DIALOGICAL

A dialogue Management System For Unity

v 1.4

by Gru @ Ennoble Studios

How to use this manual:

Import the extension into an empty project. Read this manual from start to finish. Follow along with explanations and examples. If you have any questions or issues, email me at dialogical.help@ennoble-studios.com and we'll find a solution.

Version Changes:

v 1.0 – Initial release.

v 1.2 – Added inline parameters and parameter sets. Added section "Techniques for getting parameters from dialogue options". See section "Upgrading from v1.0 or v1.1."

v1.3 – Added grayed out options, in addition to existing disabled options. See section "Working with Node Options & Node Option's Events". UnityLegacyGUIBasic script has been updated to support disabled options. All the scripts previously used only for demonstration and test scenes have been separated to a namespace Dialogical.TestEventHandlers to minimize "namespace pollution". In addition, these test scripts have been renamed corresponding to their function. Parameter sets no longer require casting in handler methods. See section "Technique 4: Using Parameter Sets" for details. Added "Search All Dialogues" menu option, see section with the same name for details. Corrected a small bug where window wouldn't close if some other window was open. Added "Search the open dialogue tree" option, see a separate section for details. Added "Once per conversation" node option parameter, see dialogue node options section for more info. Slightly tweaked fonts and colors to further aid readability. Added "Long range mode" option, see section with the same name for details. Added "Reference Node", see section with same name for details. Corrected bug where newer example scenes wouldn't load in some earlier versions of Unity.

v1.4 – Added the ability to export/import text from nodes as CSV. Migrated to GUIDs for nodes identification. Internal improvements. See sections: "Upgrading from v1.3 to v1.4" and "Using Import/Export features".

Purpose:

Dialogical was created to be best-in-class, lightweight, intuitive, extensible, applicable to any kind of project - dialogue management solution for your game. Modern looking, polished and fast, this extension was just what you need if your game features complex branching dialogue between characters. From a visual novel to a complex RPG, we got you covered.

It wasn't designed to create the GUI for you, but to allow you to integrate it with any kind of GUI solution out there. It also wasn't designed to provide any kind of visual coding support, but to allow your code to plug in at the right places. This is in line with the general philosophy of extensibility and adaptability, and thus allows you to use any kind of complex logic or variable type. This dialogue system is totally abstracted from the UI implementation and your variables and logic - it just feeds data through events.

Features:

- Intuitive Node System: quick, slick and polished.
- Nodes have their text, audio, a set of responses and series of events that call your code in a specific way.
- Native Multilanguage Support (no automatic translation).
- "Choice Node" that lets your code (or random choice) decide which node should go next.
- Unique "Minimap" node view for maximum convenience with large number of nodes.
- Ability to hide options (your runtime code decides).
- Specially designed Event System that allows you to not only call events on Dialogue steps, but call unique node-specific events for every option on every node. Those events allow you to do these things from your code side:
  - Interact with any type of variable, and not just base types.
  - Create any kind of logic, and later act on that logic with a Choice Node.
  - Not clutter the canvas with visual scripting or variables management.
  - Integrate Dialogical with any kind of Inventory and Quest System.

- Ability to display your variables in dialogue text or options.
- Timed Events to allow Automatic choosing of a certain option at a certain time.
- Ability to go back in dialogue; multiple nodes can to lead to the same node.
- Provided working Examples with detailed comments of:

    - Implementation of Old Unity GUI
        - Basic Scene: how it all works.
        - Basic Scene with events: where and how to call events.
        - Basic Scene Multilanguage
        - Choice Node Example
        - Portrait Switching Example for dialogue with multiple NPCs
        - Example showing Parameters from code.
        - Example with timed option auto-choice.
        - Example with choosing a random node with random delay (e.g. for ambient sounds).
    - Implementation with New Unity GUI
        - Basic Example.
- Saves to Scriptable Object format, which is a standard “Unity way” and allows editing even in Play mode. Small asset footprint.
- Automatic node resize for maximum visibility.
- Dialogue Tree validation.
- Unity Rich text support.
- Automatic and manual Saving.
- Works with all platforms, and both Unity Personal and Professional .
- Full Source. Code is tidy and well commented (C#).
- A detailed and comprehensive manual and tutorial.

How it all works:

General Architecture:

The main workhorse of this extension at runtime is a script Dialogue.cs. It takes your conversation asset as a variable, traverses from node to node, and fires appropriate events. For the system to work properly there is a recommended architecture to be followed (although with understanding you can, of course, deviate from it). Follow the instructions below to create a simple working scene with Unity's new GUI.

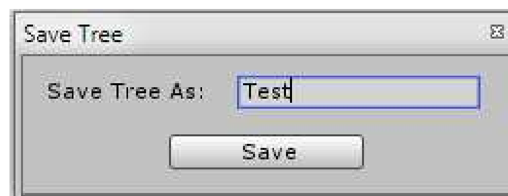
1. Import the extension into an empty project if not already.
2. Create a new scene.
3. Let's create an object that will represent the GUI holder – an object that holds GUI elements when dialogue comes up. Create a new Game Object, let's name it GUIHolder. Add to it the script called UnityLegacyGUIBasic from the 'Dialogical/Dialogical Test Scenes/Basic Scene' folder. Skin and background will be auto populated.
4. Let's create another game object in the root of the scene. This one will represent the NPC root object. Inside it, you would normally have a mesh, controllers, animations, other scripts etc. To it add a script called 'Dialogical/Dialogical Test Scenes/DialogueActivator.cs'. This will simply present a button on screen to activate a conversation. Keep in mind that in a real game you'd activate a conversation in a different way.
5. Create a child game object of the NPC and call it Dialogue. Attach 'Dialogical/Dialogue.cs' script to it. On a Tree variable in the Inspector you add a conversation asset. For this example let's use "Basic Example With Conditions" asset from 'Dialogical/Dialogues' folder (drag and drop it).
6. This example uses events, so we do have to have a script that holds these events on the same game object that has Dialogue.cs. Drag a script called NPC1Events.cs from "Basic Scene With Events" folder. For a subtle performance benefit, those two scripts should be the only ones on a game object (that's why we created a child object).
7. Run the game and choose option 1 and follow through the dialogue. Audio will start playing. This dialogue tree is used for the purpose of another example, so any questions about it you may have now will be explained later in this document.
8. Save the scene if you want.

You implemented yourself a simple dialogue. Now, to create one yourself from scratch. From the main menu open Window / Dialogical and the canvas window will open. Follow with the rest of this manual.

