Dialogue Editor - 1.0.0
Technical Documentation

Summary

1 - Overview

The goal of this editor is to allow quick iterations on dialogues writing, and to provide the necessary tools for gameplay flow and data exports.
It's inspired by the tools used on Cyanide's Game of Thrones RPG, and the Aurora tool used on Neverwinter Nights.

The main functions of this editor pipeline are :
- edition of the dialogues trees in a standalone editor.
- exports for the targeted game engines and localization, voicing, and any external tools.
- samples for easy integration and iteration in game engines, with both editors running in parallel.

2 - Project content

2.1 - Directories

- Build : the generated binary builds (release) of the demos.
- Dev : the Visual Studio solution.
- Dev/External : the external dependencies' dlls.
- Docs : the documentations.
- Icons : the working directory for icons.
- ProjectDefault : the default "hello world" project assets.
- ProjectDemo : the demo project assets.

2.2 - Visual Studio Solution

When you open DialogueEditor.sln, you will find several projects inside :

- DialogueEditorDll : the main part of the solution, containing all the editor's components.

- DefaultBuild : the "hello world" build demo. You can use it to open the ProjectDefault/HelloWorld.project, and use it as a base for your project build.

- DemoBuild : the sample build demo. You can use it to open the ProjectDemo/Demo.project, and use it as a reference when creating your project build.

- TestSingleWindow : a small build test using a single dialogue view with its properties. This is intended as a base for projects willing to integrate the editor inside another tool.

3 - Design

- Unique Files names : all dialogues are referenced by their names, and thus must all be unique accross the whole project. This ensures that all the files can be imported as a single big pack of assets in a common folder in any game engine. It could also allow dialogues to reference each other wherever they are stored in the hierarchy.

- Directories : paths are not stored in the resources or the project. When the editor opens a project file, it will parse the adjacent folders and look for any .dlg files they may contain to fill the ResourcesHandler and the ProjectExplorer. This allows the users to move the files outside of the editor through any versionning software, such as svn or mercurial, without breaking anything in the editor. All they will need to do is reload the project.

- Nodes IDs : dialogue nodes are stored as a flat list, in their order of creation. They will receive a incremental numeric ID upon creation. Those IDs are meant to be unique for each node inside a dialogue, and must never be modified through a node lifetime.
The dialogue tree is defined by references between the nodes, through properties like NextNodeID, RepliesIDs, GotoID and BranchID.
This ID is also used to have a consistent VoicingID and translation ID during production.
When moved accross the tree, the nodes will not be moved in the serialized flat list. This limits the risks of conflict when using merge tools through svn or mercurial.
This means that after some time, nodes IDs will appear as if they are disordered on the dialogue tree view. This is perfectly normal and intended.
IDs are not indexes !

- CustomLists : this is a centralized dictionnary of dictionnaries. It is used as a helper to manipulate static lists of data throughout various parts of the editor.
Several helpers already exist to manipulate fields associated with a specific CustomList, such as fields decorators converting string properties into a combo box (look for PropertyCustomListConverter).
By default, five lists are registered as samples for necessary project properties (scene types, actors species, actors genders, actors builds, and voicing intensities).
Those lists can be overriden or completed, and additional lists can be added in your project build (see part 4.1).

- Nodes attributes : most nodes properties are built-in and specific to each type of node, except for nodes attributes. Those are declared in your custom build and can be instanciated on every node. There are three types of attributes, each with a different goal.
Conditions are used to decide if a node should be executed or skipped.
Actions are triggered at the beginning or end of a node execution.
Flags act as static parameters that can be read during the node execution to provide more information.

- Voicing ID : This ID is generated and stored in the .dlg for every sentence, in the form "DialogueName_NodeID". It is exported inside the Wwise text helper file and the voicing spreadsheets. Sound files recorded should match this ID in order for your game to dynamically find them during gameplay execution.

