This documentation was generated in this format for your convenience; however, it may not show all images and animations, and it may be abridged. Please see the website for the complete documentation.

Minibuffer Console README

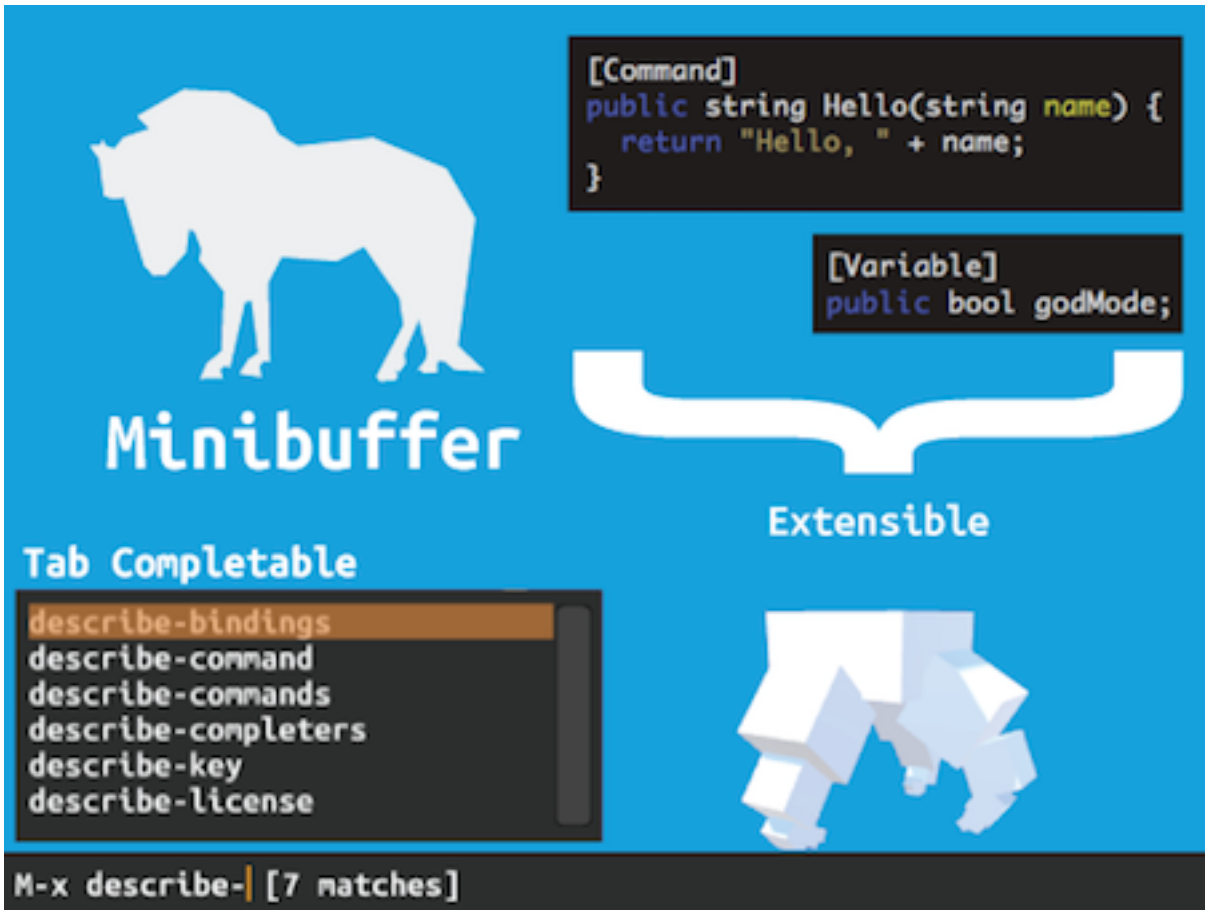
Minibuffer is a developer console for Unity. It is designed to help developers and testers exercise their game in the multitude of ways that making a game requires. It does this by allowing one to define interactive commands, variables, and key bindings for use at runtime. An input prompt—that is, a *minibuffer*—offers tab completion, history, groups, and annotations. Tab completion with type coercion and lookup avoids tedious string handling and delivers the objects of interest. Interrogate Minibuffer at runtime to show its available commands, variables, and key bindings. Minibuffer delivers a consistent, discoverable, and extensible UI for developers and testers that is easy to integrate.