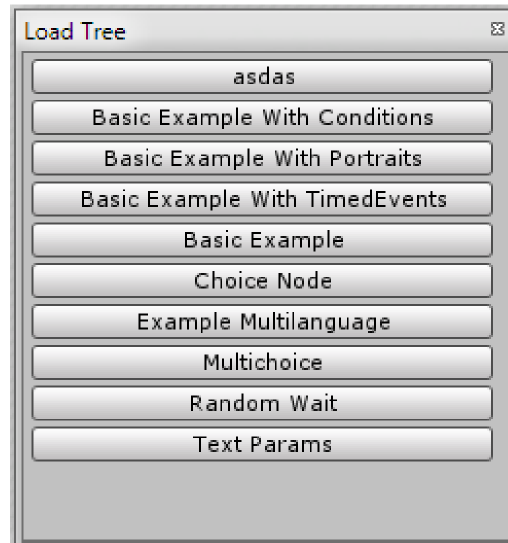
Top menu:

When you open the Extension, you will mostly find an open canvas and the top menu. Here is what the buttons do:

- New Tree: Creates a new empty canvas with just the start node. Your current tree will be automatically saved before that if it has been "Saved as…" previously. If not, and you have created nodes on the canvas, you will be asked if you want to save the current tree.
- Save Tree As: Bring up a dialog for saving the current working tree for the first time. After the first save, this button will be replaced with Save button. You are prompted to provide just the file name, and the asset will be saved by default to <Project Folder>/Assets/Dialogical/Dialogues. It is important for all your dialogues to be inside this specified folder if you want to have them available for loading. Saving will not be allowed if the file name is invalid. Do note that, if you provide a tree name that already exists it will be overridden without prompt.



- Load Tree: Looks inside <Project Folder>/Assets/Dialogical/Dialogues and provides you all the dialogues to load. As noted before, after loading the dialogue every change will be auto-saved.



- Save Tree: Saves all changes to disk. This saving happens automatically when appropriate, just like Unity saves assets at specific points, and that would be on: Enter/Exit play mode, script recompilation, exiting Unity. However you may want to periodically save manually (the only case where you'd lose data since last auto-save is if Unity crashes). This is equivalent of File -> Save Project if you have a habit of doing that.
- New Node: Creates a new node in the middle of the view.
- New Choice: Creates a choice node in the middle of the view.
- Delete Node: You will be prompted if you want to delete the node that was last selected. The node for deletion will be marked in red.
- Delete Option: You will be prompted to delete the option that was last selected. The option for deletion will be marked in red. Note that if you have just created the canvas or have no nodes other then Start these last 2 options will not be available.

Alternative buttons:

Menu buttons get the alternative function if you hold Ctrl key. Note: for this to work, Dialogical canvas has to be in focus, so if it is not click once on the empty area.

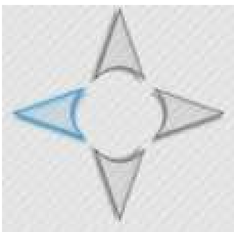
- Save Tree As…: If you have a tree loaded already and you hold Ctrl, you will get the option to save this tree under a different name. You will then work off of that tree.
- Load for Viewing: If you hold Ctrl instead of Load Tree this option appear. This option only loads the tree in "viewing mode". This means that any changes you do here will not be applied to the tree you loaded (but to a temporary tree called "newTree"). For confirmation, look at the bottom left and notice that a tree with name "newTree" is active. Since every change is automatically saved,

this is a useful option. Also, there is another caveat to note: since every change saves the asset (even panning the view) just by viewing you are probably changing the asset and that may be annoying if you work with source control. Use this option if you don't want to apply changes, and then use "Save As…" if you decide to save them later. In that case the dialog will automatically be populated with the name of the tree loaded for viewing, and on hitting Save old one will be overridden.

Working with source control:

There are no special changes to note except that newTree.asset and newTree.asset.meta should be ignored. This one gets recreated every time new canvas is created or loaded for viewing and temporary changes are saved here.

Navigation Arrows:



In the top right corner, there are 4 compass-like arrows. The blue arrow shows there are nodes that completely lie outside the viewport (in this case to the left).

Panning around, selecting, dragging and deleting nodes:

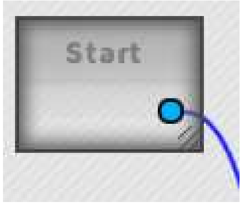
You pan the view by pressing and holding any mouse button (left, middle, right) on an empty space and dragging around.

If you left-click on a node that node will be "last selected" and the title label will become white. If you were to press Delete Node in the menu, this last selected node will be marked for deletion and the prompt will appear. Note that in order to select a node you must click on a space that is not inside another control (e.g. a text field) of a node, otherwise that one will take the click event. I usually click only on Title label.

Click and drag to move a node. Right click on a node (not its option) to bring up the delete dialog.

Due to a way Editor GUI works in Unity you need to be careful where you click if nodes are overlapping. Click on a unique space of a node and drag them out of overlap.

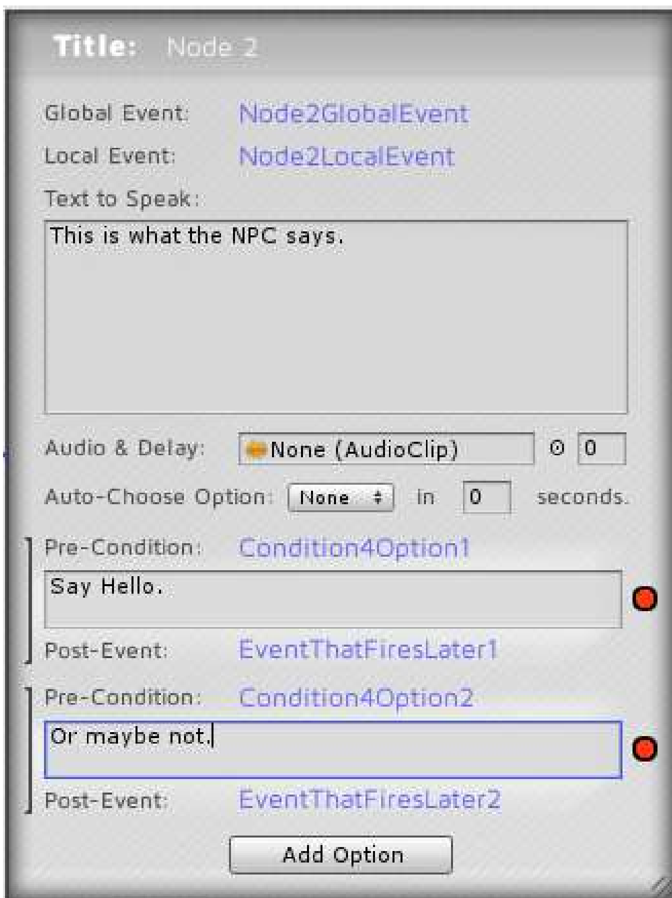
Start Node:



Start node is the first node that is automatically created when new canvas is open. It can not be deleted, and also the connection from it can not be deleted, only redirected to another node. When execution starts, Dialogue.cs will first find this node, and then evaluate connections from there.