4 - Code

4.1 - Libraries

The editor is written in standard C#.
It's based on several libraries :

- WinForms for the UI (Native with Visual Studio) :
https://msdn.microsoft.com/en-us/library/dd30h2yb%28v=vs.110%29.aspx

- Json.Net (Newtonsoft) for the json serialization (provided dll) :
http://www.newtonsoft.com/json

- DockPanel Suite (Weifenluo) for the UI panels docking (recompiled dll) :
http://dockpanelsuite.com/

The provided builds are compiled with Visual 2013, using .Net 4.5, other versions have not been tested.

4.2 - Naming conventions

Dialog and dialogue disambiguation : a dialog is a popup, while a dialogue is a conversation (uk english).

Node properties : this term references all the serialized parameters of a dialogue node.

Node attributes : this term is used to reference nodes conditions, actions, and flags under a common term.

Concerning the code, all methods and public variables use UpperCamelCase, while private variables use lowerCamelCase.

4.3 - Architecture

- EditorCore.cs is a static class holding all the project's static resources and callbacks overrides. It also provides access to most of the editor components, such as windows, panels, logs, and settings.

- ResourcesHandler.cs is a static class holding all the serialized project resources (dialogues and project). It provides the necessary methods to manipulate and serialize the project's files.

- EditorHelper.cs is a set of static methods, classes and decorators used to manipulate various parts of the editor's component, such as manipulating listviews and treeviews icons or checking errors in fields and resources.

- EditorSettings.cs is a serialized class used to store the user's local settings.

- EditorVersion.cs is a static class holding the editor's version numbers.

- Extensions.cs is a set of extensions for existing classes in the standard language and WinForms.

- Utility.cs is a set of static methods used as helpers for the standard language, such as helpers to manipulate file paths or datetimes.

- WIN32.cs is a set of static methods used for low-level operations.

- Controls/ contains custom Winforms controls used by other forms.

- Dialogs/ contains all the dialogs (popups) used by the editor.

- Documents/ contains the forms used to view and edit the project's resources. They will be used as docked document in the center of the MainWindow.

- Exporters/ contains static classes for the various import/export formats used by the editor's tools.

- Panels/ contains the panels docked inside the MainWindow.
PanelProjectExplorer.cs handles the files hierarchy and gives access to the project's resources.
PanelOutputLog.cs handles the log messages sent by other components.
PanelProperties.cs handles the list of editable properties associated with the selected dialogue node.
Properties/ handles the different edition panels dedicated to each type of dialogue nodes. They will appear inside the PanelProperties.

- Resources/ contains the classes used for serializing all the project's data.

- Windows/ contains all the top-level forms.
WindowMain.cs is the main window of the editor. It holds the docking area and the top menu.
WindowViewer.cs is a basic dialogue parser, used to test how a dialogue plays out.

5 - Integration

5.1 - Editor build

To start using the editor efficiently with your project, you will need to create a custom build. This will allow you to declare custom attributes for your nodes, as well as injecting custom data into various properties.
As a first step, you can use the DefaultBuild to prepare the integration with your game. You will be able to create actors and interactive dialogue trees, but you won't have access to any actions or conditions associated with your nodes.

You can use the DefaultBuild as a base for your own build, and use the DemoBuild as a reference when iterating on it. You can declare all your overrides in your own Program.cs, and declare your own node attributes classes (conditions/actions/flags) in an adjacent folder.

5.1.1 - Node attributes

To declare a node attribute, you will first need to declare the class itself, with *NodeCondition*, *NodeAction* or *NodeFlag* as a base (look for *NodeConditionHasHonor* as a sample).
You can declare as many public variables on your new condition, it will be serialized inside your dialogues.
You can provide an override to the *GetDisplayText_Impl* and *GetTreeText_Impl* methods, they are used when displaying the attribute in the tree view and on the node properties edition panel.

To integrate the new attribute in the editor, you have to bind it in the editor menus and serializer. To do that, you can use this method in your Program.cs :

EditorCore.BindAttribute(typeof(NodeConditionHasHonor), "ConditionHasHonor", "Has Honor");

The first string parameter will be used as the serialized type name.
The second string parameter will be used as text for the context menu option.

You can use decorators on your node attributes to define custom behaviours when editing their properties. A decorator already exists for Actors (*PropertyCharacterNameConverter*), as well as a base for CustomLists (*PropertyCustomListConverter*) which needs to be overriden for the lists you want to use. Those can be used as a reference for your own decorators.

