SplitShade: WebGPU Playground

Thesis - MSc. In Computer Animation and Visual Effects

Anu Kriti Wadhwa (s5647918)

Bournemouth University

Table of Contents

1 Introduction

The web has developed into a primary platform for communication, commerce, education, entertainment, and creative work. Alongside this growth, there has been increasing demand for visually rich and responsive applications delivered directly through the browser [1, 2]. Real-time graphics are central to these experiences, yet achieving them requires accessible tools for programming modern graphics hardware.

Graphics Processing Units (GPUs) were originally designed to accelerate three-dimensional rendering by offloading tasks from the Central Processing Unit (CPU) [3, 4]. Their highly parallel architecture now supports not only real-time rendering but also general-purpose compute, including applications in high-performance computing, machine learning, and large-scale data processing [4, 5]. Browser-based access to GPU functionality has followed a steady trajectory: from early plugin-based solutions such as Adobe Flash, to standardised APIs like WebGL, and most recently to WebGPU – a low-level API intended to support both graphics and compute workloads [6, 7].

While WebGPU represents a significant advance, its adoption is hindered by complexity. Programming in the WebGPU Shading Language (WGSL) requires developers to construct full pipelines, manage buffers and bindings, and resolve low-level errors before producing any visual output [8]. This technical overhead creates a steep learning curve, particularly for beginners and those interested in rapid prototyping or creative experimentation. In contrast, environments such as ShaderToy and GLSL Sandbox made WebGL more accessible by abstracting boilerplate code and providing immediate visual feedback. Equivalent tools for WebGPU remain underdeveloped.

This thesis presents **SplitShade**, an interactive shader playground designed to address this gap. SplitShade enables users to write fragment and vertex shaders in WGSL directly in the browser and view the results in real time. It provides built-in uniforms, texture and mesh support, error reporting, and a minimal interface intended to reduce barriers to entry. By abstracting low-level setup and surfacing results immediately, SplitShade lowers the threshold for engaging with WebGPU, thereby supporting learning, experimentation, and creative exploration.

The remainder of this thesis is organised as follows. Section 2 reviews prior work on browser-based graphics programming and shader playground environments. Section 3 outlines the technical background of WebGPU and WGSL. Section 4 describes the implementation of SplitShade, including system design, user interface, and technical challenges encountered. Section 5 concludes with a discussion of potential applications, limitations, and future directions.

2 Previous work

WebGPU continues a long trajectory of efforts to enable GPU access in the browser. Situating SplitShade within this context requires examining how earlier technologies and tools shaped current approaches to browser-based graphics programming. This section reviews key milestones in that evolution: the introduction of WebGL, the emergence of shader-focused platforms such as GLSL Sandbox and ShaderToy, and the subsequent transition toward WebGPU. These developments highlight the growing need for accessible, web-based environments that reduce the technical barriers to shader programming.

2.1 WebGL

WebGL marked a pivotal moment in the history of browser graphics. Introduced in 2011, it was the first widely adopted API to expose GPU functionality directly within the browser, without relying on external plugins like Flash. Based on OpenGL ES 2.0, it allowed developers to write interactive 2D and 3D graphics using JavaScript and GLSL (the OpenGL Shading Language) [9], enabling everything from data visualizations to games and creative visual experiments [1].

Its release sparked a wave of experimentation and tool development on the web [10]. With access to programmable shaders in a browser context, artists and developers began to explore GPU programming for both educational and expressive purposes. WebGL was built with experienced graphics programmers in mind, those familiar with shaders, matrix math, and GPU buffer management. Rendering a basic scene would require setting up a canvas context, compiling and linking vertex and fragment shaders, managing data buffers, and handling camera transforms through projection matrices [1].

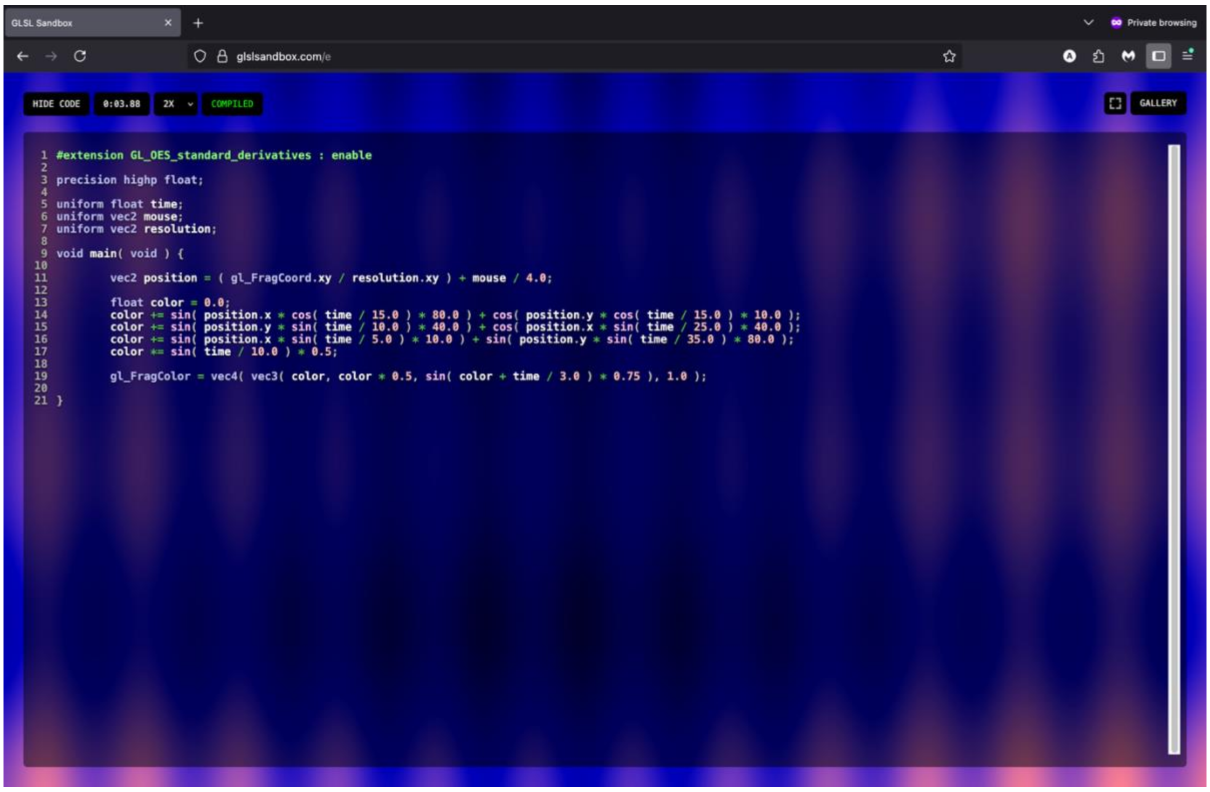
While WebGL brought programmable shaders to the browser, its low-level nature required significant boilerplate code and a solid grasp of graphics fundamentals. In parallel with its adoption, developers began building environments that streamlined this workflow and made shader development more approachable. **GLSL Sandbox** and **ShaderToy** are some examples of these, enabling users to write fragment shaders and see visual output immediately, directly in the browser.

2.2 GLSL Sandbox

One of the earliest browser-based shader editors was GLSL Sandbox [11].

It offered a minimalist environment for writing GLSL fragment shaders with real-time visual feedback in the browser, significantly lowering the barrier to shader experimentation [11]. The shader was automatically applied to a full-screen quad, removing the need to handle rendering setup, buffers, or camera logic [12]. By removing the need to set up a WebGL context or manage buffers manually, GLSL Sandbox opened the door for more casual experimentation and creative coding, particularly among artists and newcomers to GPU programming. The interface (Figure 1) consisted of a live editor with syntax highlighting and an immediate visual output panel, making the feedback loop tight and intuitive.

GLSL Sandbox offered a simple, minimal environment for writing fragment shaders in the browser, but it lacked mechanisms for authorship or community interaction. Users could share shaders, but there was no way to attribute or discuss them. **ShaderToy**, released in 2013, addressed these limitations by supporting user accounts, metadata, and threaded comments, strengthening both the technical tooling and the surrounding community [13, 14].



```
#extension GL_OES_standard_derivatives : enable

precision highp float;

uniform float time;
uniform vec2 mouse;
uniform vec2 resolution;

void main( void ) {

        vec2 position = ( gl_FragCoord.xy / resolution.xy ) + mouse / 4.0;

        float color = 0.0;
        color += sin( position.x * cos( time / 15.0 ) * 80.0 ) + cos( position.y * cos( time / 15.0 ) * 10.0 );
        color += sin( position.y * sin( time / 10.0 ) * 40.0 ) + cos( position.x * sin( time / 25.0 ) * 40.0 );
        color += sin( position.x * sin( time / 5.0 ) * 10.0 ) + sin( position.y * sin( time / 35.0 ) * 80.0 );
        color *= sin( time / 10.0 ) * 0.5;

        gl_FragColor = vec4( vec3( color, color * 0.5, sin( color + time / 3.0 ) * 0.75 ), 1.0 );

}
```

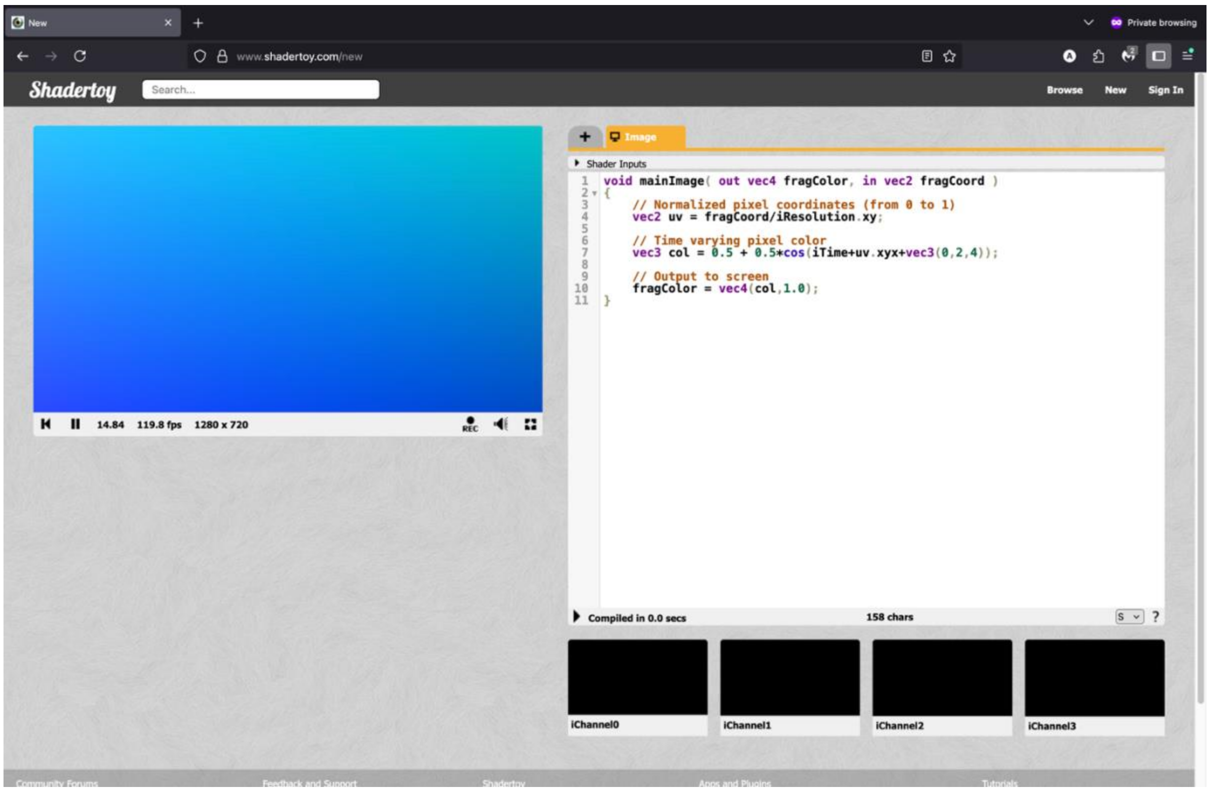
Figure 1: The GLSL Sandbox interface. A hide-able full-screen fragment shader editor is overlaid directly on top of the rendered output, allowing immediate visual feedback [11]

2.3 ShaderToy

ShaderToy was launched with the goal of building a shader programming community [13, 14].

ShaderToy's interface (Figure 2) embraces simplicity and immediate feedback just like GLSL Sandbox. It restricts the user to writing a single fragment shader, which is automatically rendered over a full screen quad, eliminating the need to manage geometry or rendering setup [15].

The ShaderToy interface (Figure 2) is split into two main areas: (1) a real-time visual output panel and (2) a built-in code editor. The editor features syntax highlighting, error reporting, and metadata such as compilation errors, while the output panel includes controls for playback, as well as contextual information like shader descriptions and user comments.



```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // Time varying pixel color
    vec3 col = 0.5 + 0.5*cos(iTime+uv.xyx+vec3(0,2,4));

    // Output to screen
    fragColor = vec4(col,1.0);
}
```

Figure 2: The ShaderToy interface. The environment is split into a real-time output window (left) and a code editor (right), with additional panels for input channels (bottom right) [13]

ShaderToy also supports limited multimedia input. Users can attach textures, videos, or audio streams to their shader via designated input slots, which then become accessible in code through uniform bindings. Additional rendering passes (such as audio or buffer passes) can be added through the UI, allowing more complex multi-stage effects while keeping the workflow approachable [15].

2.4 From WebGL to WebGPU

While WebGL enabled real-time 3D graphics in the browser through OpenGL ES 2.0, it lacked the explicit control and advanced features of newer low-level APIs such as Vulkan, Metal, and Direct3D 12, including support for multithreading, fine-grained memory management, and high-performance compute workloads [6].

To address the gaps of WebGL, WebGPU was designed as a modern graphics and compute API for the web, modelled on the contemporary graphics standards of the aforementioned newer APIs. WebGPU supports both graphics and general-purpose compute workloads, and promoting more predictable performance across platforms. Officially reaching its first stable release in 2023, WebGPU has marked a significant shift in how web applications can leverage GPU power [6]. Refer to Section 3.1 for the technical background of a WebGPU setup.

Despite its potential, WebGPU's adoption has been gradual due to its relative complexity and limited browser support until recently [7]. Tools and online playgrounds (akin to GLSL Sandbox and ShaderToy) that abstract away boilerplate code have not yet proliferated at the same scale. As of this paper, there is a gap for new playground environments tailored specifically for WGSL and WebGPU workflows.

2.5 Existing WebGPU playground environments

While several WebGPU editors and demo sites exist, none yet replicate the tightly scoped, user-friendly model established by WebGL-era tools like ShaderToy.

For instance, *WebGPUniverse* [16] appears promising but was observed to be unreliable in loading. Although it aims to foster community-driven sharing and abstracts some boilerplate code, its interface is cluttered, making the user experience less approachable.

The *06wj WebGPU Playground* [17], while offering an editor, is primarily a gallery of the developer's own shader examples. It does not abstract away low-level setup. Users must still manage pipeline configuration and resource bindings.

The *WebGPU Compute Playground* [18] supports live compute shader experimentation but does not render graphics or support fragment shaders, limiting its use for visual exploration.

Likewise, the *Babylon.js Playground* [19] offers robust WebGPU support for advanced 3D development, but it does not provide a minimal, fragment-shader-only editing experience.

These gaps highlight the absence of a true ShaderToy-style playground for WebGPU – a no-frills, intuitive interface where users can write a minimal fragment shader and instantly view the result, with all boilerplate code abstracted away.

3 Technical background

This section provides the technical context required to understand the design and implementation of SplitShade. It introduces the WebGPU API (Section 3.1) and its shading language (WGSL) (Section 3.2), explains the roles of vertex and fragment shaders within the graphics pipeline (Section 3.3), and outlines the forms of external input (uniforms and textures) that shaders can consume (Section 3.4). Together, these concepts establish the foundation on which the system is built.

3.1 WebGPU API basics

At the heart of WebGPU lies a structured pipeline for drawing to the screen, as illustrated in Figure 3 [8]. This pipeline begins with setting up key objects such as the **device**, **pipeline**, and **bind groups**, which together control how data flows from the Central Processing Unit (CPU) to the GPU and ultimately onto the **canvas**.

