MRuby-Zest: a Scriptable Audio GUI Framework

Mark McCurry
DSP/ML Researcher
United States of America
mark.d.mccurry@gmail.com

Abstract

Audio tools face a set of uncommon user interface design and implementation challenges. These constraints make high quality interfaces within the open source realm particular difficult to execute on volunteer time. The challenges include producing a unique identity for the application, providing easy to use controls for the parameters of the application, and providing interesting ways to visualize the data within the application. Additionally, existing toolkits produce technical issues when embedding within plugin hosts. MRuby-Zest is a new toolkit that was build while the ZynAddSubFX user interface was rewritten. This toolkit possesses unique characteristics within open source toolkits which target the problems specific to audio applications.

Keywords

Interface Design, LV2, VST, Ruby

1 Introduction

MRuby-Zest was created to address long standing issues in the ZynAddSubFX[1] user interface. The MRuby-Zest framework was built with 5 characteristics in mind.

Scriptable: Implementation uses a first class higher level language

Dynamically Resizable: Fluid layouts which do not have any fixed sizes

Hot Reloadable: Reloads a modified implementation without restarting

Embeddable: Can be placed within another UI without conflicts

Maintainable: Relatively simple to read and write GUI code

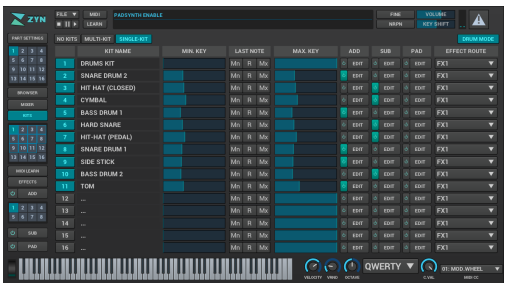
Several examples of the toolkit can be seen in Fig. 1, 2, 3, and 4.

Figure 1: Zyn-Fusion Add Synth

1.1 History

Historically the ZynAddSubFX interface was written in FLTK[2] and the user interface processed a number of usability issues as well as look and feel consistency issues. Additionally the multi-window FLTK design ZynAddSubFX previously used did not embed cleanly into plugin hosts. Mid 2014 a series of mockups by posted online by Budislav Stepanov[1]. The mockups provided an overhaul of the workflow of the GUI, but it was a new design which did not make use of any of the existing widgets, nor widgets available in other toolkits. Since the new interface was not small some tools would be needed to increase the speed of development.

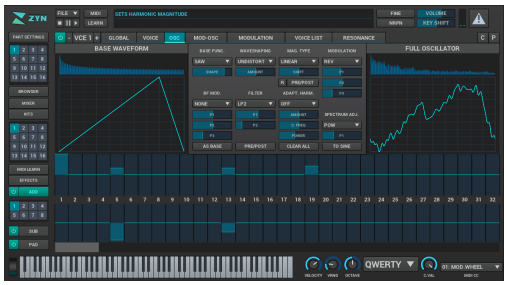
Figure 2: Zyn-Fusion Kit Editor



The first prototypes were written in the Qt Meta Language (QML)[3; 4] QML is a domain

[1] `http://www.kvraudio.com/forum/viewtopic.php?f=47&t=412173`

Figure 3: Zyn-Fusion Oscillator



specific language commonly used to describe a group of components and properties within a user interface. In addition to purely describing components, QML can also define callbacks and new functionality for widgets using a scripting language. Within Qt, this scripting language is javascript.

While prototyping ZynAddSubFX's UI, the prototype frequently ended up accessing the C++ to QML layer of Qt which received much less documentation than the pure QML layer. Some of the logic/drawing routines for the program ended up in C++ portion which couldn't be effectively hotloaded, which slowed development. Additionally the barrier between C++ and Qt's javascript engine was non-trivial. Overall, this process highlighted that for the prototype and the version of QML used:

- QML's javascript was not sufficiently flexible when extending widgets
- QML's layout algorithms did not meet the requirements of the new design
- None of the QML components were heavily used beyond primitives (rectangles, component-repeaters, etc)

Figure 4: Zyn-Fusion Pad Synth

QML at a high level was useful, concise, and easy to dynamically manipulate. The infrastructure around it was limiting for the ZynAddSubFX use case. So, at this stage of prototyping the question was posed: "Why does QML need to be tied to Qt and the specific scripting language of Javascript?"

