Performative Code: Strategies for Live Coding Graphics

Shawn Lawson
Rensselaer Polytechnic Institute
lawsos2@rpi.edu

ABSTRACT

Performing real-time, live coded graphics requires a streamlined programming environment, efficient implementation techniques, improvisatory inventiveness, a tuned ear, and above all, aesthetic sensibility. The performer must not only pull together these concepts, but maintain an awareness and forethought, of the graphical results of and performance expectations of the live coding environment.

1. Introduction

Live coding is a unique and extend-able area of computer-based research and artistic practice, as evidenced by the wide variety of integrated development environments (IDEs) that currently exist for live coding.[1][2][3]

This text includes observations, thoughts, and discoveries the author has made over the past year of developing and performing with his own live coding IDE.[4] Specifically, this text addresses the challenges the live-coding graphic artist faces when collaborating with an audio performer on a set piece, including primarily the necessity for for an efficient and accurate coding environment. The IDE, which runs in Google Chrome, uses the OpenGL fragment shader language, features autocompilation and execution, has a customizable autocompletion engine, real-time FFT analysis and Open Sound Control (OSC) connectivity, and supports multi-output projection-mapping. Code writing and editing occurs in a text editor layer on top of the computed visual output.

While the autocompilation and autocompletion features contribute to the performer's speed and flexibility in a live situation, one must be aware of code artifacts, lines or blocks inadvertently uncommented or simply unintended, must be taken into account. While introducing the potentials for interesting discoveries, these artifacts may also produce undesirable effects, and by understanding these fallibilities to the degree that their implementation, expected or not, can be successfully integrated, the live coder expands his or her aesthetic toolbox beyond what is already known.

2. Language Hacks

The OpenGL shader language (GLSL), which executes directly on the graphics hardware, not sandboxed by another environment, is syntactically strict. Below are several low-level techniques that have become integral my performance practice; techniques that have improved aesthetic flexibility and coding speed, while satisfying the aforementioned syntactical requirements. These descriptions are specific to GLSL, but their concepts are applicable to other IDEs and languages.

2.1. Safety First

It is common practice to initialize a value at variable declaration, not only to prime each variable for instantaneous implementation, but to eliminate the potential for bugs and unexpected results. In the code sample below the value of color1 is unknown, whereas the value of color2 is black. In its current, undefined instantiation, adding *red* to color1, would likely return an unknown, potentially jarring result depending on the context. Color2, which is initialized to be black, can more accurately predict the result of adding red: black becomes increasingly red.

[1]"Live Coding." 2015. http://en.wikipedia.org/wiki/Live_coding
[2]"Software." 2015. http://toplap.org/category/software/
[3]"ToplapSystems." 2015. http://toplap.org/wiki/ToplapSystems
[4]Link to IDE *The_Force*: http://shawnlawson.github.io/The_Force/ Link to github repository: https://github.com/shawnlawson/The_Force

```
//vec4(red, green, blue, alpha) with 0.0 - 1.0 range.
vec4 color1;
vec4 color2 = vec4(0.0, 0.0, 0.0, 1.0);

color1 = color1 + vec4(1.0, 0.0, 0.0, 1.0);
color2 = color2 + vec4(1.0, 0.0, 0.0, 1.0);

//  unknown output color
gl_FragColor = color1;
//  output color will be black plus red
gl_FragColor = color2;
```

Furthermore, if the a performer decides to delete the line of code that adds red to color2, the outgoing fragment color would revert back to the initialized value of black, thereby reducing the potential for rogue values to modify the color if that line is implemented in some other process(es). Similarly, if the performer deletes the line of code that adds red to color1, the resultant value is unknown, and may lead to an undesirable, or at least unexpected outgoing fragment color.

