Gamification of Fjell Fortress

System Documentation

Version <1.0>

Dokumentet er basert på Systemdokumentasjon utarbeidet ved NTNU. Revisjon og tilpasninger til bruk ved IDER, DATA - INF utført av Carsten Gunnar Helgesen, Svein - Ivar Lillehaug og Per Christian Engdal. Dokumentet finner også i engelsk utgave.

Date	Version	Description	Author
19.05.2022	<1.0>	Creating figures for architecture and structures, as well as explaining how these work. Project set up, project installment, code structure	Griffin Retzius, Simon Vaular and Oneal Lane

Table of contents

1. Introduction

The main purpose of this document is to further examine and explain Fjell Festning architecture and system documentation. The document will also discuss the game’s class diagram and architectural models. The main purpose of the document is letting the reader understand how the project is set up and how the main architectural concepts of Fjell are designed, these consist of the Event System and Node Parser. Other aspects of the project will also be discussed, for example the class diagram and NPC Handler.

Further, the system documentation will introduce some of the game's code, how the project files are structured and how the project/game can be installed.

2. Architecture

This chapter will include the relevant systems used in the game and how these are structured with the help of figures for further explanation.

2.1 Event Manager

The Event Manager is a system built for maintaining the game's state at all times during runtime. The manager is a script that attaches itself to an Empty GameObject in the scene. After the script is attached, the GameObject's inspector makes it possible to set the Game State to whatever the developer desires. This is done by making the GameState variable public in the script, which lets the developer choose the state the game should start in.

In Fjell Festning, each state changes after the player has finished a dialogue. There are 12 dialogue interactions, therefore 12 states were added, as well as the "StartGame" and "EndGame" state. This lets the developer alter the flow of the game, making decisions based on how far the player has come in the scene.

GameObjects like the MapHandler and NPCHandlingScript subscribe to the Event Manager in their own respected scripts. When a state changes, these GameObjects will know if a state has changed and can check if the state that changed is necessary for their actions. If not, it will just listen for the next state change.

The NodeParser and Gatekeeper can publish a new state, by changing the game's state from the player's actions through the script attached to that GameObject. After conversing with an interactable character, the NodeParser will determine what the state will change to. The developer can determine the next state for a character by using the GameObjects inspector. The script designed by the team lets the developer choose the required state needed to converse with a character and they can also choose what the state is changed to after the dialogue is finished.

Other GameObjects with scripts attached also listen to the Event Manager by subscribing to it. The spitfire, group of German soldiers, and trucks all listen for a state to be changed. When the state changes to their desired function, it will then execute the function.

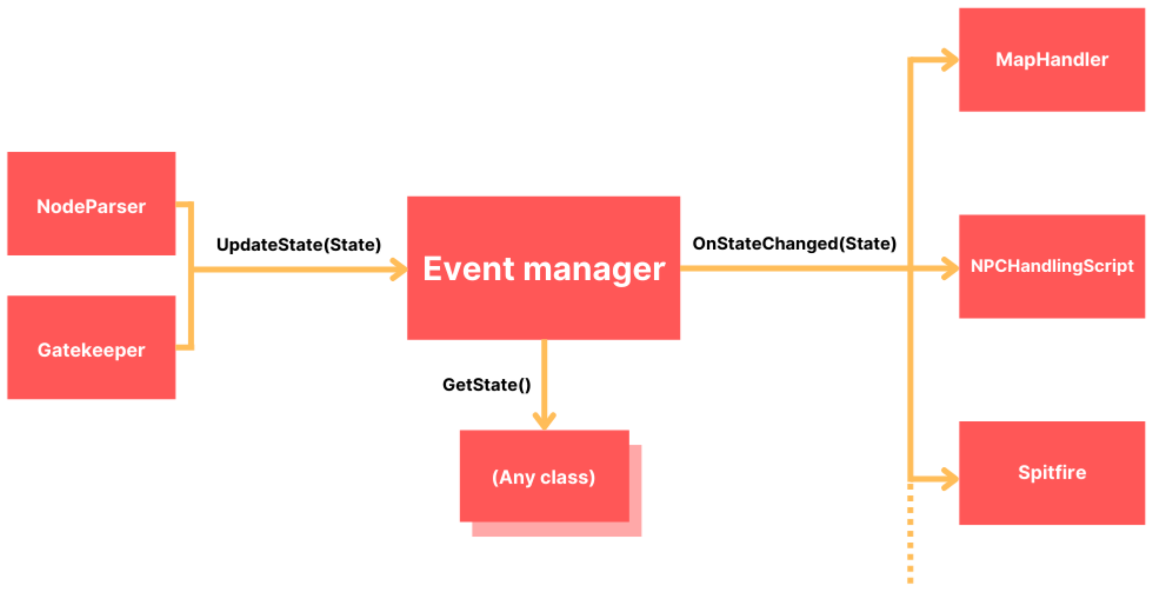


Figure 1: Structure of the event system, with publishers on the left, and subscribers on the right.

2.2 xNode and Dialogue System

xNode is a system installed from the Unity Asset Store for developing the dialogue system and was remade from scratch with the help of the xNode framework. It made it easy to create a custom node based dialogue system where the developer can create custom nodes for a specific task. Here, the NodeParser script is used to parse through the different nodes, processing the data that is inside that node. The framework does not come with a premade set of nodes, as well as it does not include a parser that parses the data. It does help the developer with the necessary supplies to create these nodes.

The node parser script is responsible for cycling through the nodes and processing their data. It starts by finding the node labeled start, and uses that as the starting node. The name of the nodes determine the action to be performed. When a question node is parsed, the parser tells the UI to show the buttons, and hide every other UI element. The buttons are then populated so that each button represents one question. Each button is linked to a specific output port, and fires an event when clicked, telling the parser to follow that particular output. The buttons are then hidden.

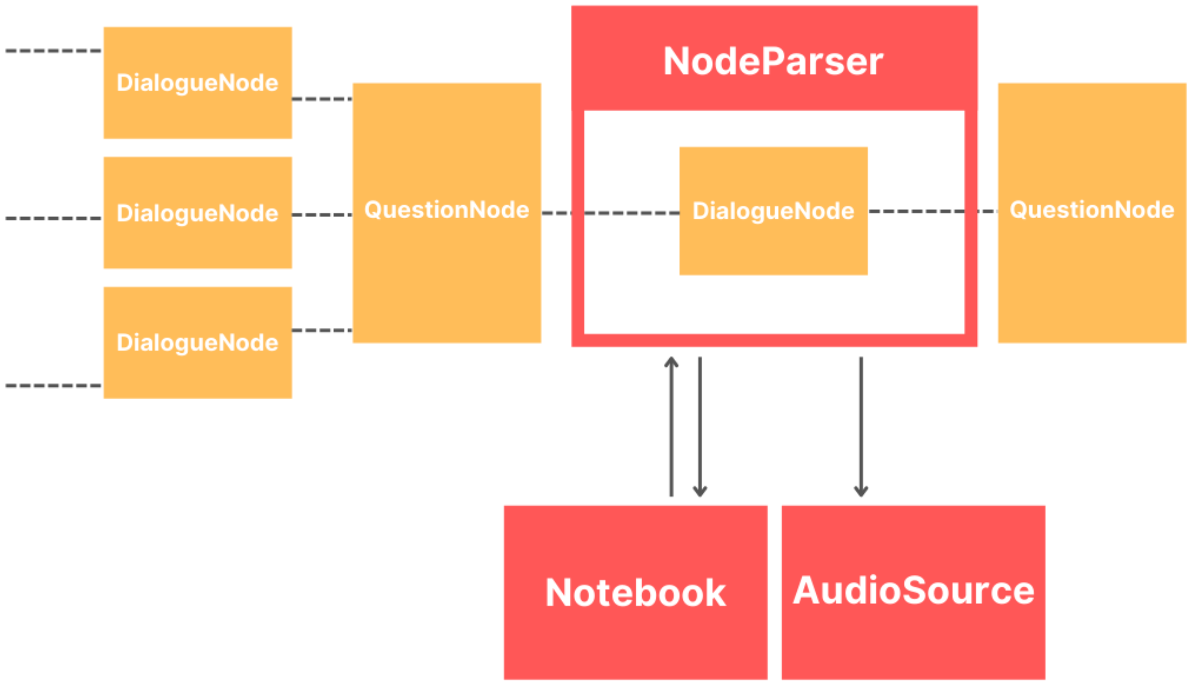
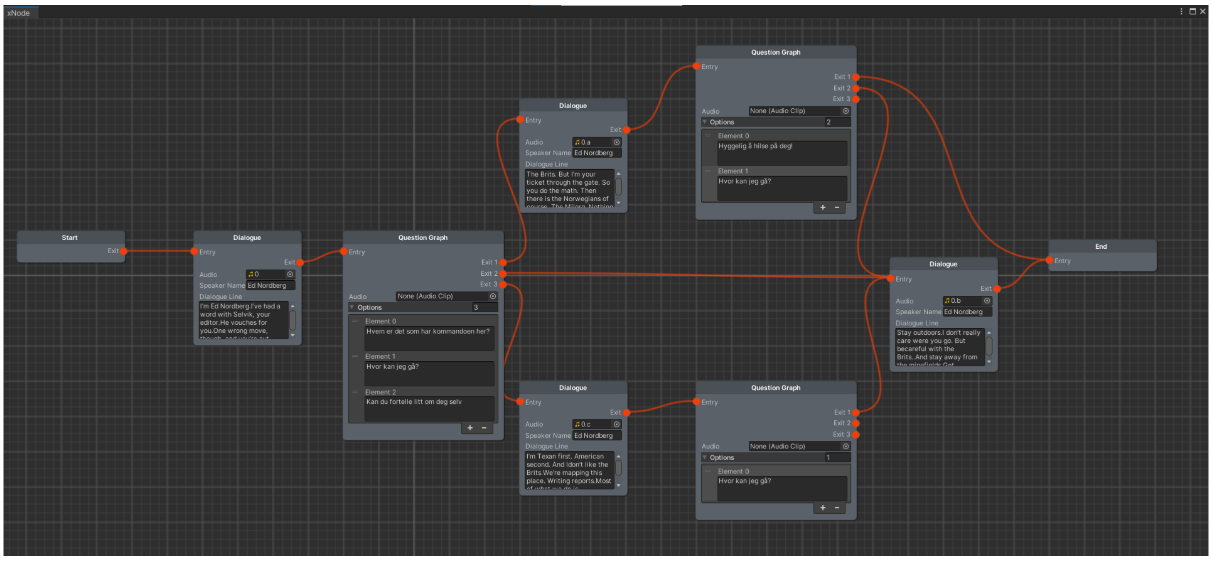


