

**University of the West of England**

**Commercial Games Development**

**Academic Social Skills**

# **Technical Design Document**



# Charlie Thomas

## Project Structure

### Dialogue Tree asset Creation and Saving System UML:

[https://drive.google.com/file/d/1pOVkeWEIFlpyd\\_WM3z6AtLOAkDaRotdt/view?usp=sharing](https://drive.google.com/file/d/1pOVkeWEIFlpyd_WM3z6AtLOAkDaRotdt/view?usp=sharing)

1. **Asset Creation:** The project system follows a process of dialogue asset creation, saving and loading. The dialogue tree is constructed using a front end *GraphView* interface, which constructs save data through serialisation into Unity *ScriptableObject* types for different node types, node groups and trees. These node based dialogue trees are built in collaboration with the client, who inform the content, tips and options available throughout the dialogue system.
2. **Scene creation:** Each new scene can be created with a canvas prefab that contains *CTNodeIOUtility* and *CTUISetupUtility* objects, as well as container objects for background, character anchor, next node button, tip button anchor, responses anchor and a dialogue text window. This scene is then referenced in the main menu scene.
3. **Scene setup:** Dialogue tree data is assigned for a given scene, which is set up for the needs of a specific tree through the *CTUISetupUtility*, including character names, character images and background images. This data is read through the *CTNodeIOUtility* and fed into the *CTUISetupUtility* to display this information to screen and handle player interactions with the dialogue system.
4. **CTNodeIOUtility setup:** On this prefab, the *CTNodeIOUtility* takes a reference to the *CTUISetupUtility*, and a "Start" node object, which should be the first node in a dialogue tree.

Also on this prefab, the *CTUISetupUtility* has serialised member variables that may be assigned in inspector for scene character names, character sprites, background sprites, and the prefab objects used to create the scene visually with information extracted from the current node via *CTNodeIOUtility*.

## **Dialogue Tree System.**

In researching potential solutions to the problems with the 1.0 dialogue tree system, a candidate was found in the Unity *GraphView API*. This seemed directly appropriate, as the *GraphView API* is an inherently node based visually constructed system and would make for a maintainable front end to the *CTNode* data system. In this system, nodes are defined with parameters and instantiated into the graph and may be connected via defined rules.

There are several components that need to be in place to have dialogue trees that can be saved, loaded, or edited using the *GraphView API*:

- *GraphView* window.
- Base graphview elements.
- *SaveData* containers.
- *PersistentData* types.
- Utilities.

### **GraphView window.**

**CTGraphViewWindow:** is a derived class of type *UnityEditor.EditorWindow* and is responsible for the base editor window for the dialogue tree editor. It contains a *UnityEditor.MenuItem* attribute which adds “Dialogue Tree” as an option in the inspector window context menu. It also contains the functionality to initiate saving, loading and clearing of the graph view through the use of *UnityEngine.UIElements.Button* objects, and these buttons are enabled or disabled based on the current error status of the graph to disallow invalid save states. It makes use of the *CTComponentUtility* class to create these interactable elements.

**CTGraphView:** is a derived class of type *UnityEditor.Experimental.GraphView.GraphView* and is responsible for setting up the graphview window. It sets up the manipulators that allow the user to interact with the window, such as ‘*ContentDragger*’, ‘*SelectionDragger*’ and ‘*RectangleSelector*’. Furthermore, it defines what elements appear in contextual menus to be added to the graph view and has methods to do so by instantiating *CTGroups* and *CTNodes*. It tracks what elements are added to the graphview, as well as the current number of ‘errors’ in the graph view. It has data containers in the form of a 3<sup>rd</sup> party class “*SerializableDictionary*” (which is used as Unity does not natively support Dictionary class serialisation) to store error information for *CTGroups*, grouped *CTNodes* and ungrouped *CTNodes*.

**CTError:** is a base error class, which has a *UnityEngine.Color* property. It contains one function, which is to set a random colour to this property with RGB values within the range of 0-255, and a fixed alpha of 255.

**CTGroupError:** is the error type for *CTGroup* title conflicts. It has a property *CTError*, and a *List<CTGroup>* list of groups that contribute to the error.

**CTNodeErrorData:** is the error type for *CTNode* title conflicts. It has a property *CTError* and a *List<CTNode>* list of nodes that contribute to the error.

