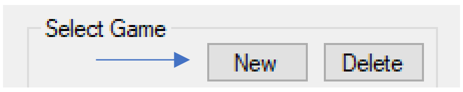
Text Adventure Creator Documentation

"Text Adventure Creator" is a tool for creating modern text adventures through a flowchart-style interface. This document will guide you through the process of creating a simple text adventure with the tool to get you started.

Contents

Creating a new game instance

To start your new game project, you'll first need to create an instance of it within the tool. Open "Text Adventure Creator" and press the "New" button.

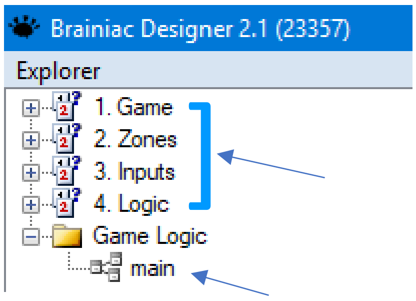


You will then be prompted to enter four things:

1. **Game title** – this is a unique title used to identify your game within the tools. While this cannot be changed directly, the title displayed in-game can be edited at any time.
2. **Game developer** – the name displayed in-game as the developer of the text adventure. Ideally your name or your studio's name. This can be changed at any time.
3. **Invalid input response** – the text shown in-game when the user enters something that your logic isn't set up to handle. This can be changed at any time.
4. **Disabled input response** – the text shown in-game while the system is typing a response to the user and input is disabled. This can be changed at any time.

Once you've entered this information, your new game will be created and selected in the tool's dropdown.

Editing game logic



With your game instance selected in the tool's dropdown, press "Edit Logic". A new window will open – when it does, double click on "main" in the top left. This is the logic tree for our current game. All new game instances come with a demo to show you the standard for node layout and tree structure. Click on the "+" next to "1. Game", "2. Zones", "3. Inputs" and "4. Logic" in the top left to see all available nodes.

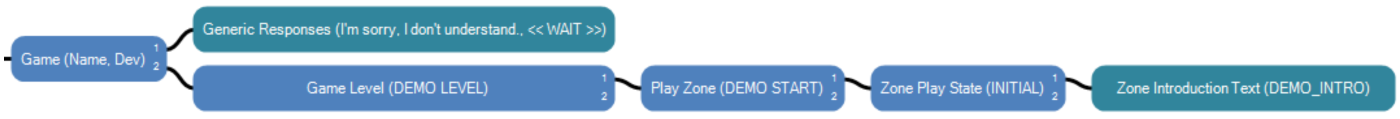
The logic editor has a series of features and shortcuts. Outside of the regular copy/paste, delete and zoom/drag controls as you would expect, there are a few additional features accessible by holding SHIFT when doing so.

A comment can be added to a branch of nodes by clicking a parent node and typing the comment in the node's parameters with a choice of colours available. These do not affect the compiler and are useful to keep track of in-progress features and potentially cumbersome logic branches.

It's important to understand the different node types, the logic they provide and also the tree structure before editing your game. Read on to find out more.

Core game structure nodes (game, level, zone, state)

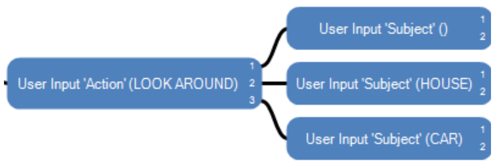
These nodes make up the core of the game, defining information and structure. This is essentially the hierarchy of the game – GAME, LEVEL, ZONE, STATE. Zones are made up of states, levels are made up of zones, the game is made up of levels.