Figure 2: Node Parser visualization

The nodes in Fjell consist of a Start, Question, Dialogue and End node. These work together to create a complete dialogue graph. The xNode framework also makes it possible for the developer to add nodes and connect these via their ports in a vizual graph system. Nodes can be added easily by the developer by right clicking the graph and selecting the desired node. It also lets the developer add data needed inside of these nodes. Nodes can be connected by dragging the output port towards an input port creating an edge between them.

Question nodes contain a list of questions. The amount of questions is decided in the node graph. The node does not have an upper limit to the number of questions, but the current parser script will only display as many questions as can fit on the UI.

A dialogue node features an audio clip field for the dialogue audio, and a field for the text version of the dialogue. In the script these are represented as public variables. It contains a single input and output port.



Dialogue
Audio: 0
Speaker Name: Ed Nordberg
Dialogue Line: I'm Ed Nordberg.I've had a word with Selvik, your editor.He vouches for you.One wrong move,

Question Graph
Audio: None (Audio Clip)
Options: 3
Element 0: Hvem er det som har kommandoen her?
Element 1: Hvor kan jeg gå?
Element 2: Kan du fortelle litt om deg selv

Dialogue
Audio: 0.a
Speaker Name: Ed Nordberg
Dialogue Line: The Brits. But I'm your ticket through the gate. So you do the math. Then there is the Norwegians of

Question Graph
Audio: None (Audio Clip)
Options: 2
Element 0: Hyggelig å hilse på deg!
Element 1: Hvor kan jeg gå?

Dialogue
Audio: 0.c
Speaker Name: Ed Nordberg
Dialogue Line: I'm Texan first. American second. And Idon't like the Brits.We're mapping this place. Writing reports.Most

Question Graph
Audio: None (Audio Clip)
Options: 1
Element 0: Hvor kan jeg gå?

Dialogue
Audio: 0.b
Speaker Name: Ed Nordberg
Dialogue Line: Stay outdoors.I don't really care were you go. But becareful with the Brits..And stay away from

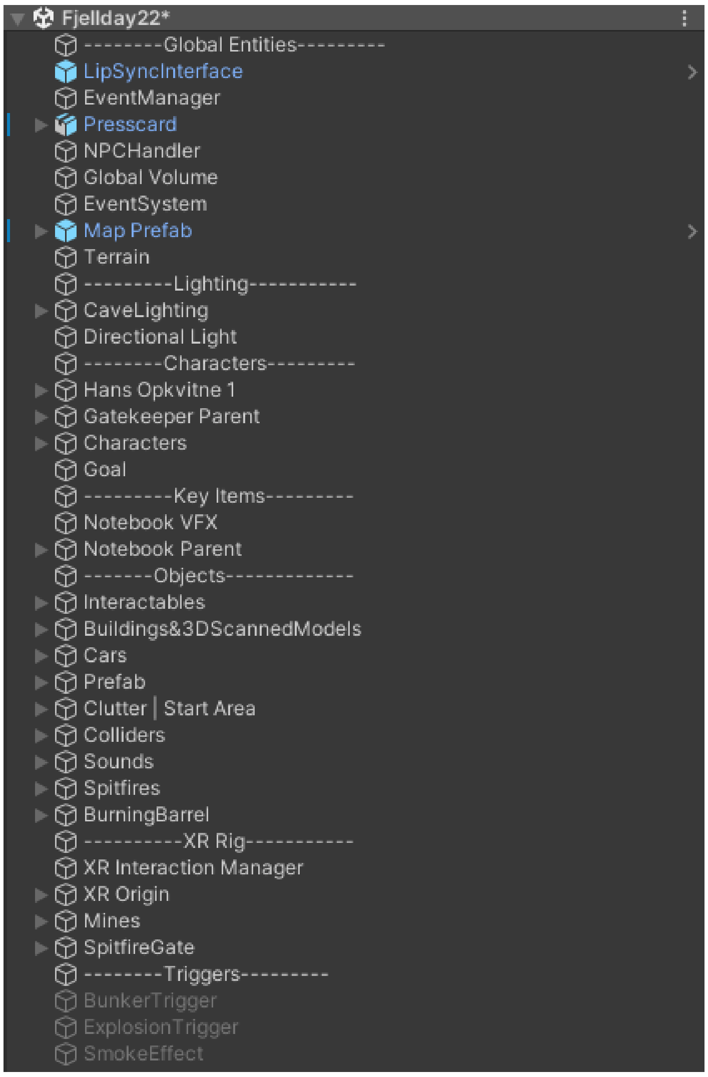
Figure 3: An example dialogue graph. The graphs used in the game are typically much bigger

3. Project Structure

Chapter 3 will explain and show the project's file architecture and how certain scripts and files are structured. In the Unity Editor, a project is separated into two parts. The scene hierarchy and the project tab folder which includes all the projects assets. This folder is the Asset folder and will be discussed later in *Chapter 3*.

3.1 Project Hierarchy

The scene hierarchy is structured in a way for easy access of all GameObjects in the scene. It's divided into seven main parts. Global Entities, Lighting, Characters, Key Items, Objects, XR Rig and Triggers. All GameObjects in Figure 4 act the same, where they are structured in a way where all children are under a specific GameObject, for example the XR Origin GameObject which has everything the player needs to be able to see, move and touch other objects.



