TINSL:
Tinsl Is Not a Shading Language

by

Cole Granof

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Interactive Media and Game Development

May 2021

APPROVED:

Professor Charlie Roberts, Major Thesis Advisor

Professor Gillian Smith, Thesis Reader

Jennifer deWinter, Head of Department

Abstract

The fragment shader is a powerful, efficient tool for image processing. However, shaders are often unwieldy to use, even for simple post-processing operations. We have developed multiple systems in order to simplify the use of fragment shaders for image processing, and applied these systems across different domains, including live coding, creative coding and game development. Through this iterative process, it became clear that a custom domain-specific language (DSL) could simplify the authorship of a non-trivial rendering pipeline that involves multiple fragment shaders. As a result, we developed Tinsl (Tinsl Is Not a Shading Language) which is a DSL for specifying multipass render-to-texture (RTT) post-processing effects. This language uses the familiar syntax of GLSL, but adds semantics for specifying how to utilize temporary textures. This "shader metaprogramming system" enables live coders, creative coders and game developers to create complex post-processing effects that involve feedback and multipass blurs in a single file. We evaluate the effectiveness of this system using Petre and Greene's cognitive dimensions framework to provide a qualitative analysis. We also perform a user study that gathers feedback from participants who were asked to follow a short tutorial on the language in a web-based creative coding environment. We released the software as a set of open-source libraries and programming environments.

Contents

1 Introduction

Despite graphics programming being an endeavor often fraught with technical hurdles, fragment shaders are, by contrast, surprisingly elegant. A fragment shader is a small program that runs once on every pixel to produce a final output. The GPU (graphics processing unit) in a computer specializes in parallelizing these kinds of operations. One area where the elegance of fragment shaders shine is in live coding. Live coding environments offer a playground-like user interface to rapidly see the results of editing programs, often in real time; the simplicity they offer enables these environments to be used in audio-visual performances. Many graphical live coding environments focus on fragment shaders, abstracting away the tedium of dealing directly with graphics API (application programming interface). However, single fragment shaders operating with a single texture can only do so much. Many post-processing effects can only be implemented (in a reasonably efficient way) by using temporary rendering targets and running multiple passes.

This form of control flow is outside the scope of the fragment shader. We hypothesize that a DSL could allow programmers to write logic for render-to-texture (RTT) steps directly alongside shader code. In this paper, we discuss the language we created, Tinsl (Tinsl Is Not a Shading Language) which has the succinctness and familiarity of a shading language like GLSL (OpenGL shading language). Unlike GLSL, Tinsl also lets the user specify intermediate rendering to off-screen textures, and sample from those textures later in the effect. The compiler divides (and combines, where possible) the operations into multiple fragment shaders. The compiled output of this language is a tree with leaves containing fragment shader source code. We have also created a live coding environment that focuses on writing post-processing effects on webcam video with Tinsl code. This has the dual purpose of testing the core design of Tinsl, as well as offering live coders a unique tool to enhance performances with easily-authorable multipass rendering techniques. We will also discuss merge-pass, which is a post-processing effect library that offers an API in TypeScript, and highlight the domains where it has been applied. Tinsl is not written on top of merge-pass; it is a completely separate library.

We evaluated Tinsl using Green and Petre's "cognitive dimensions framework" to perform a qualitative analysis. We also performed a user study to gather feedback on the design of the language. Finally, we perform a quantitative analysis based on lines of code needed to implement a "bloom" effect on an image with Tinsl and without.

2 Background

In this section, we offer a brief history and explanation of shader metaprogramming techniques, and where these techniques fit into the current landscape of graphical apps and games in the web browser. We will also go over existing theory about live coding systems with examples.

2.1 Shader Metaprogramming

In the paper "Abstract Shade Trees", the authors note that GLSL lacks modern language features for encapsulation and abstraction, making it "hard to create and maintain long GPU programs" [14, p. 1]. While this remark was published in 2006, there are many domains where this is still the case. This problem is especially acute when we consider the web world, where we are limited to earlier versions of GLSL.

"Shader metaprogramming systems" are one solution to these problems and surprisingly common in the world of game engines and other high-performance graphical applications intended to run on native hardware. Metaprogramming techniques range from simple textual replacement, DSLs embedded in higher-level languages, and even forks of existing shader compilers. "Sh" is perhaps one of the earliest examples of bringing high-level programming features to shader development. It is a C++ API that provides a metaprogramming language for shaders with operator overloading and macros [13]. Such an approach would not be feasible for a language that does not provide such features natively, like Java or JavaScript. (Sh eventually developed into the RapidMind Multi-core development platform at Intel; both systems are no longer maintained [20].) Other embedded DSLs for shader metaprogramming exist in Haskell and Python [6][12].

Slang is a relatively recent example of shader metaprogramming. Slang extends HLSL with modern language features that help with managing complexity, such as generics and interfaces [10]. Instead of being implemented as an embedded DSL leveraging the features of the host language, the team behind Slang accomplished this by developing a custom compiler.

Robust shader metaprogramming techniques are not as prevalent in web programming; the solutions that have been engineered have suffered from waning or non-existent maintenance in recent years. When WebGL was a relatively new standard for browsers in 2012, David William Wallace made the observation that "WebGLSL lacks many high-level language features such as user-defined namespaces and modules" [22], which mirrors the comments made by by McCool et. al. in 2002, who note "custom shading languages usually will not be as powerful as full programming languages. They often may be missing important features such as such as modularity and typing constructs useful for organizing complex multipart shaders" [13]. In an effort to make building large-scale 3D applications for the web more manageable, Ashima Arts developed *gloc*, which is briefly described in the paper "*gloc*: Metaprogramming WebGL Shaders with OCaml". It would appear that this library is no longer maintained as the latest commit is from May 2012.[1] Another library, glslify, attempts to bring shader modularity to the web, allowing the user to make use of many prewritten shader functions. This open-source project is no doubt still used by many, however contributions to this project have slowed down considerably. The latest official release is from 2016, and glslify still lacks proper WebGL 2 and GLSL 3.00 support.[2]

[1]To the best of our knowledge, the most recent version of *gloc* is only available at this link: `https://github.com/dsheets/gloc`

[2]As of April 2021, this 2015 issue on the GitHub page of glslify is still open, which can be

String concatenation and textual replacement is certainly a form of metaprogramming, in the sense that it is code that manipulates other code as data. In general, this is what we see in PlayCanvas[3], Babylon[4] and Threejs,[5] which are three popular open-source platforms for 3D programming. These frameworks/engines treat the shader code as a "black box". The shaders live as a string inside the code of the host language; there is little to no semantic understanding this string of embedded shader code before it is compiled. Such techniques lack the robust features and early error detection.

One possibility for this disparity is that lightweight metaprogramming solutions are sufficient for the common use cases of web-based game/rendering engines. By contrast, in the AAA games industry, advanced shader pipelines are developed due to the need to coordinate a team of graphics programmers and artists to produce a massive game with high graphical fidelity. TFX, the shader pipeline for the game Destiny, is a prime example of this [27]. A "TFX for the WebGL" would be over-engineered for the current landscape of web games, as no web game has come close to the level of production seen in Destiny. We can speculate about why this is the case, but determining a concrete reason for this is out of scope for this thesis. At the same time, we believe there is some middle ground between the complex shader metaprogramming of the AAA games industry and the simplicity of unembellished string concatenation that could serve to elevate the ease of use and maintainability of shaders, namely for post-processing effects, in the web. While a system like Tinsl might serve this niche in the video game industry, we have instead created a minimalistic live coding environment for creating post-processing effects that includes syntax and error highlighting.

2.2 Live Coding

Live coding systems enable programmers to see the results of changes to code very rapidly, often in real time. In the paper "VIVA: A Visual Language for Image Processing", Steven Tanimoto describes four levels of "liveness", a term which refers to the extent to which the programmer receives live feedback from the system. The third level of liveness is described as "edit-triggered updates", an environment where the user only needs to make a change to the model in order to see the result. No "run button" is required. The fourth level of liveness describes a system that is updating its output based on some time-varying stream even when the user is not directly editing the model. In the paper, author presents this as a layered taxonomy, even going so far as to create a diagram where each level encompasses the last [26, p. 129]. However, now that we have many of these reactive systems, we see that a system can exist between levels. While The_Force [25, p. 2]tries to compile your code as you type, forgoing any kind of "run button", many other systems require the user to select blocks

seen here: `https://github.com/mrdoob/three.js/`

[3]`https://github.com/playcanvas/engine`

[4]`https://github.com/BabylonJS/Babylon.js`

[5]`https://github.com/mrdoob/three.js`

of code. This has the advantage of giving the user more control (at the cost of simplicity), preventing the computer from executing on what is essentially an "incomplete thought". By this description of "liveness", such a system would sit below level 3. At the same time, these environments also present a time-varying visualization, a characteristic of level 4 liveness systems. (The authors of this paper implement the live visual programming system two years later. They observe that their implementation is at "level 3" liveness. Edits trigger a change in the output, but the image does not change based on time [2].)

In the paper "Live Coding Ray Marchers with Marching.js", Charlie Roberts characterizes "two camps" of live coding environments. [16] The first camp includes a host of live coding environments that focus on the "traditional" graphics pipeline, which involves rasterizing meshes built of triangular faces. The second camp includes environments that take full advantage of the fact that modern graphics pipelines are very programmable, leaning heavily on fragment shaders to drive the visuals.[6] VEDA is a live coding environment that is an easy to use GLSL runtime [1]. VEDA provides very few abstractions over raw GLSL. In this way, it is similar to The_Force, which is browser-based instead of an plugin for the text editor Atom [25, p. 1]. Hydra, by contrast, is a live coding environment that provides many abstractions over GLSL in order to mirror analog synthesizers (although it is still possible to drop down to the level of GLSL if desired) [11]. These synthesizer-inspired abstractions get compiled into fragment shaders. Marching.js is similar to Hydra but with a focus on constructive solid geometry [16, p. 1]. The declarative syntax of Marching.js gets compiled into a single fragment shader using ray marching techniques, abstracting away the linear algebra typically required to write a fragment shader that performs volumetric rendering.

2.3 Multipass Effects in Different Environments

Multipass effects are not possible to specify within a single fragment shader. As such, different environments offer unique ways to author multipass effects.

2.3.1 Unity

Unity uses the language HLSL for its shaders. In previous versions, shaders were written in Cg. Because HLSL was created for use in DirectX, Unity uses an open-source program called HLSLcc to cross-compile HLSL bytecode to GLSL or Metal when targeting non-Windows platforms [21]. It is also possible to insert snippets of GLSL directly into the fragment shader code [8].

In order to perform multipass post-processing effects that require a texture for intermediate calculations, a user can call `RenderTexture.GetTemporary`. Unity keeps a pool of these temporary textures for this purpose. Calling the function `RenderTexture.ReleaseTemporary` returns this texture to the pool, allowing the resource to be reused [15]. In Ronja Böhringer's Unity tutorial series, she makes use of this function to implement a Gaussian blur effect [3].

[6]Our system, Tinsl, falls into this category.

2.3.2 TouchDesigner

Touch Designer enables users to perform post-processing effects through a visual node-based language. Texture Operators, or TOPs, allow the user to perform real-time image processing on the GPU. Touch Designer provides a Feedback TOP for this purpose [7].

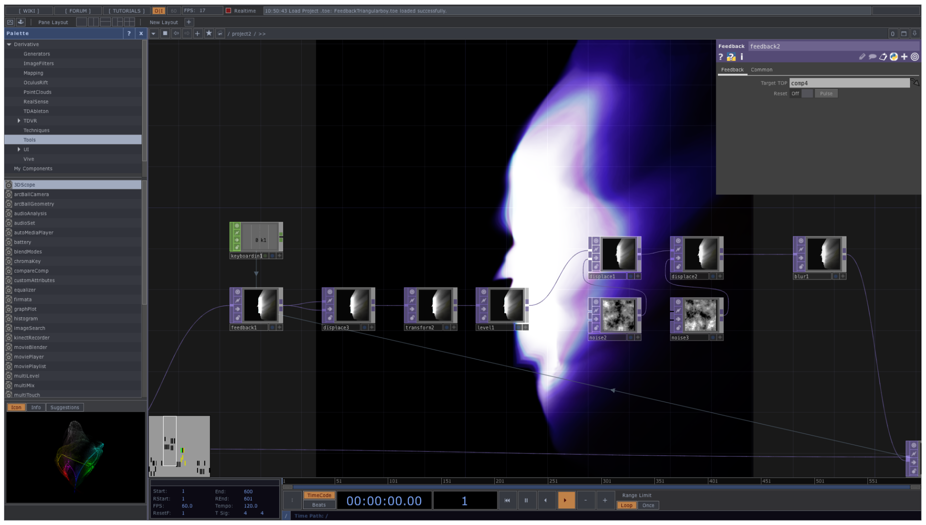


Figure 1: Section of a freely available Touch Designer project available at the following link: `https://www.youtube.com/watch?v=X8MPdeMM__U`

2.3.3 Shadertoy

The user has access to multiple channels of textures, which can be sampled from by passing in `iChannel0` (up to `iChannel3`) to a call to `texture`. A channel can also be set up as a buffer instead of a texture. Using the "+" button to create a new tab, one of the options revealed includes `BufferA` up to `BufferD`. The user can sample from the same buffer that is being written to. Reading from and writing to the same render target is prohibited in WebGL, so one can only assume that Shadertoy provides some abstraction over this.

One usability issue is with running multiple passes of the same shader. In Shadertoy, this can only be done by duplicating the same code in the `BufferA` editor into the `BufferB` editor, but making the code for `BufferB` sample from `BufferA`'s channel. This can be chained all the way to `BufferD`.

2.4 Implementing Bloom in OpenGL with GLSL

Many post-processing effects require intermediary rendering targets and multiple passes. Consider how we might implement a bloom post-processing effect, which creates a glowing haze around all bright objects in a scene. This is a widely used example for image processing [12, p. 7]. A summary for this procedure is described in the Learn OpenGL blog written by Joey de Vries [28]. Walking through this process will highlight some of the "tedious" aspects of creating a multipass effect that requires rendering to a temporary off-screen texture. The purpose of this section is to demonstrate how the implementation of multipass texture-to-texture effects lack the satisfying elegance of singular shaders, which are, by contrast, one program that runs once once on each fragment, fully described in one file. While the management of resources is by no means difficult for an experienced graphics programmer, this process can be frustrating and error-prone to those who are less experienced.

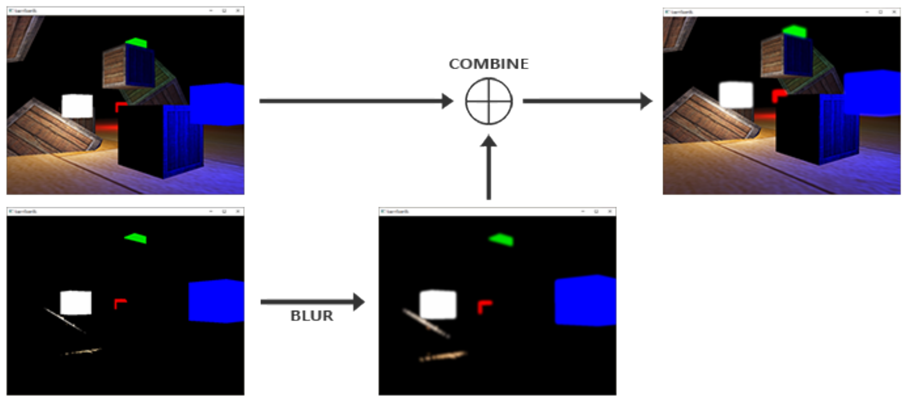


Figure 2: Steps for performing a bloom effect. Image by Joey de Vries, from the following article: `https://learnopengl.com/Advanced-Lighting/Bloom` licensed under Creative Commons version 4: `https://creativecommons.org/licenses/by-nc/4.0/` Joey de Vries on Twitter: `https://twitter.com/JoeyDeVriez`

To begin our bloom effect, the scene needs to be copied over to a separate texture with all fragments that fall below a certain brightness masked off. This requires an intermediary render target. We must bind an intermediary texture to the current framebuffer before executing this first program. This is a relatively simple program which only serves to "drop" all bright fragments "on the floor".

