DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

ConText – A Text/Choice Adventure Game Framework

Paul Preissner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

ConText – A Text/Choice Adventure Game Framework

ConText – Ein Text/Choice Adventure Game Framework

Author:	Paul Preissner
Supervisor:	Prof. Gudrun Klinker
Advisor:	David Plecher
Submission Date:	15.08.2016

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Garching, 15.08.2016 Paul Preissner

Acknowledgments

Abstract

This thesis encompasses the development, documentation and evaluation of a framework for creating serious choice based text adventure games for mobile devices in Unity. The goal is to provide a tool and template that enables amateur users to create such educational games with ease as well as to provide experienced users with basic functionality that is expandable with individual modules.

Contents

1. Introduction

1.1. Project description

This thesis encompasses the development, documentation and evaluation of the ConText framework. As the title suggests, ConText aims to be a tool that lets the user create text adventures for mobile devices in Unity Technologies' Unity game engine. Considering the ideal usage principles of smartphones and tablets, however, the term "text adventure" needs to be somewhat modified here. The simplest and most intuitive way to navigate interfaces on smartphones is to tap and swipe, as such the games created with ConText fall more in line with choice based adventure games or gamebooks in how they are controlled. The main interest of the chair for Augmented Reality is the usage of this framework for serious, specifically educational games. Additionally, ConText is intended to provide both simplicity so inexperienced users can create a game without the need for specialized knowledge as well as expandability, so developers with existing Unity, C#, JavaScript or otherwise relevant knowledge can use the framework as a baseline and only need to add whichever additional parts they require. While similar solutions do exist already, the author determined that most of them that were readily and freely available either only offered simplicity or expandability, or were only available for programming languages or barebones tools, thus lacking the immediate potential functionality provided through Unity, such as efficient and extensive 2D and 3D rendering, mobile platform integration and active community support. The framework is a plugin made in and made for the Unity game engine in version 5.x. Development first required research on what existing solutions offered in possibilities, on what existing mobile text adventure games offered in features as well as specialist literature and proceedings on the topics of interactive storytelling, serious games and usability. Development included working out an initial concept, implementing it in Unity with C# and continuously reiterating and adapting the concept to stepping stones that turned up along the way. A later stage was user studies, which consists of two separate user studies and their respective evaluation regarding the user experience and usability of ConText. The sections of this thesis appear in the order this introduction set up, first covering research, followed by development and user studies and concluding with an outlook and evaluation of the thesis results.

1.2. Project goal

In the spirit of the project description, the goal of this thesis was to ideally end up with a tool that provides a sufficient baseline for creating mobile text adventures, providing both a simple and intuitive interface for amateur users and a flexible, modular and expandable structure that allows experienced users to add their specific functionality relatively hassle-free. The user studies should provide feedback on what features are prime candidates for included or future implementation and how successful the project was in reaching this goal. Sufficiently reaching the goal would set ConText up for actual public or prolonged use as well as future expansion into a more refined and feature rich framework, topics covered in chapters 4 and 5.

2. Research

2.1. Inspiration

The concept of text adventures, choice based games or gamebooks is by far not new, and dates back long before electrical computers were even invented. They are a type of game that work almost inherently well on any type of device capable of displaying text and accepting input, as that in its most basic form is all they need. However, the question with every type of device remains how that is done intuitively and most true to the user's preferred consumption method with that device. With modern smartphones and tablets, the most popular input methods are tapping and swiping, and extensions of those to perform more complex actions. For a text adventure, this is contrast to the classic physical typing on a keyboard or even the pointing and clicking with a mouse. As such, it is clear a mobile text adventure ideally adapts and adopts tapping and swiping. In the spirit of the aforementioned game genres, that means swiping through texts or other forms of content and tapping items, replies or similar. Another aspect of these types of games has always been the deliberate lack of visual stimuli. Much like the classic understanding of books, they aim to elicit purely imaginary experiences. The user should mentally picture the events, draw their own version of the world, and not be bound to an artist's specific interpretation. One game in particular that we think captures the essence of these two sides very well is the Lifeline series of games developed by Three Minute Games in 2015 and 2015 for Apple iOS and Google Android. The player is confronted with an at the beginning unknown world and one character. Descriptions of the world and events are delivered through the eyes of that character, and all the player can sometimes do is choose between two replies to a prompt by the character. Despite this simplicity, through the course of the game, the player envisions the entire world, the looks, the events and everything as they see fit. At the same time, the game is simple enough to be played even while being mobile with a smartphone in hand and does not require constant attention. This sparked the question about what options are available nowadays for creating this type of game. As looking at related work showed, the readily available options are often lackluster, but also provide valuable insight into what features need to be combined into one tool to provide satisfying overall functionality to both inexperienced and experienced user groups.

