

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma Ferramenta Web para Apoio ao Ensino de Shaders em Computação Gráfica

Felipe Dantas Borges

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Carla Denise Castanho

Brasília
2025



Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma Ferramenta Web para Apoio ao Ensino de Shaders em Computação Gráfica

Felipe Dantas Borges

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Carla Denise Castanho (Orientador)
CIC/UnB

Carla Silva Rocha Aguiar
FCTE/UnB

Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 16 de dezembro de 2025

Dedicatória

Dedico este trabalho à minha família e aos amigos que, de perto ou de longe, me acompanharam ao longo da graduação, oferecendo apoio, paciência e incentivo nos momentos em que foi mais difícil continuar.

Dedico também, com muito carinho, à Triz, minha namorada, pela presença constante, pelas conversas, pelos conselhos sinceros e por ter dividido comigo tantas madrugadas de cansaço, ansiedade e, eventualmente, alívio. Sua companhia tornou todo esse processo muito mais leve e possível.

Estendo ainda esta dedicatória aos meus quatro gatos, que são como filhos para mim e que, sem saber, estiveram presentes em cada página escrita, fazendo companhia silenciosa nas longas horas em frente ao computador.

Por fim, dedico este trabalho ao meu estúdio de jogos, Indigo Dreams, e aos meus sócios, na esperança de que tudo o que aprendi ao longo do curso contribua para tornar nossos jogos, nossas ideias e nosso estúdio cada vez melhores.

Agradecimentos

Agradeço, em primeiro lugar, à minha orientadora, Professora Carla Denise Castanho, por ter oferecido a disciplina de Introdução a Jogos e por todo o material e discussões que a acompanharam. Essa matéria abriu meus olhos para uma carreira que eu não imaginava ser possível, despertando um lado criativo que eu desconhecia e mostrando como a Computação pode ser também um espaço de expressão artística e experimental. Também agradeço aos Professores Carla Rocha e Ricardo Jacobi por terem aceito o convite para participar da banca de avaliação deste trabalho.

Agradeço à minha família e amigos, pelo apoio incondicional em todas as etapas da graduação, pela confiança depositada em mim e por compreenderem as ausências, o cansaço e as mudanças de humor típicas dessa fase. Sem esse suporte emocional e prático, este trabalho dificilmente teria chegado ao fim.

Um agradecimento especial à Triz, minha namorada, por ter estado ao meu lado durante todo o percurso: pelas palavras de incentivo, pela escuta atenta, pelas revisões ocasionais, pela paciência com os prazos apertados e por acreditar em mim mesmo quando eu não acreditava tanto assim.

Agradeço também aos meus sócios e colegas do estúdio Indigo Dreams, por compreenderem as demandas da vida acadêmica, ajustarem rotinas quando necessário e, ao mesmo tempo, continuarem sonhando junto comigo com projetos de jogos cada vez mais ambiciosos. Que o conhecimento adquirido aqui retorne em forma de experiências melhores para quem joga o que criamos.

Por fim, agradeço a todos os professores e colegas de curso que, direta ou indiretamente, contribuíram para a minha formação em Computação, seja em sala de aula, em projetos, em monitorias ou em conversas de corredor. Cada disciplina, dúvida respondida e discussão técnica ajudou a construir o caminho que culmina neste trabalho.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Este trabalho apresenta uma ferramenta interativa, executada diretamente no navegador, concebida para reduzir a distância entre a discussão teórica sobre shaders e a prática efetiva de programá-los em Computação Gráfica (CG). A aplicação organiza, em uma única tela, texto explicativo, editor de código (vertex e fragment) e visualização em tempo real, de modo a favorecer um ciclo rápido de experimentação no qual o estudante lê, modifica o shader e observa imediatamente o efeito na cena. Do ponto de vista técnico, a ferramenta foi implementada em JavaScript, utilizando *three.js* como camada de alto nível para cena, câmeras e materiais programáveis, e um editor embutido com realce de sintaxe para o código em GLSL. O escopo deste trabalho foi deliberadamente delimitado à validação do funcionamento da ferramenta e de seu fluxo básico de uso por estudantes, por meio de um estudo exploratório com 19 participantes que resolveram três lições introdutórias e responderam a um questionárion sobre compreensão conceitual, usabilidade e percepção dos exercícios. Os resultados indicam alta concordância quanto à compreensão do papel de shaders e à utilidade das atividades, ao mesmo tempo em que evidenciam pontos de melhoria em UI/UX e na estrutura didática das lições iniciais.

Palavras-chave: computação gráfica, shaders, ensino, WebGL, GLSL

Abstract

This work presents an interactive, browser-based tool designed to reduce the gap between theoretical discussion about shaders and the practical task of programming them in Computação Gráfica (CG). The application arranges, on a single screen, explanatory text, a code editor (vertex and fragment), and a real-time visualization, fostering a fast experimentation cycle in which the student reads, modifies the shader, and immediately observes its effect on the scene. From a technical standpoint, the tool was implemented in JavaScript, using *three.js* as a high-level layer for scenes, cameras, and programmable materials, together with an embedded syntax-highlighted editor for GLSL code. The scope of this work was deliberately limited to validating the tool's behavior and its basic usage flow from the students' perspective, through an exploratory study with 19 participants who completed three introductory lessons and answered a questionnaire on conceptual understanding, usability, and perception of the exercises. The results indicate strong agreement regarding the understanding of the role of shaders and the usefulness of the activities, while also highlighting opportunities for improvement in UI/UX and in the didactic structure of the initial lessons.

Keywords: computer graphics, shaders, teaching, WebGL, GLSL

Sumário

1 Introdução **1**

1.1 Contexto: da Computação Gráfica aos shaders 1
1.2 Origem e ensino de shaders 2
1.3 Desafios no ensino em shaders 3
1.4 Objetivos e escopo do trabalho 3
1.5 Organização do texto 4

2 Fundamentação Teórica **5**

2.1 Panorama, origem e aplicações dos *shaders* 5
2.1.1 Aplicações e impacto 6
2.1.2 Origem e evolução 8
2.1.3 Aplicações e impacto 9
2.2 Pipeline programável e fluxo de dados 9
2.2.1 Espaços de coordenadas, PVM e *clip space* 9
2.2.2 *Vertex* e *fragment shaders* 10
2.2.3 GLSL em linhas gerais 10
2.2.4 Texturas, amostragem e mapeamento UV 11
2.2.5 *Z-buffer* e testes 11
2.3 Ensino de Computação Gráfica 12
2.3.1 Desafios no aprendizado 12
2.3.2 Ferramentas de apoio ao ensino/aprendizagem de *shaders* 13
2.3.3 Discussão 13

3 A Ferramenta **16**

3.1 Objetivo e contribuição 16
3.1.1 Justificativa e posicionamento 17
3.1.2 Cenários de uso e público-alvo 17
3.1.3 Arquitetura geral (visão de implementação) 17
3.1.4 Tela única da aplicação 18

3.1.5 Formato de projeto (`.chroma`) e autoria de conteúdo 22

4 Implementação **24**
4.1 Pilha tecnológica e motivos de escolha 24
4.2 Arquitetura geral de módulos 25
4.3 Ciclo *compilar* → *ligar* → *desenhar* 26
4.4 Gestão de *uniforms*: didáticos e internos 27
4.4.1 Injeção automática de *uniforms*. 27
4.5 Geometrias, câmeras e parâmetros de cena 31
4.6 Texturas e canais `u_texture0..3` 31
4.7 Tratamento de erros e mapeamento de linhas 32
4.8 Formato de projeto (`.chroma`) e *blobs* de textura 33
4.9 Integração teoria–editor e restauração de sessão 34

5 Testes **35**
5.1 Desenho do estudo 35
5.2 Espaço amostral 36
5.3 Análise das questões 38
5.3.1 Compreensão conceitual 38
5.3.2 Interface e fluxo de uso 40
5.3.3 Percepção das lições 42
5.4 Discussão e implicações 46

6 Conclusão **49**
6.1 Considerações finais 50
6.2 Trabalhos futuros 50
6.2.1 Extensão dos recursos gráficos 50
6.2.2 Melhorias de layout e UI/UX 51
6.2.3 Maturação do material didático 51
6.2.4 Avaliação com docentes e em contexto de disciplina 51

Referências **53**

Anexo **53**

I Formulário de avaliação aplicado aos participantes **54**

Lista de Figuras

2.1 Ilustração simplificada do *graphics pipeline* (ex.: Vulkan API). Em amarelo, os estágios programáveis (*shaders*); em verde, estágios de *fixed function*. Fonte: https://en.wikipedia.org/wiki/Shader. 6

3.1 Visão geral da tela única da aplicação. (1) Painel de teoria e ações de projeto (coluna esquerda); (2) editor com abas *Input Geometry*, *Vertex Shader* e *Fragment Shader* (direita, parte superior); (3) área de *preview* com canais de textura adjacentes (direita, parte inferior). 19
3.2 Detalhe da aba *Input Geometry*. Controles para seleção da geometria de entrada (por exemplo, *plane*, *box*), subdivisões, modo *wireframe*, cor de fundo do *canvas* e parâmetros de câmera (tipo, posição, plano *near*/*far*). . 20
3.3 Aba *Vertex Shader*. Edição do código em GLSL responsável pelas transformações de vértices e encaminhamento de atributos ao FS. 20
3.4 Aba *Fragment Shader*. Edição do código em GLSL responsável pelo cálculo da cor final dos *fragments*, combinando luz, materiais e texturas. 21
3.5 Detalhe da visualização de erros de compilação/ligação diretamente nos editores de VS e FS. As mensagens indicam estágio e, quando possível, a linha associada, apoiando a depuração de *shaders*. 22

4.1 Pseudocódigo de inicialização da classe *Graphics* e registro de *callback* para logs de compilação. 26
4.2 Atualização por quadro no método `_onUpdate(time)`: sincronização de *uniforms* dinâmicos e renderização da cena. 27
4.3 Injeção automática de *uniforms* didáticos via `onBeforeCompile` e preservação de valores prévios (por exemplo, texturas) entre compilações. 28
4.4 Normalização da posição do mouse no *canvas* para o intervalo $[-1, 1]^2$, com origem no centro. 29
4.5 Inicialização do painel de texturas: associação entre *slots* da interface e os *uniforms* `u_texture0..u_texture3`. 30

4.6 Conversão de arquivo local em textura three.js usando `URL.createObjectURL` e atualização do objeto `Texture`. 32
4.7 Interceptação de `gl.compileShader`: captura de logs e redirecionamento para rotina interna de compilação. 33

5.1 Distribuição dos participantes quanto ao contato prévio com *shaders*. . . . 37
5.2 Distribuição dos participantes quanto à experiência geral em programação. 37
5.3 Distribuição dos participantes quanto à confiança em álgebra. 38
5.4 Distribuição das respostas para o Item 1 do questionário. 38
5.5 Distribuição das respostas para o Item 2 do questionário. 39
5.6 Distribuição das respostas para o Item 3 do questionário. 39
5.7 Distribuição das respostas para o Item 4 do questionário. 40
5.8 Distribuição das respostas para o Item 5 do questionário. 40
5.9 Distribuição das respostas para o Item 6 do questionário. 41
5.10 Distribuição das respostas para o Item 7 do questionário. 41
5.11 Distribuição das respostas para o Item 8 do questionário. 42
5.12 Distribuição das respostas para o Item 9 do questionário. 42
5.13 Distribuição das respostas para o Item 10 do questionário. 43
5.14 Distribuição das respostas para o Item 11 do questionário. 43
5.15 Distribuição das respostas para o Item 12 do questionário. 44
5.16 Distribuição das respostas para o Item 13 do questionário. 44
5.17 Distribuição das respostas para o Item 14 do questionário. 44
5.18 Distribuição das respostas para o Item 15 do questionário. 45
5.19 Distribuição das respostas para o Item 16 do questionário. 45
5.20 Distribuição das respostas para o Item 17 do questionário. 46
5.21 Distribuição das respostas para o Item 18 do questionário. 46

OpenGL ES Shading Language Normalized Device Coordinates Physically Based Rendering

Lista de Abreviaturas e Siglas

API Application Programming Interface.