Error handling is important to the saving of a dialogue tree, as groups and nodes are saved with their titles as their scriptable object names. Without this error tracking, *CTGroups* and *CTNodes* (which can either be global or grouped, with grouped nodes causing errors within the same group and global nodes causing errors with other global *CTNodes*) that share the same title will all be saved, but each node of the same title saved to the same parent folder will overwrite the save data of the previous node with the same title. Only allowing saving when there are no errors ensures that all dialogue nodes added to the tree are saved without data loss from overwriting.

## Base GraphView Elements

**CTNodeOptionData:** is a data model class to hold option data for *CTNode* ports.

**CTNode:** is the dialogue tree base node type. It has properties for all information on what a node is composed of. It has an initialise function which defines the node name, the *CTGraphView* object it is associated with, and its position in 2D space within the graph view window. It has a method that defines its draw behaviour, checks for errors and what components it has, such as Ports, *TextFields*, *DropdownFields* and Foldouts. Furthermore, it contains definitions for behaviour for context menus, and events when components' values are changed as well as a handful of utility functions. It can have an associated *CTGroup* reference, if it is grouped.

**CTNarrationNode:** is a derived class of type *CTNode* with an override extension of the *Initialise()* and *Draw()* functionality from *CTNode*. The *Initialise()* override defines the *CTNode* type as *CTNodeType.Narration*, gives the output Port the label "Next Node" and adds it to the list of options. The *Draw()* override sets up the Port for the node.

**CTChoiceNode:** is a derived class of type *CTNode* with the same *Initialise()* override, except it sets the *CTNode* type as *CTNodeType.Choice*. The *Draw()* override is similar, except it also creates a button which, when clicked, adds a new option to the list through a *callback* function. This new option is given a default text of "New Option", before creating an output port for the new option and adding it to the output container. *CTChoiceNode* contains an additional function '*CreateOutputPort*' to facilitate this.

**CTGroup:** is a derived class of type *UnityEditor.Experimental.Graphview.Group*. It has a unique ID set in the constructor using *System.Guid*, a globally unique identifier. It also has methods for setting and clearing error colour if there are *CTGroup* conflicts in the dialogue tree.

## SaveData Containers

**CTNodeSaveData:** is a data model class to hold *CTNode* information.

**CTNodeOptionSaveData:** is a data model class to hold *CTNode* port data.

**CTGroupSaveData:** is a data model class to hold *CTGroup* data.

## PersistentData Types

CTNodeDataSO, CTNodeGroupingSO and CTTreeDataSO are derived classes of type *UnityEngine.ScriptableObject*. They all contain initialise functions which copy data from the corresponding class to save as a *ScriptableObject* as persistent data.

**CTTreeSaveDataSO:** is a derived class of type *UnityEngine.ScriptableObject* with data containers for *CTNode* types, sorted by grouped and ungrouped nodes. Groups are stored in a *SerializableDictionary*. The key to this dictionary is the *CTNodeGroupingSO*, and the value is a list of nodes associated with the group using *List<CTNodeDataSO>*. It has an initialise function which takes a string for the name of the saved dialogue tree.

## Utilities

**CTComponentUtility:** is a small static utility class used to create elements for *CTNodes*. It has methods to create *UnityEngine.UIElements: TextField, Foldout, DropdownField, Button* and *UnityEditor.Experimental.GraphView Port*. Each element has a static type function taking parameters associated with the information for the requested component.

**CTIOUtility:** is a controller script used to organise all functionality around saving and loading of dialogue tree data. It is the most important script to the functionality of the dialogue tree system.

**CTIOUtility Saving:** There are several methods to serialise nodes, groups, and node connections as both individual scriptable objects and to the node data container object of the *CTTreeSaveDataSO* scriptable object.