2.2. Related work

When analyzing related work, several categories were sampled. For one other popular text adventure games, both mobile and web based, to see what resonates well with audiences so far. Secondly other frameworks designed for the same or similar purposes, creating text adventures, to see what feature sets are available and seen as worthwhile by developers. Lastly, several specialist literary works on interactive storytelling, serious games and usability. Additionally, some functionality was deduced through logical inference and, frankly, common sense, since some basic features are simply necessary for such a framework to function at all. Whenever a game was analyzed, a list of relevant features was extracted and matched up with the existing feature list in order to determine which additions or changes may need be made.

2.2.1. Games

Lifeline The first game analyzed was Lifeline by Three Minute Games on iOS, as it also posed as inspiration for the thesis. What can be seen in the game is that simplicity is key when aiming for accessibility with a large user group of smartphone users of all skill levels. Lifeline offers a simple structure of 1:1 communication between the player and one game character through a sequential text message stream. The interactive part for the player is reduced to the ability of choosing one of two replies when prompted by the game character every few messages. The story is acted out in real time, or rather with 1:1 time scaling. When the character claims to need several hours for an action, the game will accurately remain inactive for that long and only notify the player once as much time has passed. A brief settings menu allows trivial changes such as toggling sounds and music, changing language, and more relevant to the game, to rewind the story to a previously reached decision and to speed up the gameplay by way of reducing UI effects and animation. The player can always scroll back through the entire story and through these settings choose a particular previous state once they have completed one story branch. What could be gathered from Lifeline is that ConText would need at least a basic dialogue system with multiple reply options offered to the user and possibly timed firing, as well as a mechanism to save new and load existing story progress. Additionally, it would need to provide at least a simple UI with potential customization for the user, as well as possibly settings for language support, sounds and the UI.

Storynexus.com Another sample was taken from a web based service called storynexus.com. Storynexus is a website where users can create and more prominently play text adventures through their web browser. The story "The Thirst Frontier" was chosen for analysis. It is mostly reminiscent of choice adventures or choose-your-own-adventure stories in that the user is guided through a story with partially branching story arcs and with every new page is asked to choose from available options that can include virtual actions or dialogue replies. The story is segmented into locations or chapters of sorts. While it does not offer 1:1 time scaled playback as Lifeline, it does stand out through a more complex item and character quality system that can additionally influence the success of attempted actions or their mere existence. This analysis expands the list of viable features for ConText by a system for splitting the story into chapters, which may not reflect in the created App but for clarity in the framework interface. As a possible secondary feature it promotes extended player properties that influence and are influenced by how the story unfolds, as well as a dedicated inventory for items to be used in gameplay actions.

Other Lorem ipsum.

2.2.2. Frameworks

Quest

WibbleQuest

Other

2.2.3. Literature

...

3. Development

With research done, the list of proposed features as well as the basic requirements and constraints of the tool had to be compiled into a sound implementation concept. As is usual and likely inevitable for development, changes had to be made over the course of the project and certain obstacles were hit, both covered in the latter two subsections of this chapter.

