MONTE BEARO

Discourse

Installation

Discourse requires TextMeshPro is installed in the project. Please navigate to Window > PackageManager and ensure TextMeshPro is installed. If you plan to import the Example Content, please also import the Essential Resources, via Window > TextMeshPro > Import TMP Essential Resources. This should be done before opening any of the Demo scenes.

Example Content:

During the Asset Store import process, un-check the Example Content folder if you do not want to import the Demo content. However, we do recommend you familiarise yourself with the package using this content, before starting from scratch.

If you plan to view the Example Content, ensure the Reference and Preference Database assets nested within the Example Content folder are the only Database assets in the project. Discourse may attempt to create default Database assets in the installation folder, before Unity has fully imported the Example Content.

New Input System:

Discourse ships with some components that advance the graph by using the default Unity Input system. These components can be found in the Extensions folder, within the Discourse installation folder. If you are using the new Input system, you should either not import these components, or delete them from the project after installation and add corresponding components that use the new Input API.

Introduction

Thanks for using Discourse! The tool has been written with ease of use, flexibility and quality of life in mind, so if you have any issues or suggestions, please feel free to get in touch at **support@montebearo.co.uk**. We'll endeavour to get back to you in a timely manner and answer any questions you have.

This document serves as a quick introduction to the general workflow in using Discourse, and a brief overview of the general concepts involved. Discourse does require a basic understanding of Unity and as such this document assumes you are familiar with prefabs, assets, components, UI and other Unity fundamentals.

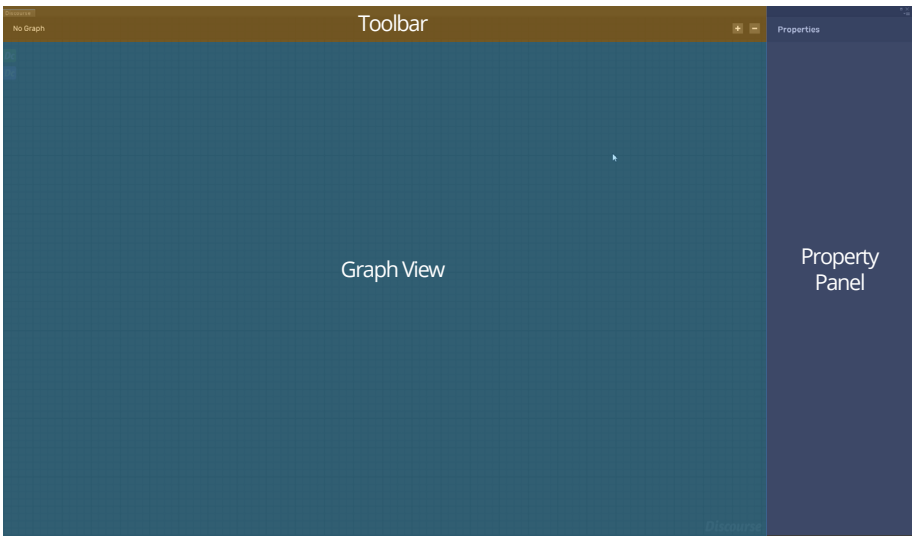
Databases

After importing the asset package, you may notice some logs to the console about missing a **ReferenceDatabase** and **PreferenceDatabase** assets - this is nothing to worry about. These are assets required by Discourse, and will be automatically created if they aren't found. The default location of these assets is Montebearo.Discourse/Resources. You may move these to any other /Resources folder if you wish.

The Reference and Preference Databases are assets that between them will point to your actors, localised text, global variables and much more. In code, the methods and properties on these assets can be accessed through the static classes References and Preferences correspondingly.

Discourse Window

To get started crafting your stories, open the Discourse main window by navigating to **Tools > Discourse > Main Window**. The first time you start using Discourse, the References and Preferences databases will be created in the Resources subfolder. Once the window is open, you can select these by clicking on the green and purple icons located on the left hand side of the window.



The **Toolbar** is located at the top of the main window, and lets you view and edit the current Event's **Display Name**. This is a string you can use for custom Event displays if you need to display the name of a story sequence to the player - for instance in a Quest system. The Display Name is separate from the Graph's name, which should only use characters safe for file names.
Also on the toolbar are buttons to add Nodes, and zoom the graph.

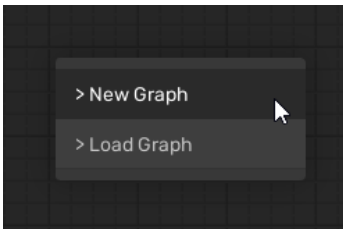
The **Graph View** is where your Event's nodes will be laid out. You can pan this space with the middle mouse button or by holding Alt and Left Mouse Button dragging. It can be zoomed with the scroll wheel, or using the buttons on the toolbar.

The **Property Panel** acts as an inspector for the selected node, drawing each of its fields and allowing you to edit any settings related to that Action. The width of this panel can be adjusted by dragging on the left edge, bordering the graph view.

Graphs & Events

Graphs and **Events** (DiscourseEvents) form the bread and butter of Discourse. Graphs contain Events, which you will run to play your cutscenes and dialogue at runtime. These Events are composed of several **Actions** (DiscourseActions) strung together, which are represented in the Graph as Nodes. Each node houses its own Action, which will fire in sequence until an Exit node is reached.

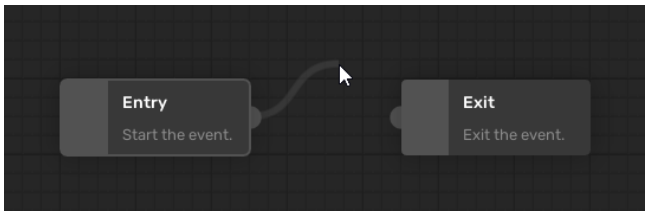
You can create a new Graph by right clicking the Graph View and selecting "New Graph". This will open a file dialogue to let you choose the name of the Graph and the location to save it. Note this should be a project-relative directory (i.e somewhere in the Assets directory).



All Events start with an **Entry** node and an **Exit** node. The Entry node is the point at which the Actions tree starts, so nothing will run if it isn't connected to anything. You cannot delete the Entry node. Exit nodes are points at which the Event can quit, and optionally transition to a provided scene.

Nodes

Nodes are containers for the Actions that will compose your story events. Nodes have '**Hooks**' and '**Links**' to connect between one another - both of which are types of '**Connection Points**'. Hooks are the circular pins on the left hand side of the Node's body, and gather incoming connections, whereas Links are on the right hand side and show outgoing connections. Links can only point to one other node, but a Hook can gather any number of incoming connections; forming a non-linear tree structure. Similarly, Nodes can only have one Hook, but as many Links as needed.

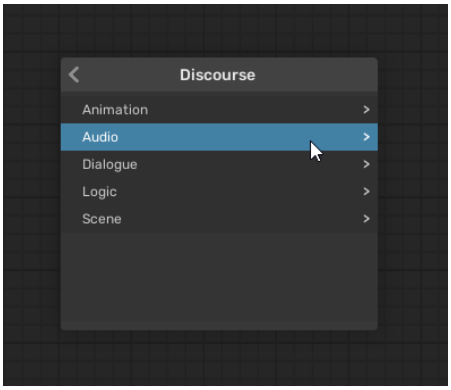


To make a connection between nodes, left click the outgoing Link, then click the target Hook, or first click the Hook and then the Link. You do not need to click and drag to form this connection (meaning you can still pan the graph with an open connection).