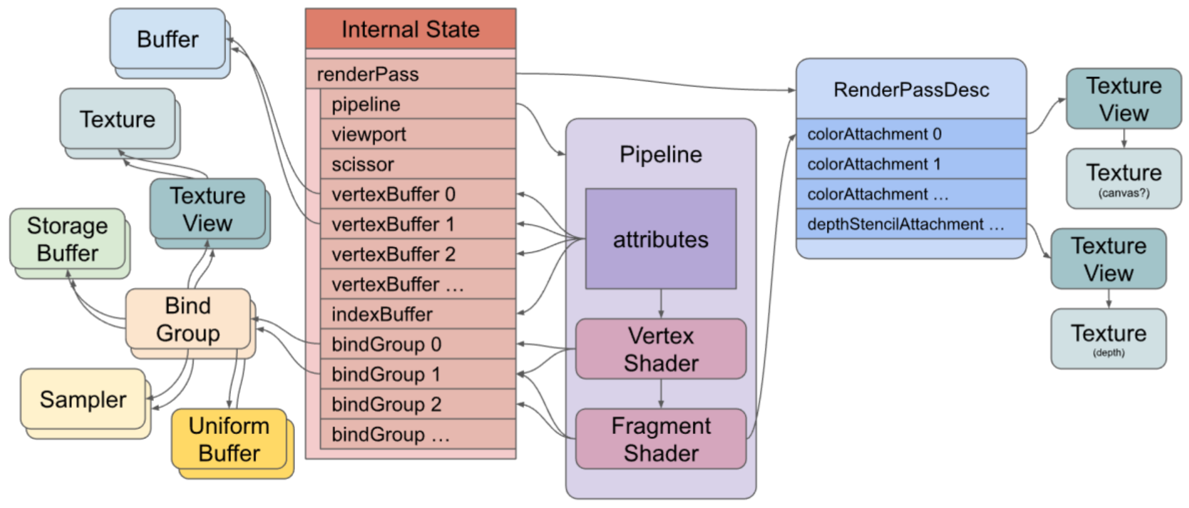


Figure 3: Simplified diagram of WebGPU setup to draw triangles by using a vertex shader and a fragment shader [8]

The following components represent the core building blocks of a basic WebGPU rendering pipeline (illustrated in Figure 3) Each plays a distinct role in setting up and executing a single draw call.

- **Device** and **Context**: These form the starting point. The WebGPU **device** serves as the interface to the GPU. It is used to create all subsequent GPU resources such as shaders, buffers, and pipelines. The **context** refers to the **canvas** WebGPU draws into

- **Render Pipeline**: This object links together the compiled vertex and fragment shaders, along with information about how the GPU should interpret vertex data and which format to render to. For non-graphics workloads, WebGPU instead uses a separate compute pipeline that contains a compute shader.
- **Vertex and Fragment Shaders**: These are the GPU programs executed per vertex and per pixel, respectively. The vertex shader processes position and attribute data, while the fragment shader determines the final colour output.
- **Compute Shaders**: These are general-purpose GPU programs that are not tied to the graphics pipeline. Instead of producing pixels, they operate on arbitrary data in parallel, making them useful for tasks such as physics simulation, or data processing [6].
- **Buffers and Attributes**: Data such as vertex positions and colours are stored in GPU buffers. The pipeline defines how this data is interpreted and passed to the vertex shader as attributes.
- **Bind Groups**: These act as bridges between shaders and the external resources they require, such as uniform buffers, textures, or samplers. Each bind group corresponds to a slot in the shader where resources are bound.
- **Uniforms and Texture Sampling**: Uniforms (like `iResolution`, `iTime`, and `iMouse` discussed in Section 3.4.1) provide dynamic information to shaders. Textures and samplers allow for image-based lookups and visual complexity.
- **Command Buffers and Encoders**: Once resources and state are set, you record commands (like draw calls) into a command buffer via an encoder. This buffer is submitted to the GPU for execution.

[8, 20]

3.2 WGSL

WGSL, the shader language used for WebGPU projects, is designed to be more explicit, safe, and portable than GLSL, aligning more closely with modern low-level graphics APIs such as Vulkan and Metal [6, 21]. These differences have direct implications for SplitShade: they require abstraction layers to keep the user experience simple while accommodating WGSL's stricter syntax.

Some key WGSL features relevant to SplitShade's design include:

- **Explicit shader entry points**: Functions must be explicitly marked using `@vertex`, `@fragment`, or `@compute`, making the shader stage unambiguous.
- **Strict typing and layout declarations**: WGSL requires explicit use of annotations like `@binding`, `@group`, and `@location` to associate inputs, outputs, and resources with specific slots. For example, even reading a texture requires defining a sampler and declaring its binding group.
- **No implicit uniforms**: WGSL does not provide built-in global variables like `gl_FragCoord`. Instead, all uniforms must be passed manually via uniform buffers and referenced using structured bindings.

[6, 22, 23]

These features shape the design of shader playgrounds such as SplitShade, particularly in areas such as boilerplate code injection, uniform handling, and error reporting. Their implementation is discussed later in Section 4.

3.3 Shaders

This section introduces the two main types of shaders used in the graphics pipeline: vertex shaders (Section 3.3.1) and fragment shaders (Section 3.3.2). Their roles are best understood in the context of the render pipeline (Figure 4). A render pipeline begins with vertex data and progresses through several stages: the vertex shader transforms geometry, which is then assembled into primitives. These are rasterized into fragments, and the fragment shader computes the final pixel colours that appear on screen. Each stage builds on the results of the one before it [24, 25].

In WebGPU, this pipeline is explicitly defined, and even minimal shaders must conform to this structure. The interaction between vertex and fragment shaders therefore forms the foundation of SplitShade's design.

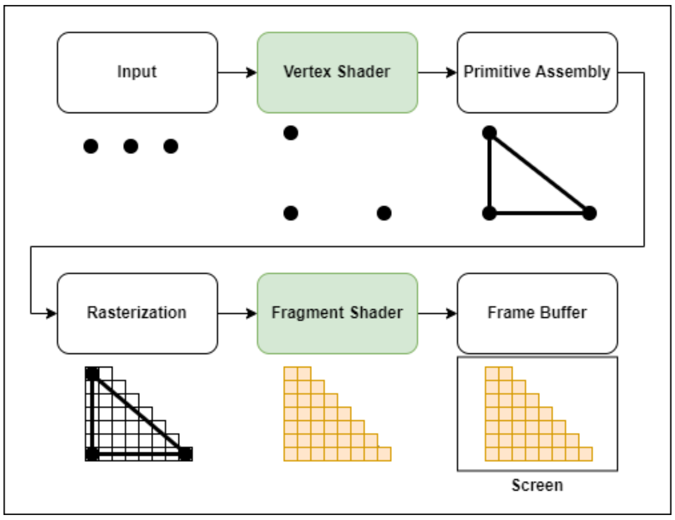


Figure 4: The Rendering Pipeline [25]

3.3.1 Vertex shaders

Vertex shaders are the first programmable stage in the traditional graphics pipeline. Their primary role is to transform input geometry into clip space coordinates before rasterization. For each vertex, the shader computes its clip space position and can pass additional attributes, such as colour or texture coordinates, to the next stage. These outputs are then interpolated across the geometry's surface and made available to the fragment shader (Section 3.3.2) [24, 26].

In WebGPU, every render pipeline requires both a vertex shader and a fragment shader (except in compute-only workflows, which follow a separate pipeline entirely) [8, 20]. Even for rendering a full-screen effect with no complex geometry, a minimal vertex shader is still required to initiate the pipeline.

In tools like **ShaderToy**, which only expose a fragment shader interface to the user [13], a hidden vertex shader is still required to initialize the pipeline.

This model relies on a technique called full screen fragment shading, where a simple geometric primitive, typically a triangle (Figure 5) or quad, is drawn to cover the entire viewport [27, 28]. The GPU then invokes the fragment shader once per pixel, enabling pixel-by-pixel procedural effects.

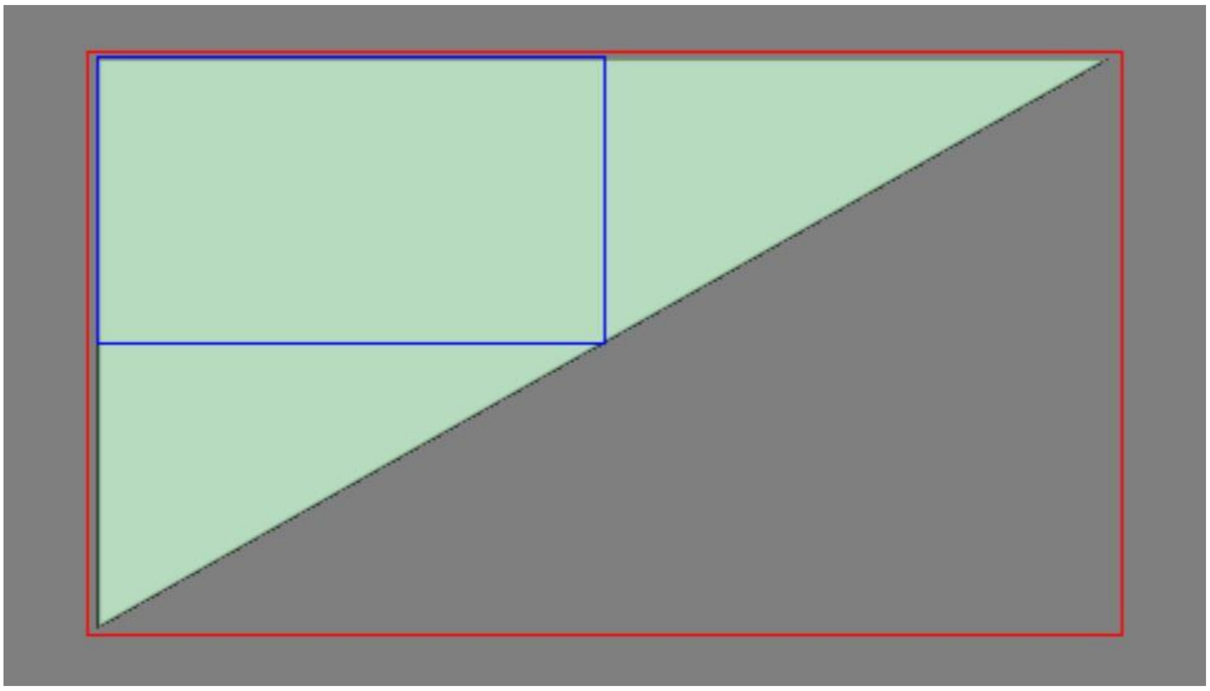


Figure 5: Full screen vertex shader rendering using a single triangle. The red outline shows the triangle's geometry bounds, which extend beyond the viewport (blue) to ensure full coverage [28]

3.3.2 Fragment shaders

The fragment shader is responsible for computing the final colour of each pixel, or *fragment*, generated by rasterizing geometry sent through the vertex shader. In most real-time graphics applications, this is where lighting calculations, texture lookups, and shading logic take place [29] .

In shader playgrounds like ShaderToy, fragment shaders are the central focus. Users write only a single fragment shader that runs once per screen pixel, with no need to provide geometry or define a rendering pipeline manually [13]. This setup allows the user's code to define the colour of each fragment (pixel) based on parameters like screen position, time, and mouse interaction, which will be discussed in Section 3.4.1.

3.4 External Input

GPU Shaders can use externally provided input. This section will discuss two such inputs – uniforms and textures.

3.4.1 Uniforms

Uniforms are global, read-only variables that remain constant for all shader invocations during a given render pass. They are commonly used to provide per-frame data such as time, resolution, mouse position, etc.

In WGSL, uniforms are passed via uniform buffers and are bound explicitly to the shader pipeline. Uniforms cannot be modified within the shader – they must be set externally, usually by the application before issuing a draw call [30].

To maintain simplicity in its shader playground environment, ShaderToy automatically provides a fixed set of common uniforms so that users do not need to set up them up manually.

```
uniform vec3 iResolution;
uniform float iTime;
uniform vec4 iMouse;
```

These values are used to adapt to the screen size, animate effects, and implement interactivity – all without changing the underlying shader logic.

3.4.2 Textures

In addition to scalar values passed through uniforms, shaders can also access external data in the form of **textures** or other **resources**. These are typically 2D images, videos, or intermediate render buffers that a shader can sample during execution [21].

Sampling a texture allows the shader to read colour or data values at specific UV coordinates, which can be used for image processing, feedback effects, or stylized rendering. In WGSL, this is done using functions like `textureSample` [31].

ShaderToy exposes four resource channels (`iChannel0` to `iChannel3`) that the user can fill with images, videos, or the output from previous rendering passes [13, 32]. This allows for complex, layered effects while keeping the shader interface simple. Figure 2 shows the resource channels on the bottom right of the interface.

4 Implementation

This section details the implementation of **SplitShade**, an interactive WebGPU shader playground designed to simplify WGSL development in the browser. It begins with an overview of the system's architecture (Section 4.1), followed by the design of the WebGPU pipeline (Section 4.2) and the real-time rendering loop that drives shader execution (Section 4.3). The section then examines the user interface and user experience (Section 4.4), along with the accompanying user documentation (Section 4.5). Key technical challenges and their solutions are discussed (Section 4.6), before outlining the testing strategy (Section 4.7), continuous integration and deployment (CI/CD) (Section 4.8), deployment approach (Section 4.9), and supporting development workflows (Sections 4.10 and 4.11). The Section concludes with an evaluation of the system in relation to its goals (Section 4.12).

4.1 System Overview & Architecture

This section outlines the overall structure of SplitShade, describing how its architecture, technology choices, and core components work together to enable real-time shader development in the browser.

4.1.1 High-level Architecture

The system implements a client-side architecture that eliminates server dependencies and enables immediate shader execution. The high-level data flow follows this pattern:

1. **Input Processing**: User WGSL code is parsed and validated in real-time
2. **Pipeline Assembly**: Dynamic WebGPU render pipeline creation based on shader type detection
3. **Resource Management**: Uniforms are injected automatically, textures bound, and meshes uploaded for vertex shaders when required.
4. **Render Loop**: Rendering once per display refresh (typically ~60 frames per second (FPS)) with live uniform updates (time, mouse, resolution)
5. **User Feedback**: Immediate error reporting and visual output display

This architecture prioritizes low latency between code changes and visual feedback, essential for interactive shader development.

4.1.2 Technology Stack

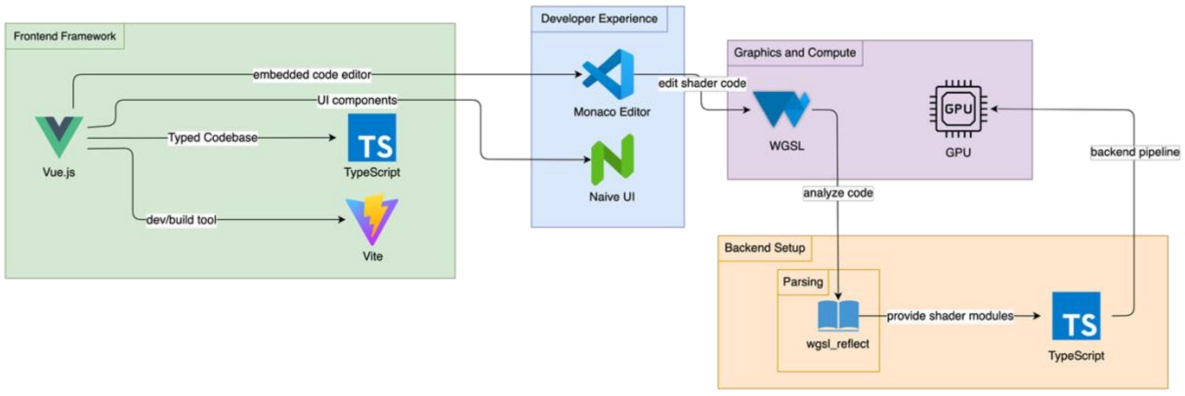


Figure 6: A diagram showing the tech stack

Figure 6 shows the Technology Stack used for the SplitShade project, which is also outlined below:

Frontend Framework:

- Vue.js **3.5.13**: Reactive component architecture with Composition API [33]
- TypeScript: Strong typing for WebGPU API interactions and error prevention
- Vite: Fast development server with hot module replacement [34]

Graphics & Compute:

- WebGPU: Native GPU access for high-performance rendering
- WGSL: Shading language for WebGPU

Developer Experience:

- Monaco Editor **0.52.2**: VS Code-based editor with syntax highlighting [35]
- @guolao/vue-monaco-editor: Vue integration for seamless Monaco editor embedding [36]
- Naive UI: Component library for consistent dark theme implementation [37]

Parsing & Reflection:

- `wgsl_reflect` **1.2.3**: WGSL introspection for entry point detection and shader analysis [38]