QML within Qt was script-able, layout routines were flexible enough that resize-ability wasn't a major issue, and it was built with hot loading in mind. Per embed-ability, Qt does not embed well; specifically, loading two plugins which use different Qt versions (e.g. Qt4/Qt5) is known to cause issues with symbol name conflicts and global variable conflicts. When initial prototyping was done with QML it was acknowledged that eventually the project may need to move away from Qt and MRuby-Zest was born. MRuby-Zest took the QML language, replaced the scripting language with Ruby, integrated it with the nanovg OpenGL rendering library, and began to leverage parameter metadata that ZynAddSubFX produces via the rtosc library[5].

1.2 Prior Art

The problem of creating a good looking embeddable GUI isn't a new task in the open source audio realm. Audio plugins are a challenging design space. Complex information needs to be presented to a reasonably non-technical audience in a way that they can quickly understand how to manipulate it. To facilitate this, an audio plugin needs to differentiate itself from other applications and provide a consistent and easy to understand visual and interactive language for the user to tune.

There's certainly plenty of tools based upon more standard toolkits like GTK or Qt. A few of the open source audio plugin toolkits include: AVTK[6], robtk[7], fffltk[8], DPF[9], rutabaga[10], JUCE[11], and a few PUGL based non-toolkit options also exist in some smaller applications.

Compared to these toolkits, MRuby-Zest desires to be generally built for larger more complex applications as well as having a distinct look and feel. Additionally the heavy use of Ruby scripting makes MRuby-Zest more geared towards rapid development of a large complex interface.

2 Implementation

The MRuby-Zest framework is implemented through a combination of different layers. This includes QML parsing/processing, OSC communication, event handling, and the widget classes themselves.

2.1 QML

QML is a domain specific language commonly used to describe a group of components within a user interface. More generally, QML defines a tree of objects, methods on object instances, a set of interrelated properties, and bindings for the properties. Within Qt, QML runs on Javascript on top of the normal tools that Qt provides. MRuby-Zest's QML uses Ruby for scripting, but otherwise shares most structural similarities.

Through the use of a dynamic language QML gains a number of properties which make interface development easier. First and foremost is the conciseness of the language. Using C++ a simple widget ends up being rather verbose:

Listing 1: C++ Widget

```
class SubWidget: public Rectangle
{
    public:
      SubWidget(void) {
          fooVar = "foo";
          barVar = true;
          structure = new Structure;
          model = new Model;
          structure->add_parent(this);
          model->add_parent(this);
      }

      ~SubWidget(void)
      {
          delete structure;
          delete model;
      }


      string fooVar;
      bool barVar;
      Structure *structure;
      Model *model;

      void fn(string args)
      {
          cout << args << endl;
          structure->method();
      }
};
```

With ruby methods/callbacks QML would look virtually the same. Indeed parsing all of the QML I had written thus far didn't depend upon the scripting language at all. With ruby it was possible to use QML to create something like:

Listing 2: QML Widget

```
Rectangle {
    id: window

    property String fooVar: "foo"
    property Bool barVar: true

    Structure { id: structure }
    Model { id: model }

    function fn(args) {
        puts args
        structure.method()
    }
}
```

And translate it to something similar to:

Listing 3: Ruby Widget Result

```
class Instance < Rectangle
    attr_reader :structure, :model
    attr_property(:fooVar, String)
    attr_property(:barVar, Bool)

    def initialize()
        add_child(@structure =
            Structure.new)
        add_child(@model =
            Model.new)
        set_property(:fooVar, "foo")
        set_property(:borVar, true)
    end

    def fn(args)
        puts args
        structure.method
    end
end
```

While this transformation may seem trivial, the organizational structure that QML's Qt Modeling Language provides is helpful at understanding complex widget hierarchies at a glance.