Declare like this :

```
public class PropertySkillNameConverter : PropertyCustomListConverter
{
   public PropertySkillNameConverter()
   {
      CustomListName = "Skills";
   }
}
```

Use like this :

```
[TypeConverter(typeof(PropertySkillNameConverter))]
public string Skill { get; set; }
```

5.1.2 - Custom data

You can edit the version number of your project build by editing *EditorCore.VersionProject*.
This project version number will be stored in the header of all the saved files, next to the editor version number.

You can plug a delegate that will be called when the project is loaded through *EditorCore.OnProjectLoad*.

Animations are defined through *EditorCore.Animations*. It's a dictionary of animsets defined by their name and the list of animations they contain. You can also use the delegate *EditorCore.GetActorAnimsets* to fill the animset combobox depending on the actor.

To inject lists of values associated to particular properties (a bit like enums), you can use *EditorCore.CustomLists*. It is aimed to store a list of named dictionaries, with keys being used for serialization, and values used for the editor display.
You can either modify/replace existing lists (some are used to customize actors), or add your own new lists. The advantage is that some methods exist to make their edition easier through decorators, but you could have your own datasets and decorators instead.

The delegate *EditorCore.OnCheckDialogueErrors* can be used to log additional information when the "Check Dialogue" command is used.

To add your own menu on the top menubar of the editor, you can declare it in Program.cs then add it through *EditorCore.MainWindow.AddCustomMenu*, like this :

```
ToolStripMenuItem menuItemImportGameData = new
System.Windows.Forms.ToolStripMenuItem();
menuItemImportGameData.Text = "&Import Game Data";
menuItemImportGameData.Click += delegate
{
	//...
};

ToolStripMenuItem menuItemGame = new System.Windows.Forms.ToolStripMenuItem();
menuItemGame.Text = "My Game";
menuItemGame.DropDownItems.AddRange(new System.Windows.Forms.ToolStripItem[] {
	menuItemImportGameData,
});

EditorCore.MainWindow.AddCustomMenu(menuItemGame);
```

5.2 - Game integration

To use the dialogues in your game, you will need to implement two main parts :
- the dialogue parser, to read the json files in your game.
- the dialogue interpreter, to run the dialogues in your game.

5.2.1 - Dialogue parser

To write the parser, you can use the description of the files structure (see part 3.5).
Unreal provides a json parsing helper, and the json.net library used in this editor can be used in any engine using csharp, such as Unity. Solutions for json parsers ion other platforms should not be a problem to find.

The parsing will essentially consists in a loop on the ListNodes. Each node and attribute will have its type defined by the property "$type", allowing you to instanciate objects matching those types.

5.2.2 - Dialogue interpreter

To write the interpreter, you can look at the WindowViewer inside the editor for inspiration.

The flow of a dialogue starts from the root node, then plays the node referenced by the NexNodeID property, then the NexNodeID again and again until there is no next node, meaning the dialogue is finished.
If a node's conditions are false, the node should be skipped, and it's NextNode should be played instead.
The node's actions should be played at the beginning and the end of its execution (bool property OnNodeStart).
Special nodes can fork the flow of the dialogue. Those nodes should not use their NextNodeID property unless they are skipped because of their conditions being false. Those nodes include Choice, Goto and Branch, and will use the properties RepliesIDs, GotoID and BranchID respectively.

Sentences will be the core of the execution. They hold the displayed text, as well as informations about the speaker and the animations.
The speaker is the actor actually speaking the sentence's text. The listener is the actor looked at by the speaker.

Your design should include a central manager for all the dialogues played. It would be in charge of starting, updating and closing the dialogues, as well as handling priorities on UI and voicing.
An interactive dialogue should always be unique and have top priority, while several ambiant dialogues may be played simultaneously.

Credits

This editor was developed as part of the Cthulhu project at Cyanide.

Author :
Adrien Cambon "Legulysse"

Additional programmers :
François Bogaert
Antoine Lemaire

Technical director :
Vianney Lançon