4.1.3 Core Components

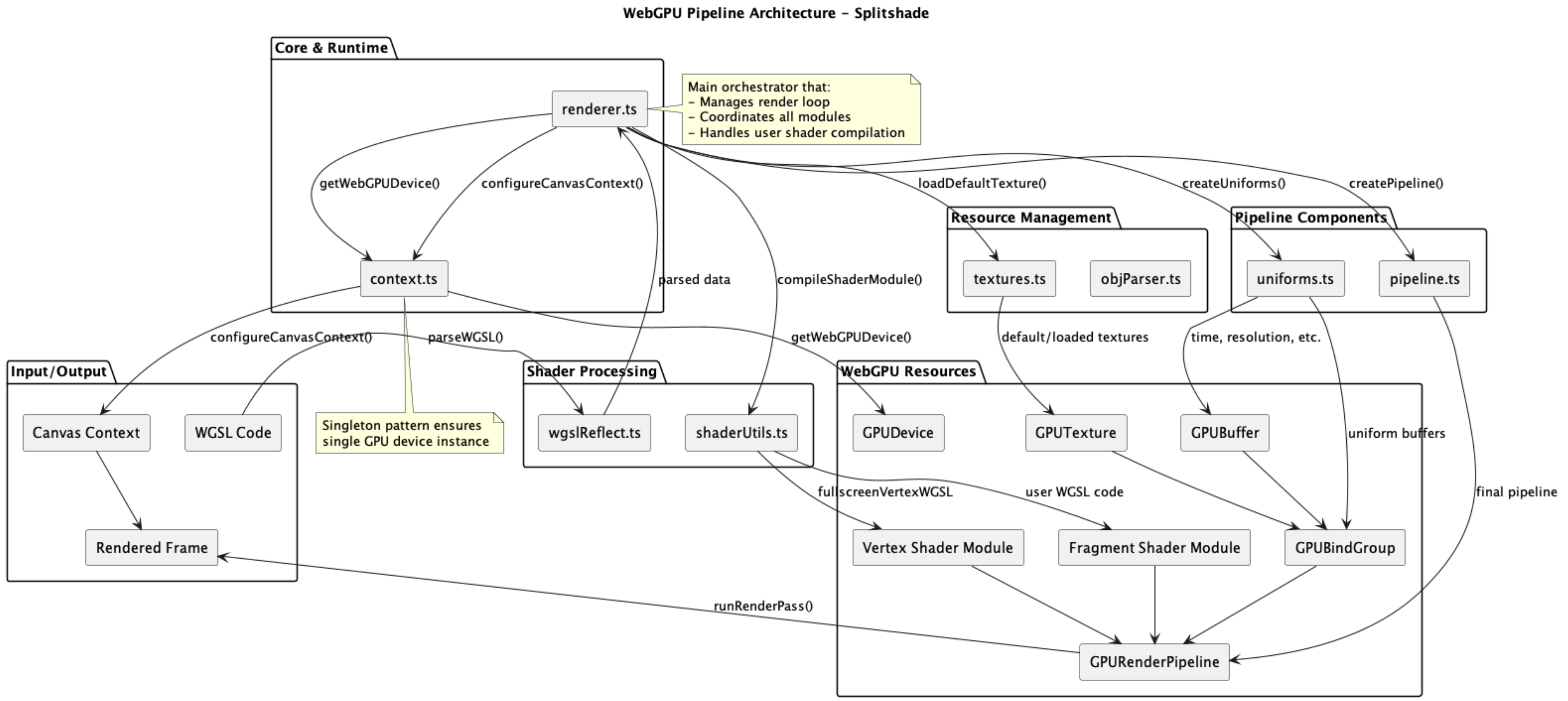


*Figure 7: **Component-level architecture of SplitShade's WebGPU shader workflow.** The diagram illustrates how system modules (e.g., context management, shader processing, resource management, and pipeline components) interact with WebGPU resources to transform user-provided WGSL code and inputs into a rendered frame.*

While Figure 6 outlined the broader tech stack and tooling used in SplitShade, the component diagram in Figure 7 dives deeper into the internal structure of SplitShade's WebGPU workflow orchestration.

Each inner box corresponds to a specific module in the system, while the larger group boxes indicate subsystems that organise related functionality. Arrows represent both data flow (i.e. buffers, textures, WGSL code) and control flow (i.e. function calls that trigger compilation or pipeline creation).

- **Core WebGPU Modules** (`renderer.ts`, `context.ts`) manage device setup and render loop control, acting as the central coordinator.
- **Pipeline Components** (`pipeline.ts`, `uniforms.ts`) handle render pipeline creation and uniform injection.
- **Resource Management** (`textures.ts`, `objParser.ts`) is responsible for loading textures and parsing mesh data.
- **Shader Processing** (`wgslReflect.ts`, `shaderUtils.ts`) supports shader parsing, reflection, and compilation.
- **WebGPU Resources** (`GPUDevice`, `GPUTexture`, `GPUBuffer`, `GPUBindGroup`) are the objects created by the pipeline and resource components, enabling the `GPURenderPipeline` to execute.
- **UI-driven Shader Modules** (Vertex and Fragment Shader Modules) are dynamically compiled via user-edited WGSL code and integrated into the render pipeline.

This modular architecture ensures clear separation of concerns – shader parsing, GPU resource management, and render pipeline configuration remain decoupled and reusable. Data generally flows in a forward direction, from user input through parsing and render pipeline setup to the GPU, while control signals (such as device access or error feedback) are routed back through the orchestrating modules.

4.2 WebGPU Render Pipeline Implementation

While SplitShade's architecture comprises six subsystems (Figure 7), this section focuses on three of them most directly involved in shader execution: Context Management (part of the Core WebGPU Modules in Figure 7), Shader Processing, and Resource Management. Together, these subsystems transform user WGSL code into executable GPU programs and enable real-time rendering in the browser.

4.2.1 Context Management

Context management forms the foundation of SplitShade's WebGPU workflow, ensuring that the GPU device and canvas surface are consistently initialised and accessible across the application.

4.2.1.1 Singleton pattern for GPU device management

The `context.ts` module implements a **singleton pattern** to ensure efficient GPU resource sharing across the application. This approach prevents multiple device instances and ensures consistent GPU state management across all across all system components that rely on the GPU device. The singleton also handles device loss scenarios and provides centralized error reporting for GPU-related failures. In addition, consolidating device access through a single instance simplifies testing (Section 4.7), as unit tests can rely on a predictable and reproducible GPU context rather than reinitialising hardware resources for each test case.

4.2.1.2 Canvas context configuration

Canvas context setup involves configuring the WebGPU surface for rendering output. Configuring the canvas aids in establishing the connection between the logical GPU device and the HTML canvas element.

In the codebase, the configure function queries the preferred canvas format to ensure optimal performance across different hardware configurations. It also assumes and sets up opaque alpha blending, keeping it simple for standard rendering workflows.

4.2.1.3 Device capability detection and fallbacks

The context management system includes comprehensive error handling for unsupported environments. Browser compatibility is checked at multiple levels:

1. WebGPU API Availability: Detection of `navigator.gpu` object
2. Adapter Acquisition: Hardware/software adapter availability
3. Device Creation: GPU device initialization success

Each failure point provides specific error messages to guide users toward compatible browsers or hardware configurations. This ensures graceful degradation rather than silent failures in unsupported environments.

4.2.2 Shader Compilation Process

Shader compilation begins with WGSL analysis using the `wgsl_reflect` library [38]. The parser identifies entry points and determines the shader type – vertex, fragment, or compute. This introspection enables dynamic pipeline setup based on the detected configuration.

User-provided WGSL code is compiled into shader modules with error handling and UI feedback. A standard header containing uniform declarations (`iTime`, `iMouse`, `iResolution`) is automatically prepended to ensure compatibility with the runtime environment (Section 4.3).

SplitShade currently supports both **fragment-only** and **vertex-fragment** workflows.

4.2.2.1 Fragment-only Shader Support

For fragment-only workflows (similar to ShaderToy), SplitShade uses a full screen triangle approach to eliminate the need for vertex buffers (see Section 3.3.1 and Figure 5). This allows users to see immediate visual results while avoiding the complexity of vertex attributes or coordinate systems.

4.2.2.2 Vertex-Fragment Shader Support

When both vertex and fragment entry points are detected, the system configures a full render pipeline without injecting a full screen shader. This allows users to progress from simple fragment shaders to more complex vertex-driven rendering.

4.2.2.3 Error handling and user feedback

Compilation also includes robust error reporting at multiple stages:

- **Parsing**: Syntax errors with line numbers
- **Compilation**: Shader module creation failures
- **Runtime**: Pipeline issues, including resource conflicts, etc. caught at execution.

All errors are surfaced in the console panel to aid iterative shader development (see Section 4.4.4).

4.2.3 Resource Management

This section describes how SplitShade manages runtime resources – uniforms, textures, and mesh data – to support real-time shader execution. Each resource type is initialized, uploaded, and bound using explicit layouts, enabling reliable access from user-authored shaders without requiring manual setup.

To keep the authoring model predictable and avoid runtime binding errors, the system constrains shader inputs to a fixed set of supported resources: a built-in uniform block, up to four texture–sampler pairs, and mesh vertex buffers. This fixed schema ensures that the bind group layout always matches the shader's declared `@group/@binding` locations, avoiding the need for layout regeneration and re-binding when shader code changes.

It also guarantees that shaders created in one session will run in another without additional host-side configuration. By limiting allocations to resources with corresponding UI controls, the design eliminates unused bindings that would otherwise incur GPU/CPU overhead without adding functionality. The result is a stable, portable, and iteration-friendly resource model, similar in philosophy to ShaderToy but implemented within WebGPU's stricter binding requirements.

4.2.3.1 Uniform buffer creation (`uniforms.ts`)

SplitShade, drawing inspiration from ShaderToy-styled uniforms [32], provides a standardized set of uniform inputs. The system automatically binds the following resources to **bind group 0**, making them accessible across all user-provided shader modules without explicit declaration:

- **`iResolution`**: Canvas dimensions as `vec3<f32>`
- **`iTime`**: Elapsed time since shader start as `f32`
- **`iMouse`**: Mouse coordinates and click state as `vec4<f32>`
- **Texture Channels**: Up to 4 texture samplers (`iChannel0-3`, `iChannelSampler0-3`) each bound as a texture-sampler pair (Section 4.2.3.2)

The built-in uniforms (`iResolution`, `iTime`, `iMouse`) are updated per frame and passed into both vertex and fragment stages as needed. While `iResolution` remains static for most sessions, `iTime` and `iMouse` are updated every frame – yet all are bound in the same group for simplicity and uniform accessibility across shaders.

Although some WebGPU guidelines suggest placing more static data (like resolution) in lower-indexed bind groups and rapidly changing values in higher groups [39], in practice this is most critical for scenarios with multiple draw calls per frame where bind group changes incur performance costs. Since SplitShade performs a single draw call per frame and binds only once (Section 4.3.1), placing all uniforms into `@group(0)` offers a clean and practical solution without measurable performance penalty.

Furthermore, SplitShade deliberately avoids using `layout: "auto"` when generating pipeline layouts. While auto layout can simplify development, it introduces potential pitfalls: it produces layouts tightly coupled to the specific shader module and may omit bindings that are not statically referenced at pipeline creation [39]. This would break compatibility with shared runtime resources like `iTime` or `iMouse`, especially in interactive use cases. For instance, if a user did not use any of the pre-established uniforms in their input code, the bind group would throw an error as the layout would not be expecting it. By explicitly defining the bind group layout, SplitShade guarantees compatibility and avoids unnecessary duplication when switching between shaders that use the same uniform structure.

Looking forward, when SplitShade expands to support compute shaders (discussed in Section 5.3.1), multiple pipelines (i.e. render and compute) will be required. These may share a common set of uniforms, and explicit layout definitions will ensure consistency and efficiency across all pipeline types without redundancy or risk of mismatch.

4.2.3.2 Texture loading and binding (`textures.ts`)

SplitShade supports up to four texture inputs (`iChannel0 - iChannel3`), just as ShaderToy allows [13, 32] . These textures are passed into the shader as sampled textures via explicitly bound texture and sampler pairs.

Texture setup on the backend proceeds in four stages:

1. **Asset Loading**: Image assets are fetched via `HTMLImageElement` and decoded into `ImageBitmap` objects for efficient GPU transfer.
2. **GPU Upload:** Bitmaps are uploaded to device-local GPU textures using `copyExternalImageToTexture()`. Textures use the `rgba8unorm-srgb` format to support standard 8-bit colour with sRGB correction for consistent rendering across devices.
3. **Sampler Configuration**: Textures are sampled with `linear` filtering (`minFilter`, `magFilter`) for smooth interpolation, and `repeat` address modes to support UV wrapping – a common requirement for procedural and tiled effects.
4. **Bind Group Integration**: Each texture-sampler pair is injected into the predefined layout at fixed bindings, maintaining a consistent mapping from `iChannelN` to GPU resource bindings.

All texture bindings are placed in bind group 0 alongside the standard uniforms discussed in Section 4.2.3.1. Each active texture is represented by a `GPUTextureView` and a `GPUSampler`. These are dynamically assigned to consecutive binding slots, starting at index 3 (following `iResolution`, `iTime` and `iMouse`). For each texture channel, two entries are added to the bind group layout:

- A `texture` entry (`@binding N`) specifying the texture view
- A corresponding `sampler` entry (`@binding N+1`) used for filtering

To balance efficiency with predictable behaviour, texture channels are allocated conditionally. If the user shader does not reference any textures, none are bound in the pipeline. When at least one texture is referenced, all four channels (`iChannel0` - `iChannel3`) are allocated and initialised with default textures. This ensures that any shader referencing a texture channel compiles and runs correctly, while avoiding the overhead of unused bindings in cases where no textures are needed. The approach also preserves a consistent binding order, removing the need for users to manage slot indexing manually (also discussed in Section 4.6.5.3).

The explicit layout ensures that all user shaders can access texture channels through the identifiers without having to manage resource declarations themselves. By centralizing the texture setup on the backend, SplitShade guarantees predictable binding order and enables seamless integration with both user-specified and auto-injected shader code.

4.2.3.3 Mesh loading

In addition to texture inputs, SplitShade allows users to load 3D mesh assets for use with custom vertex shader programs. Mesh loading proceeds in the following stages:

1. **Asset Loading**: SplitShade supports the `.obj` format for uploaded files. The OBJ parser on the backend can extract vertex positions and optional attributes (e.g., colours, normals). Currently, SplitShade focuses on extracting only vertex positions and assigns white as a default colour for simplicity and compatibility.
2. **Interleaved Vertex Buffer Creation**: The parsed vertex data is packed into a single interleaved `Float32Array`, with each vertex represented by a tightly packed sequence (i.e. `[position.x, position.y, position.z, color.r, color.g, color.b]`). This interleaved format improves GPU cache coherence and memory locality by storing all attributes of a single vertex together, allowing efficient fetches during vertex processing [40, 41].
3. **GPU Upload**: The resulting vertex array is uploaded to a `VERTEX GPUBuffer`. This buffer is created once during mesh upload and reused during each render pass.
4. **Vertex Buffer Binding**: When rendering with a vertex shader, the pipeline defines a `vertexState` that matches the layout of the interleaved buffer. Each attribute (e.g., position and colour) is assigned a specific `shaderLocation` and byte offset, enabling the GPU to correctly fetch and decode each vertex's data.

Following mesh upload, **SplitShade also provides users with auto-generated starter code** to simplify shader development. This includes a minimal vertex and fragment shader designed to work with the interleaved vertex format, transforming the mesh into clip space and applying vertex colour. Users can customise this code further to suit their rendering needs. (See Section 4.4.2.2 for more.)

4.3 Real-time Rendering Loop

The rendering loop represents the core of SplitShade's real-time interactivity, executing once per display refresh (typically ~60 FPS) to provide immediate visual feedback from the currently compiled shader. This section details the frame-based architecture that synchronizes uniform updates, processes user input, and manages GPU command submission for consistent performance.

4.3.1 Frame Loop Architecture

The rendering process in SplitShade is orchestrated by a continuous frame loop driven by the browser's `requestAnimationFrame` mechanism. Each frame follows a structured sequence that ensures all GPU resources, shader inputs, and user interactions remain in sync. The architecture is event-driven: input events are captured asynchronously, while frame updates are handled on a per-frame basis.

Figure 8 presents the sequence diagram of interactions between the WebGPU context, render pipeline, uniform buffers, and the canvas during a single frame. The diagram illustrates the following key responsibilities within each iteration:

1. **Frame Scheduling**: The `requestAnimationFrame` callback signals the start of a new frame.
2. **Uniform Synchronisation**: Time, resolution, and input states are updated in GPU-accessible buffers.
3. **Render Pass Configuration**: The WebGPU context prepares the swap chain's current texture and defines the render pass parameters.
4. **Draw Call Execution**: The pipeline and associated resources (shaders, vertex buffers, bind groups) are bound, and draw commands are recorded.
5. **Command Submission**: The encoded commands are submitted to the GPU for execution, after which the loop awaits the next frame callback.

