TreeStory Documentation

Release 0.1.0

DL Techy

Mar 18, 2017

Table of Contents

Scripting is a huge part of game development in Unity. But it can also be very time consuming even for the most basic of games. To enable game developers to focus on the more important parts of actually making games, packages can be obtained from Unity's asset store which should handle all of the tedious parts of game development. One such package is TreeStory.

TreeStory is a Unity add-on to make story-based game creation as simple as possible. It uses a node editor to create dialog and events, and prefab templates to create the actual game and UI. Branching of dialog and events are also made much simpler using this tool.

Using TreeStory removes most, if not all, of the hassle of scripting for your game, enabling you to focus on more important things, like creating the game's actual story, creating game assets, and more. And even if what you need is not yet implemented, it is very easy to do so yourself, and it's still not as much of a hassle as when you start scripting from scratch.

CHAPTER 1

Getting Started

The main component of TreeStory is a node library. Node libraries contain the flow of the game you are going to make.

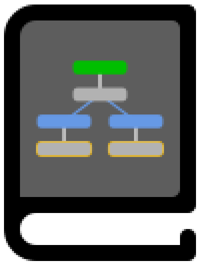


Fig. 1.1: The Node Library

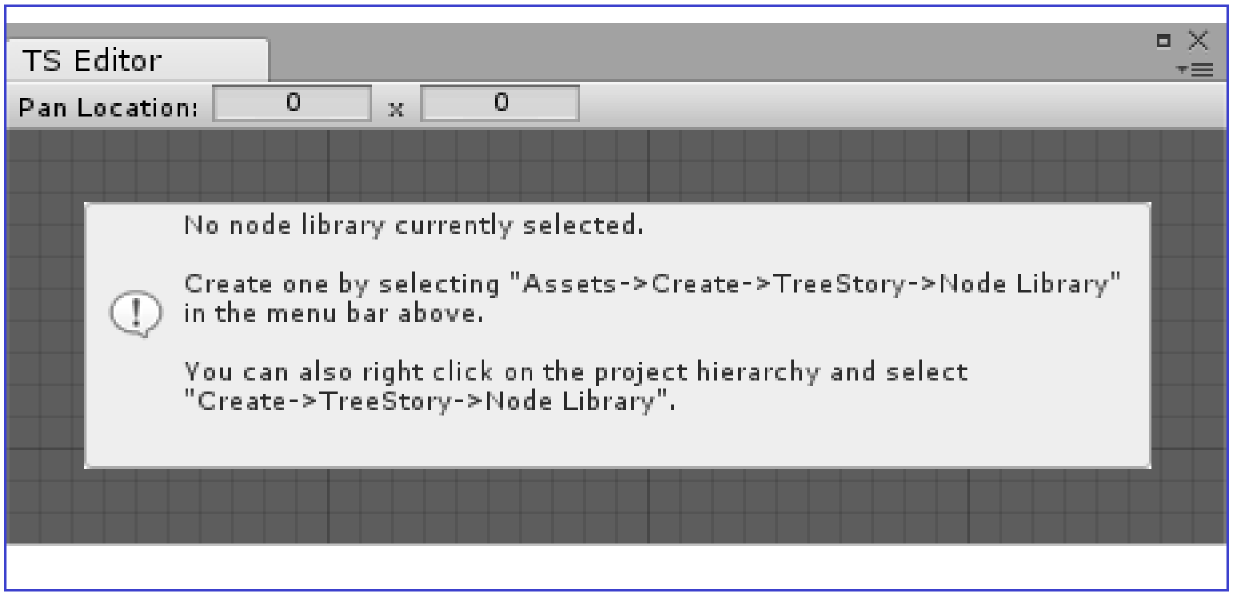
Creating a node library

1. Either right-click on the project window or click on “Assets” in Unity’s menu bar.
2. Go to Create -> TreeStory -> Node Library.
3. This will create a new node library asset.

Opening the TreeStory Editor

Either:

- Double click the created node library asset.
- Click on Window -> TreeStory Editor in Unity’s menu bar.



No node library currently selected.

Create one by selecting "Assets->Create->TreeStory->Node Library" in the menu bar above.

You can also right click on the project hierarchy and select "Create->TreeStory->Node Library".

Fig. 1.2: The TreeStory Editor

CHAPTER 2

Creating Game Data

To use your own assets in TreeStory, you first need to convert them into useable game data. This can be done by using TreeStory's custom libraries which stores game data like actors, locations, audio and video. Each library is stored in its own asset file. Creating any TreeStory library is similar to how you create a node library shown previously.

Creating a library

1. Either right-click on the project window or click on "Assets" in Unity's menu bar.
2. Go to Create -> TreeStory.
3. Select which library you want to create.

- Due to the nature of how TreeStory handles the saving and loading of files, you must place ALL libraries inside any folder names "Resources" to be able to utilize TreeStory's save system.
- You can create multiple libraries of each type. TreeStory is able to find any game data even if they are separated in multiple library assets. This is useful for organization of your project especially if your project grows too large. This is also useful if you work in a group, since each group member can work on separate libraries.

Editing library entries

- Select the library asset that you want to edit from Unity's hierarchy window.
  - This will show you a data list inside Unity's inspector window (The image below shows the actor data list as an example).
- Press the (+) button on the lower right of the data list to create a new entry to the library.
- Press the (-) button on the lower right of the data list to remove the selected entry from the library.
  - Alternatively, you can press the delete key on your keyboard.
  - You can also right click on the library entry and press the "Delete" button from the menu that pops up.

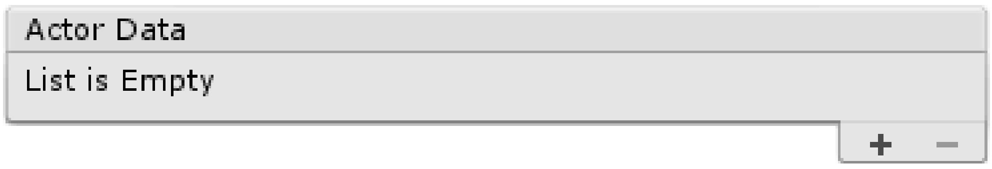


Fig. 2.1: Actor Data List (Empty)

- When you have multiple entries in this list, you can drag them around to rearrange them.

Searching through library entries

1. Select the library that you want to search for entries.
2. Locate the search bar on top of the inspector window.
3. Type the name of the library entry/entries that you want to find.

- **Note:** Some libraries implements searching by category.
  - If this is the case, the search bar will have a label to its left and a dropdown indicator in its icon.
  - Click on either the label or the icon to reveal a dropdown menu which shows all categories to limit the search to.

Library specific creation

Every game data has their own properties. Because of that, each data type has its own way of creating its entries. Each one is discussed in its own section in this document.

Creating Actor Data

Note: This section contains data relevant only to the provided TreeStoryVN package. If you plan to create actors for a different game genre, or if you want to create a different implementation for your game actors, please proceed to the "Extending the Library" section of this document.

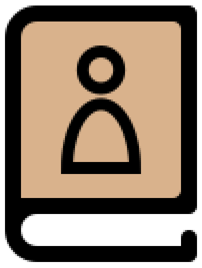


Fig. 2.2: The Actor Library

TreeStoryVN actor data properties

- **Actor name**

- The name of the actor.

- **Actor name color**
  - The color that is used for the actor name text that is shown inside the game.
- **Actor sprites**
  - The sprites that are used to show the actor inside the game.
  - Multiple sprites can be attached to a single actor.
    - This can be useful if you want to add multiple poses for each of them.

The image below shows the actor data list with 3 actor entries, with their names filled-up and colors selected.

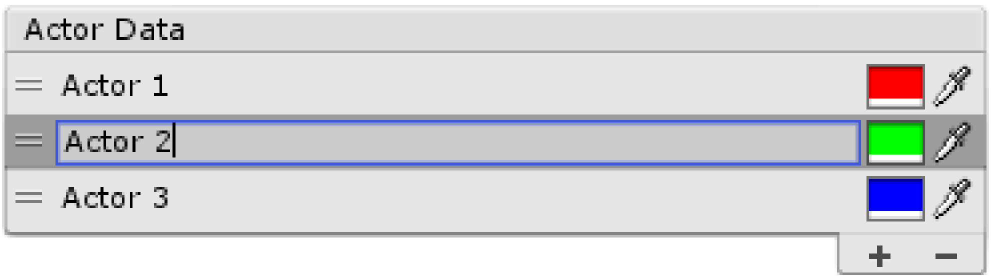


Fig. 2.3: Actor Data List (Multiple Entries)

Editing the actor name

1. Click on the actor entry that you want to edit.
2. Click on the actor entry again to reveal a textbox.
3. Edit this textbox to set the actor name.

Editing the actor name color

1. Select the color picker at the right of the actor entry that you want to edit.
2. Select the color that you want from the window that pops up.

Editing actor sprites

1. Select the actor entry that you want to attach sprites to.
   - This will show you another list below the actor data list.
   - This is the actor sprites list and this is what you will use to attach sprites to the selected actor.
2. Press the (+) button on the lower right of the actor sprites list to create a new sprite entry to the selected actor.
   - You should see an empty sprite object.
3. Drag your actor asset from the hierarchy window into this field to attach it to the selected actor.
   - Alternatively, you can use the select button on the lower right of the sprite field.

- Press the (-) button on the lower right of the actor sprites list to remove the selected sprite from selected actor.
- As with any data list, you can also drag actor sprites around inside this list to rearrange them.

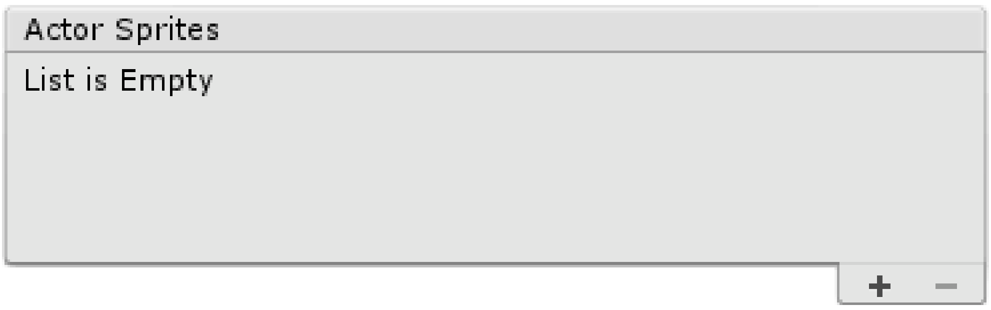


Fig. 2.4: Actor Sprites List (Empty)

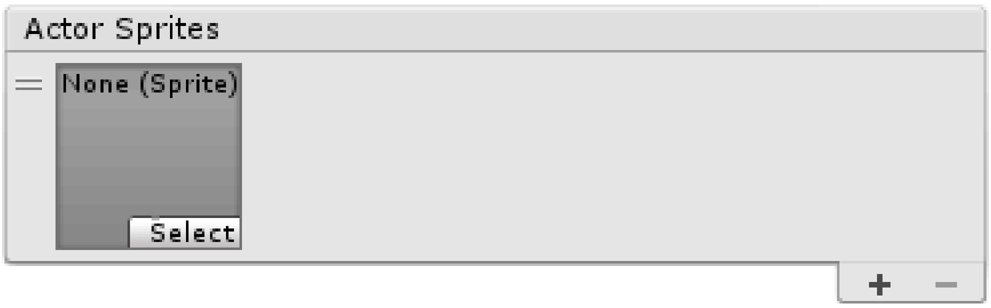


Fig. 2.5: Actor Sprites List

Creating Location Data

Note: This section contains data relevant only to the provided TreeStoryVN package. If you plan to create locations for a different game genre, or if you want to create a different implementation for your game actors, please proceed to the "Extending the Library" section of this document.

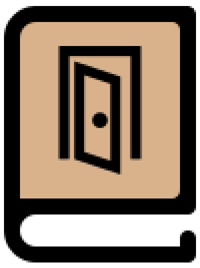


Fig. 2.6: The Location Library

TreeStoryVN location data properties

- **Location name**
  - The name of the location.
- **Location sprite**
  - The sprite that is used as the background of the game for the corresponding location.
  - Unlike actors, you can only attach one sprite to a location.

The image below shows the location data list with 3 location entries, with their names filled-up.

