Let's Fix OpenGL

Adrian Sampson

Cornell University
asampson@cs.cornell.edu

Abstract

From windowing systems to virtual reality, real-time graphics code is ubiquitous. Programming models for constructing graphics software, however, have largely escaped the attention of programming languages researchers. This essay introduces the programming model of OpenGL, a ubiquitous API for real-time graphics applications, for a language-oriented audience. It highlights six broad problems with the programming model and connects them to traditions in PL research. The issues range from classic pitfalls, where established thinking can apply, to new open problems, where novel research is needed.

1998 ACM Subject Classification D.3.m Miscellaneous

Keywords and phrases language design; real-time graphics; OpenGL; GPUs; heterogeneity

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Throwing Some Shader

Nearly every consumer computing device on the market, from smartwatch to workstation, comes with a graphics processing unit (GPU). Any software that renders graphics in real time must exploit a GPU for reasonable performance, which entails programming to one of the mainstream APIs that graphics cards support. GPU vendors have settled on two common GPU interfaces, OpenGL [44] and Direct3D [33], so graphics software almost exclusively builds on one of these two APIs.

OpenGL and Direct3D may have been created as hardware abstractions, but they do double duty as programming models. The two APIs use a common structure consisting of two components: a full-fledged programming language for writing programs that run on the GPU, and a set of C functions for communicating between the CPU and an attached GPU. Both components contend with a vast array of classic problems in programming languages: abstraction and reuse; the need to avoid obscure run-time errors; expressiveness without sacrificing performance; and so on. The APIs, however, have largely avoided adopting the answers that programming languages research has developed to these problems—even basic, conventional wisdom in our community has escaped the design of graphics APIs.

This essay introduces OpenGL and its pitfalls for the PL-minded reader and advocates for more research that applies language ideas to this underserved domain. It enumerates six language problems that OpenGL programmers face and proposes possible directions for solving them. Some problems invite straightforward applications of established traditions in PL research, and others are open problems without clear solutions. Despite its difficulties, GPU-accelerated graphics programming is enormously popular—it underlies a $90 billion global video game industry, for example [34]—so research that addresses its shortcomings has potential for real-world impact.

Graphics programming also represents the tip of the spear for *heterogeneous programming*, the general problem of orchestrating separate, specialized hardware units in a single program. As the capabilities of traditional CPUs stagnate, software will need to exploit increasingly

```
// Vertex shader:
in vec4 position;
in float dist;
out vec4 fragPos;
void main() {
  fragPos = position;
  gl_Position = position + dist;
}
```

```
// Fragment shader:
in vec4 fragPos;
void main() {
  gl_FragColor = abs(fragPos);
}
```

Listing 1 A GLSL shader pair.

exotic hardware to continue making advances [40]. Ensembles of oddball hardware beyond the GPU, from FPGAs to fixed-function units, will only increase the need for heterogeneous programming models with the same set of challenges as OpenGL.

2 Graphics Programming with OpenGL

This section dissects a tiny OpenGL program.[1] While this essay does not illustrate Direct3D directly, the programming model there is similar and exhibits similar pitfalls.

2.1 Shader Programs

The soul of a real-time graphics application is its *shader programs*. A shader is a short program that runs on the GPU as part of the rendering pipeline to define the shape and appearance of objects in the scene. There are several kinds of shaders, but the two most common are the *vertex shader* and the *fragment shader*, which respectively compute the position of each vertex in 3D space and the color of each pixel on an object's surface. In OpenGL, shaders are written in the special-purpose GLSL programming language, which is a variant of C. Direct3D has its own shader language, HLSL, which is a similar but incompatible C variant.

Listing 1 shows a vertex and fragment shader in GLSL. Each shader consists of a `main` function and some global definitions. The global definitions use `in` and `out` qualifiers to mark variables that represent the shader's inputs and outputs. In this vertex shader, for example, a `position` vector and a `dist` scalar both come from the CPU. This shader assigns the magic `gl_Position` variable to this parameter—this is the vertex shader's output. The `position` value is only available at the first stage—in the vertex shader—so more work is required to pass it along to the fragment stage. This shader pair declares a second variable, `fragPos`, in both programs to hold the `position` value from the vertex stage and make it available in the fragment stage. Finally, the fragment shader uses `fragPos` as an input to compute its output: the `gl_FragColor` magic variable.