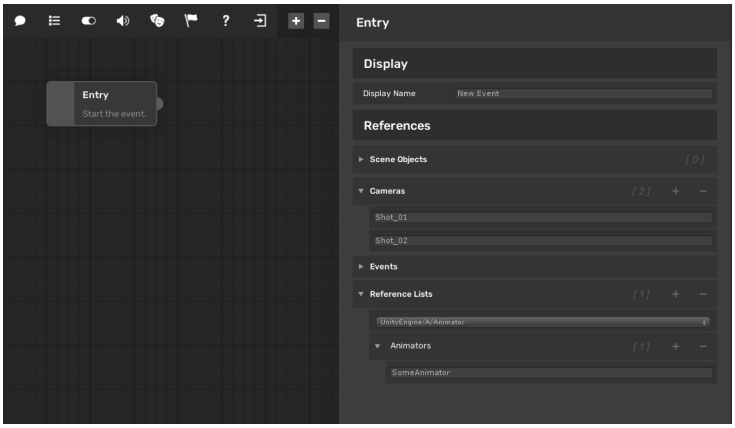
You can **break connections** between nodes by right clicking any of its Connection Points. Breaking a link will disconnect only that connection, whereas breaking on a Hook will disconnect all incoming connections.

To add nodes, either click any of the quick-access buttons on the Toolbar, or press spacebar to open the Search widget. This widget can be navigated either with the mouse, or with the arrow keys (left/right to go back/forward categories, up/down to scroll the list, and Enter/Return to confirm the selection and add the node). This list shows every Node found in the project, including custom nodes.

To delete nodes, select them and press delete or right click and choose Delete from the context menu. **To duplicate nodes**, select them, right click and choose Duplicate.



Entry Node



The Entry Node marks the start of the Event.

- **Display Name:** Change the Display Name of the Event

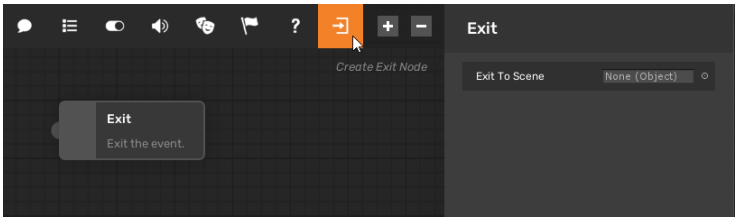
- **Scene Objects:** Define a list of scene references (Transforms) you can reference on other nodes. These must be referenced by a SceneObjectReferenceHandler on your EventBehaviour at runtime.

- **Cameras:** Define a list of cameras which you can switch between over the course of the graph. These must be referenced by a Camera Transition Handler on your EventBehaviour at runtime.

- **Events**: Define a list of UnityEvents which you can invoke via nodes in the graph. These must be referenced by a corresponding UnityEventReferenceHandler on your EventBehaviour. These will reference UnityEventBehaviours, on which you can add any listeners.

- **Reference Lists:** Similar to Scene Objects, reference any object inheriting UnityEngine.Object. These must be referenced by a corresponding ISceneReferenceHandler of the same type.

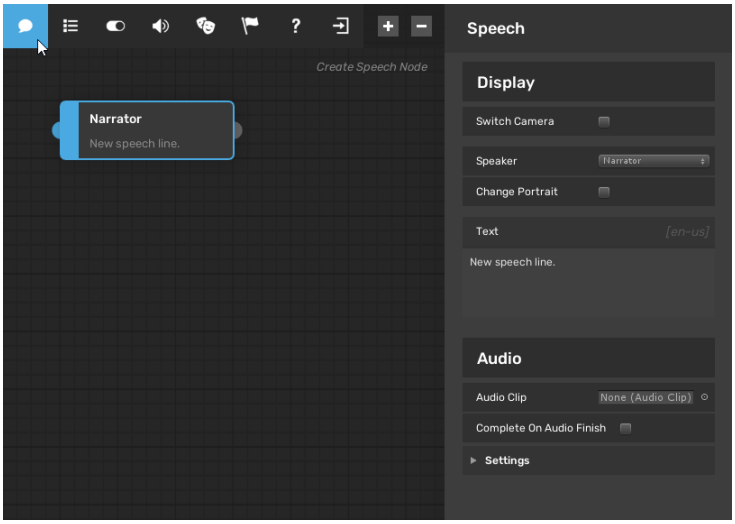
Exit Node



The Exit node will leave an open Event.

- **Exit to Scene:** Reference a scene to switch to on leaving the Event at this point. Scene Transitions (such as dip-to-black) are handled via a prefab on the Preference database.

Speech Node



The Speech node can show subtitles and play Audio on an Actor.

- **Switch Camera:** Enable to switch to the Target Camera when this action is run. The list of cameras must be defined on the Entry node.

- **Speaker:** The Actor who should speak the line (using their AudioHandler), or optionally a Narrator (with custom styling that can be set on the Preferences Database).

- **Change Portrait:** Enable to switch the displayed portrait to another defined on the Actor's Info asset. If disabled, will leave the portrait unchanged until text is displayed for a different Actor.

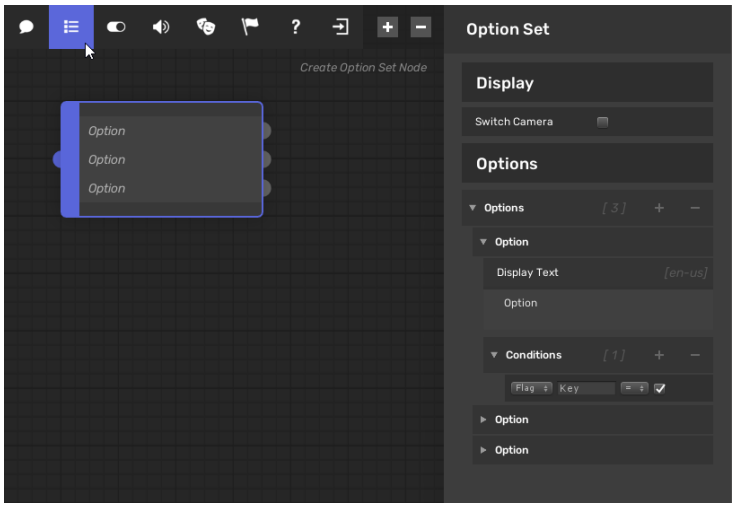
- **Text:** The text to display as subtitles (defined in your active Source Language).

- **Audio Clip:** The audio to play on the Actor's AudioHandler.

- **Complete on Audio FInish:** Enable to automatically advance the dialogue when the Audio finishes.

- **Settings:** Tweak settings for this audio clip.

OptionSet Node



Display a list of choices to the Player.

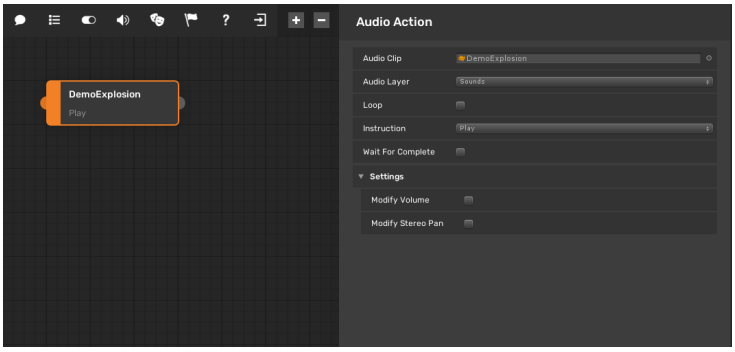
- **Switch Camera:** See explanation on 'Speech Node'

- **Options:** A list of options that should be displayed to the player using an OptionDisplay component.