Second, the masked version of the original scene needs to be blurred. While there are many ways to blur an image, one of the most efficient ways is to perform a Gaussian blur in multiple passes, sampling from only a small kernel of colinear points and repeating the process in an orthogonal direction to the first pass. Since one cannot render to the same texture from which you are sampling from, this requires that we "ping-pong" between two textures to get the final effect. Additionally, since we need the first pass to blur the image horizontally, and the second pass to blur vertically, this requires writing and maintaining two separate but very similar shaders. Again, we are burdened with managing more resources and attaching the right texture to the current framebuffer and changing the sampler uniform to point to the previous texture. Also, how many times do we want to perform this feedback-based blur effect? More passes will result in a stronger blur. This detail is written into the implementation of the texture-swapping loop, not the GLSL shader code, further fragmenting our effect into multiple locations in source code.

At this point, we have a texture that contains a blurred version of all the bright objects in the scene set on a black background. Now, we must add this new image back on top of the original image. This requires yet another shader with two samplers, one for the texture that contains the original scene and another texture for the blurred masked version of the scene. We also need to tell the framebuffer to render to the screen since this is the final step.

Some steps, however, do not require an entirely new shader pass. For example, if we wanted to change the contrast of the final image, we could do so by modifying the final shader that combines the original image with the blurred image, since increasing the contrast does not require sampling from adjacent fragments. We only need to perform a single operation on the final color before assigning the current fragment color.

Another complication arises from splitting all of this logic across many shaders. Typically a given uniform can only be set when a program is in "use". In OpenGL with C/C++, this is following the call `glUseProgram`. In JavaScript, this is `gl.useProgram`. In most cases, you cannot have a single uniform across multiple shaders, which is inconvenient if steps of the process need access to the same information.[7] Suppose you wanted to increase the size of the Gaussian kernel by updating a float uniform. You would need to do this across two different shaders, since the Gaussian blur is split into a horizontal blur shader and a vertical blur shader. These are two uniforms with two separate locations in memory that can only be updated at two separate points in the program, only when that shader is "in use".

The library merge-pass partially solves these issues. It will generate a tree of shader programs that contains information about how many times to repeat each program, which texture to render to and how many texture will be needed. Then, it compiles the programs based on the source code in the tree, generates the required resources, and executes these programs in the correct order. It also provides a simple way to update uniforms on the GPU regardless of how the final post-processing effect might be split up, potentially resulting in a uniform being repeated across multiple shaders.

3 merge-pass

We created a post-processing library in order to bring post-processing effects to Charlie Robert's Marching.js playground. This library, titled merge-pass,

[7]In later versions of OpenGL, there exists the `shared` keyword that exists for this purpose, but it is not available in all versions of OpenGL. Most importantly, it is not available in OpenGL ES, the subset of OpenGL used for WebGL.

simplifies the creation and implementation of multipass post-processing effects. In Section 2.3, we discuss how multipass effects are authored in different environments, which will illuminate why we decided to build a unified approach to generating code for multipass post-processing effects for our own purposes.

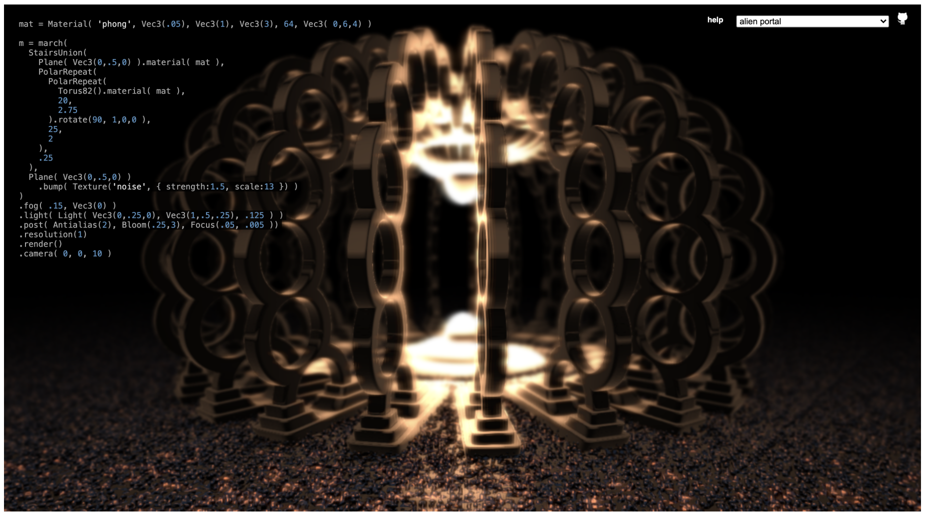
3.1 Applications

merge-pass has been applied (by the author) in other projects, including a p5 library, called post5, and a generative art toy called artmaker.

3.1.1 Marching.js

Marching.js is a live coding environment created by Charlie Roberts. This environment focuses on volumetric rendering using ray marching.

In the paper “Live Coding Ray Marchers with Marching.js”, Charlie Roberts concludes by noting that the output of the ray marched scenes could be described, as “cold”, “sterile”, “technical”, or “clinical”, and that post-processing effects could be applied to address this [16, p. 6]. In the summer of 2020, merge-pass was integrated into Marching.js in order to provide optional post-processing effects. Some of the effects include hue rotation, antialiasing, bloom, depth of field and edge detection. The user can chain however many of these effects, and manipulate properties of these effects in realtime.



```
mat = Material( 'phong', Vec3(.05), Vec3(1), Vec3(3), 64, Vec3( 0,6,4) )

m = march(
  StairsUnion(
    Plane( Vec3(0,.5,0) ).material( mat ),
    PolarRepeat(
      PolarRepeat(
        Torus82().material( mat ),
        20,
        2.75
      ).rotate(90, 1,0,0 ),
      25,
      2
    ),
    .25
  ),
  Plane( Vec3(0,.5,0) )
    .bump( Texture('noise', { strength:1.5, scale:13 }) )
)
.fog( .15, Vec3(0) )
.light( Light( Vec3(0,.25,0), Vec3(1,.5,.25), .125 ) )
.post( Antialias(2), Bloom(.25,3), Focus(.05, .005 ))
.resolution(1)
.render()
.camera( 0, 0, 10 )
```

Figure 3: Antialiasing, bloom and depth of field effects running in Charlie Robert’s Marching.js using merge-pass.

Some of these effects require a depth texture, namely “god rays” (a sunbeam effect) and depth of field. Because ray marching differs from the traditional graphics pipeline, where polygonal meshes are rasterized, the depth of the scene was not immediately accessible. Therefore, the developer of Marching.js added an additional code in Marching.js to render the depth of each fragment to a floating point texture which is then consumed by merge-pass.

The user can use these post-processing effects to add realism with to the scene, or take the aesthetic in an abstract direction. Charlie Roberts, creator of Marching.js, made a fractal look like undulating underwater coral by using blue-tinted sunbeams and by bringing the foreground out of focus, as seen in Figure 4. This was accomplished with the `Focus` and `Godrays` effects.

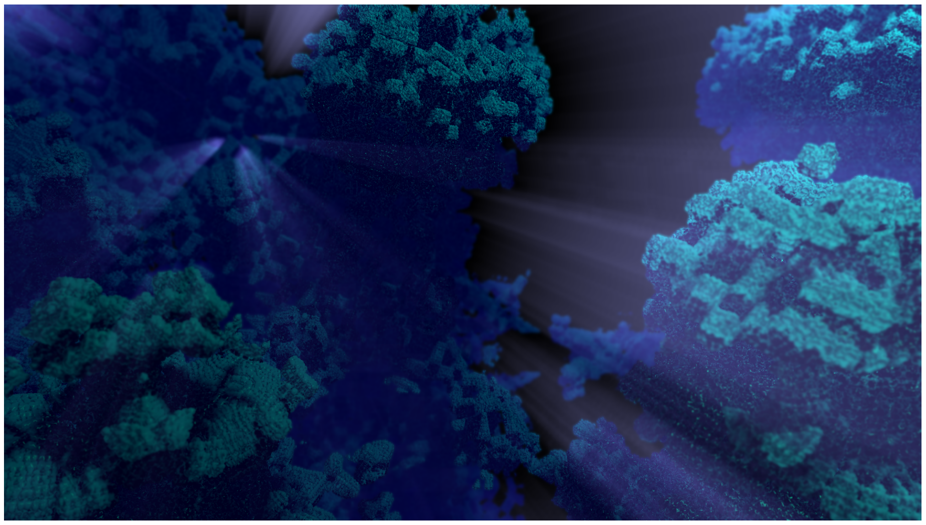


Figure 4: Underwater scene created with ray marching and post-processing with merge-pass.

While the original intention of including post-processing effects in Marching.js was to add "cinematic" effects, these effects also allow for non-photorealistic rendering. In contrast to the underwater fractal scene, teacher and creative coder Daniel Opreza used a long chain of effects to transform a largely black-and-white scene into a colorful neon matrix, as seen in Figure 5.

3.1.2 post5

In order to make merge-pass more easily accessible for creative coding, we created post5, which is a thin wrapper for merge-pass and postpre, a set of merge-pass presets. This allows the the user to add a variety of effects to simple p5 sketches with very little effort or lines of code.

As discussed in Section 3.3, merge-pass encodes GLSL's type system into TypeScript's type system so that it is theoretically not possible to create an effect that has type errors when compiled to GLSL fragment shaders. However, p5 is primarily used in JavaScript. Therefore, it is possible to pass in incorrect values to merge-pass expressions to produce erroneous GLSL code. In practice,

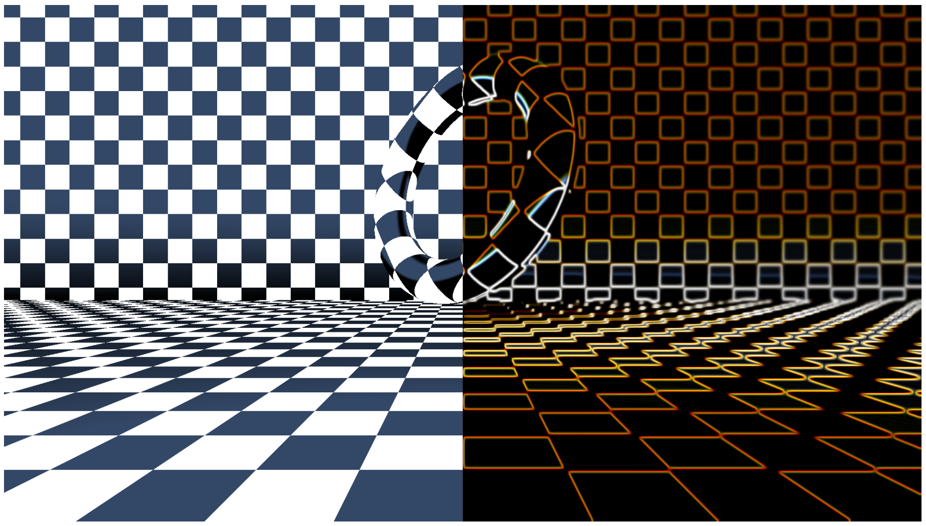


Figure 5: An abstract use of post-processing effects. On the left is the original image, and on the right is the image after post-processing.

this mostly amounts to a usability issue; the user will have to debug type errors without the help of the TypeScript compiler.

We have also created six examples, available in the online p5 web editor, to get users of all skill levels started with creative coding using post5. The code for some of these examples is included in the Appendix. The p5 code for the sketch depicted in Figure 6 is only 20 source lines of code, and easily fits into a reasonably sized editor window.[8]

3.1.3 artmaker

artmaker is a generative art web toy with a minimalist UI. Upon hitting 'R' or loading the page for the first time, the program generates a simple abstract 2D animation with randomized parameters, such as color, speed, size, and shape. Some of the animations were adapted from creative coding pieces by the author; others were created anew. Various post-processing effects are chosen and layered on top of the image in a random order. The randomly chosen effects range in complexity from simple color grading to overlaid dancing beams of light with colors sourced from the original image. The order in which these effects are applied to the simple, abstract source image have surprising results that are enjoyable to contemplate. The user can save a still image of the generated art, or copy the link and send it to another person so they can view it in realtime in the browser.

[8]All interactive sketches can be viewed in the browser at `https://editor.p5js.org/bandaloo/sketches`. Desktop Chrome/Firefox is required.

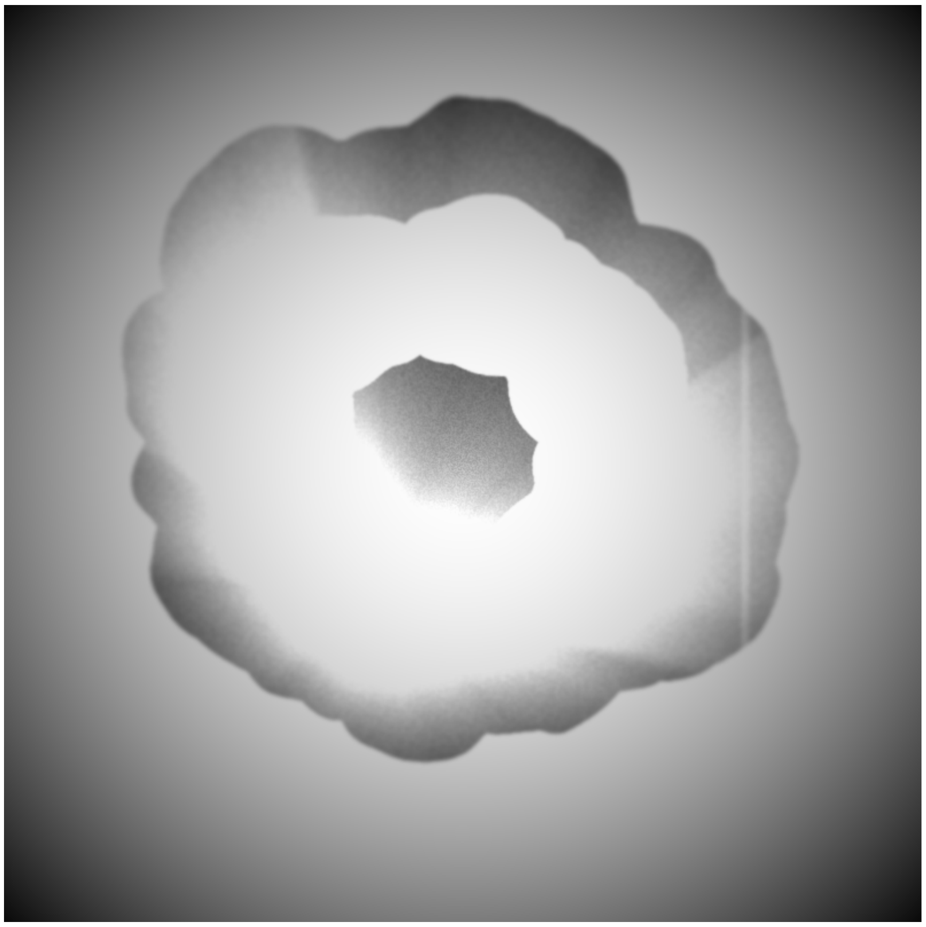


Figure 6: "Inky sunbeams" interactive example using post5, available at `https://editor.p5js.org/bandaloo/sketches/zXMOZkuYU`. The user can create blots of black or white using the mouse. Compare this to the unprocessed version in the Appendix, Figure 12.