Dialogue Node:

Dialogue Node is your base node and one of two available types of nodes for you to place. You will be using it most of the time, so note that it is very versatile. Double left click on a canvas to create it (it will be centered on the clicked position).



It has the following fields:

- Node Title for identification
- Global Event
- Local Event
- Text to Speak
- Audio to play
- Delay for this audio
- Which option to auto-choose …
- … and in how many seconds.
- Options below with their events

Input fields for events are hidden, you click on the appropriate place and cursor will come up.

Text, audio and audio delay are language-specific.

For audio, you just drag the audio clip and enter delays as floats.

Auto-choice will be activated if the auto-choice drop-down is different then “None” and delay is bigger then 0. This drop-down numbers the options by index starting from 1, so available options in the example above would be: “None”, “1”, “2”.

The node will first be "evaluated" when execution comes to it from another node (start, choice node, other node). Evaluation means the events will be executed first, then text parameters will be substituted, then auto choice will be set up and finally the event will fire which notes that node has been evaluated and is ready to display on screen – detailed explanation follows.

When the first node on a canvas is created, Start node is automatically connected to it.

Connecting Nodes:

Connecting nodes is very intuitive – just drag the connection from an option's circle to any node's body. Not-connected options have a red circle (drag from inside the circle). Connected Nodes have a blue circle, and connected Choice nodes have a green circle.

To delete a connection, right click the originating circle (except from Start node). You can also re-connect from an already connected option.

Working with Node Events:

These Events are not usual events that you find in Unity or C#. If you remember the scene you made at the start of this manual, you added a special script that held events called NPC1Events. You can name it whatever you like, but it must contain methods with names entered in the nodes' event fields. Those functions will have signature like below – they would take no arguments and return void:

```
void Node2GlobalEvent() {
    Debug.Log("GlobalEvent fired.");
}
```

… would correspond to the node image on the previous page.

What happens is, when the node evaluation starts, SendMessage() is called by Dialogue.cs to fire events on the same game object. That's why it's important for this script to be attached to the same game object as Dialogue.cs. If any methods are missing, errors will be thrown at execution time and execution will continue without them. A good practice is to first create the methods and paste their names inside the node's input fields. If you are not using events on a particular dialogue tree the script is not required. You may notice that one tree usually corresponds to one NPC, so it would be a good idea to implement a naming convention for created dialogues (e.g. AreaName_NPCType_NPC-ID) and events scripts (e.g. DialogueEvents_NPCType_NPC-ID). NPCs can, of course, share dialogue trees. Methods could be separated in different script files attached to the same game object for convenience.

Basic Node has 2 separate Events: a global and local one. There is no significant difference between these, but global will be fired first. It is meant for Events that are shared between nodes, to minimize code duplication (an example would be portrait switching functionality when multiple NPCs are talking). Local events are meant for events specific to that node and will be used most of the time. Take note that these functions take no arguments and return void. There are, however, ways to feed data back to the nodes from inside them and influence execution via Dialogue's public variables, which will be explained later. The thing to realize here is that the place to set any variables for a node is inside its local or global event in your code. These variables will be consumed and reset before the node exits.

Working with Node Options & Node Option's Events:

A Node Option consists of:

- A Pre-Condition Event
- Text to display
- A Post-Event.
- A toggle for once-per-conversation options.

When a node is first created a default option "Continue" is created.

You delete the option by right clicking it or from the top menu button while the cursor focus is on it.

At runtime, Options are not evaluated automatically after the node is evaluated. Instead, after node evaluation a C# delegate is called to let you know the node is ready. Inside the script that manages the GUI you subscribe to that delegate (example script provided) so you know when to refresh your GUI, and when it's called you get dialogue choice options array by calling GetNodeOptions(). Inside this method call, node options are evaluated and only "passing" ones are sent back. This method returns a list of ConversationOption-s, which is a provided type that has an optionText public variable from where you get the text of a specific option. This process, although somewhat long to explain in writing, is commented out inside the examples which will be walked through later, and is in practice quite simple.

Pre-Condition Event is very similar to the events of nodes described above, they have the same signature. If you don't want the option to be sent to your GUI, in its pre-condition handling method you set Dialogue.lastCheckPassed = false, like this:

```
void Condition4Option1() {
    // Your conditions setup and checking.
    if (! myConditionsPassed) Dialogue.lastCheckPassed = false;
}
```

If your condition is passing, you don't have to set anything. You only set this variable to false if you don't want the option shown. This tells Dialogical.cs that this option should not be sent back. Naturally, you can do any other thing inside this event.

Post-Events are again similar. They are fired only when the option is clicked. Typical uses would include accepting a quest, adding XP to the player for successful check, adding an item etc.

You should set up your GUI so that on every button click you call a method CallOption (thisOption) of a current dialogue, so the clicks to buttons fire the option. This is very simple to do and can be found inside the provided examples.

Since version v1.3, there is a new feature to make node options disabled but still displayed in the GUI. This is done similarly to failing options, like this:

```
void MyOptionEvent() {
    Dialogue.conversationOptionIsDisabled = true;
}
```

Where MyOptionEvent is a sample Pre-Condition method on a node that will become disabled. This is relevant only to Pre-Conditions for dialogue options (as opposed to other events).

In your GUI implementation you would check for isDisabled field on the option itself and disable it in the GUI accordingly. UnityLegacyGUIBasic script shows an example.

Another new feature in v1.3 is the ability to hide conversation options that have already fired inside the current conversation. This is useful in, for example, RPG games, where we may want to ask about a range of things, but remove the conversation options that have already been explored. This option is valid only inside one conversation, and if the conversation with the same character is restarted all the conversation options will be available again. In case where we want to make options permanently unavailable, other techniques (such as checking for pre-conditions and choice nodes) must be used.

This is achieved via a checkbox on the top-right of every conversation option. All nodes crated in previous versions will have this field disabled by default and no change occurs. Once the option is checked, this option becomes available as demonstrated in "OncePerConversationOptions" scene.

Even if this option is graphically in line with the pre-condition events, it does not influence the firing of these. All the rules still apply if the option is not being filtered out by once-per-conversation feature, which is evaluated first.

Choice Node & Choice Node Options:

Choice node is the other type of node. It does not do with dialogue presentation, but with logic. Create it with double right click on an empty space or from the top menu.

During runtime, when execution comes to this node, all its options (called Choice Options) will be automatically evaluated.

Typical uses include: an NPC that treats you differently based on your relationship, different dialogue entry points if the NPC has been talked to before, different dialogue sub-trees based on quest status.