2.2 Hot-loading

When developing or maintaining a synth a considerable amount of time is spent on improving the user interface. GUI development can be slow going work and compared to other tasks it can be harder to obtain a fast feedback loop. Generally GUI development in these cases has the loop of:

1. Build - Compile from source
2. Open - Launch the application

3. Navigate - Get to the part of the application which is modified
4. Observe - See how the application behaves
5. Close - Close application
6. Modify - Change behavior
7. Repeat - From step 1 repeat

MRuby-Zest on the other hand makes it possible to load code into live instances of the user interface. Hotloading code in MRuby-Zest is possible since the vast majority of code can be relatively simply converted to Ruby code and loaded into the active Ruby VM during execution. Using hotloading the development loop becomes:

1. Build - Compile from source
2. Open - Launch the application
3. Navigate - Get to the part of the application which is modified
4. Observe - See how the application behaves
5. Modify - Change behavior
6. Repeat - From step 4 repeat until done
7. Close - Exit after desired behavior is obtained

Reducing the feedback loop makes it much easier to tune graphics, layout, and the feel of input handling.

2.3 OSC Communications

Different GUI toolkits have different approaches on communicating state to the rest of the application outside of the interface (the backend). MRuby-Zest leverages Open Sound Control (OSC) to communicate to in-process and out-of-process backends. This submodule is known as the OSC-Bridge.

The OSC-Bridge controls communication to the optionally-remote synthesis engine, and provides metadata for modeling parameters in the user interface. The OSC interface specifies the minimum value, maximum value, short names, tooltips, and other information about parameters that can be accessed. Additionally, this layer provides several mechanisms for tracking and synchronizing the value of remote parameters. These mechanisms abstract away synchronization mechanisms, simplifying the widget programming.

2.4 Drawing model & events

MRuby-Zest is an OpenGL based toolkit which uses PUGL[12] for platform specific event handling and nanovg[13] for a drawing API. OpenGL 2.1 (with the framebuffer extension) was used to simplify embedding and enable complex animations in future versions. NanoVG was used to simplify drawing vector graphics, which were necessary for simplified fluid resizing of the GUI.

When drawing in the MRuby-Zest toolkit, widgets are drawn depth first for each layer of the user interface. These layers are:

- the background - where most widgets are drawn
- the animation layer - simple drawings expected to update many times a second
- the overlay - drawing on top of the interface (e.g. modals/dropdowns)

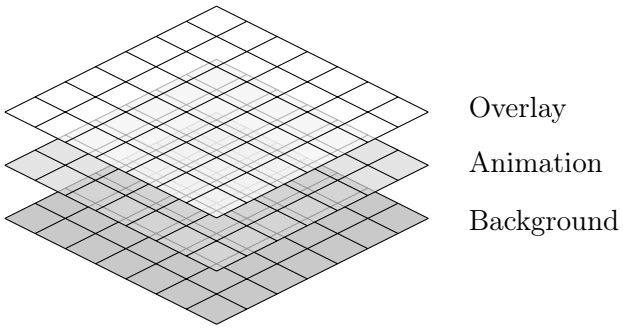


Figure 5: Framebuffer layers

Since the widgets define strict bounding boxes for drawing, redrawing can be cheaply done. First, the damaged part of the altered layer can be masked. Then, all widgets which intersect with the layer and damaged region are redraw. Finally, the three framebuffer layers are redrawn producing the final GUI.

On the event handling side, MRuby-Zest behaves fairly traditionally. At the time of writing MRuby-Zest responds to:

- Key presses/releases
- Mouse presses/releases
- Mouse drags
- Mouse hovering
- Window resizing

2.5 Widgets

The current version of MRuby-Zest has 182 widgets. These range from simple buttons, labels, and boxes to complex views of parameters. Two major types of widget that are available in MRuby-Zest are layout widgets and parameter controlling widgets.