Minibuffer Console is available on the Unity Asset Store.

Demo

- Try the demo!

Core Features



- Write and run commands
- Input prompt
  - Tab completion
  - History
  - Type coercion and lookup
- Bind keys to commands
- Get and set variables
- One window
  - Multiple buffers

Auxiliary Features

- Unity specific commands
  - Switch to a scene
  - Show debug log
  - Capture a screenshot
  - Toggle a game object
- GIF recording commands
- Twitter commands

  - Tweet a message
  - Tweet a screenshot
  - Tweet a GIF

Motivation

The 5 Stages of a Developer UI abridged

Developers often require specialized facilities when making a game, e.g., jump to a different scene, turn off damage, provoke an event. The issue is how does one expose those facilities? There are many ways and often a game will go through a series of stages. This short video caricatures The 5 Stages of a Developer UI (<1 minute).

- Stage 0: No developer UI
- Stage 1: Hot keys
- Stage 2: Buttons
- Stage 3: Command line interface
- Stage 4: Minibuffer

Each stage of the developer UI is better than the last but all can become unwieldy; moreover, creating a UI is hard. Why not reserve such hard work for one's game proper? The player deserves a great UI. The developer deserves a consistent, discoverable, and easily extendable UI among other qualities.

Minibuffer proposes one create a developer UI primarily through code decoration. Have a method you want to execute at runtime? Add a `Command` attribute. Done. Have a property or field you want to inspect or edit at runtime? Add a `Variable` attribute. Need a more specific prompt for the user? Add a `Prompt` attribute.

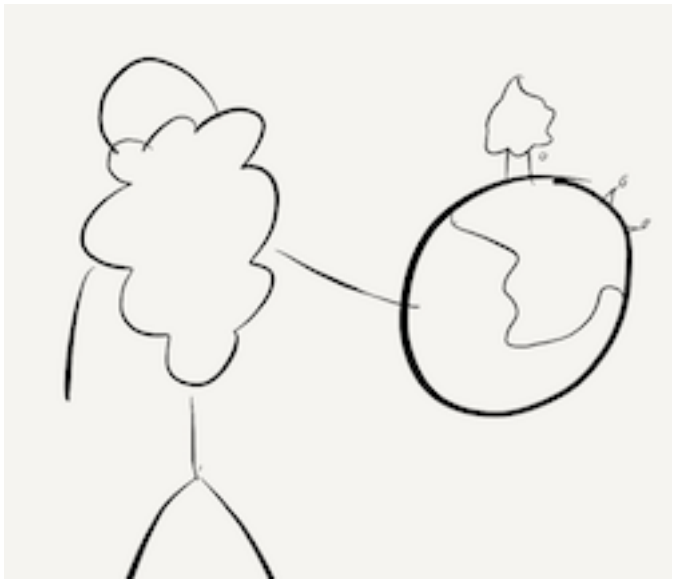
Installation

To install Minibuffer import the asset from the Unity Asset Store. Drag the "Minibuffer Console" prefab into a scene. By default, Minibuffer is configured to remain loaded after scene changes.This behavior can be turned off by disabling the `DontDestroySingleton` script on the 'Minibuffer Console'.

Usage

Minibuffer remains hidden normally. It appears when user input is requested. To run the command 'hello-world' type alt-x hello-world. That will print Hello, World! to the echo area. The code for this command is shown below but first a word about the key sequence notation. You may also see M-x used instead of alt-x. They mean the same thing; you can configure Minibuffer to use whichever notation you prefer. See below for more details.

Tab Completion



> In the beginning the world was without form, and `void*`; And lo Dennis said, "Hello, World." And it was good. And Dennis said, "Let there be text." And there was text. And it was good. But the text was difficult to produce. So Dennis said, "Let there be tab completion." And it was very good.

Minibuffer offers extensive tab completion. For instance, type alt-x tab to see all the available

commands in a popup window. Scroll up or down the list of completions using the arrow keys, the mouse, or by typing pagedown or pageup respectively.