This node is used both for Conditional traversal – where you set up events to decide which option we should go to next, and for Random choice. You can combine both as in the picture above.

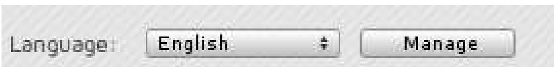
The process goes like this: of all the options, first the conditional ones (ones with Event names entered) are separated from the random ones (empty Condition Event Name). Then, first conditional ones are evaluated in the order of entry until the first one that doesn't fail is found. Condition fails satisfaction just like from before – if somewhere inside the your event definition you set Dialogical.lastCheckPassed to false.

If all conditional options fail, a random option is chosen from a set of random ones. They don't have to come after the conditional ones, the order can be mixed.

If no conditions satisfy and there are no random nodes, dialogue ends.

As before, you delete this node by right clicking it and delete its options by right clicking them – or from the top menu.

Languages:



Dialogical has built-in support for multiple languages. There is no automatic translation. Again, Node's and Option's text, Node's Audio and Audio Delay are influenced by the language that's currently selected.

To select the language in Edit mode, click the drop down on the bottom left of the canvas. Click Manage to open the shown Management window.

To change it in Play mode, somewhere from your code you set the public static index of the language in Dialogical.currentLanguageIndex to the index of the desired language. This index starts from 0 for a default language, in order of entry in the Manager window.

Languages are managed by index of creation and not name, so if you'd exchange names in the Manage window you'd just misname the languages. Having multiple languages with no name entered is allowed. When you create a language, it is immediately saved to all nodes. You just close the window when the operations are over.

All languages can be renamed, and all except the default one can be deleted with a "-" button on the side. If you delete the language, you will get a prompt, and accepting it all data inside all the nodes for this language will be deleted.

The language metadata is saved inside <Project Folder> /Assets / Dialogical /Dialogues /_languages . Do Not delete that file, especially if you're using more then 1 language in your project. The language translations are saved inside the nodes.

Reference node:

While working with nodes many times the dialogue option points to a previous node, that is one that is to the left of the current node. Same issue arises if a node points to self. This leads to dialogue representations with many connections that become hard to follow and don't look very good. That's why in version v1.3 there is a new type of node – a Reference node.

These are created by pressing the small "+Ref" button in the lower left corner of regular nodes or choice nodes. Any number of options can lead to this type of node, and the result is the same as if they were linked to the original. This node has a button "Show Original" that centers the view on the original node. The demonstration can be seen in "OncePerConversationOptions" scene.

You will notice this node does not have editable title. Its title is the same as the title of the original and as original title changes so does the reference.

Ending the Dialogue:

Dialogue ends when there is no link from the option chosen (manually or automatically) to another node. There is no special End Node.

Setting up your GUI with Dialogical:

Setting up any kind of GUI with Dialogical should be very easy. Examples are provided for Legacy Unity GUI and New Unity GUI Systems. Those are un-styled, but they work and with visual modifications can be used in your game. Other GUI types should be quite similar to Legacy GUI and should provide no trouble to integrate with.

Link between your GUI and Dialogical is through 6 delegates:

```
   // Delegate "Definition"
    public delegate void DialogueDelegate();

    // Methods called when any conversation is started.
    public static DialogueDelegate ConversationStartedStatic;

    // Methods called when this conversation is started.
    public DialogueDelegate ConversationStarted;

    // Static method that is called when the node is ready to display - we have text and we can fetch
options for drawing.
    public static DialogueDelegate NextConversationNodeStatic;

    // Method that is called when the node is ready to display - we have text and we can fetch options
for drawing.
    public DialogueDelegate NextConversationNode;

    // Methods called when any conversation is over.
    public static DialogueDelegate ConversationOverStatic;

    // Methods called when this conversation is over.
    public DialogueDelegate ConversationOver;
```

You will probably be using the static variations, inside the Start of your GUI script you'd do:

```
        Dialogue.ConversationStartedStatic  += Activate;
        Dialogue.ConversationOverStatic     += Deactivate;
        Dialogue.NextConversationNodeStatic += DialogueNodeSet;
```

Where `Activate()`, `Deactivate()` and `DialogueNodeSet()` would respectively: Bring up the GUI, Bring Down the GUI, Refresh text of the conversation and fetch the options. From there on you'd draw the text and options in a way that suits you best.

Setting your parameters within node and options display text:

This refers to picking up a variable from your code and displaying it somewhere within the dialog GUI. Similar to rules from before, you set a public static variable (of type `string[]`) in the event that happens just before desired text.

Internally, C#'s string.Format() method is used, which means that you set up your variables by numbers: {0} is a first variable, {1} is second etc. You just place them within the text, so you'd get something like this inside the node text:

NPC: "I am willing to sell this sword to you for {0} gold."

In the pre-condition event that fires on this node you'd set:

```
Dialogue.substituteTextParams = new string[1] { _swordPrice.ToString() };
```

This is explained in a separate example. If you had to expose 2 (or more) variables, you have something like this in the option:

Player: "[{0} / {1} Gold] I will buy this sword from you!"

And in the pre-condition event set:

```
Dialogue.substituteTextParams = new string[2] { _playerGold.ToString(), _swordPrice.ToString() };
```

String array that you supply is put in place of original parameter numbers. If in doubt, check the link below, along with our supplied example:

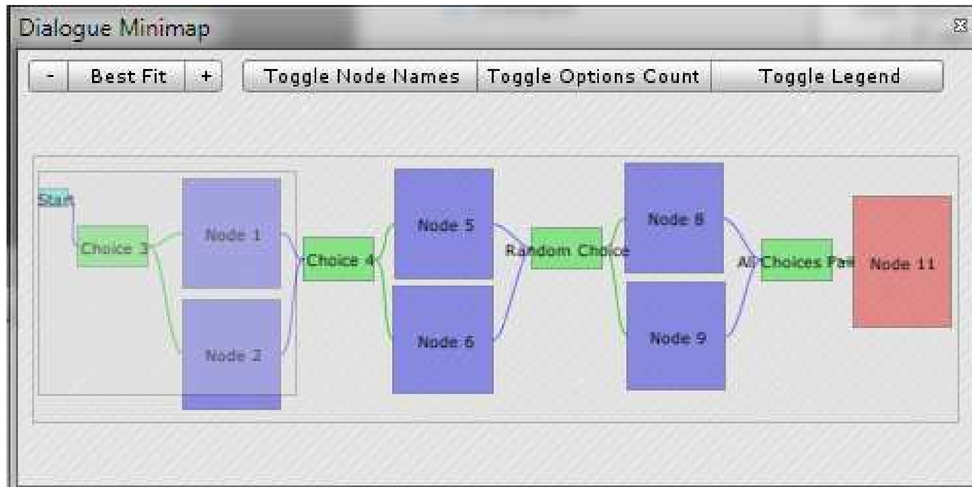
https://msdn.microsoft.com/en-us/library/system.string.format(v=vs.110).aspx

Validating Dialogue Tree:

Since the methods have to be input like strings, there is a chance one could mistype the method name, or not implement the method specified on the node. The way this is handled is if there is a missing method, no exceptions will be thrown by events (unless you specify otherwise in Dialogue._requireReceiver) and gameplay will continue like there was no method specified. To mitigate this situation, once you attach the tree you should validate it with the button just below it "Validate Tree". This will notify you if there are any errors (including missing methods) and will print these errors to the console if there are any.

