



Live-coding plug-ins UI with Wren

How it works?
Limitations?
Future?



Meeting
Jan 11th 2022
(Happy new year!)

wren

A CLASSY LITTLE SCRIPTING LANGUAGE

Wren is a small, fast, class-based concurrent scripting language

Think Smalltalk in a Lua-sized package with a dash of Erlang and wrapped up in a familiar, modern syntax.

```
System.print("Hello, world!")

class Wren {
  flyTo(city) {
    System.print("Flying to %(city)")
  }
}

var adjectives = Fiber.new {
  ["small", "clean", "fast"].each {|word| Fiber.yield(word) }
}

while (!adjectives.isDone) System.print(adjectives.call())
```

- **Wren is small.** The VM implementation is under 4,000 semicolons. You can skim the whole thing in an afternoon. It's *small*, but not *dense*. It is readable and *lovingly-commented*.
- **Wren is fast.** A fast single-pass compiler to tight bytecode, and a compact object representation help Wren compete with other dynamic languages.
- **Wren is class-based.** There are lots of scripting languages out there, but many have unusual or non-existent object models. Wren places classes front and center.
- **Wren is concurrent.** Lightweight *fibers* are core to the execution model and let you organize your program into a flock of communicating coroutines.
- **Wren is a scripting language.** Wren is intended for embedding in



[Getting Started](#)
[Contributing](#)
[Blog](#)
[Try it!](#)

[GUIDES](#)
[Syntax](#)
[Values](#)
[Lists](#)
[Maps](#)
[Method Calls](#)
[Control Flow](#)
[Variables](#)
[Classes](#)
[Functions](#)
[Concurrency](#)
[Error Handling](#)
[Modularity](#)

[API DOCS](#)
[Modules](#)

[REFERENCE](#)
[Wren CLI](#)
[Embedding](#)
[Performance](#)

A language:

- easy to integrate
- kind-of like Lua

int3 Systems translated the main implementation to D.



Demo time!

This is Distort, modified to use live-reload absolute path AND optimizations.

(Also everything was moved to the “reflow()” Wren callback because only reflow is called repeatedly.)

Let’s shape the “Drive” button.



Live-reload internals

Using live-reload necessitates absolute paths, not intended for release. (but optimizations = nice)

```
debug
| // debug => live reload, enter absolute path here
| context.wrenSupport.addModuleFileWatch("plugin", `~/my/absolute/path/to/plugin.wren`);
else
| // no debug => static scripts, for release binaries
| context.wrenSupport.addModuleSource("plugin", import("plugin.wren"));
```

Because it's hard to “forget” about a module once loaded, **the whole Wren VM is restarted** every time the file change && about 200ms has elapsed.

=> You have to SPAM the save button in live-edit.

The things you can do from Wren

What can you do exactly from Wren?

Here is what you can do from Wren: *(as of Dplug v12.3)*

- Set positions of a `UIElement` at UI creation or reflow.

```
static reflow() {  
    var S = UI.width / UI.defaultWidth  
    ($"_inputSlider").position = Rectangle.new(190, 132, 30, 130).scaleByFactor(S)  
}
```

- Set/get values of fields in `UIElement`-derived classes that are marked with `@ScriptProperty`, at UI creation or reflow.

```
static reflow() {  
    var S = UI.width / UI.defaultWidth  
    ($"_inputSlider").litTrailDiffuse = RGBA.new(151, 119, 255, 100)  
}
```

- Plus everything you can normally do in [Wren](#).

`ui.wren` is a sort of standard library for Dplug + Wren.
Overall, limited to UI properties for now.

Expose your own properties



How to expose your own properties to Wren

```
class UIMyButton : UIElement
{
  @ScriptProperty RGBA color;
  @ScriptProperty RGBA colorPushed;      // All that exposed to Wren if it knows about UIMyButton.
  @ScriptProperty float animationSpeed;
}
```

Write a normal custom widget, and use `@ScriptProperty` on fields you want to be accessed from Wren.

Expose your own properties



Supported types for @ScriptProperty

@ScriptProperty fields can be of the following types: *(as of Dplug v12.3)*

- bool
- byte / ubyte / short / ushort / int / uint
- float
- double
- RGBA
- enums or L16 (but this is lowered to integers)

Expose your own properties



2 ways:

```
// Option 1:

// UI constructor
context.wrenSupport.registerScriptExports!MyGUI;

// UI fields
@ScriptExport
{
    UIMyButton _mybutton;
}

// Option 2
context.wrenSupport.registerUIElementClass!UIMyButton;
```

- Registering classes enumerates @ScriptProperty fields and save their layout.
- Property access are not calling Wren functions, but calling one Wren function that directly write bytes into the objects.
- **Consequently**, you can't validate fields, or have complex prop.



FUTURE: Tuning PBR and audio variables with Wren?

- it's NOT possible for the Dplug user to expose arbitrary APIs
BUT you can expose @ScriptProperty fields in a custom widget...

=> you can “tune” anything with live-reload if you write (say) audio variables in your onDrawRaw/onDrawPBR (!?)

Future widgets: UIAudioTuningCenter, UIPBRSettings?

Also: interactive color correction.



Wren: at what cost?

- ~200kb in binary
- 1 to 10 mb of RAM. We have a GC again ^^
- slower at resize and UI opening (don't know how much)
- Wren significantly different from D (functional, dynamically typed, significant space...)
- super-limited for now: can't create widgets, or change their visibility, or dirty them.

Most probably: live-reload it is going to save you a lot of time, and get better visual results.



All informations on the Dplug Wiki

<https://github.com/AuburnSounds/Dplug/wiki/Making-a-Scriptable-UI>

See also: the Distort example which has a scriptable, resizable, PBR UI.



Thank you

Questions?