- **Option Summary:** The text to display to the player

- **Conditions List:** A list of conditions that must be met in order for the option to be interactable.

Audio Action Node



Play Audio on the EventBehaviour's AudioHandler.

- **Audio Clip:** The clip to play.

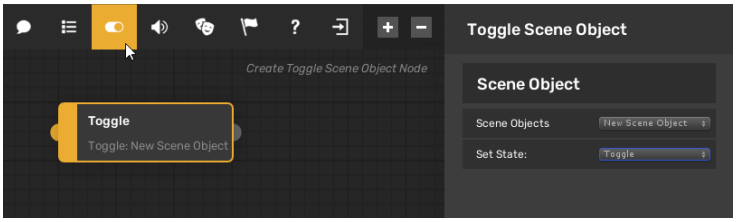
- **Audio Layer:** The layer to play the sound on. Configure these on the AudioSettingsProfile referenced on the PreferenceDatabase.

- **Instruction:** Play, PlayOneShot, Pause, Resume, Stop. Please see the Unity AudioSource documentation for information on these methods.

- **Wait for Complete:** Do not advance the graph until the instruction has finished.

- **Settings:** Tweak the audio source's settings.

Toggle Scene Object Node

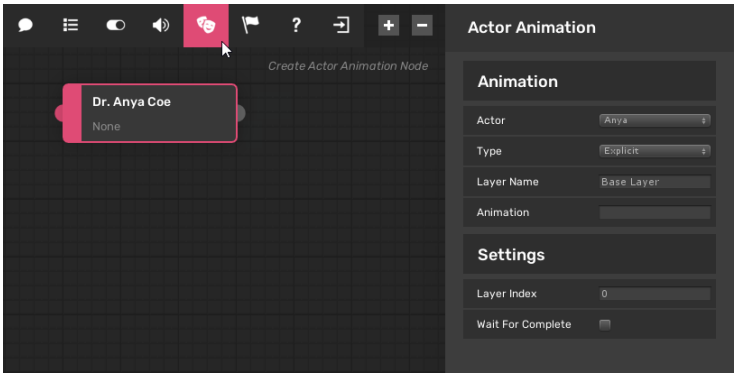


Toggle, Enable, or Disable a specified Scene Object.

- **Scene Object:** The object to toggle; pick from a list defined on the Entry node.

- **Set State:** Enable, Disable or Toggle the object's enabled state (toggle: invert the current state).

Actor Animation Node



Play an Animation through an Actor's Animator component.

- **Actor:** The Actor to animate.

- **Type:** Explicit; specify the name of a state in the AnimatorController to immediately switch to, Set Trigger/Float/Bool/Integer; set the value of a parameter on the AnimatorController.

- **Layer Name:** The name of the layer the state exists on.

- **Animation:** The name of the state to play.

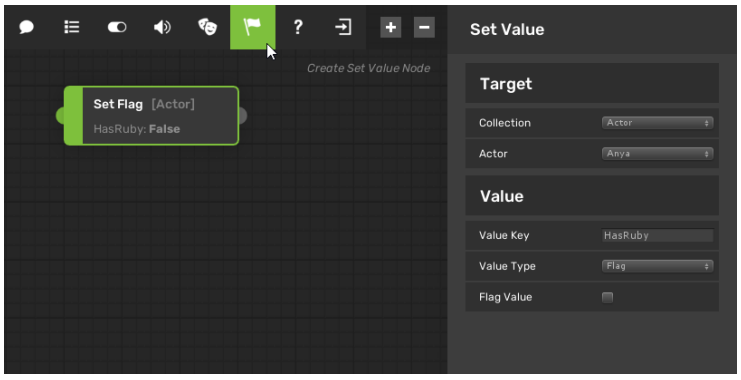
- **Parameter Name:** The name of the Trigger/Float/Bool/Integer to set the value of.

- { } **Value:** The value to set the parameter to.

- **Layer Index:** The index of the layer the state transitions should occur on (required for Wait for Complete).

- **Wait for Complete:** Don't advance the graph until the state has transitioned and completed.

Set Value Node



Set the value of a Global or Actor variable.

- **Collection:** Set Globally or on an Actor?

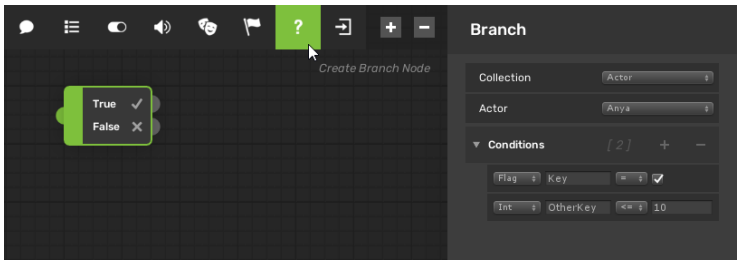
- **Actor:** The target Actor.

- **Value Key:** The key of the Variable in the Collection. (E.g. "Cash" / "HasStaffOfPower").

- **Value Type:** The type of variable to set; Flag/Float/Integer/String.

- { } **Value:** The value to set the variable to.

Branch Node



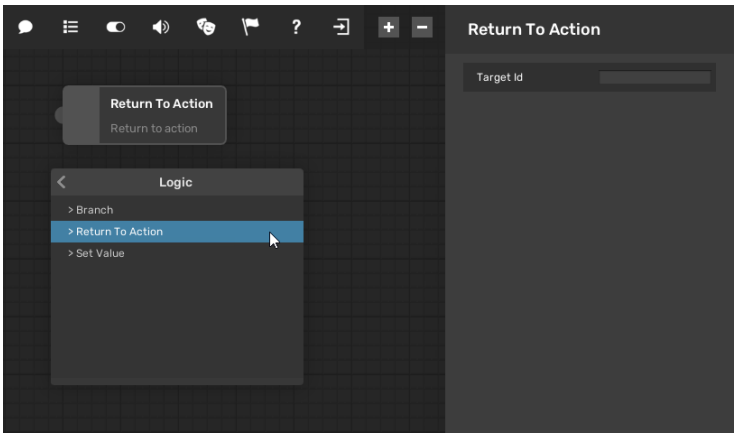
Branch the graph depending on a list of Conditions.

- **Collection:** Evaluate Globally or on an Actor?

- **Actor:** The target Actor.

- **Conditions:** Define a list of conditions to meet; Float/Int/Flag/String. Ints and Floats can use < <= = >= > operators.

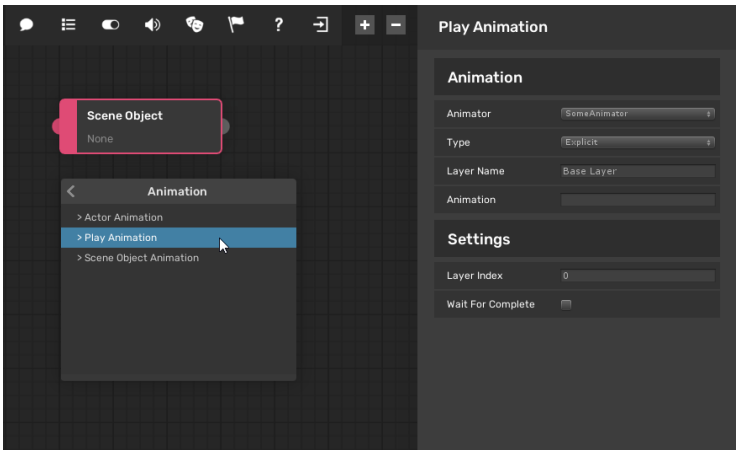
Return to Action Node



Rewind the graph until you return to a previously-visited Action.