Using a Minimap:

Minimap is the special feature of Dialogical. It is both quick and convenient, and can be brought up by clicking the bottom-right button on the canvas. For technical reasons, this solution is more stable then doing the zoom in/out on the canvas and GUI components.



The outer rectangle is the complete rectangle that encompasses all the nodes. The smaller rectangle represents the viewport on the canvas.

By default, minimap is open in "Best Fit" mode. You are free to resize this helper window how you see fit, and you can click Best Fit button again if you want to see the minimap in its entirety. That is because zooming in and out of the minimap is supported (click – and + in the upper left) so you can scroll the view of the minimap with the desired zoom level.

If you click and drag anywhere on the minimap, viewport will be moved both on the minimap and on the canvas.

Different nodes are colored differently by their connection status, and you can see how by bringing up the legend with Toggle Legend button.

Toggle Node Names switches node names on and off, useful if you have lots of nodes.

Toggle Options Count switches on and off the number of options on each node.

Test Scenes Comments:

Now would be a good time to walk through all the test scenes and study the examples a bit. Make sure your Console tab is visible, some examples print to it. It is recommended to go through test scenes in this order:

1. Basic Scene: just assembles the minimal scene for demonstration. Analyze all the scripts you haven't so far used in that project.
2. Basic Scene With Events: Just introduces events. Note how the missing methods are listed with "Validate Tree" and how the correct methods print to console.
3. Basic Scene With Events Multilanguage: Demonstrates Multilanguage use.
4. Choice Node: example demonstrating features of Choice node.
5. Portrait Switching: Demonstrates how you'd set up portrait switching if your conversation involved multiple NPCs.
6. Random: since on a node itself auto-choice is a single value, this demonstrates how you'd go about setting it to a random value within a range. This also demonstrates dialogues without visible options (there is actually one non-displayed choice in that case).
7. Text Parameters: Substitution of text parameters, from the example mentioned above.
8. Timed Choice: Demonstrates how you'd set up visible or invisible timed choice.
9. UnityNewGUIExample: Example utilizing the new GUI system.
10. Passing Parameters From Options: self-descriptive, as explained in "Technique 3: Sending the string parameter(s) from the Dialogical Node" section.
11. Disabled Options: Demonstrates working with disabled (grayed out) options.
12. OncePerConversationOptions Scene: demonstrates work with conversation options that get hidden once they are clicked, for the duration of the dialogue.

Techniques for getting parameters from dialogue options:

[Note: For getting the parameters in the other direction, from your script into the text of the dialogue option, see section called: "Setting your parameters within node and options display text"]

Dialogical does not try to draw variables and operations on the canvas itself. The variables are accessed directly from your code, using techniques described below. This allows you to do 4 unique and important things:

1) Access and set any type of variable available in Unity, and not just base types.
2) Create any kind of logic, and later act on that logic with a Choice Node.
3) Not clutter the canvas with visual scripting or variables management.
4) Integrate Dialogical with any kind of Inventory and Quest System that supports scripting.

There are a couple of techniques when getting parameters based on which option the user clicked. They will be explained individually.

➢ Technique 1: Getting parameters for a single dialogue option

Let's say for example that you have an NPC that wants to give a certain item to the player when a quest is complete. You would expose in the inspector of the script that gets called (on post-event) the parameter(s) you want, and create a separate method without parameters to feed those exposed parameters into the method that does the actual work. Consider the following dummy code:

```
public class SellItemToPlayer : MonoBehaviour {
     public GameObject   itemToGive;
     public int          goldForItem;

     void GivePlayerItem() {
         SellItem(itemToGive, goldForItem);
     }

     void SellItem(GameObject item, int price) {         // Just for example...
         if(Player.Inventory.HasEnoughGold()) {
             Player.Inventory.AddItem(itemToGive);
             Player.Inventory.RemoveGold(goldForItem);
         }
     }
 }
```

This script would go onto the same game object that has the Dialogue.cs script. The post event for the node that leads to item selling would be called `GivePlayerItem`.

This is the default way to get parameters in simpler conversation trees. It applies equally to 1 or more parameters for that particular dialogue option.

➢ Technique 2: Getting parameters for multiple dialogue options

If we have multiple dialogue choices that would need parameters, then the previous technique multiplies.

```
public class SellItemToPlayer : MonoBehaviour {
     public GameObject   itemToGiveX;
     public int          goldForItemX;
     public GameObject   itemToGiveY;
     public int          goldForItemY;

     void GivePlayerItemX() {
         SellItem(itemToGiveX, goldForItemX);
     }

     void GivePlayerItemY() {
         SellItem(itemToGiveY, goldForItemY);
     }

     void SellItem(GameObject item, int price) {         // Just for example...
         if(Player.Inventory.HasEnoughGold()) {
             Player.Inventory.AddItem(itemToGive);
             Player.Inventory.RemoveGold(goldForItem);
         }
     }
 }
```

It becomes clear that in this configuration you would need to expose all the parameters separately, for each node option that does the actual work, alongside with a separate method for each option that wraps the actual call to the method that does the work. This is sufficient for most cases, however there are cases with complex dialogue trees where creating a multitude of methods may not be appropriate. For such a case, the next solution applies.

➢ Technique 3: Sending string parameter(s) from the Dialogical Node

This is the new functionality added in Dialogical v1.2. First, note that we are using SendMessage internally to call methods on pre- and post- events. No reflection (other than what SendMessage does) is used, and that is by intention. This also means only 1 parameter can be supplied with this method internally. You can still have multiple parameters with this technique as described below.

The parameter is sent to the method by providing a value next to the method name, separated by a space (" "). This would look like:

Consult the scene titled "Passing Parameters From Options" and the accompanying script.

The parameters are received exclusively as string array in your handling methods. This means your method must have the signature such as `FloatParameter(string[] parametersRaw);` If this signature is not obeyed for methods with parameters runtime error will be thrown. Methods without parameters can still have the signature without the argument. The string you provide is sanitized only in a way that excessive spaces are removed and the method call is separated from the arguments and arguments are separated as individual strings. Any additional special characters are allowed in the arguments as they may have importance when you parse them. These arguments are then fed back to the appropriate methods as a string array.

