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.

This documentation does only cover topics specific to this framework, features it adds on top of what Unity can do in itself. Thus, if one wants to know more about basic scripting and concepts of Unity, it's advised one look at the official Unity documentation and tutorials as well, to be found here:

https://unity3d.com/learn/tutorials

https://unity3d.com/learn/tutorials/topics/scripting

https://docs.unity3d.com/ScriptReference/

As well as perhaps the largest unofficial documentation: http://wiki.unity3d.com/index.php/Scripts

Topics of this documentation:

1. ConText Options Back to top

The ConText Options screen is intended to provide quick access to the most important core features of the framework right from the main screen. That includes

shortcuts to the **Game settings** comprised of

Game UI Settings,

'Character & Story' settings and

'Player Settings', as well as

shortcuts to **Story** related screens like

the **first story module**,

the **latest story module** (as far as possible, with branching storylines this will only return the single highest branch),

an option to attempt to **fix IDs** of the active storyline (which is experimental and will only work correctly given the function behind it is implemented correctly in each module's code) and

to **reset the save file** (which should be used whenever elements in the existing storyline are changed).

2. Module system Back to top

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**, however the subpart ID is not set manually**)** which uniquely designates each module,

the **sending character** of the message,

the **content** of the message,

a **previous module**

as well as **up to several next modules** and

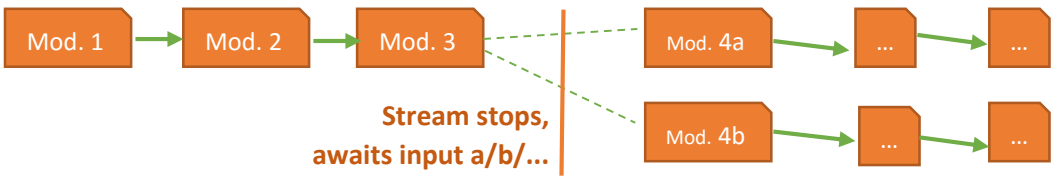
a **log entry**.

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

(Additionally, each module type is mapped to a specific UI template in the UI settings (4.).)

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 a module does not tell the module manager which module is next (e.g. to await user input). At that point, the automatic stream is stopped and has to be restarted by that latest module (e.g. with the Reply Module, where pressing any reply will start the automatic stream again with whichever module was coupled to 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.



3. Managers Back to top

The entire eventual game and parts of the framework use several manager classes to control actions, the UI, module stream, logs, game state.

The **Module Manager** handles most manner of actions regarding the modules, such as the upkeep of the automatic stream, initial loading of existing and saving of new story progress on a superficial level, tracking of all currently instanced modules as well as resetting previously fired modules when resetting the save file within the game.

The **Log Manager** does likewise for the log entries.

The **State Manager** provides lower level functions for creating, loading and deleting 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.

4. Characters & Story settings Back to top

Character is mostly used for data storage. Characters each contain their **name**, **messages' color, background image** (i.e. for more detailed message bubbles) **and alignment**, as well as their **character ID** (which until now is not specifically used).

The **Story Settings** (or 'Character & Story' settings) contain a **list of Characters** and a corresponding string list of just their names (for internal use), as well as both a variable for the **default message sound** to be played whenever a message is triggered and for a **background track**, which will be played on loop while the game is running.

5. UI Settings Back to top

The UI Settings store a range of data including

the **font and font size for modules**,

the **font and font size and color for the three 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.)

When viewed through the Inspector, the **Module UI properties** segment of the UI Settings includes a list of **Module Templates**. This should contain all prefabs that represent a module's UI template.

The **Module UI templates** segment at the bottom shows list of pairings between module UI templates and module scripts. 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.

6. Custom inspectors Back to top

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 logistic 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 common functionality among the inspectors.

The class specifies the **OnEnable** function in its basic form, defining the labels for each section of the inspector as well as initializing the foldout status variables. This should ideally only be extended, not overridden for custom inspectors.

The class specifies the basic structure of the **OnInspectorGUI** function, consisting of six functions describing the six parts of the inspector layout. These parts are **PartInfo, PartPrevious, PartMessage, PartLog, PartNext, PartDelete**. Each of them is hidden in a foldout to give the inspector a cleaner look.

> **PartInfo** includes basic utility functions such as Expand/Collapse all foldouts, toggle hints, and should be overridden and extended should one wish to display additional description or utilities at the top of the inspector.
>
> **PartPrevious** includes any functionality related to the previous module, which by default is display of the object field specifying the module and a button to navigate directly to that module.
>
> **PartMessage** by default holds only functionality to display an object field for the message sound and is the most likely to be overridden in another custom module inspector. It should contain all functionality regarding the message content such as the character, the message/module ID and of course the content itself in whichever form (e.g. text or image).
>
> **PartLog** provides functionality for linking the respective log entry to a module, similar to how PartPrevious does for the previous module.
>
> **PartNext** works analogously to PartPrevious but for the next module, with the addition that one can choose the type of module to create. This should be overridden when a custom module should offer multiple next modules, as is for example the case for ReplyModule and TicTacToe.
>
> **PartDelete** offers functionality for deleting the currently focused module. By default it assumes one previous and one next module, upon pressing the Delete button will ask to confirm the choice, then connects the two surrounding modules to close the gap. For modules containing multiple next modules, this function should be overridden in order to make sure the surrounding modules can be connected correctly.