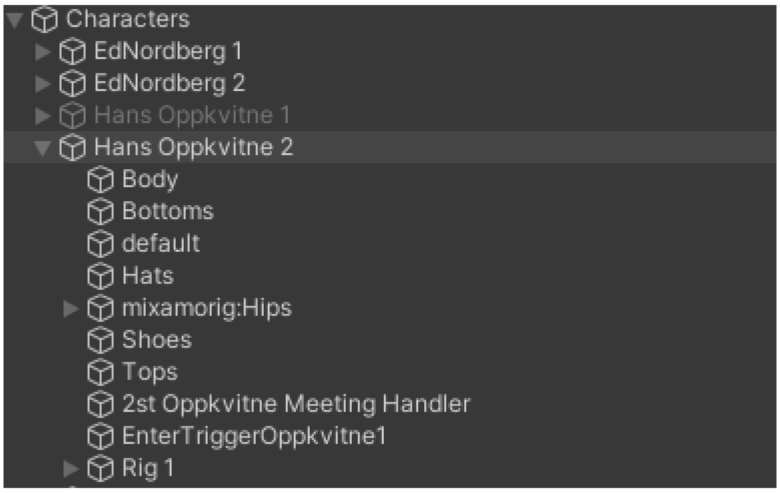
```
Fjellday22*
  --------Global Entities---------
  LipSyncInterface
  EventManager
  Presscard
  NPCHandler
  Global Volume
  EventSystem
  Map Prefab
  Terrain
  ---------Lighting----------
  CaveLighting
  Directional Light
  --------Characters---------
  Hans Opkvitne 1
  Gatekeeper Parent
  Characters
  Goal
  ---------Key Items---------
  Notebook VFX
  Notebook Parent
  -------Objects------------
  Interactables
  Buildings&3DScannedModels
  Cars
  Prefab
  Clutter | Start Area
  Colliders
  Sounds
  Spitfires
  BurningBarrel
  ----------XR Rig----------
  XR Interaction Manager
  XR Origin
  Mines
  SpitfireGate
  --------Triggers---------
  BunkerTrigger
  ExplosionTrigger
  SmokeEffect
```

Figure 4: Project Hierarchy in the main scene

Global Entities consist of the most important GameObjects in the scene, which include GameObjects like the Event System, NPCHandler, Map Prefab that listens to the Event Manager with help of the Map Handler script. Other GameObjects like the base terrain generated from the previous bachelor group are also placed here for easy access if needed.

All lighting is stored under Lighting. This part consists of a directional light and a GameObject “CaveLighting” with all the bunker spotlighting as children.

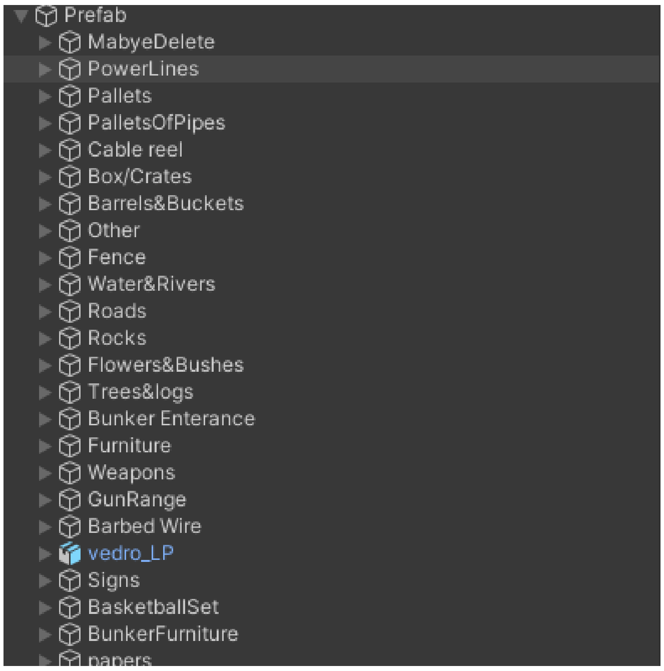
The Characters section has all of the scenes characters in one spot. These characters are placed as children under the Characters GameObject for better organization. Each character also has a Meeting Handler which has the Node Parser script attached to it for the dialogue system. The Node Parser decides which dialogue graph should be used on that character. Each character also consists of every body part that model has.



```
▼ Characters
  ▶ EdNordberg 1
  ▶ EdNordberg 2
  ▶ Hans Oppkvitne 1
  ▼ Hans Oppkvitne 2
      Body
      Bottoms
      default
      Hats
    ▶ mixamorig:Hips
      Shoes
      Tops
      2st Oppkvitne Meeting Handler
      EnterTriggerOppkvitne1
    ▶ Rig 1
```

Figure 5: Screenshot of a characters children GameObjects

Other sections like Key Items and Objects consist of all the models that are placed in the scene. These GameObjects are everything the player can see except the terrain itself. It is organized in such a way that it is easy to find where certain groups of objects are. For example the gun range area where the player can fire a weapon in VR, all pallets, cables, rocks, signs, etc. All of the GameObjects under the Prefab GameObject also have children.



```
▼ Prefab
  ▶ MabyeDelete
  ▶ PowerLines
  ▶ Pallets
  ▶ PalletsOfPipes
  ▶ Cable reel
  ▶ Box/Crates
  ▶ Barrels&Buckets
  ▶ Other
  ▶ Fence
  ▶ Water&Rivers
  ▶ Roads
  ▶ Rocks
  ▶ Flowers&Bushes
  ▶ Trees&logs
  ▶ Bunker Enterance
  ▶ Furniture
  ▶ Weapons
  ▶ GunRange
  ▶ Barbed Wire
  ▶ vedro_LP
  ▶ Signs
  ▶ BasketballSet
  ▶ BunkerFurniture
  ▶ papers
```

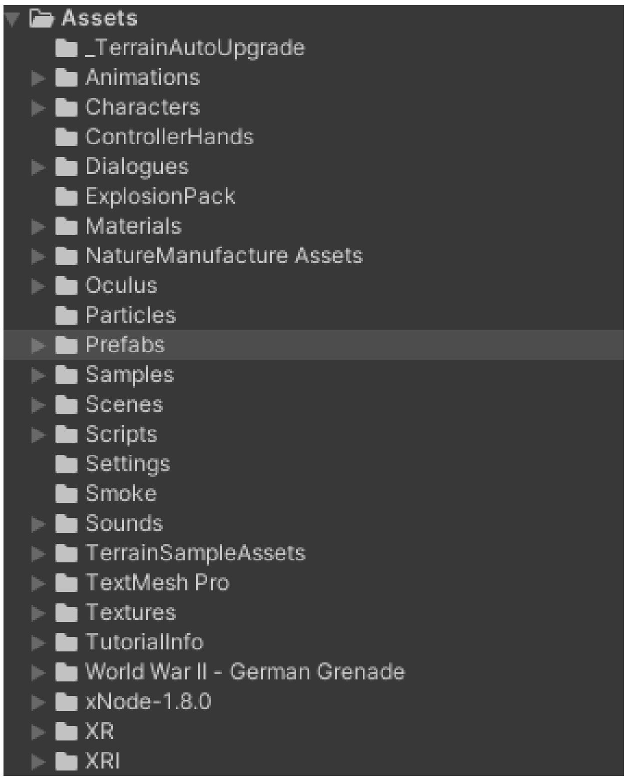
Figure 6: All children of the Prefab GameObject

The XR - Rig section behaves in the same way, where it consists of everything around the XR - Rig in the beginning of the game, as well as the XR - Rig GameObject itself. The XR - Rig is basically the player itself inside the game. Its children are the hands and the hands controllers, camera and locomotion settings.

The final section in the hierarchy is where all the Triggers are placed. These are invisible GameObjects in the scene that execute some predefined code that executes/triggers something when entering or exiting the trigger’s radius. All triggers have a script made by the team to perform a certain task.

3.2 Asset Folder

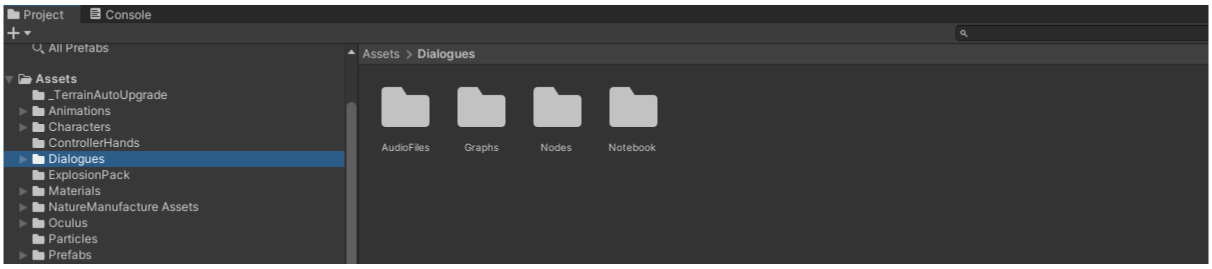
The asset folder is where all the project's assets are stored and can be seen in the Unity Editor. The folder is where the team has organized the different assets that are used in the project. This is the main folder used if a developer chooses to further develop the game.



```
Assets
  _TerrainAutoUpgrade
  Animations
  Characters
  ControllerHands
  Dialogues
  ExplosionPack
  Materials
  NatureManufacture Assets
  Oculus
  Particles
  Prefabs
  Samples
  Scenes
  Scripts
  Settings
  Smoke
  Sounds
  TerrainSampleAssets
  TextMesh Pro
  Textures
  TutorialInfo
  World War II - German Grenade
  xNode-1.8.0
  XR
  XRI
```