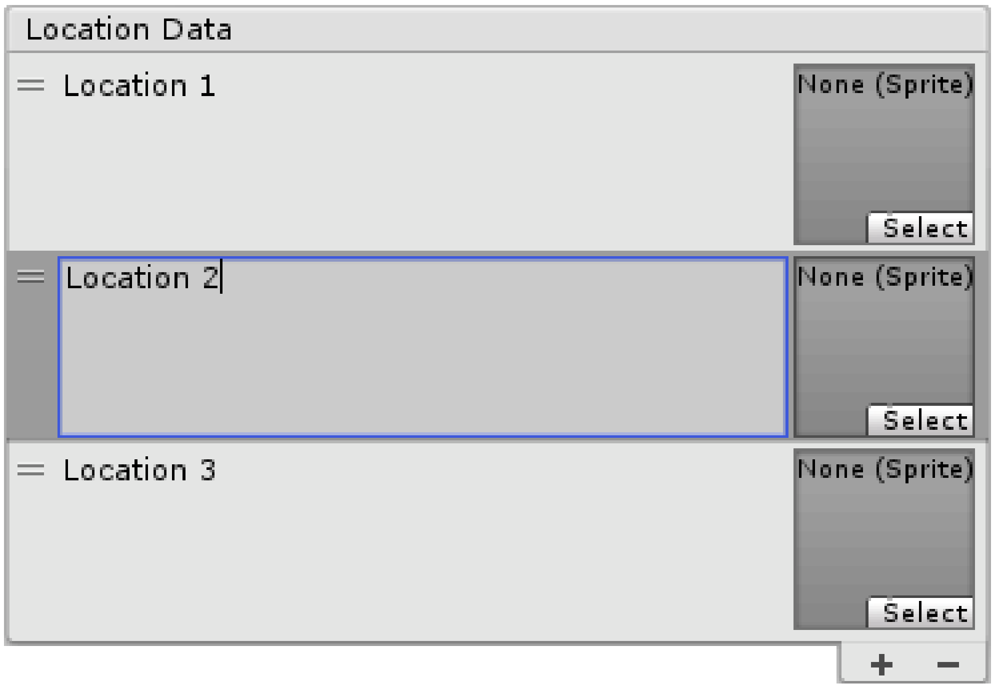


Fig. 2.7: Location Data List (Multiple Entries)

Editing the location name

1. Click on the location entry that you want to edit.
2. Click on the location entry again to reveal a textbox.
3. Edit this textbox to set the location name.

Editing the location sprite

1. Locate the sprite field to the right of the location entry that you want to edit.
2. Drag your location asset from the hierarchy window into this field to attach it to the selected location.
   - Alternatively, you can use the select button on the lower right of the sprite field.

Creating Variable Data

Fig. 2.8: The Location Library

Creating a new variable entry

- Unlike other libraries which you only need to click the (+) button to create a new entry, clicking the button instead opens a popup menu.
- This popup menu shows all available variable types that you can create.
- Select the type that you want to add to create a new variable entry of the selected type.

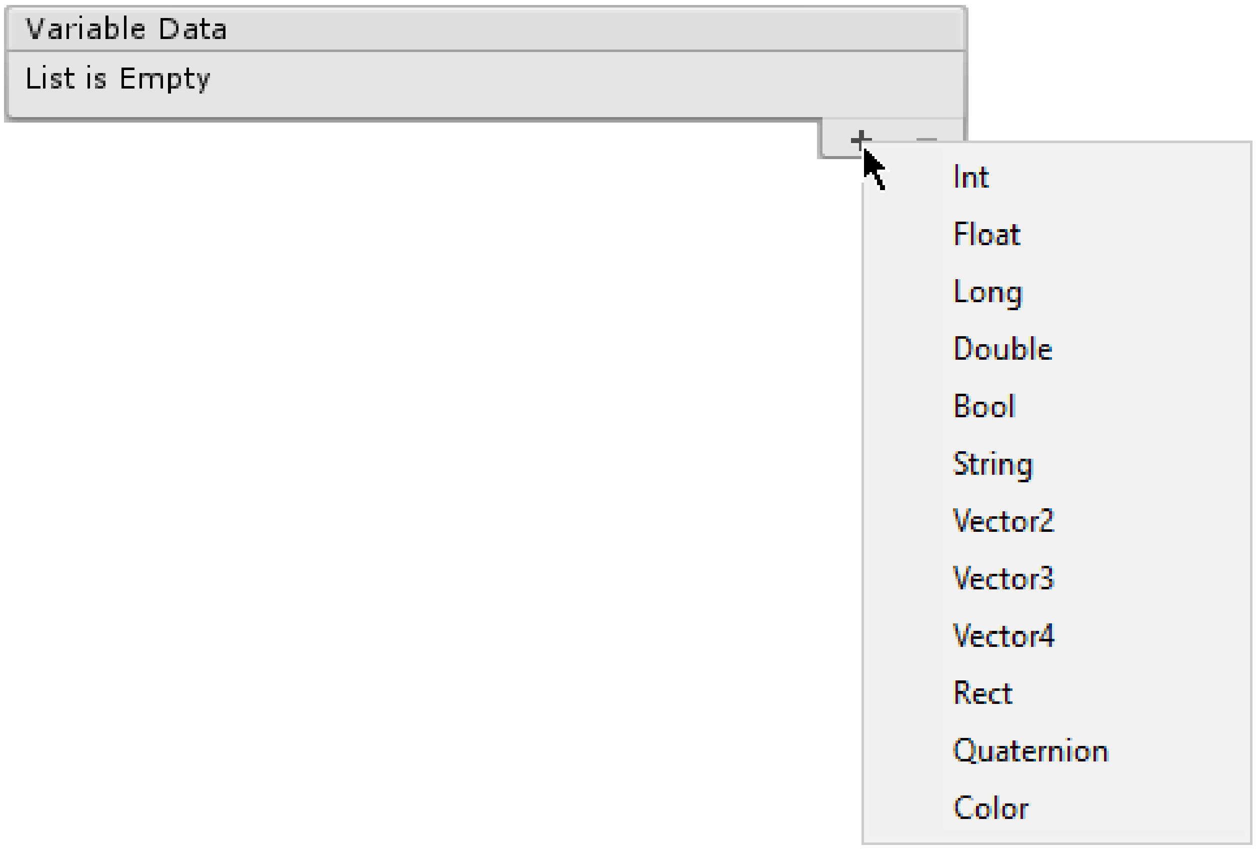


Fig. 2.9: Creating A New Variable Data

Changing the variable type

1. Right click on the variable entry that you want to change.
2. Go to the "Change Type" sub-menu.
3. Select the type that you want to change to.

TreeStory variable data properties

- **Variable name**
  - The name of the variable.
- **Variable value**
  - The value of the variable.

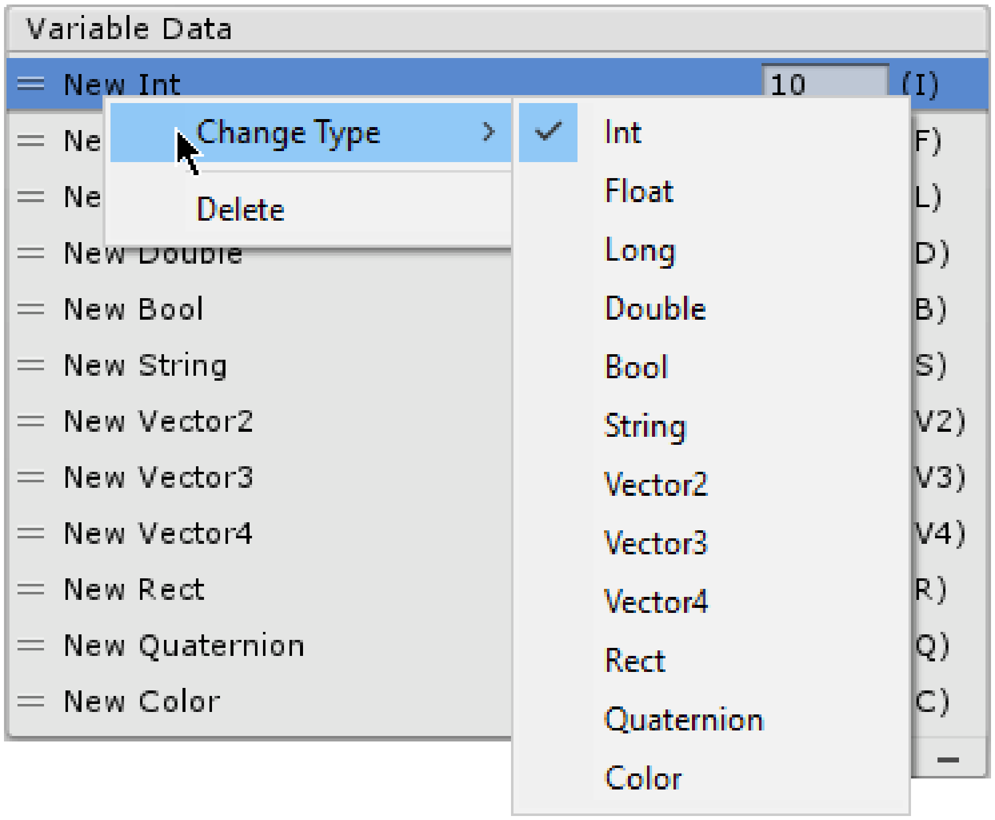
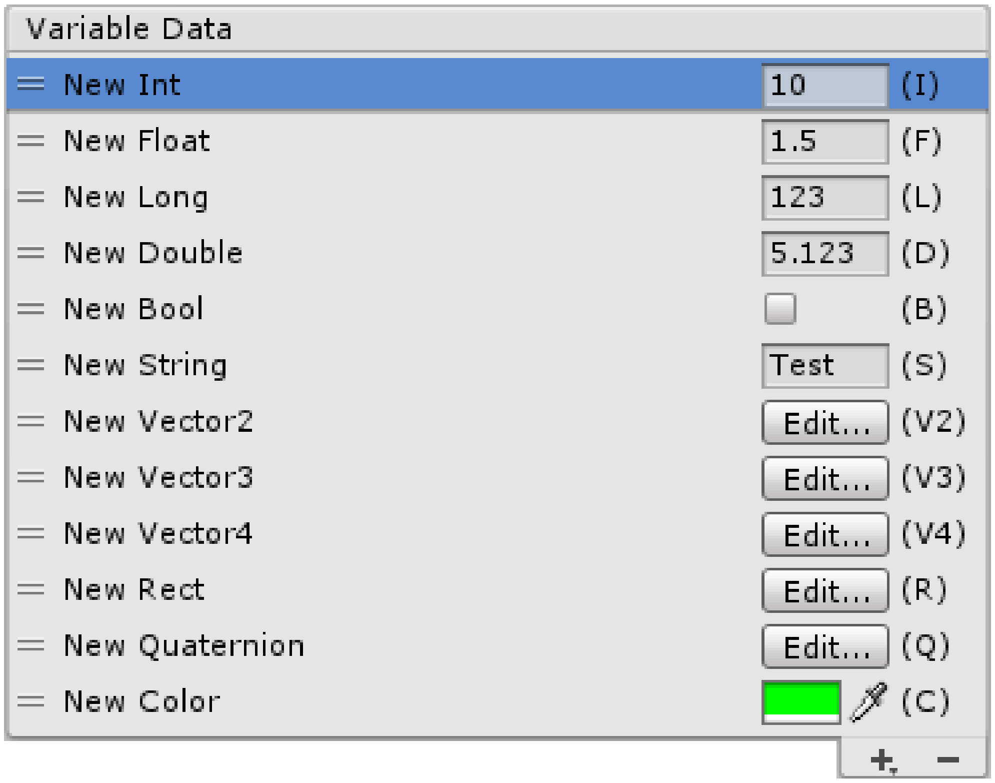


Fig. 2.10: Changing The Variable Type

- **Variable type identifier**
  - A text identifier to enable you to differentiate the type of each variable entry inside the library.
  - **Values:**
    - Int (I)
    - Float (F)
    - Long (L)
    - Double (D)
    - Bool (B)
    - String (S)
    - Vector2 (V2)
    - Vector3 (V3)
    - Vector4 (V4)
    - Rect (R)
    - Quaternion (Q)
    - Color (C)

The image below shows the variable data list with all types of variable entries, with their names and values filled-up.



Variable Data

New Int	10	(I)
New Float	1.5	(F)
New Long	123	(L)
New Double	5.123	(D)
New Bool	☐	(B)
New String	Test	(S)
New Vector2	Edit...	(V2)
New Vector3	Edit...	(V3)
New Vector4	Edit...	(V4)
New Rect	Edit...	(R)
New Quaternion	Edit...	(Q)
New Color	🟩	(C)

Fig. 2.11: Variable Data List (Multiple Entries)

Editing the variable name

1. Click on the variable entry that you want to edit.
2. Click on the variable entry again to reveal a textbox.
3. Edit this textbox to set the variable name.

Editing the variable value

- Each variable type have a different way of editing them.
- Most of them are just simple edittable fields at the right of its entry in the inspector.
- For complex types, buttons labelled "**Edit...**" are placed instead of a simple field.
  - Clicking this button shows a popup window that contains the fields which you could edit to change the variable's value.
- **Simple types:**
  - Int
  - Float
  - Long
  - Double
  - Bool
  - String
  - Color
- **Complex types:**
  - Vector2
  - Vector3
  - Vector4
  - Rect
  - Quaternion

CHAPTER 3

The TreeStory Explorer

When your project grows to have a lot of libraries, it can be difficult to manage them. With that in mind, an explorer window has been provided.