Much of the work in creating artmaker went into fine-tuning the dozens of randomized parameters. For example, some post-processing effects are randomly selected far less frequently than others. In addition, some effects can be selected by the algorithm multiple times, while others are stricken from the list of available choices once they have been already chosen. This continual tweaking of probabilities was performed based on the author's own aesthetic preferences. It is interesting to think about how the way in which generative aspects of the system are tuned is a form of self-expression, despite each individual piece being machine generated.

artmaker is intended to be a very passive, offering very little control over its output. Instead of using a set of tools to craft an abstract piece as you would in a more pragmatic creative tool, the user takes a back seat to the underlying generative algorithms, exploring the expressive range of the generator by repeatedly generating new animations. However, the user can override the generated color palette through the use of color selectors in the top left corner of the screen.

Researchers Kate Compton and Michael Mateas coined the term "Casual Creators" in their paper by the same name. "Casual Creators" are "creativity tools" which "privilege the enjoyable experience of explorative creativity over task-completion" [5, p. 228]. The authors define the term: "A Casual Creator is an interactive system that encourages the fast, confident, and pleasurable exploration of a possibility space, resulting in the creation of discovery of surprising new artifacts that bring feelings of pride, ownership, and creativity to the users that make them," even offering the Spirograph art toy as an analog example of a Casual Creator [5, p. 229]. The question arises: is artmaker a Casual Creator? Is its constrained design what makes it a Casual Creator, or are the limitations it places on authorial control so extreme that it prevents users from experiencing the "pride" and "ownership" they would otherwise feel if they had more control over the output? We will not attempt to answer these questions definitively in this paper. However, exploring how constrained a creative tool can become before it prohibits users from experiencing a sense of ownership is an intriguing potential for new research.

artmaker can also be used outside of the web toy in order to add abstract animations to any web application. It is available as a single minified script that can be included in a `<script>` tag without any additional build steps. Alternatively, it is also available as an npm package. The purpose of this is to encourage users with JavaScript and HTML5 knowledge to take the randomly generated animations discovered through the web toy and use them in new domains.

We used the npm version of artmaker to add moody, abstract backgrounds to a text-based prototype for a future game later developed in Unity. Although this incarnation of the game was a prototype, it stands on its own as it offers a unique aesthetic and gameplay experience, different from the final Unity version. The act of curating animated backgrounds for the game involved repeatedly hitting the "regenerate" key and copying down the "seeds" (short strings used to recreate the same animation) with an accompanying note of where this background could be used in the overall narrative. Despite the lack of authorial control, the rapidity at which we could explore generative animated backgrounds allowed us to quickly find enough animations that supported the ambiance of various parts of the narrative. Surprisingly, this generator originally conceived as a web toy offered a productive (and enjoyable) mode of exploring an aesthetic domain.

merge-pass and the collection of post-processing presets, postpre, greatly simplified the implementation of artmaker. Together, postpre and merge-pass consist of 6269 source lines of code. artmaker itself is merely 942 source lines of code. This data was calculated was calculated with the npm package sloc version 0.2.1. Versions of merge-pass, postpre and artmaker in question are versions 0.6.4, 0.1.5 and 0.4.0 respectively.[9] Minified, `artmaker.min.js` is 96 kilobytes. Beyond merge-pass and postpre, artmaker also requires the npm package seedrandom, which contributes to the size of the bundled version of artmaker.

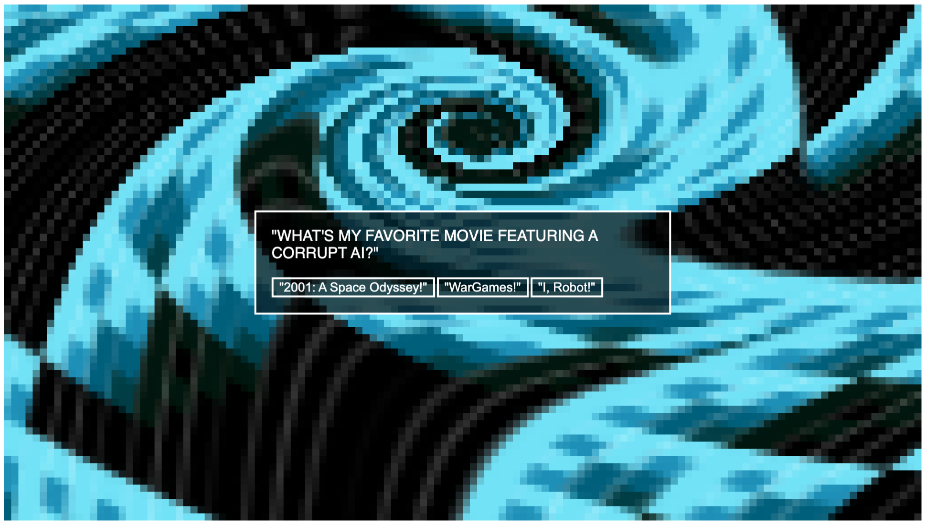


Figure 7: The player is being quizzed by a malicious AI in the text-based prototype for "Forward". The background is in motion, changing color and shape.

3.2 Regions

Using `region`, it is possible to constrain a post-processing effect to a specific area of the screen. In a single shader, this is trivial, however, recall that merge-pass may use multiple shader passes for image processing. `region` is useful for split-screen applications, as different effects can be applied to different parts of the screen in what appears to the user like a single post-processing step. For effects that sample from neighboring pixels, such as blurs, sampled positions are clamped to that particular area of the screen.[10] This prevents the colors of adjacent regions from "bleeding" into the current region. Regions can also contain nested regions, which will be obscured by the boundaries of outer regions, as seen in Figure 8.

[9]The program sloc allows us to exclude comments and empty lines in our count. Even still, these measurement should be regarded as an estimate.

[10]This works for rectangular regions only. merge-pass also supports masking a non-rectangular region based on an image.

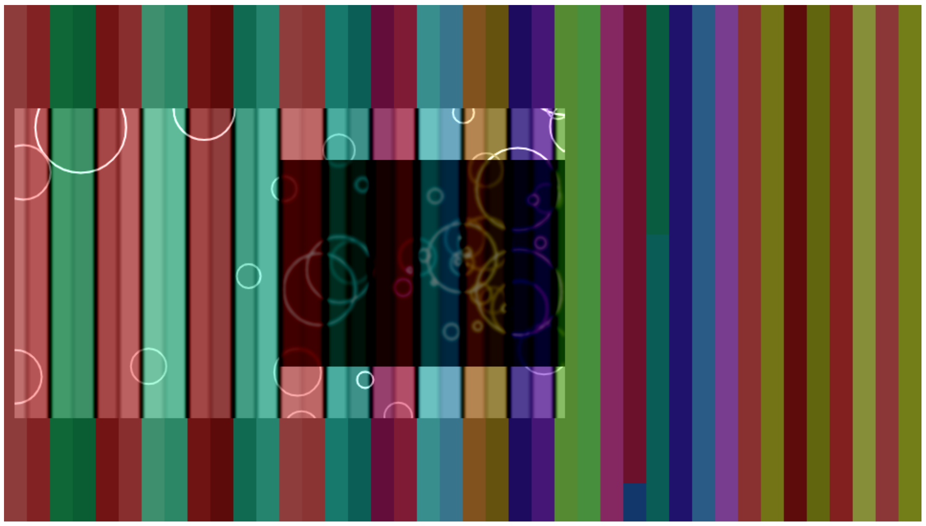


Figure 8: Nested regions each with their own arbitrary post-processing effect

3.3 Limitations

Because JavaScript and TypeScript do not have useful metaprogramming features such as macros or operator overloading without additional tooling, using merge-pass is verbose compared to shading languages such as GLSL.[11] While merge-pass provides many simple functions to perform complex operations such as simplex noise or bloom, basic math expressions can become particularly cumbersome to read and write. Consider the GLSL expression:

```
cos(fragColor.y * 270. * 3.14159)
```

In merge-pass, this becomes:

```
a1(
  "cos",
  op(getcomp(pos(), "y"), "*", 270 * Math.PI)
)
```

There are many builtins in GLSL that take one argument and return a value of the same type. To use one of these so-called “arity 1” functions, the calls the function `a1` and pass in the function name as a string. The TypeScript compiler will not compile if the user passes in a function that does not exist. However, incorrect components passed into `getcomp` can only be caught at runtime; even TypeScript’s type system is not expressive enough to detect all cases at compile time.

[11]In one sense, JavaScript is a very powerful metaprogramming language in that it allows for interpreting arbitrary strings as JavaScript code. What we mean to highlight is that it lacks the features to “metaprogram” in a structured way.

Using merge-pass feels like building up an abstract syntax tree by hand. In certain cases, such as the metaprogramming done in artmaker (Section 3.1.3) where these trees are built up procedurally, this is useful and even elegant.[12] With merge-pass, the user can create metaprogramming helper functions in the host language (TypeScript) to generate code in a type-safe way. Consider what would happen if a user of merge-pass attempts to write the code:

```
op(someVec2, "*", someVec3)
```

The TypeScript compiler will alert the user to this; there is no runtime error. Naturally, this extends to metaprogramming helper functions as well; a function that takes in merge-pass expressions as arguments to return another merge-pass expression must naturally return a merge-pass expression with no type errors if the TypeScript compiler type checks the program successfully. While TypeScript has a very expressive type system (which becomes even more flexible with every release) the series of union types, function overloads and generics used to encode GLSL's type system into TypeScript became complex. This is the reason merge-pass only deals with floating point vectors, matrices and scalars, and does not include signed and unsigned integers, booleans or array types.[13]

4 Tinsl

Like merge-pass, Tinsl aims to simplify the use of WebGL for non-trivial post-processing effects by abstracting away the details of managing multiple intermediate rendering targets. Tinsl, however, is its own language with syntax familiar to those who have interacted with fragment shaders before, even briefly.

Tinsl is designed to be similar to GLSL ES 3.00, the version of GLSL that WebGL 2 supports, with additional features and restrictions. The main benefit of this is that this allows users to take advantage of existing GLSL code. There are many highly-optimized versions of common functions used for procedural graphics that could simply be dropped in (if the licensing allows for this, which is often the case.) For example, the following function, written in GLSL, can be pulled into Tinsl without modification:

```
// https://github.com/hughsk/glsl-hsv2rgb/blob/master/index.glsl
// by Hugh Kennedy, MIT licensed
vec3 hsv2rgb(vec3 c) {
  vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
  vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
  return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
```

[12]This is also used within merge-pass itself; the `fractalize` function can take a noise-like function and call it with successively smaller octaves and amplitudes.

[13]Tinsl, by contrast, has a hand-written type checker to support all of the edge cases specified by the GLSL ES 3.00 language specification. It supports the types that merge-pass does not.

Designing Tinsl to be broadly compatible with GLSL was an important goal for us. Shadertoy offers a wealth of knowledge to those willing to dig through and understand the code. Similarly, the stack.gl ecosystem brings modularity to the creation of shaders by hosting packages that users can import directly in GLSL with a special pragma.

GLSL is a C-like language with certain restrictions. Common features of general-purpose programming languages are absent, such as pointers, recursion and multi-dimensional arrays. Type casting and the `sizeof` operator are also not supported [23]. These restrictions simplify (and in some instances complicate) the inclusion of additional features through source-to-source compilation. The parser for Tinsl is implemented with nearley, which is a parsing toolkit written in JavaScript [4].

4.1 Language Features

The features Tinsl to adds to GLSL with source-to-source compilation include: static type inference, named arguments, default arguments, and extra options for immutability. Most notably, Tinsl allows the user to specify a rendering pipeline, using "render blocks", alongside this GLSL-style code.

4.1.1 Static Type Inference

Tinsl can infer function return types and variable declaration types with the `:=` operator and `fn` keyword, as seen in the following function definition:

```
fn tinsl_rotate2d(vec2 v, float angle) {
  // declaration with := operator infers type
  m := mat2(cos(angle), -sin(angle), sin(angle), cos(angle));
  return m * v; // tinsl compiler knows that a mat2 * vec2 -> vec2
}
```

For familiarity and compatibility with existing GLSL, Tinsl also supports explicit return types and variable declaration types:

```
vec2 glsl_style_rotate2d(vec2 v, float angle) {
  mat2 m = mat2(cos(angle), -sin(angle), sin(angle), cos(angle));
  return m * v;
}
```

There are a few key differences between variable declaration in Tinsl compared to GLSL. All variable declarations must have an initial value. This is the case with both styles of declaration. It is also not possible to declare multiple variables in the same statement with a comma. However, the terseness of the `:=` operator makes it so that multiple related declarations can fit easily onto one like regardless. Precision qualifiers are also not available in Tinsl. A default precision is used across the entire program.

4.1.2 Immutability

The concept of `const` in GLSL differs from other languages. A `const` variable in GLSL must be a compile time constant; it cannot be the result of a user defined function, function argument, or even a uniform value. Tinsl introduces another keyword for immutability.

Variables declared with `:=` are "final" by default. This means they are immutable. This extends to all components of a vector and elements of an array. If the user wishes to declare a mutable variable with `:=`, Tinsl requires the user to be explicit:

```
fn foo() {
  mut bar := 1;
  bar += 2;
  return bar;
}
```

In the original GLSL style of variable declaration, the user may declare a variable as final with the keyword:

```
fn baz(int a, int b) {
  final int result = a + b;
  return result;
}
```

In line 2 of the above code example, we would not be able to use GLSL's `const` keyword because the expression `a + b` is not a compile-time constant; there is no construct in GLSL ES 3.00 to ensure immutability for non-compile-time variables. This is where `final` is useful. Again, this is the default behavior for values declared with the `:=` operator. Parameters are also immutable; if the user wishes to perform mutation on a parameter they must explicitly assign it to a mutable variable. The inclusion of `final` is a small feature, however, immutability by default is a popular design choice in modern languages. Kotlin, Rust and Swift have features that encourage users to embrace immutability first.

4.1.3 Default and Named Arguments

Unlike GLSL, Tinsl allows for default arguments with the following syntax:

```
fn godrays (
  vec4 col = frag,
  float exposure = 1.,
  float decay = 1.,
  float density = 1.,
  float weight = 0.01,
  vec2 light_pos = vec2(.5, .5),
  int num_samples = 100,
  int channel = -1
) {
  // function body omitted
}
```

Default arguments must be trailing. User-defined functions can also be called with named arguments. Combining named arguments and default arguments is particularly useful for functions that have many parameters. For example, if we only wanted to pass in an argument for `num_samples` and use the default values for all others, our function call could look like this:

```
godrays(num_samples: 50)
```

Although a module system for this version of Tinsl does not exist yet, a library could provide many robust and customizable functions that perform complex post-processing effects. The motivation for including default arguments and named arguments is to simplify the invocation of these types of functions.

The inclusion of optional arguments makes it non-obvious how best to handle cases where a function overloads collide due to the fact that not all arguments are required. In the current version of Tinsl, overloaded functions are not allowed.