Integrating Minibuffer with Code

Minibuffer's UI-by-decoration philosophy makes it easy to integrate into any class using these three steps:

Step 1. Reference the namespace

```
using SeawispHunter.MinibufferConsole;
```

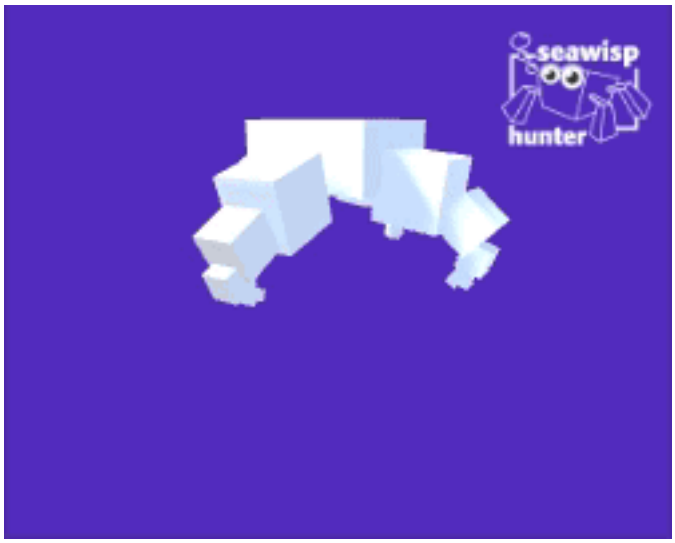
Step 2. Register with Minibuffer

```
Minibuffer.Register(this);
```

Step 3. Add [Command] to any public method

```
[Command]
public static string HelloWorld() {
  return "Hello, World!";
}
```

Type alt-x hello-world to run the command. See `HelloCommands.cs` for a complete example.



Commands Can Accept Arguments

Let's try a command that requires an argument.

```
[Command]
public string Hello(string name) {
  return "Hello, " + name + "!";
}
```

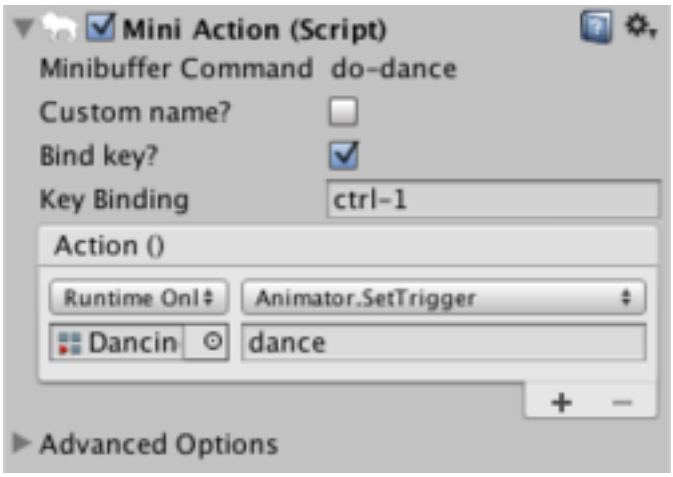
Type alt-x hello. It will prompt you String name:. Type your name or your home world. It may print Hello, Middle-earth!Any command that returns a string will be shown using `Message()`.

Integrating Minibuffer Without Code

> "Look ma, no code!"

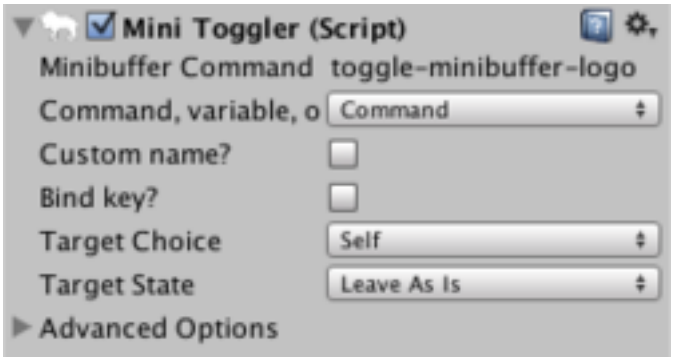
Code is not the only way to integrate Minibuffer into your game. `MiniAction`, `MiniToggler`, `MiniInfo`, and `MiniIncrementer` offer ways to add commands, key bindings, and variables to Minibuffer without writing any code.