- The explorer groups all library files into their own list.
- Entries for each list are arranged alphabetically.
- Clicking on an entry automatically selects your library asset as if you clicked on it in the project hierarchy window.
- Clicking on an entry also automatically opens the TreeStory editor.

Opening the TreeStory Explorer

1. Click on "Window" in Unity's menu bar.
2. Select "TreeStory Explorer".

Searching for libraries

1. Locate the search bar on top of the TreeStory Explorer window.
2. Type the name of the library that you want to find.

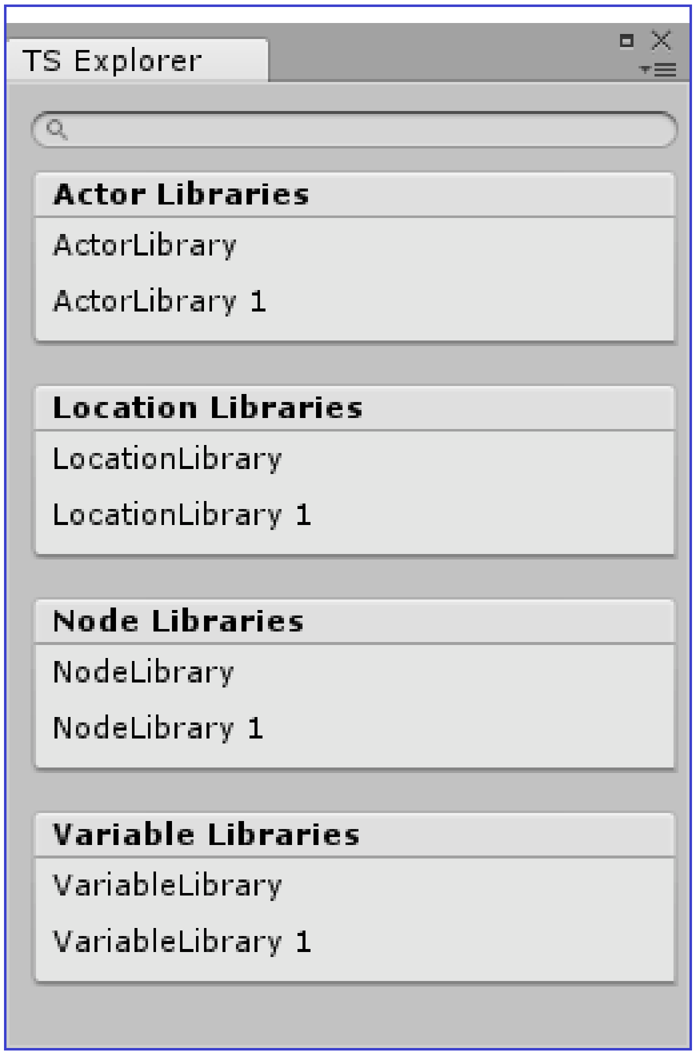


Fig. 3.1: The TreeStory Explorer

CHAPTER 4

Using the TreeStory Editor

In TreeStory, nodes are what makes your game work. It controls everything that your users will see and hear in your game. Because of that, you are most likely going to spend most of your time in the TreeStory editor, working on these nodes. Below are what you will need to start working on the nodes for your game.

Moving the editor view

- You can move around the editor view by clicking with the middle mouse button and dragging the mouse.
- You can also press and hold the alt-key from your keyboard and dragging with the left mouse button.
- Your pan position in the editor is indicated in the toolbar on top of the editor.
  - This indicator is editable and you can use it to jump quickly to different parts of your node library.
  - The values shown here are the grid position.

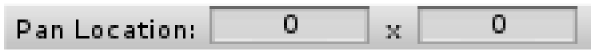


Fig. 4.1: Editor Pan Location

Resetting the editor view

1. Right-click on an empty space on the editor window.
2. Select “Reset View” from the context menu.

Adding a node

1. Right-click on an existing node or on an empty space.

2. Select any of the add node options from the context menu.

- If you right-clicked on an existing node, the new node will automatically be the target of that node.
- If you right-clicked on an empty space, the new node will just be floating and won't be used in-game.
- Node targets can be set at any time so you don't need to worry if you made a mistake.

Deleting a node

1. Right-click on the node that you want to delete.
2. Select "Delete" from the context menu.
   - Alternatively, you can just select the node and press the delete key from your keyboard.

Moving nodes

You can move your selected nodes around the editor by dragging them using the left mouse button.

Setting node targets

1. Right-click on the node that you want to set as the parent.
2. Select "Set Target" from the context menu.
   - You should see an arrow from the parent node following your cursor.
3. Select the node that you want to set as its target.
   - If you want to cancel setting of targets, you can click on an empty space on the editor window.

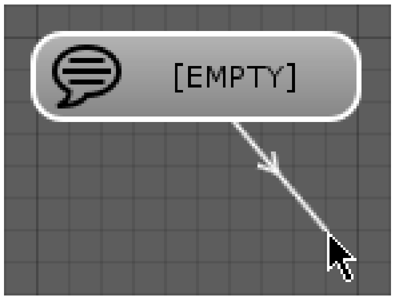


Fig. 4.2: Set Node Target

- Each node can only have one target node except for the dialog node which can have multiple target "choice" nodes.
- If you set a new target for a node that already has a target, then the new target will override the old one.
- If a dialog node targets choice nodes AND a different type of node, it will prioritize the choice nodes and disregard the other target node.

Unsetting node targets

1. Right-click on the node that you want to remove the targets.
2. Select "Unset Targets" from the context menu.
   - If you unset the targets of a dialog node which has at least one choice node as its target, you should see a red arrow from the dialog node following your cursor.
3. If you have a red arrow following your cursor, select the choice node that you want to unset as a target.
   - If you want to cancel unsetting of targets, you can click on an empty space on the editor window.

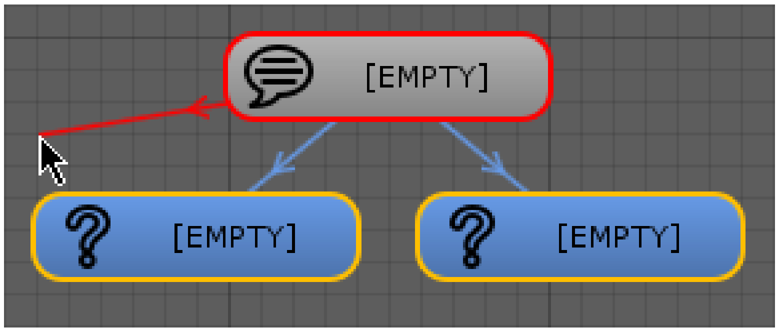


Fig. 4.3: Unset Node Target (Choice Node)

Selecting multiple nodes

Method 1:

1. Hold the control or shift key on your keyboard.
2. Select an unselected node to add it to the selection.
   - Selecting an already selected node removes it from the selection.

Method 2:

1. (**Optional**) Hold the control or shift key on your keyboard.
   - This will add the contents of the selection box to the current selection.
2. Click on an empty space on the editor window.
3. Drag over all the nodes that you want to select.

Notes

- Selected nodes have WHITE borders around them.
- End nodes have YELLOW borders around them.
  – End nodes are nodes that do not have a target.
  – They are the end points to your game.

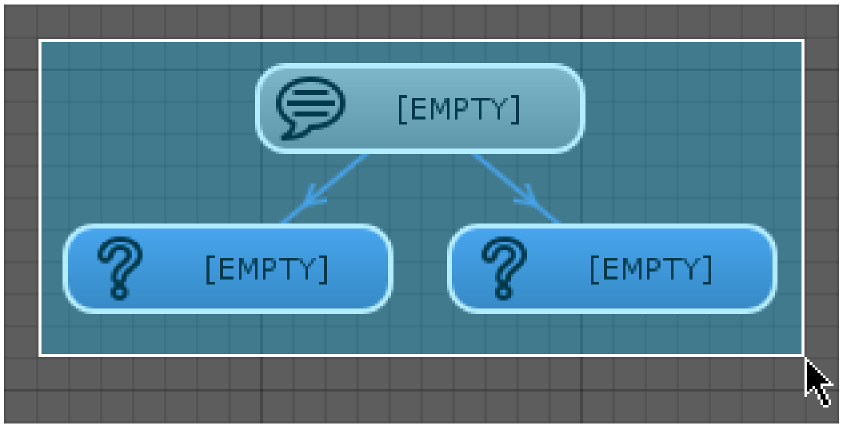


Fig. 4.4: Node Selection Box

CHAPTER 5

Editing Nodes

Editing nodes are done in Unity's inspector window. Selecting a node reveals options in the inspector window. Editing multiple nodes are possible if they are all the same type.

Common node properties

- **Node type name**
  - The name of the selected nodes' type. Used only for identification.
- **Wait to finish**
  - Only some nodes have this property editable since the rest of the nodes will function the same way no matter the value of this property.
  - If CHECKED, the game will wait for this node to finish its tasks before moving on to the next node.
  - If UNCHECKED, then the node's link in the editor window will change from a single white line to a double yellow line to represent it better.

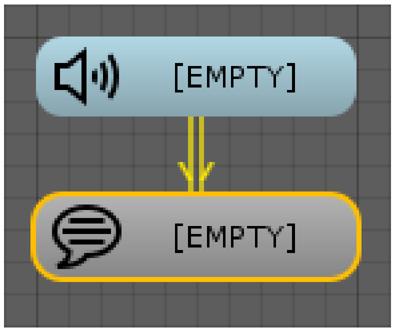


Fig. 5.1: Instant Node Link

- **Data list**

   - Some types of nodes can accept data to make them more dynamic.
   - Each node may accept different data types. These will be discussed in their own sub-section for each applicable node.
   - The index for each data entry is shown at the left of each entry in the data list for your reference.
   - The data type for each entry in the list is shown beside its index.
   - You can add your own data to this by extending the library (discussed later).

Selecting the next node

- You can select the next node that the one you are currently editing is pointing to by pressing the button at the top right of the inspector window labelled "**Select Next Node**".
- This button only appears if the current node has a child node that it accepts only one of.
- If the current node accepts more than one of its child nodes, like the choice nodes of the dialog node, then selecting the child node depends on the implementation of the nodes children within its own inspector.
   - In the case of the choice nodes of the dialog node, buttons labelled "**Select**" are placed to the right of each choice entry in the "Choices" reorderable list inside the dialog node's inspector.

Editing the Data List

1. Press the (+) button on the lower right of the data list to create a new data entry to the current node.

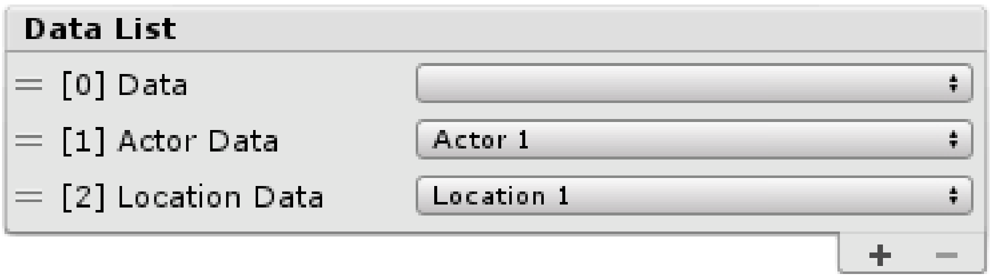


Fig. 5.2: Node Data List (Multiple Entries)

2. Click on the dropdown list of the data entry that you want to edit.
3. Select the type of library which contains the data.
   - This option only appears if the node accepts multiple data types.
   - Pressing the "[None]" option will remove the data associated with the current entry.
4. Select the library where the data is located.
   - This option only appears if you have multiple libraries of the selected type.
5. Select the data that you want to use.

- Press the (-) button on the lower right of the data list to remove the selected data from the node.
   - If there is data currently in the selected entry, pressing the (-) button will remove it, similar to pressing the "[None]" option.

  - Pressing it a second time will remove the entry altogether.
- As with any data list, you can also drag entries around inside this list to rearrange them.

Editing the Start Node

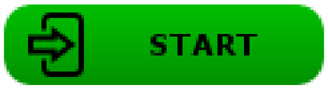


Fig. 5.3: Start Node

- The start node is a special type of node.
- There is only one of it in every node library.
- This node cannot be deleted.
- This is where your game starts reading from your node library.

Start node properties

- **Start faded**
  - If CHECKED, the game will fade in from black.

Editing the Dialog Node

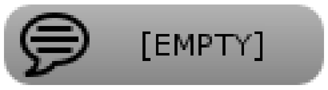


Fig. 5.4: Dialog Node

- The dialog node controls the dialog that is shown in-game.
- It also controls the actor's name that will be displayed with the dialog in-game.
- To enable it in-game, you need to have a dialog box pre-made in the game's GUI.
  - This will be discussed later in the "Creating the Game UI" section.