CG Computação Gráfica.

FS Fragment Shader.

GLSL OpenGL Shading Language.

GLSL ES Open Embedded Systems Graphics Library.

GPU Graphics Processing Unit.

NDC Normalized Device Coordinates.

OpenGL Open Graphics Library.

OpenGL ES Open Embedded Systems Graphics Library.

SPA Single-page application.

UI User Interface.

UX User Experience.

VS Vertex Shader.

WebGL Web Graphics Library.

Capítulo 1

Introdução

Este documento apresenta o desenvolvimento, a implementação e a avaliação inicial de uma ferramenta interativa para apoiar ao ensino de *shaders* em Computação Gráfica. A proposta central é aproximar, em uma única interface, o texto explicativo, o código executável e a visualização gráfica, de forma que o estudante possa experimentar o pipeline programável enquanto lê sobre ele. A sequência de ideias ao longo do texto busca permitir que o leitor compreenda (i) de onde vêm os *shaders* na história da Computação Gráfica, (ii) por que seu ensino apresenta desafios específicos e (iii) como a ferramenta proposta procura responder a parte desses desafios, bem como quais resultados iniciais foram observados com seu uso.

1.1 Contexto: da Computação Gráfica aos shaders

Desde as primeiras aplicações interativas e sistemas de visualização científica até jogos digitais e efeitos visuais, a Computação Gráfica consolidou-se como um campo em que modelos matemáticos e implementações eficientes se encontram na GPU (Marschner and Shirley, 2015, pp. 3–7). Em seus estágios iniciais, tais sistemas eram estruturados em torno de uma *pipeline* de renderização fortemente fixa, em que transformações, iluminação e mapeamento de texturas eram descritos por conjuntos de estados configuráveis, mas não por programas livremente definíveis pelo usuário (Hughes et al., 2014, pp. 35–48).

Com a evolução do hardware gráfico, essa pipeline passou por uma transição fundamental: a introdução de estágios programáveis, em particular os *vertex shaders* e *fragment shaders*, que permitiram aos desenvolvedores descrever, em linguagens específicas, como vértices e fragmentos (potenciais pixels) deveriam ser processados (Rost et al., 2009, pp. 1–10). No ecossistema OpenGL, essa capacidade se materializa na GLSL, enquanto no contexto Web essa mesma filosofia é trazida para o navegador via WebGL (Angel and Shreiner, 2014, pp. 25–40). Em ambos os casos, o programador deixa de apenas “configu-

rar" a pipeline fixa e passa a escrever diretamente o código que a GPU executa em larga escala.

Essa transição abre espaço para uma riqueza muito maior de efeitos visuais, mas também desloca para o estudante a necessidade de compreender, de forma mais explícita, como as diferentes etapas do pipeline se encadeiam e como conceitos de álgebra linear, mapeamento de texturas e sistemas de coordenadas se convertem em código de *shader* (Hughes et al., 2014, pp. 6–11).

1.2 Origem e ensino de shaders

Do ponto de vista conceitual, os *shaders* nascem da necessidade de generalizar e flexibilizar modelos de iluminação e de material em gráficos interativos. Ao invés de uma lista rígida de modos de iluminação, linguagens como a GLSL permitem especificar funções de sombreado que vão desde modelos clássicos de *Phong* até formulações mais próximas de *Physically Based Rendering* (Marschner and Shirley, 2015, pp. 73–90). Historicamente, obras de referência em Computação Gráfica incorporaram essa mudança de paradigma descrevendo o pipeline programável como elemento central da prática contemporânea, tanto em tempo real quanto em renderização offline (Hughes et al., 2014, pp. 6–13).

No ensino, essa mudança implicou um deslocamento gradual de abordagens baseadas em OpenGL de função fixa para abordagens orientadas a *shaders*. Talton e Fitzpatrick, por exemplo, descrevem uma disciplina centrada na *OpenGL Shading Language*, argumentando que expor o aluno diretamente ao pipeline programável favorece uma compreensão mais fiel da prática atual da área (Talton and Fitzpatrick, 2007, pp. 259–260). De forma semelhante, Fink, relatam o uso de um *renderer* por *software* baseado em *shaders* como ferramenta didática para tornar visíveis todas as etapas de um pipeline moderno, do espaço do objeto à imagem final (Fink et al., 2012, pp. 73–75).

Mais recentemente, se discute explicitamente o papel de WebGL no ensino, defendendo que a disponibilidade de um ambiente de programação gráfica diretamente no navegador reduz a barreira de entrada e facilita a adoção de materiais interativos em disciplinas de graduação (Angel, 2017, pp. 106–108). Em paralelo, tutoriais on-line como de Vries (2020) popularizaram o aprendizado autodirigido de *shaders*, fornecendo exemplos executáveis e explicações textuais apoiadas em código.

Nesse cenário, o desafio deixa de ser apenas "explicar o que é um *shader*", e passa a incluir questões de design instrucional: como apresentar o pipeline programável de forma incremental, como apoiar a transição da matemática para o código e como criar ambientes em que o aluno possa experimentar com segurança os efeitos de suas modificações.

1.3 Desafios no ensino em shaders

Embora o uso de *shaders* seja hoje onipresente em aplicações de gráficos 3D, o ensino desses conceitos ainda é frequentemente percebido como difícil por estudantes que se deparam simultaneamente com:

- A necessidade de dominar sintaxe e semântica de uma linguagem específica (GLSL);
- A compreensão de múltiplos espaços de coordenadas (objeto, mundo, visão, NDC, UV) e suas transformações;
- A relação entre código em *shaders*, estados da API gráfica e dados de entrada (geometrias, texturas, *uniforms*).

Relatos na literatura educacional em Computação Gráfica sugerem que parte dessa dificuldade decorre da fragmentação entre materiais teóricos, exemplos de código e ambientes de execução. De forma que muitos cursos exigem do aluno a configuração de projetos relativamente complexos em C/C++ e OpenGL antes mesmo que ele possa começar a experimentar com o pipeline programável (Talton and Fitzpatrick, 2007, pp. 259–260). Por sua vez, a importância de ambientes em que seja possível isolar etapas do pipeline para estudo, evitando que detalhes de API ofusquem o raciocínio conceitual (Fink et al., 2012, pp. 73–75).

A popularização de WebGL e de ambientes *in-browser* abre a possibilidade de aproximar esses elementos em uma única interface, mas não resolve automaticamente o problema de como estruturar a experiência de ensino: ainda é necessário desenhar lições, definir exercícios e pensar em mecanismos de feedback que ajudem o aluno a articular o que lê, o que escreve em código e o que observa na tela.

1.4 Objetivos e escopo do trabalho

Neste contexto, o presente trabalho tem como objetivo geral conceber, implementar e avaliar uma ferramenta web para apoio no ensino de *shaders* que:

- Apresente, em uma única tela, o texto da aula, o código de VS/FS e a visualização em tempo real;
- Permita a configuração básica da cena (geometria, câmera, cor de fundo) e o uso de texturas por meio de canais dedicados;
- Suporte um formato de projeto capaz de empacotar código, parâmetros e material textual em um único artefato reutilizável;

- Seja acessível diretamente no navegador, sem instalação local, facilitando tanto o uso em laboratório quanto o estudo remoto.

O escopo técnico concentra-se na pipeline programável de VS e FS, em geometrias canônicas (plano e caixa) e em quatro canais de textura 2D. Do ponto de vista pedagógico, o trabalho foca em lições introduzindo conceitos fundamentais (coordenadas, texturas, *uniforms* e interação simples) e em um estudo exploratório com estudantes, sem ainda abranger o uso sistemático da ferramenta por docentes ao longo de uma disciplina inteira.

1.5 Organização do texto

Além deste capítulo introdutório, o texto está organizado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica, cobrindo conceitos de pipeline gráfico, *shaders*, sistemas de coordenadas e aspectos de ensino de Computação Gráfica; o Capítulo 3 descreve a ferramenta do ponto de vista do usuário, detalhando objetivos, público-alvo, cenários de uso, a interface de tela única e o formato de projeto; o Capítulo 4 aprofundar a implementação, discutindo a arquitetura de módulos, o ciclo de compilação/ligação, o gerenciamento de *uniforms* e o empacotamento em `.chroma`; o Capítulo 5 apresenta o estudo exploratório com usuários, explicitando a metodologia adotada, o perfil dos participantes e a análise quantitativa e qualitativa das respostas; por fim, o Capítulo 6 reúne as conclusões, limitações e direções de trabalhos futuros, incluindo extensões de recursos gráficos, melhorias de UI/UX e estudos mais amplos com docentes e disciplinas completas.

Capítulo 2

Fundamentação Teórica

Esta seção apresenta os conceitos essenciais de Computação Gráfica (CG) que sustentam o desenvolvimento da ferramenta web de aprendizagem de *shaders* proposta neste trabalho. Na Seção 2.1, revisamos o panorama histórico do *graphics pipeline* moderno, destacando a evolução do modelo de função fixa para o *pipeline* programável e o papel central dos *shaders*. Na Seção 2.2, detalhamos o fluxo de dados no *pipeline* programável, com ênfase em *vertex* e *fragment shaders*, espaços de coordenadas e recursos como texturas e *z-buffer*. Na Seção 2.3, discutimos desafios recorrentes no ensino de CG e estratégias didáticas associadas. A Seção 2.3.2 resume ferramentas correlatas usadas no ensino e na prática de *shaders*, situando a proposta em relação ao estado da arte. Por fim, a Seção 2.3.3 sintetiza lacunas e oportunidades que motivam a ferramenta apresentada.