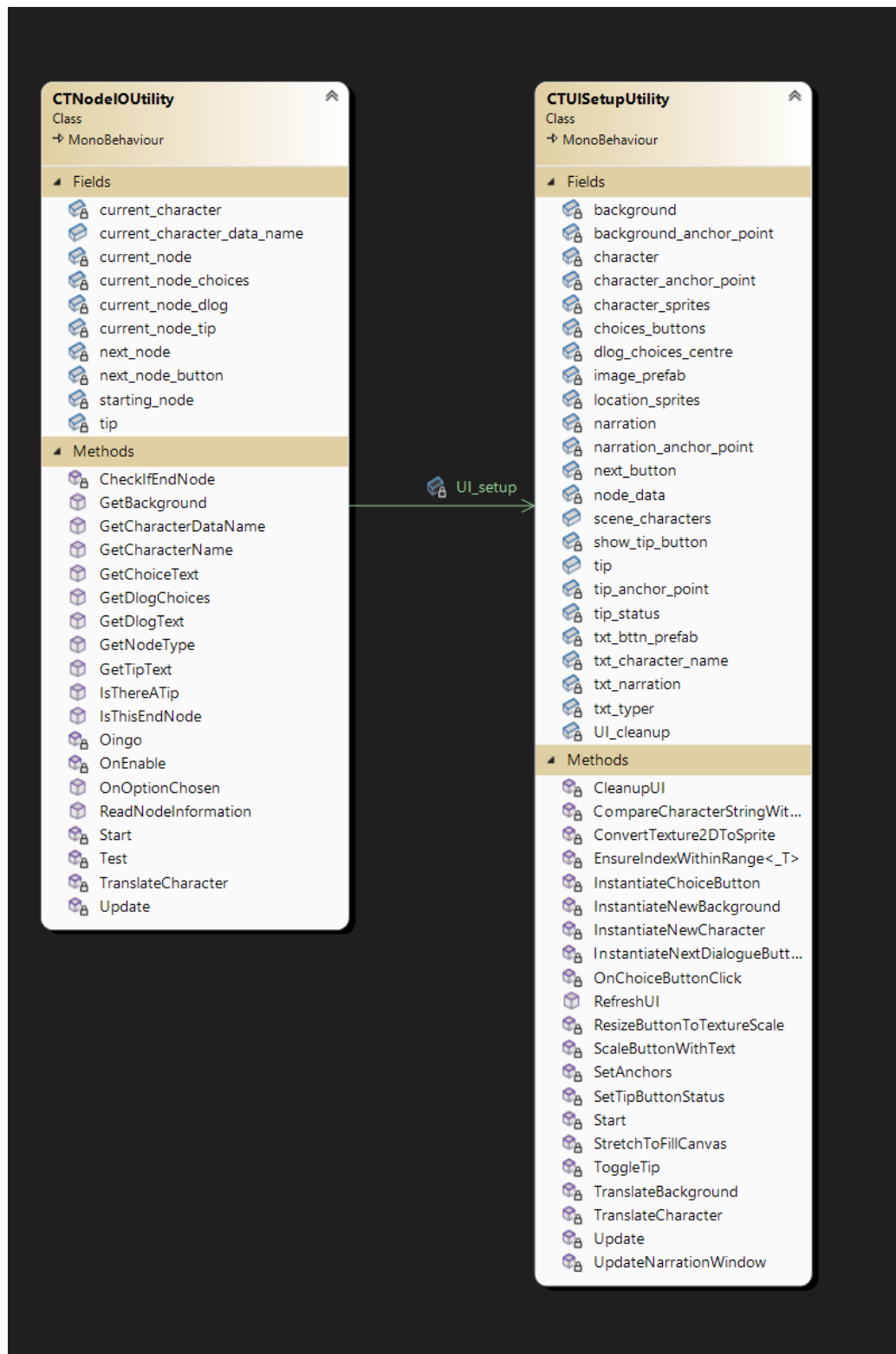
**CTIOUtility Loading:** There are several methods to deserialize the saved scriptable objects for nodes, groups, and node connections into the graph view.

**CTIOUtility Utility:** There are several utility functions in this class, their functionality includes:

- Templated *ScriptableObject* asset creation for saving.
- Asset removal for deleted elements.
- Node choices copy for copying *CTNode* data for saving.
- Folder creation with specified path and name.
- Functionality to check for removed groups.
- Folder deletion for cleaning up dead folders for removed groups.
- Default folder creation to ensure valid folder creation paths.
- Functionality to get a reference for every element in a tree saved to list data containers.
- Functionality to convert option data to the serializable format.
- Functionality to retrieve node connection information from saved data, deserialize it into the loaded graph view and re-establish connections. There are methods to perform the same task for grouped and ungrouped nodes also.