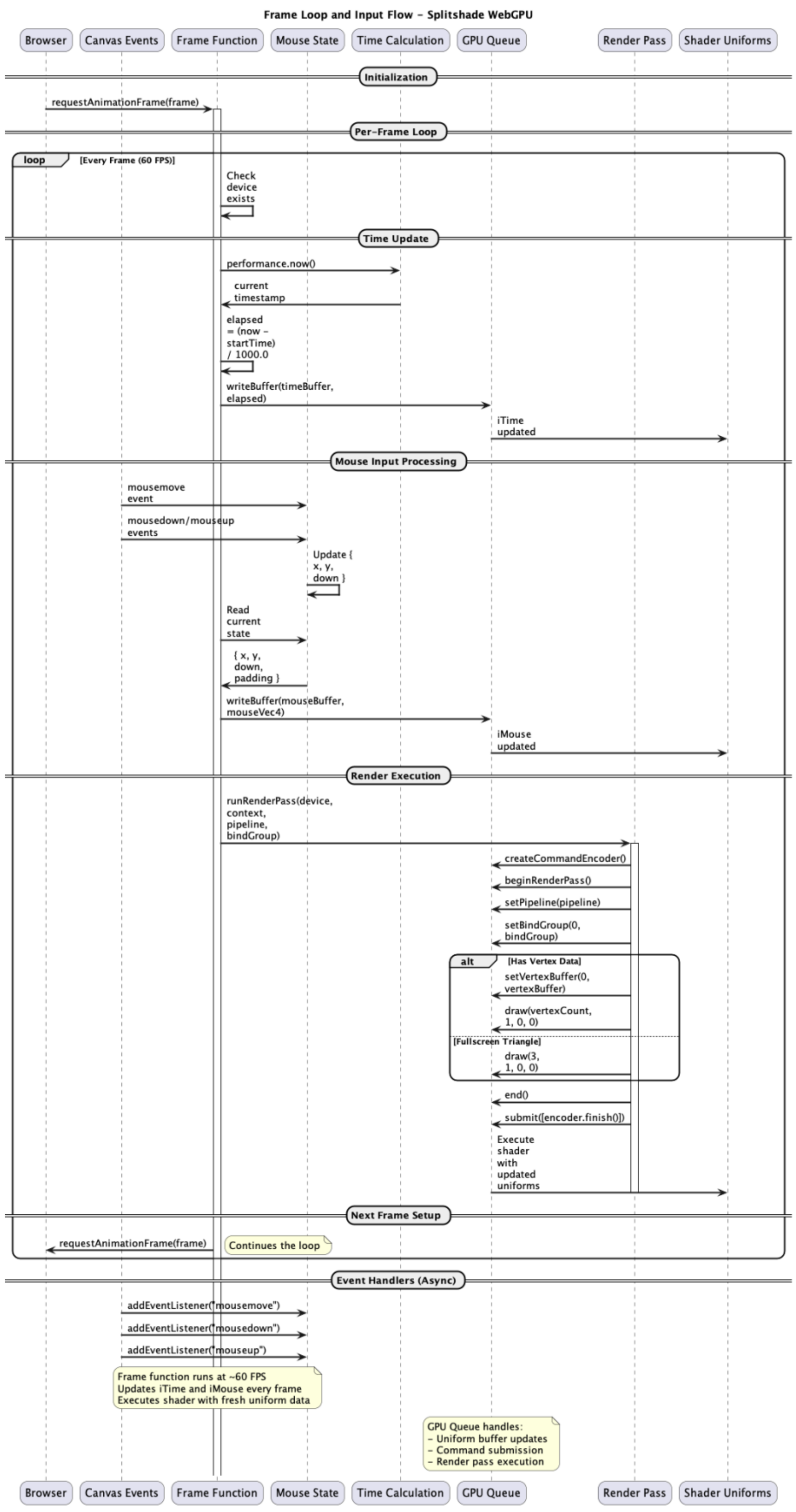


Figure 8: Frame Loop and Input Flow showing the interaction between browser animation frames, uniform updates, and GPU command submission

4.3.2 Data & Uniform Updates

During each frame, application-level state is synchronised with the GPU via uniform buffers. Core examples include:

- `iTime`: Updated using the elapsed time since the application started, allowing time-dependent shader effects.
- `iResolution`: Updated from the current canvas dimensions to ensure fragment coordinates are scaled correctly.
- `iMouse`: Populated from the latest user input, enabling shaders to respond to mouse interaction.

Uniform data is written to mapped GPU buffers (for initial vertex data) or via `queue.writeBuffer` operations (for dynamic uniforms). Most uniform updates occur before draw calls, though some time buffer updates happen both before and after rendering for the next frame.

4.3.3 Input Handling

Mouse movement and button states are tracked by event listeners attached to the canvas. These listeners normalise cursor positions to a consistent coordinate space before writing them into the `iMouse` uniform buffer.

The update path is strictly event-driven: mouse events update stored input state, which is then read and written to GPU buffers in the uniform update stage of the frame loop. This ensures that rendering remains decoupled from input polling, while still delivering responsive, frame-synchronised interaction.

4.3.4 GPU Command Submission

Once all uniforms are updated and draw calls recorded, the `GPUCommandEncoder` finalises the render pass and submits the resulting command buffer to the GPU via `device.queue.submit`. This explicit submission step ensures all queued rendering operations are executed before the next frame begins.

The WebGPU context manages the swap chain presentation, displaying the rendered frame to the canvas and preparing the next drawable surface for the subsequent iteration.

4.3.5 Performance Considerations

The loop is designed to sustain real-time rendering performance at or near the display's refresh rate (typically ~60 FPS on most systems). Key optimisations include:

- **Efficient Buffer Updates**: Using persistent GPU buffers and only rewriting modified data each frame to reduce bandwidth usage.
- **Minimising State Changes**: Grouping draw calls to avoid unnecessary pipeline or bind group switches.
- **Avoiding Overdraw**: Ensuring shaders only execute for necessary fragments, especially in full-screen effects.
- **Browser-GPU Synchronisation**: Leveraging `requestAnimationFrame` to avoid unnecessary frames and reduce CPU-GPU desynchronisation.

Potential bottlenecks include overly complex fragment shaders authored by users and the use of a generic resource binding layout. The latter simplifies the development model by providing a consistent set of uniforms and texture slots, but may introduce overhead when shaders do not require all of these resources. While these trade-offs favour usability over maximum efficiency, profiling with browser developer tools could inform future optimisation (Section 5.3.8).

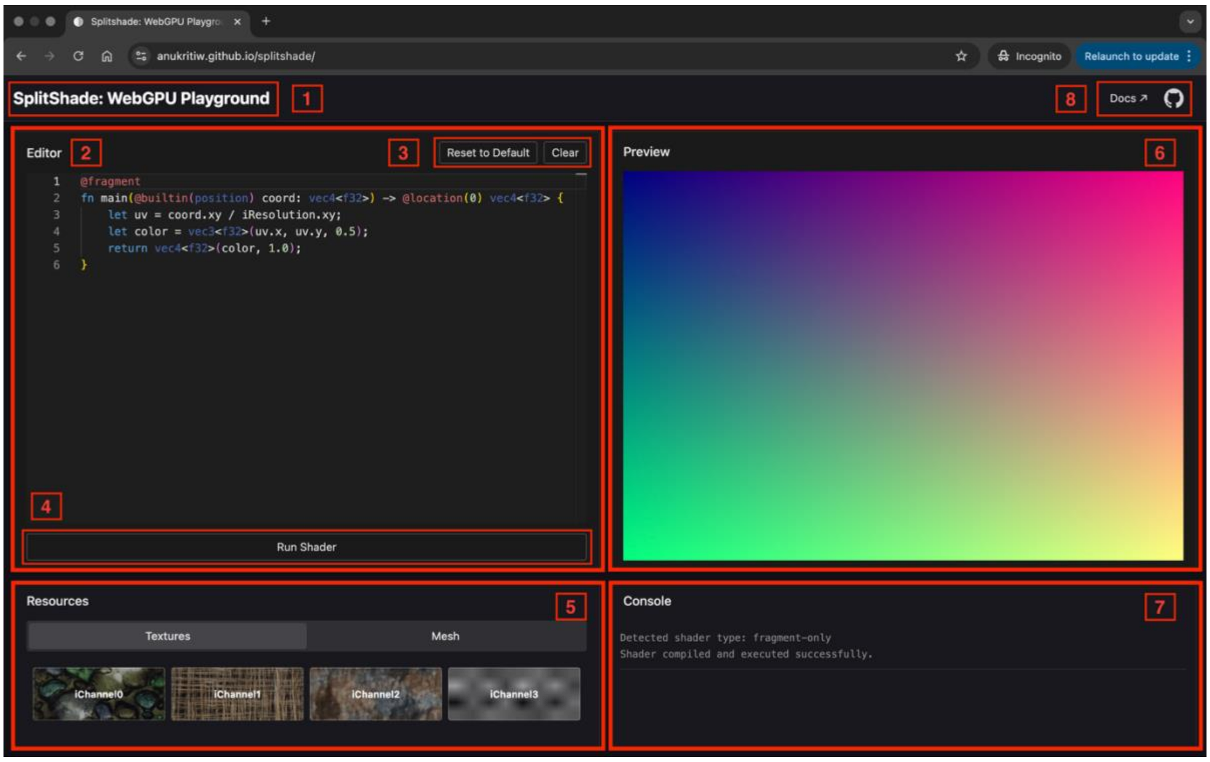
4.4 User Interface & User Experience (UI/UX)

The application is divided into four panes – a code editor, a resources panel, a preview panel, and a console panel (Figure 9). These panes are arranged side by side, allowing developers to edit code, see changes, and view diagnostic output in parallel without context switching.

4.4.1 Code Editor

The **Code Editor panel** ((2) in Figure 9) integrates the Monaco Editor [35], which is the same editor that powers Visual Studio Code [42], a popular integrated development environment (IDE) among developers [43]. As such, not only is it a familiar environment for most developers, it also automatically handles code editor necessities such as monospaced text, WGSL syntax highlighting and structured indentation, which allow for improved code readability.

When SplitShade first loads, the Code Editor is populated with a short, minimal shader example (as seen in (2) in Figure 9). This provides users with an immediate, working baseline from which to experiment – an approach similar to ShaderToy's default shader template [13], and consistent with the "recognition over recall" principle in interface design [44].



```
@fragment
fn main(@builtin(position) coord: vec4<f32>) -> @location(0) vec4<f32> {
    let uv = coord.xy / iResolution.xy;
    let color = vec3<f32>(uv.x, uv.y, 0.5);
    return vec4<f32>(color, 1.0);
}
```

```
Detected shader type: fragment-only
Shader compiled and executed successfully.
```

Figure 9: A marked up image of the Splishade UI showing four panels – (2) Code Editor, (5) Resources Panel, (6) Preview Panel, and (7) Console Panel. (1) is a clickable header to the page; (3) shows buttons to reset or clear the editor; (4) shows the 'Run Shader' button that runs the code; (8) shows hyperlinks to the Documentation Page and the Github repository.

To improve continuity between sessions, editor contents are **locally persisted** in the browser, allowing users to resume exactly where they left off without manual saving. Recognising the potential need to revert to a known state, a "Reset to Default" button ((3) in Figure 9) was added, restoring the original template in a single action. Additionally, a "Clear" button ((3) in Figure 9) enables starting from an empty state, accommodating users who prefer to build shaders from scratch (see PR #83). These buttons align with the "User Control and Freedom" heuristic by allowing quick recovery from undesirable changes [44]. Both features support error recovery and reduce the cognitive load associated with manual clearing or reconstruction, aligning with established usability heuristics for user control and freedom.

From an implementation perspective, the persistence system is designed for both usability and technical robustness. Editor state is stored in `localStorage` under a versioned key (`splitshade:editorCode:v1`), enabling forward compatibility with future data structure changes without complex migration logic (see PR #83). Event listeners attached to the Monaco editor are properly cleaned up using Vue's `onBeforeUnmount` lifecycle hook, preventing memory leaks that could degrade performance during extended sessions. The mechanism also accounts for edge cases, such as missing `localStorage` support or browser-imposed storage restrictions, ensuring the application remains functional even if persistence is unavailable. This approach maintains a consistent experience across different browser configurations while safeguarding performance over long-term use.

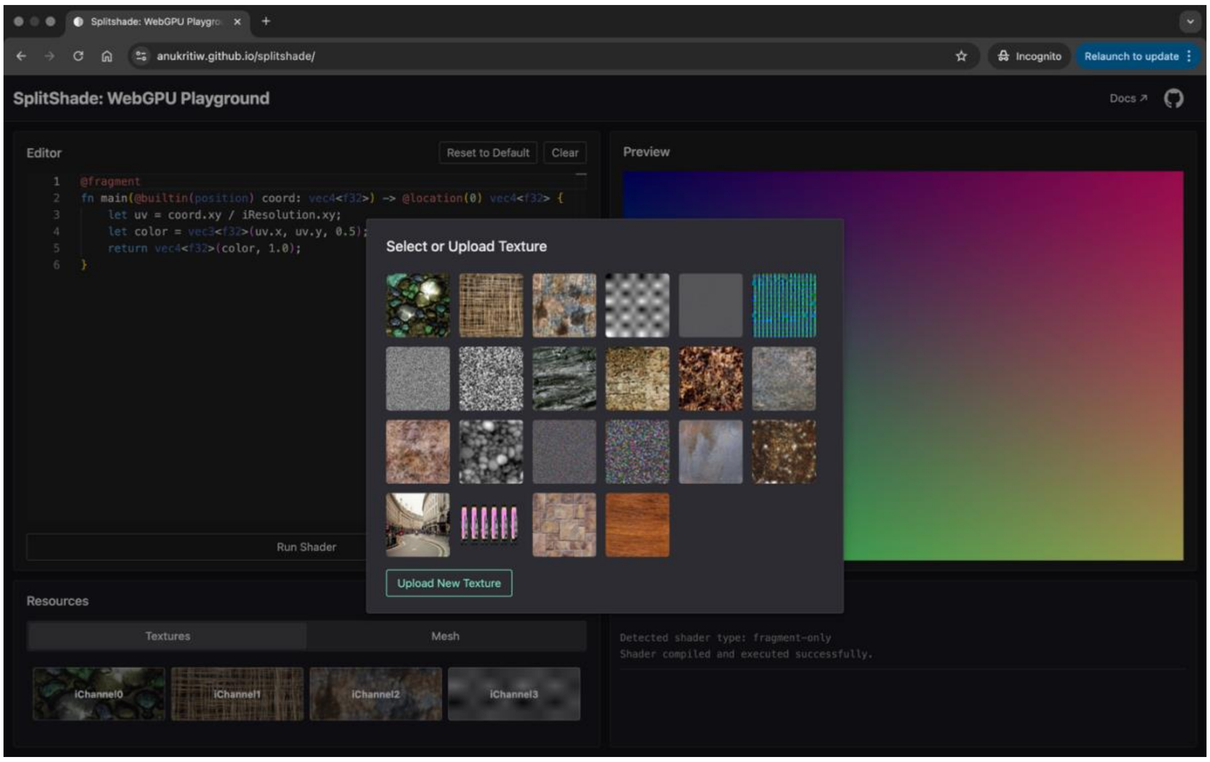
4.4.2 Resource Panel

The Resources Panel ((5) in Figure 9) provides a central location for managing all external shader inputs without leaving the main workspace. From this panel, users can assign or change texture channels (`iChannel0 - iChannel3`) and upload or select preset 3D meshes for use in custom vertex shader workflows.

4.4.2.1 Texture Management

Each texture channel is displayed with a thumbnail preview, showing the currently assigned image (Figure 9). Clicking the thumbnail opens a modal (Figure 10) where users can select from preset textures or upload their own image files.

The interface maintains the ShaderToy-style mapping between texture slots and `iChannelN` identifiers. Default placeholder textures are shown for unassigned channels, so shaders referencing a channel will still compile and run without user action.



```
@fragment
fn main(@builtin(position) coord: vec4<f32>) -> @location(0) vec4<f32> {
    let uv = coord.xy / iResolution.xy;
    let color = vec3<f32>(uv.x, uv.y, 0.5);
    return vec4<f32>(color, 1.0);
}
```

```
Detected shader type: fragment-only
Shader compiled and executed successfully.
```

Figure 10: When an iChannelN thumbnail is clicked, a modal opens

4.4.2.2 Mesh Management

The Resources Panel ((5) in Figure 9) can be toggled between Texture and Mesh (Figure 11).