2.1 Panorama, origem e aplicações dos *shaders*

Computação Gráfica (CG) trata de algoritmos e representações para gerar e manipular imagens. O percurso básico da descrição geométrica à imagem final é tradicionalmente organizado em um *pipeline* cuja visão conceitual é recorrente na literatura (Hughes et al., 2014, p. 14–16). Nas primeiras APIs prevaleceu o *fixed function pipeline*, em que transformações, iluminação e combinações de textura eram configuradas por estados pré-definidos. A introdução de linguagens de *shader* (como GLSL/GLSL ES) substituiu estágios fixos por pequenos programas executados na GPU, deslocando o foco para um modelo programável mais expressivo e didaticamente alinhado ao aprendizado baseado em experimentação (Hughes et al., 2014, p. 15).

2.1.1 Aplicações e impacto

Em linhas gerais, o *pipeline* atual contempla: (i) preparação da cena e das transformações; (ii) processamento geométrico; (iii) rasterização e *shading*; e (iv) operações de saída (*blending*, *depth test*, etc.) (Angel and Shreiner, 2014, p. 35).

Como sumariza a Figura 2.1, cada estágio recebe dados, aplica transformações e encaminha resultados ao próximo, até a composição final no *framebuffer*. Nos tópicos seguintes, são detalhados os principais estágios desse fluxo.

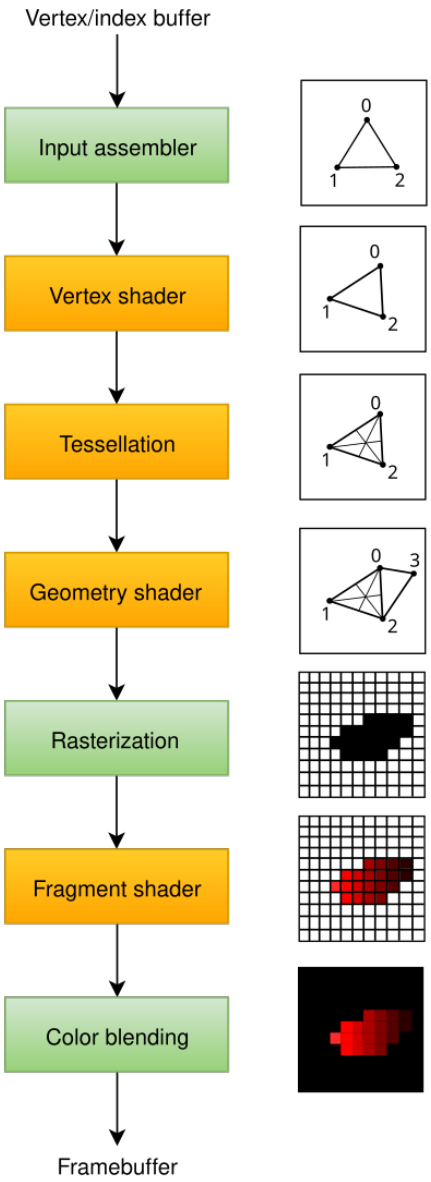


Figura 2.1: Ilustração simplificada do *graphics pipeline* (ex.: Vulkan API). Em amarelo, os estágios programáveis (*shaders*); em verde, estágios de *fixed function*. Fonte: https://en.wikipedia.org/wiki/Shader.

Input Assembler. Agrupa *buffers* de vértices e índices, interpreta a *topology* (triângulos, linhas, tiras), pode aplicar *primitive restart* e envia lotes de vértices ao VS. Não é programável; erros de configuração nessa fase costumam explicar "triângulos quebrados" logo na entrada (Hughes et al., 2014, pp. 1044–1049).

Vertex Shader (VS). Programável. Executa por vértice. Converte posições do espaço do objeto para *clip space* usando as matrizes M, V e P; pode transformar normais, gerar coordenadas UV sintéticas e preparar variáveis `out` que serão interpoladas para o FS. É o lugar indicado para posicionamento, *skinning* e manipulação de atributos por vértice (Hughes et al., 2014; Angel and Shreiner, 2014, pp. 932–937).

Tessellation. Programável. Subdivide primitivas grosseiras em malhas mais densas controladas por fatores de *tessellation*. Em APIs que o expõem (por exemplo, Vulkan), é composto por estágios de controle e avaliação, permitindo refino e deslocamento de superfícies com custo de *shading* controlado.

Geometry Shader. Programável. Opera por primitiva (triângulo, linha, ponto), podendo emitir zero ou mais novas primitivas. Útil para gerar *billboards*, silhuetas, *shadow volumes* ou depuração de *wireframe*, embora arquiteturas modernas recomendem uso parcimonioso por custo (Hughes et al., 2014, pp. 927–932).

Rasterization. Converte primitivas em *fragments* (candidatos a *pixels*), aplicando recorte (*clip*), *viewport* e regras de cobertura (amostragem por *pixel* ou *multisample*). Nesta etapa, a GPU interpola automaticamente as variáveis `out` do VS (e estágios anteriores) para cada *fragment* (Hughes et al., 2014; Angel and Shreiner, 2014, pp. 418–429).

Fragment Shader (FS). Programável. Executa por *fragment*. Amostra texturas, combina luz e materiais e calcula a cor (e, opcionalmente, a profundidade). É onde se implementam BRDFs, mapeamento de normais, *fog*, ajustes de material e efeitos dependentes de câmera (Hughes et al., 2014, pp. 711–743).

Color Blending / Output Merger. Aplica testes de profundidade e *stencil*, além da mistura de cores (*blending*) com o conteúdo já presente no *framebuffer*. Transparências, *premultiplied alpha* e composições leves são configuradas aqui, garantindo visibilidade correta e o resultado final exibido (Hughes et al., 2014, pp. 1034–1040).

2.1.2 Origem e evolução

A evolução dos *shaders* acompanha o amadurecimento do hardware gráfico e das próprias APIs de Computação Gráfica. Nos primeiros sistemas interativos, o *pipeline* gráfico era predominantemente de função fixa: a aplicação configurava matrizes de transformação, parâmetros de iluminação e estados de textura, enquanto a GPU executava uma sequência rígida de estágios pré-definidos (Hughes et al., 2014, pp. 7–9, 14–16). Esse arranjo permitia abstrair detalhes de hardware, mas restringia o controle sobre como cada vértice e cada *fragment* eram efetivamente processados.

Com a popularização do hardware programável no início dos anos 2000, surgem os primeiros *vertex* e *fragment shaders* em APIs como OpenGL e Direct3D. Nessa fase, o *pipeline* mantém a mesma estrutura conceitual (montagem de primitivas, rasterização, testes de profundidade), mas seus principais estágios passam a ser pequenos programas escritos pelo desenvolvedor, o que amplia sobremaneira o espaço de técnicas possíveis (Hughes et al., 2014, pp. 927–930).

Essa transição técnica impacta diretamente os modelos de iluminação. No *pipeline* fixo, a iluminação era em grande parte limitada a variantes de Lambert e Phong embutidas na API. Com *shaders* programáveis, torna-se possível implementar modelos mais ricos, aproximando-se de formulações físicas baseadas em BRDFs e em radiometria (Marschner and Shirley, 2015, cap. 10; cap. 18).

No ecossistema OpenGL/WebGL, essa mudança se consolida por meio do GLSL (no *desktop*) e do GLSL ES (no contexto embarcado e Web). Cursos contemporâneos de CG adotam esse *pipeline* programável como base para o ensino: mesmo em exemplos introdutórios, o estudante escreve diretamente *vertex* e *fragment shaders* em GLSL, configurando matrizes de modelo, visão e projeção, além de atributos por vértice e *uniforms* globais (Angel and Shreiner, 2014, pp. 35–38, 50–55). Na Web, o surgimento do WebGL traz essa mesma filosofia para o navegador, mantendo o modelo de *pipeline* programável e a estrutura de *shaders* compatível com OpenGL ES (Angel and Shreiner, 2014, p. 455).

Por fim, a incorporação de estágios adicionais como *tessellation* e *geometry shaders* e da computação de propósito geral na GPU (*compute shaders*) estende ainda mais o papel dos *shaders* no *pipeline*. Em vez de servirem apenas para "pintar" *fragments*, esses programas passam a controlar geração de geometria, refinamento adaptativo de superfícies e avaliações mais complexas de iluminação, aproximando a renderização interativa de técnicas tradicionais de *offline rendering*. Essa trajetória, do *pipeline* fixo aos estágios programáveis atuais, é o pano de fundo sobre o qual se insere a proposta desta ferramenta de aprendizagem de *shaders*, que explora justamente a flexibilidade do modelo programável para fins didáticos.

2.1.3 Aplicações e impacto

Shaders são hoje o mecanismo central para síntese visual em tempo real e *offline*. Em jogos digitais, definem a aparência de materiais, luzes e pós-processamento; em visualização científica e médica, aceleram filtros, cortes e mapeamentos de dados volumétricos; em CAD/CAE, viabilizam inspeção de superfícies e iluminação técnica; em VFX e animação, integram *look-dev* e renderização baseada em física; e em interfaces gráficas modernas, sustentam efeitos e composições aceleradas. A ubiquidade decorre do paralelismo massivo da GPU e do contrato de dados simples entre estágios do *pipeline*, que favorece desempenho e previsibilidade mesmo em cenas complexas (Hughes et al., 2014, pp. 55–57).

2.2 Pipeline programável e fluxo de dados

No modelo programável, a GPU executa *shaders* em paralelo: em vez de processar um vértice ou um *fragment* por vez em uma única unidade, milhares de pequenas unidades de processamento executam o mesmo programa (o *shader*) sobre dados diferentes (vértices ou *fragments*) simultaneamente. Cada invocação do *vertex shader* manipula um vértice independente; cada invocação do *fragment shader* calcula a cor de um *fragment*, o que torna o *pipeline* altamente escalável para cenas com muitos triângulos e *pixels* (Hughes et al., 2014, p. 930).

Em termos conceituais, o VS transforma posições do espaço do objeto até o *clip space*, isto é, o espaço homogêneo 4D obtido após a aplicação da matriz de projeção em que o volume visível é descrito por inequações do tipo $-w \leq x, y, z \leq w$ e onde ocorre o recorte (*clipping*) das primitivas; em seguida, a divisão por w leva às *Normalized Device Coordinates* (NDC) (Hughes et al., 2014, p. 429). O FS, por sua vez, determina a cor (e, opcionalmente, a profundidade) de cada *fragment*, combinando luz, materiais e texturas (Angel and Shreiner, 2014, pp. 35–37). A aplicação fornece *attributes* (por vértice) e *uniforms* (constantes por *draw call*); o VS escreve variáveis de saída (`out`) que a rasterização interpola e o FS lê como `in` Khronos Group (2025).

