geodesicgames.com/Tilde

Email support@geodesicgames.com with any questions, comments, bug reports, or feature requests not addressed in this document.

Features

- Fully utlizes the new Unity 4.6+ (uGUI) UI system.
- Automatic command registration using function annotation. Console command functions do not need to exist in a specific class or namespace.
- Command name is automatically generated from the annotated function name, however it can be overridden in the function annotation.
- Docstrings and `help` command make navigating complex projects easy.
- Two styles of console prefabs are included: windowed (source engine style) and drawer (quake engine style)
- Automatic command history. Run previous commands by selecting them with the up and down arrow keys.
- Tab command name autocompletion.
- All Unity log, warning and error messages are printed to the console, and colored to match their severity.
- Bind specific keys to execute console commands when pressed through the `bind` and `unbind` commands in the console.
- Save the console log to a file by pressing Ctrl+S.
- All code is fully namespaced, and all assets are contained in a single, top-level directory to help make integration as easy as possible.
- Well written and documented code developed by a senior-level professional game engineer. Easily add new console types or additional console functionality as you see fit.
- Quality look and feel developed by a professional game UI artist and art director.

Adding the console to your project.

There are two different prefabs contained in the `Tilde/Prefabs` folder. The windowed console is a standard movable, resizable, console window similar to the console in games like Dota 2 and Half Life 2. The second is an immovable drawer-style console that slides from the top of the screen, similar to the console in Quake.

To use the console prefabs in your project, you must have uGUI Canvas and EventSystem objects in your scene heirarchy. If you've never used uGUI before, simply add a Canvas (Create -> UI/Canvas) and both a Canvas and EventSystem will be added to the scene. To add a console, simply drag one of the console prefabs onto the canvas.

To see an example of the two prefab consoles in action, open the scene in `Tilde/Examples`.

Defining new console comands.

Console commands are public, static functions annotated with `[ConsoleCommand]`. Annotated command functions optionally take an array of strings containing the arguments to that command, and also can optionally return a string to be printed to the console as output. Functions of the follwing types can be declared as console commands:

```
public static void   foo();
public static string foo();
public static void   foo(string[] options);
public static string foo(string[] options);
```

The name of a console command is generated from the annotated function's name. For example, this function registers a console command "foo."

```
[ConsoleCommand]
public static void foo() { ... }
```

Overriding command registaration defaults

The name of the command defaults to the name of the function that is annotated, however this can be overriden. By specifying the `name` attribute to the `ConsoleCommand` annotation, you can control the name of the console command. For example, the following function is registered as the "bar" console command.

```
[ConsoleCommand(name:"bar")]
public static void foo() { ... }
```

Console commands can also have a docs-string associated with them. This text gives other users a brief explination of what the command does and how to use it. To display the commands, along with the first line of their docs-string, you can run the `help` command in the console. The `help` command can also show the full docs-string for a command when using the `help <command>` syntax. For example, to see the documentation for the `res` command, execute `help res` in the console.

```
> help res
List supported fullscreen resolutions on this device
```

Both docs strings and name overriding can also be used in conjunction.

```
[ConsoleCommand(name:â€my_commandâ€, docs:â€This is a command I madeâ€)]
public static void do_a_thing() { â€¦ }
```

Commands with output

Commands can optionally print output to the console on completion by returning a string. This string is printed directly to the console, and can contain newlines and rich text markup.

Commands with arguments

Commands can also consume parameters that the user specifies when calling the command in the console. These parameters are split by whitespace and passed as an array of strings to the command function. This allows commands to accept and interpret an arbitrary number of arguments as they see fit. For example, consider the following function:

```
[ConsoleCommand]
public static string play_sound(string[] options) {
    // TODO do something with options here...
    return "bang!";
}
```

When the command `play_sound bang.wav playerPosition` is executed in the console, the string array passed to the annotated `play_sound` function contains the strings â€˜bang.wavâ€™ and 'playerPosition'. Additionally, when `play_sound` completes execution, "bang!" is printed to the console window.

Version History

1.0 - *3/31/15*

- Initial release