Game (Name, Dev)
Generic Responses (I'm sorry, I don't understand., << WAIT >>)
Game Level (DEMO LEVEL)
Play Zone (DEMO START)
Zone Play State (INITIAL)
Zone Introduction Text (DEMO_INTRO)

- **Game** – this is the initial starting point of any tree. Make sure it is placed first. Provides two parameters:
  - **Title** – this is what you entered when creating your game instance – the game title.
    - Be aware this will only change the title in-game, and not in the tools.
  - **Developer** – this is what you entered when creating your game instance – the game's developer.
- **Generic Responses** – this should be linked after the **Game** node, it provides two parameters:
  - **Invalid input** – this is what you entered when creating your game instance – the response for an input that isn't supported by your logic.
  - **Disabled input** – this is what you entered when creating your game instance – the text to display when the input is disabled.
- **Game Level** – this should also be linked after the **Game** node. A game can have as many levels as you like, they are loaded in sequence so the level at position "2" on the **Game** node will be the default level. This node provides one parameter:
  - **Level name** – the name of this level to use as a reference when moving the player. Must be unique.
- **Play Zone** – this should be linked after a **Game Level** node. A level can have as many zones as you like, they are loaded in sequence, so the zone at position "1" on the **Game Level** node will be the default zone. This provides one parameter:
  - **Zone name** – the name of this zone to use as a reference when moving the player. Must be unique.
- **Play Zone State** – this should be linked after a **Play Zone** node. A zone can have as many states as you like, they are loaded in sequence so the state at position "1" on the **Play Zone** node will be the default state. This provides one parameter:
  - **State name** – the name of the state to use as a reference when moving the player. Must be unique.
- **Zone Introduction Text** – this should be linked after a **Play Zone State** node. This defines the text you see upon entering a zone at a certain state. Note this intro text is placed in each state, not in the zone itself. This provides one parameter:
  - **Intro text** – this is a localised string for the introduction text. Enter the string ID here from the localisation editor – do not enter plain text, it will not be interpreted by the compiler.

Game user interaction nodes

These nodes all begin the logic branches within a zone state. Both must be defined, even if the subject is not required to trigger the logic. Multiple subjects can be defined for one action to trigger different logic for different subjects.

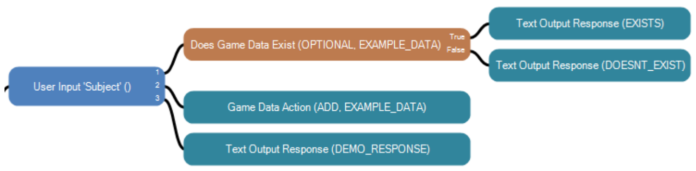

User Input 'Action' (LOOK AROUND)
User Input 'Subject' ()
User Input 'Subject' (HOUSE)
User Input 'Subject' (CAR)

- **User Input 'Action'** – this is the foundation of the input logic and should be linked after a **Play Zone State** node. This node defines the "action" text that the user must type to trigger this branch of logic. Provides one parameter:
  - **Input action** – the "action" text from the user's input - for example, if the user inputs "GO TO HOUSE", the "action" text here would be "GO TO". It's best to keep these "action" triggers uniform throughout your game to make it easily understandable for the player.
    - Examples include: "GO TO", "LOOK AROUND", "OPEN", "USE", "LOOK AT"
- **User Input 'Subject'** – this is another core foundation to the input logic and should be linked after a **User Input 'Action'** node. Be aware you can link multiple "subject" nodes to an "action" node, this stops you needing multiple "action" nodes defining the same string. This node defines the "subject" text of a user's input to trigger this branch of logic. Provides one parameter:

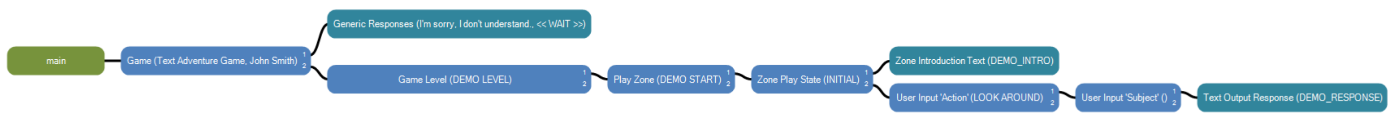
- **Input Subject** – the “subject” text from the user’s input – for example, if the user inputs “GO TO HOUSE”, the “subject” text here would be “HOUSE”. Subject can be left blank – but this node still must be placed.