2.2 Shaders are Strings

GLSL code only runs on the GPU. *Host code* on the CPU uses a traditional general-purpose language—usually C or C++. To draw an object in a 3D scene, the host code needs to compile the GLSL source code to the GPU's internal instruction set, send its parameters,

[1] Details are omitted here for focus. Full source code is online: `http://adriansampson.net/doc/tinygl/`

```
// Embed shader source code in string literals.
static const char *vertex_shader = "in vec4 position; ...";
static const char *fragment_shader = "in vec4 fragPos; ...";

// Compile the vertex shader.
GLuint vshader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vshader, 1, &vertex_shader, 0);

// Compile the fragment shader.
GLuint fshader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fshader, 1, &fragment_shader, 0);

// Link the pair together.
GLuint program = glCreateProgram();
glAttachShader(program, vshader);
glAttachShader(program, fshader);
glLinkProgram(program);
```

Listing 2 Compiling a shader pair.

and invoke it. Each GPU vendor uses a different internal representation for shaders, so GLSL source is the *lingua franca* that provides compatibility: every GPU driver includes its own GLSL compiler.

Listing 2 shows the C boilerplate for compiling and linking a vertex/fragment shader pair. Here, the GLSL source code is embedded in the executable using a string literal; it is also common to use `fopen` to load the source from a text file when the program starts up. Later, to draw an object in a frame, the host code uses the `program` reference to tell the GPU which linked shader pair to use when drawing an object.

2.3 CPU–GPU Coordination

To supply the shaders' inputs, the host code looks up *location* handles for each `in` variable in the GLSL code. There are two main options: the shader code can mark each variable with a fixed index, or the host code can look the variables up by name. Listing 3 shows the latter, which manifests as a series of `glGet*Location` calls.

Our example shaders use two kinds of input variables: `position` is a *vertex attribute*, meaning that it takes a different value for every invocation of the vertex shader's `main` function; and `dist` is a *uniform*, so it remains constant across the object's vertices. For attributes, the program needs to allocate a *buffer* representing the GPU's memory region for the variable.

Finally, to draw each frame, the program selects the compiled shader pair with a call to `glUseProgram(program)`. Then, to provide a value for the `position` attribute, it executes `glBufferSubData` to copy data from the host memory—i.e., a plain C array—to the GPU-side buffer. For the uniform, the render loop uses a `glUniform*` call to set the variable.

3 Problems & Potential Solutions

Even this abridged example should raise some language-design alarms in the mind of a PL researcher. The problems start, but do not end, with the ordinary infelicities of any aging C API design: hidden state, minimal static safety checks, and so on. This section

```
// Setup code:

// Look up shader variable locations.
GLuint loc_position = glGetAttribLocation(program, "position");
GLuint loc_dist = glGetUniformLocation(program, "dist");

// Allocate a buffer for the attribute.
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glVertexAttribPointer(loc_position, size, GL_FLOAT, GL_FALSE, 0, 0);

// ...
// In the render loop:

glUseProgram(program);

// Copy the vertex positions into the buffer.
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);

// Set the uniform variable.
glUniform1f(loc_dist, 4.0);
```

Listing 3 Communicating with the shaders.

enumerates six obstacles in graphics programming. The first problems are classic pitfalls with established answers in the PL literature on language extensibility, static safety, and metaprogramming. Here, real-time graphics represents a new application domain for existing lines of research. The final three issues expose new open problems that pertain specifically to graphics: rate-oriented language formalisms, type systems for linear algebra, and defining "correctness" for visual systems.

3.1 Shader Languages are Subsets of Supersets of C

GLSL and its Direct3D equivalent, HLSL, are for the most part plain, everyday imperative programming languages. They have variable declarations, `if` conditions, `for` loops, function calls, and global mutable state—just like any ordinary imperative language. Shader languages, however, are unique, one-off reinventions that *resemble* C without being clean extensions. The distinction makes life more difficult for programmers, who need to carefully keep track of the subtle differences between GLSL and "real" C. In C, for example, the name of the type declared by `struct t {...}` is `struct t`; in GLSL as in C++, the name is just `t`. In C++, variable declarations can appear inside an `if` condition; in GLSL as in C, they cannot. These myriad incompatibilities make it difficult to move code between the CPU and GPU.

