Praxis LIVE - hybrid visual IDE for (live) creative coding

ABSTRACT

This paper introduces Praxis LIVE, a hybrid visual IDE and actor-model inspired runtime for (live) creative coding. Originally designed as a rapid development environment for audio-visual projects, Praxis LIVE blurs the lines between visual dataflow (patcher) and code editing, allowing components to be written and extended on-the-fly. Recent releases have placed increased emphasis on Praxis LIVE’s utility as an environment for live-coding alongside its use as a development tool. This paper discusses Praxis LIVE’s runtime architecture, its visual & code editing support, comparisons with similar tools, and some pros and cons of its design.

1 Introduction

Praxis LIVE (Smith) is an open-source hybrid visual IDE and runtime environment for (live) creative coding, running on the Java Virtual Machine, and blurring the lines between visual patchers (Puckette 1988) and code editing. Originally designed as a rapid-development tool for audio-visual projects, Praxis LIVE has always supported live reconfiguration and recoding. Releases since 2015 have increased emphasis on live and interactive coding, focusing on the concept of a visual patcher where all components can be rewritten on-the-fly. Its primary features include -

- A concurrent runtime designed for working with multiple, real-time media pipelines, possibly distributed across multiple processes or machines.
- Built-in generic dataflow components, along with specific components for video, audio and TinkerForge (open-hardware electronics). Support for OSC / MIDI bindings and custom GUIs.
- Compiler-as-a-service, with built-in components defined as source code that can be forked, extended and recompiled into a running pipeline.
- A hybrid-visual IDE that provides a graphical editor for visualising and live editing dataflow pipelines, and an integrated code editor with code completion and error highlighting for live rewriting components.

2 Architecture

The underlying actor-model (Hewitt 1977; Agha 1986) inspired architecture of Praxis LIVE is essential to its utility for live and interactive coding. Lock-free messaging between pipelines and various system services provide the ability to atomically swap in new components, code or resources. Components (actors) in Praxis LIVE are arranged in trees. Each component has an address in a familiar slash-delimited format (eg. /audio/filter). There is no single root - / has no meaning. Instead, the first element of the address refers to the root component, of which there can be many - a hub of roots creating a forest of actors. Roots generally encompass a thread of execution.

Praxis LIVE diverges in various ways from the typical actor model. Mailbox functionality is handled by each root, with external messages being delivered through a single lock-free queue. Each component has zero or more controls, which act as end points for messages and are addressed in a form that echoes function calls (eg. /audio/filter.frequency). Controls may wrap simple properties, actions, or complex functions that set off a chain of further messages. Praxis LIVE adheres to a principle of no shared state between components, and control call communication uses a simple immutable type system.

As well as controls, components can have input and output ports, sometimes reflecting the same data. Ports provide a lightweight way for sibling components to communicate, and it is port connections that are reflected in the visual dataflow editor. Port communication itself is synchronous, although components can create or delay onward port messages, or use them to trigger asynchronous control calls. Port communication uses the same immutable type system, but extends this with the ability to pass mutable data such as audio buffers and video frames.
Alongside an unlimited number of user root components, there are various system-defined roots. Some of these roots provide services such as the compiler or resource loader. The runtime is modularized and decoupled, and a service discovery mechanism is used by components to find the addresses of services they require. Another of these system roots is the IDE itself. The IDE can only communicate with the runtime via message passing, and other components have no knowledge of the IDE, making it easy to swap in another editor that represents the underlying model in a different way.

2.1 Distributed root hubs

A key advantage of this architecture is its ability to work transparently with roots running in separate processes. This makes it possible to edit code and dataflow across multiple machines. It also allows a project to run across multiple local processes, eg. to improve real-time performance by mitigating garbage collection effects. The current mechanism for distributed root hubs takes a master and slaves approach, translating internal messages to Open Sound Control (OSC) messages for inter-process communication. All features required to control the runtime are exposed through this OSC interface, opening up the possibility of a non-Java editor controlling a Praxis LIVE runtime in slave mode.

3 Live visual editing

The graph editor provides the primary editing interface for most root types (audio, video, etc.) within the IDE. It provides a visual representation of dataflow. Components and connections may be added or removed at any time, properties viewed and updated, new resources loaded, etc. The IDE uses a proxy model of the real components, which may not be in the same process or even on the same machine - the representation is cleanly separated and selectively synced as required (eg. for display, save, etc.).

A palette provides a list of built-in components to drag into the graph, encompassing video playback and capture, video mixing and effects, audio sample players and effects, and a range of components for processing control messages. Control components include lots of support for exploring temporal patterns in a live-coding context. These include timers (system and soundcard clock), unlimited numbers of which may be added to trigger events. Message delays and filters may be used to implement forms of temporal recursion, and there are a range of sequencing components from simple list iteration to an 8-channel tracker with table editor. And of course, any instance of one of these components can be forked and rewritten at any time.

4 Live coding

Components in Praxis LIVE contain two primary parts - a user-editable delegate that handles behaviour and a context that hooks that delegate into the surrounding architecture. User code is defined as a Java class body - fields and methods without class definition. Java’s limited ability to alter the code of a loaded class is used as an advantage, ensuring no state is carried forward by default from one iteration of code to the next. Instead annotations link fields and methods to ports and controls on the context. Doing this helps ensure the user’s code reflects the full definition of the component, and thus able to be saved to disk and recreated.