4.1.4 No Preprocessor

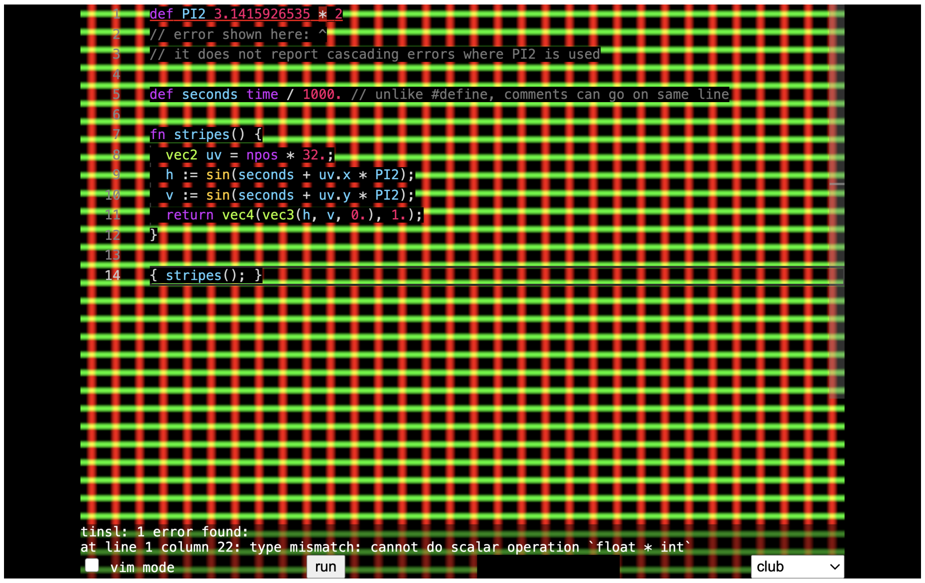
Tinsl does not have preprocessor pragma such as `#define` and `#ifdef`. Instead, Tinsl has the keyword `def` that serves the purpose of `#define` but does so without raw textual replacement. Consider the GLSL program which can be run in Shadertoy:

```
#define PI2 3.1415926535 * 2
// our real error is here: ^ (it should be a float)

void mainImage(out vec4 fragColor, in vec2 fragCoord) {
  vec2 uv = fragCoord / iResolution.xy * 32.;
  float h = sin(iTime + uv.x * PI2); // compile time error here
  float v = sin(iTime + uv.y * PI2); // and here
  fragColor = vec4(vec3(h, v, 0.), 1.);
}
```

Preprocessor macros with `#define` are very powerful. However, they can cause many cascading errors, the error messages of which can be very confusing since the compiler is unaware that the erroneous code is actually a result of text that the preprocessor expanded. The GLSL preprocessor, like the C preprocessor, is essentially a separate language; it does not have a semantic understanding of the code it is operating on, and sees everything after `#define`s and within `#ifdef` blocks as mere text.

Anecdotally, programmers often use `#define` to inline an expression in order to avoid repeated code. In Tinsl, `def` supports this use case, and reports type errors on the correct line and column, as seen in Figure 9.



```
def PI2 3.1415926535 * 2
// error shown here: ^
// it does not report cascading errors where PI2 is used

def seconds time / 1000. // unlike #define, comments can go on same line

fn stripes() {
  vec2 uv = npos * 32.;
  h := sin(seconds + uv.x * PI2);
  v := sin(seconds + uv.y * PI2);
  return vec4(vec3(h, v, 0.), 1.);
}

{ stripes(); }
```

```
tinsl: 1 error found:
at line 1 column 22: type mismatch: cannot do scalar operation `float * int`
```

Figure 9: The type error shows only on line 1 in the Tinsl programming environment.

By contrast, GLSL reports an error wherever the macro from `PI2` is used. Furthermore, the error message refers to expressions that do not appear on the line on which the error is reported due to the nature of the preprocessor. This can be seen in the Shadertoy environment in Figure 10.

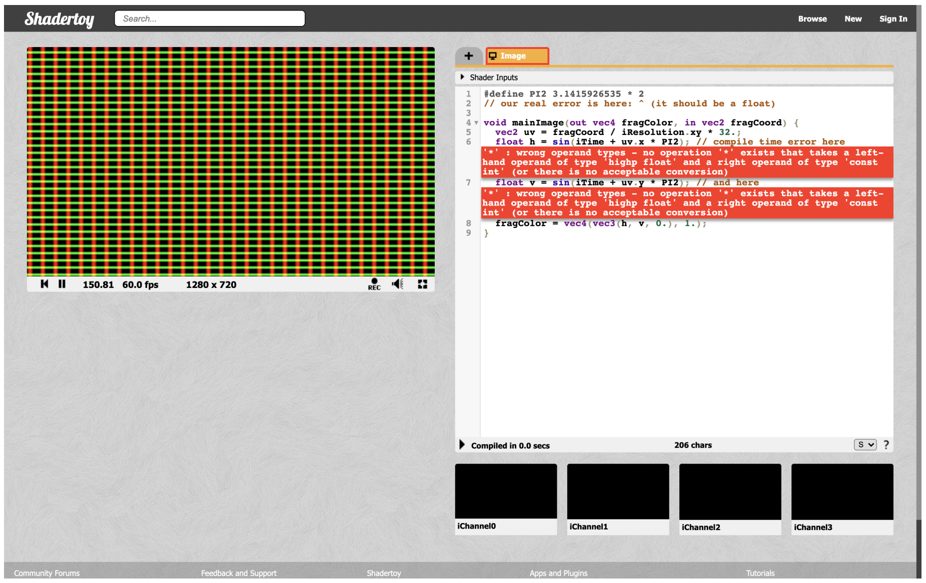
```
def PI2 3.1415926535 * 2
// error shown here: ^
// it does not report cascading errors where PI2 is used

def seconds time / 1000. // unlike #define, comments can go on same line
// the `def' expression could also span multiple lines

fn stripes() {
  vec2 uv = npos * 32.;
  h := sin(seconds + uv.x * PI2);
  v := sin(seconds + uv.y * PI2);
  return vec4(vec3(h, v, 0.), 1.);
}

{ stripes(); }
```

Additionally, the `def` expression cannot contain undefined symbols, unlike `#define`. It is true that the removal of preprocessor directives does remove with it some convenient features that are available in GLSL, and `def` in Tinsl is not a



```
#define PI2 3.1415926535 * 2
// our real error is here: ^ (it should be a float)

void mainImage(out vec4 fragColor, in vec2 fragCoord) {
  vec2 uv = fragCoord / iResolution.xy * 32.;
  float h = sin(iTime + uv.x * PI2); // compile time error here
'*' : wrong operand types - no operation '*' exists that takes a left-
hand operand of type 'highp float' and a right operand of type 'const
int' (or there is no acceptable conversion)
  float v = sin(iTime + uv.y * PI2); // and here
'*' : wrong operand types - no operation '*' exists that takes a left-
hand operand of type 'highp float' and a right operand of type 'const
int' (or there is no acceptable conversion)
  fragColor = vec4(vec3(h, v, 0.), 1.);
}
```

Figure 10: Type errors do not appear on line one, but multiple times wherever the macro is used in GLSL.

complete replacement. However, the simplification of `#define` into `def`, and the removal of so-called “unhygienic” macros such as `#ifdef` has multiple benefits for tooling; we have already seen how `def` allows for better error messages in the Tinsl live-coding environment.

4.1.5 Color Strings

Since GLSL is a language for graphics programming, and not for handling text, there is no concept of strings. Therefore, text enclosed with single or double quotes has no semantic meaning; a string-like token would result in a syntax error. In Tinsl, we use this as an opportunity to add syntactic sugar for creating a vector expression that represents a color. For example, `"#f00"` or `"#ff0000"` is equivalent to `vec3(1., 0., 0.)`. Conveniently, so is `"red"`; all 140 named HTML5 colors are available, and are insensitive to casing and whitespace, meaning `"cornflower blue"` is the same as `"CornflowerBlue"`. Single quotes can also be used instead of double quotes; they have the same meaning. Hex colors with alpha values such as `#f00f` or `#ff0000ff` will be de-sugared into `vec4`s instead of `vec3`s. To include an alpha value for a named color, Tinsl uses the syntax `"red"4`; the expression `"red"3` is also valid and potentially preferable to the user since it is more explicit, but ultimately unnecessary since `vec3` is the default for named colors.

4.1.6 Render Blocks

What separates Tinsl from shading languages is that Tinsl includes information about when to render to an intermediate target using “render blocks”, which is functionality that is beyond the scope of shaders. For this reason, Tinsl is not a shading language but rather a DSL to specify a pipeline for processing an image. The compiler emits a tree of shader source code with extra information about how to handle each rendering pass, such as whether to render to an intermediate texture or repeat a rendering pass (or group of passes) multiple times. Render blocks allow us to succinctly represent render-to-texture effects. Consider this short three-line program for a pixel-accumulation motion blur:

```
// prime the accumulation texture (texture 1) so it isn't empty
// do this only one time
once { frag0; } -> 1
// blend the new image with the accumulation texture
{ 0.03 * frag0 + 0.97 * frag1; } -> 1
// render the accumulation texture to the screen
{ frag1; }
```

The first render block primes the texture we are using for blending so that it does not start empty. This is so the image does not slowly fade in as the alpha value of the accumulation texture approaches 1 asymptotically. The `once` keyword indicates that this should only happen once. (If we copied texture 0 to texture 1 every draw call, there would be no blur effect.)

The second render block mixes the new image, `frag0`, with the accumulation texture, `frag1`. We then update the accumulation texture with the new blended image. Notice that we can read from texture 1 (with `frag1`) and render to texture 1 (with `-> 1`) in the same step, unlike OpenGL. This restriction that exists within OpenGL is abstracted away by the Tinsl runtime. (Because of this, when we say “texture 1”, we are not referring to the actual resource; we are referring to Tinsl’s abstraction around textures.) If we were to increase the coefficient of `frag0` and decrease the coefficient of `frag1` accordingly, we could make the effect of the motion blur less pronounced.

The final render block renders the contents of the accumulation texture, `frag1`, to the screen. The final render block (or final iteration of the render block, if it is looped) will always be rendered to the screen.

4.2 Implementing Bloom in Tinsl

Many post-processing effects require intermediary rendering targets and multiple passes. Bloom is an example of this; bright parts of an image are made to seem more bright by adding a blurry haze around “glowing” parts of the scene.

```
def threshold 0.5

fn luma(vec4 color) { return dot(color.rgb, vec3(0.299, 0.587, 0.114)); }
fn blur(vec2 direction, int channel) {
  uv := npos;
  off1 := vec2(1.3333333333333333) * direction;
  mut color := vec4(0.);
  color += frag(uv, channel) * 0.29411764705882354;
  color += frag(uv + (off1 / res), channel) * 0.35294117647058826;
  color += frag(uv - (off1 / res), channel) * 0.35294117647058826;
  return color;
}

pr two_pass_blur(float size, int reps, int channel = -1) {
  loop reps {
    blur(vec2(size, 0.), channel); refresh;
    blur(vec2(0., size), channel);
  }
}

{ vec4(frag.rgb * (step(1. - luma(frag0), 1. - threshold)), frag.a); } -> 1

1 -> { @two_pass_blur(size: 1., reps: 3);} -> 1

{ frag0 + frag1; }
```

The function `luma` calculates the brightness for a color that is passed in. The function `blur` takes a linear sample of points and returns a weighted average in order to blur the image along a single direction.

`two_pass_blur` is not a function, but a "procedure". Chunks of render blocks can also be extracted into procedures which allow the user to reuse these blocks of code. These are different than functions since they cannot return a value and can only be called within a render block. Procedures in Tinsl are like a limited form of macro that expand to a series of `vec4` expressions or render blocks—the kind of statement that can go within a render block. We use the procedure by preceding the name with `@`. This makes it clear that they do not behave like functions, and cannot be used like them. The `two_pass_blur` procedure runs a horizontal blur, refreshes the texture, then runs a vertical blur, and repeats this process `reps` times. The `refresh` statement will render out to the target texture before continuing; this enables the blur to happen in two separate passes. Procedures can be used to parameterize the source texture number of a render block (the optional number preceding the first arrow) or a texture number passed into `frag`. When any of these values are `-1`, this will default to the source texture number of the outer render block. We see this with the default argument for `channel`.

The first render block masks all pixels off that are below a certain brightness, and renders this result out to texture 1. The second render block blurs the contents of texture 1 and writes to texture 1. This happens three times for a stronger blur, since we passed in `3` as `reps` to the procedure `two_pass_blur` defined above. The final pass adds the blurred bright objects (on texture 1) with the original scene (stored on texture 0) together for the final effect. Being the last render block, this gets rendered to the screen.

4.3 Error Reporting

Of course, a compiler must not only compile valid source code; it must also reject invalid source code and report to the user where the problem arose, and why. Unfortunately, we cannot rely on the underlying GLSL compiler to provide users a "friendly" error message that describe why they are unable to run their program. Not only will the line and column numbers be completely different in the compiled GLSL source code compared to Tinsl; there are various features of Tinsl that GLSL does not have natively and are "de-sugared" into an equivalent version that will compile in GLSL. Therefore, the corresponding GLSL error message would have little to do with the language feature being misused in the corresponding Tinsl code, and would only confuse the user.

Furthermore, as we have touched upon throughout the paper, Tinsl does not get compiled to a single shader, but multiple shaders with metadata on how to handle the different render-to-texture steps. Therefore, not all Tinsl code generates corresponding GLSL; the code that governs these render-to-texture passes manifest as information outside of the GLSL fragment shader. Thus, the GLSL compiler would not be able to report such errors.

In Tinsl, there are nearly 400 unit tests, and many of these tests check that the proper error is thrown when an erroneous program is checked by the Tinsl compiler. There are approximately 80 such errors that can be reported to the user.

4.3.1 Type Checking Expressions

We have to check the GLSL-style code within the Tinsl program in order to meet our goal of providing early error detection. This includes type checking every expression, function call and return statement of every function definition within the program. Ideally, it is impossible to write Tinsl code that generates GLSL that does not compile for any reason. If the user is able to do this, we would consider it to be a bug in the Tinsl compiler. Type checking the Tinsl code is not merely a redundant check on top of GLSL's type-checker in order to provide better error messages to the user. Implementing a robust understanding of GLSL's type system enabled us to include static type inference in Tinsl, as discussed in Section 4.1.1. Providing accurate type checking that catches all errors before the code is generated involved implementing and testing many edge cases as specified by the OpenGL ES 3.00 language specification.

We also wanted to provide access to as many of GLSL's builtin functions as possible.[14] All of these functions are listed in Chapter 8 of the OpenGL ES 3.00 language specification [24, p. 84]. However, these functions are listed in a generic way. This generic-style syntax exists only in documentation and is not valid GLSL syntax. Tinsl uses this style of listing in order to type check GLSL's builtin functions. For example, consider the following example from page 90 of the language specification:

[14] If the Tinsl compiler becomes a frontend for supporting more languages other than GLSL, this will become more difficult to support.

```
genType mix(genType x, genType y, genType a)
genType mix(genType x, genType y, float a)
genType mix(genType x, genType y, genBType a)
```

This indicates that the builtin function `mix` can take in, as arguments, all floats or all floating point vectors of the same length. Alternatively, the first two elements can be floating point vectors of the same length if the last argument is a float. As a third option, the last argument can be a boolean vector of the same length as the first two floating point vectors. In all cases, the return type of `mix` is the same type as the first two arguments. In GLSL, if we were to make a user-defined function that conforms to this schema, we would have to define four nearly identical functions for the first listing alone. A GLSL programmer could potentially get rid of repeated code by utilizing a macro, however this solution does not provide the safety and readable error messages that a generic type system would supply. The creators of Slang noticed this lacking feature in HLSL and added generics into their HLSL-based shading language, which is just one of the many higher-level language features that Slang provides [10]. Tinsl does not yet support generics, however we hope to explore options for generics in a future version. One syntactic possibility for generics is to make `genType` and its variations a keyword in Tinsl.