The need for *ad hoc* language extension also complicates compiler implementations: current compilers either need to reinvent a complete C-like parser and compiler from scratch [28] or hack an existing frontend such as GCC or Clang. Both approaches are error prone: recent work by Donaldson et al. [15, 16] has revealed crashing bugs in a staggering array of vendor-supplied GLSL compilers. Apple's new Metal shading language [5] is based on the C++14 standard, but even it relies on a custom Clang fork with informal restrictions on certain features, such as subclassing and recursive calls.

Potential solutions. Shader languages' needs are not distinct enough from ordinary imperative programming languages to warrant ground-up domain-specific designs. They should should instead be implemented as extensions to general-purpose programming languages. There is a rich literature on language extensibility [27, 36, 39] that could let implementations add shader-specific functionality, such as vector operations, to ordinary languages. The potential productivity benefit is large, especially for host languages other than C: programmers could opt for more diverse languages without needing to context-switch to C-like syntax and semantics to write shader code.

Some existing work implements *embedded DSLs* for generating GLSL code [2, 6, 7, 9, 10, 17, 30–32, 43], which is a useful first step. But this embedded approach yields code that looks much different from the host language. It also typically requires that host programs generate GPU code on the fly, at run time. A language-extension approach could match the syntax and semantics of the host language without incurring the cost of dynamic code generation.

3.2 Loose CPU-to-GPU and Stage-to-Stage Coupling

In OpenGL, interactions between the CPU and GPU are *stringly typed:* the `glGet*Location` calls in Listing 3 look up variables in the shader programs by their names. Communication between shaders is similarly brittle: the two separate programs in Listing 1 need to agree on a name for `fragPos`, which must not conflict with the CPU-to-GPU name `position`. Even though both C and GLSL are statically typed languages, neither compiler can statically check their naming agreement. Shader source code is only compiled after the host code begins executing, and the program might arbitrarily pair vertex and fragment shaders together.

The lack of static semantics comes with all the same productivity pitfalls as programming in a dynamic language like JavaScript. Typos in variable names are not reported until run time; type errors are similarly deferred; refactoring tools in IDEs are hobbled; and static compilers must conservatively eschew optimizations. Regardless of opinions on static typing, programmers tend to agree that at least an unsound, optional *lint*-like static checker can be helpful—but OpenGL programmers do not enjoy even that basic luxury.

Potential solutions. Language research should endeavor to clean up the abstractions between shader code and CPU code. At a bare minimum, CPU–GPU and inter-stage communication must be made type safe. In the near term, researchers should explore backwards-compatible approaches to giving static semantics to complete C++/GLSL hybrid programs. A program analysis could ingest the OpenGL API calls in the host code and the variable declarations in the shader code to check that they align and to propagate type information between the two languages. A sound analysis that rules out any possible CPU–GPU disagreement may be too difficult to achieve, but even a best-effort checker could help avoid unnecessary run-time failures.

In the long term, more research should unify CPU–GPU programming in a single language that spans the CPU and all GPU stages. Communicating a value from the vertex stage to the fragment stage should introduce no more syntactic or cognitive overhead than defining and referencing a variable. Instead of relying on the programmer to divide the complete computation into stages, the compiler should take responsibility for splitting CPU host code from GPU shader code.

In a hypothetical unified programming model, the primary question is how much to rely on compiler automation and where to use explicit programmer control. A binding-time analysis [35], for example, could automatically determine the earliest possible stage for each computation, but earlier is not always better: running code once on the CPU and communicating it to the GPU can be more costly than running the same code redundantly on the GPU. Two recent languages from the graphics community, Spark [20] and Spire [25], propose to use a type system instead. Type annotations let the programmer control where and when each expression in a unified program is executed. I am currently exploring a similarly explicit design where a multi-stage programming language [47] models the GPU's pipeline stages.