2.2.1 Espaços de coordenadas, PVM e *clip space*

Objetos são definidos inicialmente no espaço do objeto (*model space*), em coordenadas locais do modelo. Para exibi-los, aplica-se a cadeia de matrizes

$$p_{\text{clip}} = P \cdot V \cdot M \cdot \begin{bmatrix} p_{\text{obj}} \\ 1 \end{bmatrix},$$

onde M (*model*) posiciona/rotaciona/escala o objeto no mundo, V (*view*) define a câmera (posição e orientação), e P (*projection*) projeta o espaço da câmera para o volume de visualização (*view frustum*), seja em perspectiva, seja ortográfico (Hughes et al., 2014, p. 429). O produto PVM é frequentemente pré-combinado em uma matriz única no lado da aplicação (`uProj * uView * uModel`) para reduzir custo de multiplicações no *shader*.

O resultado dessa multiplicação está em *clip space*: um espaço homogêneo 4D no qual o volume visível é descrito por um conjunto de inequações (tipicamente $-w \leq x \leq w$, etc.). Antes da rasterização, ocorre a divisão por w (*perspective divide*), produzindo as *Normalized Device Coordinates* (NDC), que então são mapeadas para coordenadas de tela (*viewport transform*) (Hughes et al., 2014, p. 429). Assim, "estar em *clip space*" significa já ter passado por PVM, mas ainda não ter sido normalizado por w.

2.2.2 *Vertex* e *fragment shaders*

- ***Vertex shader***: executa por vértice; recebe atributos (posição, normal, UV etc.) e produz saídas interpoláveis para o estágio de fragmentos. Tipicamente calcula p_{clip} e repassa variáveis ao FS.
- ***Fragment shader***: executa por *fragmento* gerado pela rasterização; calcula a cor final (e, opcionalmente, a profundidade), combinando luzes, materiais, texturas e efeitos de pós-processamento.

2.2.3 GLSL em linhas gerais

Na Web, *shaders* são escritos em GLSL ES, preservando a estrutura de *attributes* (dados por vértice), *uniforms* (constantes por *draw call*) e variáveis interpoladas entre estágios. Um par mínimo VS/FS ilustra o fluxo (Angel and Shreiner, 2014, pp. 85–87):

Listing 2.1: Exemplo mínimo de vertex e fragment shader em GLSL ES.

```
// Vertex shader (GLSL ES)
uniform mat4 uModel, uView, uProj;
in vec3 aPosition;
out vec2 vUv;

void main() {
  gl_Position = uProj * uView * uModel * vec4(aPosition, 1.0);
}

// Fragment shader (GLSL ES)
```

```
precision mediump float;

in vec2 vUv;
uniform float uTime;
out vec4 fragColor;

void main() {
  float shade = 0.5 + 0.5 * sin(10.0 * vUv.x + uTime);
  fragColor = vec4(vec3(shade), 1.0);
}
```

2.2.4 Texturas, amostragem e mapeamento UV

Texturas acrescentam detalhe às superfícies ao associar a cada ponto visível um valor obtido de uma imagem (ou outro arranjo de dados) indexada por coordenadas de textura UV. Em vez de armazenar toda a variação de cor ou de parâmetros de material diretamente na geometria, mapeia-se uma imagem 2D para a superfície, de modo que cada vértice receba coordenadas (u, v) em $[0, 1]^2$ e a rasterização interpole esses valores para cada *fragment*. Como as UVs raramente coincidem exatamente com o centro de um *texel*, a GPU reconstrói o valor consultado por meio de filtros (*nearest*, *linear*) e, quando a textura aparece muito reduzida, recorre a *mipmaps* (versões em baixa resolução) para mitigar *aliasing*. Modos de endereçamento como *repeat* e *clamp-to-edge* controlam o comportamento para coordenadas fora do intervalo $[0, 1]$. No FS, as amostras de textura são combinadas com o modelo de iluminação em uso, substituindo ou modulando parâmetros como cor difusa, normal ou rugosidade do material (Hughes et al., 2014, pp. 547–548).

2.2.5 *Z-buffer* e testes

O *z-buffer* (ou *depth buffer*) guarda, para cada *pixel*, a menor profundidade já registrada durante a rasterização; antes de um novo *fragment* ser aceito, o teste de profundidade compara sua profundidade com o valor armazenado e descarta automaticamente aqueles que estão "atrás", garantindo remoção de superfícies ocultas sem necessidade de ordenar explicitamente todos os polígonos da cena. Esse mecanismo, implementado em hardware, é hoje a base da maioria dos algoritmos de visibilidade e ainda viabiliza efeitos adicionais, como composição de múltiplos passes e filtros de pós-processamento baseados em profundidade. Em paralelo, o *stencil buffer* permite restringir a renderização a regiões específicas da tela (portais, espelhos, máscaras), e o *blending* controla como a cor recém-

calculada é combinada com o conteúdo já presente no *framebuffer*, sendo fundamental para transparência e outros efeitos de composição (Hughes et al., 2014, p. 393).

2.3 Ensino de Computação Gráfica

O ensino de Computação Gráfica costuma ocupar uma posição singular nos currículos de computação: é, ao mesmo tempo, uma disciplina fortemente aplicada (com resultados visuais imediatos) e conceitualmente densa, exigindo que o estudante articule álgebra linear, geometria analítica, estruturas de dados e programação de baixo nível em APIs gráficas. Diferentes relatos de experiência convergem ao apontar que escolhas de abordagem (*pipeline* fixo vs. programável, OpenGL vs. WebGL, uso de *shaders* desde o início) impactam diretamente a motivação e o desempenho dos alunos (Angel, 2017, pp. 106–108). Nesta seção, sintetizamos alguns desafios recorrentes e destacamos características de ferramentas que têm sido usadas para aproximar teoria e prática no aprendizado de *shaders*.

2.3.1 Desafios no aprendizado

O ensino de CG envolve múltiplos pré-requisitos (álgebra linear, geometria analítica, cálculo vetorial e noções de programação paralela/assíncrona) e tipicamente apresenta desafios cognitivos e logísticos bem documentados na literatura de educação em gráficos por computador (Angel, 2017; Fink et al., 2012, pp. 106–108):

1. **Abstrações em múltiplas escalas.** O estudante alterna entre matemática (matrizes/quaternions), sistema gráfico (*pipeline*, *state machine*) e implementação (GLSL, APIs, depuração).
2. **Carga cognitiva e latência de feedback.** Muitas soluções didáticas exigem compilar ambiente local, o que atrasa o feedback visual e desloca o foco do conceito para a ferramenta.
3. **Depuração pouco intuitiva.** Erros de compilação de *shader* podem ser enigmáticos para iniciantes; variáveis *uniform* inconsistentes, espaços de coordenadas confundidos e precisão inadequada são causas comuns.

Estratégias didáticas úteis vistas na literatura incluem aprendizagem ativa (modificar o código e ver o efeito), *scaffolding* com tarefas graduais, *microlições* com objetivos claros e feedback rápido, além de exemplos mínimos executáveis para focar um único conceito por vez (Angel, 2017, pp. 110–111).

2.3.2 Ferramentas de apoio ao ensino/aprendizagem de *shaders*

Diversas plataformas em navegador têm buscado aproximar teoria e prática de *shaders*, oferecendo editores integrados, exemplos executáveis e, em alguns casos, trilhas conceituais. A seguir, descrevemos brevemente as que inspiraram este trabalho (descrições sumárias baseadas na observação direta de uso).

The Book of Shaders. Curso interativo voltado a fundamentos de *fragment shaders*, com textos curtos e exercícios executáveis no navegador. A proposta didática facilita a entrada no tema, embora o escopo permaneça majoritariamente centrado no estágio de fragmentos.

Shader Learning. Ambiente que combina material explicativo com um editor WebGL, permitindo praticar tanto *vertex* quanto *fragment shaders*. A curadoria guiada favorece o estudo progressivo; por outro lado, nem todo conteúdo está aberto e a autoria de novas lições pelo usuário não é o foco da plataforma.[1]

GLSL Shader Editor. Editor direto no navegador para experimentar GLSL rapidamente, com suporte a estágios de vértice e fragmento e opções de geometrias prontas. É prático para prototipagem e demonstrações, ainda que traga menos contexto teórico.[2]

Shadertoy. Comunidade e *playground* de referência para *fragment shaders*, repleto de exemplos e técnicas visuais. É excelente para exploração criativa e estudo de padrões, mas, por ser focado no estágio de fragmentos e na experimentação livre, exige que a fundamentação conceitual venha de fontes externas.[3]

CG Guide. Coleção de *gizmos* e aparatos para visualização de conceitos de CG em OpenGL, útil para consolidar intuição sobre transformações, luz e materiais. A ênfase está na experimentação prática, com pequenos exemplos focados em um conceito por vez.[4]

2.3.3 Discussão

O levantamento acima evidencia um espectro entre: (i) ferramentas com forte apoio didático e experimentação *in-browser*, mas frequentemente limitadas ao estágio de fragmentos ou com curadoria fechada; e (ii) *playgrounds* ricos para prototipagem, que favorecem

[1] https://shader-learning.com/explore
[2] https://glsl-editor.web.app/
[3] https://www.shadertoy.com/
[4] https://github.com/guicattani/cgguide

a exploração, porém carecem de trilhas pedagógicas estruturadas e de mecanismos de acompanhamento do aprendizado. Relatos de experiência em disciplinas de Computação Gráfica indicam que, quando teoria e experimentação não são articuladas em um mesmo ambiente, parte dos alunos tende a se concentrar apenas no “tentar e errar” visual, enquanto outra parte fica presa à notação matemática sem conseguir relacioná-la ao comportamento do *pipeline* (Angel, 2017, pp. 106–112). Persistem, portanto, lacunas didáticas importantes.

Diante desse cenário, este trabalho tem como objetivo: "Desenvolver uma ferramenta web, integrada e reproduzível, que una teoria, edição de código e visualização com feedback imediato para apoiar o ensino e o aprendizado introdutório de shaders."

Para alcançar esse objetivo, foram definidos cinco pilares de design, orientados pelas lacunas identificadas na literatura e pelas limitações observadas nas plataformas existentes.

(1) Integração entre conceito teórico e experimento reproduzível. A ferramenta deve articular, no mesmo ambiente, a explicação conceitual e a execução do exemplo correspondente, reduzindo o custo de alternar entre materiais externos, código e resultado visual. Além disso, os experimentos precisam ser reproduzíveis: o aluno deve conseguir abrir uma lição e obter o mesmo estado inicial (código, cena e recursos), possibilitando retomar o estudo, comparar versões e acompanhar o progresso. Essa integração e reprodutibilidade reduzem a dependência de configurações complexas de projeto e diminuem a latência entre aprender um conceito e observar seu efeito visual.

(2) Suporte simultâneo a *vertex* e *fragment shaders*. Embora muitos recursos educacionais se concentrem em *fragment shaders*, o domínio do *pipeline* moderno exige contemplar também o estágio de vértices. Assim, a ferramenta deve permitir editar e executar, de forma acessível, tanto o *vertex* quanto o *fragment shader*, incluindo recursos didáticos recorrentes (por exemplo, tempo, resolução, entradas do usuário) e suporte a texturas.

