Basic Documentation for *ConText – A Text/Choice Adventure Game Framework*

This brief documentation attempts to explain the core features and structure of the framework.

It is by no means final or as detailed as possible. Feel free to suggest additions you'd find helpful.

Module system

The fundamental idea behind the framework is to provide a flexible and modular system for creating text/choice adventure games. In order to achieve this, messages are based on modules. There are four types of modules in the framework: Text, Image, TicTacToe and Reply Module. Their parent class is ModuleBlueprint. The basic implementation of a module contains a module ID comprised of four integers (hierarchy ID, branch ID, sequential ID, subpart ID) which uniquely designates each module, the sending character of the message, the content of the message, a previous as well as up to several next modules and a log entry.

Additionally, each module type is mapped to a specific UI template in the UI settings.

Depending on the individual implementation, certain details may change, such as the type of the message's content or the number of next modules.

The way the modules work in playback is as follows: The module manager triggers the first module of the story and as long as no user input is required will keep firing the respective next module until some form of user input is expected (or until a module for some reason breaks the cycle and does not tell the module manager which module is next). At that point, the automatic stream is stopped and has to be restarted by whichever user input is expected. This for example happens with the Reply Module, where pressing any reply will start the automatic stream again with whichever module was coupled with that reply. Whenever a module is fired and has an attached log entry, the respective entry is added to the log list or updated if it already is an element of the list.

Managers

The entire eventual game and parts of the framework use several manager classes to control actions, the UI, etc.

The Module Manager handles most manner of actions regarding the modules, such as the automatic stream, initial loading of existing and saving of new story progress on a superficial level and tracking of all currently instanced modules.

The Log Manager does likewise for the log entries.

The State Manager provides lower level functions for handling save files and tracks the current game state.

The UI manager handles instancing of modules and their representation in the game as well as switching of the three main screens.

Unify is not a manager class per se, but unifies access to all managers into one class.

Story Settings and Characters

Characters and story settings are simple classes and mostly used for data storage. Characters each have their name, color of the messages, background image for messages (i.e. for more detailed message bubbles), a character ID and message bubble alignment.

The Story Settings contain a list of characters and a corresponding string list of just their names (for internal use).

UI Settings

The UI Settings store a range of data like the font and font size for modules, font, font size and color for the main screens as well as the list of pairings between module UI templates and module scripts. (Note here: due to internal constraints, these pairings consist of the template as a GameObject and the script represented by its class name only. If you want to add a custom pairing of your own, make sure the string value is exactly the class name of the intended class.)

Custom inspectors

The framework contains a custom Unity Inspector for each type of module as well as for UI Settings, Story Settings, Characters and log entries. These inspectors actually implement a lot of logic as they apply data and settings considering existing conditions and constraints.

The inspectors for modules have a parent class called ModuleInspectorAncestor which is intended to be a centralized place for handling creation of modules depending on a selected type. When you create a custom module class of your own, it is advised you expand the two provided functions in that class as well as the ModuleTypes and ModuleTypeEnumDescriptions enums in the Module Manager. That way your module will be available for selection in each module inspector.

Custom module classes

When creating a custom module class of your own, you should make it inherit from the ModuleBlueprint class and make sure to implement and thus possibly override all functions provided in that class. See the comments/descriptions for these functions in the class itself to learn what they do.