3.1. Concept

Technical constraints Constraints presented through the platform the framework is to be built on are mostly bound to Unity engine. Since the Unity toolset only works on x86-based computers running Windows 7 or up or Mac OS X 10.8 or up and requires graphics hardware supporting DirectX 9 with shader model 3.0 or up [Uni16], the same applies for ConText. Since the source code for ConText is entirely written in C#, code modification on Mac OS requires an IDE capable of handling C# such as MonoDevelop. As Unity Technologies put it, "The rest mostly depends on the complexity of your projects", at least for the development system. Depending on the target platform the resulting game is ported to, additional constraints are the availability of the respective devices with sufficiently recent operating system versions.

Layers With these underlying system requirements given, the ConText framework itself was envisioned as a layered construct with modular structure both internally and externally. Ideally, this would provide a high degree modularity and flexibility. The layered construct borrows an idea similar to that found in standardized system layer definitions with categories of actions and objects in the framework assigned to different layers, with communication or interfacing function calls happening between the layers. This would encapsulate and decouple similar functionality and make it easier to identify and modify or expand specific parts. The following layers are part of the concept:

1. The **Manager Layer** consists of all managers, which essentially represent the central logic handling the communication, tracking and updating of the game elements. The following managers are part of the Manager Layer:

   a) The **Module Manager** which would keep track of all story modules in the storyline, provide functions for loading modules in or out, prompting the Log Manager to trigger log entries and the UI Manager to display modules in the game UI.
   b) The **UI Manager** would keep track of the UI objects for instanced story modules, handle scaling of the UI on the playing device and provide functions for displaying story modules, log entries and other info.
   c) The **Input Manager**, intended to handle (touch) input and redirect or trigger the according reactions to the respective affected managers.
   d) The **State Manager** which would handle saving and loading of story progress, keep track of the current game state and generally juggle the bits of data that don't fit into any of the other managers.
   e) The **Log Manager**, to keep track of all log entries and provide functions for loading and updating entries.

   The Input Manager would directly communicate with the Input Layer, the UI Manager with the UI Layer and the Module Manager with the Module Layer.

2. The **Input Layer** is very compact and only contains immediate input handling, basically entirely computed by Unity's internal input handler. As such it might be seen as only a theoretical layer, or at least not an actual layer of ConText itself.

3. The **UI Layer** which contains the UI settings, visual ingame representations of the text/story module stream, the log list and a UI wrapper to channel calls and communication of the former to the UI Manager.

4. The **Modules Layer** is essentially the whole collection of modules and module instances and directly communicating the the Module Manager to exchange information about next modules and the storyline's elements.

Where the layer concept actually differs somewhat from the usual definition of layers is that they don't communicate exclusively vertically. The Modules, Input and UI Layers all communicate with and via the Manager Layer.

Modules The initial concept did not contain specific implementation details for the inner logic of individual parts, however the internal messages were defined as far as that they were to be designed as modules, with each module encapsulating one message and containing pointers to both the previous and next modules, akin to a double-linked list, the message data such as the respective sending character, the message content and a unique message ID and functions interfacing in a given way with the managers.

As such, the user would be able to plug together the story with a list of these modules, or even expand the system with their own modules if they want specific functionality. It would only require them to implement the specific details of the interfacing functions to work in line with the module system.

3.2. Final structure

Regarding the final structure of the ConText framework an overview with consideration of the differences is presented first, followed by a more detailed look at perhaps the most central part, the message module system.

Overview For the development system used, the same technical constraints as with the concept are present, as Unity is still used for the tool. The layer structure is mostly retained with Input, Manager, Modules and UI Layers still in place, but communication between the layers is not as channeled as in the concept. Most calls from Input, Modules and UI still happen with and via the Managers layer, but both Input and UI layers also directly communicate with parts of the Modules layer. Thus there is no real distinction between horizontal and vertical layers and the structure is more akin to components in a graph than strict layers. As such one proposed change for a future revision would be to assess and rework the irregular communication between the components and decouple them in a more organized manner.