You would then convert the string from that array to the appropriate type in the way that suits you best. In this implementation, Dialogical prefers speed of execution at runtime rather then user convenience, so client methods have to do the casts themselves and no automatic reflecting is involved. Examples how you would convert the parameters:

```
// "Silent" way
void FloatParameter(string[] parametersRaw) {
    float parameter = 0;
    if(float.TryParse(parametersRaw[0], out parameter)) {
        Debug.Log(parameter / 2);
    }
}

// Preferred way
void FloatParameterAltParsing(string[] parametersRaw) {
    float parameter = float.Parse(parametersRaw[0]);
    Debug.Log(parameter / 2);
}
```

Take a look at Parse() and TryParse() .NET methods for your appropriate type. The Parse() methods will throw and exception, and TryParse() will return the operation status.

Note: Unity does not handle overloading of methods called by SendMessage correctly, which means you should not have the same method with and without parameters overloaded.

Reference: https://docs.unity3d.com/ScriptReference/GameObject.SendMessage.html
"For performance reasons SendMessage does not handle overloaded methods correctly."

Multiple parameters are sent the same way: separated by spaces and populate the string array.

Note: when sending floats or similar types they should not have the suffix (e.g. not "0.03f", but "0.03").

Make sure to use the Validate Tree button on Dialogical.cs script to take full advantage of type recognition and reflection while in the editor, it will catch most of the errors in regard to methods and calling them.

➢ Technique 4: Using Parameter Sets

Parameter sets are a new concept in v1.2. They allow users to create special subtypes of Scriptable Objects, which hold the parameters and will be forwarded to caller methods. They are very powerful and require only a slight modification of the workflow described above. Most real situations can be resolved without them, but advanced users may want to use the unique flexibility they provide.

Consult the scene “Parameter Sets” with a representative example.

The procedure starts like this: first, type that represents the group of parameters should be created (aka parameter set). That type is a subtype of a class DialogueParameterSetBase, which in turn is a Scriptable Object. Using Scriptable objects has the special benefit of showing Unity’s inspectors for any possible type allowed in Unity. The creation procedure is fully automated, click Window/Dialogical/Parameter Sets/Create Parameter Set Class and the new window will pop up prompting for the name of the class. The new class will be created in Assets/Dialogical/EditorScripts/CustomParameterSets/. After creation, parameters (and optionally methods) can be created inside it as desired, those will be passed from the method to the receiver. Parameters are created identically like in normal scripts – usual MonoBehaviours (obviously outside of `#if UNITY_EDITOR` section). Then an asset should be created from that type. This is automated by clicking on Window/Dialogical/Parameter Sets/Create <ClassName>, where <ClassName> is the name of the class just created. It will be created in Assets/Dialogical/DialogueParameterSets/ with a class name and an asset index (they are safe to rename). For each event that will utilize parameter sets, one such asset should be created. The type itself is not connected to any specific conversation, so types should be created only for unique sets of parameters, regardless of which dialogue they are used.

Next steps are in Dialogical main window. The nodes now have a button in the right top corner names PS (as in Parameter Sets). This simply shows or hides the parameters set input boxes on the node. Once these are revealed, the created asset can be dragged down in the field (or selected using the picker). Once hidden, the name of that asset will be displayed instead of the selection box. When the PS input box is revealed, double click the already attached Parameter Set to quickly show it in the Inspector.

Now this asset will be sent to the receiving method. This means any string parameters from the previous technique are ambiguous with this call and this causes an error (either on tree validation or runtime if it was not previously checked).

Also, the method call now has to have a different signature. The receiver will look like this:

```
void ParameterSetMethod(DialogueParameterSetBase parameterSet) {
```

```
        var castParam = (MyDialogueParameterSet) parameterSet;

        Debug.Log(castParam.param1);
        Debug.Log(castParam.param2);
        castParam.DoWork();
    }
```

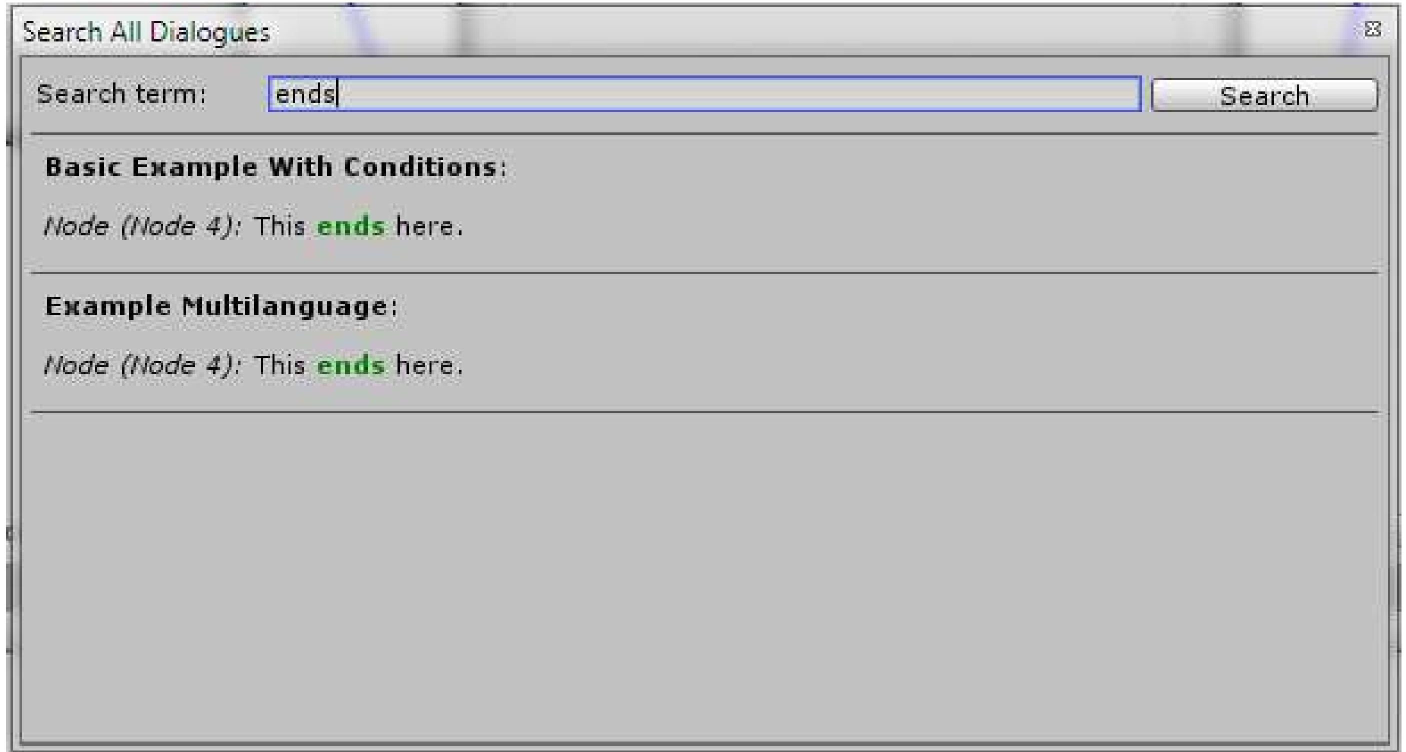
or

```
    void ParameterSetMethod(MyDialogueParameterSet parameterSet) {
        Debug.Log(parameterSet.param1);
        Debug.Log(parameterSet.param2);
        parameterSet.DoWork();
    }
```