4.3.2 Checking Texture Numbers

The other remaining step in verifying a Tinsl program is checking the Tinsl code that does not become compiled into GLSL directly. As mentioned previously, some sections of Tinsl code do not become GLSL when compiled. Instead, these sections of code relate to how off-screen textures are utilized, and manifest as metadata surrounding the fragment shader code. Namely, these features are “render blocks” and “procedures”. Consider the block of code:

```
pr copy(int src, int dest) {
  src -> { frag; } -> dest
}
```

This procedure is a helper that copies the contents of one texture to another. As a reminder, procedures in Tinsl are not functions; they get expanded to the contents of their body and can only be used within render blocks. Consider a program that abuses the fact that the source and destination textures in Tinsl are represented as `int`s for simplicity.

```
pr erroneous_copy(int src, int dest) {
  src -> { frag + float(src + dest); } -> dest
}
```

If a parameter in a procedure is used as a texture num (as `src` and `dest` are used in the procedure `copy`) it is essentially regarded as a different type at compile time. There are new restrictions on how to use these “texture ints”.

In the above code example, `src` and `dest` are being used for the source and destination textures as well as for basic arithmetic, as seen in the expression `src + dest`. In Tinsl, this would result in a "mixed use" error.

Additionally, `int`s used as textures have to be determined at compile time. Recall that we can use `frag` as a function and pass in a number for which texture to sample from. The number passed in is a "texture int", and is subject to the same restrictions as `src` and `dest` in the procedure definition `copy`. One of these restrictions is that it must be a simple integer so that it can be determined at compile time. Consider the following series of expressions:

```
// compiles
frag(0)

// does not compile (even though it's 0)
frag(int(sin(0)))
frag(1 - 1)

// does not compile
// user defined function cannot be determined at compile time)
frag(some_user_defined_func())
```

In Tinsl, we used integers to replace GLSL's `sampler2D` type with the hope that this would lead to a simpler, more terse language design. However, as we have seen through the code examples, integers used as texture numbers are subject to many restrictions that normal integers are not. This is because these integers are samplers "under the hood"; they become compiled into a corresponding sampler when translated into GLSL. Therefore, it is not completely fair to say we have coalesced integers and textures into the same type; instead, we have made `int` refer to two separate types, a true integer, and a sampler. The compiler determines whether an `int` is a sampler or an integer through usage. This has two major disadvantages. Firstly, this complicates the program validation step in the compiler; determining whether an `int` expression is not being used as a sampler number and a normal integer was non-trivial for us. Even worse, it is unlikely that a programmer would unwittingly use a procedure parameter for arithmetic and as a sampler at the same time unless they were trying to abuse the system. However, as compiler authors we are beholden to catching every semantic edge-case as unlikely as they may be. Secondly, this makes the signatures for functions and procedures less explicit. It is impossible to know without looking at the implementation whether a function or procedure accepts a normal integer or a texture number. For these reasons, we regard this as a design mistake; the next version of Tinsl will likely have an explicit type for "texture numbers" that is distinct from `int`.

4.4 Live Coding Environment

The Tinsl live coding environment can be thought of as existing between an environment like Hydra and an environment like The_Force, systems discussed on Section 2.2. It allows the user to program in what is essentially plain GLSL, while also providing Hydra-like semantics for rendering to other targets. Reading from and writing to intermediate textures will enables the user to create the playful, chaotic feedback-based rendering loops of Hydra, while giving GLSL-style code a first-class treatment instead of the "escape hatch" that Hydra provides. While Tinsl's success as a language for live coding will not be our primary method of evaluation, such an environment allows us to test and share Tinsl easily, while exploring possible use cases for the surrounding framework.

5 Evaluation

There are many possibilities for how to evaluate our work, since there are numerous ways in which our system could be used. There exist potential ways to audit code understandability automatically, such as the methodology presented in "'Automatically Assessing Code Understandability' Reanalyzed: Combined Metrics Matter"[17], using hundreds of features collected from code snippets combined with surveys given to developers. While such a methodology seems initially attractive, the amount of resources to assess "understandability" in this way exceeds the scope of this project. Instead, in this paper we will primarily build a qualitative evaluation using the framework in the paper "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework" [9]. The dimensions we will focus on (which are defined in the aforementioned paper) include: abstraction gradient, consistency, diffuseness, error-proneness, progressive evaluation and viscosity. This framework will guide our analysis of Tinsl, and has the benefit of going beyond "understandability" to include the many other factors listed previously. We also performed a user study where we asked participants to follow a short tutorial and answer a survey. Finally, we will compare GLSL shader code with the Tinsl source code of the bloom effect. This will give us a sense of the shaders the programmer would have to manage manually when doing raw GLSL programming.

5.1 Cognitive Dimensions Framework Analysis

Petre and Green present the "cognitive dimensions framework" as a "broad brush evaluation technique". This differs from traditional HCI evaluation techniques where the goal is to predict the amount of time the user will need to carry out specific tasks and learn new ideas. Petre and Green make the claim that the cognitive dimensions framework avoids "death by detail" and "is extremely quick and cheap: an afternoon of careful thought about a system is probably all that is needed" [9, p. 4]. The authors note that this type of evaluation is best utilized when supplemented with other techniques [9, p. 40]. In this paper, we will also evaluate Tinsl with a brief user study and an analysis based on the number of source lines of code. However, the cognitive dimensions framework will be our primary method of evaluation. We will step through several dimensions, modeling the structure of our analysis off of the example set in Petre and

Green's analysis of Basic, LabVIEW and Prograph. We will be analyzing the structure of the "bloom" program presented in Section 2.4, where it applies. While this program does not utilize every single language feature available in Tinsl (such a program would be rather contrived) it does use many important features. Bloom is an image processing operation that requires multiple fragment shaders, operating on the image in a pipeline, making use of more than one temporary texture. We will also make comparisons to how bloom might be accomplished working directly with GLSL, a process described in Section 2.4.

5.1.1 Abstraction Gradient

Abstraction gradient refers to the level of abstraction permitted within a language. Petre and Green claim that languages can be described as either "abstraction-hating, abstraction-tolerant, or abstraction-hungry" [9, p. 18]. One might be eager to claim that using GLSL combined with C or JavaScript is a particularly "abstraction-hating" way to get graphics on the screen. Compared to using a library like Threejs, this is certainly true. However, in the browser, the programmer is not interfacing with their graphics hardware as directly as it would seem. In Chromium-based browsers as well as Firefox, Google's ANGLE (Almost Native Graphics Layer Engine) is used to convert OpenGL calls to a native API. For example, when targeting Windows, ANGLE converts OpenGL calls to Direct3D calls, and translates the shaders from GLSL to HLSL [29]. In reality, OpenGL ES in the browser is an abstraction over non-GLSL API calls, and the GLSL code may get translated to one of many high-level shading languages. Unsurprisingly, these target APIs do not have complete feature parity with one another; OpenGL ES is a convenient "lowest common denominator" as it is a limited version of OpenGL designed for mobile devices. In this sense, the use of GLSL in the web is, in fact, a highly impressive abstraction to support hardware accelerated graphics on a wide array of target architectures. However, to the user, this abstraction is merely an implementation detail. For example, many programmers new to graphics programming are surprised by just how much "boilerplate" code is required to render something as simple as a textured rotating cube.

In Tinsl, of course, we are not concerned with rendering a cube, or any 3D mesh for that matter.[15] Specifically, Tinsl aims to simplify the use and authorship of post-processing effects. Thus, we provide abstractions for operations that are necessary or common when using GLSL to perform hardware-accelerated image processing. The user does not have to concern themselves with setting up a vertex shader and pushing data to a buffer on the GPU, among many other small details just to get OpenGL to render a flat rectangular image.

The main abstraction that Tinsl provides is render blocks (see Section 4.1.6). We would not suggest that graphics APIs like OpenGL would "do better" to include Tinsl-style render blocks. The way this is framed, such a suggestion does not make sense; the goal of GLSL and OpenGL is to program the GPU in a way

[15]In fact, Tinsl does render a mesh, however this mesh is two "big triangles" the size of the screen on which to render a texture.

that is "close to the metal" in order to perform arbitrary graphics tasks. The only reason we can create a DSL like Tinsl in order run image post-processing efficiently is because of the level of control that OpenGL gives us over the GPU.

5.1.2 Consistency

Consistency refers to what elements of a language can be inferred once a portion of the language is learned. Tinsl borrows many of GLSL's semantics since it is a familiar (and already well-specified) way to work with vectors and matrices. A large part of Tinsl is a parser and checker for a sizable subset of GLSL. In this sense, we have tried to make Tinsl as consistent as possible with GLSL.

For simplicity, we have not renamed any of GLSL's builtin function names. However, many of these function names are inconsistent in terms of casing and style. For example, `inversesqrt`, `smoothstep` and `faceforward` are not written in "camelCase" style and while functions like `matrixCompMult` and `outerProduct` are. Perhaps there is some internal consistency that follows a set rule that eludes us. However, we felt it best to not do any renaming so that GLSL's documentation for these functions could serve as Tinsl documentation as well.

There are corners of Tinsl that are actually stricter than GLSL, especially in regards to constructors, for the sole reason that we failed to glean everything from the language specification. For example, in GLSL, you can specify a `mat2` by passing in four floats, or by defining it by column vector by passing in two vectors. Tinsl supports these two forms. However, it is perfectly legal to specify all 4 elements of a `mat2` in this way:

```
mat4(vec3(.1, .2, .3), .4)
```

This would be a particularly strange way to construct a two-by-two matrix, however it is perfectly legal in GLSL. All vectors are flattened into their components in matrix and vector constructors. This was an oversight of ours, and we fully expect to find other ways in which Tinsl inconsistent with GLSL that are not by design.

Other syntactic differences between GLSL and Tinsl have to do with array declarations. In GLSL, there is some flexibility with where the square brackets can go. Consider the following code block:

```
// doesn't work in Tinsl, but works in GLSL
float a[3] = float[3](1., 2., 3.);
float b[] = float[](1., 2., 3.);

// works in tinsl and GLSL
float[3] c = float[3](1., 2., 3.);
float[] d = float[](1., 2., 3.);
```

We elected to use the latter syntax (`float[] a` instead of `float a[]`) because it aligns more closely with the constructor call, where the square brackets must immediately follow `float`.[16] Another inconsistency, which we argue is a design choice, is that variables can not be declared without being defined. This is discussed in greater detail in Section 4.1.1. It is not possible to use assignments like expressions.

Unlike GLSL, assignments can only be used as statements, similar to Go or Python prior to version 3.8. This simplification was made early on to make splitting Tinsl code into multiple shaders simpler. However, this constraint ended up not being necessary for implementation, and could be lifted in a future version of Tinsl.

5.1.3 Diffuseness/Terseness

Diffuseness and terseness refer to the amount of syntactic elements that are needed to express an idea. The GLSL-style syntax within Tinsl allows the user to be very terse. Syntactic features such as "swizzling" allows the user to reorder the components of a vector to create a new vector in a delightfully short and intuitive way.[17] Arithmetic operators can be used with scalars, vectors and matrices in a way that is natural for those who have familiarity with linear algebra. In Section 3.3, we highlight the diffuse nature of using merge-pass for simple arithmetic expressions; this is much improved in Tinsl. Beyond the GLSL style code, Tinsl provides render blocks, which gives the programmer the ability to describe arbitrarily complex texture-to-texture effects within a single file

The decision to use `int`s as texture numbers was made in the interest of terseness. However, in Section 4.3.2, we discuss some of the issues with folding integers and samplers into a single type.

5.1.4 Error-proneness

Error-proneness refers to how easy it is to make a mistake in the language, and whether elements of the language invite or prevent making errors. Perhaps the hardest and most time consuming part of implementing Tinsl was finding and testing all of the semantic errors that could occur, and reporting as many of these errors to the user as possible at the same time.

In order to parse the program into an abstract syntax tree, we used nearley, a parsing toolkit written in JavaScript. This allowed us to define a parsing expression grammar (PEG) to define the syntax of the language. nearley uses the Earley algorithm which handles left-recursive grammars very well; this let us specify our BNF-style (Backus-Naur form) grammar in a way that is easier to read. With another parser, such as Jison (which is based on GNU's Bison) we would have had to reorient our grammar so that it contained no left-recursive production rules [4].

While using a parser generator is convenient, this gave us less control over "synchronizing" syntactic errors. With a hand-written parser, it might have

[16]This form happened to be slightly easier to parse as well.

[17]For example, the user can reverse a three-component vector with `vec.zyx`.

been easier to report multiple syntactic errors, and synchronize by skipping to the end of the statement, which in Tinsl is demarcated by semicolons. In the current version of Tinsl, when nearley encounters an unexpected symbol, parsing stops.

However, our compiler is much better about reporting multiple semantic errors. Type errors and undefined identifier errors are examples of semantic errors. These errors make their way past the initial parsing because detecting them requires greater context than the grammar provides. Reporting as many errors at once is in some ways more art than science, because the compiler has to recover for programmer mistakes that result in a program that is incorrectly specified.

An implication of the basic static type inference we have in Tinsl is that if there is a type error on the right hand side of a variable declaration, the compiler cannot determine the type of that variable when it is used later in the program. We have a similar problem when a function's return type is ill-defined. For example, if a function returns values in different branches that have non-matching types, or if a function does not return at all, we can not say for certain what the intended return type of that function was meant to be. We do not have this problem in GLSL; even if the right hand side of a variable declaration is ill-formed, we know whether that variable is a float, int, etc. because the programmer is forced to be explicit by providing a type name before the identifier: `int x = 1;`. So that we do not get cascading errors where an ill-defined variable declaration or function is used, internally, we have an `undefined` type. Like the `any` type in TypeScript, operating on an `undefined` type with another type coalesces the entire type of that expression to `undefined`. For example:

```
undefined + int -> undefined
float - undefined -> undefined
undefined * undefined -> undefined
foo(undefined) -> undefined
```

In general, we aimed to provide descriptive semantic error messages that match up to the GLSL error messages provided by the browser. The Tinsl playground highlights the line and column where the error message appeared. While good error messages ensure that the programmer can correct their mistakes quickly, this does not necessarily speak to the error-proneness of a language. There are a handful of possible common errors we have observed in regard to the syntax of render blocks. One such example is that render blocks must contain semicolons after all statements within them, even if that render block only contains one statement on one line. For example, the following program would result in a syntax error:

```
0 -> { frag } -> 1 // BAD!!! does not compile; missing semicolon
```

The following render block, which copies the contents of texture 0 onto texture 1, is missing a semicolon after `frag`. Potentially, the semicolon is easier to remember when each statement is on separate lines.

```
// `brighten' and `sharpen' are user defined funcs that return vec4s
0 -> {
  brighten(frag, 1.);
  sharpen(prev, 1.);
} -> 1
```

Another source of potential errors is the way in which texture numbers are handled in Tinsl. In the Tinsl playground, the lowest texture number is the texture the video is written to. If the user "comments out" a block of code that contains the lowest-used texture number, the next lowest number will be used. This can cause confusing errors when testing the effect by breaking it into separate steps.