- **Target Id:** The Id of the Action to return to. You can get this from a node by right clicking it and selecting “Copy Action Id”.

Play Animation Node

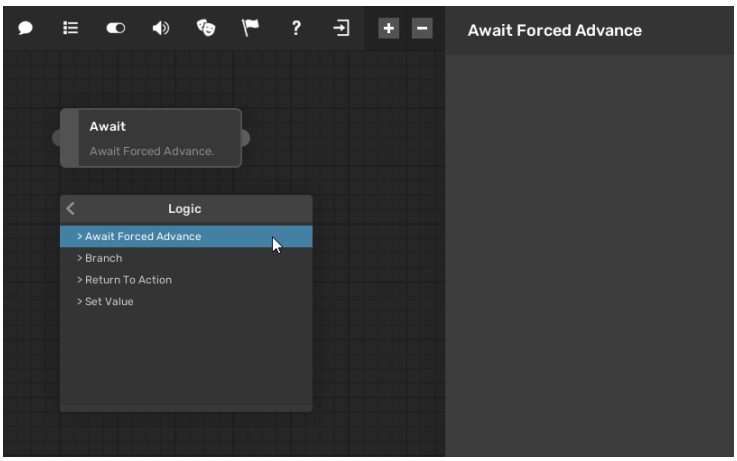


Play an Animation on a referenced Animator component.

- **Animator:** The Animator component on which to play the animation (runtime reference must have an Animator attached to it).

- See Actor Animation Node for other settings.

Await Forced Advance



Temporarily stop the EventBehaviour from listening for EventAdvancer Advance events, and wait until it is called directly on the EventBehaviour with ForceInvokeAdvance(). This is useful if for example you are using the MouseClickAdvancer, but need to display some interactable UI that would otherwise trigger the Advancement of the graph.

Playing an Event

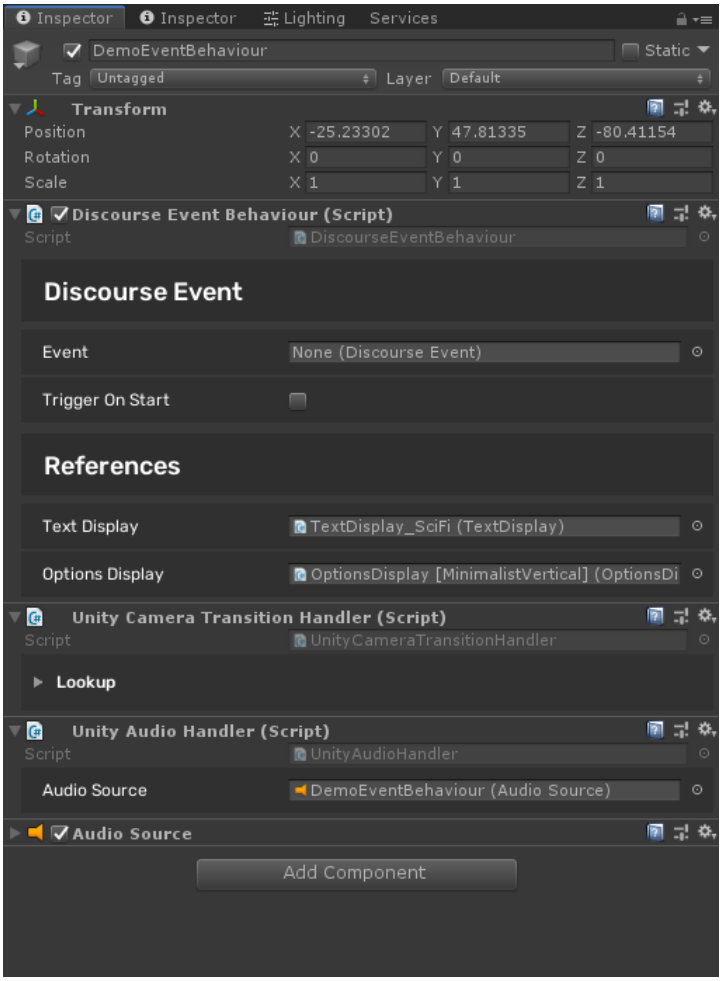
Events are triggered and displayed at runtime with a DiscourseEventBehaviour component. You must reference the Event you wish to play on this component (Note: not the graph, to drag-and-drop this asset, click the foldout button on the Graph asset and drag the orange Event asset into the Event field).

The EventBehaviour requires a TextDisplay component to display subtitles, actor names and portraits, and an OptionsDisplay component to display choices to the player. For each of these, you can use the template prefabs provided, or swap in your own.

If your Event references Scene Objects or Cameras, you will need to add a SceneObjectHandler and CameraTransitionHandler respectively. Once these components are added, you can begin linking the list to scene references by "Update Reference Handlers" button. You will need to do this whenever the list of Objects or Cameras is changed on the Event.

You can make an Event play immediately after the Scene loads by enabling "Trigger on Start", or use an EventTrigger component to start it another way. A common example is the ColliderEventTrigger, which requires a Collider marked as a Trigger and will start the Event when another object calls OnTriggerEnter. You can specify a tag and component this object must have before starting the Event.

Events should have an AudioHandler component for running AudioInstruction Actions.

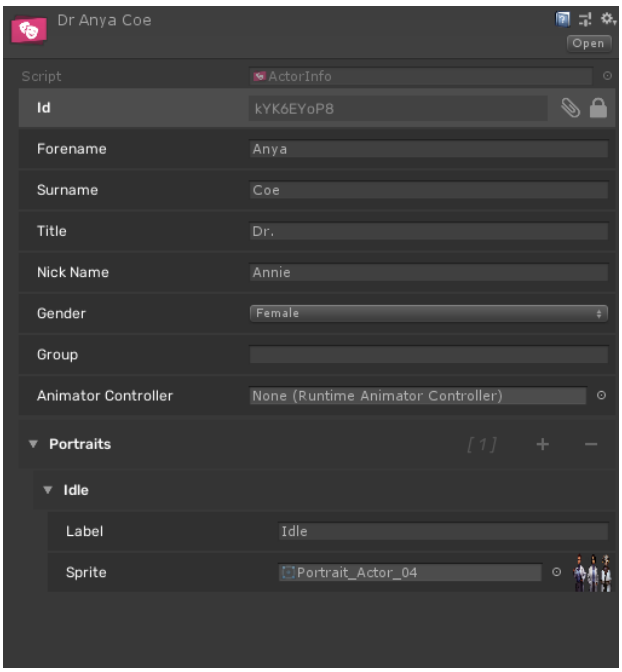


Event Advancers

Event Advancers are components that can invoke an "Advance" C# EventHandler which nodes can subscribe to advance parts of the dialogue. An example is the Speech node, which will move to the next node if this event is invoked. Discourse ships with an example MouseClickAdvancer which invokes this event on Input.GetMouseButtonUp(0). You can write your own Advancers for other Input systems with just a couple of lines of code, by implementing the "Advance" EventHandler required by the IEventAdvancer interface.

```
public class MouseClickAdvancer : MonteBehaviour, IEventAdvancer
{
    //------------------------------------------------------------------------------------------
    // Events
    //------------------------------------------------------------------------------------------

    public event EventHandler Advance;

    //------------------------------------------------------------------------------------------
    // Unity Lifecycle Methods
    //------------------------------------------------------------------------------------------

    private void Update()
    {
        if(Input.GetMouseButtonUp(0)) Advance.InvokeSafe();
    }
}
```

Actor Info Assets



Actors are Discourse's understanding of the characters in your story, and ActorInfo assets are where you list their details and how you reference Actors between graphs and components. These can be created from the CreateAssetMenu, or right clicking in the project panel, and selecting Discourse > Actors > New Actor.