Game logic and response nodes

These nodes all define the logic to perform once the input conditions have been met through the “action” and “subject” nodes.


User Input 'Subject' ()
Does Game Data Exist (OPTIONAL, EXAMPLE_DATA)
True
False
Text Output Response (EXISTS)
Text Output Response (DOESNT_EXIST)
Game Data Action (ADD, EXAMPLE_DATA)
Text Output Response (DEMO_RESPONSE)

- **Text Output Response** – this outputs text to the user in the game’s response area and should be linked after a **User Input ‘Subject’** node. Provides one parameter:
    - **System response** – this is a localised string to output to the player. Enter the string ID here from the localisation editor – do not enter plain text, it will not be interpreted by the compiler.
- **Does Game Data Exist** – this is a decision node with a true/false output. It runs logic based on the user’s “game data” and allows for more depth to the game logic. Only one logic query can be performed per logic branch. This node provides two parameters:
    - **Data requirement** – this determines if the “game data” identifier specified on this node is OPTIONAL or REQUIRED. If OPTIONAL, any other logic on the input’s logic branch will run as well as this node’s output no matter if the user has the specified “game data” or not. If REQUIRED, the output of this node will run, but any other logic will be disabled on the branch if the user doesn’t have the specified “game data”.
    - **Game data** – this is the identifier for the “game data” which is either REQUIRED or OPTIONAL. This identifier must be consistent with any other mentions of this “game data” in other nodes.
- **Game Data Action** – saves data to the game’s memory, which can then be queried by the **Does Game Data Exist** node to cause dynamic responses to player actions. You can have as many game data actions per logic branch as you wish. This node provides two parameters:
    - **Data action** – this specifies if the node should cause the game to save or remove your specified “game data”. Select either ADD or REMOVE to do the associated action.
    - **Data name** – the name of the “game data” to save to memory, this is an identifier and therefore must be unique. Use the identifier to query its existence in future.
- **Move To** – this node is key to progressing the game and navigating the player through the world. It can move players to new zones, states or levels. It provides two parameters:
    - **Location type** – are you moving the player to a new zone, level or state? Select the appropriate one.
    - **Location name** – the unique name of the new area or state to move to. Must be consistent with spelling and case of existing area/state definitions.
- **Referenced Action** – this is a good node to improve user experience. It references an existing logic branch to allow for multiple commands with the same action. It provides two parameters:
    - **Input action** – the action node to look for to reference behaviour from.
    - **Input subject** – the subject node to look for branched off of the previously defined action node to reference behaviour from.
- **Game Over** – this node can be added alongside any other logic on a node and will trigger a win/lose state in the text adventure. You do not need a game over node, however without it your adventure will run indefinitely. It provides one parameter:
    - **Finish as** – this allows the game to understand if you are finishing the game as a WIN or LOSE state.

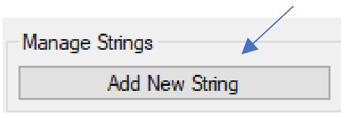

main
Game (Text Adventure Game, John Smith)
Generic Responses (I'm sorry, I don't understand., << WAIT >>)
Game Level (DEMO LEVEL)
Play Zone (DEMO START)
Zone Play State (INITIAL)
Zone Introduction Text (DEMO_INTRO)
User Input 'Action' (LOOK AROUND)
User Input 'Subject' ()
Text Output Response (DEMO_RESPONSE)

Failure to follow standards will result in a compiler error and at worst game crashes. Always double check your spelling of area/state names and “game data” identifiers. Make sure to follow the structure lined out in the documentation and do not, for example, create a zone with no states.

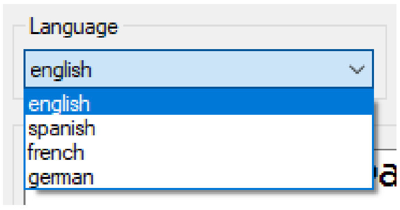
Localising text for your game

Alongside editing your logic, you'll want to use the localisation tool to create strings that you can use in-game. To do this, on the main tool window, select "Edit Localisation". In this new window you will be presented with a list of existing strings in your game. The demo project comes with two – "DEMO_INTRO" and "DEMO_RESPONSE". Fire up the logic editor to see where these are used.