(3) Autoria de lições. Além de consumir exemplos prontos, docentes e estudantes devem conseguir criar, editar e adaptar roteiros de aprendizagem. Isso inclui organizar a narrativa conceitual, fornecer código inicial, propor tarefas e definir parâmetros de cena e recursos. Ao favorecer autoria, a ferramenta reduz dependência de conteúdos fixos e amplia sua adequação a diferentes turmas, níveis e objetivos pedagógicos, indo além da mera prototipagem.

(4) Acessibilidade e portabilidade. O uso em navegador, sem instalação local, reduz barreiras de entrada e facilita a adoção em laboratórios e em ambientes heterogêneos.

(5) Depuração orientada ao estudante. A depuração em *shaders* é um dos pontos de maior fricção para iniciantes. Assim, a ferramenta deve apresentar mensagens claras e contextualizadas, além de auxiliar na identificação de erros comuns (por exemplo, *uniforms* não vinculados, incompatibilidade de tipos). O objetivo é conectar sintomas visuais a causas conceituais no *pipeline* e tornar o processo de correção mais formativo, em vez de meramente punitivo.

Em síntese, tais pilares justificam a proposta de uma ferramenta web de aprendizagem de *shaders* que combine *pipeline* completo (vértices e fragmentos), lições editáveis e uma aba de teoria integrada ao editor, reduzindo a latência entre aprender o conceito e observar o efeito visual correspondente (Angel, 2017, pp. 106–112).

Capítulo 3

A Ferramenta

Esta seção apresenta, com ênfase no encadeamento entre decisões pedagógicas e opções de implementação, a ferramenta desenvolvida para apoio ao ensino de *shaders*. O sistema concretiza-se como uma Single-page application (SPA) que concentra, em uma única vista, (i) o conteúdo teórico necessário para contextualizar cada atividade, (ii) um editor de código voltado aos estágios de **Vertex** e **Fragment** e, por fim, (iii) um *preview* que executa o programa resultante em tempo real. Tal arranjo procura reduzir a fricção entre leitura e experimentação, de modo que a passagem *teoria* → *prática* ocorra com interrupções mínimas e, portanto, com maior probabilidade de retenção conceitual.

3.1 Objetivo e contribuição

O objetivo central consiste em prover um ambiente *in-browser*, isto é, acessível diretamente no navegador, no qual cada conceito apresentado possa ser imediatamente exercitado no editor e verificado no *canvas*, de maneira controlada e reprodutível. Como contribuição concreta desta versão, destacam-se: (a) a co-localização de teoria e código em uma única tela, o que encurta o ciclo de *learning-by-doing* e desencoraja o vai-e-vem entre janelas; (b) a cobertura explícita dos estágios VS/FS, permitindo que o discente diferencie com clareza o papel de cada um no pipeline programável; (c) a configuração parametrizável da cena (geometria de entrada, câmera e cor de fundo), que, combinada a quatro canais de vrtextura (`u_texture0`–`u_texture3`), possibilita cenários experimentais comparáveis ao longo do tempo; e (d) a execução sem instalação local, o que facilita tanto sessões em laboratório quanto o estudo remoto e assíncrono.

3.1.1 Justificativa e posicionamento

Considerando o panorama de ferramentas correlatas (Seção 2.3.2), em que se observam soluções ora fortemente voltadas à experimentação de *fragment shaders*, ora estruturadas em trilhas de acesso restrito ou com pouca flexibilidade para autoria, a presente ferramenta busca um ponto de equilíbrio: ao mesmo tempo em que expõe o pipeline programável em seus dois estágios essenciais (vértice e fragmento), preserva um grau de controle suficiente sobre entradas (texturas, geometria) para que fenômenos clássicos como mapeamento UV, filtragem, e relações entre espaços de coordenadas possam ser observados de forma relativamente isolada e, portanto, didaticamente mais nítida. Em outras palavras, a ênfase reside menos na acumulação de efeitos e mais na *transparência* dos mecanismos que os produzem.

3.1.2 Cenários de uso e público-alvo

A ferramenta foi concebida com duas vertentes principais de público. De um lado, **docentes, monitores e formadores** em Computação Gráfica e áreas afins, que podem utilizá-la para registrar e organizar roteiros de aula sobre *shaders*, construir materiais customizados (com texto, código e parâmetros de cena acoplados) e compartilhá-los com diferentes turmas ao longo do tempo. De outro, **estudantes em formação** seja em disciplinas introdutórias, seja em oficinas temáticas ou em estudo autodirigido que acessam esses materiais como "projetos" executáveis, nos quais é possível experimentar variações de código e observar imediatamente seus efeitos visuais.

A ferramenta se destina, portanto, tanto a disciplinas formais de Computação Gráfica quanto a oficinas em que se privilegia um tópico específico (por exemplo, texturas e amostragem, ou transformações e enquadramento), além de cenários de autoestudo. O fluxo típico de interação, coerente com os objetivos acima, pode ser descrito como: selecionar ou carregar um projeto → ler o enunciado no painel de teoria → editar o *vertex* e/ou o *fragment shader* → executar o código → observar o *preview* e, se necessário, ajustar geometria, câmera e texturas para isolar efeitos ou comparar resultados.

3.1.3 Arquitetura geral (visão de implementação)

A ferramenta é organizada como uma *single-page application* modular, na qual a lógica de interface, edição de código e execução gráfica é separada em componentes especializados, mas coordenados por uma camada única de aplicação, em sintonia com propostas recentes de ambientes de ensino de Computação Gráfica com foco em execução *in-browser* e *feedback* visual imediato (Angel, 2017, pp. 106–108).

- **three.js** é uma biblioteca JavaScript de alto nível para gráficos 3D em navegador, construída sobre WebGL, que fornece uma abstração de cena com câmeras, malhas, materiais e luzes, reduzindo a necessidade de manipular diretamente *buffers*, estados e chamadas de desenho de baixo nível (Three.js, 2025). Nesta ferramenta, three.js é responsável por estruturar a cena, configurar câmeras e associar os *shaders* definidos pelo usuário a materiais programáveis, preservando acesso explícito a *uniforms* e *samplers*.
- **GLSL ES** é adotado nos estágios de *vertex* e *fragment*, mantendo a semântica do ecossistema OpenGL ES/WebGL (qualificadores, variáveis interpoladas, *uniforms* e *samplers*) e favorecendo a portabilidade de exemplos entre diferentes ambientes compatíveis com a especificação oficial (Khronos Group, 2025).
- **CodeMirror** é um editor de código embutível em páginas Web, que oferece realce de sintaxe, numeração de linhas, recuo automático e suporte a múltiplos modos de linguagem; aqui, é configurado para edição de *shaders* em GLSL diretamente no navegador, aproximando a experiência de escrita daquela de um ambiente de desenvolvimento tradicional (CodeMirror, 2025).
- **Tecnologias Web padrão** (HTML, CSS e JavaScript) estruturam a tela única, definem o layout responsivo e integram painel de teoria, editor de código e área de *preview*, permitindo que alterações no código ou nos parâmetros de cena sejam refletidas de maneira imediata na visualização.

Do ponto de vista do *ciclo de execução*, a interface aciona explicitamente as fases de *compilação* e *ligação* do programa de *shader*; uma vez bem-sucedidas, o pipeline de desenho é disparado com os parâmetros vigentes (geometria, câmeras, *uniforms* e texturas), de modo que o *preview* exponha, em tempo quase real, o efeito de mudanças incrementais no código, favorecendo estratégias de ensino baseadas em experimentação guiada e *feedback* rápido (Fink et al., 2012, pp. 73–75).

3.1.4 Tela única da aplicação

A interface organiza-se de forma a preservar, lado a lado, a dimensão textual (que sustenta o raciocínio conceitual) e a dimensão operacional (que materializa o experimento). A Figura 3.1 ilustra essa composição geral da tela única da aplicação (Angel, 2017, pp. 106–108).

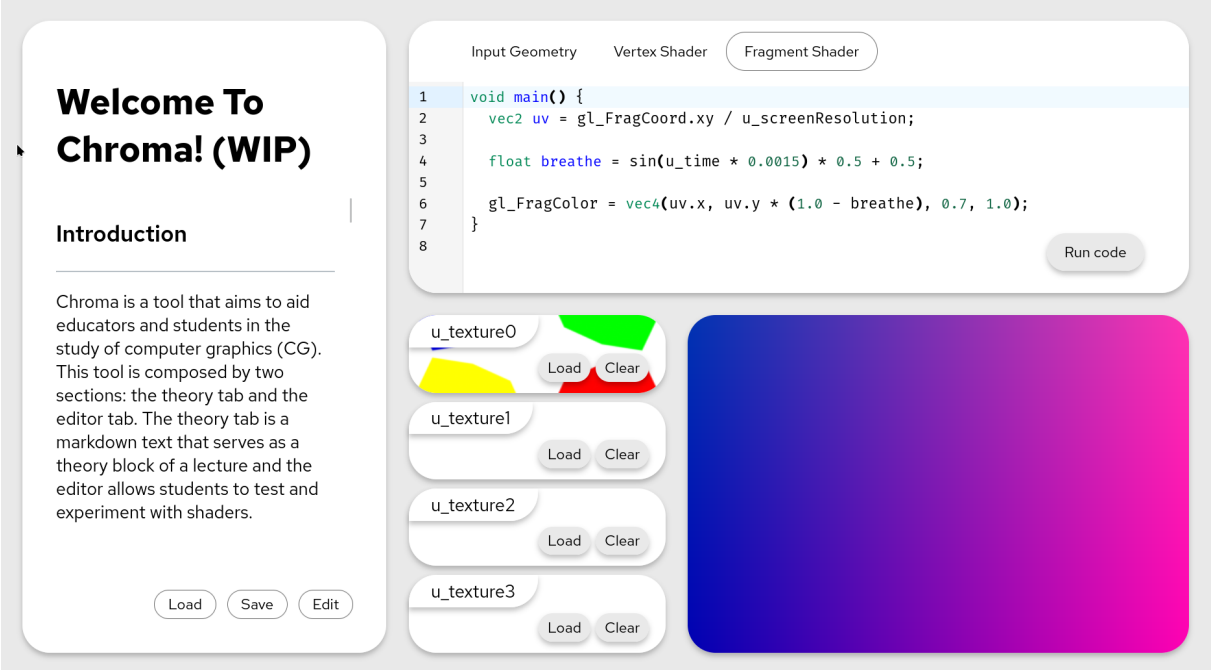


Figura 3.1: Visão geral da tela única da aplicação. (1) Painel de teoria e ações de projeto (coluna esquerda); (2) editor com abas *Input Geometry*, *Vertex Shader* e *Fragment Shader* (direita, parte superior); (3) área de *preview* com canais de textura adjacentes (direita, parte inferior).