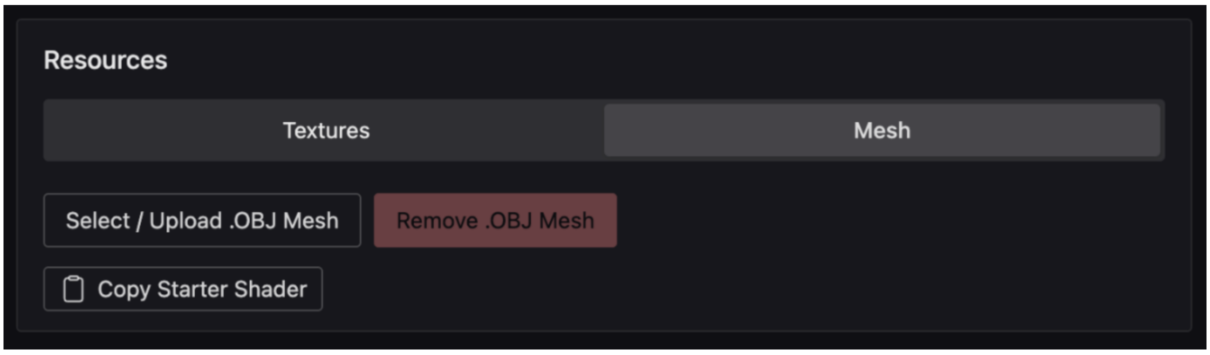
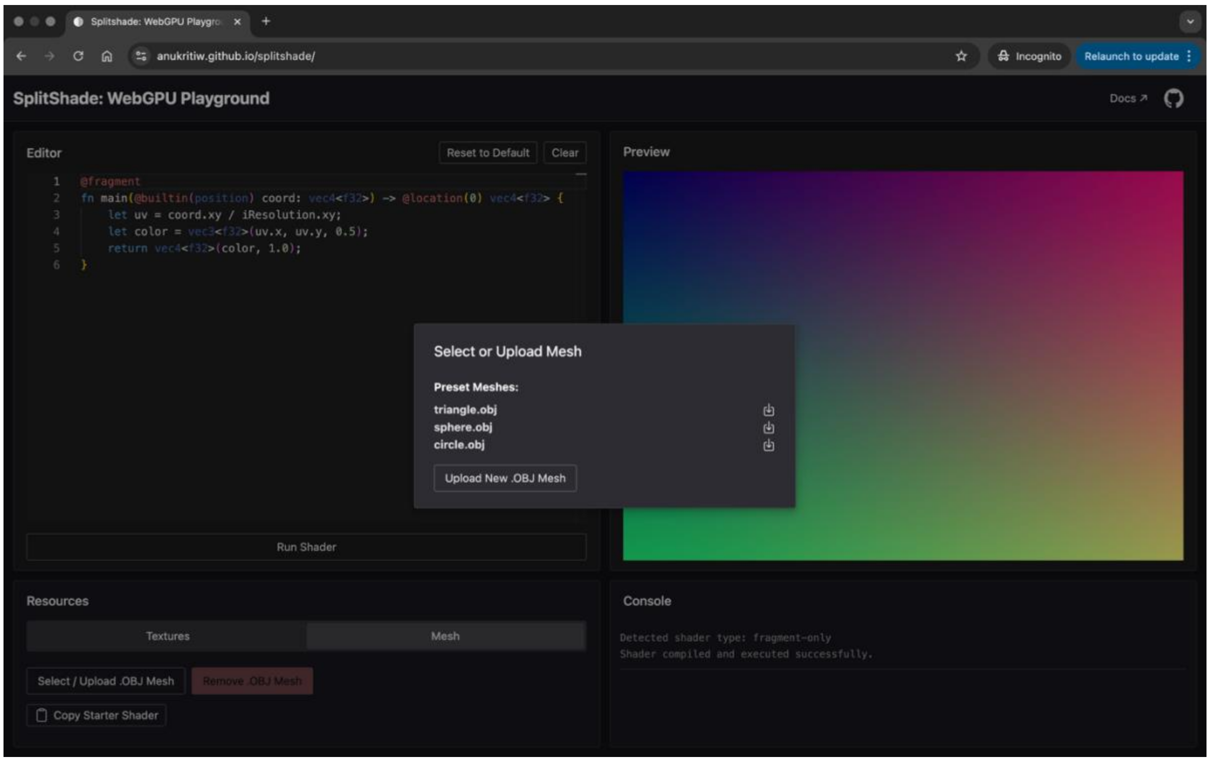


Figure 11: Resources Panel toggled to 'Mesh'

There are three buttons on the Mesh side of this panel:

1. "**Select / Upload .OBJ Mesh**": Opens a modal (Figure 12) with some preset `.obj` file options and the option to upload one's own `.obj` mesh file. `OBJ` files are parsed using the open-source `obj-file-parser` library [45], which extracts vertex positions and attributes. SplitShade then restructures this data into an interleaved vertex buffer (position and colour), ensuring compatibility with the starter shader code and backend conventions (also discussed in Section 4.2.3.3).

2. "**Remove .OBJ Mesh**": Only enabled when a `.obj` file has been set. While the performance impact of an unused mesh in SplitShade's single-draw pipeline is minimal, removing it prevents unnecessary vertex buffer bindings and keeps the render state clean for future shader runs. This feature also aligns with the "User Control and Freedom" heuristic [44], ensuring users can easily undo unwanted states and restore a clean rendering context.
3. "**Copy Starter Shader**": Copies a starter shader code to the user's clipboard. The copied code is a minimal, pre-configured WGSL shader pair that is guaranteed to compile and render with the currently loaded mesh. It is generated to match SplitShade's backend assumptions about mesh data layout – specifically, an interleaved vertex buffer containing position and colour attributes bound at `@location(0)` and `@location(1)` respectively. By aligning with these conventions, the code ensures that vertex data is correctly interpreted by the GPU. Having this button on this panel also reduces onboarding friction for new users, supports rapid prototyping, and prevents common compatibility errors when integrating uploaded geometry into custom shaders.



```
@fragment
fn main(@builtin(position) coord: vec4<f32>) -> @location(0) vec4<f32> {
    let uv = coord.xy / iResolution.xy;
    let color = vec3<f32>(uv.x, uv.y, 0.5);
    return vec4<f32>(color, 1.0);
}
```

```
Detected shader type: fragment-only
Shader compiled and executed successfully.
```

Figure 12: When "Select / Upload .OBJ Mesh" button is clicked, a modal opens

On the "**Select or Upload Mesh**" modal (Figure 12), users can also download any of the preset `.obj` files. This allows them the flexibility to examine the `.obj` files, or make any changes to the presets and re-upload if they wish to.

4.4.3 Preview Panel

The **Preview Panel** ((6) in Figure 9) renders the output of the active shader pipeline in real time after each "Run Shader" action. It reflects the current state of the editor, resource assignments, and any interactive inputs such as mouse movement. The panel is designed to fill the available grid space completely, maximising the rendered output area within the constraints of the responsive grid layout.

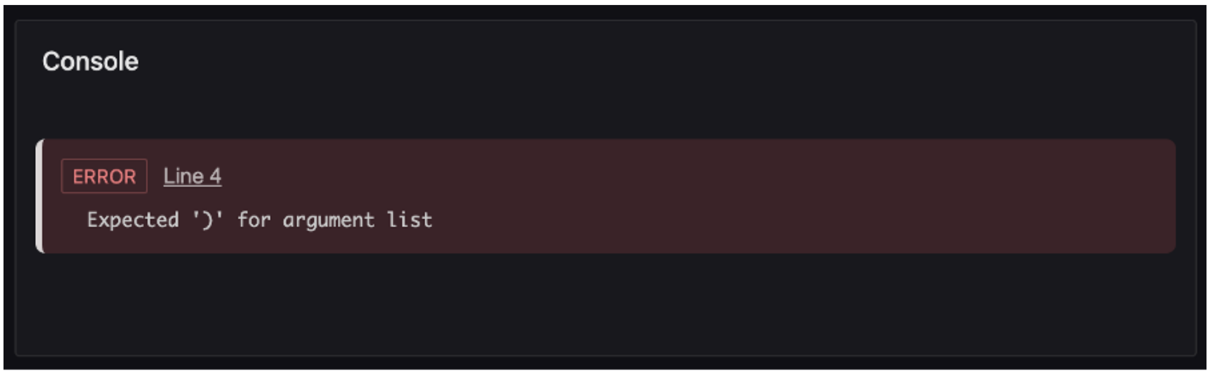
4.4.4 Console Panel

The **Console Panel** ((7) in Figure 9) provides a structured, visually distinct feedback stream during shader development. Output is separated into two categories:

- **System messages** generated by SplitShade (e.g., parser output, status updates), shown above a dividing line.
- **WebGPU feedback**, including parser errors from `wgsl_reflect` [38], compilation diagnostics from the WebGPU API, and runtime validation errors, shown below the divider.

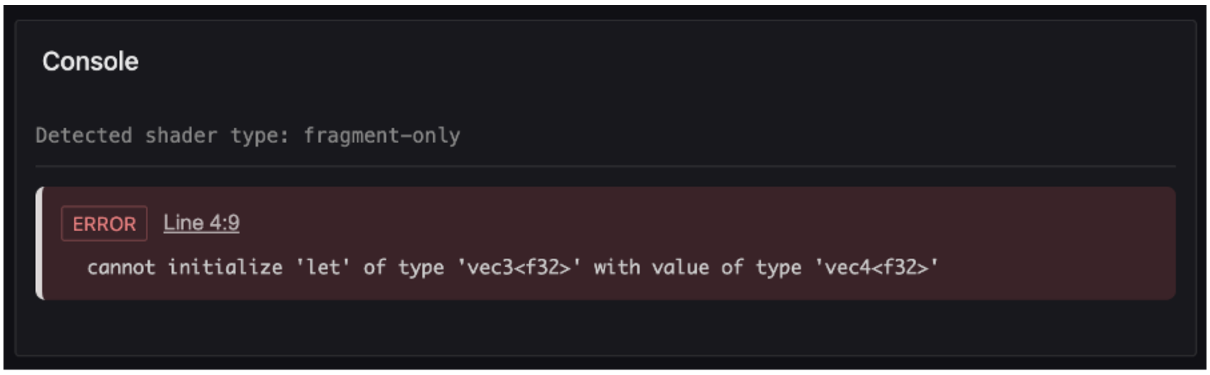
All WebGPU-related errors are styled in a red, rounded rectangle for quick visual scanning (Figure 13 to Figure 15) following the "Help Users Recognize, Diagnose, and Recover from Errors" heuristic for UI Design [44]. When an error includes a `line:column` reference, the system consistently extracts it, regardless of origin, and renders it as a clickable link. Clicking the link emits a signal and triggers a function that moves the Monaco editor cursor to the exact location of the error, allowing immediate inspection and correction. Longer error messages activate a scroll within the console panel (Figure 15).

Figure 13 shows a parser error caught before execution, preventing invalid WGSL from running. Figure 14 illustrates compiler diagnostics flagged during WebGPU pipeline creation. Figure 15 demonstrates runtime errors captured during execution, such as mismatched bind group layouts.



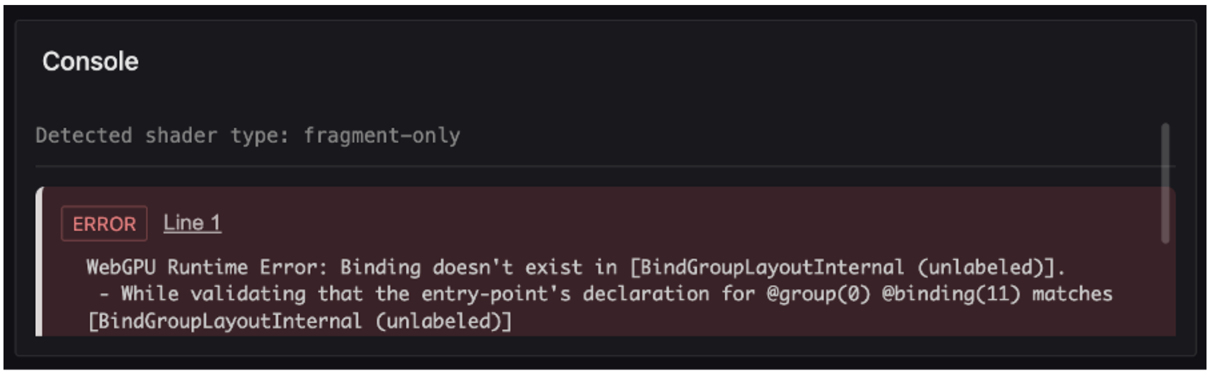
```
Expected ')' for argument list
```

Figure 13: Console Panel with Parser errors



```
Detected shader type: fragment-only
cannot initialize 'let' of type 'vec3<f32>' with value of type 'vec4<f32>'
```

Figure 14: Console panel with WebGPU Compiler errors



```
Detected shader type: fragment-only
WebGPU Runtime Error: Binding doesn't exist in [BindGroupLayoutInternal (unlabeled)].
 - While validating that the entry-point's declaration for @group(0) @binding(11) matches
[BindGroupLayoutInternal (unlabeled)]
```

Figure 15: Console panel with WebGPU Runtime errors

4.4.4.1 Error Reporting Architecture

Internally, error reporting is implemented as a structured pipeline rather than relying on unstructured console output parsing (see PR #82). Earlier iterations used a text-based parser to heuristically extract line numbers and error types (see PR #66), but this was brittle: compiler output formats could vary, and message phrasing changes could break the parser.

The current system leverages the `GPUCompilationInfo` API [46] as the primary source of compiler diagnostics. This API provides machine-readable data for each message – including text, severity, line and column positions, and byte offsets – ensuring:

- **UI Integration**: While the current implementation primarily surfaces errors, the architecture is designed to also handle warnings and informational messages if and when they become available, enabling visual distinction and clickable navigation in the editor.
- **Future Compatibility**: The API follows the WebGPU specification, making it less susceptible to breakage from browser changes.
- **Extensibility**: The architecture supports aggregating vertex and fragment shader diagnostics into a unified error list.
- **Fallback Support**: For browsers lacking `GPUCompilationInfo`, the original regex-based parser is retained, ensuring cross-browser operability during ongoing WebGPU adoption.

Line numbers are adjusted according to the error source: parser errors use the editor's exact positions, while compiler and runtime errors account for injected backend headers (e.g., built-in uniforms) by subtracting their offsets to maintain accurate mapping.

This architecture supports three primary error modes:

1. **Parser errors** (Figure 13) - detected before execution.
2. **Compiler errors** (Figure 14) - surfaced during pipeline creation.
3. **Runtime errors** (Figure 15) - captured during execution.

The result is a predictable, iteration-friendly workflow where all feedback is presented in a unified, interactive format without requiring the user to leave the SplitShade interface.

4.5 User Documentation

SplitShade includes a dedicated documentation site (Figure 16) to guide users through setup, navigation, and feature usage. The documentation is designed for quick reference rather than exhaustive API coverage, reflecting the practical workflows supported by the tool. It can be accessed by clicking the Title “SplitShade: WebGPU Playground ((1) in Figure 9), or the “Docs” button ((8) in Figure 9). For users seeking more technical detail, a link to the project’s GitHub repository is also provided (Github icon button at (8) in Figure 9), offering access to the full backend implementation and source code.

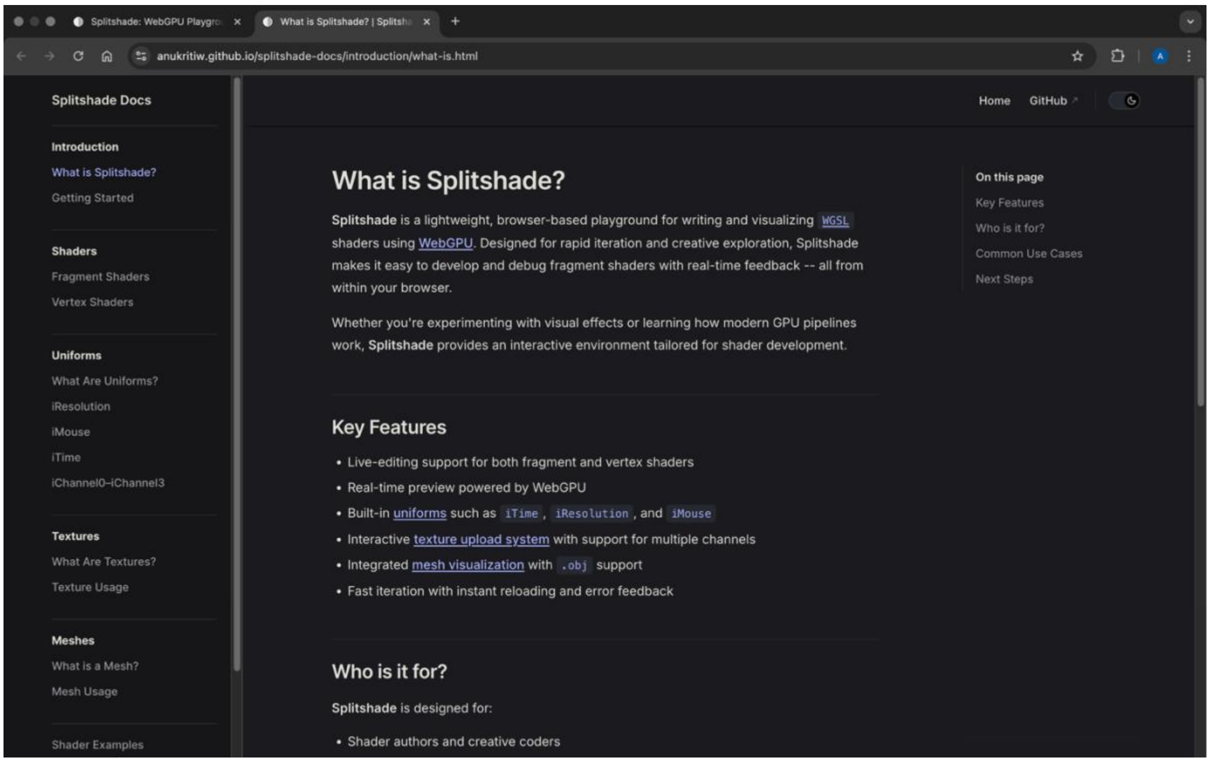
4.5.1 Purpose and Audience

The documentation is intended for users who wish to understand how to operate the SplitShade playground, regardless of their prior experience level with shader development. While the tool can support those learning shader programming for the first time, the documentation focuses on guiding users through SplitShade’s interface, features, and workflows rather than teaching WGSL itself. It assumes that users will consult official WGSL references for language syntax and semantics, and instead concentrates on showing how WGSL code interacts with the resources and controls provided within the application.

4.5.2 Design Considerations

The documentation site is implemented using VitePress, a Markdown-based static site generator optimised for speed and developer experience [47]. This allowed for a clean separation between instructional content and application code, enabling updates to the documentation without affecting the deployment of the SplitShade application itself. VitePress’s lightweight structure, built-in navigation, and support for syntax highlighting made it particularly well-suited to producing concise, easily navigable guidance for end users.

Its inclusion aligns with Nielsen’s usability heuristic of “Help and Documentation”, which emphasises that users should have access to clear, focused information when needed [44]. Rather than overwhelming the user with exhaustive technical detail, the SplitShade documentation is structured for quick reference, allowing users to resolve questions about interface elements, resource management, or workflow steps without leaving the application context for extended periods.