When you create a custom module class of your own, it is advised you expand and or override the Part[…] functions as necessary.

In order to make custom modules available for selection in the module inspectors (e.g. as next modules), first the **ModuleTypes** and **ModuleTypeEnumDescriptions** enums in the Module Manager need to be expanded with values (enum ID, name string). Secondly, in the ModuleInspectorAncestor class contains another two functions related to the handling of

custom modules. For one, the **getShortDesc** function aims to provide a short description string for any custom module. It is used to display the little "(name; type)" info in the module inspectors. The other function is **createNextModule**, which is used to create and add actual asset instances of modules as e.g. next modules. It uses a switch statement over the ModuleTypes enum in Module Manager to distinguish which module type to use for creation. It also initializes the handful of necessary variable values such as the sending character and the module IDs.

7. Custom module classes Back to top

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.

It is likely if not certain that the functions one will need to override are

setContent, which is dependent on the actual structure of the UI template, and is supposed to put whatever content is specified for the module into the actual UI instance,

getNextPart, which when called returns whatever is the next module for the automatic stream to fire. Note here, if you want to stop the automatic stream (e.g. to await user input), simply return null.

getModForChoice, which is given the ID of a choice and an **IDChoiceCapsule**. This function is called when loading existing story progress and expects in return the module that corresponds to the given choice ID. For example for a reply module with three replies, getModForChoice(2, […]) should return the second reply's module. The IDChoiceCapsule does not necessarily have to be considered, but can be used to check whether the modules match up (loaded data vs asset)

getHighestModule, which is a quasi-recursive function supposed to return the module furthest in the story line. Essentially, this needs to return the result of the function being called on the next module, or if the next module is not set, return the current module. If multiple next modules are set, query each next module and return whichever result has the largest/highest ID.

fixNextIDs, which is intended to repair the IDs of the story line. It should check for whether there is a next module, if so, set its ID according to the current module's ID (i.e. seq +1, branch and hierarchy remain), then call the function on the next module. Similar when multiple next modules are present (i.e. seq remains, branch increasing with each next, hierarchy +1 across all next modules)

resetModule, which should only be necessary when the module may be split into multiple subparts or contains data that is changed during runtime and needs to be reset for repeated firing (e.g. as in TextModule).

pushChoice may only possibly need to be changed. It is supposed to forward the IDChoiceCapsule corresponding to this module and being created at runtime to the choices list in the module manager. It is unlikely that a user needs to change it. It is called whenever a user's reply/next module choice (which may also be -1 if the module has only a single next

module) is made. In that case, whichever class or instance calls the function so far generates the IDChoiceCapsule depending on that choice on its own and calls the function with it.

8. Export to Android Back to top

If you want to export your game to Android for testing, you should read into the Build process for Unity at

https://docs.unity3d.com/Manual/android-GettingStarted.html

and specifically follow the Android SDK setup instructions at

https://docs.unity3d.com/Manual/android-sdksetup.html

Once you have completed those steps, in Unity click on **File** -> **Build Settings…**.

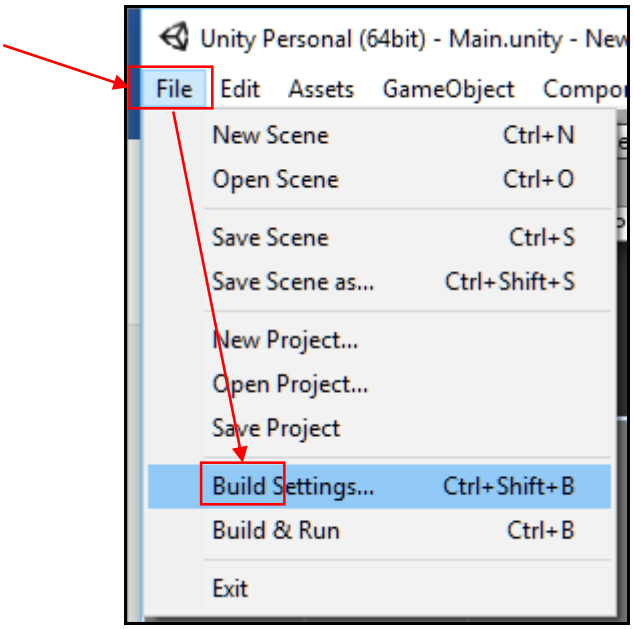


Figure 1: To Build

This will open Unity's build settings window, where you can choose the platform to build for (**Android** since these instructions are about Android). Once **Android** is selected, you may set Texture Compression to ETC which may improve performance a bit. Additionally, while the game scene/level is open in the editor, press Add Open Scenes. This will add the scene to the actual standalone game build.