In Dialogical version v1.2 the casting form DialogueParameterSetBase was mandatory. In version v1.3, the cast is not required and the type of argument inside the method call can be the same type of the custom parameter deriving from DialogueParameterSetBase. Tree validation allows for both options now.

Search All Dialogues

This is a new option in v1.3, accessible through top menu Window/Dialogical/Search All Dialogues.

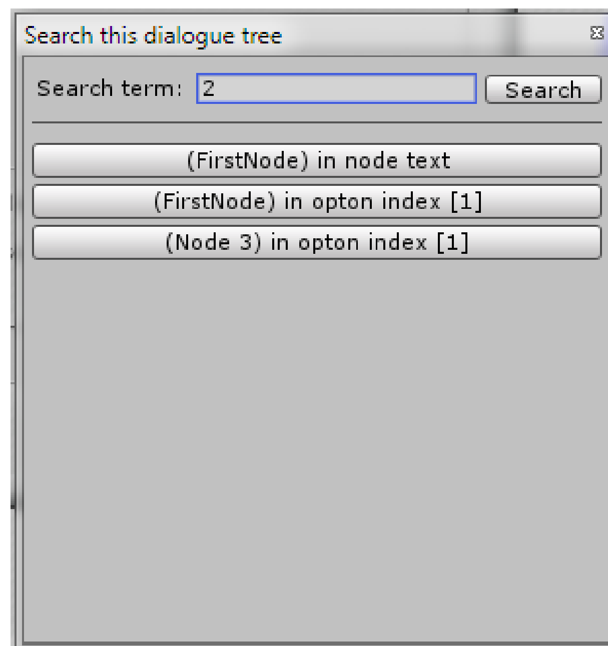


The search can be conducted by clicking the button on the right or pressing Enter (Return). Search looks into the text of nodes and nodes options, event names are not taken into account. The search is global across all dialogue assets in the project, and takes into account all the languages. Results show the asset name in bold, and where the result was found in italic. Matching strings are case-insensitive and displayed in green.

Search the open dialogue tree

There are instances where we want to search and quickly navigate to a node within the currently open tree. In the lower-right of Dialogical main window there is a Search button for that purpose since v1.3. The rules of search are the same as for the global search, except this search takes into account only the currently open dialogue node.

After the search is performed, a series of buttons appear, each one identifying the node by title and listing where the results were found. Results can either be in node text or in node option, and by clicking the button the window is quickly centered on that node.



Long range mode

Oftentimes when connecting nodes we want to connect to a node that is currently out of screen view. Before version v1.3 we would either drag the node into view or use Dialogical in "Maximize" window mode (accessed by right clicking the tab). This new option is enabled by holding Ctrl while creating node connections and the screen view is conveniently panned for longer reach. The speed and reach of panning can be increased if desired (see _longReachMultiplier option in "Changing Advanced Options" section).

Using Import/Export features

In order to better support workflows related to Multilanguage games, in v1.4 import/export to .csv format has been introduced. There are multiple ways to use this feature:

- Export:
    - All dialogues as individual files
    - All dialogues in a single file
    - Selected dialogues in separate files
    - Selected dialogues in a single file
- Import
    - All individual exports
    - Full export from a single file
    - Selected dialogues from separate files
    - Selected dialogues from a single file

Two new folders have been added to the project: *Dialogues_Exported* and *Dialogues_ToImport*. All export operations will write to the first and all import operations will be started from the second. These folders also have _info.txt files inside them in order to better support Asset Packaging process and are safe to delete.

Running "all dialogues as individual files export" will simply go through all assets present in the project (including examples assets if present) and export each one into individual files with the same name as the source file and .csv extension. This naming convention is important, as the individual imports will try to match the assets (text files to asset files on disk) by name first. If the names match, then match will be further confirmed by checking guids (global unique identifiers), which will be discussed in detail later.

Export will always overwrite existing text files without prompt.

The generated files consist of a header describing the columns, which is there for user convenience, and the following columns:

- Type of the entry, which includes: Tree, Node, Choice. This information must not change.

- GUID of the entry. This information must not change through the export/import cycle as the import process depends on getting the data back into the nodes based on the guids.
- Default language text. This serves as the base to translate from and this data will not be imported with the import process. This allows users to change the text in the default language on the nodes themselves while the translation process is going on without the fear of importing back old data.
- One column for each additional language. On import, all languages will be imported, unless the data for that language is an empty string. This allows workflows with split translations for different languages by setting the values of non-interesting columns to empty.

The data in the file is delimited by tabs. If the text in the nodes contained tabs for any reason, they will be converted to spaces irreversibly on the export process. The number of spaces chosen (11) for a tab is so the text looks the same inside the Dialogical's window.

This system is made to be resilient to most common changes. These include: adding or removing nodes between export and import operations, inserting empty lines in the translation files, changing the order of entries inside the translated tree etc.

There are, however, a couple of additional things that must not change across the export/import cycle. The most important one is the number and order of columns must not change. The languages are identified by index, so no deletion of columns is allowed. If we wish to not include a language in the import, the column relating to that language should be cleared from data.

The typical workflow could be like so: John the developer wants his game translated into 2 additional languages. He has 2 separate translators for these languages. First, he exports the existing dialogues, either as a single file or as separate files.

As a simple workflow, he just shares this file with translators and instructs them to use the appropriate column.

As a more complicated workflow, he prepares the files, perhaps by creating new Excel files that contain only the language of interest. When the first translation arrives, he puts this data in the appropriate column in the .csv and places that .csv in the Dialogues_ToImport folder and completes the import. He makes sure the data for the other languages is blank in order to not override existing data for other languages. When the other translation arrives, he again prepares the data in the .csv making sure the other languages are blank. This workflow is useful when doing partial imports and when we don't want translators to see each other's data.

It is always recommended to backup Dialogue assets before big imports. While best care has been taken to ensure the import process proceeds smoothly, there could exist certain peculiar cases leading to incomplete imports. It is every user’s own responsibility to protect their own data.

