPostChange that makes it possible to know for example if a variable has been modified. This overridden function was made to check if the variable which changed was the mission object, and then updates the start node of the graph. Because the start node is showing information of the mission object, so it is possible that the start node is showing old mission name and description if those were changed there the update is necessary. The initialize function, which is called in the Figure 27, is saving the mission pointer received in the editor class, if the mission's UEdGraph pointer is a nullptr a new UEdGraph is created, and it is of type UQMissionCreatorEdGraph which inherits from the UEdGraph. This graph has an initialize function which is called after being created in the editor initialize function, this function creates the start node and initializes it as seen in Figure 28.

```
void UQMissionCreatorEdGraph::InitializeGraph(UQMission* Mission)
{

    // Creating the startnode for the graph, which is the node where user can start adding objectives nodes to, start node can't be removed
    if (UQMissionGraphStartNode* StartNode = NewObject<UQMissionGraphStartNode>(Mission))
    {
        StartNode->Rename(NULL, this, REN_NonTransactional);
        AddNode(StartNode, true, false);

        StartNode->CreateNewGuid();
        StartNode->PostPlacedNewNode();
        StartNode->AllocateDefaultPins();
        StartNode->NodePosX = 0;
        StartNode->NodePosY = 0;
        StartNode->SnapToGrid(GridSnapSize);
        StartNode->SetStartNodeForMission(Mission);
    }

}
```

Figure 28. Initializes the custom UEdGraph, by creating a start node object and initializing it.

From the editor's initialize function the SGraphEditor widget is also created and its GraphToEdit variable is set to mission's UEdGraph object. Here is also the commands available registered to the widget, in this case only the delete command is mapped, which makes it possible to delete nodes in the graph. In the custom UEdGraph's constructor the Schema is set to point to custom created schema class of type UEdGraphSchema. This schema is responsible of creating the context menu actions when right clicking and making possible to create new nodes into the graph. The schema also has the CreateConnectionDrawingPolicy function, which is set to return the custom created drawing policy class of type FConnectionDrawingPolicy. This policy class for example sets what colours and how the splines are drawn between the connections. The schema then has the CanCreateConnection function, which sets how the connection are possible to be created. In this mission graph the connections were made possible only between input and output, and from dependency pin to dependency pin and normal pin to normal pin.

6.6 Mission graph nodes

Similarly like the runtime nodes, the editor nodes were made to have a base class, objective, start, failed, and completed nodes. Here instead of one end node, they were separated, and both sets the owner runtime nodes status to either failed or completed and they get different node backgrounds. The editor module got another node type which is only in the editor, the dependency node which has a different look from the other nodes, and it does not create a runtime module representation node. These graph nodes inherit from the UEdGraphNode class, and in the constructors of these editor nodes the actual runtime node is constructed. The base node has a lot of functions, it handles the background images of the nodes, and has function that gets called when node gets destroyed so the connection can be cleared from connected nodes and remove all the references to the destroyed node.

6.7 Node widgets and pins

The node slate widgets are based on the SGraphNode, all nodes except the dependency node is based on same class. The dependency node has output pin areas in each side and a button together with a ListView widget for adding and showing dependencies added to that dependency node. This when attached to another objective, failed, or completed node will add from that dependency node the dependencies to that runtime owner node of that editor node. The pin slate widgets are based on the SGraphPin class, here a base pin and a dependency pin class is made, to differentiate the connections between objective, start, failed, and completed nodes, and connections from a dependency nodes output nodes.

7 Testing

For testing the mission creator, a set of test mission assets were created, to thoroughly test all the features included. To be able to see the information required and confirm the mission is working a simple UI was setup as seen in Figure 29. This was an actor created and placed in the world, which had a widget component attached to it, and widget class set to a widget that was created.