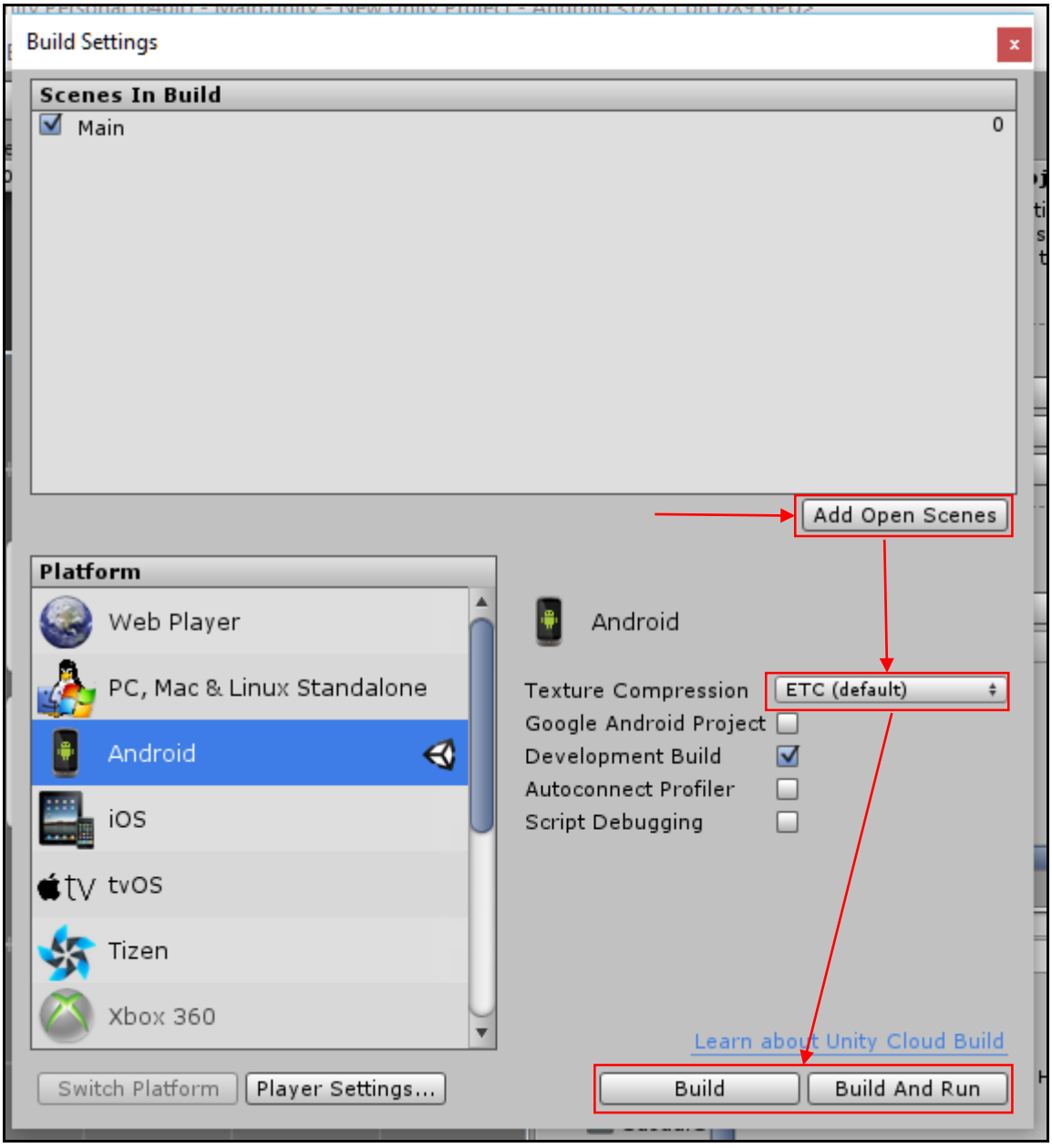


Figure 2: Build Settings

Once this is done, you may click **Build** or **Build And Run**. The difference here is

> **Build**: this will prompt you to specify the name and location of the .apk file that will contain the standalone build of the game. After the process is done, you can move the .apk file manually over to your Android device, install it and then play.
>
> **Build And Run**: this will prompt the same as Build, but after the process is done will attempt to run the game as well right away.

9. FAQ Back to top

Q: What do I do if I want to create a new story?

A: So far there is no way of just creating a new story in the framework except for altering the existing message modules. If you want to create a new story without altering existing assets, you will need to create a new Unity project and import the plugin (and layout file) again, as described in ConText Tutorial Part 1.