Actors have a unique ID that gets auto-generated upon creation. This ID is used instead of the name to reference Actors, so that you can change the character's name halfway through development and not break your project (see: Parsing).

You can change this ID for convenience's sale, but it must remain unique and we recommend never changing it after initially setting up the Actor. An example of an ID change you may want is to set your Main Character's ID to either "MC" or "Protag" so that you have an easily memorable shorthand to use in the future.

To change an ID, click the lock icon to Unlock it, and then type your new ID. Lock the ID once you have updated it. You can copy an ID to your clipboaard by clicking the paperclip button.

Actors have Forenames, Surnames, Titles and Nicknames. These can be used in different ways with Parsing options (see: Parsing).
An Actor's gender can be selected from the list of Genders you supply on the ReferenceDatabase asset (by default, Male and Female).

You can define a list of Portraits to switch between here, by clicking the Portraits collection (to expand it) and then the plus button. Actors should have at least an Idle portrait if you intend to use this feature.

An optional Group can be supplied, which will appear as a tab in any Actor Name dropdowns. You can also supply an AnimatorController here to make use of the ActorAnimationDropdown attribute, if you wish to access the names of all AnimationClips in use on that controller in a custom Action.

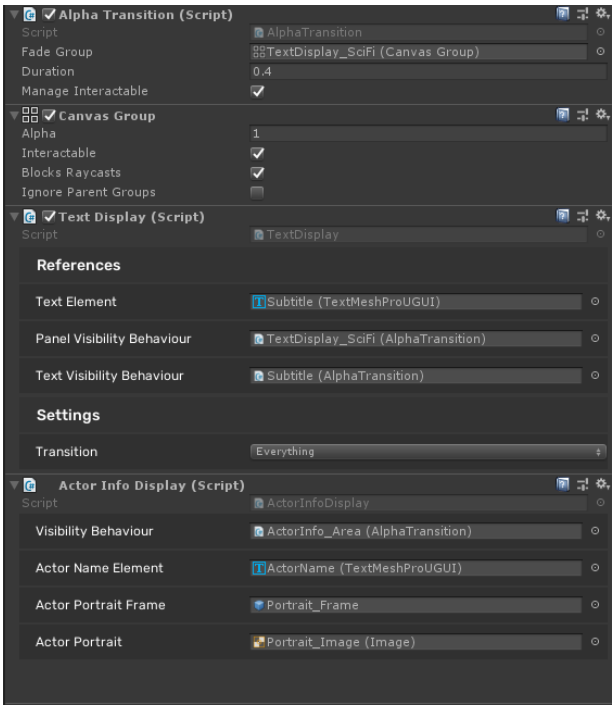
UI Components

You can use or adapt the prebuilt UI prefabs for any of your projects, or make your own to swap in. These are really simple to create.

On the right are the TextDisplay and ActorInfoDisplay components, which between them handle the display of dialogue text, Actor names and Actor portraits.

Many of the stock Discourse UI components make use of VisibilityTransitionHandlers; an example of which can be seen to the right in AlphaTransition. These components can help transition text and other UI elements before updating their information. The AlphaTransition component uses a CanvasGroup to fade a panel down before updating its information.

For an example of how to construct per-speaker UI, please see the PerSpeakerUI demo scene in the Example Content.



Parsing

A large part of Discourse's power shines when you make use of Parsing. All text in the supplied nodes will be run through the Parsing system before being displayed on screen. It is capable of inserting Global or Actor-specific variables in-line (and applying mathematical operations to these results), switching between gendered text, and displaying Actor's names in different formats. Below is a list of each of the patterns available in Parsing:

Actor Names

In order to keep your game's data independent of character name choices, you should reference all characters by their Actor IDs. This means if you decided to change your main character's name from John Smith to Jack Sparrow the night before you ship, the game wouldn't break.

I was speaking to **$A[actorID]** the other day

Results in: "I was speaking to Forename the other Day" -- insert the actor's Forename by their Info's ID.

I was speaking to **$A[f/actorID]** the other day

Results in: "I was speaking to Title Forename Surname the other Day" -- insert the actor's Full, titled name by their Info's ID.

I was speaking to **$A[l/actorID]** the other day

Results in: "I was speaking to Title Surname the other Day" -- insert the actor's Titled Surname by their Info's ID.

I was speaking to **$A[n/actorID]** the other day

Results in: "I was speaking to Nickname the other Day" -- insert the actor's Nickname by their Info's ID.

I was speaking to **$A[/actorID]** the other day

Results in: "I was speaking to the other Day" -- blank out the referenced Actor (this is useful if you're only getting variables on the Actor and don't want their name to appear in the text.

Case Matching

$G[actorID]:{M|he said|F|she said} you were in town.

Results in: "he said you were in town" -- case-match gendered text, using the gender of an Actor with actorId. In this case, their Gender is set with an Id "M" (Male) - these can be edited on the ReferenceDatabase.

Variables

Variables can be inserted in-line into text, either from a Global collection (on the ReferenceDatabase) or local to an Actor. Variables that do not exist before being requested will return the default value (Floats: 0, Integers: 0, Strings: "", Flags: false).

Global

Apparently **$I[integerId]** men attacked the castle

Results in: "Apparently 50 men attacked the castle" -- insert an integer by an ID stored globally (in this case, a variable set to 50).

You took **$F[FloatId]** seconds to answer

Results in: "You took 1.54 seconds to answer" -- insert a Float by an ID stored globally (in this case, a variable set to 1.54) to 2 dp.

He called his dog **$S[stringId]**

Results in: "He called his dog Billy" -- insert a string by an ID stored globally (in this case, a variable set to 'Billy').

Actors

He has **$A[/actorID]=>I[intIdOnActor]** cars in his collection.

Results in: "He has 50 cars in his collection" -- insert an Integer on an Actor with actorId (in this case, set to 50).

$A[actorID] said he has **=>I[intIdOnActor]** cars in his collection.

Results in: "Forename said he has 50 cars in his collection" -- insert an Integer on the last-used VariableCollection (in this case, an Actor with actorId).