Painel de teoria (coluna esquerda). O painel de teoria, destacado à esquerda na Figura 3.1, apresenta o material textual que acompanha cada atividade (títulos, listas, imagens e referências) de forma concisa e progressiva. Como esse conteúdo permanece visível durante a edição do código, o estudante pode, sem alternar de janela, confirmar hipóteses, revisar definições e ajustar gradualmente a solução, reforçando a articulação entre explicação conceitual e experimento prático (Angel, 2017, pp. 110–111). Na base do painel, encontram-se as ações de **carregar**, **salvar** e **editar** projeto, por meio das quais o estado corrente, composto por código, parâmetros e, quando aplicável, referências a arquivos de textura — pode ser preservado para continuidade posterior.

Editor de *shaders* (direita, parte superior). Na região superior direita da Figura 3.1, o editor organiza-se em três abas funcionais: *Input Geometry*, *Vertex Shader* e *Fragment Shader*. O usuário alterna entre a configuração da entrada geométrica e a edição dos estágios programáveis do pipeline (Angel and Shreiner, 2014, pp. 35–37).

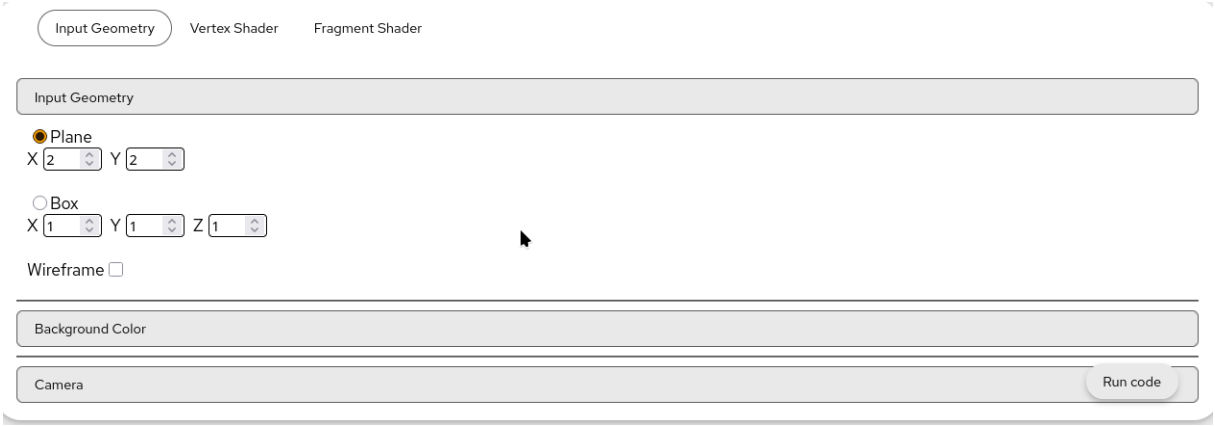


Figura 3.2: Detalhe da aba *Input Geometry*. Controles para seleção da geometria de entrada (por exemplo, *plane*, *box*), subdivisões, modo *wireframe*, cor de fundo do *canvas* e parâmetros de câmera (tipo, posição, plano *near*/*far*).

Na aba **Input Geometry** (Figura 3.2), agrupam-se controles relativos à *geometria de entrada*: seleção do tipo de primitiva (por exemplo, plano ou caixa), ajustes de tamanho e subdivisões, e ativação opcional do modo *wireframe* para visualização da malha. No mesmo painel, o usuário define a cor de fundo do *canvas* e a configuração de câmera (perspectiva ou ortográfica), incluindo parâmetros usuais como campo de visão, posição da câmera e planos *near*/*far*. Essa organização permite isolar, de maneira explícita, o papel da geometria e da câmera na formação da imagem antes de se concentrar na lógica dos *shaders*.

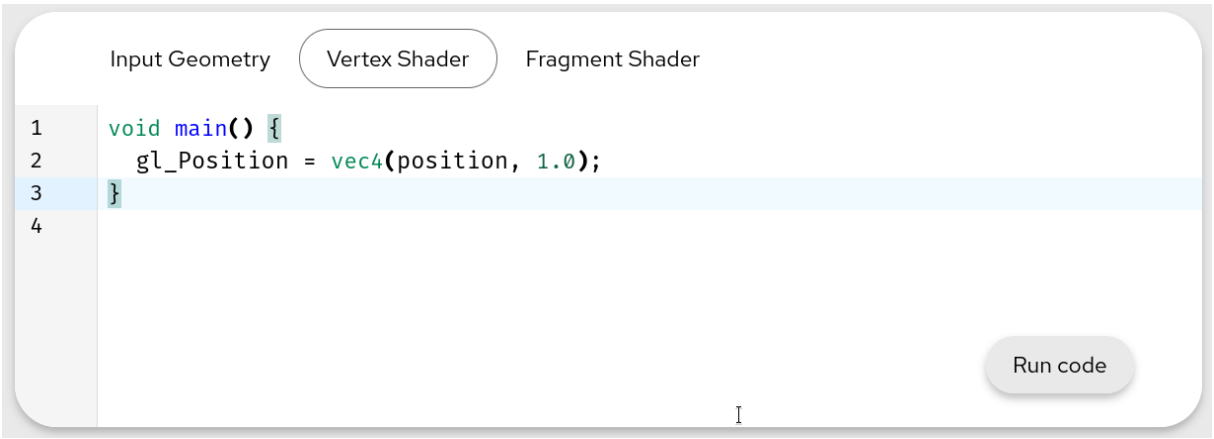


Figura 3.3: Aba *Vertex Shader*. Edição do código em GLSL responsável pelas transformações de vértices e encaminhamento de atributos ao FS.

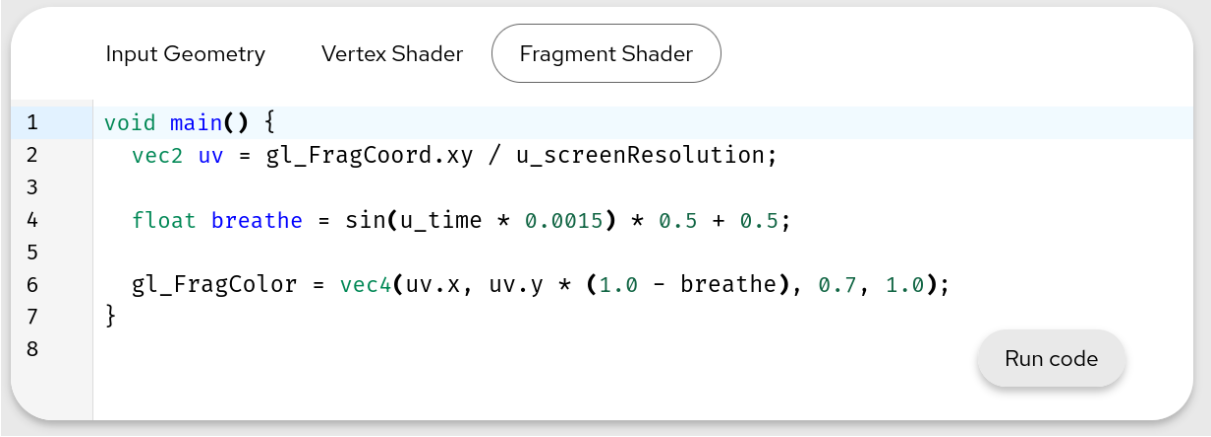


Figura 3.4: Aba *Fragment Shader*. Edição do código em GLSL responsável pelo cálculo da cor final dos *fragments*, combinando luz, materiais e texturas.

As abas **Vertex Shader** (VS, Figura 3.3) e **Fragment Shader** (FS, Figura 3.4) se destinam à edição do código em GLSL[1] correspondente a cada estágio. Em ambas as abas, o editor oferece numeração de linhas, realce de sintaxe e ações de execução. Ao acionar o comando **Run code**, a aplicação recompila e religa o programa, exibindo mensagens de erro associadas ao estágio e à linha afetada, de modo a orientar a correção de forma objetiva.

Canais de textura e *preview* (direita, parte inferior). Na parte inferior direita da Figura 3.1, o *preview*, implementado sobre GPU, reflete, *frame a frame*, as alterações do código ou dos parâmetros de cena. Ao lado do *preview* encontram-se quatro **canais de textura** (`u_texture0u_texture3`) com ações *Load/Clear*; esses canais são ligados ao FS como *samplers*, permitindo atividades envolvendo mapeamento UV, filtros e *mipmaps* sem necessidade de configuração externa adicional (de Vries, 2020). Em cenários de animação ou de interação com o ponteiro, *uniforms* usuais (por exemplo, tempo transcorrido e resolução do *canvas*) são atualizados pela aplicação e disponibilizados ao programa de *shader*.

Mensagens de erro e depuração. Em vez de um console separado, a ferramenta exibe mensagens de *compilação* e *ligação* diretamente no contexto dos editores de VS e FS, associando cada erro ao estágio correspondente e, sempre que possível, à linha afetada do código como visto na Figura 3.5. Dessa forma, problemas como erros sintáticos, incompatibilidades de tipo ou *uniforms* não vinculados são apresentados de maneira localizada,

[1]Neste trabalho, GLSL é empregado conforme as especificações oficiais do ecossistema OpenGL e OpenGL ES, disponíveis em https://registry.khronos.org/OpenGL/.

facilitando que o estudante relacione sintomas visuais a trechos específicos de código e reduzindo o custo cognitivo da depuração inicial (Fink et al., 2012, pp. 76–78).

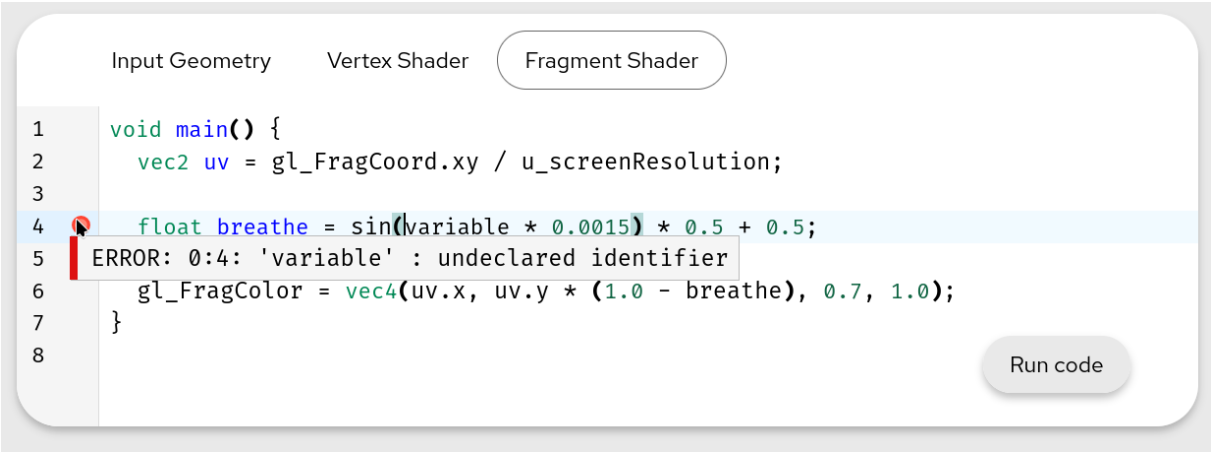


Figura 3.5: Detalhe da visualização de erros de compilação/ligação diretamente nos editores de VS e FS. As mensagens indicam estágio e, quando possível, a linha associada, apoiando a depuração de *shaders*.