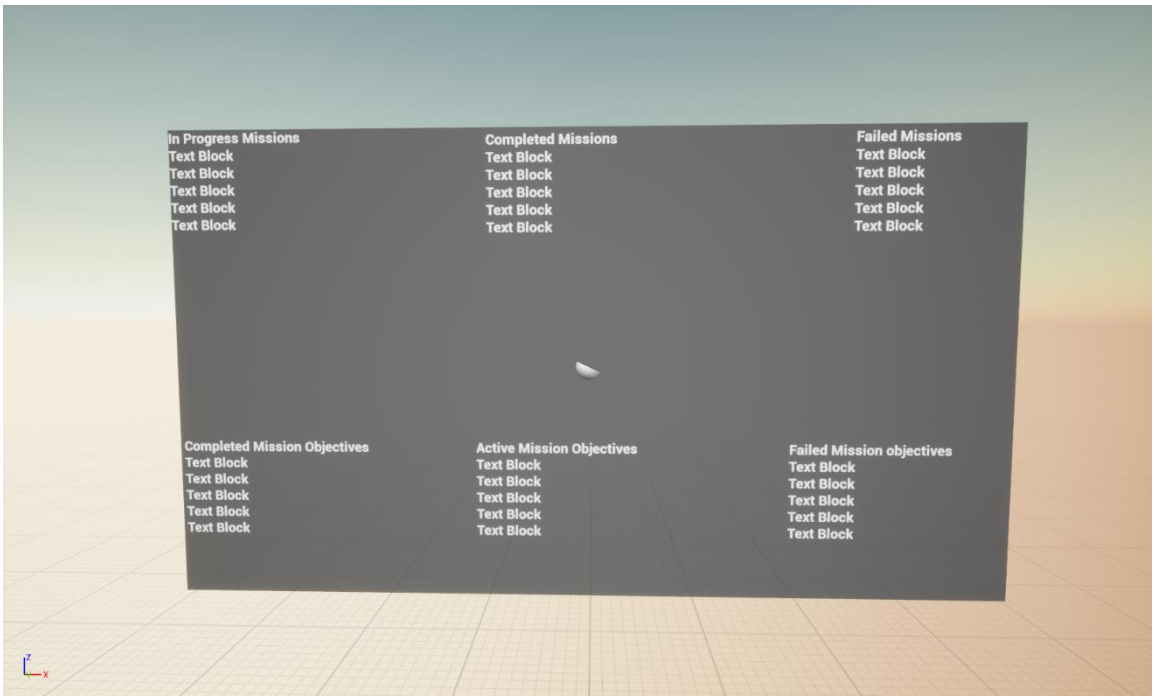


Figure 29. Actor in the world with a widget showing mission information.

The widget was made to have a function with the mission manager as an input parameter, so that the mission manager that is starting the missions could be given to the widget. For easy setup, the widget was given the manager from the level Blueprint, where one can easily take the actor in the world and reference it in the Blueprint, and the player controller is simply received by the GetPlayerController function which is then cast to the created player controller class. In the widget, functions were bound to delegates from the mission manager, such as OnMissionStarted and OnMissionEnded, then these were added to add the mission's name to the textbox for respective event. And additionally, from the OnMissionStarted, when it executes the started mission's delegates would be used such as OnObjectiveStarted and OnObjectiveEnded, those were also bound to functions that shows the names of the objectives in the textboxes.

A player controller was created, and a mission manager component was attached to it. Then on different keyboard button prompts a test mission would be started, and the mission's objectives would correspond to a certain keyboard key. By pressing a key it would try to complete an objective with the name of the key. Additionally, to the keyboard, oculus touch controllers were also bound to these, to test that it also works on the Oculus Quest series running on Android. As the runtime module's classes were not tested before finishing the editor module, a few issues were discovered by the testing and those were fixed, such as android build not compiling because of shadowing variables in function parameter names, as the compiling for android is stricter than what runs on Windows. Other minor issues such as forgotten savegame specifier on some status variables in mission and objective class were discovered when saving and loading was tested.

8 Conclusion

The purpose was to create a plug-in that makes it possible to create missions in Unreal Engine 4, at first some background about plug-ins, UE4 and missions were brought up. In the implementation some of the key classes and functionality of this plug-in were explained.

The process of making this plug-in went as planned, some more difficulties was met with the serialization of the missions and expanding the editor. But overall, the plug-in ended up quite well and functioning.