3.3 Massive Metaprogramming

Performance is a first-order concern in real-time graphics, so programmers need to avoid all unnecessary overhead in shader programs. To avoid the overhead that comes with generality, applications typically generate many specialized variants of more general shader programs called *übershaders* [23]. An übershader for metal materials, for example, might combine many parameterized effects to support different settings for color, shininess, damage, rust, texture, and so on. Übershaders are convenient for artists and other non-programmers, who can tweak parameters to design a specific effect without writing any code. But these monolithic designs pay a performance penalty for their generality: pervasive parameters incur CPU–GPU communication overheads and add costly branching to the shader code.

To avoid these overheads, some implementations recover efficient shader code by generating *specialized* shader programs that "bake in" each set of parameters and strip out unneeded functionality. Shader specialization occurs at a massive scale: modern video games can generate hundreds of thousands of shader variants [25, 26]. The only tool OpenGL offers for shader specialization, however, is the C preprocessor with its familiar `#define` and `#ifdef` directives. Unhygienic token-stream rewriting may not be so bad for small-scale metaprogramming, but it does not scale to large-scale shader specialization. Graphics programmers resort to developing *ad hoc* toolchains to stitch together snippets of GLSL code into whole shader programs [49].

Potential solutions. The urgent need for programmable specialization of general shader programs is an opportunity for metaprogramming research. Graphics programmers should be able to write and reuse libraries of tactics for manipulating shader code for efficiency. Both run-time and compile-time metaprogramming can be useful: while it is less common in current practice, dynamic specialization could eliminate some shader overhead that is out of reach for static techniques.

Metaprogramming techniques from the programming languages community are up to the task. They can enforce safe program generation [47], allow composition of compile-time macros from multiple, independent libraries [19], and even incorporate dynamic profiling data [12]. Shader specialization represents an opportunity to stretch the scalability of this classic work. Where most work on metaprogramming focuses on implementing language extensions or generating a single target program, shader specialization requires the system to synthesize thousands of variants and choose between them at run time. The massive scale creates new challenges: programmers may need new mechanisms to *limit* specialization, for example, to stay within practical limits.

3.4 Informal Semantics for Multiple Execution Rates

Each stage in a GPU's graphics pipeline runs at a different rate. Interactions between the rates have subtle implications for the semantics of complete, multi-shader programs. The fragment shader, for example, runs many times for every execution of the vertex shader: it interpolates the pixels between adjacent vertices on a surface. The values passed between the vertex and fragment stage are also interpolated. The `fragPos` variable in Listing 1 is exactly equal to `position` in the vertex shader, but it takes on interpolated values in the fragment shader. Therefore, an expression involving `fragPos` has subtly different semantics depending on which stage it appears in. The story gets more complicated with other shader types: *geometry shaders*, for example, operate on multiple adjacent vertices simultaneously.

The OpenGL standard defines the meaning of each stage individually. It does not attempt a general theory for the semantics of arbitrary shader rates and their interaction. If future generations of GPUs introduce new programmable stages to the graphics pipeline, each new rate will need a new *ad hoc* definition. Some work defines the semantics of *general-purpose* GPU programming models such as CUDA [22, 24, 29], but these simpler GP-GPU languages do not have multi-rate execution or fixed-function interpolation logic. Programmers are left with only informal descriptions of the semantics of interacting systems of shader programs.

Potential solutions. Language research should develop a core calculus for massively parallel, multi-rate programs. Programs in a hypothetical λ_{GPU}-calculus would describe how and when state from one stage becomes visible to a set of parallel invocations in another stage. The new multi-rate semantics may resemble an existing multi-stage semantics [18, 47] where control flows linearly through a series of nested stages. Graphics-specific phenomena such as inter-stage interpolation should also be made explicit in this calculus. In λ_{GPU}, researchers could not only formalize the semantics of real, mainstream GPUs but also explore the space of alternative GPU designs to inform future hardware development.

3.5 Latent Types for Linear Algebra

Graphics code—both inside shaders and in host code—consists mainly of vector and matrix operations. Points in space are floating-point vectors (called `vec3` or `vec4` in GLSL) and transformations between vector spaces are represented as 4×4 matrices (the `mat4` type). Every realistic system of shaders needs to juggle a handful of common vector spaces: typically, a *model* space, where vectors are relative to a specific object's position; *world* space, which all objects share; *camera* space, relative to the camera's perspective; and *projection* space, relative to the 2D canvas where the scene will be drawn.