The Manager Layer has seen the most drastic changes since the initial concept compared to the other layers. The Input Manager is removed entirely as the input is redirected straight to the UI manager and UI objects and state manager changes related to UI actions as well. The State manager concept was essentially split up into the State manager keeping track of the current game state and the Unify class holding references to all managers. It is proposed the two are merged into one class. Additionally, a Story Settings part was split out designed to hold data related to the story properties of the game, namely a list of characters with their respective properties as well as the default message sound and background track for the game. The Module layer calls directly to the Story settings to get relevant data instead of requesting the information through the Module manager. The Module manager retained most of its concept design with the only differences being that module loading and storing is now managed by itself entirely and does not communicate with the State manager and that the UI layer can restart the module stream once it is stopped (while the concept envisioned the Module manager to keep exclusive control over the module stream).

The Input layer is even more of a theoretical construct than in the concept, as it turned out that all relevant input handling is done entirely by Unity's internal systems. As such it is still a layer of the structure but not in any way part of the ConText code base. In the UI layer the text stream and log reference are stripped from the final implementation as the UI manager accesses the objects contained within the two parts in the concept are now directly accessed by the UI manager. What remained are the UI wrapper which keeps references to some of the UI menu's controls and parts and handles transitions between the three main UI views, as well as the UI settings which adhering to its title keeps track of settings such as the mapping of message modules with their UI templates and of visual properties of the UI views and modules.

The Modules layer is implemented almost entirely as envisioned in the concept, with the small difference that the unique message ID is constructed from multiple partial IDs instead of one single number.

Module system The module system is by far not the only complex part of ConText, but it is certainly the most central part. As mentioned before, the story is made up of modules, which each encapsulate a message of some sort. As the game should be able to offer not just text, but perhaps also images or sounds to keep up a sense of variety, so the message modules have the following structure:

1. The **previous module**, which will always have been fired right before the current
2. The **message ID**, made up of four parts
   a) A *sequential part*, which denotes the message's sequential ID within its branch,
   b) A *branch part* denoting the branch the message is in, respective to the latest prior module with multiple outgoing connections,
   c) A *hierarchy part* representing the hierarchy level of the message, with the hierarchy increasing for each branching module,
   d) A *subpart ID* which is only calculated internally when the message has to be split up into multiple partial instances.

   With this structure, each message has its unique identifier. The hierarchy part first specifies which level the message is on, after which branch split it follows, the branch part specifies which of the split branches it lies in, the sequential part then specifies which in line it is in that branch, and the subpart part specifies another order when split partial messages between two consecutive sequential messages happen.

3. The **sending character** as in the character the message belongs to. The character does not need to be an actual person of the story if the message is not to be sent as a message in the visually classical sense, for example one might call the character "Narrative" or not give it any name and set the message property to disregard alignment in order to display it more like a simple text field.

4. The **message send delay** or **delay before send** in seconds which specifies how long the automatic stream should wait before deploying the message.

5. The **message sound**, not to be mistaken with the default message sound in the Story Settings. The latter is a sound played every time a message is triggered, the former is only played for this specific message and could for example be used to imitate a voice message.

6. The **message content** which is fully dependent on the message or rather module type. Default included in the framework are modules of with the content types **Text**, **Image**, both self-explanatory, as well as **Tic Tac Toe** to show how a mini game may be used to provide more involved gameplay to the user and **Reply** which provides the player with several choices to reply to a message and can be used to query knowledge, opinions an direct the story along different paths. These types specify which form of content can be set.

7. The **log entry** which may contain additional information not fit for the message itself and which can then be viewed in the Log view.

8. The **next module** analogously to the previous module specifying which module comes next. Depending on the module type, there can be multiple next modules, such as with the Reply module. To note here is that implementation is subpar as a single next module is part of the basic module type, but multiple next modules are additions made only to the code of branching modules. A proposed change is that next modules be a list with variable length in order to enable better compatibility with calls working with next module data.