2.2. Commenting

A simple method of working within the functionalities of an auto-compiling and executing IDE is to write code behind a single line comment maker (see example below).

```
//line of code written, ready to be uncommented
//vec4 color1 = vec4(1.0, 1.0, 1.0, 1.0);
```

It has proven, effective to implement a key command into the IDE to toggle comment/uncomment on the current line(s). The potential pitfall of this functionality is that if the code behind the comment marker has errors, they won't be apparent until after the uncommenting (which will be addressed in the next subsection), but as a process for hiding/revealing code from/to the compiler quickly, it is generally effective. Specific to GLSL, if lines or blocks of code are not directly needed then commenting out portions of code can have large performance improvements, specifically regarding frame rate and rendering resolution, but also legibility.

As in most programming environments, comments can be used to organize and structure. An IDE with a color syntax highlighter on a static background color will substantially reduce the performer's visual search for specific code locations. When the code lengthens to the degree that it scrolls off the screen, a higher level of search can be implemented using single line comments. In the comment below, the performer can do a quick visual scan of the reverse indented, color-coded comments, then narrow the search to the specific code desired, as opposed to the performer hunting through the syntax-colored code on what may be a similarly-colored, animated background. In the code example the comment color is light blue whereas in the IDE it is grey, causing it to stand out more, not dissimilar to a written text's section headings that may be in bold and larger font. The offset and color difference simply makes the comment easy to find and mentally parse.

```
    vec4 color1 = vec4(1.0, 1.0, 1.0, 1.0);
    vec4 color2 = vec4(0.0, 0.0, 0.0, 1.0);

//H Bars
    color1 = color1 * step(.4, fract(gl_FragCoord.x * .1));
    color2 = color2 * step(.6, fract(gl_FragCoord.x * .2));
```

2.3. Conditional Help

As previously mentioned, there are potential issues with the single line commenting method. Below is a workaround to those potential pitfalls that subtly exploit the auto-compiler.

```
vec4 color1 = vec4(0.0, 0.0, 0.0, 1.0);

if (false) {
    color1 = color1 + vec4(1.0, 0.0, 0.0, 1.0);
}
```

Using a conditional statement that always evaluates false allows the embedded code to be auto-compiled, but prevents it from executing. This encapsulation within a conditional statement provides the performer with a safety net for building complex or detailed code sections without worry for inadvertent or premature compilation. When successfully compiled, the performer can utilize the comment method to hide this code from the compiler, and safely remove the conditional wrapper (this is expanded in the snippets section).

2.4. Decimal Cheats

During a performance, the performer may want to create a transition or continuously change a value. Slowly increasing a value from 0.1 to 0.9 is quite easy; whereas, changing from 0.9 to 1.0 is not. To change the leading 0 to 1 would result in an intermediate value of 1.9 before being able to change the 9 to 0. A similar problem exists in the reverse direction. To change the trailing 9 to 0 would result in an intermediate value of 0.0 before being able to change the leading 0 to 1. Both 1.9 and 0.0 are quite far out of sequence from our value of 0.9.

The cheat is to intentionally commit an error, then rely on the autocompiler's memory. When an error occurs, the autocompiler retains the most recent successfully compiled shader. The performer simply inserts a second decimal point, in this example, next to the existing decimal, resulting in 0..9. Changing numbers on either side of the decimal can now happen without consequence. Once we have our desired value of 1..0, the extra decimal point is removed resulting in a final value of 1.0, which the autocompiler will accept (an additional method towards continuous change is addressed in the external input section).

3. Linguistics

```
vec4 white = vec4(1.0, 1.0, 1.0, 1.0);

//each line below has one error
vec4 color1 = whie;
vec4 color2 = vec4(1.0, 1.0, 1.0 1.0);
```

It is much easier to see the letter "t" missing from the word "white" in the declaration of color1, than to find the comma missing after the third parameter in color2. When performing, typing and text scanning are virtually simultaneous, and without considerable legibility error resolution becomes increasingly difficult. Anxiety and stress mounts when while tracking down errors, because a set performance piece continues; gradually, one falls behind and misses cues.

Debugging color2, under some time pressure, we would mentally check type (vec4); the operation (=); the parameters (count of 4 matches vec4), values (1.0 1.0 1.0 1.0), type (float float float float), separated (yes yes no yes) - finally we have the problem. And, it is a very detail oriented problem, which is not an ideal situation.