Figure 7: Asset folder in the Unity Editor

In Figure 7, all folders in the Asset folder are structured in a way such that it is easy to identify where certain assets are stored. For example, the Scene and Script folder is where we store absolutely every scene and script that is used in the game. Inside these folders are other folders that help the developer find the code or model they are looking for. An example of this is in Figure 8. Something one may have to take into consideration is if the developer chooses to delete a file from the Asset Folder in the Editor, it will also be deleted where the Asset folder is stored on the computer (Unity Documentation, 2022).



```
Project   Console
All Prefabs
Assets
  _TerrainAutoUpgrade
  Animations
  Characters
  ControllerHands
  Dialogues
  ExplosionPack
  Materials
  NatureManufacture Assets
  Oculus
  Particles
  Prefabs

Assets > Dialogues
AudioFiles   Graphs   Nodes   Notebook
```

Figure 8: Example of how each folder is structured

4. Class Diagram

The **Event Manager** controls the game's state. The event manager's state can be accessed and changed by other components. The event manager will notify other components that are subscribed to it that the state has changed, and what the new state is. Other components can declare a function to be run in the event of a state change. A component can also check the current state without waiting for an update.

The **NodeParser** is responsible for handling interactions with NPCs. Every dialogue interaction has a NodeParser instance. The **PlayerSensor** component activates the notebook, and starts a selected NodeParser's dialogue when the player is near an interactable NPC. A dialogue can only be started if the game is in the correct game state. There are in total 14 game states.

The **notebook** is a GameObject that contains the UI elements for the dialogue system, including text, buttons and visual effects. These are controlled through the NodeParser components.

The **MapHandler** component listens for changes in game state and updates the map GameObject accordingly. The map always points to the next interactable NPC.

The **NPCHandler** component is the script primarily responsible for hiding and showing NPCs depending on the game's state. Every character could have their individual event listeners for game state changes, and react accordingly. However having it all in one script makes it easier to manage, and lowers the amount of subscribers to the event system. Over the course of development, the NPC handler was extended to handle more objects than just NPCs. The NPCHandler script has references to transforms, audio, animations and scripts. You can drag the GameObjects into the NPC Handlers inspector which has the references that are needed to execute the code inside the script.

The class diagram displayed in figure 9, shows the connections between the most important components. GameObjects are in the diagram displayed as gray containers with components within them. All components are attached to GameObjects, but only components where GameObject methods are used are displayed in the diagram. A common use of these methods are to activate and deactivate the object.

What was difficult when creating the class diagram for the main systems used is that in Unity or game development in general, it is complicated creating a class diagram. There are a lot of scripts for a lot of different GameObjects in the scene which have a specific task to execute when needed. This makes it difficult to pinpoint which classes are relevant for the main system.

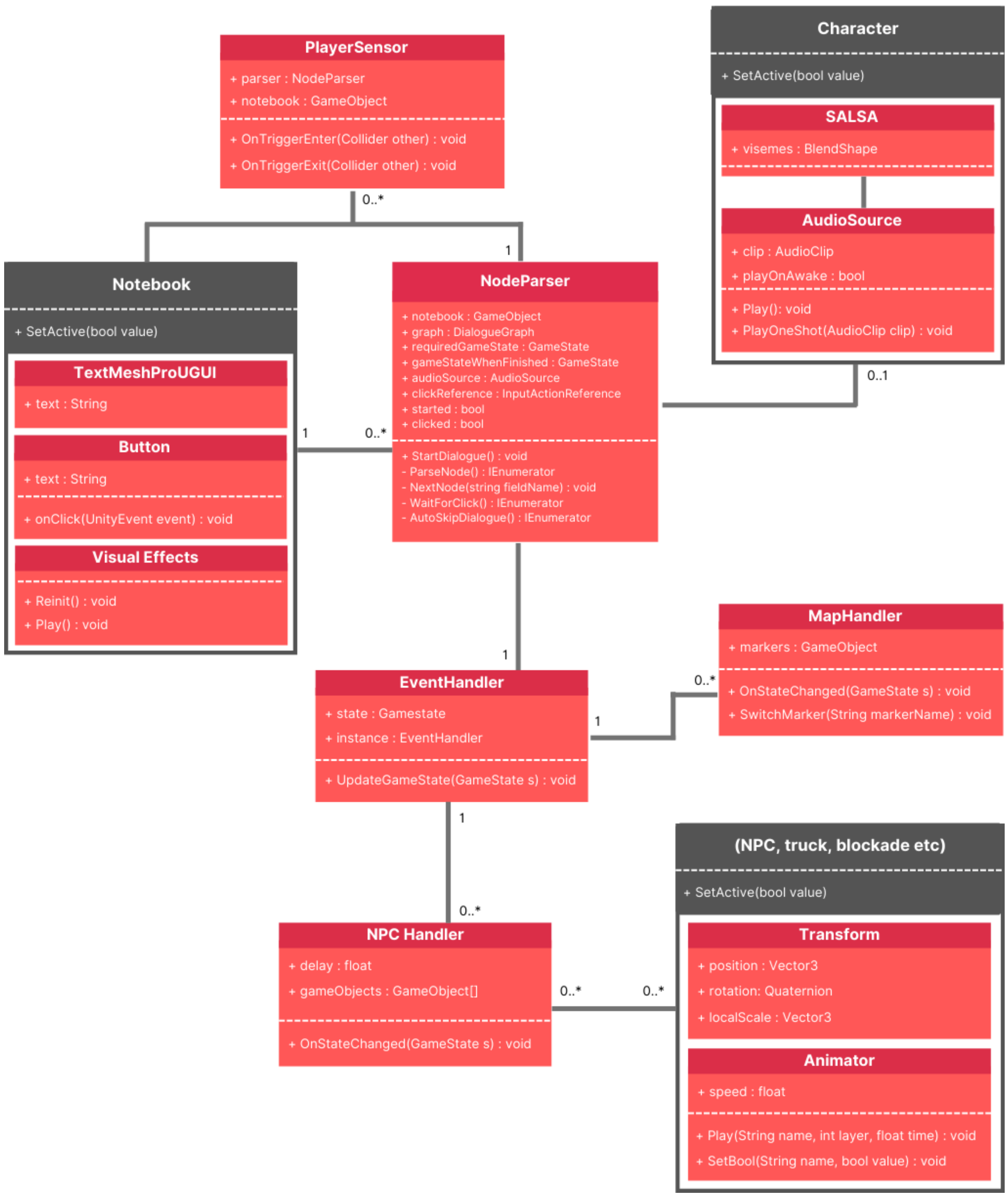

PlayerSensor
+ parser : NodeParser
+ notebook : GameObject
+ OnTriggerEnter(Collider other) : void
+ OnTriggerExit(Collider other) : void
0..*
1
Character
+ SetActive(bool value)
SALSA
+ visemes : BlendShape
AudioSource
+ clip : AudioClip
+ playOnAwake : bool
+ Play(): void
+ PlayOneShot(AudioClip clip) : void
0..1
Notebook
+ SetActive(bool value)
TextMeshProUGUI
+ text : String
Button
+ text : String
+ onClick(UnityEvent event) : void
Visual Effects
+ Reinit() : void
+ Play() : void
1
0..*
NodeParser
+ notebook : GameObject
+ graph : DialogueGraph
+ requiredGameState : GameState
+ gameStateWhenFinished : GameState
+ audioSource : AudioSource
+ clickReference : InputActionReference
+ started : bool
+ clicked : bool
+ StartDialogue() : void
- ParseNode() : IEnumerator
- NextNode(string fieldName) : void
- WaitForClick() : IEnumerator
- AutoSkipDialogue() : IEnumerator
1
MapHandler
+ markers : GameObject
+ OnStateChanged(GameState s) : void
+ SwitchMarker(String markerName) : void
0..*
1
EventHandler
+ state : Gamestate
+ instance : EventHandler
+ UpdateGameState(GameState s) : void
1
0..*
(NPC, truck, blockade etc)
+ SetActive(bool value)
NPC Handler
+ delay : float
+ gameObjects : GameObject[]
+ OnStateChanged(GameState s) : void
0..*
0..*
Transform
+ position : Vector3
+ rotation: Quaternion
+ localScale : Vector3
Animator
+ speed : float
+ Play(String name, int layer, float time) : void
+ SetBool(String name, bool value) : void