Splitshade: WebGPU Playgro
What is Splitshade? | Splitsh
anukritiw.github.io/splitshade-docs/introduction/what-is.html
Splitshade Docs
Home
GitHub ↗

Introduction
What is Splitshade?
Getting Started

Shaders
Fragment Shaders
Vertex Shaders

Uniforms
What Are Uniforms?
iResolution
iMouse
iTime
iChannel0–iChannel3

Textures
What Are Textures?
Texture Usage

Meshes
What is a Mesh?
Mesh Usage

Shader Examples

What is Splitshade?

Splitshade is a lightweight, browser-based playground for writing and visualizing `WGSL` shaders using WebGPU. Designed for rapid iteration and creative exploration, Splitshade makes it easy to develop and debug fragment shaders with real-time feedback -- all from within your browser.

Whether you're experimenting with visual effects or learning how modern GPU pipelines work, **Splitshade** provides an interactive environment tailored for shader development.

Key Features

- Live-editing support for both fragment and vertex shaders
- Real-time preview powered by WebGPU
- Built-in uniforms such as `iTime`, `iResolution`, and `iMouse`
- Interactive texture upload system with support for multiple channels
- Integrated mesh visualization with `.obj` support
- Fast iteration with instant reloading and error feedback

Who is it for?

Splitshade is designed for:

- Shader authors and creative coders

On this page
Key Features
Who is it for?
Common Use Cases
Next Steps

Figure 16: Screenshot of the SplitShade Documentation page

4.5.3 Structure and Content

The documentation is organised into six sections:

- **Introduction**: Summarises SplitShade and outlines browser requirements.
- **Shaders**: Provides a brief introduction to fragment and vertex shaders, followed by the syntax conventions required for the playground.
- **Uniforms**: Explains the available uniforms, with instructions, examples, and expected outputs.
- **Textures**: Describes how to upload and use textures.
- **Meshes**: Explains the mesh upload process and supported formats.
- **Shader Examples**: Presents two larger-scale example shaders, illustrating practical applications of the platform's features.

4.6 Technical Challenges & Solutions

This section outlines the main technical challenges encountered during SplitShade's development and the solutions adopted to address them. These include language design choices (Section 4.6.1), mesh integration (Section 4.6.2), UI design (Section 4.6.3), balancing inspiration with independent design (Section 4.6.4), and performance considerations (Section 4.6.5).

4.6.1 Designing for WGSL vs. GLSL

A significant challenge in the early stages of development was translating familiar ShaderToy-inspired workflows, originally based on GLSL, into a system that fully supports WGSL. While the two languages share conceptual similarities, WGSL imposes stricter structural requirements. For example, whereas GLSL allows direct texture access via a sampler2D, WGSL requires explicit separation between texture and sampler bindings. This necessitated design changes to accommodate both objects in the binding model without increasing complexity for the user.

To resolve this, SplitShade adopts a conditionally allocated set of texture slots (discussed in Sections 4.2.3.2 and 4.6.5.3). Decisions of this nature – balancing WGSL's stricter requirements with usability and predictability – were made repeatedly throughout the system's design, ensuring a consistent and user-friendly shader development experience.

4.6.2 Mesh Workflow Design

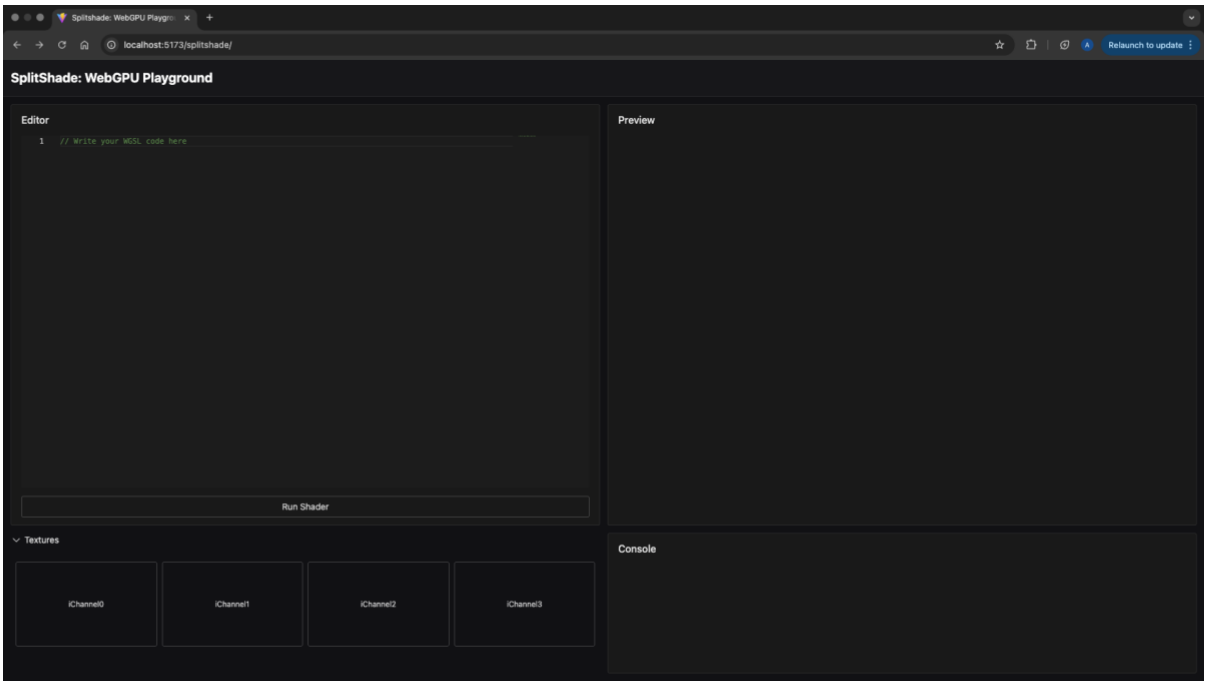
Mesh integration posed a unique challenge due to the relative scarcity of publicly available WebGPU/WGSL examples that incorporate geometry uploads. In most cases, available examples demonstrated vertex shaders with hardcoded geometry rather than supporting external mesh file uploads. This lack of established patterns required the mesh workflow to be designed from first principles.

The solution was to support the Wavefront `.obj` format, chosen for its simplicity and broad adoption in graphics workflows (discussed in Section 4.4.2.2) [48]. This choice allowed the mesh-loading feature to be implemented without the need for complex geometry formats or preprocessing pipelines. Designing the workflow around a consistent vertex layout also made it possible to provide users with a baseline shader that reliably produces a valid render for any uploaded `.obj` mesh, reducing setup complexity and helping users focus on shader logic rather than debugging data format issues. Limitations and future enhancements regarding mesh uploads are discussed in Sections 5.2.5 and 5.3.3 respectively.

4.6.3 UI Design

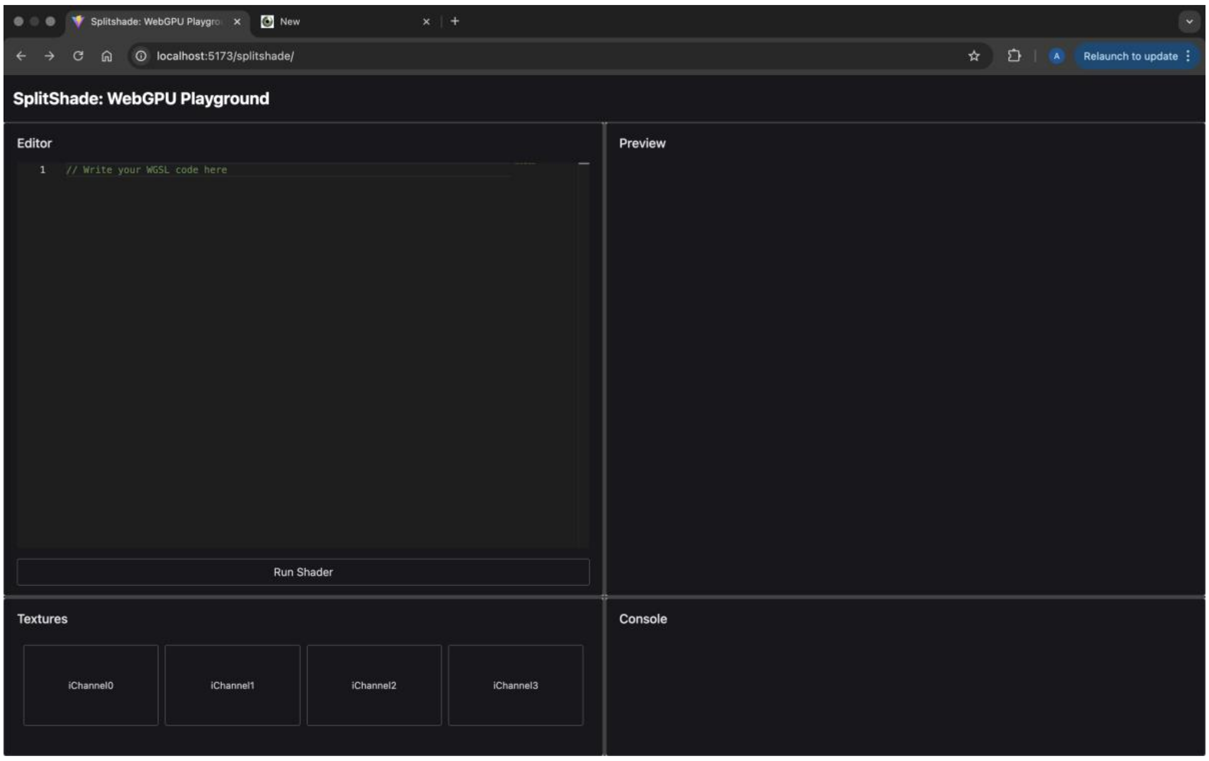
The adoption of a modern front-end stack, including Vue and Naive UI, introduced challenges in both technical proficiency and layout design. Implementing a polished, multi-pane environment required extensive iteration. Initial attempts to make the Resources Panel collapsible (Figure 17), and to implement a split layout with draggable dividers (Figure 18) proved difficult to style to a professional standard while maintaining responsiveness, and was ultimately rolled back in favour of a fixed four-pane arrangement.

The fixed layout offers a stable, predictable interface that displays the code editor, preview, resources, and console panels simultaneously. The layout is responsive, adapting automatically to different window sizes to maintain usability across a range of display resolutions and devices (see PR #85). While less flexible than a fully dynamic layout, this approach removes the complexity of manual resizing and ensures that all panels remain consistently visible. The overall design aligns with ShaderToy's emphasis on immediate access to core panels over highly customisable layouts.



```
// Write your WGSL code here
```

Figure 17: Attempt to make the Textures Panel (later changed to Resources Panel) collapsible



```
// Write your WGSL code here
```

Figure 18: Attempt to follow a split-layout with dynamically resizeable panels

4.6.4 Balancing Interface Inspiration and Independent Design

The interface was inspired by ShaderToy due to its intuitive side-by-side arrangement of code and visual output, as well as its accessible texture management paradigm. However, the goal was not to replicate ShaderToy, but to design a standalone tool optimised for WebGPU's capabilities.

Elements considered effective in ShaderToy, such as the adjacency of the code editor and render preview, were adapted into SplitShade's design. At the same time, notable departures were made, such as implementing a dedicated console panel. GLSL-based environments often provide minimal compilation feedback, whereas WebGPU offers more detailed and structured error messages [6, 21]. By surfacing these messages in a persistent console, the interface aligns with workflows common to modern development environments, where compile-time diagnostics are readily visible alongside the source code. This design choice increases the utility of error messages for debugging and supports an iterative, feedback-driven development process.

4.6.5 Performance Optimizations

The architecture of SplitShade incorporates several strategies to maintain responsiveness and minimise GPU/CPU overhead during shader development.

4.6.5.1 Minimal resource recreation

To reduce unnecessary GPU state changes, SplitShade avoids continuously recompiling shaders or re-binding resources during code editing. The render pipeline, bind groups, and buffers remain allocated between runs, with updates triggered only when the user explicitly presses the "Run Shader" button. This user-driven execution model prevents the performance penalties associated with live recompilation on every keystroke, particularly for shaders with complex pipelines or large mesh/texture inputs, while still providing rapid feedback.

4.6.5.2 Efficient uniform updates

Uniform buffers such as `iTime`, `iMouse`, and `iResolution` are persistent `GPUBuffer` objects that are updated in place each frame using `device.queue.writeBuffer()`. This avoids buffer reallocation or remapping, minimises memory churn, ensures uniform changes propagate efficiently to the GPU without stalling the pipeline.

4.6.5.3 Lazy Texture Allocation

Textures in SplitShade are allocated only when they are actively used in the rendering pipeline (previously discussed in Sections 4.2.3.2). This lazy allocation strategy ensures that GPU memory is not consumed unnecessarily for unused textures. During shader execution, only when at least one texture is explicitly referenced in the shader code are all the textures loaded and bound to the pipeline. This approach minimizes memory usage and avoids redundant GPU resource allocation, particularly in scenarios where multiple textures are available but not all are required for a given shader.

4.6.5.4 Memory management strategies

Default textures and uploaded meshes are transferred to GPU memory only once per assignment and reused across frames; re-uploads occur only when the user replaces an asset. Uniform buffers are persistent and updated in place, further limiting per-frame allocations and keeping CPU-GPU memory traffic predictable.

4.7 Tests

Unit testing formed the core of SplitShade's verification strategy. Tests were written for frontend components, composables, and backend modules to ensure correctness of individual functions and predictable behaviour across different parts of the system.

4.7.1 Frontend Component Tests

Unit tests were developed for key Vue single-file components (`ConsolePanel`, `EditorPanel`, `ResourcesPanel`, `MeshModal`, `TextureModal`, `PreviewPanel`, `WebGPUWarning`) to ensure correct user interface behaviour and reliable interaction flows. These tests verify prop handling, event emission, and conditional rendering under various UI states. For example, tests for the `ResourcesPanel` component verify user actions such as uploading a mesh or selecting a texture trigger the appropriate modal windows. Similarly, `ConsolePanel` tests validate that error and warning messages are rendered in the expected format when compilation feedback is received.

4.7.2 Composable Tests

The project employs Vue composables to manage stateful logic independently of the rendering layer. Dedicated unit tests were written for composables such as `useTextures`, `useMesh`, and `useShaderRunner`. These tests focus on verifying reactive state updates, correct execution of core methods, and proper integration with resource management logic. For instance, `useMesh` tests confirm that uploading a mesh updates both the reactive metadata and the associated parsed vertex data. Testing at this level ensures that core logic functions remain predictable when invoked from multiple components.

4.7.3 Backend Module Tests

Backend-oriented TypeScript modules responsible for WebGPU setup and resource management were tested using mocked GPU device objects to simulate the WebGPU API in a non-browser environment. Modules (`renderer.ts`, `pipeline.ts`, `context.ts`, `shaderUtils.ts`, `wgslReflect.ts`, `uniforms.ts`, `textures.ts` and `objParser.ts`) were validated for correct creation and binding of GPU resources, adherence to predefined binding orders, and appropriate error handling in cases of invalid configuration. For example, `uniforms.ts` tests confirm that all expected uniform buffers are initialised and written to before pipeline execution. These backend tests provide confidence that the lower-level rendering infrastructure behaves consistently regardless of the user's shader code.

4.7.4 Test coverage

Table 1 shows the test coverage for SplitShade.

```
--------------------------|---------|----------|---------|---------|
File                      | % Stmts | % Branch | % Funcs | % Lines |
--------------------------|---------|----------|---------|---------|
All files                 |   83.27 |    77.04 |   67.05 |   83.27 |
 src                      |   84.89 |    84.21 |      60 |   84.89 |
  App.vue                 |    88.1 |    83.33 |   57.14 |    88.1 |
  main.ts                 |       0 |      100 |     100 |       0 |
 src/core                 |     100 |       80 |     100 |     100 |
  context.ts              |     100 |       80 |     100 |     100 |
 src/pipeline             |     100 |      100 |     100 |     100 |
  pipeline.ts             |     100 |      100 |     100 |     100 |
  uniforms.ts             |     100 |      100 |     100 |     100 |
 src/resources            |     100 |       75 |     100 |     100 |
  textures.ts             |     100 |       75 |     100 |     100 |
 src/resources/mesh       |     100 |      100 |     100 |     100 |
  objParser.ts            |     100 |      100 |     100 |     100 |
 src/runtime              |   48.51 |    47.05 |   57.14 |   48.51 |
  renderer.ts             |   48.51 |    47.05 |   57.14 |   48.51 |
 src/shader               |   89.09 |    77.55 |     100 |   89.09 |
  shaderUtils.ts          |     100 |     87.5 |     100 |     100 |
  wgslReflect.ts          |   87.16 |     75.6 |     100 |   87.16 |
 src/ui/components        |     100 |      100 |     100 |     100 |
  WebGPUWarning.vue       |     100 |      100 |     100 |     100 |
 src/ui/components/modals |   84.21 |      100 |      50 |   84.21 |
  MeshModal.vue           |      80 |      100 |   57.14 |      80 |
  TextureModal.vue        |   91.42 |      100 |      40 |   91.42 |
 src/ui/components/panels |   83.26 |    83.33 |   45.83 |   83.26 |
  ConsolePanel.vue        |     100 |       50 |     100 |     100 |
  EditorPanel.vue         |   71.42 |       70 |    62.5 |   71.42 |
  PreviewPanel.vue        |     100 |      100 |     100 |     100 |
  ResourcesPanel.vue      |    87.2 |      100 |    37.5 |    87.2 |
 src/ui/composables       |   98.64 |    76.92 |     100 |   98.64 |
  useMesh.ts              |    97.5 |    84.61 |     100 |    97.5 |
  useShaderRunner.ts      |     100 |    33.33 |     100 |     100 |
  useTextures.ts          |     100 |      100 |     100 |     100 |
--------------------------|---------|----------|---------|---------|
```

Table 1: Test Coverage for SplitShade

Coverage is high across core modules and UI logic. Lower coverage in `renderer.ts` reflects its role as an orchestration layer for browser events, `requestAnimationFrame` scheduling, and WebGPU error scopes – behaviour that is more appropriately validated via integration tests. Given the project scope and timeline, unit tests focused on deterministic logic, while orchestration behaviours were validated manually. Targeted unit tests were added for cancellation, texture channel selection, and fragment-only versus vertex-fragment code paths; remaining branches (e.g., error-scope outcomes and exceptional paths) are planned for future integration testing (Section 5.3.9).