Shader code is correspondingly littered with duplicate variables that represent the same vector in different spaces. For example, most programs pass model, view, and projection matrices to their shaders, each of which can transform from one vector space to the next. Shaders then create camera-space and world-space versions of input vectors and use them in computations. For example, *lighting models* for simulating reflections typically start by computing the angle of light, which involves subtracting the light source position vector from the model's position vector:

```
in mat4 model, view, projection;
in vec4 position;  // in object space
in vec4 light_position;  // in world space
void main() {
  vec4 position_camera = view * model * position;
  vec4 position_world = model * position;
  // ...
  vec4 light_direction = light_position - position_world;
}
```

The subtraction `light_position - position_world` happens in world space. The result would be meaningless if the program instead used `position_camera`: the spaces do not match. There are clearly legal and illegal ways to combine matrices and vectors, but the shader language offers no help with enforcing these rules: programmers resort to naming conventions and boilerplate to keep things straight.

Potential solutions. The vector-space problem in graphics code is an opportunity for type system research. A linear-algebra type system could take inspiration from type systems for units of measure: the type of a vector value would tag it with a vector space. The corresponding transformation matrix would be marked with a pair of vector spaces: the space it translates *from* and the one it translates *to*. For example, a vector `v` might have type `vec4<A>` to indicate that it is in space A, and a matrix `m` of type `mat4<A, B>` would translate from vector space A to B. Using these two argument types, the type system can give the multiplication expression `m * v` the type `vec4<B>`. It is an error to multiply `v` by a different matrix of type `mat4<C, D>` where $C \neq A$ because the result has no meaningful vector space. This hypothetical type system could automate the process of tagging vectors and checking their correspondence. Because a vector space type is defined by a transformation matrix value, such a linear-algebra type system may benefit from exploiting a dependent type system [46].

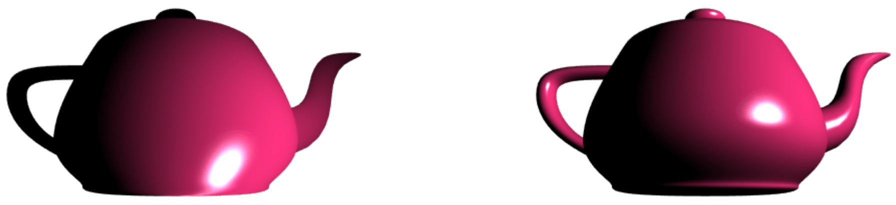
Beyond basic checking, the type system could help *synthesize* the appropriate transformations rather than relying on the programmer to write the boilerplate. For example, a new expression form `v in B` could automatically find the right matrix to multiply by `v` to produce a B-space vector. This implicit approach would avoid the need for a convoluted naming scheme to distinguish `position` vs. `position_camera` vs. `position_world`. Synthesizing transformations automatically would also enable new optimizations: a tool could avoid redundant computation and communication by separating vector-space transformations from the main program text. For example, an expression `(v1 * v2) in B` can be computed by first transforming both vectors into space B and then multiplying them; equivalently, the program might multiply the vectors in some other space and then transform the product. These diverging possibilities form a search space for synthesis.

3.6 Visual Correctness and Quality Trade-Offs

While learning to program my first few shaders, I implemented the textbook Phong lighting model [38], a "hello world" of shader programming. In my first implementation, I made a mistake I cautioned against in the previous section: I used the wrong vector when converting between vector spaces. This single-token bug got lost amid the conversion boilerplate. The result, depicted in Figure 1a, looked ugly: the reflections were too intense and failed to light the entire object. It was not bad enough, however, to raise suspicion—I assumed that the simplistic lighting algorithm itself was to blame. According to my version control logs, the bug stayed in place for *nine months* before I found and fixed it (Figure 1b). The problem was that the result, while incorrect, was plausible enough that it was not *clearly* incorrect.

Testing and verification tools only work when programmers are willing to specify correctness, and specifying correctness is particularly difficult in graphics. The human visual system's tolerance to error makes it challenging to define *correctness* for rendering systems. Is a bug really a bug if most humans do not notice anything wrong with a scene? How do you write a unit test for "visual correctness"? Based on conversations with graphics programmers, testing seems to be very rare: developers instead make incremental changes and spot-check them manually to deem the output acceptable.