In the case of color1, we would mentally check type (vec4); the operation (=); the value (whie) - spelled incorrectly. We're more easily able to locate the spelling mistake than to dig down into the nitty-gritty.

To conclude this sub-section, one is less error-prone when using "white" than "vec4(1.0, 1.0, 1.0, 1.0)", so long as "white" is correctly defined. Plus readability and comprehension are much faster. Predefining commonly utilized variables like colors and functions, as we will discuss in the next subsection, is one way to achieve this.

3.1. Helper Functions

The development and utilization of helper functions aid with the overall performance and help to maintain a graphical pacing and aesthetic interest. In the case of GLSL, a random function does not exist in many Graphics Processing Unit (GPU) hardware implementations. To write a random function for the performance may certainly have some appeal if

it were integral to the primary concept of the performance; however, when needed solely to have some random data to work with, the writing out of helper functions can consume valuable time. A resolution to the issue was to include a select set of often used helper functions into the IDE for random numbers, shapes, and sprite sheet animation.

4. Debugging

While the integration of a step through debugger may be an interesting aspect of one's performance, should the debugging process drag-on too long, the flow of the visual experience may be compromised. In order to avoid a protracted debugging session, a graceful, scale-able debugger can be an excellent addition. One method of subtle error indication is to make a slight change to the line's background color (see Figure 1).

```
1 void main () {
2
3     gl_FragColor = vec4(0.0, 0.0, 0.0 1.0);
4
5 }
```

Figure 1: *Example of a debugger highlighting a line of code that has an error. Note, the missing comma again.*

Should more assistance or information about the error be needed to resolve the issue, a slightly more intrusive technique can be used. In figure 2, error markers are placed in the line number column. Additionally, when the cursor hovers over one of these markers a debug message is displayed. There is some efficiency lost with this method; because, the performer must pause typing, move the cursor, read, move the cursor to hide the message, and finally start to type again.

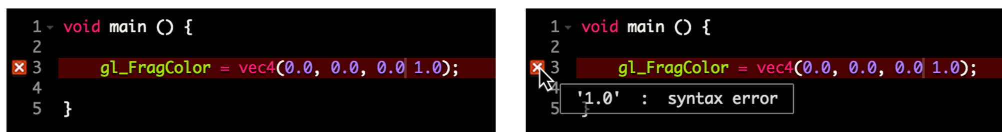

Figure 2: *Left: Example of an error marker. Right: Revealed debug message.*

An even more visually intrusive method would be an in-line debug message information system (see Figure 3). While this example is visually disruptive, especially when it shifts down lines of code when inserted, it is significantly faster for the performer to see the debugging information and make a correction.

```
1 void main () {
2
3     gl_FragColor = vec4(0.0, 0.0, 0.0 1.0);
  '1.0' : syntax error
4
5 }
```

Figure 3: *Example of a debugger providing additional information, in-line to the code. Note, again, the missing comma.*

All of these approaches detract less from the overall show experience by avoiding a separate panel with a GDB style breakpoint debugger. And, as with most things, with experience and practice, these assistive debugging indicators can be scaled back as the performer gains confidence and proficiency.

5. Autocompletion and Snippets

Many autocompletion engines exist in many forms, but it is important that they are *configure-able* for one's specific needs: accuracy and speed.

5.1. Reducing and Appending

The OpenGL specification for WebGL contains many built-in OpenGL functions. By default, many autocompletors will contain all the OpenGL functions, storage classes, built-in math functions, reserved words, etc. To reduce clutter from the autocompletor, anything relating to the vertex shader is removed. Furthermore, anything that is not commonly used

Figure 4: *Example of autocompletor showing possible known completions.*

is removed. Predefined variables and functions are included in the autocompletor list, for example: colors, shapes, helper functions, and uniform names. This custom reduction and addition creates a highly optimized list, meaning that in most cases our desired autocompetion match is reduced to a single solution faster.