In MRuby-Zest there are grid pack (Fig. 6), module pack (Fig. 7), tab pack, vertically packed, horizontally packed, and other layout specific widgets. Historically the resizing was taken care of by a constraint layout system which solved a set of linear-equations via GLPK[14], however this approach proved too computationally expensive and was removed to maintain a more consistent framerate.

Figure 6: Grid Layout

Figure 7: Control Rows Layout

There are also a wide array of options to represent parameters. This includes Knobs (Fig. 8), sliders (Fig. 9), drop downs (Fig. 10), buttons, plots (Fig. 11), text editors, piano keyboards, and more.

Figure 8: Knob Widget

Figure 9: Horizontal Slider Widget

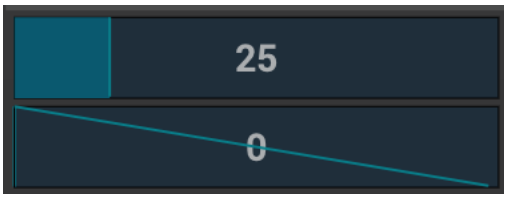


Figure 10: Drop down Widget

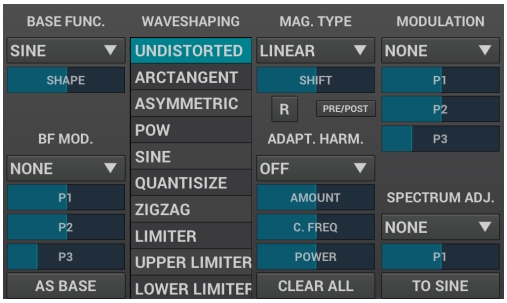
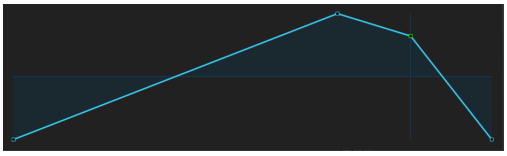


Figure 11: Envelopes/2D plotting Widget



3 Conclusion

Audio applications are a complex design and programming domain. Existing toolkits pose embedding challenges as well as difficulties in rapid development. MRuby-Zest provides one new approach to audio plugin GUI development and is available at `https://github/mruby-zest/` under a mixed MIT and LGPL license. Using MRuby-Zest, the ZynAddSubFX project has been able to build the new ZynFusion interface. This interface serves as a complex example of the MRuby-Zest framework and shows that the chosen approach can speed up development on non-trivial designs.

References

[1] N. O. Paul, M. McCurry, *et al.*, "Zynaddsubfx musical synthesizer." `http://zynaddsubfx.sf.net/`, 2018.

[2] B. Spitzak *et al.*, "Fast light toolkit (fltk)," 1998.

[3] H. Nord, E. Chambe-Eng, *et al.*, "Qt - software toolkit." `http://qt.io/`, 2018.

[4] Q. Contributors, "Qt - software toolkit." `https://doc.qt.io/qt-5.10/qtqml-index.html`, 2018.

[5] M. McCurry, "rtosc - realtime safe open sound control." https://github.com/fundamental/rtosc, 2018.

[6] H. van Haaren, "Avtk." https://github.com/openAVproductions/openAV-AVTK, 2018.

[7] R. Gareus, "robtk." https://github.com/x42/robtk, 2018.

[8] S. Jackson, "Infamous plugins." https://github.com/ssj71/infamousPlugins, 2018.

[9] F. Coelho, "Dpf." https://github.com/DISTRHO/DPF, 2018.

[10] W. Light, "Rutabaga." https://github.com/wrl/rutabaga, 2018.

[11] "Juce." https://juce.com/, 2018.

[12] D. Robillard, "Pugl - cross platform windowing abstraction layer." https://drobilla.net/software/pugl, 2018.

[13] M. Mononen, "nanovg - canvas api for opengl." https://github.com/memononen/nanovg, 2018.

[14] A. Makhorin, "Glpk linear programming kit manual." http://www.gnu.org/software/glpk/glpk.html, 2014.

[15] Y. M. Matsumoto *et al.*, "Mruby - embeddable ruby interpreter." https://github.com/mruby/mruby, 2018.