4.8 Continuous Integration and Deployment (CI/CD)

SplitShade uses two GitHub Actions workflows to keep contributions stable and reproducible.

4.8.1 Unit tests on pushes and Pull Requests

The “Run Unit Tests” workflow runs for pushes to main and for pull requests (marked ‘Ready for Review’) targeting main. It executes on Ubuntu with Node 22, installs dependencies via `npm ci`, and runs `Vitest` in a matrix configuration with two shards (`1/2` and `2/2`). This approach ensures fast feedback on every change while maintaining compact job definitions and efficient test execution across multiple stages.

4.8.2 Build checks on Pull Requests

A separate “Check Vite Build” workflow runs on pull requests marked ‘Ready for Review’. It sets up Node 22, installs dependencies with `npm ci`, and verifies that the project builds successfully via `npm run` build. This catches integration issues (type errors, bundling problems, missing assets) before merge.

4.8.3 Automated deployment on merges

In addition to CI, SplitShade implements continuous deployment using “Deploy to GitHub Pages” workflow. When changes are merged into `main`, a dedicated workflow automatically builds the application and deploys it to GitHub Pages. This ensures that the live version of SplitShade is always up to date with the latest merged code, eliminating manual deployment steps and reducing the risk of version drift between the repository and production.

4.9 Deployment

SplitShade and its accompanying documentation site are both deployed using **GitHub Pages**. This choice was driven by its zero-cost hosting model, direct integration with the GitHub repository, and ease of continuous deployment [49] .

The build output from Vite is pushed to a `gh-pages` branch for both SplitShade and its Documentation, making it accessible directly from the browser without requiring server-side processing. This deployment approach allows both the application and its documentation to be versioned alongside their source code and updated as part of the normal Git workflow, reducing operational overhead.

4.10 Git Workflows

Development for SplitShade was managed in a dedicated GitHub repository, following a consistent branch-and-pull-request workflow for all changes. Every modification, whether a new feature, bug fix, test addition, or CI/CD adjustment, was made on a separate branch.

Pull requests (PRs) were opened for each branch before merging into the main branch, ensuring that all changes were documented and reviewed in a structured manner. Each PR outlined the purpose of the change, its expected impact, and, where relevant, included interim screenshots to capture visual progress. This practice created a clear, chronological record of the project's evolution.

A GitHub Project board was used to track issues and feature requests, giving a visual overview of work in progress, completed tasks, and future priorities. Issues were linked to their corresponding PRs to maintain traceability between planned work and implemented solutions.

CI/CD integration (Section 4.8) complemented this workflow by automatically verifying that all changes built successfully, passed tests, and deployed correctly before and upon merging to main.

4.11 Open Source Contributions

As part of SplitShade's development, contributions were made to `wgsl_reflect` [38], a library used internally for shader introspection. Two pull requests were merged: (1) correcting a typographical error in the source, and (2) committing the generated build artifacts, which were previously absent from the repository. This ensured that SplitShade could reliably consume the library without requiring local builds.

Additionally, a suggestion was made to add continuous integration for automatic build file generation, reducing the risk of future commits becoming out of sync. While still under review, this proposal reflects a proactive approach to ensuring that upstream dependencies remain stable and compatible with SplitShade's build process.

4.12 Evaluation

This section focuses on how well the system meets its original objectives, the user experience it delivers, and its capacity for future growth.

4.12.1 Feature completeness vs. goals

The final implementation delivers all planned core features: WGSL shader editing with a preview panel, ShaderToy-style uniform support, multi-channel texture handling, and optional mesh uploads. These are implemented in a stable form, integrated into a single-page interface with minimal context switching. More advanced goals, such as compute shader support, multi-pass rendering, and expanded mesh attribute handling, were intentionally excluded from the scope to ensure timely delivery of a robust core system (discussed in Section 5.3).

4.12.2 Usability

The interface was designed and iterated upon during development to ensure that shader editing, resource management, and previewing could be performed without navigating away from the main workspace. Error messages from WebGPU were surfaced directly in the console panel, providing immediate feedback and reducing debugging time.

4.12.3 Code maintainability and extensibility

The system's architecture separates backend rendering logic, resource management modules, and Vue-based UI components, enabling targeted changes without affecting unrelated parts of the codebase. Automated unit tests cover the majority of frontend and backend modules, providing confidence in refactoring and future feature integration. Areas for potential improvement include extending test coverage to the frame loop logic and adding performance profiling tools for benchmarking more complex workloads (discussed in Section 5.3.8).

5 Conclusion

The conclusion considers SplitShade's value as both a finished tool for shader experimentation and a foundation for future extensions, examining its impact (Section 5.1), present limitations (Section 5.2), and future potential (Section 5.3). It ends with a an overall reflection (Section 5.4).

5.1 Potential Use Cases and Impact

SplitShade has been designed with several potential practical applications in mind. Its accessibility, immediate feedback, and adherence to WebGPU standards make it suitable for education, creative practice, and developer prototyping. Together, these use cases demonstrate SplitShade's broader impact: lowering the barrier to GPU programming, supporting adoption of the WebGPU standard, and fostering open, browser-based experimentation.

5.1.1 Educational

SplitShade lowers the barrier to entry for learners encountering GPU programming for the first time. By removing boilerplate setup and providing immediate feedback, it allows beginners to concentrate on shader logic rather than configuration. This immediacy makes the technology more approachable and encourages exploration without the usual overhead of environment setup. The UI support for textures and meshes further reduces complexity while still giving users meaningful control over their shaders, enabling them to create impressive results early in the learning process. As such, SplitShade is well suited for computer graphics courses, workshops, or self-directed study, where interactive experimentation accelerates understanding.

5.1.2 Creative

For shader artists and creative practitioners, SplitShade serves as an accessible canvas for generative design. The ability to write WGSL shaders directly in the browser encourages rapid prototyping and iteration, supporting artistic exploration without specialised software or build pipelines. Support for textures and meshes further extends its potential for both 2D and 3D generative art practices.

5.1.3 Developer prototyping

Developers working with WebGPU can use SplitShade as a lightweight prototyping environment. It provides a controlled context for validating shader code, testing resource bindings, and exploring new API features before integrating them into larger applications. In this way, SplitShade contributes not only to learning and creativity but also to practical development workflows.

5.1.4 Broader Impact

Taken together, these use cases demonstrate SplitShade's contribution to the WebGPU ecosystem. By making shader programming more accessible and interactive, it encourages both learning and innovation. Its open-source availability extends this impact, allowing the community to adapt and extend the tool for their own contexts. More broadly, SplitShade illustrates how browser-based tooling can reduce entry barriers to advanced GPU programming, supporting the wider adoption of WebGPU as a standard.

5.2 Limitations

While SplitShade deliberately adopts a simplified design to prioritise accessibility and fast iteration, this approach naturally imposes several limitations compared to production-grade graphics frameworks. Many of these constraints reflect conscious trade-offs that balance functionality with usability, and also serve as opportunities for future extensions.

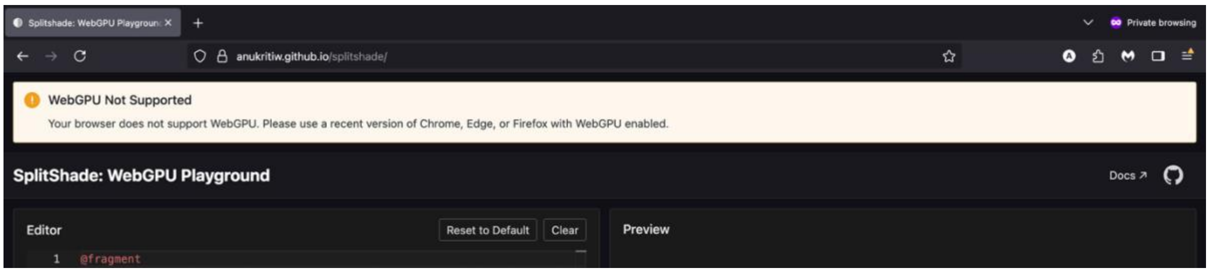
5.2.1 Simplified Rendering Architecture

SplitShade's rendering engine is designed for accessibility and rapid prototyping rather than production-grade complexity. Its single-draw-call architecture, executed through `runRenderPass()`, ensures predictable behaviour and lowers the barrier for first-time shader authors. While this simplicity means advanced techniques such as multi-pass rendering, deferred shading, or complex geometry pipelines are not currently supported, the trade-off allows users to focus on shader logic without being overwhelmed by configuration. Multipass rendering as a future enhancement is discussed in Section 5.3.5.

Similarly, the render pipeline creation in `createPipeline()` focuses on straightforward vertex-fragment workflows. By abstracting away advanced configurations such as multiple render targets or custom blending modes, the system ensures that pipeline setup remains simple and reliable. This intentional constraint reduces the risk of errors and makes it easier for users to iterate quickly, while still leaving a clear pathway for more complex pipeline configurations in future extensions.

5.2.2 Browser support requirements

SplitShade relies on WebGPU, a standard that remains under active development. As a result, browser support is narrower than for legacy APIs like WebGL, with certain platforms lacking full compatibility [7]. This reflects the current state of the ecosystem rather than a platform-specific constraint. To mitigate this, SplitShade provides clear feedback through a warning banner (Figure 19), ensuring users are informed of compatibility issues instead of encountering silent failures.


Splitshade: WebGPU Playgroun ×
Private browsing
anukritiw.github.io/splitshade/
SplitShade: WebGPU Playground
Docs ↗
Editor
Reset to Default
Clear
Preview

WebGPU Not Supported

Your browser does not support WebGPU. Please use a recent version of Chrome, Edge, or Firefox with WebGPU enabled.

```
@fragment
```

Figure 19: Screenshot showing warning banner on unsupported browsers.

5.2.3 Limited debugging capabilities

SplitShade provides effective basic debugging through comprehensive error reporting in `compileShaderModule()` and clear structured error messages for shader compilation issues. However, the platform does not expose several advanced debugging capabilities that WebGPU makes available, such as GPU timing queries for performance measurement, debug labels and markers for resource tracking, or fine-grained error scopes for device state monitoring. This omission is deliberate: SplitShade's simplified interface prioritises accessibility and rapid prototyping over detailed profiling. For those requiring advanced analysis or optimisation, this represents a limitation of the current implementation rather than of WebGPU itself, with potential profiling-related enhancements discussed in Section 5.3.8.

5.2.4 Texture format constraints

To keep the authoring model predictable, SplitShade currently supports a fixed texture format (`rgba8unorm-srgb`) and a maximum of four texture channels (`iChannel0 - 3`). While this constrains experimentation with alternative formats or resource binding strategies, it mirrors ShaderToy's established model and ensures shaders are portable and consistent across browsers.

5.2.5 Mesh import constraints

Currently, SplitShade's mesh import pipeline extracts only vertex positions from OBJ files, assigning a default white colour to all vertices. This simplified approach creates several constraints that limit the platform's compatibility with advanced 3D workflows:

- **No Texture Coordinate Support:** The parser does not extract UV coordinates (`vt` entries) from OBJ files yet, preventing proper texture mapping on uploaded meshes.
- **Limited Material Information:** OBJ files often reference Material Template Library (MTL) files that define surface properties including diffuse colours, specular properties, and texture maps [50]. The current implementation does not yet support MTLs, resulting in all geometry appearing with uniform white colouring regardless of the original material definitions.
- **No support for Vertex Attributes:** The parser omits other standard vertex attributes such as normals (`vn` entries), which are essential for realistic lighting calculations. This prevents users from implementing normal-based shading techniques or advanced lighting models on imported geometry.

These limitations stem from the need to keep the initial mesh workflow minimal and predictable within project time constraints. Parsing richer attribute sets (normals, UVs, materials) would have required updating shader interfaces, binding layouts, and fallback handling for cases where attributes were absent, which are complexities that exceeded the scope of the first implementation. Section 5.3.3 discusses related future enhancements.

5.2.6 Memory and Resource Management

GPU memory management in SplitShade follows simplified allocation patterns that abstract away low-level details. This design avoids exposing learners to complex lifetimes and allocation strategies, which can be error-prone, and instead guarantees predictable resource behaviour. While this restricts the fine-grained control available when working directly with the WebGPU API, the omission is deliberate. It aligns with the platform's goal of providing a safe, consistent, and accessible learning environment.

5.2.7 Absence of Compute Shader support

At present, SplitShade is limited to graphics rendering via vertex and fragment shaders. Compute shaders, which extend GPU programming into areas such as physics simulations or parallel data processing [6], are not yet supported. This keeps the platform focused on its core use case of shader authoring while leaving a clear pathway for future expansion (see Section 5.3.1).

5.3 Future Enhancements

Looking beyond its current implementation, SplitShade has been designed with extensibility in mind. The following subsections describe potential enhancements that would broaden its functionality and support a wider range of educational and creative use cases.

5.3.1 Support for Compute Shaders

Currently SplitShade focuses exclusively on graphics rendering through vertex and fragment shaders, utilizing WebGPU's render pipeline for visual output. However, the platform's foundational architecture demonstrates strong compatibility with compute shader workflows, positioning it well for future expansion into general-purpose GPU computing.

The existing infrastructure already incorporates several key components essential for compute operations. The centralized WebGPU device management in `getWebGPUDevice()` provides the same GPU context required for compute pipelines, while the comprehensive buffer creation and management system in `createUniforms()` establishes patterns for GPU memory allocation that translate directly to compute shader data buffers. The command encoder architecture used in `runRenderPass()` demonstrates the platform's capability to orchestrate GPU command submission – a core requirement for compute dispatch operations.

Compute shaders are a significant advancement that WebGPU offers in comparison to WebGL. Supporting compute shaders would unlock significant new capabilities for users, enabling GPU-accelerated algorithms such as physics simulations and parallel data processing [6]. This would transform SplitShade from a solely graphics-focused playground into a comprehensive WebGPU development environment. The existing error handling and shader compilation infrastructure could seamlessly extend to compute shader validation.

The architectural pathway to compute support requires minimal disruption to existing systems. The current `createPipeline()` function demonstrates the pattern for pipeline creation that would extend naturally to `createComputePipeline()`. Buffer management already handles uniform data updates via `device.queue.writeBuffer()`, which forms the foundation for compute input/output buffer operations. The existing binding group system could accommodate compute shader resource bindings alongside the current texture and uniform layouts, maintaining the platform's clean separation of concerns while expanding its computational scope. Likewise, the Console Panel could provide a straightforward channel for surfacing compute outputs such as buffer values or performance metrics, complementing visual feedback with textual results.

5.3.2 Expanded uniform support

Currently SplitShade exposes only `iResolution`, `iTime`, and `iMouse` uniforms. Expanding this set would align the system more closely with platforms like ShaderToy. Examples include per-channel resolution (`iChannelResolution`), time deltas between frames (`iTimeDelta`), frame counts (`iFrame`), and date/time inputs (`iDate`). Such additions would enable more complex time-based effects or procedural animations.

The platform's uniform binding architecture also creates a foundation for advanced 3D graphics workflows. The current binding layout (0-2 for core uniforms, 3+ for textures) reserves ample space for camera matrices (view, projection, model) and lighting parameters (light positions, colours, material properties).

Users can already implement lighting effects within fragment shaders for screen-space effects and vertex shaders for mesh-based lighting using the existing position and colour attributes. The modular `createUniforms()` function in the pipeline can be extended to include additional uniform buffers without architectural changes, ensuring that advanced 3D features like proper camera transformations and sophisticated lighting models can be added progressively without disrupting the existing shader development workflow.

5.3.3 Expanded Mesh Parsing and Format Support

Currently, SplitShade's mesh import pipeline extracts only vertex positions and assigns white as a default colour. Trivially, a field allowing users to select colour manually could be easily added to the UI to replace the default white.

More substantial, however, would be supporting the OBJ format's additional attributes such as normals and texture coordinates [50]. OBJ files can also reference Material Template Library (MTL) files that define surface properties including diffuse colours, specular properties, and texture maps. Supporting MTL parsing would enable material-based colouring by extracting diffuse colour values (Kd) and mapping them to mesh faces, providing more realistic visual representation.

Importantly, SplitShade already employs the `obj-file-parser` library [45], which is capable of extracting normals, texture coordinates, and material references, ensuring that the pipeline is well-positioned for future extension with minimal architectural change. Supporting these extended attributes would unlock a wider range of rendering techniques, including normal-based lighting calculations and texture mapping.