Beyond bugs, graphics programmers also *intentionally* compromise visual quality in return for efficiency. Real-time graphics animations are not perfect recreations of the real



(a) Buggy. **(b)** Correct (probably).

Figure 1 Output from a buggy and corrected implementation of the Phong lighting model. The difference is obvious now but was hard to detect without a ground-truth comparison.

world; it is more important that they maintain a high frame rate than for every object to look as realistic as a ray-traced reference image. Applications can even dynamically switch between multiple *levels of detail* for the same object depending on its salience in a given scene [26]. It is typically up to the programmer to manually select and implement quality-compromising optimizations, although some recent graphics work has proposed to automate the process [26, 37, 45, 48].

Potential solutions. Controlling output quality is the central challenge in *approximate computing* research [11, 13, 41, 42]. Researchers should treat graphics programming as an instance of approximate computing: the same set of statistical quality controls could apply.

Software engineering research should seek to understand how graphics programmers currently reason about correctness. What *ad hoc* processes have developers invented to cope with a world where perfect correctness is unachievable and bugs are in the eye of the beholder? With this baseline understanding, languages research can build tools to improve existing modify-and-check workflows. Recent work on live coding [21], for example, could help shorten the cognitive distance from source code modifications to visual feedback. More radical tools could seek to alleviate the need for manual output inspection—for example, by incorporating crowdsourced opinions [8].

4 Postscript

Like any outmoded but entrenched programming model, OpenGL remains universal despite its flaws. Many content designers avoid interacting with graphics APIs directly by building on monolithic *game engines* such as Unity [3] or Unreal [4], which sacrifice flexibility in exchange for abstraction. And real-world programmers can be wary of new language tools from academia, so adoption will be slow for research on graphics programming—even for proposals that unambiguously improve on the status quo.

However, 2017 is a particularly fertile moment for new ideas in real-time graphics programming. The standards body that specifies OpenGL recently published the largest change yet to its recommendations: Vulkan [1] is a ground-up redesign. Vulkan is a response to industry demands for a *lower-level* API than OpenGL [14], which hides too many performance knobs that software needs to tune. While OpenGL played a dual role as a hardware abstraction layer and a programming layer and arguably failed at both, Vulkan promises to abandon the pretense of being programmable: it is designed solely as a system abstraction. This shift has the potential to create an ecosystem of new, high-level programming tools that build on top of Vulkan and finally dislodge OpenGL's monopoly on graphics programming. The iron is hot, and programming languages research should strike.

Acknowledgments

Conversations with Yong He, Kayvon Fatahalian, and Tim Foley introduced me to real-time graphics programming and its infelicities. Their patient explanations pointed me in this direction. Todd Mytkowicz and Kathryn McKinley endured my early floundering with language design questions while I visited Microsoft Research. The anonymous SNAPL reviewers and Pat Hanrahan were exceptionally insightful with their suggestions for framing.

References

1 Khronos Vulkan registry. `https://www.khronos.org/registry/vulkan/`.

2 LambdaCube 3D. `http://lambdacube3d.com`.

3 Unity game engine. `https://unity3d.com`.

4 Unreal game engine. `https://www.unrealengine.com/`.

5 Apple. Metal shading language specification, version 1.2. `https://developer.apple.com/metal/metal-shading-language-specification.pdf`.

6 Chad Austin and Dirk Reiners. Renaissance: A functional shading language. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2005.

7 Baggers. Varjo: Lisp to GLSL language translator. `https://github.com/cbaggers/varjo`.

8 Daniel W. Barowy, Charlie Curtsinger, Emery D. Berger, and Andrew McGregor. AutoMan: A platform for integrating human-based and digital computation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012.

9 Tobias Bexelius. GPipe. `http://hackage.haskell.org/package/GPipe`.

10 Kovas Boguta. Gamma. `https://github.com/kovasb/gamma`.

11 Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability type inference for flexible approximate programming. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

12 William J. Bowman, Swaha Miller, Vincent St-Amour, and R. Kent Dybvig. Profile-guided meta-programming. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2015.