Dialog node properties

- **Wait to finish**
- **Actor Name**
  - The name of the actor that is associated (speaking, thinking, etc.) with the current dialog.
  - TreeStory's actor name accepts Unity's rich text formatting.
    - Please refer to Unity's documentation on how to use rich text formatting.
  - It also accepts C# style text formatting with its parameters taken from the data list discussed below.
    - Example: "Mister {0}"

      - Where {0} is the first entry in the data list.
    - Be careful about your indices as adding, removing, or moving your data might alter their index.
      - When this happens, you must adjust the indices that you have already used accordingly.
  - It is recommended to use the data list for the actor names in case you rename any actor in the future.
    - Doing this prevents the need for you to change every single name for each dialog node, and instead just change one entry in the actor library.
  - A preview for the formatted actor name can be seen at the bottom of the dialog node's inspector window.

- **Text**
  - The text displayed for the dialog in-game.
  - TreeStory's dialog accepts Unity's rich text formatting.
    - Please refer to Unity's documentation on how to use rich text formatting.
  - It also accepts C# style text formatting with its parameters taken from the data list discussed below.
    - Example: "Hello my name is {0}. I live in {1}."
      - Where {0} and {1} are the first and second entry in the data list, respectively.
    - Be careful about your indices as adding, removing, or moving your data might alter their index.
      - When this happens, you must adjust the indices that you have already used accordingly.
  - A preview for the formatted dialog text can be seen at the bottom of the dialog node's inspector window.
- **Data list**
  - A list of data attached to the current dialog node used to make its text dynamic.
  - **Accepted data:**
    - Actor data
    - Location data
    - Variable data
- **Choices**
  - A list of text of all choice nodes which is targeted by the current dialog node.
  - These choice node texts can be edited directly from here.
  - You can rearrange the order of choices from this list by dragging them.
    - This affects how the choices are ordered in-game as well.
  - But you cannot add or remove choice nodes from this node.
- **Data list (Selected choice)**
  - The data list for the selected choice from the choices list.
  - Refer to the data list property of the choice node below.

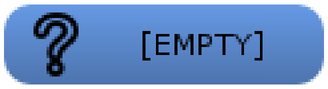


Fig. 5.5: Choice Node

Editing the Choice Node

- The choice node control the choices that are shown in-game.
- It can only be targeted by a dialog node.
- To enable it in-game, you need to have enough buttons pre-made in the game's GUI to accommodate for the number of choice nodes.
  - This will be discussed later in the "Creating the Game UI" section.
  - **Quick example:** If you have 3 choice nodes for a dialog node, you must have 3 or more choice buttons pre-made in the game's GUI.

Choice node properties

- **Text**
  - The text displayed for the choice in-game.
  - TreeStory's choices accepts Unity's rich text formatting.
    - Please refer to Unity's documentation on how to use rich text formatting.
  - It also accepts C# style text formatting with its parameters taken from the data list discussed below.
    - **Example:** "Hello my name is {0}. I live in {1}."
      - Where {0} and {1} are the first and second entry in the data list, respectively.
    - Be careful about your indices as adding, removing, or moving your data might alter their index.
      - When this happens, you must adjust the indices that you have already used accordingly.
  - A preview for the formatted choice text can be seen at the bottom of the choice node's inspector window.
- **Data list**
  - A list of data attached to the current choice node used to make its text dynamic.
  - **Accepted data:**
    - Actor data
    - Location data
    - Variable data

Editing the Actor Node

Note: This section contains data relevant only to the provided TreeStoryVN package.

- The actor node controls the behavior of an actor in-game.
- The actors that this node uses should be first made inside the actor library.

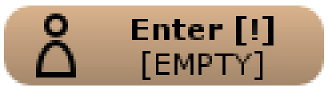


Fig. 5.6: Actor Node

Actor node properties

- **Wait to finish**
- **Actor**
  - The actor which is associated with this node.
  - You can also set this property by right-clicking on the node inside the TreeStory editor and selecting “Set Actor”.
  - The rest of the properties won’t appear in the inspector until you have selected an actor.
- **Event**
  - The type of event that will happen with this node’s associated actor.
  - **Values:**
    - **Enter**
      - The actor enters the scene.
    - **Move**
      - Move an already existing actor’s position.
    - **Sprite**
      - Change an already existing actor’s sprite.
    - **Exit**
      - Exit an already existing actor out of the scene.
  - Each event, when selected, will show different properties for the current actor node.

Enter event properties

- **Sprite**
  - The sprite used by the associated actor.
  - This property cannot be empty, otherwise the game will crash.
  - If it is empty, the node’s preview in the editor (the image above) will show an exclamation mark to help you identify which nodes have empty sprite properties.
- **Enter from**
  - Where the actor will enter from.
  - **Values:**
    - **Pop**
      - Appear immediately into the screen.
    - **Fade**

        · Fades slowly into the screen.
    * **Left**
        · Enters from the left of the screen.
    * **Right**
        · Enters from the right of the screen.
    * **Top**
        · Enters from the top of the screen.
    * **Bottom**
        · Enters from the bottom of the screen.

- **Screen origin**
  - The origin of the screen, in percent, where the target position will be based on.
- **Target position**
  - The target position, in pixels from the screen origin, for the actor.
- **Event duration**
  - How long the actor will take to move to its target position.
  - This property only appears if the “Enter from” property is NOT set to “Pop”.

Move event properties

- **Move X**
  - The type of the actor’s horizontal movement.
  - **Values:**
    - **None**
      - The actor will NOT move horizontally.
    - **Absolute**
      - The actor’s exact target horizontal position will be set by you.
    - **Relative**
      - The actor will move horizontally by the specified amount (in pixels) relative to its current position.
- **Screen origin X**
  - The horizontal origin of the screen, in percent, where the target position will be based on.
  - This property will only be visible if the “Move X” property is set to “Absolute”.
- **Target X position**
  - The target horizontal position, in pixels from the horizontal screen origin, for the actor.
  - This property will only be visible if the “Move X” property is NOT set to “None”.
- **Move Y**
  - The type of the actor’s vertical movement.

  - **Values:**
    - **None**
      - The actor will NOT move vertically.
    - **Absolute**
      - The actor's exact target vertical position will be set by you.
    - **Relative**
      - The actor will move vertically by the specified amount (in pixels) relative to its current position.

- **Screen origin Y**
  - The vertical origin of the screen, in percent, where the target position will be based on.
  - This property will only be visible if the "Move Y" property is set to "Absolute".
- **Target Y position**
  - The target vertical position, in pixels from the vertical screen origin, for the actor.
  - This property will only be visible if the "Move Y" property is NOT set to "None".
- **Event duration**
  - How long the actor will take to move to its target position.

Sprite event properties

- **Sprite**
  - The sprite used by the associated actor.
  - This property cannot be empty, otherwise the game will crash.
  - If it is empty, the node's preview in the editor (the image above) will show an exclamation mark to help you identify which nodes have empty sprite properties.

Exit event properties

- **Exit to**
  - Where the actor will exit to.
  - **Values:**
    - **Pop**
      - Dissapear immediately from the screen.
    - **Fade**
      - Fades slowly out of the screen.
    - **Left**
      - Exits to the left of the screen.
    - **Right**
      - Exits to the right of the screen.

        - **Top**
            - Exits to the top of the screen.
        - **Bottom**
            - Exits to the bottom of the screen.

- **Event duration**
    - How long the actor will take to move to its target position.
    - This property only appears if the "Exit to" property is NOT set to "Pop".

Editing the Location Node

Note: This section contains data relevant only to the provided TreeStoryVN package.

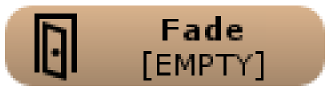


Fig. 5.7: Location Node

- The location node controls the behavior of locations in-game.
- The locations that this node uses should be first made inside the location library.
- For the TreeStoryVN package, the locations is represented as the background image in-game.

Location node properties

- **Wait to finish**
- **Location**
    - The location associated with this node.
    - You can also set this property by right-clicking on the node inside the TreeStory editor and selecting "Set Location".
- **Transition**
    - The transition effect used to transfer locations.
    - **Values:**
        - **No transition**
            - Don't use transitions while transferring locations.
        - **Fade**
            - Starts with the screen black and fading in.
- **Exit actors**
    - If CHECKED, all actors currently in-game will exit.

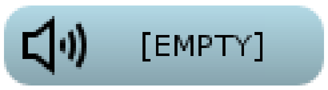


Fig. 5.8: Audio Node

Editing the Audio Node

- The audio node controls the behavior of audio in-game.
- The audio data that this node uses should be first made inside the audio library.

Audio node properties

- **Wait to finish**
- **Audio**
  - The audio file that will be played.
  - Set this to none to only control the background music.
- **Is BGM**
  - If checked, the associated audio will be set as the background music of the game.
  - Only one BGM is allowed to play at a time.
  - If you want to stop the background music from playing, set the audio to none and check this property.
- **Change BGM volume**
  - If checked, the volume of the game's "background music" will be set to the selected value.
- **Target BGM volume**
  - The amount of volume that the game's "background music" will be set to.
  - This option will only appear if the "Change BGM Volume" property above is checked.
  - This property can be set from 0.0 to 1.0
  - **Note:** If this property is set to 0.0, the background music will still be playing, the user just won't hear it.

Editing the Video Node

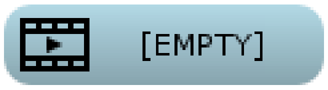


Fig. 5.9: Video Node

- The video node controls the behavior of video in-game (like cut scenes).
- The video data that this node uses should be first made inside the video library.

Video node properties

- **Is skippable**
  - If CHECKED, the video will be skippable by the user using the normal dialog controls.
- **Video**
  - The video file that will be played.
- **Stop all sounds**
  - If CHECKED, all sounds that are currently playing (BGM and all sound effects) will stop.
  - **Note: The sounds will NOT start again once the video has finished.**
    * If you want to continue the BGM playback, you have to play it again using a new audio node.

Editing the Library Node

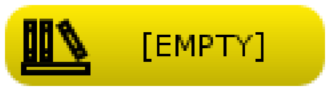


Fig. 5.10: Library Node

- The library node makes the game continue using a different node library.
- This enables you to split your project into multiple node libraries but still use it like a single one.
- This is useful if you work in a team and each member works on a different part of the game.
- It's also useful in making your project more manageable.

Library node properties

- **Library**
  - The TreeStory node library that the game will continue on.
  - The game will finish that library before moving on to the next node, if any.
  - You can also set this property by right-clicking on the node inside the TreeStory editor and selecting "Set Library".

Editing the Scene Node

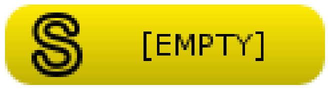


Fig. 5.11: Scene Node

- The scene node controls the behavior of scenes in-game.
- An example for this node's use is for your game's menus.

- **Note: Changing scenes with the "single" scene mode makes Unity forget all of its current data unless you use specific scrip**
    - TreeStory provides you with prefabs to keep some data for saving and loading data to your game.

Scene node properties

- **Scene**
    - The new scene that will be loaded.
    - You can also set this property by right-clicking on the node inside the TreeStory editor and selecting "Set Scene".
- **Load scene mode**
    - The mode that Unity will use to load the new scene.
    - **Values:**
        - **Single**
            - Close all open scenes and load the new scene.
        - **Additive**
            - Open the new scene alongside all open scenes.
- **Transition**
    - The transition effect used to transfer scenes.
    - **Values:**
        - **No transition**
            - Don't use transitions while transferring scenes.
        - **Fade**
            - Starts with the screen black and fading in.

Editing the Save Node

Fig. 5.12: Save Node

- The save node saves game data to a file.
- You can load the created data using the load node.
- The save files are located in Unity's persistent data path.
    - For windows it is located in:

```
"C:\Users\<User Name>\AppData\LocalLow\<Company Name>\<Project Name>\Save␣
↪Data\"
```

Save node properties

- **File name**
  - The file where the save data will be stored.

Editing the Load Node

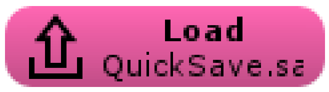


Fig. 5.13: Load Node

- The load node loads game data from a file.
- The save files are created using the save node.
- This node automatically loads the main scene where the save file was created.

Load node properties

- **File name**
  - The file where the save data are stored.

Editing the Quit Node

Fig. 5.14: Quit Node

- The quit node quits the game.
  - **Note:** Unity does not quit the game if it runs in the editor. Only in the published game.
- This node has no additional properties.

CHAPTER 6

TreeStoryVN Prefabs

- The TreeStoryVN package provides you with prefabs to help you make your game faster and easier.
- The prefabs are controlled by the nodes that you created in the node library.
  - You can opt not to add specific prefabs if they have no node counterpart in the node library.
    - **Example:** If you don't have any video node in the node library, then you don't need to add a video player prefab to your game.
- Each prefab works independently from each other.
  - You can add or remove each game object without worrying that you'll break another.
- The order of prefabs inside your game doesn't matter (Except for the drawing order).
  - The only requirement is that all prefabs should be inside a parent object which contains a "TreeStoryVN" script.
- If you want to use more than one of a specific prefab to your scene, you have to provide additional scripts to make them work.
  - **Example:** If you want to have 2 name boxes for multiple actors at the same time, then you need to provide your own script.
  - An exception for this are the regular and choice buttons as TreeStory can use an unlimited number of those.