Predefined

$TIME - UnityEngine.Time.time value

Reference Database

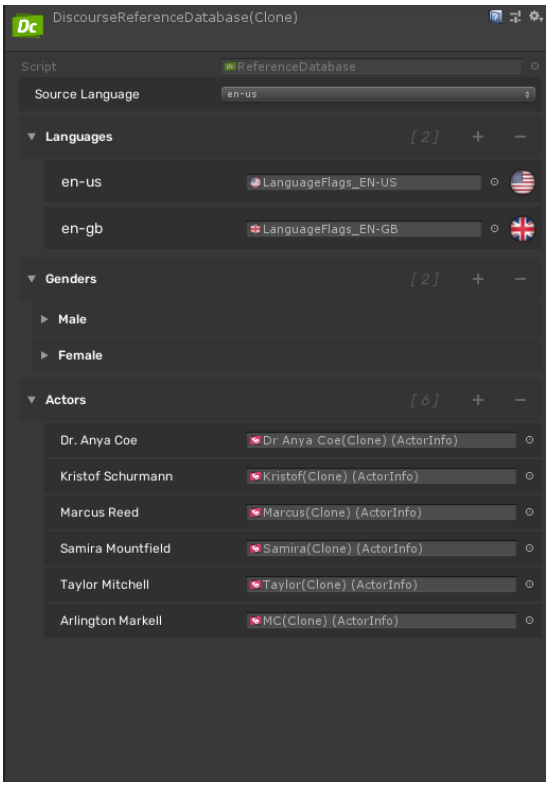
The reference database asset lets you assign many important references for your story to function correctly.

You can add language codes here for each language you plan to localise to. Changing the Source Language will change the language graphs are viewed and edited with. You can assign a Sprite to language codes if you want an easy way of displaying an image beside it in a custom UI.

Here you can also add Genders (by default, this database is initialised with M: 'Male' and F:'Female', if your story has races with other genders that you need to case-match text for.

You must also list your Actors on this asset. Actors are listed here to avoid Resource loading potentially dozens of assets, which Unity recommends against.

By default, the ReferenceDatabase is cloned at runtime so that any changes you make to variables don't carry across editing sessions. Developers can manage their own Reference loading by using the ReferenceLocator class.



Preference Database

The preference database asset is used to select between less frequent settings that should be mostly developer-facing. Here you can set the preferred language; the language that the game will display any localised text in.

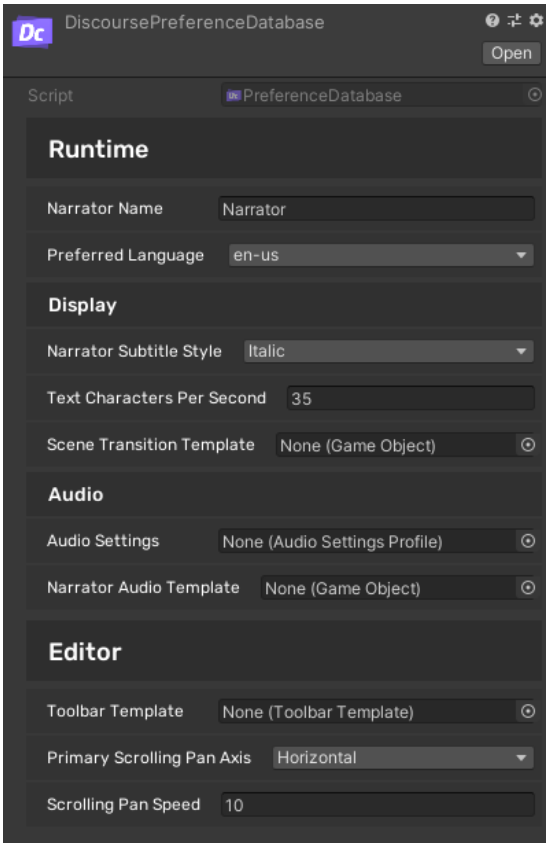
You may also define a TextStyle to use for the Narrator text.

A SceneTransitionTemplate prefab can be referenced here which will be instantiated to handle scene transitions. This field allows selection of any asset, but you should only pick a prefab with an IVisibilityTransitionHandler component on its root.

You may also supply a prefab to use for the Narrator's Audio, which similarly should have an IAudioHandler component on its root.

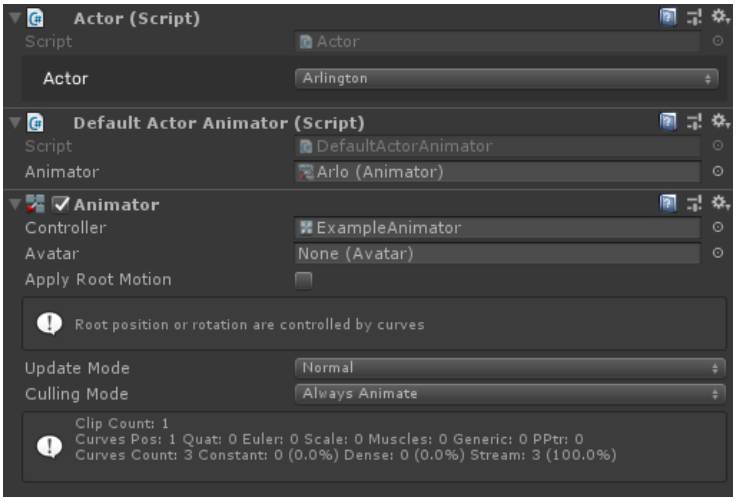
By default, the PreferenceDatabase is cloned at runtime so that any changes you make to variables don't carry across editing sessions. Developers can manage their own Preference loading by using the PreferenceLocator class.

You can create custom toolbars to change the Node buttons in the Main Window by assigning a custom Toolbar Template. These can be made by navigating to Create/Discourse/Editor/Toolbar Template. Add whatever nodes you want to the list, and assign the template to the Preference Database's "Toolbar Template" field.



Actor Runtime

In order for Actor's to be animated, or use Audio, they should be registered with the Actor component. This will register the selected actor's Id on the ReferenceDatabase, and get references to the ActorAnimator and AudioHandler components on this object.



Localisation

All display text in the default nodes are built to work with DIscourse's easy localisation system. Local strings are stored in lookups local to each Event. You can preview these strings by changing the Source Language on the Reference Database (before entering playmode). Data will be saved for each of the language codes you have listed on the database, keyed by that language code (e.g. en-us).

You can export these strings to a CSV file for easy localisation by your translator. To do this, select and right click the Event you wish to export in the project window, and choose Discourse > Export Localised Strings CSV. This will prompt a file dialogue where you can select a directory to write the file to, and then open in your spreadsheet editor of choice. An example of such a file can be seen below.

	A	B	C
1	ID	en-us	en-gb
2	c5397cc6-8698-4549-88da-d226ee0ee133	Lots of US text	Lots of GB text
3	8ce77505-87bd-4aed-9f73-dfc0d0c357b8	More US text	More GB text
4	d41f8b81-4563-4ee6-82e7-1a199eae8651	US English text here	GB English text here
5	52ff9d73-900d-418d-9a2c-25547d453b7a	Another american english line	Another British English line
6	bc71293c-2f54-48ca-aec3-adbc9d43079f	Some more English for Americans	Some more English for Brits
7	557c9d7d-c9da-461e-9a8b-d2a4ccbac959	Plenty of US English	Plenty of British English
8	06c3487f-4bfe-4df2-91bc-113ef74cd42a	Lots more US English	Lots more GB English
9			
10			
11			
12			
13			

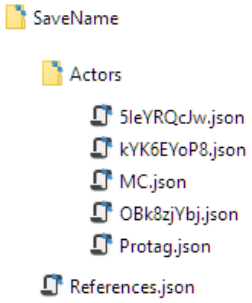
You can add more columns for other language codes, but these will need to be added to the References Database asset in order to bre previewed, After handing this off for translation or editing the file yourself, save and close it, then select and right click the Event again, and choose Discourse > Import Localised Strings CSV. Navigate to the file and accept the dialogue. All strings will be updated with their new values.