13 Michael Carbin, Sasa Misailovic, and Martin Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

14 Advanced Micro Devices. Mantle programming guide and API reference 1.0. `https://www.amd.com/Documents/Mantle-Programming-Guide-and-API-Reference.pdf`.

15 Alastair F. Donaldson. Crashes, hangs and crazy images by adding zero. Medium, November 2016. `https://medium.com/@afd_icl/689d15ce922b`.

16 Alastair F. Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In *Workshop on Metamorphic Testing (MET)*, 2016.

17 Conal Elliott. Programming graphics processors functionally. In *Haskell Workshop*, 2004.

18 Nicolas Feltman, Carlo Angiuli, Umut A. Acar, and Kayvon Fatahalian. Automatically splitting a two-stage lambda calculus. In *European Symposium on Programming (ESOP)*, 2016.

19 Matthew Flatt. Composable and compilable macros: You want it when? In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2002.

20 Tim Foley and Pat Hanrahan. Spark: Modular, composable shaders for graphics hardware. In *SIGGRAPH*, 2011.

21 Mark Guzdial. Trip report on Dagstuhl seminar on live coding. September 2013. `http://cacm.acm.org/blogs/blog-cacm/168153`.

22 Axel Habermaier. The model of computation of CUDA and its formal semantics. Master's thesis, Institut für Informatik, Universität Augsburg, 2011.

23 Shawn Hargreaves. Generating shaders from HLSL fragments. In *ShaderX3: Advanced Rendering with DirectX and OpenGL*. 2004.

24 Chris Hathhorn, Michela Becchi, William L. Harrison, and Adam M. Procter. Formal semantics of heterogeneous CUDA-C: a modular approach with applications. In *Conference on Systems Software Verification (SSV)*, 2012.

25 Yong He, Tim Foley, and Kayvon Fatahalian. A system for rapid exploration of shader optimization choices. In *SIGGRAPH*, 2016.

26 Yong He, Tim Foley, Natalya Tatarchuk, and Kayvon Fatahalian. A system for rapid, automatic shader level-of-detail. In *SIGGRAPH Asia*, 2015.

27 Görel Hedin and Eva Magnusson. JastAdd: An aspect-oriented compiler construction system. *Science of Computer Programming*, 47:37–58, 2003.

28 Khronos Group. glslang. `https://github.com/KhronosGroup/glslang`.

29 Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. GKLEE: Concolic verification and test generation for GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

30 Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. In *SIGGRAPH*, 2004.

31 Michael McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2002.

32 Sean McDirmid. Two lightweight DSLs for rich UI programming. `http://research.microsoft.com/pubs/191794/ldsl09.pdf`.

33 Microsoft. Direct3D. `https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466.aspx`.

34 Newzoo. Top 100 countries by 2015 game revenues, 2015. `https://newzoo.com/insights/articles/newzoos-top-100-countries-by-2015-game-revenues/`.

35 F. Nielson and R. H. Nielson. Automatic binding time analysis for a typed λ-calculus. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 1988.

36 Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction (CC)*, 2003.

37 Fabio Pellacini. User-configurable automatic shader simplification. In *SIGGRAPH*, 2005.

38 Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

39 Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013.

40 Adrian Sampson, James Bornholt, and Luis Ceze. Hardware–software co-design: Not just a cliché. In *Summit on Advances in Programming Languages (SNAPL)*, 2015.

41 Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011.

42 Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.

43 Carlos Scheidegger. Lux: the DSEL for WebGL graphics. http://cscheid.github.io/lux/.

44 Mark Segal and Kurt Akeley. The OpenGL 4.5 graphics system: A specification. https://www.opengl.org/registry/doc/glspec45.core.pdf.

45 Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. In *SIGGRAPH Asia*, 2011.

46 Chris Stucchio. Type-safe vector addition with dependent types, December 2014. https://www.chrisstucchio.com/blog/2014/type_safe_vector_addition_with_dependent_types.html.

47 Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1997.

48 Rui Wang, Xianjin Yang, Yazhen Yuan, Wei Chen, Kavita Bala, and Hujun Bao. Automatic shader simplification using surface signal approximation. *ACM Transactions on Graphics*, 33(6), November 2014.

49 Steven Wittens. ShaderGraph: Functional GLSL linker. https://github.com/unconed/shadergraph.