The easiest way to access the Export/Import option is from the main Dialogical window, at the bottom right. This will export only the currently open dialogue to a single file. All other options are in “Window/Dialogical/Export” (or Import).

Single file export will create a file called “_Full_Export.csv”, and this naming convention is significant. This file will have one additional line at the top, noting the count of dialogues inside this file. This number is significant and should not be changed.

The “selected dialogues only” options refer to the Dialogue assets that are selected before the option in the menu is chosen. For export, we select only the dialogues we want to export, the file that will contain the export is named “_Selection_Export” and same rules apply like in full export. For import, we also select only the dialogues we want to import into (not the source file, the source will be the file above).

Don’t forget to put the translated .csv’s into the Dialogues_ToImport folder. If they are in the Dialogues_Exported folder they will not be seen by the importer.

The elements are identified with guids. These are int64 (19-digit) numbers and not strings, to save memory. They are randomly generated and not based on file name or timestamp. This means the guids have to be regenerated whenever the assets are created by duplication, e.g. when pressing Ctrl + d on an existing asset, otherwise the new asset would inherit the same guids as the source. Dialogical does this automatically, so no user intervention is needed. Duplicating assets in the operating system while Unity is running is also supported, although users should never duplicate assets in such manner anyway (for example, because of .meta files). However, if the asset is duplicated in the OS while Unity is not running, the correct results are not guaranteed. That’s why in the Export menu there’s an option “Find and Fix Duplicates”. Again, users should never duplicate assets in this way, but even in a rare case duplication happens like described, this tool can find and fix the duplicates. Running this tool will print results to the console.

Upgrading from v1.0 or v1.1 to v1.2

Users already having dialogue trees should run the upgrader found at "Window/Dialogical/Upgraders/Upgrader to v1.2" before continuing work. The only change the upgrade does is correct the widths of the ChoiceNodes. If you don't have choice nodes in any of your dialogues you don't have to run the upgrade. Results of the upgrade will be printed to the console.

Upgrading from v1.3 to v1.4

Users already having dialogue trees should run the upgrader found at "Window/Dialogical/Upgraders/Upgrader to v1.4" before continuing work. This will prepare your existing dialogues for work with new features. This wizard should be run only once per project, but running it multiple times will produce no side effects. It is always recommended to backup your assets (ones in Dialogues folder in this case) before any upgrade.

Changing Advanced Settings:

If you would like to modify fonts and colors, you can do so by modifying the GUISkin asset at Assets/Dialogical/Resources/DialogicalSkin.guiskin/CustomStyles. These options would, naturally, be overwritten with new updates.

`Dialogue.cs` has some useful options that you can set from your code:

`doPlayAudio` – should the audio be played if it's attached to the node? Disable this to prevent automatic creation of Audio Sources on the object holding the script.

`_requireReceiver` – if there is a method missing at runtime, do you want exceptions from Send message in the log? Default is no. If you do, set it to `SendMessageOptions.RequireReceiver`.

`substituteTextParams` – Set it in your events, for text params substitution.

`currentLanguageIndex` – Set it in your code, current language index.

`normalPlayDelay` – this is a delay before every node's audio starts playing. Added to this is specific node's audio.

`conversationIsOver` – Is this particular conversation over?

`Instance` – Gets the instance of the latest conversation.

`Tree` – Dialogue Tree assigned in the inspector.

`Activate()` – Call this to start conversation from start node.

`GetNodeText()` – Gets the text of a current node.

`GetNodeOptionsNumber()` – Returns options count.

`GetNodeOptions()` – As explained, returns Conversation Option list. Each returned result is of `ConversationOption` type.

`GetNodeOptionAt(int index)` – Returns only one option with a specified index from the list above.

`CallOption(ConversationOption option)` – Explained, Calls the option supplied.

`PlayNodesAudio()` – Starts the playback of audio clip of a current node.

`GetNodesAudioClip()` – Returns just the Audio clip of a current node.

`NotShown` – Shorthand method for a hidden option.

`SetAutoChoiceDelay(float delay)` – Sets the delay before autochoice.

There are some other less important settings exposed in the code. Open the code file and edit the value, then re-run the extension. Those variables are at the top of the script.

Script Name:	**Variable Name:**	**Meaning:**
DeleteDialog.cs	deletionColor	Color of the rectangle drawn when deletion dialog is on.
DialogueEditorWindow.cs	longReachModeEnabled	Enables long reach mode when holding Ctrl. True by default.
	_longReachMultiplier	How fast should the scrolling happen when in this mode? Default is 2. Can be increased for large nodes.
	_minEditorSize	Minimal size of the window.
	_dblClickCreatesNode	Is double-click for node creation enabled?
	_showAllMenuButtons	Should the part of the main menu for node creation and deletion be shown?
	_assetCreationPath	Default path for creation and Loading of dialogues. You must change this if you change your paths.
	_performanceMode	Repaint frequency (apply settings to main view from minimap, smooth dragging of nodes): 0 - very low (no repaint before focus), 1 - low (10fps), 2 - normal (100fps); Lower this if you're having performance issues.
DialogueMinimapWindow.cs	_connectedColor _connectedFromColor _connectedToColor _notConnectedColor _choiceNodeColor _visibleAreaColor _canvasAreaColor	Coloration for nodes of the minimap and minimap canvas background. See Legend.
DialogueMinimapWindow.cs	**_performanceMode**	Repaint resolution (apply settings here from main view): 0 - very low (no repaint before focus), 1 - low (10fps), 2 - normal (100fps); Lower this if you're having performance issues.

DialogueNode.cs	_createDefaultOption	When new Node is created, should it have a default option "Continue"?
	_allowCopyPasteNodeText	If we allow Copy/Paste of Node text, then on first click inside the node's text everything is selected. This is due to how unity controls work.
DialogueNodeOption.cs	**_allowCopyPasteOptionText**	If we allow Copy/Paste of Option text, then on first click inside the option's text everything is selected. This is due to how unity controls work.
DialogueOtherNodes.cs	_nodeTitleBaseColor _nodeTitleSelectedColor	Color of the title label on the node when it's not selected; and when it's selected.
Helpers	_connectingColor _nodeConnectionColor _choiceConnectionColor _minimapNodeLinkColor _minimapChoiceLinkColor	Coloration for drawing connections.

Further steps and importing in your project:

By now you should know all you need to start using this extension proficiently and efficiently. Export from this project and import in your project. What you need is:

1. Everything in the Dialogical folder, except the contents of the Dialogues folder (you have to have that folder though, so create it manually or delete an import from it) and 'Dialogical Test Scenes' folder.
2. Any other scripts from example scenes if you want to work off of them.

Best of luck on your projects,

- Gru