The MainCamera Prefab

- Prefab location:

```
TreeStory/Prefabs/Singletons/MainCamera
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/Singletons/MainCamera.cs
```

- This is basically Unity's camera object.
- This prefab is a singleton object.
  - Only one MainCamera object exists in the game at all times.
  - If a new scene loads with a new MainCamera object, then that object gets destroyed.
  - It doesn't get destroyed by a scene change.
- If your game needs more than one camera, then don't use this prefab.
- No customizable script properties.

The EventSystem Prefab

- Prefab location:

```
TreeStory/Prefabs/Singletons/EventSystem
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/Singletons/EventSystem.cs
```

- This is basically Unity's event system object.
- This prefab is a singleton object.
  - Only one EventSystem object exists in the game at all times.
  - If a new scene loads with a new EventSystem object, then that object gets destroyed.
  - It doesn't get destroyed by a scene change.
- No customizable script properties.

The SaveSystem Prefab

- Prefab location:

```
TreeStory/Prefabs/SaveSystem/SaveSystem
```

- Script location:

```
TreeStory/TreeStorySaveSystem/Scripts/SaveSystem.cs
```

- This prefab is actually part of the TreeStorySaveSystem package.
- It handles saving and loading of data for your TreeStory game.
- This prefab is a singleton object.
  - Only one SaveSystem object exists in the game at all times.
  - If a new scene loads with a new SaveSystem object, then that object gets destroyed.
  - It doesn't get destroyed by a scene change.

- No customizable script properties.

The AudioManager Prefab

- Prefab location:

```
TreeStory/Prefabs/Audio/AudioManager
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/Audio/AudioManager.cs
```

- **Controlled by:**
  - Audio nodes
  - Video nodes
    - Only able to stop all audio controlled by this prefab.
- Handles both the background music and all sound effects.
- This prefab is a singleton object.
  - Only one audio manager object exists in the game at all times.
  - If a new scene loads with a new AudioManager object, then that object gets destroyed.
  - It doesn’t get destroyed by a scene change.

Customizable script properties

- **Audio source prefab**
  - A prefab which contains an audio source component.
- **Fade speed**
  - The percentage per frame (0.0 to 1.0) at which the game’s background music fades to the value set by an audio node’s “BGM Fade” property.

The Background Prefab

- Prefab location:

```
TreeStory/Prefabs/Background/Background
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/Background/Background.cs
```

- **Controlled by:**
  - Location nodes
- Shows the background image of the locations in your game.

Customizable script properties

- **Aspect mode (Aspect ratio fitter script)**
  - The type of stretching that will happen to your background image depending on the screen size of the user.
  - **Values:**
    - **None**
      - The background image stretches to the full screen size.
    - **Width controls height**
      - The background image's width will stretch to the full width of the screen.
      - Its height will change to match the original aspect ratio of the image.
    - **Height controls width**
      - The background image's height will stretch to the full height of the screen.
      - Its width will change to match the original aspect ratio of the image.
    - **Fit in parent**
      - The background image will fit itself to the size of the user's screen while keeping its original aspect ratio.
    - **Envelope parent**
      - The background image will fill the size of the user's screen while keeping its original aspect ratio.

The VideoPlayer Prefab

- Prefab location:

```
TreeStory/Prefabs/Overlay/VideoPlayer
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/Overlay/VideoPlayer.cs
```

- **Controlled by:**
  - Video nodes
- Plays the cut scenes for your game.

Customizable script properties

- **Background color**
  - The color used to fill the empty spaces of the screen if the video is smaller than the user's screen.
  - Set this color to transparent (alpha = 0) if you don't want to fill the empty spaces of the screen.
- **Aspect mode**
  - The type of stretching that will happen to your video depending on the screen size of the user.

  - This property overrides the aspect mode property of the aspect ratio fitter component.
  - **Values:**
    - **None**
      - The video stretches to the full screen size.
    - **Width controls height**
      - The video's width will stretch to the full width of the screen.
      - Its height will change to match the original aspect ratio of the image.
    - **Height controls width**
      - The video's height will stretch to the full height of the screen.
      - Its width will change to match the original aspect ratio of the image.
    - **Fit in parent**
      - The video will fit itself to the size of the user's screen while keeping its original aspect ratio.
    - **Envelope parent**
      - The video will fill the size of the user's screen while keeping its original aspect ratio.

The FadeImage Prefab

- Prefab location:

```
TreeStory/Prefabs/Overlay/FadeImage
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/Overlay/FadeImage.cs
```

- **Controlled by:**
  - Start nodes
  - Location nodes
  - Scene nodes
- Enables the fade transition effect for scene and location changes.

Customizable script properties

- **Fade speed**
  - The percentage per frame (0.0 to 1.0) at which the game fades in/out every time the scene or location changes.
  - This is only applicable if the scene or location node's transition is set to fade.

The ActorDrawer Prefab

- Prefab location:

```
TreeStory/Prefabs/Actors/ActorDrawer
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/Actors/ActorDrawer.cs
```

- **Controlled by:**
  - Actor nodes
  - Location nodes
    - Only able to exit all actors controlled by this prefab.
- Handles the behavior of all actors in the scene.

Customizable script properties

- **Actor image prefab**
  - A prefab which contains an image component and an actor image script.
  - The actor image script controls the behavior of a single actor.

The NameBox Prefab

- Prefab location:

```
TreeStory/Prefabs/Dialog/NameBox
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/Dialog/NameBox.cs
```

- **Controlled by:**
  - Dialog nodes
- Shows the name of the actor associated (speaking, thinking, etc.) with the currently shown dialog, if available.
- Hidden if the dialog does not have an associated actor.

Customizable script properties

- **Source image (Image script)**
  - The background image used by the name box.
  - Useful to add borders to your name box.

The DialogBox Prefab

- Prefab location:

```
TreeStory/Prefabs/Dialog/DialogBox
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/Dialog/DialogBox.cs
```

- **Controlled by:**
  - Dialog nodes
- Shows the current dialog being spoken.

Customizable script properties

- **Character speed**
  - The time in seconds that it takes to show each text character to the screen.
- **Source image (Image script)**
  - The background image used by the dialog box.
  - Useful to add borders to your dialog box.

The ChoiceLayer Prefab

- Prefab location:

```
TreeStory/Prefabs/Choices/ChoiceLayer
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/Choices/ChoiceLayer.cs
```

- **Controlled by:**
  - Dialog nodes
  - Choice nodes
- Handles the selection of choices in-game.
- You can set audio clips to this game object to add sound effects every time a choice is highlighted and/or selected.

Customizable script properties

- **Highlight sound**
  - The sound effect used when the user highlights a choice.
  - If this property is not empty, then it overrides the highlight sound property of all its child choices.
- **Select sound**

  - The sound effect used when the user selects a choice.
  - If this property is not empty, then it overrides the select sound property of all its child choices.
- **Choice delay**
  - The time in seconds that the choice blinks when selected before moving on to the next node.
  - Set this to 0 if you want your game to immediately continue to the next node when selecting a choice.
- **Source image (Image script)**
  - The background image used by the choice layer.
  - Useful to add borders to your choices.

The Choice Prefab

- Prefab location:

```
TreeStory/Prefabs/Choices/Choice
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/Choices/ChoiceButton.cs
```

- **Controlled by:**
  - Choice nodes
- Handles a single choice in-game.
- Needs to be placed as a child to the choice layer game object.

Customizable script properties

- **Is enabled**
  - If checked, the button can be selected.
  - Controlled by the ChoiceLayer prefab.
- **Is override enabled**
  - If checked, input events won't affect the highlight and pressed state of the button.
  - Instead an external script can control these states.
  - Controlled by the ChoiceLayer prefab.
- **Normal image**
  - The background image used for the button normally.
- **Highlight image**
  - The background image used for the button when it is highlighted.
- **Pressed image**
  - The background image used for the button when it is pressed.
- **Normal color**

    - The color used for the background image of the button normally.

- **Highlight color**
    - The color used for the background image of the button when it is highlighted.
- **Pressed color**
    - The color used for the background image of the button when it is pressed.
- **Highlight sound**
    - The sound effect used when the user highlights then button.
- **Select sound**
    - The sound effect used when the user selects the button.
- **Node libraries**
    - The node libraries to load inside the game when the choice is pressed.

The Button Prefab

- Prefab location:

```
TreeStory/Prefabs/EventHandlers/Button
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/EventHandlers/Button.cs
```

- A button that loads node libraries to the game whenever it is pressed.
- You must place instances of this prefab inside a parent object which contains a "TreeStoryVN" script component.

Customizable script properties

- **Is enabled**
    - If checked, the button can be selected.
- **Is override enabled**
    - If checked, input events won't affect the highlight and pressed state of the button.
    - Instead an external script can control these states.
- **Normal image**
    - The background image used for the button normally.
- **Highlight image**
    - The background image used for the button when it is highlighted.
- **Pressed image**
    - The background image used for the button when it is pressed.
- **Normal color**
    - The color used for the background image of the button normally.

- **Highlight color**
  - The color used for the background image of the button when it is highlighted.
- **Pressed color**
  - The color used for the background image of the button when it is pressed.
- **Highlight sound**
  - The sound effect used when the user highlights then button.
- **Select sound**
  - The sound effect used when the user selects the button.
- **Node libraries**
  - The node libraries to load inside the game when the button is pressed.

The MouseButtonHandler Prefab

- Prefab location:

```
TreeStory/Prefabs/EventHandlers/MouseButtonHandler
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/EventHandlers/MouseButtonHandler.cs
```

- Loads node libraries to the game whenever the specified mouse button is pressed.
- You must place instances of this prefab inside a parent object which contains a "TreeStoryVN" script component.

Customizable script properties

- **Button**
  - The mouse button that will trigger the loading of the node libraries.
- **Node libraries**
  - The node libraries to load inside the game when the mouse button is pressed.

The KeyboardHandler Prefab

- Prefab location:

```
TreeStory/Prefabs/EventHandlers/KeyboardHandler
```

- Script location:

```
TreeStory/TreeStoryVN/GUI/EventHandlers/KeyboardHandler.cs
```

- Loads node libraries to the game whenever the specified keyboard key is pressed.
- You must place instances of this prefab inside a parent object which contains a "TreeStoryVN" script component.

Customizable script properties

- **Key code**
  - The keyboard key that will trigger the loading of the node libraries.
- **Node libraries**
  - The node libraries to load inside the game when the keyboard key is pressed.

CHAPTER 7

Creating a Simple Visual Novel

- Creating a visual novel using TreeStory is very simple.
- You control how your game works by creating node libraries.
- The next step is figuring out how to present it to your users.
- TreeStory makes that job easier for you by providing you with prefabs as shown in the previous section.
- To make it even easier, TreeStory also provides you with template prefabs.
  - These templates does the job of setting the other prefabs for you.

Setting Up the Base for Each Scene

- Set up the base for all of the scenes for your game using the following steps and TreeStory will make them work seamlessly with each other.

1. Delete the default camera from the scene.

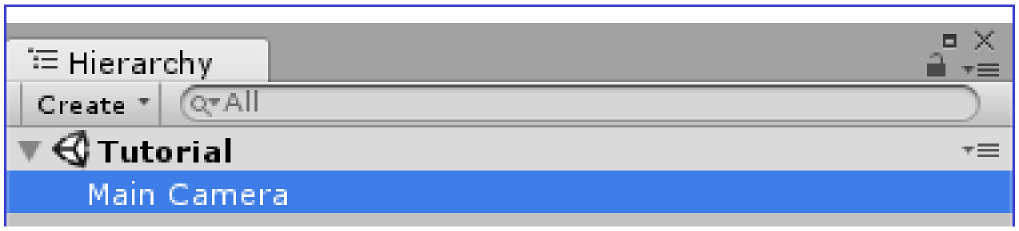
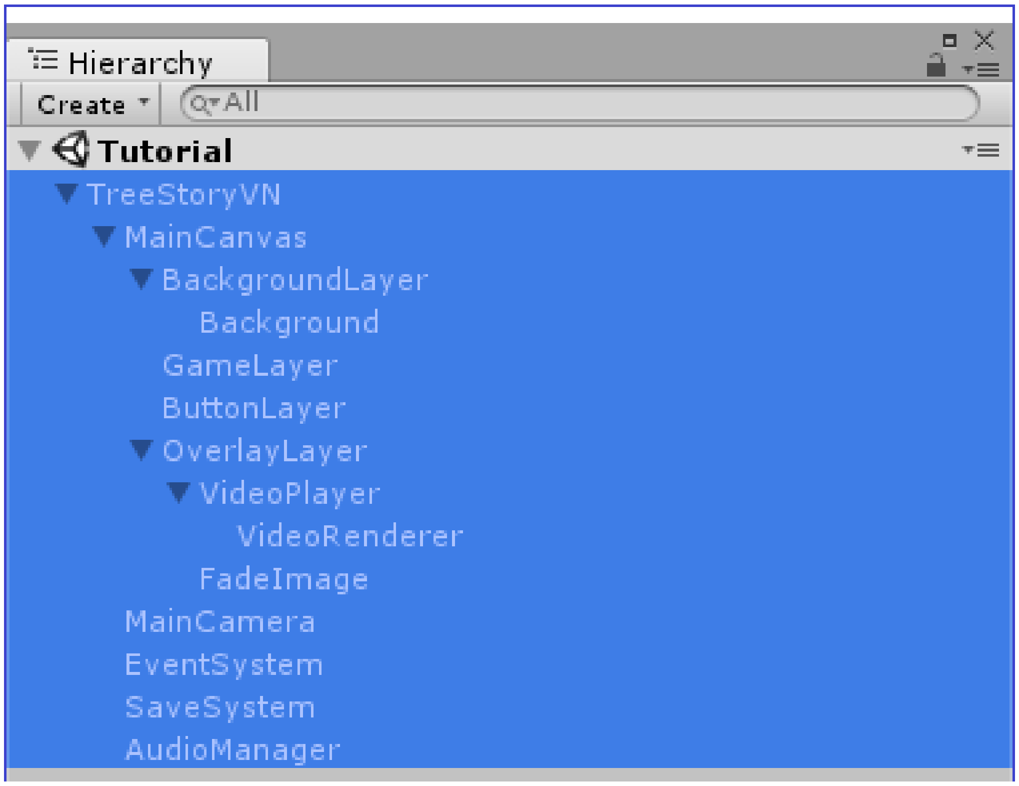