## Play UI Engine.

The system to control play scene UI is dependent on two *Monobehaviour* classes. *CTNodeIOUtility* is responsible for handling data retrieval and managing active nodes. *CTUISetupUtility* handles dynamic instantiation, creation and updating of UI elements in the scene with active node information as retrieved through *CTNodeIOUtility*.



## CTUISetupUtility

### Member Variables:

Has several [SerializeField] member variables for assignment through the inspector. These include: references to images used for character sprites, character names, background sprites, references to UI prefab types, references to Unity Canvas anchor point elements.

### Unity Methods:

**Start( ):** is used to initialise container objects and add listeners to button objects.

**Update( ):** simply has a listener for the escape key to exit the application.

### Button Methods:

**ToggleTip( ):** is a private void function that checks for the presence of a tip for a given node and sets up associated tip objects appropriately, hiding or showing the tip reveal button as necessary.

**InstantiateChoiceButton( ):** is a private void function responsible for creating buttons for each choice for the active dialogue node. It destroys the old choice button game objects from the previous node and removes them from the list of *choice\_buttons*. If the node type is CTNodeType.Narration, the function returns as no choices are possible for narration nodes. Otherwise, the list of choices is populated with data from the active node.

There is a horrible hack associated with this method. Each of the choice buttons, on instantiation, is assigned a name that corresponds with the index of the for loop creating them. This gameobject name is used to create the listener associated with the choice button. The OnChoiceButtonClick method accepts an int as a parameter, and into this is passed the button name parsed as an int32. This ensures that the instantiated choice buttons can be used to correctly index the list of choices using their game object name, as it will always correspond with their instantiation in the for loop.

**UpdateNarrationWindow( ):** is a private void function that takes a string as a parameter. This string is passed to a *StartTypingText* method that prints the string as if it is being typed. The character name text for the window is updated with the node data stored character name.

**OnChoiceButtonClick( ):** is a private void function that takes an integer as parameter. As explained above, this makes use of the horrible hack around using the assigned gameobject names as the index for this method. The *OnOptionChosen* method is called, passing this index.



## Sprite Methods:

***InstantiateNewCharacter()*** is a private void function taking an integer as a parameter. This index is used to select a sprite from a list assigned in the inspector. The *EnsureIndexWithinRange* method is called to check if the requested index exists. The function then destroys the existing character image object and creates a new one using a prefab, setting the sprite to the selected character sprite. Lastly, this object is assigned anchors.

***InstantiateNewBackground()*** is a private void function responsible for creating a new background image. It takes an index as a parameter, which it uses to select the appropriate sprite from an array of background sprites assigned in the inspector. *EnsureIndexWithinRange* is called, to check that the requested sprite index exists. The function then destroys the existing background image object, creates a new one using a prefab and sets its sprite to the selected background sprite from the list. Finally, the anchors are set for this object, and it is set as a child of the background anchor point.

***StretchToFillCanvas()*** is a private void function that stretches an image to fill a canvas. It takes an image and canvas reference as parameters. It does this by setting the anchor variables of the parameter image's *RectTransform*.

## Utility Methods:

***RefreshUI()*** is a public void method used to set up the game UI with new values. It calls: *CleanupUI*, *InstantiateNewBackground*, *InstantiateNewCharacter*, *InstantiateChoiceButton*, *UpdateNarrationWindow* and sets the tip button active status to true/false based on the presence of a tip for the current node.

***CleanupUI()*** is a private void method used to destroy any unwanted UI elements when switching between nodes. *GameObjects* are added to the list *UI\_cleanup* if they are only necessary in the scope of the current node, a new list is created and each of the objects in *UI\_cleanup* are added. Each object in the new cleanup list is then iterated through, whereby they are removed from the *UI\_cleanup* list and destroyed.

***ConvertTexture2DToSprite()*** is a sprite return method that takes a *Texture2D* parameter and converts it to a sprite using *Sprite.Create*.

***SetAnchors()*** is a void method that takes a *GameObject* parameter and sets the *RectTransform* component min, max and pivot anchors to the centre of the parent.