The core:math:add component is one of the simplest built-in components. It adds a value to any number received on its input port and outputs the result - note the use of annotations in the following code to define these ports. The @P annotation defines a property control with port. This code can be edited at any point. When the new code is hot-swapped in, the context will automatically set the value field. The context also always reflects the delegate - were the value field instead removed then its associated port and control would automatically disappear.

```java
@P() double value;
@Out() Output out;

@In() void in(double x) {
    out.send(x + value);
}
```

Property field types are restricted to those that map to the immutable type system, or the special Property wrapper type. This type includes the useful ability to set an animation function that works seamlessly across code updates - eg. using @T to create a port and control to trigger a method and animate value -
void animate() {
  value.to(random(10), 0).in(2, 0.5).easeInOut();
}

All code is delivered as text to a code property on the component, before forwarding to a service for compilation and code analysis. Treating code like any other text property means that the user can write components that rewrite the code of other components, or selectively open up code to OSC control. Compile-time errors keep the previous delegate in place, and are also highlighted in the code editor (runtime errors get caught and reported).

Figure 1: Praxis LIVE graph and code editors with a 3D transform component being edited. Note the relationship between code and visual view, and error highlighting in the code editor.

4.1 Live coding video / OpenGL

Video coding is based on the Processing API (Processing Foundation). User code has almost unlimited access to Processing’s 2D and 3D OpenGL renderers, including live-codeable custom GLSL shaders. However, access is carefully wrapped to work around Processing’s large amount of global state and ensure each component is fully encapsulated, while maintaining the impression the user is working with multiple independent Processing “sketches”. Annotations again connect user code to the environment using familiar Processing types -

public void draw() {
  copy(in); release(in); // zero-copy input (Praxis LIVE extension)
  blendMode(MULTIPLY, mix);
  image(mask, 0, 0, width, height); // scale mask to output
}
4.2 Live coding audio and DSP

The audio coding API mixes unit generators (ugens) with the ability to pass in arbitrary low-level DSP functions. The @UGen annotation injects and copies across ugens. At injection time, ugens are reset and disconnected, with some useful transient state maintained, such as delay buffers or oscillator phase. Output ports make use of the switch-and-ramp technique (Puckette 2006) to filter discontinuities. Built-in ugens encompass sample playback, band-limited oscillators, filters, delays, chorus and reverb. This code defines a component that modulates an input signal with a partial sine wave (simple function example):

```java
@In(1) AudioIn in;
@Out(1) AudioOut out;
@UGen LFO lfo;
@UGen OpGen op;
public void setup()
{
    lfo.waveform(SAW).frequency(0.5);
    op.function(d -> 0.5 * sin(d * HALF_PI) + 0.5);
    link(lfo, op, mod(in), out); // mod() implements amplitude modulation
}
```

5 Comparisons with other tools

Praxis LIVE obviously shares some ideas and features with other tools for live and interactive coding. There are parallels with Extempore (Sorensen), and Praxis LIVE has been influenced by Sorensen’s ideas on cyberphysical programming (Sorensen and Gardner 2010). Praxis LIVE also provides a hot-swappable runtime, flexible concurrency and compiler-as-a-service with the ability to live edit low-level code such as DSP. Unlike Extempore, it does not provide its own language, looking to visual patching for higher-level expression. It chooses the Java VM over LLVM, trading a modicum of performance for the advantages of managed memory.

Field (OpenEndedGroup) is another hybrid visual system on the Java VM that supports live-coding Processing. Praxis LIVE does not provide such a radical interface, but fluid switching between two conventional paradigms (visual patcher and code). Unlike Field, it wraps Processing in a far more modular way, and provides a concurrency model more suited to working with multiple real-time (distributed) pipelines.

Many of the patcher environments that have influenced Praxis LIVE also include the ability to include custom coded behaviour, but usually in a scripting language with performance and feature penalties. Praxis LIVE differs in keeping to the principle of user code as a first-class citizen, never at a disadvantage from the built-in components.

6 Conclusion

This paper has offered a short introduction to Praxis LIVE and its use for live and interactive coding. Praxis LIVE has received positive feedback for its hybrid visual editor interface, and the intuitive relationship between code and visual representation. The interface has proven an expressive way of working with and comprehending even large projects. The forest of actors architecture and rigid approach to concurrency has generally paid dividends for both performance and stability. While not easy to benchmark, experience suggests any overhead added by message dispatching and immutable types is outweighed by the benefits of a lock-free architecture; while the ability to work across processes dramatically improves real-time performance with the Java VM.

Of course, there are compromises to be made too. The current immutable types make it hard to work with complex data (e.g., geometry) efficiently, although work is ongoing here. While no shared state between code iterations is beneficial, it is also hit by this issue with complex data, making some things hard to live-code smoothly and efficiently. The strict separation of roots always as if they’re in separate processes places limits on the richness of information that can be shown efficiently within the visual editor. This separation also limits the ability to “show us your screen” when working with video, as the editor and video pipeline cannot share a rendering surface (although there is a workaround capturing the editor as a video stream).

Over time, Praxis LIVE has proven to be both performant and resilient, used by the author and others for live performances and creating installation work. It continues to be actively developed, with a community building around it. Videos of it in action can be seen at http://www.praxislive.org/iclc-videos/