Fig. 7.1: The Default Camera

2. Drag the "TreeStoryVN" prefab into the scene.
   - Location:

```
TreeStory/Prefabs/Templates/TreeStoryVN
```



```
Tutorial
  TreeStoryVN
    MainCanvas
      BackgroundLayer
        Background
      GameLayer
      ButtonLayer
      OverlayLayer
        VideoPlayer
          VideoRenderer
        FadeImage
    MainCamera
    EventSystem
    SaveSystem
    AudioManager
```

Fig. 7.2: The TreeStoryVN Template Prefab

3. Select the "TreeStoryVN" game object from the hierarchy (The root game object).
   - You should see a "Node Library" property field in the inspector window.
4. Drag your main node library asset into this field.
   - Alternatively, you can click on the circle on the right of this field.

- **Note:** You can add buttons to your game by using the "button" prefab.
  - A ButtonLayer object is provided as a parent object for your buttons.

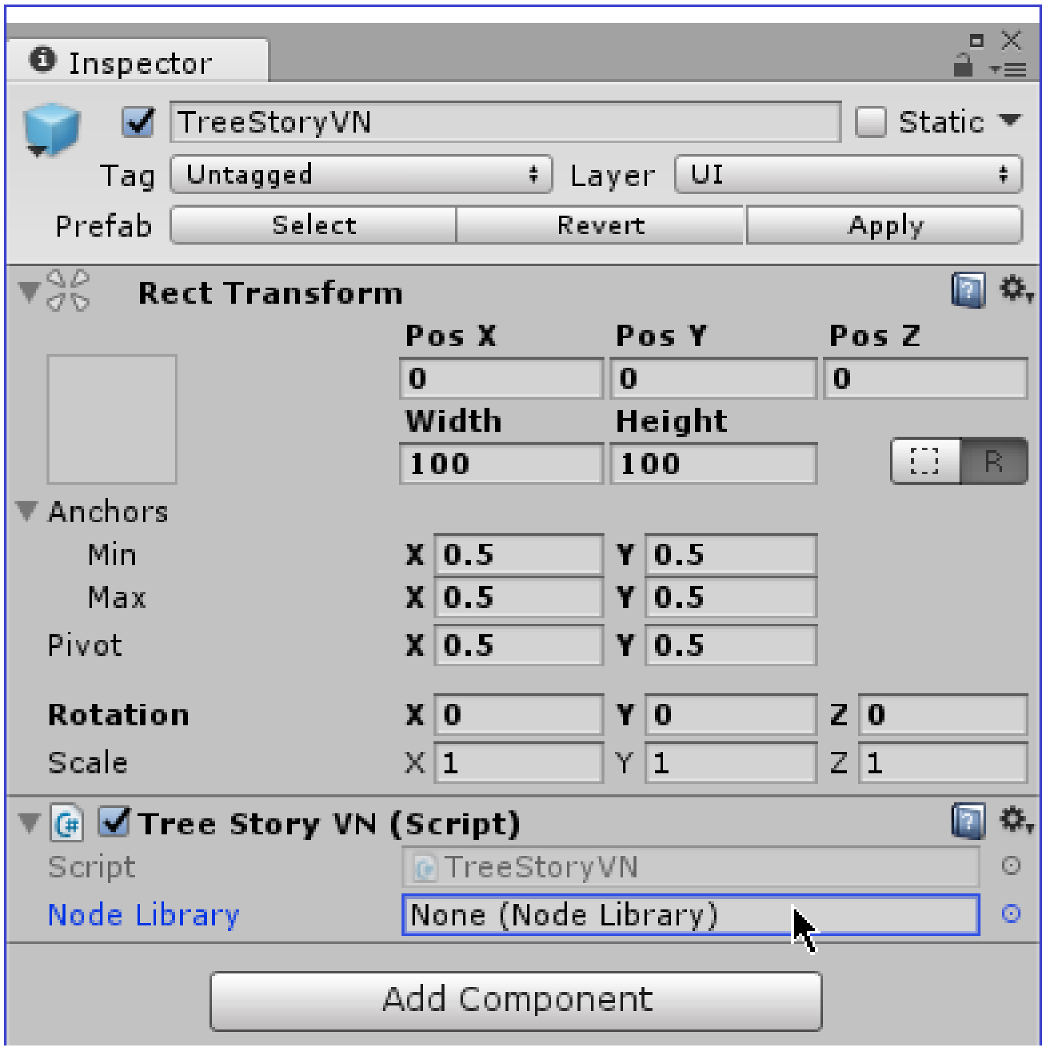
Creating the Main Game

Steps

- Drag the "Game" prefab into the "GameLayer" object in Unity's hierarchy window.
  - Location:

```
TreeStory/Prefabs/Templates/Game
```

  - You only place it in the GameLayer so that draw order of the game components are between the background and the overlay components.
  - If you know what you are doing, you can place the game prefab however you want inside the hierarchy.



TreeStoryVN

Tag Untagged   Layer UI

Prefab

Rect Transform

	Pos X	Pos Y	Pos Z
	0	0	0
	Width	Height	
	100	100	

Anchors

Min	X	0.5	Y	0.5
Max	X	0.5	Y	0.5
Pivot	X	0.5	Y	0.5

Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

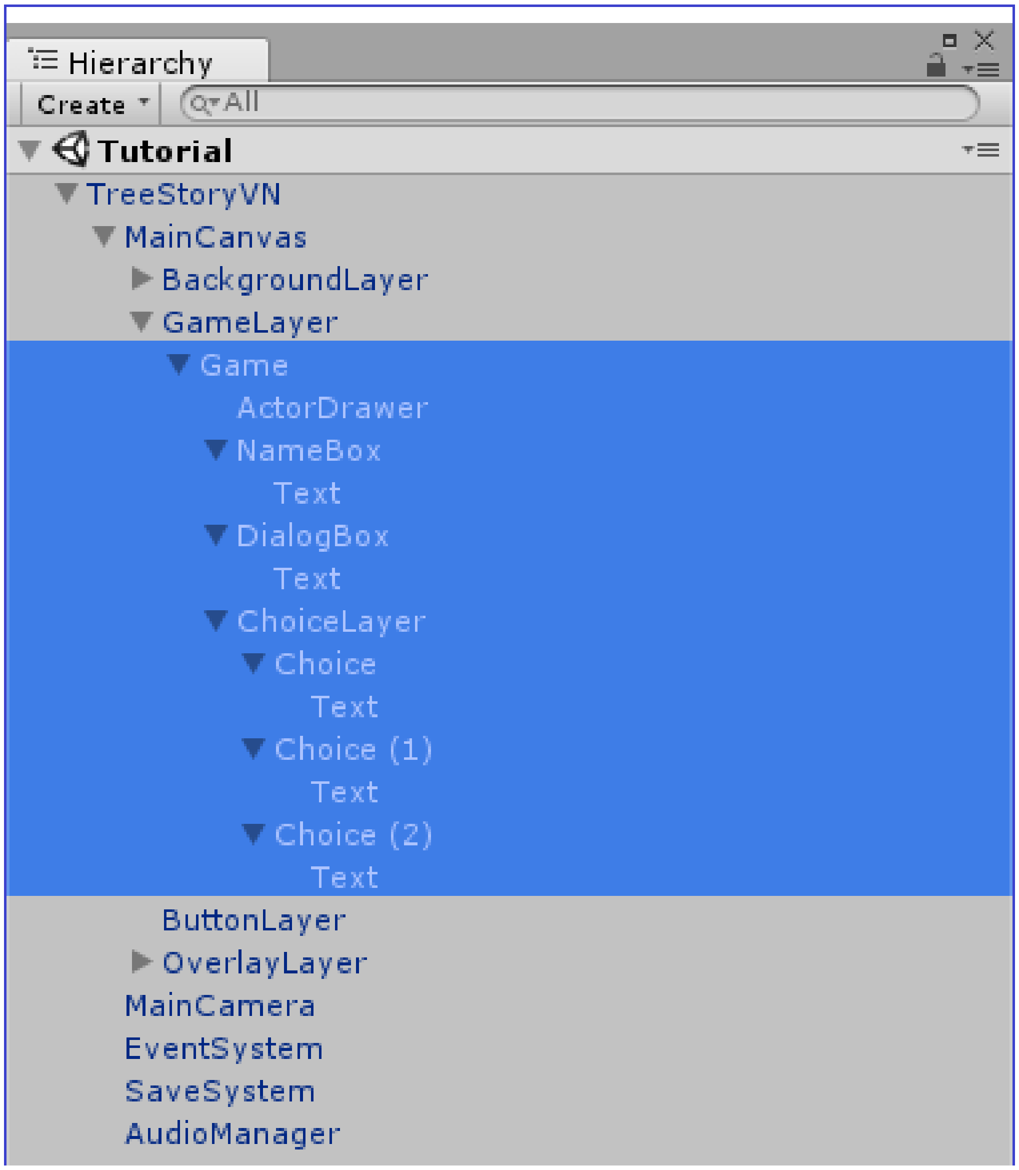
Tree Story VN (Script)

Script TreeStoryVN

Node Library None (Node Library)

Fig. 7.3: The TreeStoryVN GameObject Inspector

- Be careful about how Unity saves ordering for objects inserted between the objects of prefab instances inside the hierarchy.



```
Tutorial
  TreeStoryVN
    MainCanvas
      BackgroundLayer
      GameLayer
        Game
          ActorDrawer
          NameBox
            Text
          DialogBox
            Text
          ChoiceLayer
            Choice
              Text
            Choice (1)
              Text
            Choice (2)
              Text
      ButtonLayer
      OverlayLayer
    MainCamera
    EventSystem
    SaveSystem
    AudioManager
```

Fig. 7.4: The Game Template Prefab

- Customize each game object to your liking.
  - Details on how to do that are discussed previously in each prefab's sub-section.
- The color property of the source image component for the DialogBox, NameBox, and ChoiceLayer game objects are set to transparent by default in the game template prefab.
  - Change the image component's color to white (or any color that you prefer) to make its background visible.

Creating a Menu

- Menus in TreeStory are usually implemented by creating new scenes for each.

- Using this method, you can make the menu shown by calling it using "scene nodes" inside your node libraries.
- You can either make it as a standalone scene or as an overlay to your game depending on the "load scene mode" property of the scene node that calls it.
  - A standalone menu is shown by setting it to "single".
  - An overlay is shown by setting it to "additive".

Steps

- **Drag the "Menu" prefab into the "GameLayer" object in Unity's hierarchy window.**
  - Location:

    ```
    TreeStory/Prefabs/Templates/Menu
    ```

  - You only place it in the GameLayer so that draw order of the game components are between the background and the overlay components.
  - If you know what you are doing, you can place the menu prefab however you want inside the hierarchy.
    - Be careful about how Unity saves ordering for objects inserted between the objects of prefab instances inside the hierarchy.
- Customize each game object to your liking.
  - Details on how to do that are discussed previously in each prefab's sub-section.
- The color property of the source image component for the ChoiceLayer game objects are set to transparent by default in the menu template prefab.
  - Change the image component's color to white (or any color that you prefer) to make its background visible.