***EnsureIndexWithinRange<\_T>()*** is a bool return method taking an index and templated *List<\_T>* type. It returns true or false depending on whether the parameter index value is within the bounds of the parameter List.

***ResizeButtonToTextureScale()*** is a void method that resizes a button object to match the size of its texture.

***TranslateBackground()***: is an int return method that uses a switch to determine what index to return, using the *node\_data* background.

***TranslateCharacter()***: is an int return method that uses a switch to determine what index to return, using the *node\_data.current\_character\_data\_name*.

## CTNodeIOUtility

### Member Variables:

***private CTUISetupUtility UI\_setup***: a serialised reference to a *UI\_setup* object.

***private CTNodeDataSO starting\_node***: a serialised reference to the starting *CTNodeDataSO* object for the desired tree, as saved through the graphview editor.

***private CTNodeDataSO current\_node***: a serialised reference to the current *CTNodeDataSO* object to read information from, as saved through the graphview editor.

***private CTNodeDataSO next\_node***: a serialised reference to the next *CTNodeDataSO* object to read data from, as saved through the graphview editor.

***private List<CTNodeOptionData> current\_node\_choices***: a serialised reference to the list of *CTNodeDataSO* objects stored in the “options” *List<CTNodeDataSO>* of a given *CTNodeDataSO* object. This list represents the forward connected nodes in the dialogue tree for any given node.

***private string current\_node\_dialogue***: a serialised reference to the stored “text” string variable for a given *CTNodeDataSO* object. This contains the text to be displayed in the dialogue box, the “active” dialogue.

***private string current\_node\_tip***: a serialised reference to the stored “tip\_text” string variable for a given *CTNodeDataSO* object. This contains the text to be displayed in the tip box, if the player opens the tip.

***private string current\_character***: a serialised reference to the stored “character” string for a given *CTNodeDataSO* object. This contains the text to be displayed in the character name box.

***public string current\_character\_data\_name***: a string used to store the name of the character in terms of what is stored by *CTNodeDataSO*. This means: “Narrator, Character\_0,...,Character9”. This is “translated” based on the inputs to the character names to set *current\_character* to a real name.

***private bool tip***: a bool that denotes whether there is a tip for the current node.

## Unity Methods:

**OnEnable( ):** the *current\_node* variable is set to the *starting\_node* information, and a call is made to the *ReadNodeInformation* method to populate member variables with data from the starting node.

**Start( ):** the next node button is allocated a listener.

### Node Access Methods

**ReadNodeInformation( ):** is a void function that reads current node information and performs data sanitation to remove carriage returns. Node *choices* and *character* are set and translated. If there is a tip, the tip string is stored and the bool denoting a present tip is set to true.

**OnOptionChosen( ):** is a void function that handles the functionality for the player selecting a dialogue option. Firstly, we check to determine if the current node is an end node and return. If the node has forward options, the next node is selected based on the index of the chosen option. *ReadNodeInformation* is called to pull information from the object, and *UI\_setup.RefreshUI* is called to update the UI with the new node's information.

**CheckIfEndNode( ):** is a bool return function that checks if the current node has forward nodes, returning false if there are subsequent nodes and true if not.

**GetTipText( ):** is a string return function that checks for a tip, returning the tip string if it exists and otherwise returning an empty string.

**GetDlogChoices( ):** is a *List<CTNodeOptionData>* return function that returns the list of options for the current node.

**GetChoiceText( ):** is a string return function that takes an int index and returns the text for a choice at that index from *current\_node\_choices[index].text*.

**GetNodeType( ):** is an *Enum CTNodeType* return function that returns the type of node. This type can either be "Narration" or "Choice".

**TranslateCharacter( ):** is a void function that takes the *current\_character* string and uses the serialised character names in the *CTUISetupUtility* to replace the fixed data names with the unique character names allocated in the inspector.

## Polish features

**TextTyper:** The *TextTyper* class is used to create the effect of text printing to screen character-by-character. It takes a complete string input and iterates through the string with a given time step delay between updates until the string is complete.

*TextTyper* takes a *[SerializeField]* reference to the TMP\_Text object used to show the dialogue for the current node. The rate at which the typing effect can be adjusted with the *type\_speed* variable.