The communication procedure for message modules is best explained through an example. Assumed be the following parameters:

1. The framework is correctly configured.

2. The story consists of the modules
   a) M1-T of type Text Module,
   b) M2-R of type Reply Module and

   c) three Mi-T of type Text Module with 3 <= i <= 5,

   with M2-R being the successor to M1-T and M3-T, M4-T and M5-T as the replies outgoing from M2-R.

3. Playback starts with M1-T as the first story module.

Once the story playback is started, the module manager will initiate the automatic text stream. As the first module is specified as M1-T, the manager will access M1-T to grab the sending delay and message sounds to delay the message firing accordingly and trigger then sound. Then it will pass the module object to the UI manager for instancing. The UI manager attempts to find the correct UI template mapping for M1-T's type and instance a GameObject using the template. Next the module's setContent function will be called with the template instance as a parameter in order to input its specific message info into the UI instance. The details of this are down to the specific type, but for the case of M1-T, the setContent function will put the message's sender and message send time at the top of the message bubble, the message text as the main text portion and adjust width and side padding for correct alignment, quite like a text bubble in a common messenger app. Once the content is set, the UI manager takes over again and adds the instance into the Text View's scroll area, scrolls to the bottom to make the new module the latest displayed and finally adds the instance to the module manager's list of instanced messages. At this point, control is returned to the module manager. With M1-T's instancing completed, the automatic stream attempts to grab the next module in line via the getNextPart function implemented in each module type's class. This will return whichever module is set as the next module or a null return. When null is returned, the automatic stream will stop and be halted until something starts it again, such as specific user input. In M1-T's case, the call will return M2-R. The automatic text stream's loop starts from the top again, computing M2-R's UI template mapping, waiting the send delay, playing the message sound and handing over the module to the UI manager. Once the content is set for M2-R, the module manager will again request the next module. This time the call will return null as the Reply module offers reply choices to the player and does not know the next module until the player has made a choice. Once the player chooses an option, the corresponding option will call the restart function of the module manager which will continue the automatic stream with the now known next module. As such, the stream loop will again start from the top with M3-T, M4-T or M5-T. A similar system is used when loading previously saved story progress, excluding any sending delays or user input interrupts.

3.3. Stepping stones

4. Accessibility and user perception

4.1. Study procedure

4.2. First user study

4.2.1. Evaluation and results (del)

4.3. Second user study

4.3.1. Changes (del)

4.3.2. Evaluation and results (del)

5. Closing words

5.1. Outlook and possible future work

5.2. Conclusions from surveys

5.3. Project conclusion

5.3.1. Summary and accuracy to goal (del)

5.3.2. what else? (del)

A. Appendix

A.1. User study documents

A.1.1. Survey form

A.1.2. Tutorial

A.1.3. Documentation

ConText Documentation Revision 1

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.

ConText Documentation Revision 1

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.

ConText Documentation Revision 2

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

ConText Documentation Revision 2

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

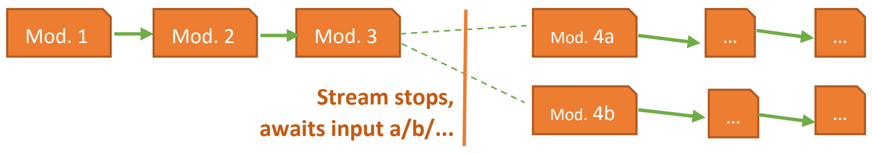
a **log entry**.

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.



ConText Documentation Revision 2

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.

ConText Documentation Revision 2

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

ConText Documentation Revision 2

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…**.