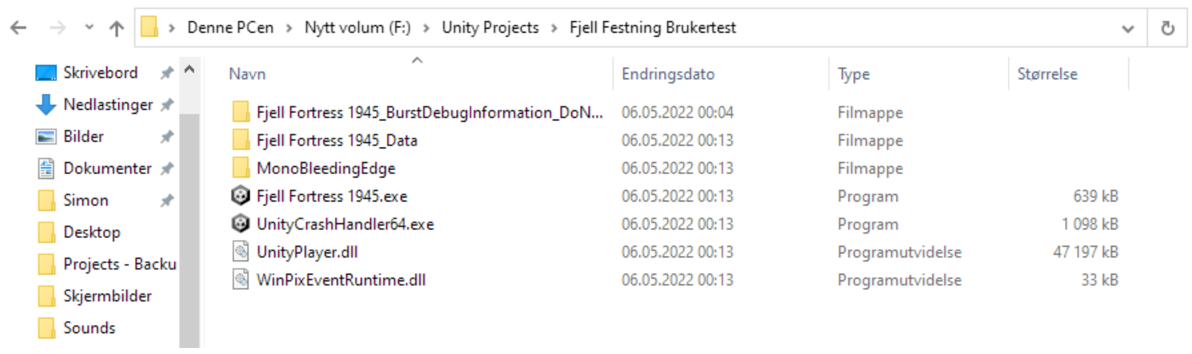
Figure 9: Modified class diagram of the most central features.

5. Installation and Execution

In order to run the game, the Oculus App must be installed. The app will ask which VR headset the user is using and the app will start to configure. If the Open XR Runtime is set to SteamVR, the user must run the SteamVR program from Steam. The Oculus App is necessary for the VR headset to connect to the PC the game is running from.

When connecting the VR headset with the computer, the user has two options. The first option is through the Oculus Air Link. Here the user must connect the headset via the internet and run the headset in Developer Mode. The other option is with an Oculus Link Cable, which connects the Oculus VR headset with the computer. For better performance and less input delay, a Link Cable is recommended for a better user experience.

As of today, the application is not downloadable from any game platform. The user must unzip a given zip file which includes the build of the game. From the team's understanding, the final build of the game will be sent to any school requesting to use the game. When unzipped, the user can then run the .exe file included in the project folder. If the user was able to configure the VR Headset correctly and the game is running, the user should be able to see the game through the headset and play as intended.


Denne PCen > Nytt volum (F:) > Unity Projects > Fjell Festning Brukertest

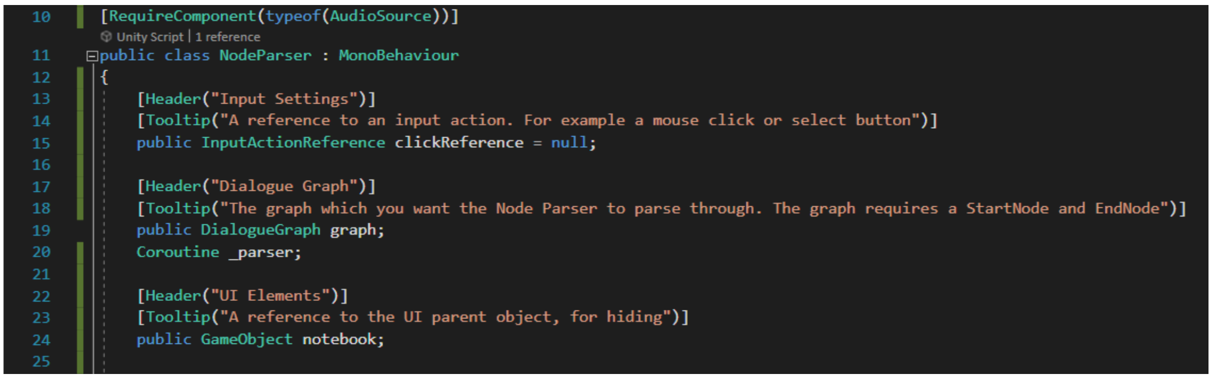
Navn	Endringsdato	Type	Størrelse
Fjell Fortress 1945_BurstDebugInformation_DoN...	06.05.2022 00:04	Filmappe	
Fjell Fortress 1945_Data	06.05.2022 00:13	Filmappe	
MonoBleedingEdge	06.05.2022 00:13	Filmappe	
Fjell Fortress 1945.exe	06.05.2022 00:13	Program	639 kB
UnityCrashHandler64.exe	06.05.2022 00:13	Program	1 098 kB
UnityPlayer.dll	06.05.2022 00:13	Programutvidelse	47 197 kB
WinPixEventRuntime.dll	06.05.2022 00:13	Programutvidelse	33 kB

Figure 10: Showing where to find the .exe file

6. Documentation

6.1 General Information

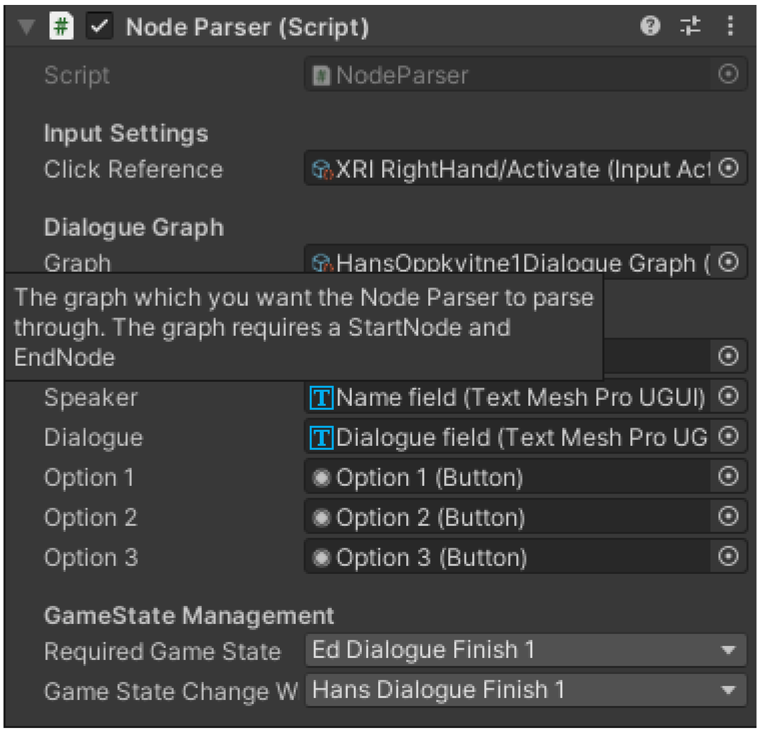
The code is documented within the scripts as well as through helpful headers and tooltips you can access when hovering over the different settings and references in the inspector. This is done through Unity's inspector customization tags. Figure 11 shows what some of these tags look like inside of a script. The RequireComponent tag above the class name is there to ensure the GameObject has the required Components. Unity prevents anyone from removing the required component without removing the script that depends on it. When adding a script with requirements Unity will automatically add the appropriate components.



```
[RequireComponent(typeof(AudioSource))]
Unity Script | 1 reference
public class NodeParser : MonoBehaviour
{
    [Header("Input Settings")]
    [Tooltip("A reference to an input action. For example a mouse click or select button")]
    public InputActionReference clickReference = null;

    [Header("Dialogue Graph")]
    [Tooltip("The graph which you want the Node Parser to parse through. The graph requires a StartNode and EndNode")]
    public DialogueGraph graph;
    Coroutine _parser;

    [Header("UI Elements")]
    [Tooltip("A reference to the UI parent object, for hiding")]
    public GameObject notebook;

```

Figure 11: Header and tooltip tags describing the different references.

In the inspector the references are divided under different headers. A tooltip describing the reference or setting further is displayed when hovering above it. Figure 12 displays what the NodeParser looks within the inspector when hovering above one of the required references.