Something more to add to the implementation if time would have allowed, would have perhaps been more features in the editor module, such as being able to copy and paste nodes, and even dragging missions into another mission, combining them into a larger mission. Maybe a compile button for making sure the mission graph nodes are setup properly so that the mission will work, because now it is for example possible to leave out an ending from a mission which causes a mission to never be completed without forcing it to be completed from the manager.

References

(1) Menn J. Exclusive: Massive spying on users of Google's Chrome shows new security weakness. 2020; Available at: https://www.reuters.com/article/us-alphabet-google-chrome-exclusive-idUSKBN23P0JO.

(2) The history of Photoshop. 2005; Available at: https://www.creativebloq.com/adobe/history-photoshop-12052724.

(3) Sterne J. Plug-in. 2019; Available at: https://www.britannica.com/technology/plug-in.

(4) Leypoldt T. What is Virtual Studio Technology (VST)? Available at: https://andyaxmusic.com/what-is-virtual-studio-technology-vst/.

(5) The Steinberg Story. Available at: https://www.steinberg.net/en/company/aboutsteinberg.html.

(6) VST3: New Standard for Virtual Studio Technology. Available at: https://www.steinberg.net/en/company/technologies/vst3.html.

(7) Murphy CJ. What Is a DAW (And What Can You Do With It)? 2020; Available at: https://www.careersinmusic.com/what-is-a-daw/.

(8) Mayeda R. HBO's Westworld turns to Unreal Engine for in-camera visual effects. 2020; Available at: https://www.unrealengine.com/en-US/spotlights/hbo-s-westworld-turns-to-unreal-engine-for-in-camera-visual-effects.

(9) Haj-Assaad S. Here's Why the Unreal Engine Is Coming to Your Car. 2020; Available at: https://www.techspot.com/article/2130-unreal-engine-in-your-car/.

(10) Drake J. 15 Great Games That Use The Unreal 4 Game Engine. 2020; Available at: https://www.thegamer.com/great-games-use-unreal-4-game-engine/.

(11) Farris J. Forging new paths for filmmakers on "The Mandalorian". 2020; Available at: https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian.

(12) McDonald L. The scoop from a major player. 1998; Available at: https://books.google.de/books?id=1AEAAAAAMBAJ&q=Tim+Sweeney+epic&pg=PT8&redir_esc=y#v=snippet&q=Tim%20Sweeney%20epic&f=false.

(13) Thomsen M. History of the Unreal Engine. 2010; Available at: https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine.

(14) Grove R. Focus: Unreal Engine - A Brief History of Unreal. 2019; Available at: https://magazine.renderosity.com/article/5330/focus-unreal-engine-a-brief-history-of-unreal.

(15) Unreal Promo Art. Available at: https://www.mobygames.com/game/unreal/promo/imageType,1/promoImageId,75491/.

(16) Brightman J. An Epic Interview With Tim Sweeney. 2012; Available at: https://www.gamesindustry.biz/articles/2012-03-13-an-epic-interview-with-tim-sweeney.

(17) Sweeney T. Welcome to Unreal Engine 4. 2014; Available at: https://www.unrealengine.com/en-US/blog/welcome-to-unreal-engine-4.

(18) Sweeney T. If You Love Something, Set It Free. 2015; Available at: https://www.unrealengine.com/en-US/blog/ue4-is-free#:~:text=Unreal%20Engine%204%20is%20now,to%20VR%2C%20film%20and%20animation.

(19) Assassin's Creed Odyssey Story Creator Mode User Manual. Available at: https://ubistatic-a.akamaihd.net/0130/PROD/v1/docs/user-manual.pdf.

(20) REDkitEditor –Designer Basic Manual. 2018; Available at: https://nanopdf.com/download/redkiteditorbasicmanual-port-of_pdf.

(21) How to create your adventure in REDkit. 2012; Available at: https://www.youtube.com/watch?v=I-CEQFvsiTs.

(22) Hardware and Software Specifications. Available at: https://docs.unrealengine.com/en-US/Basics/RecommendedSpecifications/index.html.

(23) How to create Unreal Engine plugins. Available at: https://docs.unrealengine.com/en-US/ProductionPipelines/Plugins/index.html.