3.1.5 Formato de projeto (`.chroma`) e autoria de conteúdo

Além da execução em tempo real, a ferramenta define um formato de projeto cujo objetivo é preservar, de maneira coesa e reprodutível, tanto o estado técnico (código e parâmetros) quanto o estado didático(texto da aula e recursos visuais). Adota-se, para isso, um contêiner único com extensão `.chroma`, isto é, um arquivo ZIP com estrutura interna padronizada que facilita o compartilhamento entre docentes e discentes e reduz o esforço de reconfiguração entre sessões de aula (Angel, 2017, pp. 110–111).

Conteúdo geral do pacote e manifesto `JSON`. No modelo atual, um projeto `.chroma` é composto por um arquivo central `JSON` e por pastas internas que armazenam os dados binários (*BLOBs* — do inglês *Binary Large Object*) de imagens e texturas. O `JSON` funciona como manifesto e instantâneo completo da sessão: nele estão concentrados (i) o código-fonte dos estágios de *vertex* e *fragment shader*; (ii) o texto de aula exibido no painel de teoria; (iii) a configuração da cena (geometria selecionada, parâmetros de câmera, cor de fundo e ligações de texturas); e (iv) os identificadores dos BLOBs de imagem armazenados em pastas separadas dentro do contêiner. Ao salvar, a aplicação atualiza esse documento com o estado em vigor; ao carregar um `.chroma`, o carregador lê o `JSON`, reconstitui a cena, religa cada textura ao seu *slot* correspondente e repovoa os editores de VS/FS com o código salvo. Esse desenho mantém o projeto autocontido,

mas evita inflar o manifesto com dados binários extensos, tornando o pacote mais leve e alinhado à ideia de exemplos mínimos reproduzíveis em ensino de CG (Fink et al., 2012, pp. 73–75).

Texto de aula e estado técnico integrados. Como o texto de aula é armazenado no mesmo `JSON` que concentra os *shaders* e a configuração da cena, cada projeto captura, em um único artefato, tanto a narrativa pedagógica quanto o experimento associado. Na reabertura de um `.chroma`, o painel de teoria é imediatamente preenchido com esse texto, enquanto os editores recebem os códigos de *vertex* e *fragment*; assim, explicação e código executável permanecem co-localizados, em espírito semelhante a tutoriais de OpenGL/WebGL que integram descrição e exemplos de código (de Vries, 2020).

Exportação, importação e execução imediata. Durante a exportação, o aplicativo serializa o estado corrente para o `JSON` (código, texto de aula, parâmetros de cena, referências às texturas) e empacota esse arquivo juntamente com os BLOBs de imagem em um único `.chroma`. Na importação, o processo inverso é realizado: a ferramenta lê o `JSON`, restaura geometria, câmera, cor de fundo e texturas, repõe o texto de aula e carrega os *shaders* nos respectivos editores. Em seguida, o programa de *shader* é compilado e executado automaticamente, de modo que a visualização no *preview* seja ativada assim que o projeto é aberto, encurtando o intervalo entre "abrir um exemplo" e "ver o resultado em tela".

Capítulo 4

Implementação

Esta seção aprofunda os aspectos técnicos da ferramenta apresentada anteriormente, detalhando: (i) a pilha tecnológica utilizada, (ii) a organização em módulos e responsabilidades, (iii) o ciclo *compilar* → *ligar* → *desenhar*, (iv) o uso de *uniforms* (didáticos e internos), (v) a gestão de geometrias, câmeras e texturas, (vi) o tratamento de erros, e (vii) o formato de projeto `.chroma`.

4.1 Pilha tecnológica e motivos de escolha

A aplicação é uma Single-page application (SPA) em HTML/CSS/JavaScript que combina três blocos principais:

- **three.js** como camada de alto nível para cena, câmeras e geometrias, incluindo materiais programáveis (`ShaderMaterial`). Isso permite acessar o pipeline programável de forma explícita, mas sem a verbosidade da programação WebGL de baixo nível, em linha com abordagens didáticas que utilizam bibliotecas de alto nível sobre OpenGL ou WebGL (Angel and Shreiner, 2014, pp. 35–37).
- **GLSL ES** nos estágios VS e FS, em conformidade com o modelo de pipeline programável discutido na fundamentação: o estudante escreve diretamente o código responsável por transformar vértices e por calcular a cor dos *fragments* (Angel and Shreiner, 2014, pp. 85–87).
- **CodeMirror**[1] é um componente JavaScript de código aberto que funciona como um editor de código para navegadores como editor embutido, oferecendo realce de sintaxe, numeração de linhas e integração com mensagens de erro de compilação, o que torna a experiência de edição próxima à de um ambiente de desenvolvimento completo.

[1] https://codemirror.net/

A infraestrutura de navegador (Browser APIs) é usada para seleção de arquivos, manipulação de *canvas* e criação de *BLOBs* e *object URLs* a partir de imagens, viabilizando o carregamento de texturas e a exportação de projetos inteiramente no lado cliente.

4.2 Arquitetura geral de módulos

A implementação organiza-se em quatro camadas conceituais, com responsabilidades separadas:

- **Camada de interface**: define a disposição da tela única (painel de teoria, editor, *preview* e painel de texturas), além dos controles de geometria, câmera e cor de fundo, em consonância com o arranjo descrito na Seção 3.1.4.
- **Camada de edição**: integra o editor de código (CodeMirror) às abas *Vertex Shader* e *Fragment Shader*, bem como aos campos de definição de *uniforms* visíveis ao usuário.
- **Camada gráfica**: encapsula a construção da cena three.js (renderer, cena, câmera, malhas e materiais programáveis) e o laço de animação, incluindo a gestão de geometrias, câmeras, texturas e *uniforms* internos, preservando o modelo de pipeline transformacional discutido em (Hughes et al., 2014, pp. 429–430).
- **Camada de projeto**: implementa a exportação/importação de projetos no formato `.chroma`, consolidando em um único artefato o código de *shader*, o texto da aula, os parâmetros de cena e as imagens usadas como textura.

Na inicialização, a camada de aplicação cria uma instância da classe *Graphics*, responsável pela parte gráfica, e registra um *callback* para receber logs de compilação. Em pseudocódigo simplificado, mostrado na Figura 4.1.

```
const graphics = new Graphics({
  vertexCode: getShaderSource("vertex"),
  fragmentCode: getShaderSource("fragment"),
  inputGeometryType: InputGeometryTypes.Plane,
  inputGeometryValues: [0.5, 0.5],
  backgroundColor: [0.5, 0.5, 0.5],
  cameraType: CameraTypes.Perspective,
  cameraPosition: [0.0, 0.0, 1.0],
  onShaderCompileCallback: handleShaderLogs,
});
```

Figura 4.1: Pseudocódigo de inicialização da classe *Graphics* e registro de *callback* para logs de compilação.

A função `getShaderSource` (na camada de edição) consulta o conteúdo atual do editor para cada aba (*vertex* ou *fragment*) e retorna a *string* GLSL correspondente, de modo que a camada gráfica nunca manipule diretamente o editor, apenas o código resultante.

4.3 Ciclo *compilar* → *ligar* → *desenhar*

O fluxo principal de execução é disparado pelo comando **Run code**. Em alto nível, as etapas são:

1. **Coleta do código**: a camada de edição fornece as *strings* GLSL do VS e do FS.
2. **Atualização do material**: a classe `Graphics` atualiza o `ShaderMaterial` com as novas fontes e reconfigura o conjunto de *uniforms* ativos.
3. **Compilação e ligação**: three.js, ao tentar renderizar a cena, compila e liga os *shaders* por trás da API.
4. **Desenho contínuo**: um laço de animação, configurado por `setAnimationLoop`, chama o método `_onUpdate(time)`, que atualiza *uniforms* dinâmicos (tempo, posição do mouse, resolução de tela) e desenha a cena.

O método `_onUpdate(time)` é apresentado na Figura 4.2, ilustrando a atualização de *uniforms* dinâmicos e a chamada de renderização.

```
_onUpdate(time) {
  const shader = this._mesh.material.userData.shader;
  if (shader) {
    shader.uniforms.u_time.value = time;
    shader.uniforms.u_mousePosition.value = this._mousePositionNormalized;
    shader.uniforms.u_screenResolution.value = this._screenResolution;
  }
  this._renderer.render(this._scene, this._camera);
}
```

Figura 4.2: Atualização por quadro no método `_onUpdate(time)`: sincronização de *uniforms* dinâmicos e renderização da cena.

Aqui, `time` é um contador de alta resolução fornecido pelo *runtime* de renderização; os demais *uniforms* são atualizados a partir do estado da interface (posição do mouse, tamanho do painel de *preview*).

4.4 Gestão de *uniforms*: didáticos e internos

A ferramenta diferencia dois grupos de *uniforms*: (i) **didáticos**, controlados pela interface e usados recorrentemente em exemplos (*u_time*, *u_mousePosition*, *u_screenResolution*, *u_texture0* ao *u_texture3*) e (ii) **internos**, adicionados automaticamente pelo three.js (*modelViewMatrix*, *projectionMatrix*, *cameraPosition*, entre outros), alinhados ao modelo P, V, M.

4.4.1 Injeção automática de *uniforms*.

A classe *Graphics* utiliza o gancho *onBeforeCompile* do *ShaderMaterial* para garantir que todos os *uniforms* relevantes existam e sejam declarados sem que o estudante precise escrever essas declarações manualmente. Esse gancho recebe um objeto *shader* com os campos *vertexShader*, *fragmentShader* e *uniforms*. Em código simplificado, conforme a Figura 4.3.

```
_onBeforeCompile(shader) {
  const prev = (n) =>
    this._material.uniforms && this._material.uniforms[n]
      ? this._material.uniforms[n].value
      : null;

  shader.uniforms["u_time"] = {
    type: "float",
    value: 0,
    preset: true,
  };
  shader.uniforms["u_mousePosition"] = {
    type: "vec2",
    value: new THREE.Vector2(0, 0),
    preset: true,
  };
  shader.uniforms["u_screenResolution"] = {
    type: "vec2",
    value: new THREE.Vector2(0, 0),
    preset: true,
  };

  for (let i = 0; i < 4; i++) {
    const name = "u_texture" + i;
    shader.uniforms[name] = {
      type: "sampler2D",
      value: prev(name) || null,
      preset: true,
    };
  }

  for (const [name, desc] of Object.entries(shader.uniforms)) {
    const decl = "uniform " + desc.type + " " + name + ";\n";
    shader.vertexShader = decl + shader.vertexShader;
    shader.fragmentShader = decl + shader.fragmentShader;
  }

  this._material.userData.shader = shader;
  this._material.uniforms = shader.uniforms;
}
```


Figura 4.3: Injeção automática de *uniforms* didáticos via `onBeforeCompile` e preservação de valores prévios (por exemplo, texturas) entre compilações.