**UITextFade:** The *UITextFade* class is a simple image alpha adjustment script similar to *TextTyper*. On start, the alpha is set to zero, and a coroutine is called that increments the alpha over a given time frame.

**UIImageFade:** The *UIImageFade* class is a simple image alpha adjustment script similar to *TextTyper*. On start, the alpha is set to zero, and a coroutine is called that increments the alpha over a given time frame.

Both *UITextFade* and *UIImageFade* make use of the choice button *gameObject.name* hack to produce a delayed fade in for each additional button after button zero.

These classes are added to the respective components that make up a choice button, such that on instantiation they have these properties.

```
1 reference
public void StartTypingText(string _input)
{
    //Debug.Log(_input);
    text_to_type = _input;
    index = 0;
    tmp_object.text = "";
    is_typing = true;
    StartCoroutine(Type());
}

1 reference
IEnumerator Type()
{
    while (is_typing)
    {
        if (index < text_to_type.Length)
        {
            tmp_object.text += text_to_type[index];
            index++;
            yield return new WaitForSeconds(type_speed);
        }
        else
        {
            is_typing = false;
        }
    }
}
```

```
@ Unity Script (1 asset references) | 0 references
public class UITextFade : MonoBehaviour
{
    public float fade_time = 0.5f;
    private TextMeshProUGUI tmp_text;

    @ Unity Message | 0 references
    void Start()
    {
        tmp_text = GetComponent<TextMeshProUGUI>();
        Color c = tmp_text.color;
        c.a = 0f;
        tmp_text.color = c;
        StartCoroutine(Fade());
    }

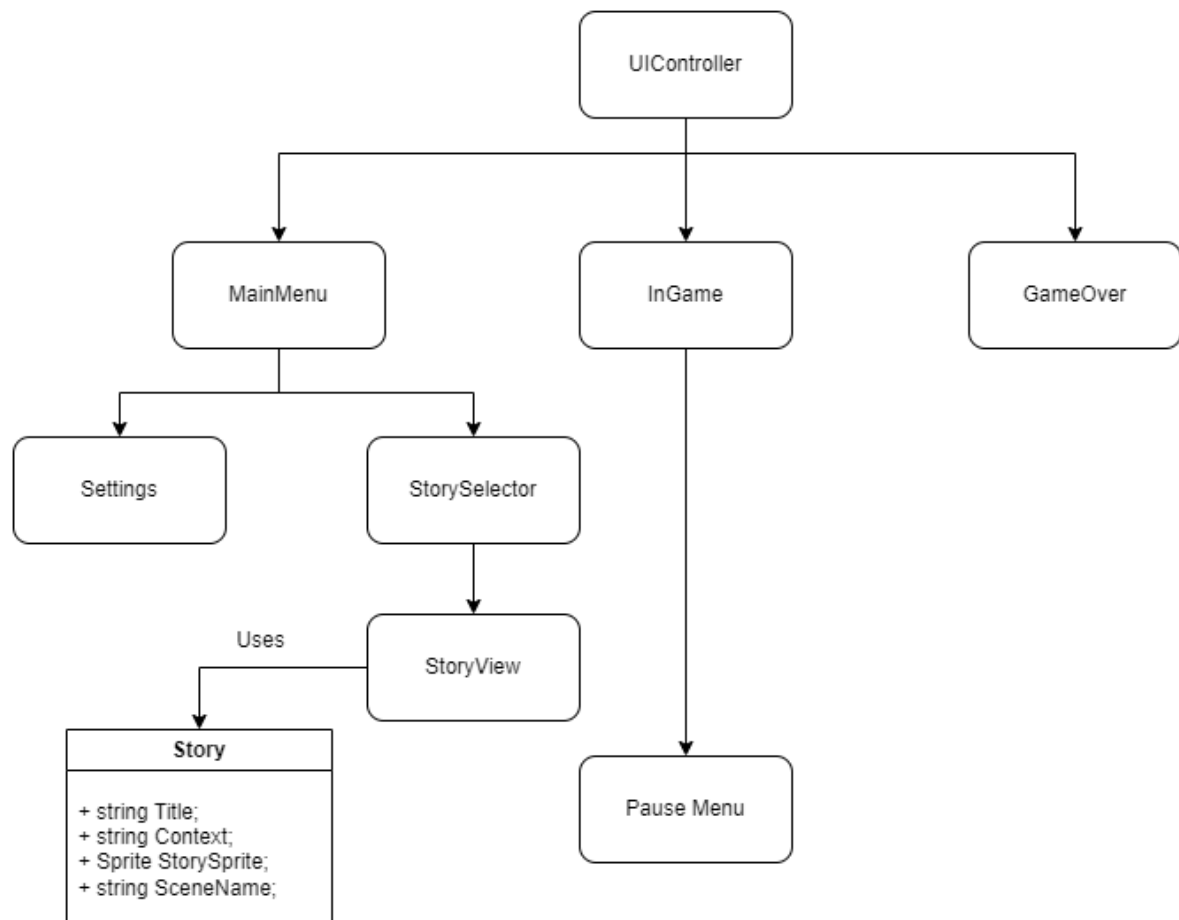
    1 reference
    IEnumerator Fade()
    {
        yield return new WaitForSeconds(int32.Parse(this.transform.parent.gameObject.name) * 0.25f);
        yield return new WaitForSeconds(0.5f);
        float interval = fade_time / 100f;
        for (float alpha = 0f; alpha <= 1f; alpha += 0.01f)
        {
            Color c = tmp_text.color;
            c.a = alpha;
            tmp_text.color = c;
            yield return new WaitForSeconds(interval);
        }
    }
}
```

```
@ Unity Script (2 asset references) | 0 references
public class UIImageFade : MonoBehaviour
{
    public float fade_time = 0.25f;
    private Image image;

    @ Unity Message | 0 references
    void Start()
    {
        image = GetComponent<Image>();
        Color c = image.color;
        c.a = 0f;
        image.color = c;
        StartCoroutine(Fade());
    }

    1 reference
    IEnumerator Fade()
    {
        yield return new WaitForSeconds(int32.Parse(this.transform.parent.gameObject.name) * 0.25f);
        yield return new WaitForSeconds(0.5f);
        float interval = fade_time / 100f;
        for (float alpha = 0f; alpha <= 1f; alpha += 0.01f)
        {
            Color c = image.color;
            c.a = alpha;
            image.color = c;
            yield return new WaitForSeconds(interval);
        }
    }
}
```