5.2. Defining Snippets

Snippets are generally small code blocks that are entered to save time or reduce errors. The snippets are written by the user into a configuration file. The autocompletor's engine builds snippets from the file as seen in the code example below. The first line is a comment that appears when the autocompletor's snippet pop-up appears. The second line starts with the *snippet* declaration, followed by the snippet's *key*. The third line is the snippet's *value*. This particular snippet enters the circle helper function. To breakdown further, *${}* indicates the location of a parameter, and where the next tab-jump point is. The number inside the curly braces gives the tab-order of the parameters. The text following the colon will be the default value for those parameters. In this case, the word *float* is a hint as to the type of value that needs to be entered. Note the fifth tab-point was added as a quick way to jump outside the parenthesis.

```
# circle
snippet cir
    circle(${1:float}, ${2:float}, ${3:float}, ${4:float})${5}
```

In figure 5, the circle fuction has been autocompleted. The first value is highlighted, meaning it will be overwritten when typing begins. Pressing “tab” will jump to the next “float” parameter and highlight it for overwriting.

Figure 5: *Example of autocompletor completing the circle function snippet.*

5.3. Common Snippets

Some testing has led to the implementation of a variety of snippets; the following are a few of the key:value pairs integrated into the IDE. The first example is a for loop with conditional check against an integer (a float version also exists with the *forf* snippet key). The second example, *iff*, we recognize from the conditional help section above. Last is a selection of commonly used data types.

```
//snippet fori
for (int i = 0; i < int; i++) {

}

//snippet iff
if (false) {

}
```

```
//snippet vv
vec2
//snippet vvc
vec3
//snippet ft
float
```

6. External Input

Many audio visual performers use hardware input devices to launch processes, activate functions, adjust values, or simply inject a sense of physicality into the live performance. In a live coding performance, additional hardware devices can similarly help to improve efficiency, facilitate real-time and continuous access to parameter values, and inject a more perceptible layer of physicality. Additionally, performers frequently configure their software to communicate with each other. This inter-performer communication adds a rich cohesiveness.

For this IDE, the Open Sound Control (OSC) protocol was implemented. This was chosen due to a familiarity with OSC, a need to receive OSC messages from both hardware (iPad with Lemur[5]) and other performers, and in preparation for the future possibilities that may include a variety of data types and ranges.

How might this be used? Let's say that an OSC device produces data that ranges from 0.0-1.0 and is coming into an GLSL shader as a uniform labeled *fader*, see the code example below.

```
uniform float fader;

void main() {
    vec3 black = vec3(0.0, 0.0, 0.0);
    vec3 white = vec3(1.0, 1.0, 1.0);
    color = mix(black, white, fader);

    gl_FragColor = vec4(color, 1.0);
}
```

With an external OSC controller, a performer could blend or mix between the black and white colors. This method could be applied to back buffers for adding motion blur, mixing effects, cross-fading between effects, or provide more accurate cuing with audio transitions. While seemingly trivial, in the above example, this OSC technique provides a stream of continuous numbers whereas the deliberate typing of numbers into the autocompiler may have very jarring or unexpected/undesired outcomes. This input method resolves the 0.9 to 1.0 discontinuity problem as mentioned in the decimal cheats section.

7. Conclusion

Live coding introduces a particular set of technical, performative, and aesthetic challenges. While this is certainly not intended to be a comprehensive survey of these issues and their potential solutions, it is meant to contribute to the already emerging dialogue, framed primarily by the author's work. To this end, the aforementioned strategies for live coding graphics are applicable to other areas of live coding.

8. Acknowledgments

Thank you to my audio collaborator, Ryan Ross Smith.[6] And, thank you to CultureHub for providing us the time and space to continue building the *Sarlacc* set.[7][8]

[5] App interface building and performance IDE. https://liine.net/en/products/lemur/
[6] http://ryanrosssmith.com
[7] http://www.culturehub.org
[8] Video link to *Sarlacc*: https://vimeo.com/121493283