Because there is much overlap between the semantic errors possible between Tinsl and GLSL, we can compare the error messages directly. Tables 1, 2, 3 and 4 compare these error messages side-by-side. We believe that GLSL's error messages are likely more consistent and explicit overall, but we also believe that Tinsl's error messages are generally more terse and offer similar information.

Language	Error message
GLSL	`'+' : wrong operand types - no operation '+' exists that takes a left-hand operand of type 'const 2-component vector of float' and a right operand of type 'const 3-component vector of float' (or there is no acceptable conversion)`
Tinsl	`cannot do vector/matrix operation 'vec2 + vec3'`

Table 1: Adding a `vec2` with a `vec3`. Tinsl is more terse.

Language	Error message
GLSL	`'unknown_ident' : undeclared identifier`
Tinsl	`undefined identifier "unknown_ident"`

Table 2: Using an identifier before declaring it results in very similar messages.

Language	Error message
GLSL	`'foo' : no matching overloaded function found`
Tinsl	`required argument for "a" not filled in`

Table 3: Invoking function `foo(a)` with no arguments. The wording changes in Tinsl as it supports optional arguments; `a` has no default value.

5.1.5 Progressive Evaluation

Tinsl allows the user to progressively evaluate their render-to-texture effects using render blocks. In order to inspect the contents of a texture, the user

Language	Error message
GLSL	`'xg' :  illegal - vector component fields not from the same set`
Tinsl	`mixed sets (rgba, xyzw, stpq) in components xg`

Table 4: GLSL and Tinsl have separate error messages, but Tinsl reminds the user which vector component sets are allowed.

would simply that texture to the screen. To inspect texture 1, the user can add the following render block to the end of the program:

```
{ frag1; } // last render block goes to the screen
```

This provides a convenient way to incrementally test the program. For example, the line above could be used to debug the bloom program shown in Section 4.2. This allows the user to verify that the darker parts of the scene are being masked off correctly, and determine whether the threshold for luminance needs to be adjusted. In a situation where the rendering pipeline is hard-coded, like using OpenGL on its own, it is difficult to inspect any arbitrary texture used for the rendering processes. Tinsl makes reworking the rendering pipeline for debugging easy.

5.1.6 Viscosity

Viscosity refers to how difficult it is to make a change to the program. The main advantage of using Tinsl for post-processing effects is that the entire rendering pipeline can be described with render blocks. Any block of effects can be looped, or extracted out into procedures and reused. Tinsl can also make use of an arbitrary amount of intermediate textures. The programmer does not have to worry about allocating these textures and managing various other resources as the needs for the post-processing effect changes. Section 2.4 describes the process of implementing a bloom post-processing effect, and the resource management that goes into this. This set up to create a bloom can be described in a single Tinsl file. In this way, Tinsl makes it fluid to change the properties of post-processing effects, or author an entirely new post-processing step.

5.2 User Study

We prepared a written tutorial for users to follow within the creative coding environment, gathering data from 13 participants. We will refrain from performing any in-depth statistical analysis on the survey data due to the relatively small sampling size, and will instead summarize the results broadly. Charts for survey data can be seen in Appendix E.

The majority of questions were Likert-scales from 1-5 asking users to rate how strongly they agree with the following statements. The survey begins with questions relating to the user's experience. 69.2% of users chose "strongly disagree" for the statement "I am experienced in writing fragment shaders with

GLSL." We expected that most of our participants would not have experience with GLSL, since it is a niche, domain-specific programming language. By the way participants responded to agreement with the statement "The syntax of Tinsl looks familiar to me based on my previous programming experience" was higher, although grouped around the neutral response. The syntax of Tinsl borrows from GLSL for the definition of its functions, expressions and statements, which itself borrows from C. This suggests to us that our decision to keep with existing GLSL syntax, where possible, helped us create an accessible language even for those who are not experienced with GLSL. However, no participant "strongly agreed" with the statement. Perhaps this is expected, as one cannot expect a programmer to feel that a novel language is completely familiar.

Other questions assessed the language in tandem with the quality of the tutorial. We decided not to "quiz" our participants to test understanding, as this would lead to a much longer survey, and perhaps intimidate participants into thinking the survey was testing their programming aptitude; this would risk making our participants feel uncomfortable. Of course, we are not testing participant aptitude but rather the core design of the language and learning material surrounding it. Instead of quizzing our participants, we simply asked users to rate how strongly they felt they understood a particular section of the tutorial. The two statements posed to the participants relating to conceptual understanding were "I understood the code in the tutorial pertaining to motion blur" and "The difference between Tinsl keywords 'prev' and 'frag' was clear to me." The mode for these Likert-scale questions were both 3. Overall, this is encouraging for an approximately 10-15 minute written tutorial. However, in a future study, we would like to drill deeper into what prevented participants from feeling they understood the concepts fully. For now, it is not clear whether the blame lies in the design of the language or the design of the tutorial. In practice, it might be impossible to separate these two related factors.

We were also interested to see if participants were inclined to experiment with the system beyond the explicit instructions given by the tutorial. 38.5% of participants noted that they spent no time with experimentation, while the rest (61.5%) spent at least some time experimenting with Tinsl outside of the tutorial steps. Two participants reportedly spent more than 10 minutes experimenting with the system. Since this user study was offered to students at the beginning of two different related online lectures at WPI, the participants were under an external time constraint. Thus, students that spent longer on the tutorial steps might have been rushed into making sure they could complete the survey and return to lecture. Despite this, it was encouraging to see that participants were inclined to experiment even within the limited timeframe.

We wanted to gather information about the error messages and overall debugging experience of using Tinsl; an additional section of the online survey opens up if the user experienced compiler errors at all throughout their experience. Since under half of our users experienced any compiler errors while going through the tutorial (46.2%), it would be imprudent to come to any conclusions based on only six respondents. We will note, however, that one respondent was very dissatisfied with the error messages and highlighting, noting some confusion they had with regard to wording in the tutorial; the feedback was helpful and we will address the noted ambiguity in the instructions. However, it is encouraging that over half of our participants reportedly moved through the tutorial without experiencing any compiler errors.

The final section of the tutorial asked participants to consider a block of code containing novel syntax not taught explicitly within the tutorial. The code is repeated here:

```
// the same luma function in the tutorial previously
fn luma(vec4 color) {
  return dot(color.rgb, vec3(0.299, 0.587, 0.114));
}

{ mix('red'4, 'blue'4, luma(frag)); }
```

The unfamiliar syntax in the above code example are the `'red'4` and `'blue'4` color strings, which are not covered in the tutorial. We encouraged the user to be creative about how they might improve this code with the following prompt:

> If you found the syntax of the block of code to be undesirable, please rewrite the code in a syntax to suit your liking. Provide comments where the meaning of your new syntax is not obvious. If you are inclined to offer suggestions about syntax outside of the scope of the given code block, feel free to do so here. You may be as detailed as you want."

Participants did not share any ideas on how the syntax of this code block might be improved. Perhaps the open-ended nature of this section of the survey intimidated some users as it essentially asked the participant to redesign the language; we can only speculate. In retrospect, it might have been more useful for us to offer a set of options for alternative language syntax. We would have then asked the user which block of code expresses a particular idea more clearly. Despite not being able to solicit all the information from participants that we desired due to the nature of the open response questions, the information we gathered from the survey will help guide our design in the future. Overall, the survey results help to confirm our hypothesis that a DSL can improve the experience of authoring post-processing effects, as many participants grasped the core ideas in a very short tutorial.

5.3 Code Analysis

Like papers on shader metaprogramming systems and image-processing DSLs that precede it, we will also perform an evaluation based on lines of code [18][12][19]. Tinsl reduces the amount of shaders and total lines of GLSL needed to execute a post-processing effect with multiple render-to-texture steps. Consider the bloom Tinsl program shown in Section 4.2. The listing below is an edited version[18] of shaders output by the Tinsl compiler. Not pictured is the surrounding

[18]Whitespace was added, redundant groupings were removed and some assignments to variables were added for clarity and to keep the length of individual lines down.

information for looping certain shaders, namely the blur shaders.

```
// new file: masking shader
#version 300 es
precision mediump float;
out vec4 fragColor;
uniform sampler2D uSampler0;
uniform vec2 uResolution;
#define THRESHOLD .5

float luma(vec4 color) {
  return dot(color.rgb, vec3(.299,.587,.114));
}

void main() {
  vec2 uv = gl_FragCoord.xy / uResolution;
  vec3 col = step(1. - luma(texture(uSampler0, uv)), 1. - THRESHOLD);
  fragColor= vec4(texture(uSampler0, uv).rgb * col, 1.);
}

// new file: horizontal blur shader
#version 300 es
precision mediump float;
out vec4 fragColor;
uniform sampler2D uSampler1;
uniform vec2 uResolution;

vec4 blur(vec2 direction, sampler2D channel) {
  vec2 uv = gl_FragCoord.xy / uResolution;
  vec2 off1 = vec2(1.3333333333333333) * direction;
  vec4 color =vec4(0.);
  color += texture(channel, uv)*0.29411764705882354;
  color += texture(channel, uv + off1/uResolution) * 0.35294117647058826;
  color += texture(channel, uv - off1/uResolution) * 0.35294117647058826;
  return color;
}

void main() {
  fragColor = blur(vec2(1., 0.), uSampler1);
}

// new file: vertical blur shader
#version 300 es
precision mediump float;
out vec4 fragColor;
uniform sampler2D uSampler1;
uniform vec2 uResolution;

vec4 blur(vec2 direction, sampler2D channel){
  vec2 uv = gl_FragCoord.xy / uResolution;
  vec2 off1 = vec2(1.3333333333333333) * direction;
  vec4 color =vec4(0.);
  color += texture(channel, uv) * 0.29411764705882354;
  color += texture(channel, uv + off1/uResolution) * 0.35294117647058826;
  color += texture(channel, uv - off1/uResolution) * 0.35294117647058826;
  return color;
}

void main() {
  fragColor = blur(vec2(0.,1.), uSampler1);
}

// new file: copy texture 1 onto texture 2
#version 300 es
precision mediump float;
out vec4 fragColor;
uniform sampler2D uSampler0;
uniform sampler2D uSampler1;
uniform vec2 uResolution;

void main() {
  vec2 uv = gl_FragCoord.xy / uResolution;
  fragColor = texture(uSampler0, uv) + texture(uSampler1, uv);
}
```

The program listed above is 58 source lines of code. The Tinsl bloom program in Section 4.2 is 20 source lines of code, and encapsulates information for the entire rendering pipeline not seen in the above shaders. This information would be represented in the host code calling the OpenGL API, potentially in C or JavaScript.

6 Conclusion

We approached this study to answer the question of whether a DSL could improve the experience of authoring post-processing effects that require multiple rendering passes and temporary off-screen textures. We developed this research question after creating the post-processing library merge-pass, which did not contain its own DSL. We integrated merge-pass into various projects that spanned across multiple creative domains. After our evaluation, which involved a user study and a qualitative analysis using Petre and Green's cognitive dimensions framework, we believe the answer to be yes, a DSL can abstract away many of the tedious aspects of managing a rendering pipeline to create a non-trivial post-processing effect. Like we have done with merge-pass, we are eager to use Tinsl in new and exciting projects to further explore the possibilities it offers.

In the future, we would like to explore the possibility of authoring and maintaining "standard library" for Tinsl, which would include common operations, such as fast, efficient functions for transforming color spaces or for procedural noise. The fact that Tinsl contains syntax very similar to GLSL mitigates the fact that there is not much (if any) Tinsl code available on the web for people to re-use. We would like to encourage the authorship of reusable Tinsl modules by incorporating a module system with namespaces. Beyond this, there are a handful of other language features that were not implemented due to the time constraints of this research project, such as generics. We hope to include some of these features in later versions of the language.

There are not many up-to-date libraries for performing code transformations on shader code for the web. GLSLx[19] and glsl-parser [20] both do not have good support for GLSL 3.00, the latest version that WebGL 2 supports. We do not make any claim that the GLSL parsing we perform in Tinsl is somehow more powerful or robust than these two excellent libraries. However, some of the code within Tinsl could be used more generally for GLSL parsing and semantic validation. It will be interesting to see what changes WebGPU brings to the ecosystem of hardware accelerated graphics in the web; as of now, the shading language for WebGPU is not fully specified.[21] We believe it is safe to say that metaprogramming shaders will still be a productive way to work when WebGPU finally hits the mainstream; these metaprogramming solutions might even be a way to ease the growing pains of transitioning away from WebGL 2.

We have created multiple live demos and released multiple open-source projects throughout the project. Listed below are links to the repositories and demos:

- Tinsl source code: `https://github.com/bandaloo/tinsl`
- Tinsl live coding environment: `https://bandaloo.fun/playground`
- merge-pass source code: `https://github.com/bandaloo/merge-pass`
- post5 source code: `https://github.com/bandaloo/post5`
- artmaker source code: `https://github.com/bandaloo/art-maker`
- post5 sketches in web editor: `https://editor.p5js.org/bandaloo/sketches`
- merge-pass live demos: `https://bandaloo.fun/merge-pass/example.html`
- postpre live demos: `https://bandaloo.fun/postpre/example.html`

[19]`https://github.com/evanw/glslx`

[20]`https://github.com/stackgl/glsl-parser`

[21]As of May 2021, the document "WebGPU Shading Language" is stubbed out with many TODOs: `https://gpuweb.github.io/gpuweb/wgsl/`

7 Acknowledgements

Thanks to Charlie Roberts, who introduced me to many unique forms of artistic expression and swathes of creative tools that intersect with art and computer science; this motivated me to pursue these fields in an academic setting and I am very glad I did. Thanks to Gillian Smith for leading the generative design seminar which allowed me to create artmaker, a unique application of mergepass, and inspiring me to explore generative art starting from my undergraduate years at WPI. Thanks to Daniel Opreza for letting me use the art he created with Marching.js that you see in Figure 5. Thanks to Matthew Puentes and Joseph Petitti for doing QA testing on Windows and Linux respectively whenever I needed it.

References

[1] Takayoshi Amagi. *VJ app for Atom*. 2018. URL: `https://veda.gl/`.

[2] J.J. Birchman and S.L. Tanimoto. "An implementation of the VIVA visual language on the NeXT computer". In: *Proceedings IEEE Workshop on Visual Languages* (1992). DOI: `10.1109/wvl.1992.275767`.

[3] Ronja Böhringer. *Blur Postprocessing Effect*. Aug. 2018. URL: `https://www.ronja-tutorials.com/2018/08/27/postprocessing-blur.html`.

[4] Kartik Chandra and Tim Radvan. *nearley: a parsing toolkit for JavaScript*. 2014. DOI: `10.5281/zenodo.3897993`. URL: `https://github.com/kach/nearley`.

[5] Kate Compton and Michael Mateas. "Casual Creators." In: *ICCC*. 2015, pp. 228–235.

[6] Conal Elliott. "Programming Graphics Processors Functionally". In: *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004. URL: `http://conal.net/papers/Vertigo/`.

[7] *Feedback TOP*. Jan. 2019. URL: `https://docs.derivative.ca/Feedback_TOP`.

[8] *GLSL Shader programs*. Oct. 2020. URL: `https://docs.unity3d.com/Manual/SL-GLSLShaderPrograms.html`.