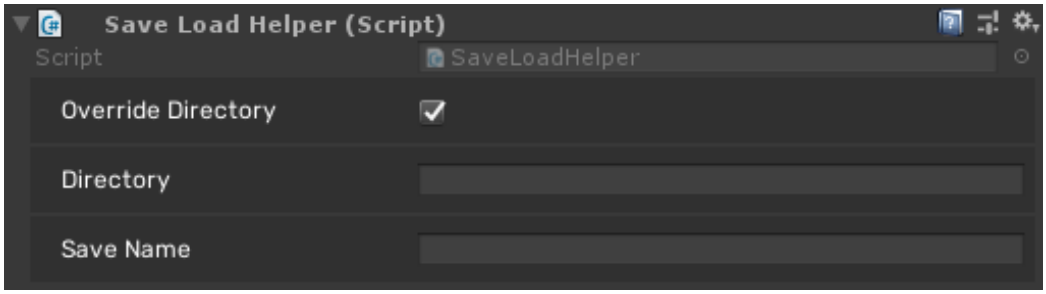
Save / Load

Discourse provides some high level helper components and also an API for saving and loading to/from **JSON**. This is built on the Unity JSON serializer and so you can expect performance to be typical of that. If you wish to write your own save/load system, or need to serialize to/from a format other than JSON, the objects you need to concern yourself with are the **ReferenceDatabase** and the **ActorInfo** objects listed in the **Actors** property on that Database (both of which are ScriptableObjects).

Discourse ships with an example **SaveLoadHelper** component that you can use to save and load the ReferenceDatabase and all referenced ActorInfos to JSON. There are methods on this component to save to a specific directory and filename, but also to use the paths listed in the inspector. These methods can be called via your own UI, and optionally through Unity's button event handler components, however their restriction of single primitive parameters limit which of these methods you can setup through the inspector. By default, this component will save data to the Application.dataPath directory.

Save data managed by the provided SaveData classes will be written to a directory at the given path, with the ReferenceDatabase immediately under it, and an Actors subdirectory with all ActorInfos listed by ID.





The example SaveLoadHelper component you may call via UI or use as reference to write your own

Developers can either write their own save system from the ground up, or make use of the **SaveData** class, which has more granular methods for controlling which objects to save, and shows the necessary steps should you choose to write your own. The SaveData class has a **Builder** and **Reader** subclass for writing/reading the necessary data to JSON, There is also a **SaveUtility** helper class that wraps these methods for easier use, such as Saving/Loading the ReferenceDatabase.

To load a ReferenceDatabase, you will need to register a new **ReferenceLocator** with the **References** class. You can use the **RuntimeReferenceLocator**, or write your own. These classes must inherit from the abstract class **ReferenceLocator**, override the **Priority** property and retrieve an instance of ReferenceDatabase via the **LocateDatabase()** method. ReferenceLocators are then registered with the References class and sorted in order of priority.

Custom Nodes

Developers can write their own custom nodes to extend the system to their needs and implement bespoke gameplay logic mid-graph. To do this, two classes need to be made: a **DiscourseAction** subclass; the runtime object that will actually enact the logic, and a **Node** subclass; the editor-side object that handles how the node should be drawn in the graph. Both require very little code at a minimum to work, but have several properties and methods you may override to get exactly what you're after. The only limitation of these nodes is that may not have incoming pins; in the interest of simplicity, Discourse was designed to avoid becoming a Visual Scripting solution and as such, any data you need to draw from at runtime should be managed through the **EventContextLog** class, which you may modify to get and set references to other objects.

DiscourseAction

Your runtime class should inherit from **DiscourseAction** and either **ISyncEventAction** or **IAsyncEventAction**. The "Sync" variant of this interface is for actions that should run their logic immediately, whereas "Async" is for those which should run over time through a coroutine. Both Sync and Async should implement a method named "**Run**" and take an EventContextLog as a parameter, with a return type of **void** or **IEnumerator** respectively. The Action can also override **OnEnter()** and **OnExit()** which are called immediately after and before switching between Actions. If an Action can run immediately or over time, implement Async and use the OptionalYield class to opt in or out of awaiting a Coroutine.

The inspector that will be drawn in the Editor's PropertyPanel is built from this class rather than the Node, so any serialized fields you need can all be put in one place here. In order to make use of Discourse's string dropdowns for things like Actor references, you can make use of the provided attributes (please see: **Attributes and Types**).

```
public class DebugActorCash : DiscourseAction, ISyncEventAction
{
    //--------------------------------------------------------------------------------------------
    // Inspector Variables
    //--------------------------------------------------------------------------------------------

    [ActorNamesDropdown("Actor", false)] [SerializeField] private ActorReference _actor = null;

    //--------------------------------------------------------------------------------------------
    // Interface Methods
    //--------------------------------------------------------------------------------------------

    void ISyncEventAction.Run(EventContextLog contextLog)
    {
        int cash = References.ActorInfo(_actor).Int("Cash", true).Value;

        Debug.Log($"{_actor.Info.FullName} has {cash} cash.");

        MoveToAction(LinkedAction);
    }
}
```

An example of a custom Action class, which logs the value of a variable named "Cash" on a specific Actor at runtime.

Once your action is complete, you should call **MoveToAction()** to advance to another Action. You can use the protected **LinkedAction** field for almost all cases here, which references the Action that was last connected in the Editor (i.e it's correct for any single-output nodes, but won't be correct for nodes with several outgoing connections, such as OptionSet).

Node

The custom node must inherit from Node (in the Montebearo.Discourse.Editor namespace), and at minimum override the ActionType property of the DiscourseAction it houses.

```
public class DebugActorCashNode : Node
{
    protected override Type ActionType => typeof(DebugActorCash);
}
```

An example of a Node for the DebugActorCash example Action; implementing the bare minimum code required.

The properties a Node can override to change its display behaviour are as follows, many of these are cached at discrete times to avoid expensive GUI calls,so you needn't worry about sacrificing performance for the sake of clarity and appearance.

- The **SettingsTemplate** property can be overriden to change the colour, size and other settings of the node, by constructing a new **NodeSettings** struct. This specifies the size, colour, vertical positions of its Links, whether the node should display a slim or wide coloured "tab", whether the node should display a Title, whether the Node should show its Hook, and whether the supplied Link positions are normalised across its height or in fixed units.
- The **Path** property can be overriden to modify the path to the Node in the Search widget (space bar). By default this will be the Action's Name, placed in the "Custom" subfolder.
- The **FormattedActionName** property is used in a few places where the Action's Name should appear in plain English (stripped of namespace and with casing spaces added). This can be overriden should it appear to work incorrectly.
- The **ActionType** property must be overriden, and point to the type of DiscourseAction this Node represents.
- The **Title** property can be overriden to change the (bold) title that appears at the top of the Node.
- The **Subtitle** property can be overriden to change the smaller text that appears below the Title on a Node (unless its settings specify otherwise).
- The **ParseSubtitle** property can be overriden if the Subtitle should first be run through the Parsing system.
- The **ParsingSelfProvider** can point to an **IVariableProvider** to use for any Parsing which makes use of the on-self syntax (for example pointing to an ActorInfo referenced in the inspector).
- The **TrimTitle** property can be overriden should the Node's title not be trimmed to fit the Node's width.
- The **EditorObject** property can be overriden should the Node need to point to a **Unity.Object** other than its own Action to draw the inspector for. In almost all cases this will not be the case.