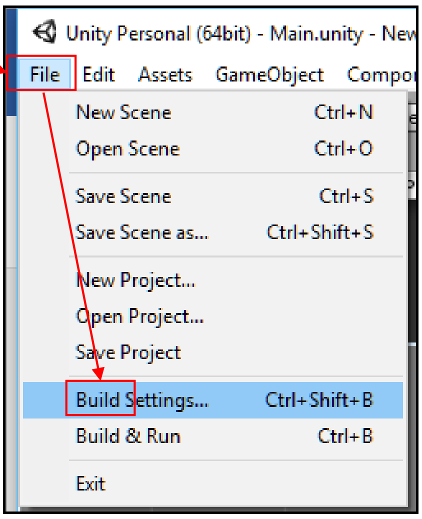
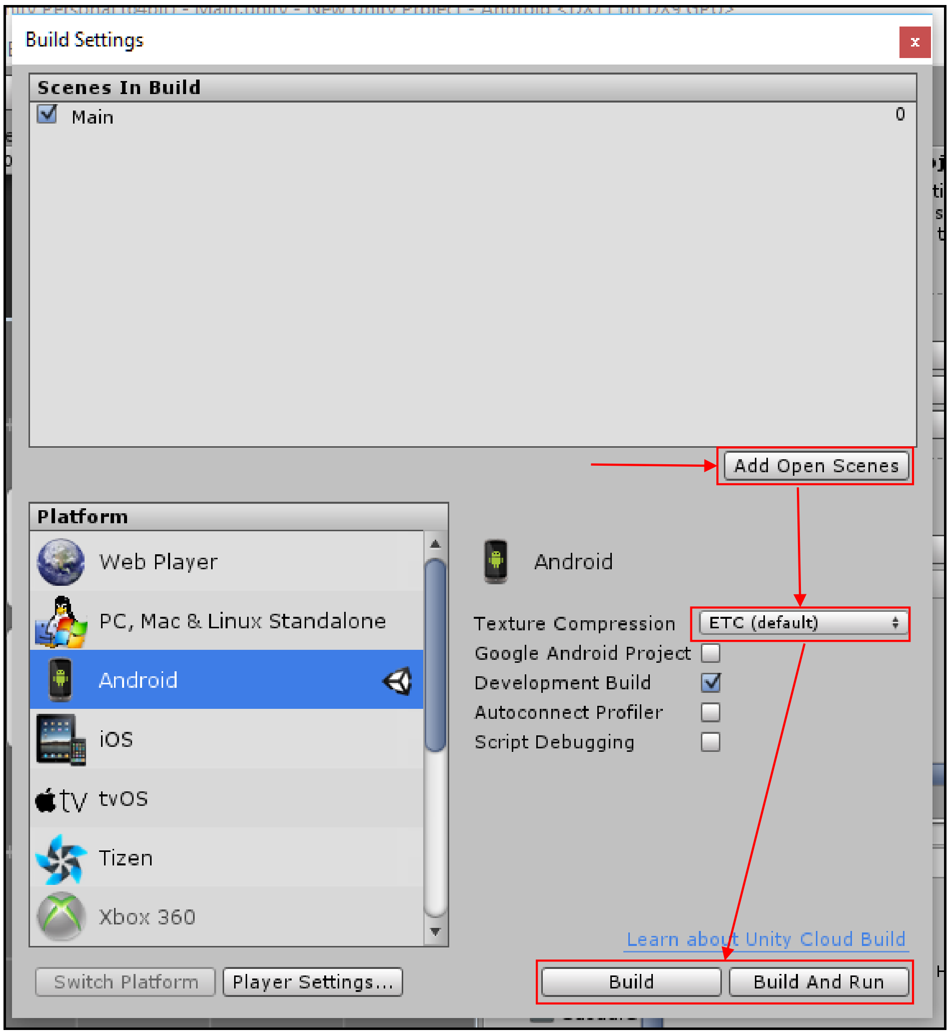


Figure 1: To Build

ConText Documentation Revision 2

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.



Texture Compression: ETC (default)
Google Android Project: ☐
Development Build: ✓
Autoconnect Profiler: ☐
Script Debugging: ☐

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.

ConText Documentation Revision 2

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.

A.2. ...?

List of Figures

List of Tables

Bibliography

[Uni16] Unity Technologies. *SYSTEM REQUIREMENTS FOR UNITY*. 2016. URL: `https://unity3d.com/unity/system-requirements`.