[9] T.R.G. Green and M. Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". In: *Journal of Visual Languages & Computing* 7.2 (1996), pp. 131–174. DOI: `10.1006/jvlc.1996.0009`.

[10] Yong He, Kayvon Fatahalian, and Tim Foley. "Slang: language mechanisms for extensible real-time shading systems". In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), pp. 1–13.

[11] Naoto Hiéda. July 2020. URL: `https://hydra-book.naotohieda.com/`.

[12] Calle Lejdfors and Lennart Ohlsson. "PyFX-An Active Effect Framwork". In: *The Annual SIGRAD Conference. Special Theme-Environmental Visualization*. 013. Linköping University Electronic Press. 2004, pp. 17–24.

[13] Michael D McCool, Zheng Qin, and Tiberiu S Popa. "Shader metaprogramming". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2002, pp. 57–68.

[14] Morgan McGuire et al. "Abstract Shade Trees (preprint)". In: *Symposium on Interactive 3D Graphics and Games*. Redwood City, CA, Mar. 2006. URL: `http://www.cs.brown.edu/research/graphics/games/AbstractShadeTrees/index.html`.

[15] *RenderTexture.GetTemporary*. Oct. 2020. URL: `https://docs.unity3d.com/ScriptReference/RenderTexture.GetTemporary.html`.

[16] Charlie Roberts. "Coding Ray Marchers with Marching.js". In: *International Conference of Live Coding*. 2018.

[17] Simone Scalabrino et al. "Automatically assessing code understandability". In: *IEEE Transactions on Software Engineering* (2019).

[18] Kerry A Seitz Jr et al. "Staged metaprogramming for shader system development". In: *ACM Transactions on Graphics (TOG)* 38.6 (2019), pp. 1–15.

[19] Kai Selgrad et al. "A High-Performance Image Processing DSL for Heterogeneous Architectures." In: *ELS*. 2016, pp. 39–46.

[20] "Sh Embedded Metaprogramming Language". In: (2010). URL: `http://libsh.sourceforge.net/`.

[21] *Shading language used in Unity*. Oct. 2020. URL: `https://docs.unity3d.com/Manual/SL-ShadingLanguage.html`.

[22] David William Wallace Sheets and Ashima Arts. "gloc: Metaprogramming WebGL Shaders with OCaml". In: (2012).

[23] Robert Simpson and John Kessenich. *The OpenGL ES Shading Language*. May 2009. URL: `https://www.khronos.org/files/opengles_shading_language.pdf`.

[24] Robert Simpson et al. *The OpenGL ES Shading Language Version 3.00*. Jan. 2016. URL: `https://www.khronos.org/registry/OpenGL/specs/es/3.0/GLSL_ES_Specification_3.00.pdf`.

[25] Ryan Ross Smith and Shawn Lawson. "The dark side". In: *International Conference on Live Coding 2017*. 2017.

[26] Steven L. Tanimoto. "VIVA: A visual language for image processing". In: *Journal of Visual Languages & Computing* 1.2 (1990), pp. 127–139. DOI: `10.1016/s1045-926x(05)80012-6`.

[27] Natalya Tatarchuk and Chris Tchou. *Destiny Shader Pipeline*. 2017. URL: `http://advances.realtimerendering.com/destiny/gdc_2017/`.

[28] Joey de Vries. *Bloom*. June 2014. URL: `https://learnopengl.com/Advanced-Lighting/Bloom`.

[29] Shannon Woods et al. "ANGLE: Bringing OpenGL ES to the Desktop". In: *GPU Pro 360 Guide to 3D Engine Design* (2018), p. 287.

A Donut Code

```
march(
  a = Torus().scale(1).translate(0, 0.4, 2.6).texture('checkers', {scale: 6}),
  p = Plane().texture('checkers', {scale: 2}),
  p2 = Plane(Vec3(0, 0, 1), 9).texture('checkers', {scale: 1.5})
)
.post(Bloom(.1), Edge(), h = Hue(), Invert(1), f = Focus(.15), Antialias(4))
.light(Light(Vec3(0, 4, 2), Vec3(0)))
.shadow(4)
.render('med').camera()

onframe = t => {
  p.texture.uv = 4 + Math.abs(t) * 0.5
  a.texture.uv = Math.abs(t) * 0.3
  a.rotate(t * 15, 1, 0, 1)
  h.shift = t / 5
}
```

B Tutorial

The following is the tutorial that was presented to participants of the user study. The links in this tutorial were removed. The language in the survey is more informal than the rest of this paper as it is meant to be an approachable way to learn the language.

If you intend to fill out the survey by the end of the tutorial, visit the survey link now in order to read and agree to the informed consent form before proceeding. Keep the survey open and return to it by the end of the tutorial. Thank you for your participation!

tinsl is a language for creating post-processing effects that can run in real time. It would be helpful to know a little bit about fragment shaders for this tutorial, however, if you are completely new to this, that is okay too. If you want to learn more about fragment shaders, The Book of Shaders] is a fun way to get started.

Follow along in this tutorial with the tinsl playground. Chrome is the recommended browser for this. Firefox should also work, but use Chrome if you have the option. Resize your window to make it wide enough so that nothing is cut off. You can enable your webcam and microphone for this. The microphone is used for audio-reactive components that are not required for this tutorial. However, if you do not have a webcam or you do not want to give permission to use it, that is okay too. You will get a test image that you can use to complete the tutorial.

What separates tinsl from a shading language? (After all, the acronym for tinsl claims that it's not one.) tinsl allows you to specify a rendering pipeline using special semantics to render to off-screen textures. This is not as complicated as it sounds. Let's see how that works:

```
{ vec4(1., 0., 0., 1.); } -> 0 // render to texture 0 (red)
{ vec4(0., 0., 1., 1.); } -> 1 // render to texture 1 (blue)
{ frag0 + frag1; } // add texture 0 and texture 1, render to screen (magenta)
```

What you see in the previous code example is a series of three "render blocks". Render blocks are a series of expressions that evaluate to a `vec4` (a four component floating point vector) separated by semicolons, enclosed by curly brackets. In this particular example, we only have one such expression statement in each block. Each component of the `vec4` corresponds to red, green, blue and alpha (transparency) in that order. In the first line, we create a vector to represent solid red, with an instruction to render the final output of that block to texture 0.

In the next line, we do the same thing, but we make texture 1 blue. In the final render block, we sample from the two textures we wrote to and add them together. Red and blue make magenta, so that's what we see.

(Note: in the playground, the lowest texture number used in the program is the texture that the video feed is on. We'll use zero for all these examples, but if you ignore texture 0 by commenting out a line of code, the video feed will be on the next lowest texture number. Keep this in mind! We're overwriting the video texture in this example because we don't care about it.)

Perhaps you don't find a pink screen particularly compelling. Tough, but fair. Let's create a feedback effect that simulates motion blur. We do this by using an extra texture texture for color accumulation. Delete everything and paste this in:

```
once { frag0; } -> 1 // prime our color accumulation texture just once!
{ frag0 * 0.03 + frag1 * 0.97; } -> 1 // accumulate colors
{ frag1; } // render to the screen
```

Run this program in the playground and wave your hand around. Many video games use this kind of effect to simulate drunkenness and I think it's pretty apparent why. (Motion blur in modern video games is not often done with color accumulation anymore. Instead, objects are blurred based on their velocity; this method allows camera movement to be removed from the motion blur equation, which is just way nicer for actual gameplay.)

The first line is a bit of a nitpick. If you got rid of it, the image would slowly fade in, but with this small addition we can copy the contents of our video stream texture on the first draw call. `once` lets us do this, well, once. If we left off the `once`, we'd be overwriting our accumulation texture each draw call, which we don't want! This would result in no blur at all.

The next line blends a bit of our video stream (3% of it) with a lot of our accumulation texture (97% of it). You can bump these coefficients around to see how this changes the final effect. (If they don't add up to 1 the feedback loop might blow up and go to white!) The values give us a very pronounced effect.

The last line renders the accumulation texture to the screen. If we forgot this line (try it, comment it out) we'll still see an image but we don't get a motion blur. This is because the last render block in a tinsl program always goes to the screen. The `-> 1` of the previous block is ignored, and texture 1 never gets used as an accumulation texture. Keep this in mind! If you think this is a footgun and overall design flaw, I'm inclined to agree. However, this is how it is for now.

Okay, let's do another effect. Delete everything! In Tinsl, we can have functions that look like GLSL. In fact, (nearly) all of the builtin functions and operators of GLSL 3.00 can be used in tinsl function definitions. Paste this in:

```
fn luma(vec4 color) {
  v := vec3(0.299, 0.587, 0.114); // coefficients for each color channel
  return dot(color.rgb, v);
}
```

You can think of this function as taking in a color and returning how bright it is. Notice that we can declare variables with `:=` to get static type inference, and we don't need to specify the return type of a function either. tinsl takes care of that. We could have been explicit and written this function in the GLSL compatible way. This is helpful if you're just pasting in GLSL functions you find on the internet:

```
float luma(vec4 color) {
  vec3 v = vec3(0.299, 0.587, 0.114);
  return dot(color.rgb, v);
}
```

But, the first way is nicer, don't you think? (Truthfully, there are arguments to be made for either style.) Moving on, let's use this function to turn our camera feed into a bespoke black-and-white. We'll write a simple function to do this:

```
fn black_and_white() {
  gray := vec3(luma(frag)); // grayscale rgb value
  return vec4(gray, 1.); // return a gray color with an alpha of 1.
}
```

You'll notice we left off the number after `frag`. When there's no number, tinsl will sample from the "in number" of the enclosing render block. We do that by including an arrow before the render block:

```
0 -> { black_and_white(); }
```

Run the program now to check that everything's in black and white. Let's do something that maps the domain of the image to a different coordinate space. In other words, let's turn the image upside down. Delete the render block we just wrote and replace it with this:

```
0 -> { frag(vec2(0., 1.) - npos); }
```

Run it, and you'll see that you're now upside down (and in full color again). I mentioned earlier that you could have multiple 'vec4' expression statements. Let's do that, and add back in our black and white filter. Augment the existing render block to look like this:

```
0 -> {
  frag(vec2(0., 1.) - npos);
  black_and_white();
}
```

If we call `frag` like a function and pass in a `vec2`, we can sample from off-center. The ‘npos‘ keyword is the normalized pixel position. We invert the y component to flip the image.

Run this, and wait...what happened? You’re in black and white again, but you’re now right side up. The issue here is that we want the color of the previous operation; we don’t want to sample from the original image again, like `black_and_white` does with `frag`. The `prev` keyword lets us do this. To fix our issue, let’s make `black_and_white` take in an argument. Update the function definition, and while we’re at it let’s just inline the luma part:

```
fn black_and_white(vec4 color) {
  return vec4(vec3(luma(color)), 1.);
}
```

Now, update the call to `black_and_white` and pass in `prev`:

```
0 -> {
  frag(vec2(0., 1.) - npos);
  black_and_white(prev);
}
```

To summarize, we can use `frag` like a function and pass in a `vec2` to sample from a different position. This is useful in conjunction with ‘npos‘, which allows you to transform the coordinate space. `prev` lets us chain together steps without breaking up a render block. As an aside, if we *really* wanted to break these into separate steps, we could have broken this into two render blocks:

```
0 -> { frag(vec2(0., 1.) - npos); } -> 0
0 -> { black_and_white(frag); } // renders to screen
// don't do this if you don't have to!!!
```

This works because the `black_and_white` call is in a separate render block, and `frag` has access to a fresh new texture 0. As the code comment indicates, don’t do this if you don’t have to! The single render block version from earlier is more efficient. However, there are cases where breaking an operation into two passes is the most efficient option (see: Gaussian blur, which is a mathematically separable image processing filter.) We can do the same thing with the `refresh` keyword:

```
0 -> {
  frag(vec2(0., 1.) - npos);
  refresh; // now `frag` has access to the updated fragments
  black_and_white(frag);
}
```

B.1 Survey Code

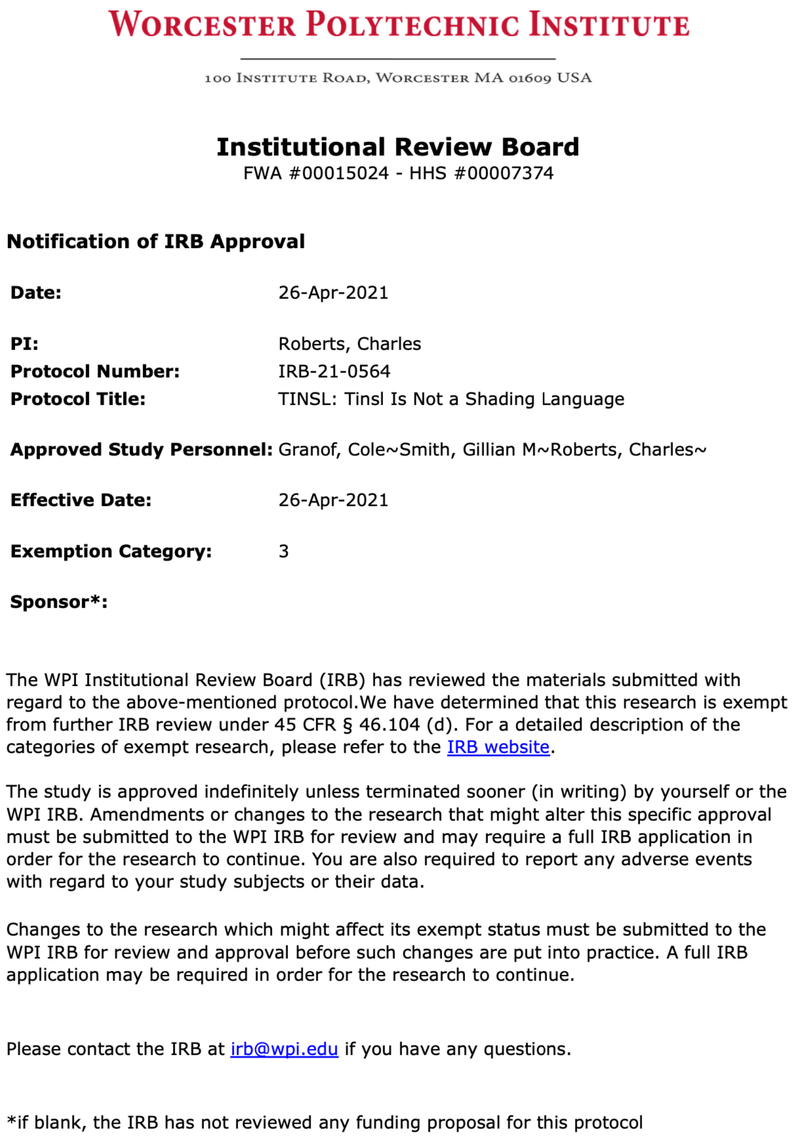
Take a look at the following block of code. It may contain syntax you are unfamiliar with, so you are not expected to accurately predict what it will do. Even so, please try to guess at the effect it will produce. Once you have made your guess, run the following code. Take a note of whether your intuition was correct; it will be asked on the survey.

```
// the same luma function in the tutorial previously
fn luma(vec4 color) {
  return dot(color.rgb, vec3(0.299, 0.587, 0.114));
}

{ mix('red'4, 'blue'4, luma(frag)); }
```

Thank you for going through the tutorial! Return to the survey (you should have the link open already; if not, here it is) and complete the survey. Thanks again!