Node Parser (Script)

Script	NodeParser
Input Settings	
Click Reference	XRI RightHand/Activate (Input Act
Dialogue Graph	
Graph	HansOppkvitne1Dialogue Graph (
Speaker	Name field (Text Mesh Pro UGUI)
Dialogue	Dialogue field (Text Mesh Pro UG
Option 1	Option 1 (Button)
Option 2	Option 2 (Button)
Option 3	Option 3 (Button)
GameState Management	
Required Game State	Ed Dialogue Finish 1
Game State Change W	Hans Dialogue Finish 1

The graph which you want the Node Parser to parse through. The graph requires a StartNode and EndNode

Figure 12: The NodeParser component as viewed in the inspector window, hovering above the graph reference

6.2 Using the dialogue system

Creating a new dialogue graph can be done by right clicking in any folder. In the pop up menu click Create > Dialogue Graph. This will open the dialogue graph in a new window.

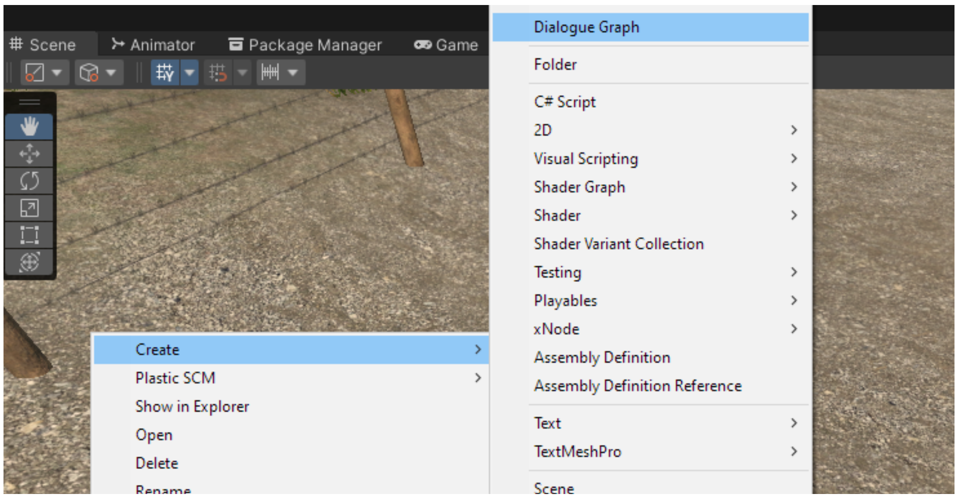
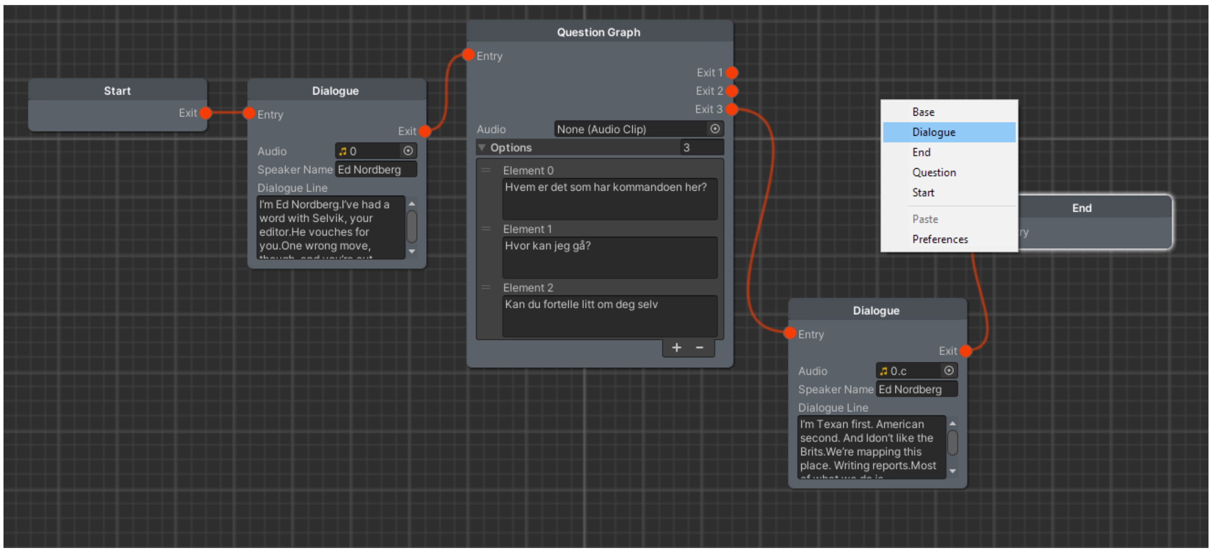


Figure 13: Pop up menu for creating a new dialogue graph.

Once the dialogue graph is open, you can right click to add nodes. The Start Node is needed in order for the Node Parser to understand where to start. In order for the appropriate state to be updated at the end of the dialogue, the graph must also end in an end node. Multiple nodes can be connected to the same end node, or to separate ones. The nodes can be connected by dragging the exit port to an entry port. In a question node, make sure to use as many exit ports as there are questions, starting with “exit1” for question 1.



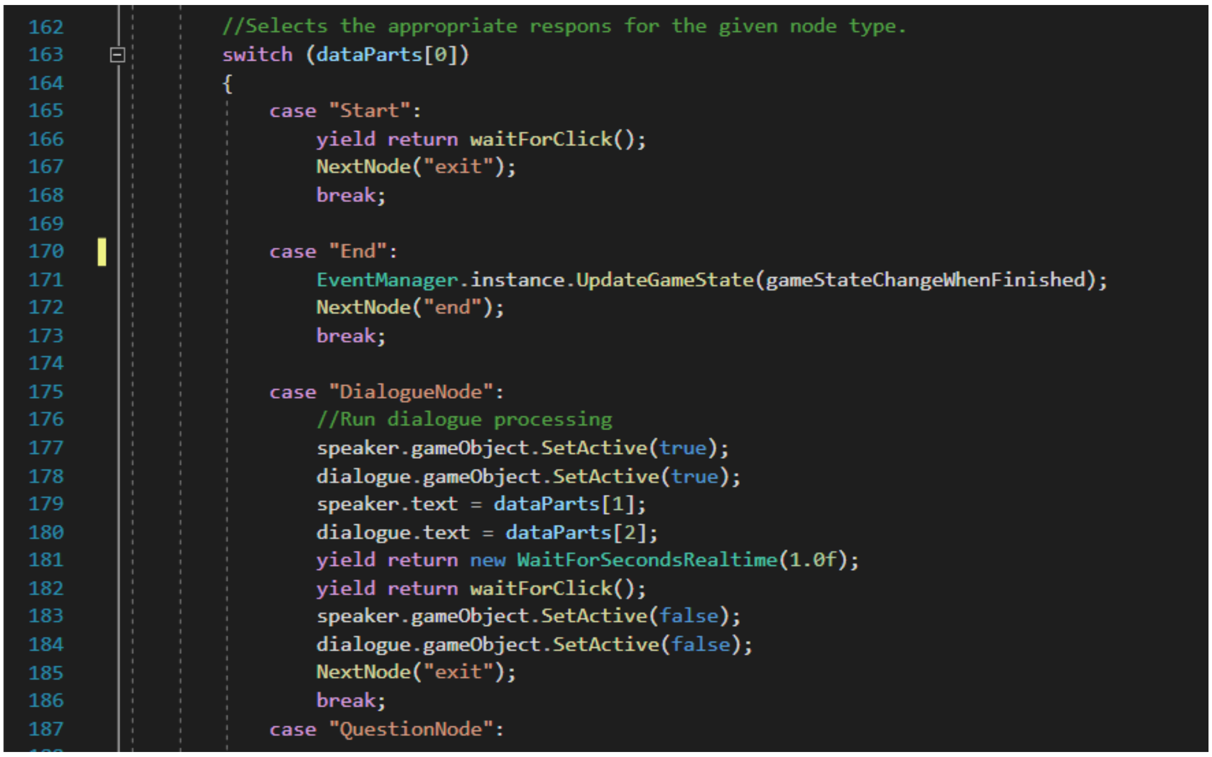
Audio ♫ 0
Speaker Name Ed Nordberg
Dialogue Line
I'm Ed Nordberg.I've had a word with Selvik, your editor.He vouches for you.One wrong move,

Audio None (Audio Clip)
Options 3
Element 0
Hvem er det som har kommandoen her?
Element 1
Hvor kan jeg gå?
Element 2
Kan du fortelle litt om deg selv

Audio ♫ 0.c
Speaker Name Ed Nordberg
Dialogue Line
I'm Texan first. American second. And Idon't like the Brits.We're mapping this place. Writing reports.Most

Figure 14: An example dialogue graph. Right click to add nodes from the list of nodes.