MiniAction



Suppose you want to trigger a "dance" animation on a key press of ctrl-1. Add the MiniAction script. Configure its action to trigger an animation on the appropriate object. Then you can trigger the animation by typing alt-x do-dance or ctrl-1.

MiniToggler

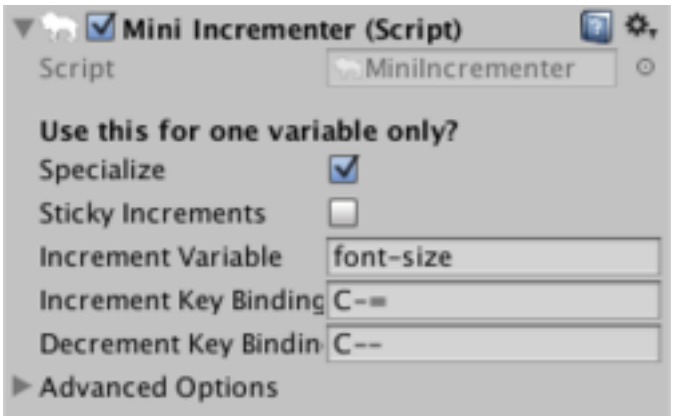


Suppose you want to activate or deactivate a game object named “Secret Door”. Add the MiniToggler script. You can configure it to add a command alt-x toggle-secret-door, or add a variable that can be edited with alt-x edit-variable secret-door, or add both. You can also add a key binding, or use a custom name for the command or variable.

MiniInfo

Suppose you want to describe your game’s instructions or release notes for testers. Add the MiniInfo script. Copy-and-paste the text. And it’s accessible by typing alt-x describe-game or ctrl-h g.A placeholder for just this purpose already exists in the ‘Minibuffer Console’ prefab.

MiniIncrementer



Suppose there is an integer variable that you want to fiddle with at runtime, like "health". You can mark it with the `[Variable]` attribute.

```
[Variable]
public int health;
```

Then you can type alt-x edit-variable health to edit health, but sometimes you'd prefer to not do that while in a fight for your life. The `MiniIncrementer` script makes it easy to increment or decrement an integer variable with a pair of key bindings.

Tab Completion

Tab completion can be setup a number of ways.

Ad hoc Completion

If you have a small set of fixed options, you can provide a list of completions. Type alt-x cheat-code return tab to see them. In the tab complete popup window, you can move the

selection line up or down by typing downarrow or uparrow respectively. Hit return on the selection line will complete to that selection. Should you ever make a mistake you can hit escape to quit or cancel that action. Quitting twice will hide Minibuffer.

```
[Command]
public string
  CheatCode([Prompt(completions = new []
                      { "GodMode", "SkipLevel", "Invulnerable"})]
                      string str) {
  // Activate cheat here...
  return str + " activated.";
}
```

These "ad hoc" completions are intended as one-offs. If you find yourself duplicating this list elsewhere, it's advisable to create a formal completer: `minibuffer.completers["cheat-codes"] = new ListCompleter(...);` Then reference the completer with the `Prompt` attribute where you need it: `[Prompt(completer = "cheat-codes")]`.

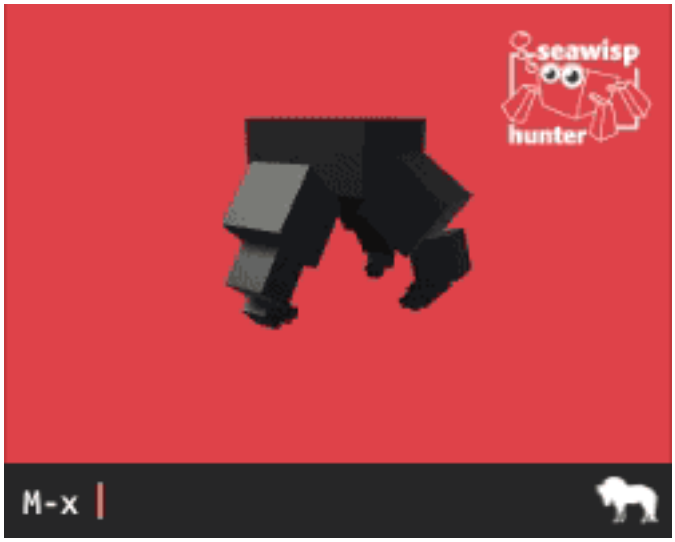
Inferring Tab Completers

Tab complete colors.