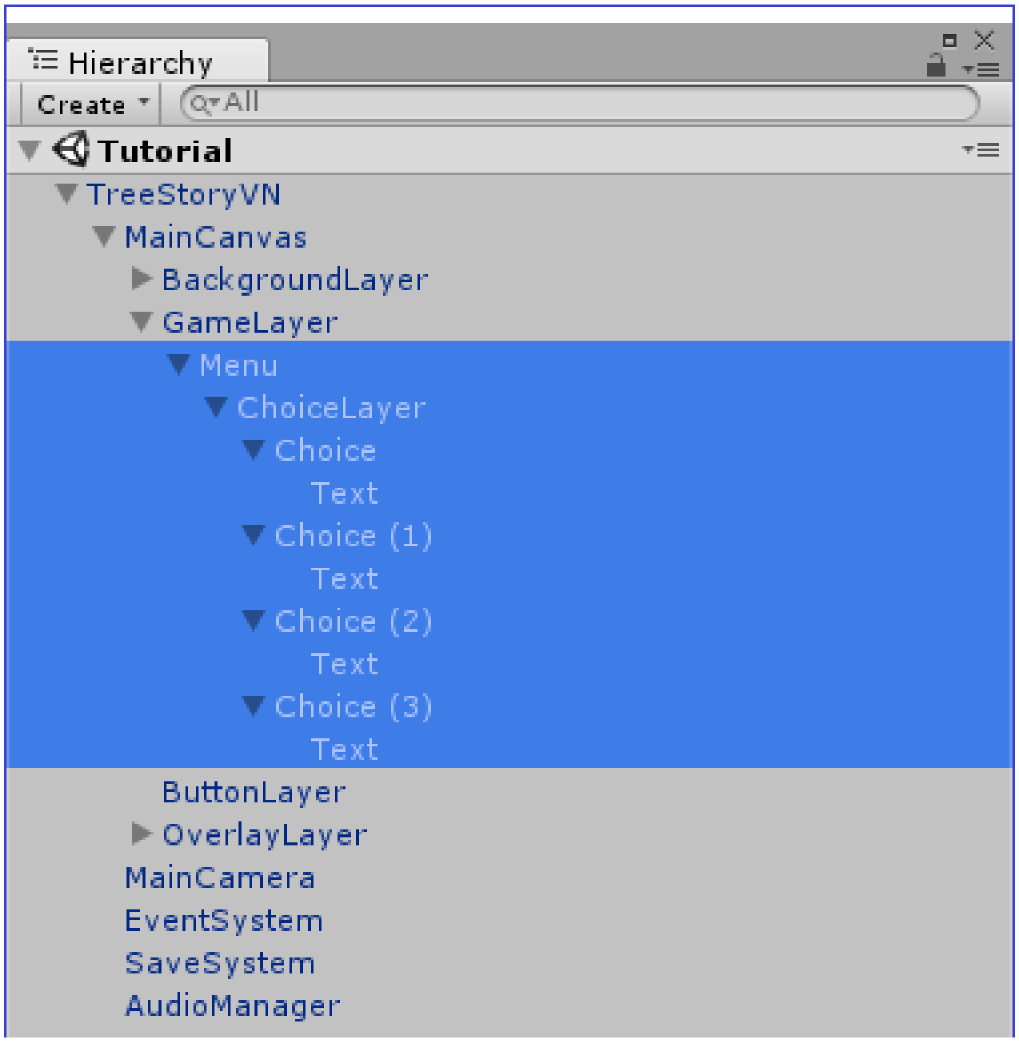


Fig. 7.5: The Menu Template Prefab

CHAPTER 8

Extending the Library

- Currently, you can only create a “visual novel” type of game if you use TreeStory exclusively.
- If you want to use TreeStory for a different game type, or if you just want to improve upon what is currently available, then you can extend the TreeStory library using your own scripts.
- TreeStory can be extended easily and anything that you add to it will automatically work with the provided editor.

Creating a Custom Game Data

Template

```
using System;
using UnityEngine;

// Add using statements here as necessary.

// You can also use your own namespace, just be careful of C#'s namespace rules.
namespace TreeStory
{
    [Serializable]
    public class /* CustomData */ : ScriptableObject
    {
        // Add variables for your class here.

        public void OnEnable()
        {
            hideFlags = HideFlags.HideInHierarchy;
        }

        // Add methods for your class here.
    }
}
```

Sample code

```
using System;
using UnityEngine;

namespace TreeStory
{
    [Serializable]
    public class LocationData : ScriptableObject
    {
        public Sprite Background;

        public void OnEnable()
        {
            hideFlags = HideFlags.HideInHierarchy;
        }
    }
}
```

Creating a Custom Library

- Libraries are basically just classes which store a list of game data.
- TreeStory uses these libraries to find game data and use it inside your game.
- As such, your custom game data is useless until you create a library to store it.

Template

```
using System;
using System.Collections.Generic;

// Add using statements here as necessary.

// You can also use your own namespace, just be careful of C#'s namespace rules.
namespace TreeStory
{
    [Serializable]
    // Add more attributes here.
    public class /* CustomLibrary */ : BaseLibrary
    {
#if UNITY_EDITOR
        public static List</* CustomLibrary */> AllLibraries = new List</*␣
↪CustomLibrary */>();
#endif

        // This is a list variable for the custom data that you have created in the␣
↪previous section.
        public List</* CustomData */> DataList = new List</* CustomData */> = new List
↪</* CustomData */>();
```

```
        // Add variables for your class here.

        // Add methods for your class here.
    }
}
```

Usable attributes

- **DataOf**
  - You can add this attribute to link this library's game data to the specified nodes.
  - These nodes will allow this library's game data to its data list property.
  - This attribute does the exact same function as the "**DataType**" attribute for the node classes.
    - You can use either, but they both exist in case you can't add attributes to one or the other.

Usable methods

- **public static void CreateDataDropdown**
  - Use this method to add a dropdown field which contains all libraries of the specified type.
  - This method is public and can be used for other classes, like the custom node editors.
  - **Parameters:**
    - **dataProperty (SerializedProperty)**
      - The serialized property of the data that you want to edit.
    - **allLibraries (IList)**
      - The list of all libraries for the type that you want to use.
      - Example:

        ```
        VideoNode.AllLibraries
        ```
    - **showNone (bool)**
      - Set this to true if you want to enable the "none" option for this field in the editor.
    - **tooltip (string)**
      - The tooltip that will be displayed when you hover your mouse over the property inside Unity's inspector window.
      - Leave this blank or set it to an empty string to not display a tooltip.
- **public static void AddMenuItems**
  - Use this method to add all libraries of the specified type to a Unity generic menu.
  - This method is public and can be used for other classes, like the custom nodes.
  - **Parameters:**
    - **menu (GenericMenu)**
      - The menu where you want to add the items to.

  * **dataProperty (SerializedProperty)**
    - The serialized property of the data that you want to edit.
  * **allLibraries (IList))**
    - The list of all libraries for the type that you want to use.
    - Example:

```
VideoNode.AllLibraries
```

  * **showNone (bool)**
    - Set this to true if you want to enable the "none" option for this field in the editor.
  * **pathPrefix (string)**
    - The path inside the context menu where the items will be added to.

Sample code

```
using System;
using System.Collections.Generic;

namespace TreeStory
{
    [Serializable]
    [DataOf(typeof(DialogNode))]
    [DataOf(typeof(ChoiceNode))]
    public class LocationLibrary : BaseLibrary
    {
#if UNITY_EDITOR
        public static List<LocationLibrary> AllLibraries = new List<LocationLibrary>
↪();
#endif

        public List<LocationData> DataList = new List<LocationData>();
    }
}
```

Notes

- You can add an icon for your custom library.
  - The name of the icon must be "<Class name>Icon" (without spaces).
    * Example:

```
CustomLibraryIcon
```

  - Place the icon inside the location:

```
<Any path including root>/Resources/Sprites/LibraryIcons/
```

Creating an Editor for Your Custom Library

Template

```
using UnityEditor;
using UnityEngine;

// Add using statements here as necessary.

// You can also use your own namespace, just be careful of C#'s namespace rules.
namespace TreeStory
{
    // The type that you place here is the class of the library that you created in␣
↪the previous section.
    [CustomEditor(typeof(/* CustomLibrary */)), CanEditMultipleObjects]
    public class /* CustomLibraryEditor */ : BaseLibraryEditor
    {
        // Add variables for your class here.

        // Add methods for your class here.
    }
}
```

Overridable methods

- **public virtual void OnEnable**
  - The usual OnEnable method of Unity.
  - If you override this method, make sure to call the “**base.OnEnable()**” method inside it.
- **protected virtual float ReorderableListHandler**
  - The method used to process data while drawing the reorderable list.
  - It is also used to create additional editor components beside the element’s data name.
  - **Parameters:**
    - **rect (Rect)**
      - The rect of the current element inside the reorderable list.
    - **index (int)**
      - The index of the current element inside the reorderable list.
    - **serializedElement (SerializedObject)**
      - The serialized object of the current reorderable object item.
  - **Returns**
    - The total width of all extra properties.
- **protected virtual bool CustomElementMenuItems**
  - Adds menu items on each element’s context menu.
  - **Parameters:**
    - **menu (GenericMenu)**

        - The generic menu for each element.
      - **serializedElement (SerializedObject)**
        - The serialized object of the current reorderable object item.
  - **Returns:**
    - True if there are custom menu items, false otherwise.
- **protected virtual void OnElementInspectorGUI**
  - The method used to render additional inspector elements when a reorderable list element has been selected.
  - Inspector elements that are controlled by this method will appear below the library data list.
  - Do not use the "OnInspectorGUI" method, unless you know what you are doing, as it contains custom TreeStory scripts.
  - Instead use this method like you would the OnInspectorGUI method.
  - **Parameters:**
    - **serializedElement (SerializedObject)**
      - The serialized object of the current reorderable object item.
- **protected virtual void CreateSearchBar**
  - Create a search bar for the library elements above the inspector window.
- **public virtual void OnSearchChanged**
  - Callback method that is called whenever the current search has changed.
- **protected virtual bool IsSearching**
  - Check if a search is being performed.
  - **Returns:**
    - True if a search is being performed, false otherwise.
- **protected virtual bool IsElementSearched**
  - Check if the specified element is part of the search results.
  - **Parameters:**
    - **element (ScriptableObject)**
      - The element to check.
  - **Returns:**
    - True if the element is part of the search results, false otherwise.

Usable methods

- **protected string EnsureUniqueElementName**
  - Ensures that the element's name is unique within the library by appending indices to the end of the name.
  - **Parameters:**
    - **elementName (string)**

            · The name of the element to check for name uniqueness.

    - **Returns:**

        * The unique name of the element.

- **protected ScriptableObject AddData**

    - Adds a new TreeStory data to the library.

    - **Parameters:**

        * **typeString (string)**

            · The type of data to add without the word "Data". (e.g. For "BaseData" pass only the string "Base".)

    - **Returns:**

        * The newly created data.

- **protected void DeleteData**

    - Deletes a TreeStory data from the library.

    - **Parameters:**

        * **index (int)**

            · The index of the data to be deleted.

- You can also use all usable methods of the custom library from the previous section since they are static.

Sample code

```
using UnityEditor;
using UnityEngine;

namespace TreeStory
{
    [CustomEditor(typeof(LocationLibrary)), CanEditMultipleObjects]
    public class LocationLibraryEditor : BaseLibraryEditor
    {
        public override void OnEnable()
        {
            base.OnEnable();

            dataRList.elementHeight = 72.0f;
        }

        protected override float ReorderableListHandler(Rect rect, int index,␣
↪SerializedObject serializedElement)
        {
            Rect spriteRect = rect;
            spriteRect.height = 68.0f;
            spriteRect.width = 68.0f;
            spriteRect.x = rect.xMax - spriteRect.width;

            var backgroundProperty = serializedElement.FindProperty("Background");
            backgroundProperty.objectReferenceValue = EditorGUI.
↪ObjectField(spriteRect, backgroundProperty.objectReferenceValue, typeof(Sprite),␣
↪false);
```

```
            return spriteRect.width;
        }
    }
}
```

Creating a Custom Node

Template

```
using System;

// Add using statements here as necessary.

#if UNITY_EDITOR
using UnityEditor;  // Only add this line if necessary.
using UnityEngine;
#endif

// You can also use your own namespace, just be careful of C#'s namespace rules.
namespace TreeStory
{
    [Serializable]
    // Add more attributes here.
    public class /* CustomNode */ : BaseNode
    {
#if UNITY_EDITOR
        static Texture2D nodeIconTexture;
#endif

        // Add variables for your class here.

#if UNITY_EDITOR
        public override void OnEnable()
        {
            base.OnEnable();
            Color = /* Node's background color */
        }

        public override void DrawPreviewText(int nodeIndex)
        {
            DrawNodeIcon(ref nodeIconTexture, textStyle.normal.textColor);

            float textOffset = 0.0f;
            if (nodeIconTexture != null)
                textOffset = nodeIconWidth + EditorGUIUtility.standardVerticalSpacing;

            base.DrawPreviewText(nodeIndex);

            // Add scripts to render your custom node's preview here.
        }

        // Add method overrides for your class here.
#endif
```

```
        // Add regular methods for your class here.

        // To be able to use the 2 Accept methods below, you must first create␣
↪extension methods for each.

        // Extend the Accept method of the "INodeVisitor" interface to be able to␣
↪override this method.
        public override void Accept(INodeVisitor visitor)
        {
            visitor.Visit(this);
        }

        // Extend the Accept method of the "INodeTester" interface to be able to␣
↪override this method.
        public override bool Accept(INodeTester tester)
        {
            return tester.Test(this);
        }
    }
}
```

Usable attributes

- **NodeSort**
  - You can add this attribute to control the order where this node appears in menus inside the editor.
- **DataTypes**
  - You can add this attribute to link TreeStory game data types to your node.
  - These data types are used for the data list property of this node.
  - This attribute does the exact same function as the "**DataOf**" attribute for the library classes.
    - You can use either, but they both exist in case you can't add attributes to one or the other.
- **ParentNodes**
  - You can add this attribute to enable a selected type of parent node to accept multiple amounts of this node as its children.