Custom nodes can be created by right clicking any folder, and from the pop up menu navigate to Create > xNode > Add Node. For information on how to program custom nodes, we recommend checking out the xNode github page (Siccity, 2021). Alternatively use one of the nodes created for this project as a baseline. The NodeParser will have to be modified in order for the parser to use the new nodes data. This can be done by adding a case within the NodeParser’s switch statement for the new node's name. We recommend using our previous nodes as a baseline should this be appropriate.



```
        //Selects the appropriate respons for the given node type.
        switch (dataParts[0])
        {
            case "Start":
                yield return waitForClick();
                NextNode("exit");
                break;

            case "End":
                EventManager.instance.UpdateGameState(gameStateChangeWhenFinished);
                NextNode("end");
                break;

            case "DialogueNode":
                //Run dialogue processing
                speaker.gameObject.SetActive(true);
                dialogue.gameObject.SetActive(true);
                speaker.text = dataParts[1];
                dialogue.text = dataParts[2];
                yield return new WaitForSecondsRealtime(1.0f);
                yield return waitForClick();
                speaker.gameObject.SetActive(false);
                dialogue.gameObject.SetActive(false);
                NextNode("exit");
                break;
            case "QuestionNode":
```

Figure 15: Switch statement containing the logic for the different node types.

The character's interaction is in our case composed of three different objects. One being the character itself. Another being an object that contains the NodeParser and the audio source, and a third object acting as a trigger, activating the Notebook and the NodeParser as the player steps within its radius.

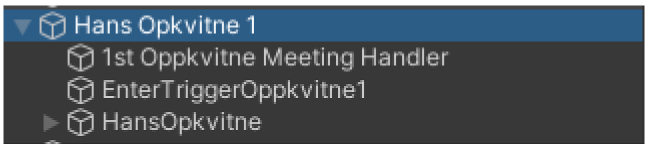
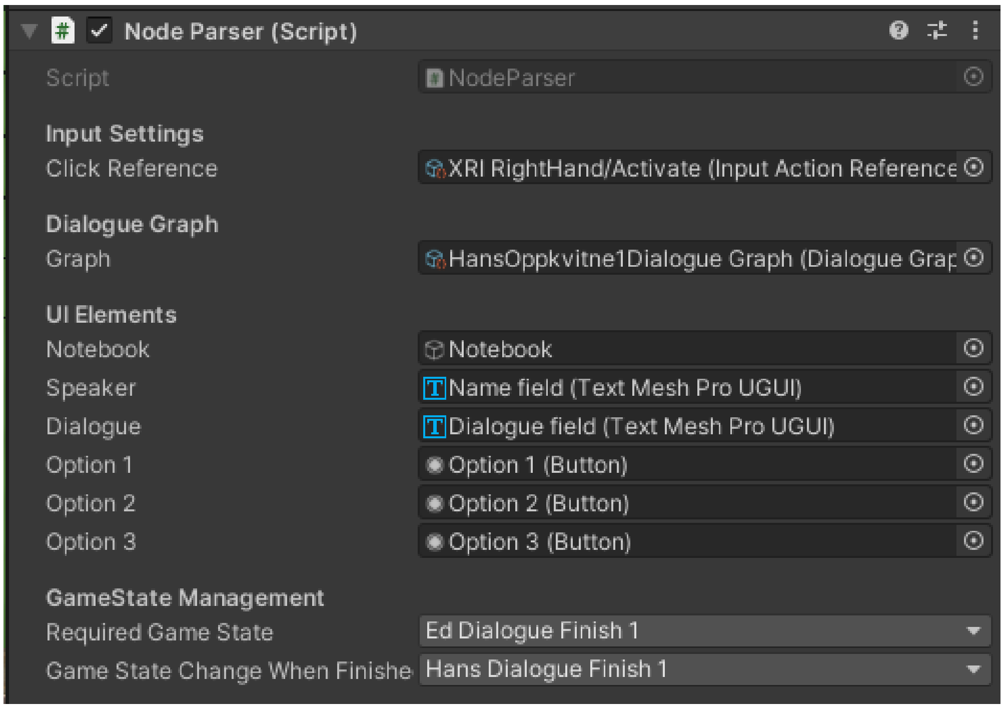


Figure 16: The three objects are children of a parent object

When the NodeParser is added to a GameObject, all references need to be set. These include the input action that the player has to perform to skip the dialogue, the dialogue graph for that particular interaction, the UI elements and what state the dialogue requires and results in. These can be set through the inspector. The NodeParser component can be added to any object, however since scripts do not run on inactive objects we recommend not adding it to any object that is meant to be disabled during the course of gameplay.



Node Parser (Script)

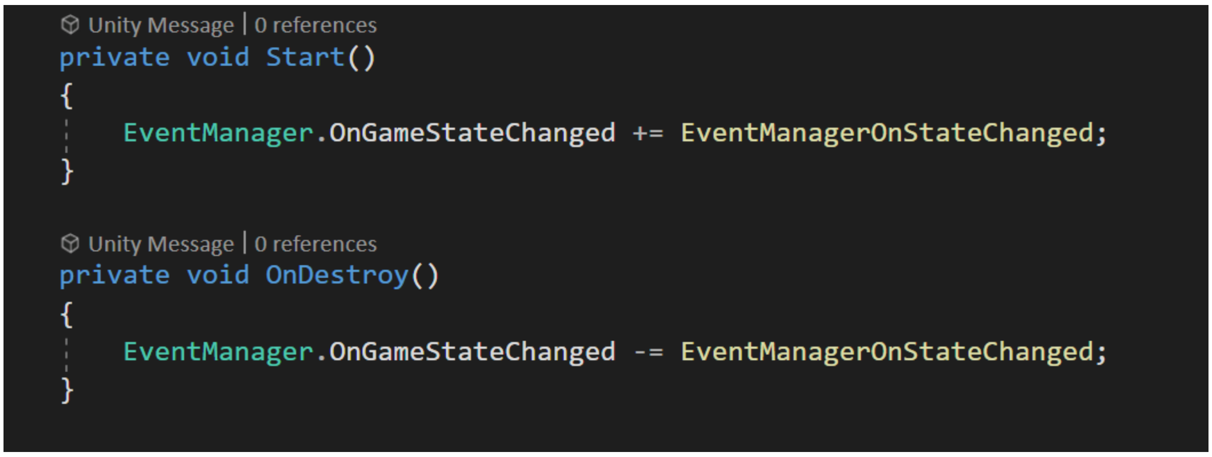
Script	NodeParser
Input Settings	
Click Reference	XRI RightHand/Activate (Input Action Reference
Dialogue Graph	
Graph	HansOppkvitne1Dialogue Graph (Dialogue Grap
UI Elements	
Notebook	Notebook
Speaker	Name field (Text Mesh Pro UGUI)
Dialogue	Dialogue field (Text Mesh Pro UGUI)
Option 1	Option 1 (Button)
Option 2	Option 2 (Button)
Option 3	Option 3 (Button)
GameState Management	
Required Game State	Ed Dialogue Finish 1
Game State Change When Finishe	Hans Dialogue Finish 1

Figure 17: The Node Parser component when added to a GameObject.

6.3 Using the Event Manager

Using the Event Manager is a simple process. The event manager is using the singleton pattern. The methods are static and can be accessed from any script.