One can specify a completer like `[Prompt(completer = "Color")]` which completes twelve common colors; however, often times the type is sufficient to infer the proper completer. For instance alt-x quadrapusuper-"color requests a Color c: that you can provide via tab completion.

```
[Command(description = "Set color of quadrapus")]
public string QuadrapusColor(Color c) {
  var renderers = quadrapus.GetComponentsInChildren<MeshRenderer>();
  foreach(var renderer in renderers) {
    renderer.material.SetColor("_Color", c);
  }
  return "Changed color to " + c + ".";
}
```

Builtin Tab Completers



Minibuffer offers a number of completers for access to itself, the file system, and Unity objects. Here are some completers: `buffer`, `command`, `directory`, `file`, `variable`, `AnimationClip`, `AudioClip`, `Color`, `Component`, `Font`, `GameObject`, `GUISkin`, `Material`, `Mesh`, `PhysicMaterial`, `Scene`, `Shader`, `Sprite`, and `Texture`. This list is not exhaustive, however, because completers may be dynamically generated for enumerations and subclasses of `UnityEngine.Object`. Thus you can tab complete any components, like `Camera` shown below, or any of your MonoBehaviour scripts. Extend Minibuffer–implement `ICompleter`–and create your own tab completers!

```
[Command]
/* There is no "Camera" completer yet. But there will be after this
   command runs because the Camera class is a UnityEngine.Object.
   Thanks, ResourceCompleter! */
public void PanLeft(Camera camera) { ... }
```

Type Coercion and Lookup

"Strings are of no use here."–Gandalf

Minibuffer can prompt the user for a string, of course. But it may also prompt the user for types it can coerce from a string like `int`, `float`, `Color`, `Vector2`, `Vector3`, `Matrix4x4`, etc. Or it can look up existing objects like: `GameObject`, `Material`, `Scene`, or any `MonoBehaviour` scripts to name a few. See `ResourceCompleter` for more details. And you may add your own coercers via `ICoercer`.

Preview Completions

Unity offers previews in its Editor for assets like Textures and Materials. Minibuffer can also present these previews. (Currently only supported while playing in Editor).

```
[Command]
public string QuadrapusMaterial(Material m) {
  foreach(var renderer in quadrapus
          .GetComponentsInChildren<MeshRenderer>()) {
```

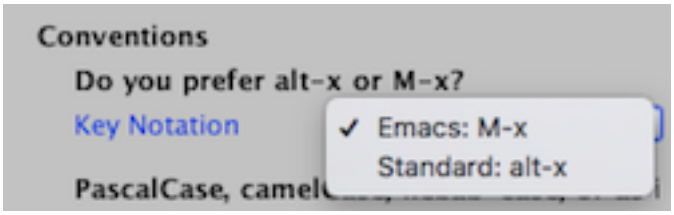
```
    renderer.material = m;
  }
  return "Changed material to " + m.name + ".";
}
```

Key Bindings

You can add a key binding by providing a value to the `keyBinding` field in Command. For instance, you could add the following to the HelloWorld command, then it would activate whenever you typed ctrl-h w.

```
[Command(keyBinding = "ctrl-h w")]
public string HelloWorld() {
  return "Hello, World!";
}
```

How Key Bindings Work



You've seen that alt-x allows you to execute a command and tab invokes a completer. In truth these are just key bindings to commands: alt-x is bound to 'execute-extended-command' and tab is bound to 'minibuffer-complete'. There are many other key bindings: See `Keymap` for the complete listing of the default key bindings. You can inspect the key bindings at runtime by typing alt-x describe-bindings or ctrl-h b. See note about key sequence notation: Emacs M-x and Standard alt-x key sequence notations are both supported by Minibuffer; select to suit your taste.