## UI Integration



The UI integration architecture follows Event-Driven architecture, where all button click events are passed to the controller classes using Actions. In essence, actions are pre-existing delegates that facilitate the creation of delegate events with a void return type. Therefore, actions can function equivalently to delegates without necessitating prior declaration.

**UIController:** It is the reference holder class for all UI classes ie MainMenu, InGame and GameOver. It gets notified by MainMenu screen whenever there is story is selected to be loaded.

**MainMenuScreen:** It holds the references for screens like Settings screen and the story selector screen and buttons for navigation to screens mentioned.

**StorySelectorScreen:** This class shows the list of stories listed in GameDataManager. It gets the list of the stories to be shown and then instantiates the story view prefabs in the screen. This class passes an action to the mainmenu whenever it gets notified of any story is selected.

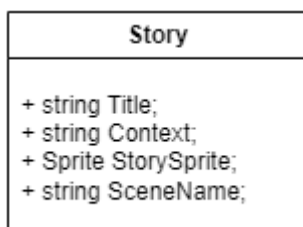
**SettingsScreen:** This screen provides control for music, sound effects and master volume levels and ambient sound selector from the dropdown menu.

**StoryView:** This UI view is attached to the prefab of story, which holds references to the story title and description. On selecting a particular story it executes an action with the story references to be notified by its listeners.

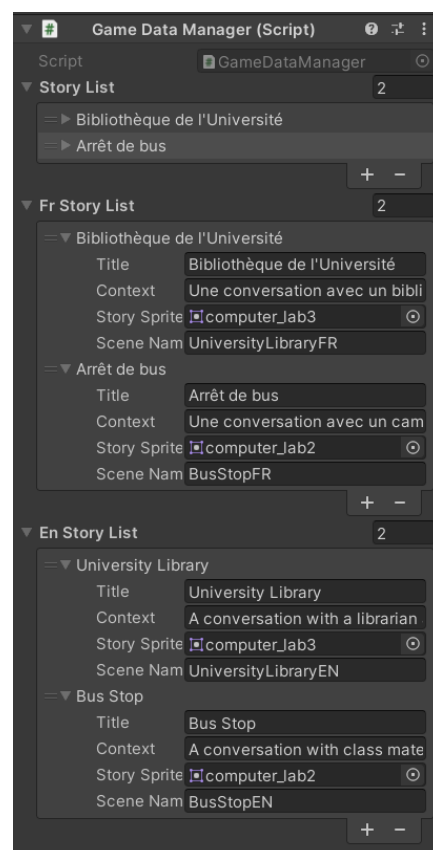
### AudioService:

This AudioService is used for playing audio clips on events. It has features like play button sounds, play background (ambient) sound, change background sounds and save and load last sound volume levels set on the computer by a player.

### GameDataManager:



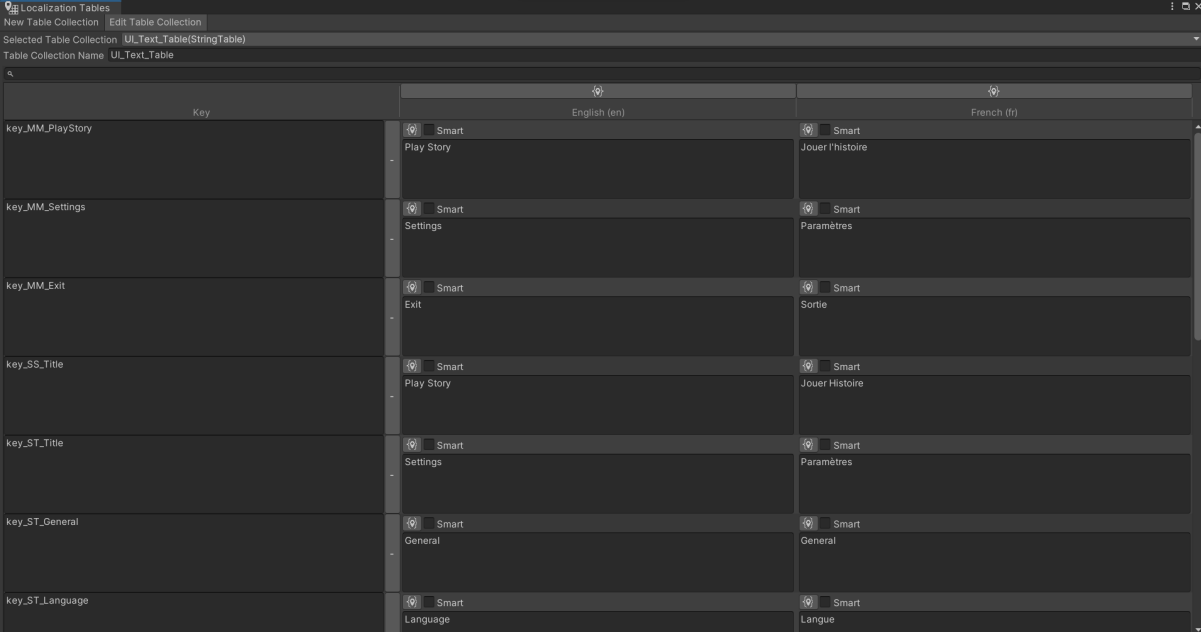
This class provides flexibility to the developer for adding new stories to the game without actual coding efforts. A model class Story is used to store story data related to story. These data includes Title, context, story sprite to be shown in the story selector screen and a scene name to be loaded on selecting a story. By Creating an object in the list it will create an instance in the story selector screen at run time and it will load the story scene, with valid scene name in the story list.



## Localization Manager:

To fulfill requirement of support for multiple language, Unity's localization library is used. Localization manager is created, which can switch between languages at runtime.

Localization table :



Key	English (en)	French (fr)
key_MM_PlayStory	Play Story	Jouer l'histoire
key_MM_Settings	Settings	Paramètres
key_MM_Exit	Exit	Sortie
key_SS_Title	Play Story	Jouer Histoire
key_ST_Title	Settings	Paramètres
key_ST_General	General	General
key_ST_Language	Language	Langue

Assigning Keys to the Text element :

This can be done by adding Localization String Event to the TextMeshProUGUI component and add update string event. This is a global listener to the localization locale update.