EventContextLog

Custom DiscourseActions must run their logic by implementing the required methods in either ISyncEventAction or IAsyncEventAction, both of which take an EventContextLog as a parameter. This object provides helpful passthrough properties for the Text & Option display, AudioHandler and Advancer components on the EventBehaviour, along with methods to retrieve referenced SceneObjects and Cameras on that EventBehaviour. You may extend this class to store references to key parts of your game should a custom Action need to reference something out of the Discourse ecosystem.

ICameraSwitchInstruction

This interface allows an Action to change the active camera via the CameraTransitionHandler component on the EventBehaviour. It must provide an ICameraSwitchInfo that details whether it should switch, and the ID of the Target Camera, both of which are both managed with the provided CameraSwitchInfo class. The camera change will happen immediately after OnEnter() is called, but before Run().

IAudioInstruction

This interface should be implemented to play audio through the AudioHandler component on the EventBehaviour. It must implement an AudioInstruction to enact (Play/Pause etc), and depending on the instruction, optionally implement an IAudioClipProvider and IAudioModifier.

IAudioClipProvider

IAudioClipProvider labels an object as a provider of some form of audio, however you will need to implement the generic variant, with the type depending on the Audio system you choose to use. If you're using Unity's default Audio system, you will want to use IAudioClipProvider<AudioClip>, whereas if you're using FMOD you might want the generic type to be string and pass the path to the audio you want to play. This type must be compatible with the AudioHandler you play the clip with (e.g. UnityAudioHandler).

IAudioModifier

The IAudioModifier points to settings to modify the Volume and Stereopan of the AudioSource you're playing on, should you need to adjust volume per-clip. By default, you can use the provided AudioSettings class to manage the required settings.

Attributes and Types

Discourse ships with several built in types and attributes to help make your life easier when working within the system, and also to help keep any extensions you make to the system in-keeping with existing UI. Much of this is built on top of the MonteBearo Core libraries, which you are free to use within your game so long as you are using Discourse and are in compliance with its included license.

BlockProperty

The BlockProperty attribute will draw a serialized field in the inspector on a block background with a larger label font, using its default PropertyDrawer. You can optionally supply a custom label and the name of a property that will determine whether to Show/Hide the field.

BlockHeader / SubBlockHeader

The BlockHeader and SubBlockHeader attributes are used to decorate an inspector with a large and medium sized title header, drawn on a block background.

BlockFoldout / Collection

The BlockFoldout and BlockFoldoutCollection attributes are used in conjunction with the IFoldout and IFoldoutCollection interfaces, to draw classes within a clickable foldout, and collections with + / - buttons to add or remove from the collection. They must implement the methods and properties of their corresponding interfaces.

ActorNamesDropdown

The ActorNamesDropdown attribute should be used in conjunction with an ID-backed reference object (such as the provided ActorReference) to serialize the ID of a referenced ActorInfo, but draw it as a convenient list of names. You can specify whether this list should allow for the selection of “Narrator” or not.

LocalEventString / Area

The LocalEventStringArea attribute is to be used in conjunction with the LocalEventString object, which manages a localised string on its parent DiscourseEvent. It will display the language being edited in its header and update the corresponding serialized string.

SceneCameras / Objects Dropdown

The SceneCamerasDropdown and SceneObjectsDropdown attributes (both SceneReferenceDropdownAttributes) are to be used in conjunction with an ID-backed reference object (such as the provided SceneReferenceInfo) to serialize the ID of a referenced Camera or SceneObject, but draw it as a convenient list of names. These references will be looked up on a SceneReferenceHandler component.

GenderDropdown

The gender dropdown will draw a list of the Genders supplied on a ReferenceDatabase for an ID-backed reference, similar to the ActorNamesDropdown attribute.

Actor Decorators

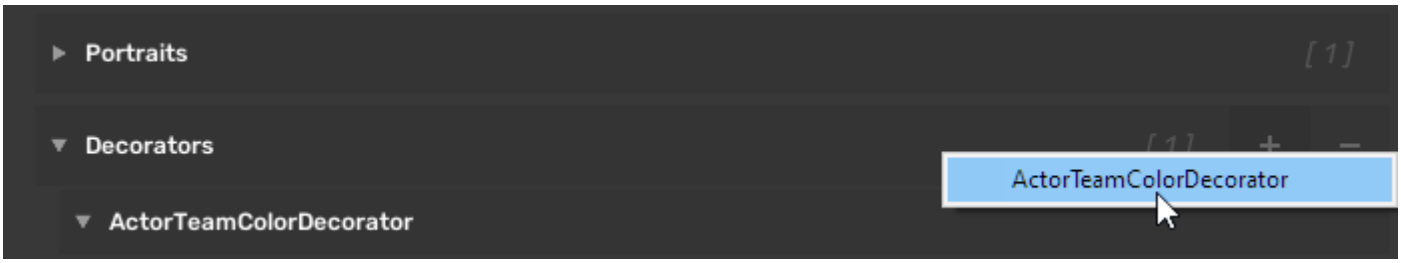
It's typical to have various labels, or pieces of information associated with characters in your game; their faction, a team color, the list goes on. Discourse offers Actor "Decorators" to achieve this.

Creating a custom decorator is very simple. We'll use the ActorTeamColorDecorator supplied in the Example Content as an example:

```
public class ActorTeamColorDecorator : ActorDecorator
{
    //------------------------------------------------------------------------------------------
    // Inspector Variables
    //------------------------------------------------------------------------------------------

    [BlockProperty, SerializeField] private Color _teamColor = Color.white;  Serializable

    //------------------------------------------------------------------------------------------
    // Public Properties
    //------------------------------------------------------------------------------------------

    public Color TeamColor => _teamColor;
}
```

Your decorator must inherit from ActorDecorator (in the Montebearo.Discourse namespace). From here, you can add any data you like, such as a TeamColor, in the example above.

Once your decorator is compiled, it will appear in the list of available decorators on ActorInfoAssets. To add a decorator to an Actor asset, select the asset, expand the Decorators foldout, press the "+" button, and select which type of decorator you want to add:



To use this data at runtime, you can use one of the many methods on ActorInfo for accessing Decorators. Below is an example of how you might display an Actor's Team Color, using the HasDecorator() method. This method will return true if the ActorInfo has a decorator of the required type, and if so, passes that decorator as an `out` parameter:

```
public class DecoratorExampleDisplay : MonteBehaviour
{
    [SerializeField] private Image _colorDisplay = default;  Serializable

    public void DisplayTeamColor(ActorInfo actorInfo)
    {
        if (actorInfo.HasDecorator(out ActorTeamColorDecorator decorator))
        {
            _colorDisplay.color = decorator.TeamColor;
        }
    }
}
```

FAQ / Troubleshooting

TextDisplay is only showing the first character of text when using the Typewriter mode.

When using Typewriter mode, the TextMeshPro reference used for the text should be set to the Page overflow mode. However, an issue can still occur where only the first character of the input text is displayed. This is an issue with TextMeshPro, present in versions up to at least 2.1.4 (at the time of Discourse 2.0's release) - where pagination fails if the input string ends with a newline character. Please ensure your text does not end with a newline, or other escape character.

Documentation

For more detailed documentation of Discourse's API, please refer to the Wiki.

Thank you for using Discourse.

We look forward to exploring your stories!