C IRB Approval Certificate



WORCESTER POLYTECHNIC INSTITUTE

100 Institute Road, Worcester MA 01609 USA

Institutional Review Board

FWA #00015024 - HHS #00007374

Notification of IRB Approval

Date:	26-Apr-2021
PI:	Roberts, Charles
Protocol Number:	IRB-21-0564
Protocol Title:	TINSL: Tinsl Is Not a Shading Language
Approved Study Personnel:	Granof, Cole~Smith, Gillian M~Roberts, Charles~
Effective Date:	26-Apr-2021
Exemption Category:	3
Sponsor*:	

The WPI Institutional Review Board (IRB) has reviewed the materials submitted with regard to the above-mentioned protocol.We have determined that this research is exempt from further IRB review under 45 CFR § 46.104 (d). For a detailed description of the categories of exempt research, please refer to the IRB website.

The study is approved indefinitely unless terminated sooner (in writing) by yourself or the WPI IRB. Amendments or changes to the research that might alter this specific approval must be submitted to the WPI IRB for review and may require a full IRB application in order for the research to continue. You are also required to report any adverse events with regard to your study subjects or their data.

Changes to the research which might affect its exempt status must be submitted to the WPI IRB for review and approval before such changes are put into practice. A full IRB application may be required in order for the research to continue.

Please contact the IRB at irb@wpi.edu if you have any questions.

*if blank, the IRB has not reviewed any funding proposal for this protocol

Figure 11: IRB approval certificate.

D Additional Screenshots

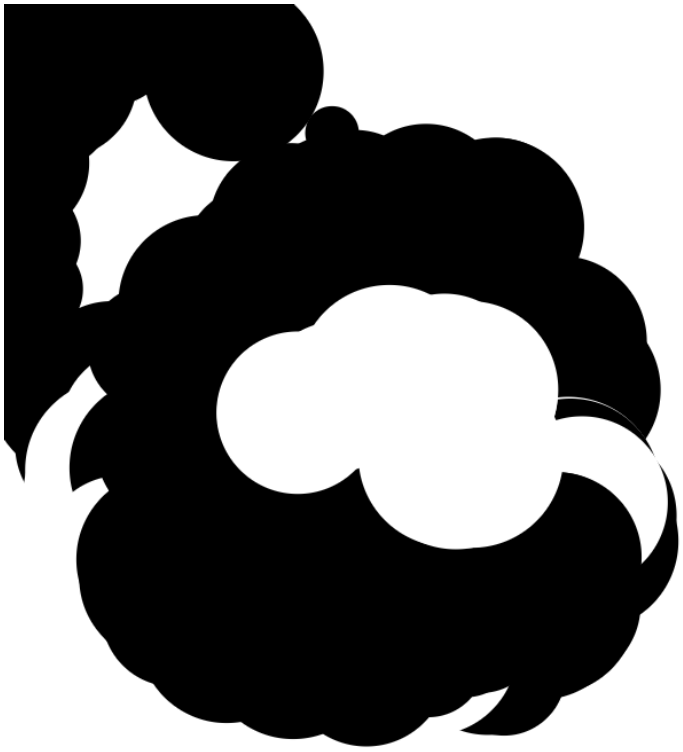


Figure 12: Processing sketch depicted in Figure 6 without post-processing.

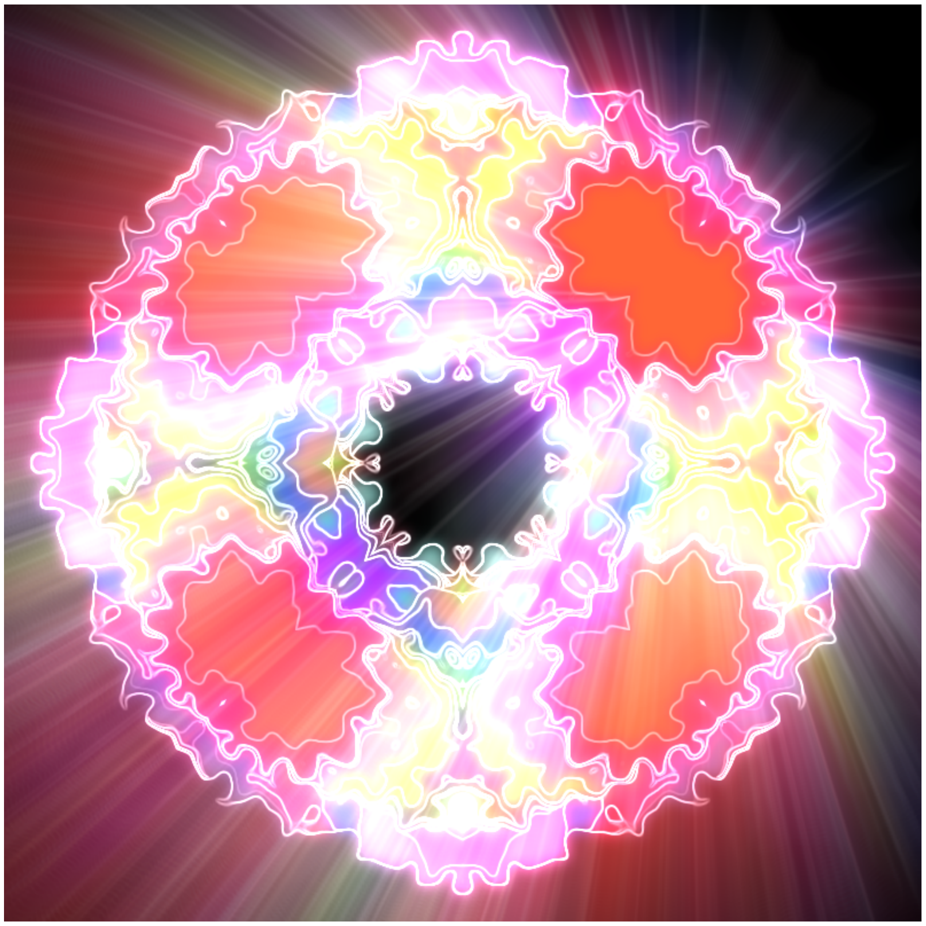


Figure 13: “Mandala” p5 sketch using layered post-processing effect.

E Survey Results

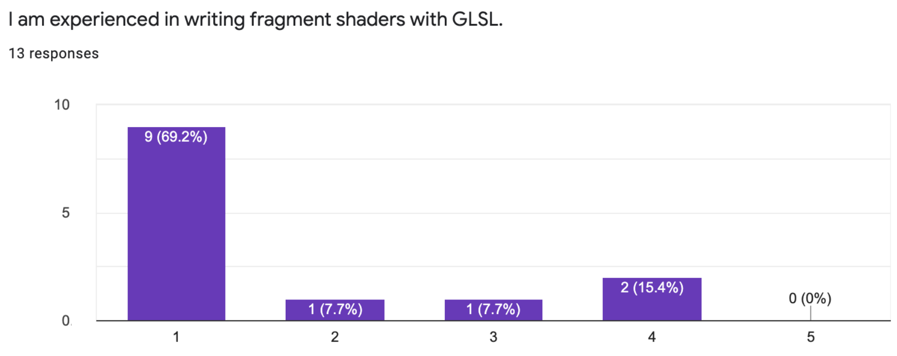


Figure 14: Results for question about GLSL experience.

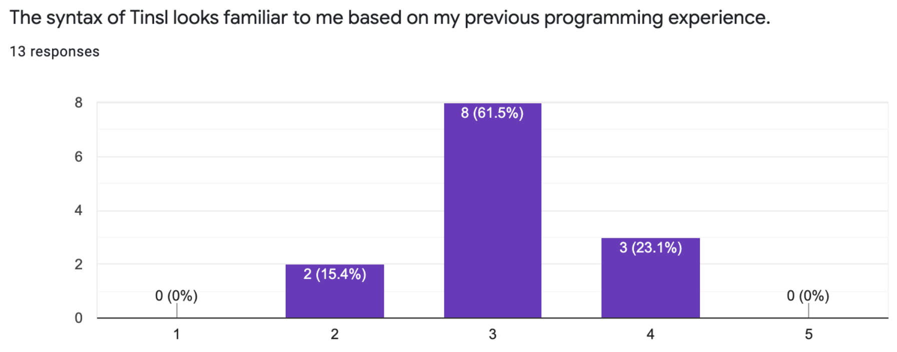


Figure 15: Results for survey question gauging familiarity of the syntax.

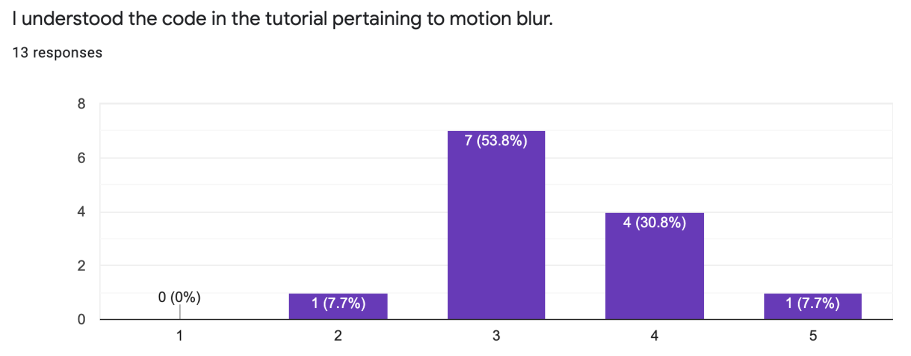


Figure 16: Results for question asking about comprehension of the motion blur effect.

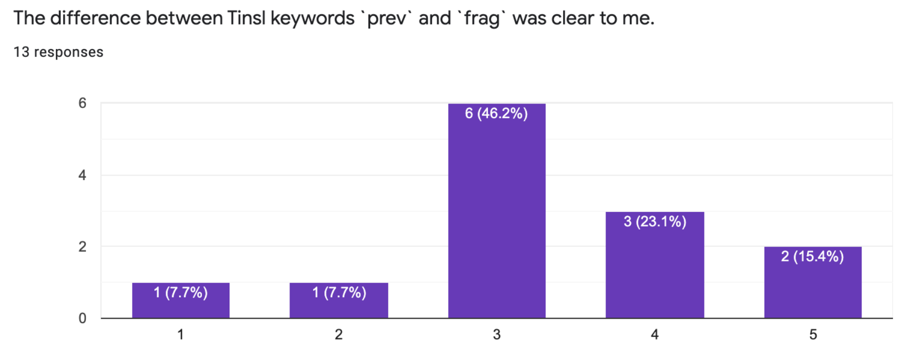


Figure 17: Results for question asking about comprehension of the `prev` and `frag` keywords.

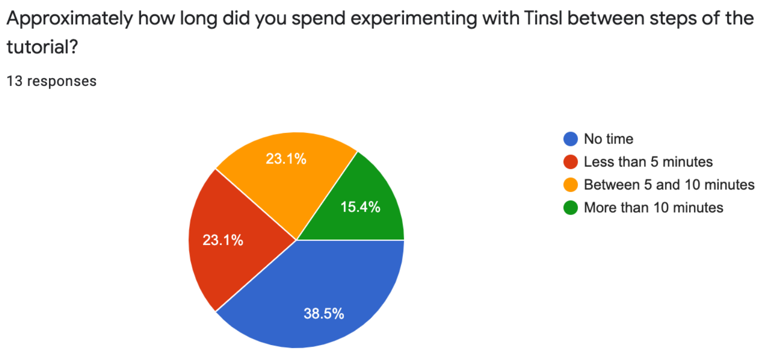


Figure 18: Results for question gauging whether participants spent time with the system outside of explicit steps within the tutorial.

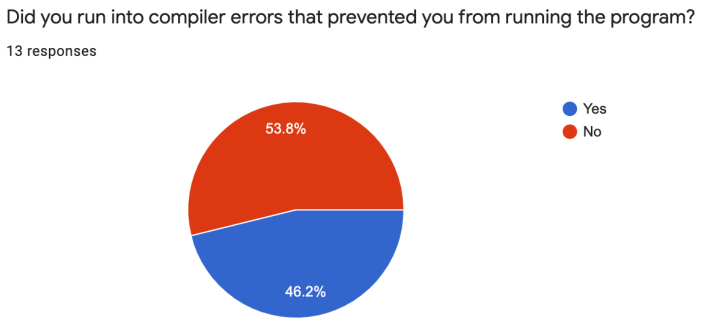


Figure 19: Results for question asking if the users experienced errors.

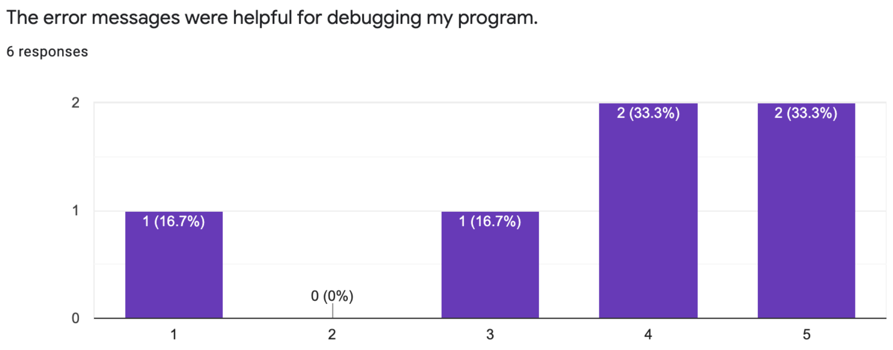


Figure 20: Results for question asking about error messages.

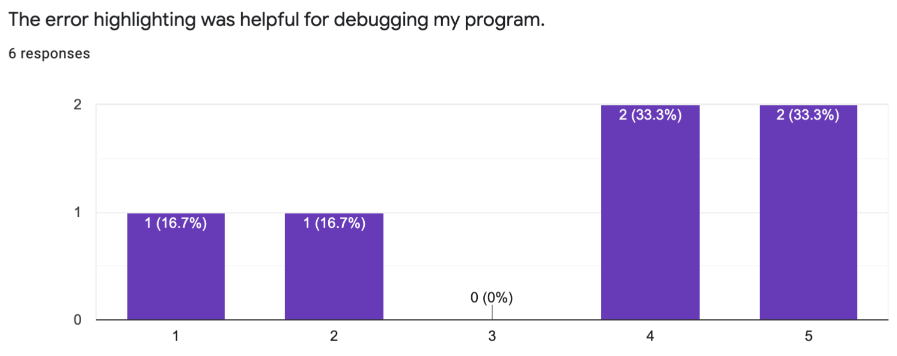


Figure 21: Results for question asking about error highlighting.

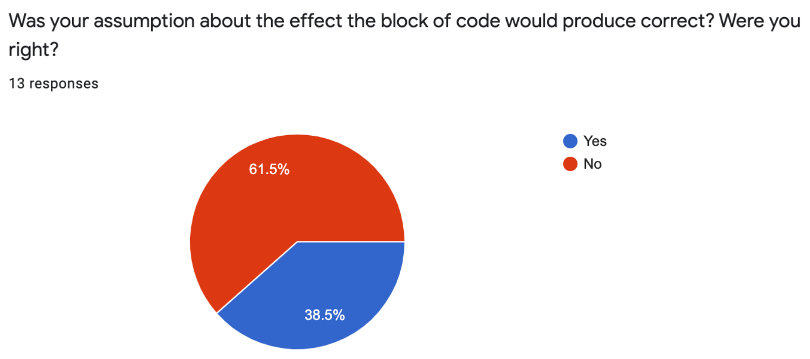


Figure 22: Results asking about comprehension of code block.