Overridable methods

- **public virtual void OnEnable**
  - The usual OnEnable method of Unity.
  - If you override this method, make sure to call the "**base.OnEnable()**" method inside it.
- **public virtual void InitializeNodeWithParent**
  - Initializes the current node with its parent node.
  - Example usage: Copy data from the parent node to this node.
  - **Parameters:**
    - **parentNode (BaseNode)**
      - The node to set as this node's parent.

- **public virtual void DrawBackground**
  - Draws the background texture of the node.
- **public virtual void DrawPreviewText**
  - Draws the preview text of the node.
  - **Parameters:**
    - **nodeIndex (int)**
      - The index of the node inside the node library.
- **public virtual bool CustomMenuItems**
  - Adds menu items on the node's context menu.
  - **Parameters:**
    - **menu (GenericMenu)**
      - The context menu of the node.
  - **Returns:**
    - True if there are custom menu items, false otherwise.

Usable methods

- **protected void DrawNodeIcon**
  - Draws the node icon texture at the left side of the node.
  - This will store the image in the nodeIconTexture variable that is passed as a parameter.
  - **Parameters:**
    - **nodeIconTexture (ref Texture2D)**
      - The variable that stores the node icon texture.
      - Pass the static nodeIconTexture variable here.
    - **iconTint (Color)**
      - The color multiplier for the node icon texture.
- **public string FormatString**
  - Adds rich text xml tags to a string.
  - **Parameters:**
    - **originalString (string)**
      - The string to format.
    - **isBold (bool)**
      - Set this to true to make the text bold.
      - You can leave this blank if you don't need to make the text bold AND if you don't need the last 2 parameters.
    - **isItalicized (bool)**
      - Set this to true to make the text italicized.

        · You can leave this blank if you don't need to make the text italicized AND if you don't need the last parameter.

    * **size (int)**

        · The size of the text.

        · Leave this blank or set this to -1 to keep the default text size.

  – **Returns:**

    * The rich text formatted string.

• **protected void CreatePreviewLabelField**

  – Creates a label which aligns the preview text using the center anchor if it fits the node area.

  – If the text does not fit the node area, then it will be aligned at the left of the specified anchor.

  – **Parameters:**

    * **centerTextAnchor (TextAnchor)**

        · The vertical position of the text.

        · **Always set its horizontal position to center.**

          Example: If you want to set the text's vertical position to the top, then use the "UpperCenter" value.

    * **previewText (string)**

        · The preview text to output.

    * **textOffset (float)**

        · The horizontal offset from the left used to render the text.

• **public string[] DataToStringParams**

  – Converts data from the node's data list property to parameters used for the string.Format() method.

  – **Parameters:**

    * **defaultColor (Color?)**

        · The default color used for the text in-game.

        · Set this to null if you don't want to add color tags to the string parameter.

        · This value is used to prevent adding the color xml tag unnecessarily.

• You can also use all usable methods of the custom library from the previous section since they are static.

Sample code

```
using System;

#if UNITY_EDITOR
using UnityEditor;
using UnityEngine;
#endif

namespace TreeStory
{
    [Serializable]
    [NodeSort(9499)]
    public class LocationNode : BaseNode
    {
#if UNITY_EDITOR
        static Texture2D nodeIconTexture;
#endif

        public enum TransitionTypes { NoTransition, Fade }

        public LocationData Location;
        public TransitionTypes Transition = TransitionTypes.Fade;
        public bool ExitActors = true;

#if UNITY_EDITOR
        public override void OnEnable()
        {
            base.OnEnable();
            Color = new Color(0.85f, 0.7f, 0.55f);
        }

        public override void DrawPreviewText(int nodeIndex)
        {
            DrawNodeIcon(ref nodeIconTexture, textStyle.normal.textColor);

            float textOffset = 0.0f;
            if (nodeIconTexture != null)
                textOffset = nodeIconWidth + EditorGUIUtility.standardVerticalSpacing;

            base.DrawPreviewText(nodeIndex);

            string previewText = "";
            if (Location != null)
                previewText = Location.name;

            string headerText = ObjectNames.NicifyVariableName(Transition.ToString());
            headerText = FormatString(headerText, true);

            CreatePreviewLabelField(TextAnchor.UpperCenter, headerText, textOffset);
            CreatePreviewLabelField(TextAnchor.LowerCenter, previewText, textOffset);
        }

        public override bool CustomMenuItems(GenericMenu menu)
        {
            SerializedObject serializedObjects = new SerializedObject(Selection.
↪objects);
            SerializedProperty locationProperty = serializedObjects.FindProperty(
↪"Location");

            BaseLibrary.AddMenuItems(menu, locationProperty, LocationLibrary.
↪AllLibraries, true, "Set Location/");

            return true;
        }
#endif

        public override void Accept(INodeVisitor visitor)
        {
            visitor.Visit(this);
        }

        public override bool Accept(INodeTester tester)
        {
            return tester.Test(this);
        }
    }
}
```

Sample method extensions

```
namespace TreeStory
{
    public interface INodeProcessor : INodeVisitor
    {
        void Visit(ActorNode node);
        void Visit(LocationNode node);
    }

    public static class NodeProcessorExtension
    {
        public static void Visit(this INodeVisitor visitor, ActorNode node)
        {
            (visitor as INodeProcessor).Visit(node);
        }

        public static void Visit(this INodeVisitor visitor, LocationNode node)
        {
            (visitor as INodeProcessor).Visit(node);
        }
    }
}
```

```
namespace TreeStory
{
    public interface INodeFinishTester : INodeTester
    {
        bool Test(ActorNode node);
        bool Test(LocationNode node);
    }

    public static class NodeFinishTesterExtension
    {
        public static bool Test(this INodeTester tester, ActorNode node)
        {
            return (tester as INodeFinishTester).Test(node);
        }

        public static bool Test(this INodeTester tester, LocationNode node)
        {
            return (tester as INodeFinishTester).Test(node);
        }
    }
}
```

Notes

- You can add an icon for your custom node.
  - The name of the icon must be "<Class name>Icon" (without spaces).
    - Example:

      ```
      CustomNodeIcon
      ```
  - Place the icon inside the location:

    ```
    <Any path including root>/Resources/Sprites/NodeIcons/
    ```
  - **Tip:** Make your icon image's color white to enable changing its color through script.

Creating an Editor for Your Custom Node

Template

```
using UnityEditor;
using UnityEngine;

// Add using statements here as necessary.

// You can also use your own namespace, just be careful of C#'s namespace rules.
namespace TreeStory
{
    // The type that you place here is the class of the node that you created in the␣
↪previous section.
    [CustomEditor(typeof(/* CustomNode */)), CanEditMultipleObjects]
    public class /* CustomNodeEditor */ : BaseNodeEditor
    {
        // Add variables for your class here.

        // Add methods for your class here.
    }
}
```

Overridable methods

- **public virtual void OnEnable**
  - The usual OnEnable method of Unity.
  - If you override this method, make sure to call the "**base.OnEnable()**" method inside it.
- **protected virtual void OnDerivedInspectorGUI**
  - The method used to render inspector elements.
  - Do not use the "OnInspectorGUI" method, unless you know what you are doing, as it contains custom TreeStory scripts.
  - Instead use this method like you would the OnInspectorGUI method.

Usable methods

- **protected ReorderableList CreateDataRList**
  - Creates a reorderable list for a node's data list.
  - **Parameters:**
    - **nodeType (Type)**
      - The type of the node which contains the data list.
    - **dataListProperty (SerializedProperty)**
      - The serialized property of the data list.
    - **headerText (string)**
      - The text displayed on the reorderable list's header.
      - Keep this empty to use the default value for the header text.
  - **Returns:**
    - The reorderable list for a node's data list.
- **protected void CreateWaitToFinishField**
  - Use this method inside the "OnDerivedInspectorGUI" method (preferably at the start of the method) if your node uses the "wait to finish" property.
- **protected static void CreateTextAreaField**
  - Use this method inside the "OnDerivedInspectorGUI" method to add a text area field with a prefix label to the editor.
  - **Parameters:**
    - **nodeTextProperty (SerializedProperty)**
      - The serialized property of the text that you want to edit.
    - **tooltip (string)**
      - The tooltip that will be displayed when you hover your mouse over the property inside Unity's inspector window.
      - Leave this blank or set it to an empty string to not display a tooltip.
- You can also use all usable methods of the custom library from the previous section since they are static.

Sample code

```
using UnityEditor;
using UnityEngine;

namespace TreeStory
{
    [CustomEditor(typeof(LocationNode)), CanEditMultipleObjects]
    public class LocationNodeEditor : BaseNodeEditor
    {
        SerializedProperty locationProperty;
        SerializedProperty transitionProperty;
        SerializedProperty exitActorsProperty;
        public override void OnEnable()
        {
            base.OnEnable();

            locationProperty = serializedObject.FindProperty("Location");
            transitionProperty = serializedObject.FindProperty("Transition");
            exitActorsProperty = serializedObject.FindProperty("ExitActors");
        }

        protected override void OnDerivedInspectorGUI()
        {
            CreateWaitToFinishField();

            BaseLibrary.CreateDataDropdown(locationProperty, LocationLibrary.
↪AllLibraries, true, "The location associated with this node.");

            GUIContent transitionLabel = new GUIContent(transitionProperty.
↪displayName, "The transition effect used to transfer locations.");
            EditorGUILayout.PropertyField(transitionProperty, transitionLabel);

            GUIContent exitActorsLabel = new GUIContent(exitActorsProperty.
↪displayName, "If checked, all actors currently in-game will exit.");
            EditorGUILayout.PropertyField(exitActorsProperty, exitActorsLabel);
        }
    }
}
```

Creating a Custom Event Handler

Template

```
using UnityEngine;

// Add using statements here as necessary.

// You can also use your own namespace, just be careful of C#'s namespace rules.
namespace TreeStory
{
    public class /* CustomEventHandler */ : EventHandler
    {
        // Add variables for your class here.

        // The virtual keyword is optional but recommended in case you want to extend␣
↪it in derived classes.
        public virtual void Update()
        {
            // Place your custom script here on how the "**IsTriggered**" variable␣
↪becomes true.
            // It automatically returns to false after one frame.
        }

        // Add methods for your class here.
    }
}
```

Overridable methods

- **public virtual void Awake**
  - The usual Awake method of Unity.
  - Its base method only initializes the "**IsTriggered**" variable to false.
- **public virtual void LateUpdate**
  - The usual LateUpdate method of Unity.
  - Its base method only resets the "**IsTriggered**" variable to false.

Sample code

```
using UnityEngine;

namespace TreeStory
{
    public class KeyboardHandler : EventHandler
    {
        [Tooltip("The keyboard key that will trigger the event.")]
        public KeyCode KeyCode = KeyCode.Escape;

        public virtual void Update()
        {
            if (Input.GetKeyUp(KeyCode))
                IsTriggered = true;
        }
    }
}
```

Creating an Editor for Your Custom Event Handler

Template

```
using UnityEditor;

// Add using statements here as necessary.

// You can also use your own namespace, just be careful of C#'s namespace rules.
namespace TreeStory
{
    // The type that you place here is the class of the event handler that you␣
↪created in the previous section.
    [CustomEditor(typeof(/* CustomEventHandler */)), CanEditMultipleObjects]
    public class /* CustomEventHandlerEditor */ : EventHandlerEditor
    {
        // Add variables for your class here.

        // Add methods for your class here.
    }
}
```

Overridable methods

- **public virtual void OnEnable**
  - The usual OnEnable method of Unity.
  - If you override this method, make sure to call the “**base.OnEnable()**” method inside it.

Sample code

```
using UnityEditor;

namespace TreeStory
{
    [CustomEditor(typeof(KeyboardHandler)), CanEditMultipleObjects]
    public class KeyboardHandlerEditor : EventHandlerEditor
    {
        SerializedProperty keyCodeProperty;

        public override void OnEnable()
        {
            keyCodeProperty = serializedObject.FindProperty("KeyCode");

            base.OnEnable();
        }

        public override void OnInspectorGUI()
        {
            serializedObject.Update();

            EditorGUILayout.PropertyField(keyCodeProperty);

            EditorGUILayout.Space();

            nodeLibrariesRList.DoLayoutList();

            serializedObject.ApplyModifiedProperties();
        }
    }
}
```