Beyond OBJ, additional mesh standards could broaden compatibility and extend the range of vertex inputs available in the vertex stage. glTF/GLB provides a modern, browser-friendly container with rich attributes (positions, normals, tangents, UVs, and colours). Already established as the *de facto* exchange format for WebGL, particularly through adoption in three.js and Babylon.js, it represents a natural candidate for future WebGPU integration [51, 52]. PLY would extend support for dense per-vertex colour data, making it particularly valuable for visualisation and generative art workflows. STL, though more limited, offers a straightforward geometry-only representation widely used in 3D printing [52]. Supporting these formats would allow shaders to leverage a wider range of vertex inputs in the vertex stage.

5.3.4 Save/Load/Share sessions, and Community building

The ability to save work and reload it in later sessions would improve usability and reproducibility. Although local persistence is already supported, providing an explicit save mechanism gives users greater confidence that their work can be reliably restored at a later time, aligning with Nielsen's principle of "user control and freedom" [44]. SplitShade could also incorporate play/pause/record functionality, similar to ShaderToy (Section 2.3), enhancing usability and supporting the reproducibility of animated shader experiments.

Extending this further, a feature allowing users to share their shaders through URL sharing would encourage collaborative shader writing, potentially fostering community. This would align with the established sharing models of many generic browser-based programming snippet storage systems such as GitHub Gists [53] or playgrounds like CodePen [54], where rapid dissemination of creative experiments fosters both learning and innovation.

Going even further than that, a community page could be set up to extend SplitShade, akin to ShaderToy's community page, encouraging shader authors to share their projects. This change would ideally also mean allowing users to set up accounts on the page, which could thereby aid in the earlier suggestions of saving and loading their shaders as well.

5.3.5 Multipass Rendering

SplitShade currently supports only single-pass rendering (discussed in Section 5.2.1), which limits shaders to producing their final output in one stage. By contrast, as mentioned in Section 2.3, ShaderToy demonstrates the creative potential of multipass rendering through its use of multiple buffers (A–D) that can feed into each other before a final image pass. This enables more advanced effects such as post-processing pipelines, fluid simulations, path tracing, and iterative feedback systems [13].

SplitShade could be extended by similarly implementing multipass rendering. This would involve extending the current render pipeline orchestration to manage multiple passes, each writing to and reading from intermediate textures. With careful design, this could be achieved without sacrificing the platform's emphasis on predictability, for example by adopting a fixed buffer naming convention similar to ShaderToy's. Multipass support would align SplitShade with a broader range of GPU programming workflows, while still keeping the user experience approachable.

5.3.6 Multi-shader comparisons

Allowing two or more shaders to be run and displayed side by side would open up possibilities for both education and experimentation. Instructors could demonstrate different implementations of the same effect, while developers could benchmark multiple approaches interactively. For creative artists, this feature would allow visual styles to be compared more easily before committing to a single version.

5.3.7 Shader preset library

Introducing a library of shader presets would lower the barrier to first use even further and provide inspiration for beginners. Users could load a set of curated examples demonstrating techniques such as texture sampling, noise functions, colour manipulation, etc. This feature has precedent in other projects, such in *06wj WebGPU Playground* [17] (see Section 2.5), and would enhance SplitShade's role as both a learning tool and a rapid prototyping environment.

5.3.8 Performance profiling tools

Performance analysis could make SplitShade more valuable for advanced users and developers. Profiling features might include visual overlays showing frame time breakdowns (e.g. uniform updates, pipeline execution, draw calls), GPU timing queries to measure shader execution cost, or heatmaps to visualise fragment overdraw. These tools would help users identify bottlenecks in their shader code, experiment with optimisation strategies, and gain a practical understanding of GPU performance trade-offs.

5.3.9 Integration and System Tests

Currently, SplitShade relies primarily on unit tests to verify the correctness of individual components, composables, and backend modules (Section 4.7). While effective at the granular level, unit tests cannot fully capture cross-component interactions or guarantee end-to-end reliability.

Future testing could therefore include integration tests that validate workflows spanning multiple modules, such as ensuring that shader compilation errors propagate from the backend to the editor interface, or that texture selections in the Resources Panel correctly bind through to the WebGPU pipeline. At the system level, full end-to-end tests could simulate user scenarios, such as writing a shader, assigning resources, and verifying correct render output. This would provide stronger confidence in the platform's behaviour under real usage conditions.

These higher-level testing layers would grow in importance as the platform expands to include features such as multipass rendering (Section 5.3.5) or shader sharing (Section 5.3.4), where cross-module dependencies are more complex. Adding integration and system tests to the CI pipeline would not only reduce regression risk but also support extensibility by giving contributors greater confidence when making architectural changes.

5.4 Reflection

Developing SplitShade has been as much an educational journey as it has been a technical exercise. Working with WebGPU and WGSL required engaging with the API at a relatively low level and translating those abstractions into a usable platform. Throughout the process, the importance of clear separation of concerns became evident: when shader compilation, resource binding, and pipeline setup were modularised, each component could be understood, tested, and extended independently.

Several of the design choices were intentionally simple. The single-draw-call architecture and fixed uniform schema imposed limits on the feature set but also kept the system focused and approachable. These trade-offs prioritised clarity over breadth, allowing shader logic to take centre stage without unnecessary configuration overhead. They also underscored the balance between exposing power and reducing friction, a theme that runs through GPU programming at large.

At the same time, SplitShade evolved beyond its technical foundations into a complete, user-facing tool. The interface was designed to introduce complex concepts in a way that remained approachable, while workflows around Git, automated testing, and version control supported maintainability and robustness. External libraries were also integrated selectively, and clear user-facing documentation reinforced the UI and lowered barriers to entry. As a result, SplitShade has emerged as more than a minimum viable product (MVP). It is an end-to-end product that combines technical soundness with thoughtful design, enabling learning, creative exploration, and a foundation for future innovation.

6 Acknowledgements

Sincere thanks are extended to **Professor Jon Macey** and to **Professor Jian Chang**, whose insight, feedback, and technical expertise provided clarity during challenging stages of development. Their mentorship was instrumental throughout this project, and their contributions are gratefully acknowledged.

Gratitude is also extended to the **MSc Computer Animation and Visual Effects (CAVE) Class of 2025**, whose encouragement and camaraderie fostered a supportive environment during the course of this project. Their enthusiasm and shared commitment to learning made the experience both rewarding and motivating.

Finally, heartfelt thanks go to my **family and friends** back home in Singapore, whose unwavering support and kindness sustained me throughout this journey.

7 References

[1] A. Evans, M. Romeo, A. Bahrehman, J. Agenjo and J. Blat, “3D graphics on the web: A survey,” *Computers & graphics,* pp. 43-61, 2014.

[2] G. Yu, C. Liu, T. Fang, J. Jia, E. Lin, Y. He, S. Fu, L. Wang, L. Wei and Q. Huang, “A survey of real-time rendering on Web3D application,” *Virtual Reality & Intelligent Hardware,* vol. 5, no. 5, pp. 379-394, 2023.

[3] C. McClanahan, “History and evolution of gpu architecture,” *A Survey Paper,* vol. 9, pp. 1-7, 2010.

[4] Intel, “What is a GPU?,” 2024. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html. [Accessed 1 August 2025].

[5] Amazon Web Services, “What is a GPU?,” 2025. [Online]. Available: https://aws.amazon.com/what-is/gpu/. [Accessed 1 August 2025].

[6] C. Wallez, B. Jones and F. Beaufort, “WebGPU: Unlocking modern GPU access in the browser,” Google (Chrome for Developers), 2023. [Online]. Available: https://developer.chrome.com/blog/webgpu-io2023#context_on_webgpu. [Accessed 1 August 2025].

[7] MDN Contributors, “WebGPU API,” Mozilla, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API#browser_compatibility. [Accessed 19 August 2025].

[8] greggman, “WebGPU Fundamentals,” 16 July 2025. [Online]. Available: https://webgpufundamentals.org/webgpu/lessons/webgpu-fundamentals.html. [Accessed 19 August 2025].

[9] MDN Contributors, “WebGL: 2D and 3D graphics for the web,” Mozilla, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API. [Accessed 1 August 2025].

[10] G. Yang, “The Future of Web Graphics is Coming,” Intel, 22 September 2021. [Online]. Available: https://community.intel.com/t5/Blogs/Tech-Innovation/Edge-5G/The-Future-of-Web-Graphics-is-Coming/post/1332534?. [Accessed 20 August 2025].

[11] R. Cabello, “GLSL Sandbox,” 2011. [Online]. Available: https://glslsandbox.com/e. [Accessed 2 August 2025].

[12] R. Cabello, “glsl-sandbox,” 2025. [Online]. Available: https://github.com/mrdoob/glsl-sandbox. [Accessed 2 August 2025].

[13] I. Quilez and P. Jeremias, “Shadertoy,” 2013. [Online]. Available: https://www.shadertoy.com/. [Accessed 2 August 2025].

[14] I. Quilez, “Interview with Inigo Quilez,” 2024. [Online]. Available: https://antoinehuot.io/2024/10/22/interview-with-inigo-quilez/. [Accessed 3 August 2025].

[15] P. Jeremias and I. Quilez, “Shadertoy: Learn to create everything in a fragment shader,” in *SIGGRAPH Asia 2014 Courses*, 2014, pp. 1-15.

[16] A. Usher, Y. Chen, S. Elliott, R. Gupta, A. Verma, V. Wang and H. Watson, “WebGPUniverse,” 2021. [Online]. Available: https://webgpuniverse.netlify.app/. [Accessed 4 August 2025].

[17] 06wj, “WebGPU Playground,” 2023. [Online]. Available: https://06wj.github.io/WebGPU-Playground/. [Accessed 2 August 2025].

[18] F. Fornwall, “WebGPU Compute Playground,” 23 September 2023. [Online]. Available: https://compute.fornwall.net/. [Accessed 4 August 2025].

[19] Babylon.js, “Babylon.js Playground,” [Online]. Available: https://playground.babylonjs.com/. [Accessed 3 August 2025].

[20] greggman, “WebGPU Compute Shader Basics,” 16 July 2025. [Online]. Available: https://webgpufundamentals.org/webgpu/lessons/webgpu-compute-shaders.html. [Accessed 18 August 2025].

[21] W3C, “WebGPU Explainer,” 2025. [Online]. Available: https://gpuweb.github.io/gpuweb/explainer/. [Accessed 4 August 2025].

[22] greggman, “WebGPU from WebGL,” 16 July 2025. [Online]. Available: https://webgpufundamentals.org/webgpu/lessons/webgpu-from-webgl.html. [Accessed 4 August 2025].

[23] W3C, “WebGPU Shading Language,” 2025. [Online]. Available: https://www.w3.org/TR/WGSL/. [Accessed 4 August 2025].

[24] Khronos, “Rendering Pipeline Overview,” OpenGL Wiki, 7 November 2022. [Online]. Available: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview. [Accessed 4 August 2025].

[25] C. Cincotti, “The Rendering Pipeline | WebGPU | Video,” 16 January 2023. [Online]. Available: https://carmencincotti.com/2023-01-16/how-to-render-a-webgpu-triangle-series-part-six-video/. [Accessed 4 August 2025].

[26] Khronos, “Vertex Shader,” OpenGL Wiki, 2017. [Online]. Available: https://www.khronos.org/opengl/wiki/Vertex_Shader. [Accessed 4 August 2025].

[27] derhass, “What's the best way to draw a fullscreen quad in OpenGL 3.2?,” Stack Overflow, 14 January 2020. [Online]. Available: https://stackoverflow.com/a/59739538. [Accessed 4 August 2025].

[28] C. Wallis, “Optimizing Triangles for a Full-screen Pass,” 18 April 2021. [Online]. Available: https://wallisc.github.io/rendering/2021/04/18/Fullscreen-Pass.html. [Accessed 4 August 2025].

[29] Khronos, “Fragment Shader,” OpenGL Wiki, 2020. [Online]. Available: https://www.khronos.org/opengl/wiki/Fragment_Shader. [Accessed 4 August 2025].

[30] greggman, “WebGPU Uniforms,” 16 July 2025. [Online]. Available: https://webgpufundamentals.org/webgpu/lessons/webgpu-uniforms.html. [Accessed 4 August 2025].

[31] greggman, “WebGPU Textures,” 16 July 2025. [Online]. Available: https://webgpufundamentals.org/webgpu/lessons/webgpu-textures.html. [Accessed 18 August 2025].

[32] ShaderToy, “Documentation,” [Online]. Available: https://www.shadertoy.com/howto. [Accessed 4 August 2025].

[33] E. You, “Vue.js,” VoidZero Inc., 2025. [Online]. Available: https://vuejs.org/. [Accessed 5 August 2025].

[34] E. You, “Vite,” VoidZero Inc., 2025. [Online]. Available: https://vite.dev/. [Accessed 5 August 2025].

[35] Microsoft, “Monaco - The Editor of the Web,” 2025. [Online]. Available: https://microsoft.github.io/monaco-editor/. [Accessed 5 August 2025].

[36] Ming, “Vue Monaco Editor,” 2025. [Online]. Available: https://imguolao.github.io/monaco-vue/. [Accessed 5 August 2025].

[37] TuSimple, “Naive UI,” 2025. [Online]. Available: https://www.naiveui.com/. [Accessed 5 August 2025].

[38] B. Duncan, “wgsl_reflect,” 2025. [Online]. Available: https://github.com/brendan-duncan/wgsl_reflect. [Accessed 5 August 2025].

[39] B. Jones, “WebGPU Bind Group best practices,” 11 April 2023. [Online]. Available: https://toji.dev/webgpu-best-practices/bind-groups.html. [Accessed 6 August 2025].

[40] Khronos, “Vertex Specification Best Practices,” OpenGL Wiki, 2022. [Online]. Available: https://www.khronos.org/opengl/wiki/Vertex_Specification_Best_Practices. [Accessed 7 August 2025].

[41] h4lc0n, “How does interleaved vertex submission help performance?,” Stack Overflow, 26 January 2013. [Online]. Available: https://stackoverflow.com/a/14535465. [Accessed 7 August 2025].

[42] Microsoft, “Visual Studio Code,” 2025. [Online]. Available: https://code.visualstudio.com/. [Accessed 8 August 2025].

[43] P. Carbonnelle, “Top IDE index,” 2025. [Online]. Available: https://pypl.github.io/IDE.html. [Accessed 8 August 2025].

[44] J. Nielson, “Enhancing the explanatory power of usability heuristics.,” in *SIGCHI Conference on Human Factors in Computing Systems*, New York, 1994.

[45] Wes, “obj-file-parser,” 23 January 2023. [Online]. Available: https://github.com/WesUnwin/obj-file-parser. [Accessed 17 July 2025].

[46] W3C, “WebGPU: Shader Module Compilation Information,” 2025. [Online]. Available: https://www.w3.org/TR/webgpu/#gpucompilationinfo. [Accessed 14 August 2025].

[47] E. You, “VitePress,” VoidZero Inc., 2025. [Online]. Available: https://vitepress.dev/. [Accessed 9 August 2025].

[48] Adobe, “A guide to 3D file types,” [Online]. Available: https://www.adobe.com/uk/products/substance3d/discover/3d-files-formats.html. [Accessed 9 August 2025].

[49] GitHub, “GitHub Pages,” [Online]. Available: https://pages.github.com/. [Accessed 9 August 2025].

[50] P. Bourke, “Object Files (.obj),” [Online]. Available: https://paulbourke.net/dataformats/obj/. [Accessed 19 August 2025].

[51] L. Blue, “Load 3D Models in glTF Format,” 2018. [Online]. [Accessed 20 August 2025].

[52] Khronos, “Introduction to glTF using WebGL,” 2021. [Online]. Available: https://github.khronos.org/glTF-Tutorials/gltfTutorial/gltfTutorial_001_Introduction.html. [Accessed 20 August 2025].

[53] GitHub, “GitHub Gists,” [Online]. Available: https://gist.github.com/. [Accessed 19 August 2025].

[54] Codepen, “Codepen,” 2025. [Online]. Available: https://codepen.io/. [Accessed 19 August 2025].

[55] G2A, “OpenGL vs. DirectX: A Guide for Gamers,” 2024. [Online]. Available: https://www.g2a.com/news/features/opengl-vs-directx-guide/. [Accessed 1 August 2025].

[56] A. Gullen, “A brief history of graphics on the web and WebGPU,” 23 April 2020. [Online]. Available: https://www.construct.net/en/blogs/ashleys-blog-2/brief-history-graphics-web-1517?. [Accessed 1 August 2025].

[57] K. Ge, “What is GPU programming?,” Red Hat Developer, 2024. [Online]. Available: https://developers.redhat.com/articles/2024/08/07/what-gpu-programming#. [Accessed 1 August 2025].

[58] B. Kenwright, “Introduction to the WebGPU API,” in *Acm siggraph 2022 courses*, 2022, pp. 1-184.

[59] G. Mugweni, “WebGPU Shaders: A Beginner's Step-by-Step Guide,” 15 December 2024. [Online]. Available: https://giftmugweni.hashnode.dev/webgpu-shaders-a-beginners-step-by-step-guide. [Accessed 4 August 2025].