Variables

In addition to commands, fields and properties can as be exposed as *variables* to Minibuffer. For instance, suppose you wanted to expose the contents of a `UI.Text` component. Simply add a `Variable` attribute. You can also give the variable a different name or provide a description similar to `Command`.

```
[Variable]
public string quote {
  get { return _quote.text; }
  set { _quote.text = value; }
}
```

Type alt-x edit-variable quote to edit the variable. Type alt-x describe-variable quote to see information on the variable.

UI-by-Decoration

Minibuffer proposes one create a developer UI primarily through code decoration. This approach has three big advantages:

1. It minimizes the effort to create runtime accessible commands.

Create commands out of methods with one attribute: `[Command]`.

2. Command methods are just methods.

   Attribute decorations are ignored by other callers in the code base. The methods can do double duty as both game machinery and developer UI.

3. The UI is discoverable at runtime and consistent.

   Forgot what key you bound to the command 'more-enemies'? Type ctrl-h c more-enemies and Minibuffer will tell you, or look at all the key bindings ctrl-h b. Discoverability is a big gain for teams and testers.

A Commons for UI-by-Decoration

A small portion of Minibuffer is licensed under the `MIT license` to encourage two things:

1. Asset developers can make their own assets "minibuffer-able."

Part of Minibuffer may be redistributed so that users without Minibuffer will not notice if it's missing from an asset that exposes its own Minibuffer commands. Users with Minibuffer, however, will have new commands, variables, and key bindings that enhance the developer's asset.

2. Asset developers can create their own UI that uses UI-by-Decoration.

Minibuffer, as its name suggests, takes a lot inspiration from Emacs in terms of the user interface it exposes. However, such a keyboard-centric approach may not be well suited for mobile platforms for instance. This license allows another developer to create the UI they prefer but participate in a commons where a command created for Minibuffer may be used in some future asset X and *vice versa*.

See the `MIT license page` and `MinibufferStub` class for more details.

† MIT License logo by Excalibur Zero licensed under the Cctrl-BY 3.0

Minibuffer-ables

Here are some projects that extend Minibuffer.

- minibuffer-arcadia-support

Write your games in Lisp! Arcadia is a Clojure environment for Unity. Install this project, so you can evaluate lisp expressions in Minibuffer. Type alt-: (+ 1 2) to ensure you've got the real thing 3. Define new commands and variables in Lisp!

- Know of any others? Let me know.

Frequently Asked Questions

- Why not use Unity's inspector instead of Minibuffer?

Guts vs Levers, Toggles, and Dials—Oh my!

Unity's inspector is a great resource for developers while editing their game, and Minibuffer is no substitute; however, Unity's editor and inspector are not available to the developer once the game has been built and distributed. Minibuffer is available while playing in the editor *and* at runtime in the build. Unity's editor also provides access to the "guts" of one's game including every component of every game object, which in an actual game can be an overwhelming number of objects.

Minibuffer provides a means of curating what the "levers, toggles, and dials" ought to be for one's game by creating variables and commands. To see what the specialized commands and variables are for a game, try typing alt-X tab which runs 'execute-user-command'. This command works just like alt-x but it only shows commands that are not built-in, so it only shows the specialized commands for that game. Should you ever come across Minibuffer in the wild, alt-X provides you with a lay of the land.

Dependencies

Minibuffer comes with batteries included; however, the included third-party libraries deserve acknowledgment.

- ctrl-Sharp-Promises: Promises library for C# for management of asynchronous operations, MIT License
- CommonMark: A Markdown parser and HTML converter, BSD License
- FullSerializer: A robust JSON serialization framework, MIT License

Only the Promises library is exposed through Minibuffer's API.

Acknowledgments

I would like to thank my fellow game developers at the Vermont Game Developers Meetup group for their input on Minibuffer. Special thanks to Chris Ziporyn, Ben Throop, Curtis Aube, and Chris DeGuise for their input and support along the way.

About the Author

Shane Celis is an evolutionary robotics researcher turned game developer, making fun virtual robot toys that hint at artificial evolution (think: educational trojan horse). His most recent games are Quadrapus Sumo and a DIY Magic Simulator. He runs the Vermont Game Developers Meetup group. You can find him on twitter @shanecelis.

Minibuffer started as a sketch and a tweet over a year ago.