To add your own strings



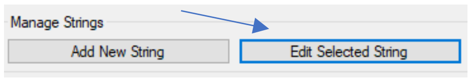
1. Press "Add New String" within the localisation tool.
2. In the popup that appears, you can enter a unique identifier for the string you are about to create. This is the "string ID" which you reference within nodes on your games logic tree to call the text. Strings can be referenced as many times as you like, once they're created, they're totally reusable. Click "OK" when you've entered the identifier. Note that a string ID cannot be changed once created.
3. You're now presented with the string editor. This window defaults to English unless you are editing a string with a language already selected (see later). Within the editor, you can enter a string for English, Spanish, French and German.
    a. Change language by selecting it from the dropdown in the top left. Swapping to a new language will reload the string input box – your previous language entry will be saved into temporary memory and will be visible if you select that language from the drop down again. Nothing is saved permanently into the game until you press "Save and Close".



    b. Enter your string into the string input box. When you reach the end of a line, **manually break the line with the enter key**. Line wrapping is not performed automatically and must be done manually. Failure to do this will result in strings extending over the edge of the game window.
    c. If you do not wish to localise the current string into other languages, you can press "Apply Current To All" and the currently selected language's string will be used across all languages. Again, this does not save the string into permanent memory.
    d. When you're finished with your string, press "Save and Close" to save the string. Failure to press the save button will lose your work.
4. The main localisation tool window will now update to show your new string in the list.

To edit an existing string

1. Select your string within the localisation tool, either by its text or by its ID.
2. Press "Edit Selected String" and the editor window will appear. The editor will launch on to the language you had selected in the localisation tool main window for ease when editing a batch of strings across languages.
    a. Follow the steps of to add your own string to understand how to use the string editor.

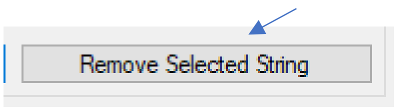


3. Upon saving you'll be returned to the main localisation tool window where you can see your edited string.

A handy shortcut for games with a large number of strings – press "Add New String" and type the name of the string you're trying to find with matching case. Upon pressing OK the existing string will be loaded into the editor.

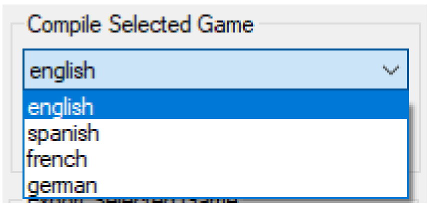
To remove an existing string

1. Select your string in the localisation tool, either by its text or by its ID.
2. Press "Remove Selected String" and confirm when asked.



Compiling and playing your game

You've added your logic and you've added your strings… or maybe you've just instanced a new game and want to try it out. Before playing you will need to compile your game to process its logic and strings into a format readable by the game's engine. Compiling is super simple and as long as you've not made any logic issues, should take a few seconds.



1. Select the language you wish to compile to in the dropdown of the main creator tool window. Make sure your logic tree is saved at this point! Press the save button in the top right of the logic editor window if it is still open.
2. Press compile!

The process should be as simple as that. If you encounter any errors, please refer to the game logic documentation as you have likely structured your logic tree incorrectly. Any localised strings that couldn't be found by the compiler will be highlighted in-game.

To play your game, in the export section, select the game version you would like and press "Play". Text adventures are available in a variety of formats through different engines – the current play options are:

- 2D
  - ASGE – a lightweight 2D OpenGL framework with only a few files to distribute.
  - Unity – a slightly bulkier 2D implementation in the game engine Unity.
- 3D
  - UE4 – a very bulky 3D implementation complete with models in Unreal Engine 4.

As well as playing your game to test it out, you can also press "Export" to save all relevant files for the currently selected play option for distribution.