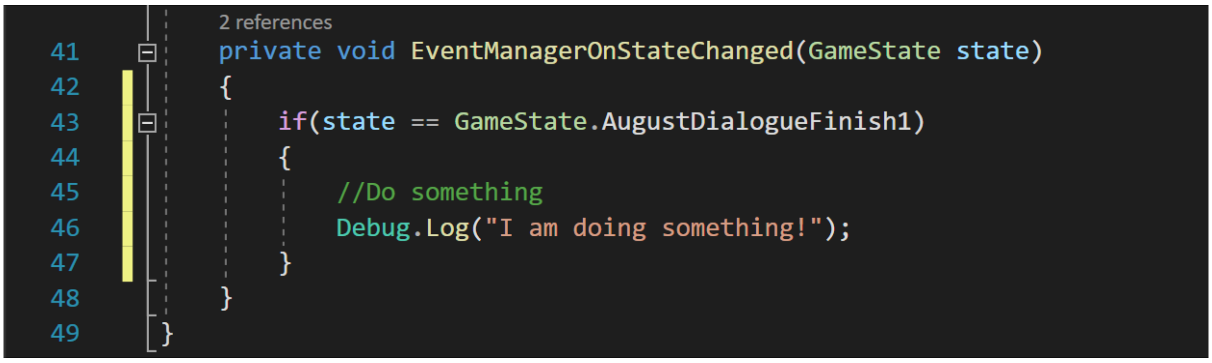
Subscribing to the event manager does not require an instance of it. You can subscribe to it from another script by using EventManager.OnGameStateChanged += yourMethod;. To increase performance, remember to unsubscribe when it is no longer needed. This is done using EventManager.OnGameStateChanged -= yourMethod;.



```
Unity Message | 0 references
private void Start()
{
    EventManager.OnGameStateChanged += EventManagerOnStateChanged;
}

Unity Message | 0 references
private void OnDestroy()
{
    EventManager.OnGameStateChanged -= EventManagerOnStateChanged;
}
```

Figure 18: Subscribing and unsubscribing to the state change event.

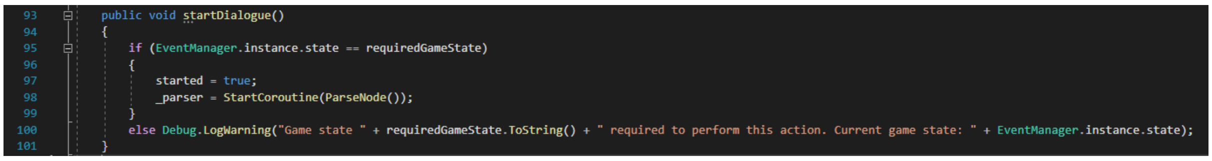
When subscribing to the OnGameStateChanged event, you need to specify a function to be executed when the event is triggered. This function takes the new GameState as a parameter. Using an if or switch statement allows you to perform logic on specific GameStates.



```
2 references
private void EventManagerOnStateChanged(GameState state)
{
    if(state == GameState.AugustDialogueFinish1)
    {
        //Do something
        Debug.Log("I am doing something!");
    }
}
}
```

Figure 19: Example implementation of the state change on the subscribers side.

In some instances it might be more appropriate to check the state in between state changes. In the Node Parser, to make sure the game state matches the requirement, the current state is fetched. To check the current game state, use `EventManager.instance.state`.



```
public void startDialogue()
{
    if (EventManager.instance.state == requiredGameState)
    {
        started = true;
        _parser = StartCoroutine(ParseNode());
    }
    else Debug.LogWarning("Game state " + requiredGameState.ToString() + " required to perform this action. Current game state: " + EventManager.instance.state);
}
```

Figure 20: Using the EventManager’s state to validate dialogue requirement

6.4 Script Overview

This subchapter will contain an overview of the custom scripts and a brief explanation about what they do.

- **NodeParser.cs**
  See chapter 6.2
- **PlayerSensor.cs**
  Trigger that starts NPC interaction. Interacts with NodeParser
- **PlayerTrigger.cs**
  General purpose trigger. Functions to execute can be added through the inspector
- **EventManager.cs**
  See chapter 6.3
- **NPCHandlingScript.cs**
  The NPCHandlingScript.cs is the script primarily responsible for hiding and showing NPCs depending on the game's state. It is subscribed to the Event Manager and listens if a state has changed. A switch statement determines which code block should be executed depending on what state the game is in.
- **MapHandler.cs**
  Handles the maps. Switches marker the next dialogue interaction upon game state change.
- **CharacterHelper.cs**
  Controls several aspects of the player character model. Shrinks the player collider according to the camera's position, allowing players to crouch under obstacles. Also handles footstep sounds.
- **FootSteps.cs**

Picks an appropriate sound clip according to the terrain underneath the player and plays it.

- **TerrainDetector.cs**
  Detects the terrain texture beneath the player and returns the index of that texture's position in the terrain textures. Used by FootSteps to determine the correct sound.
- **HandScript.cs**
  Hides the hands when holding objects.
- **HandController.cs**
  Detects when holding objects and using HandScript hides the hands.
- **OffsetMover.cs**
  Moves an object with another only with a specified offset. The offset and the target object can be set through the inspector.
- **GatekeeperBehaviour.cs**
  Controls the interaction with the Gatekeeper NPC. This interaction is fundamentally different from the others, and for that reason is independent of the NodeParser.
- **ClipPlayer.cs**
  Used to hold and play multiple AudioClips. Requires an AudioSource on the same GameObject. A list of AudioClips can be set through the inspector and played through scripts using the Play(int index) method.
- **Follow.cs**
  Used to make an object follow another smoothly. The object will also always point upwards. Used to make visual effects move with notebook without rotating with it, and prevents janky movement.
- **FollowTargets.cs**
  Similar to Follow.cs, but cycles through a list of targets that can be set through the inspector. Used in the project to make characters switch between looking at different GameObjects.
- **Highlighter.cs**
  Used to highlight objects. Starts and stops an animation specified through the inspector window. An item can automatically stop highlighting after a given time. This time can also be set through the inspector.
- **HitDetection.cs**
  Executes a function apon colliding with an object with the specified tag. Both the tag and the function can be selected through the inspector. Example use case is glass bottles shattering upon hitting the ground.
- **SimpleShoot.cs**
  Script used to determine a pistol's audio, animation and bullet count. Each part of the script can be changed through the inspector.
- **Grenade.cs**
  Grenade script can be attached to any object in the scene and will act as a grenade. The script has an audio source and clip, as well as an explosion effect. The timer can also be set from when the grenade hits the ground until it explodes, making the grande seem more realistic.
- **JuglerGoal.cs**
  Used to start the truck animation when Jugler walks into it. It also scales the character down such that the player won't be able to see the GameObject. It also destroys the GameObject the script is attached to when Jugler has entered the trigger. This is for not triggering the script again when Jugler is walking into it.
- **UniversalFadeScript.cs**

Used to fade in and fade the canvas that is in front of the player's camera. Here, one can change the color the canvas should fade in or fade out to, as well as the time it takes to fade.

- **CarCameraFollow.cs**
  Adds the transform of the XR Camera to the taxi's transform. This script is useful for positioning the camera on the taxi such that when it moves, the camera will move with it.
- **PlaneTruckTrigger.cs**
  A script attached to a non - visible GameObject that plays the truck animation, as well as all three spitfire animations when exited. It is placed in the cave and becomes active after the player has conversed with Lewis Adams.
- **FirstSceneChange.cs**
  Script that starts a timer when the game is loaded, it will then start the fadeIn method from the UniversalFadeScript which makes the canvas go from transparent to black. It also changes the scene when the fadeIn method has finished.
- **OutroSceneChange.cs**
  Works similar to the FirstSceneChange.cs, however it has some code which is different which makes it so the code can't be generalized. The script does the opposite of what the FirstSceneChange.cs script does. OutroSceneChange.cs also exits the program after a fixed time (when the video has finished).
- **GermanSpawn.cs**
  The script is subscribed to the Event Handler and executes some code when the desired event has been changed. The code being executed is setting the German soldiers as active which makes them visible in the scene after a state has been changed.
- **SpitfireSpawn.cs**
  Spawns a spitfire and starts an animation when the correct event has been changed. It is subscribing to the Event Manager.
- **SimpleGravity.cs**
  This script was in the end not used. Designed to move objects to the ground without rotating. Used as an attempt to avoid a bug with objects falling through ground.
- **MineTrigger.cs**
  Used as a trigger for determining when a player is on a mine. When the player exits the mine, an audio will be played as well as a visual effect. There is also a transform for knowing where the player will respawn when the explosion happens. It also has the reference to the player so the script can change the position accordingly.
- **BasketballScript.cs**
  The script is a trigger that gives extra time if the player is able to throw the basketball in the basketball hoop.

7. References

References

Siccity. (2021, October 4). *Home · Siccity/xNode Wiki · GitHub*. GitHub. Retrieved May 19, 2022, from https://github.com/Siccity/xNode/wiki

Unity Documentation. (2022, May 15). *Manual: Importing assets*. Unity - Manual. Retrieved May 20, 2022, from https://docs.unity3d.com/Manual/ImportingAssets.html