O campo *preset* marca *uniforms* que não devem ser editados diretamente pelo usuário na interface. A função auxiliar *prev(name)* recupera um valor anterior (por exemplo, uma textura já carregada) para preservar o estado entre compilações.

u_time

É um contador de tempo em segundos, atualizado a cada quadro. Ele permite construir animações e efeitos temporais diretamente no *shader*, como oscilações senoidais, deslocamentos periódicos ou transições suaves, sem adicionar lógica de animação em JavaScript, como é frequente em tutoriais de OpenGL moderno (de Vries, 2020, cap. 2).

u_mousePosition

Armazena a posição normalizada do ponteiro no *canvas*, no intervalo $[-1, 1]^2$, com origem no centro. A função *_onMouseMove* converte coordenadas de tela para esse sistema, permitindo exercícios em que a posição do cursor controla parâmetros visuais (máscaras, objetos que seguem o mouse, ajustes de intensidade, etc.). O procedimento de normalização é mostrado na Figura 4.4.

```
_onMouseMove(event) {
  const rect = canvas.getBoundingClientRect();
  const p = new THREE.Vector2();

  p.x = (event.clientX - rect.left) / canvas.clientWidth;
  p.y = (event.clientY - rect.top) / canvas.clientHeight;

  p.x = p.x * 2.0 - 1.0;
  p.y = p.y * -2.0 + 1.0;

  this._mousePositionNormalized = p;
}
```

Figura 4.4: Normalização da posição do mouse no *canvas* para o intervalo $[-1, 1]^2$, com origem no centro.

u_screenResolution

guarda a largura e a altura do painel de *preview*. É atualizado sempre que a área de visualização é redimensionada, permitindo normalizar coordenadas de tela e manter a

proporção correta de padrões espaciais (faixas, grades, *noise* etc.), conforme discutido nos espaços de coordenadas e projeção.

***u_texture0*–*u_texture3*.**

Os quatro *slots* de textura expostos na interface são mapeados, no *shader*, para *uniforms sampler2D* nomeados *u_texture0*, *u_texture1*, *u_texture2* e *u_texture3*. Cada *slot* do painel permite carregar uma imagem (por seleção de arquivo ou arrasto) e associá-la a um desses *samplers*, em linha com o uso de texturas em OpenGL moderno (de Vries, 2020, cap. 9).

O código de inicialização do painel de texturas ilustra esse vínculo e é mostrado na Figura 4.5.

```
document.querySelectorAll("#textures-panel .tex-slot").forEach((slot) => {
  const name = slot.dataset.channel; // "u_texture0".. "u_texture3"
  const input = slot.querySelector('input[type="file"]');

  input.onchange = (e) => {
    const file = e.target.files[0];
    if (file) {
      fileToTex(file, (tex, url) => {
        setUniform(name, tex);
        slot.style.backgroundImage = "url(" + url + ")";
      });
    }
  };
});
```

Figura 4.5: Inicialização do painel de texturas: associação entre *slots* da interface e os *uniforms* `u_texture0..u_texture3`.

A função `setUniform` garante que *material.uniforms[name]* exista e aponte para a textura criada. Quando o *shader* é recompilado, o gancho `_onBeforeCompile` reaproveita esses valores, de forma que as texturas permaneçam ligadas ao código GLSL mesmo após mudanças no *shader*.

4.5 Geometrias, câmeras e parâmetros de cena

A aba **Input Geometry** controla os parâmetros de cena que não pertencem diretamente ao código GLSL, mas influenciam o resultado visual:

- **Geometria**: o usuário pode alternar entre um plano e uma caixa (cubo retangular), ajustando dimensões e ativando/desativando o modo *wireframe*. A função *setInputGeometryValues* reconstrói a malha a partir dos valores da interface e os atribui ao `Mesh`.
- **Câmeras**: há suporte para uma câmera em perspectiva e uma ortográfica, com campos para *near*/*far*, limites e campo de visão. *setCameraType* alterna o tipo de câmera e *setCameraValues* cria uma instância de *PerspectiveCamera* ou *OrthographicCamera* compatível com os valores escolhidos.
- **Cor de fundo**: a cor de fundo do *canvas* é aplicada ao *scene.background*, e os campos numéricos correspondentes são mantidos sincronizados com o valor atual.

Ao centralizar essas escolhas na aba *Input Geometry*, a ferramenta torna explícita a diferença entre o que é responsabilidade do *shader* (cálculo da cor do *fragment*) e o que faz parte da configuração de cena (projeção, enquadramento, escala dos objetos).

4.6 Texturas e canais `u_texture0..3`

O painel de texturas expõe quatro *slots* independentes, cada um associado a um dos *samplers* *u_texture0..3*. Para cada *slot*, a interface oferece três interações principais: carregar arquivo, limpar textura e arrastar imagens/URLs diretamente para a área de *drop*.

A função `fileToTex`, apresentada na Figura 4.6, converte um arquivo de imagem em uma textura three.js, configurando parâmetros de amostragem e atualizando o estado para uso imediato no FS.

```
function fileToTex(file, cb) {
  const url = URL.createObjectURL(file);
  const img = new Image();
  img.onload = () => {
    const tex = new THREE.Texture(img);
    tex.needsUpdate = true;
    tex.wrapS = THREE.ClampToEdgeWrapping;
    tex.wrapT = THREE.ClampToEdgeWrapping;
    tex.minFilter = THREE.LinearFilter;
    tex.magFilter = THREE.LinearFilter;
    cb(tex, url, file);
  };
  img.src = url;
}
```

Figura 4.6: Conversão de arquivo local em textura three.js usando `URL.createObjectURL` e atualização do objeto `Texture`.

A função `URL.createObjectURL` gera um object URL para o *blob* de imagem, permitindo que ele seja exibido na interface (como plano de fundo do *slot*) e usado como fonte para a textura no FS. Essa estratégia é coerente com o uso de texturas em OpenGL, em que imagens são carregadas na memória e amostradas via coordenadas UV (Hughes et al., 2014, pp. 547–548).

Quando um projeto é salvo, a camada de projeto registra, em um arquivo índice `JSON`, os metadados de cada textura (*src*, *name*, *mimeType*) e armazena o conteúdo binário das imagens como *blobs* no contêiner `.chroma`. Ao reabrir o projeto, esses *blobs* são convertidos novamente em *object URLs* e restaurados por meio do método `restoreTextureFromObjectUrl`, que recria as texturas three.js e reaplica as miniaturas na interface.

4.7 Tratamento de erros e mapeamento de linhas

Para fornecer mensagens de erro alinhadas às linhas de código visíveis no editor, a implementação intercepta a chamada original de `gl.compileShader`. Na construção da classe *Graphics*, o contexto WebGL é obtido a partir do *WebGLRenderer*. O procedimento de interceptação é apresentado na Figura 4.7.

```
const gl = this._renderer.getContext();
this._glCompileShader = gl.compileShader.bind(gl);
gl.compileShader = this._onCompile.bind(this);
```

Figura 4.7: Interceptação de `gl.compileShader`: captura de logs e redirecionamento para rotina interna de compilação.

O método `_onCompile` chama a versão original, captura o log de compilação e o código-fonte completo gerado pelo three.js (que inclui blocos injetados, como *uniforms* e código de infraestrutura). Em seguida, ele recalcula a numeração das linhas para coincidir com o código que o estudante vê no editor.

O *callback* definido pela camada de edição e responsável por destacar, no editor, a linha e o trecho de código correspondentes ao erro. Esse mecanismo ajuda o estudante a relacionar diretamente a mensagem do compilador aos sintomas visuais observados no *preview*, algo particularmente útil em contextos de ensino de *shaders* (Fink et al., 2012, pp. 76–78).

4.8 Formato de projeto (`.chroma`) e *blobs* de textura

O formato `.chroma` foi definido para encapsular, em um único pacote, todos os elementos necessários à reconstituição de uma sessão: código dos *shaders*, texto da aula, parâmetros de cena, *uniforms* definidos pelo usuário e imagens usadas como textura. Conceitualmente, trata-se de um arquivo ZIP contendo:

- um **arquivo índice** em formato `JSON`, que registra o estado do projeto (código de VS/FS, texto em Markdown, parâmetros de geometria e câmera, cores de fundo, valores de *uniforms*, metadados de texturas);
- uma pasta com **imagens de textura** serializadas como *blobs*, preservando os arquivos originais (formato, nome e tipo MIME).

Na exportação, a camada de projeto coleta o código do editor, o texto do painel de teoria, o estado de *Input Geometry* e os metadados retornados, produzindo o `JSON` e copiando o conteúdo binário das imagens para o contêiner. Na importação, o fluxo é inverso: o arquivo índice é lido, os campos são repassados à camada de edição (e à aba *Input Geometry*) e as texturas são reconstituídas a partir de *blobs*, que gera URLs temporárias.

Do ponto de vista pedagógico, esse formato permite que cada exercício ou lição seja tratado como um artefato autocontido: ao abrir um `.chroma`, o estudante retoma exatamente o código, as texturas e a configuração de cena que o autor planejou, sem depender de caminhos de arquivos locais ou de passos adicionais de preparação.

4.9 Integração teoria–editor e restauração de sessão

O texto da aula é armazenado em Markdown e exibido com formatação consistente; o código GLSL é salvo e restaurado tal como estava no momento da exportação; os parâmetros de cena e as texturas são reinstaurados a partir do arquivo índice. Assim, um projeto funciona como um *checkpoint* completo do processo de aprendizagem: o docente pode criar e compartilhar exercícios reproduzíveis, e o discente pode interromper e retomar o estudo sem perda de contexto.

Capítulo 5

Testes

Este capítulo apresenta o procedimento de avaliação da ferramenta proposta, os perfis dos participantes, os cenários de uso considerados e uma análise quantitativa e qualitativa das respostas coletadas por meio do formulário "Uso de uma ferramenta interativa para aprender shaders". Os resultados apoiam a discussão sobre a aderência da interface aos objetivos didáticos definidos no Capítulo 2.

5.1 Desenho do estudo

O estudo foi conduzido como um experimento exploratório de uso guiado da ferramenta, no qual cada participante deveria abrir uma sequência de três projetos (Lição 1, Lição 2 e Lição 3) e resolver, na ordem, os exercícios propostos em cada um deles. As lições foram concebidas para progredir em complexidade:

- **Lição 1**: foco em noções básicas de coordenadas e cor;
- **Lição 2**: introdução a texturas e amostragem;
- **Lição 3**: combinação de entrada de usuário (posição do mouse) com transformação geométrica e cor no tempo.

Cada lição foi empacotada de forma que, ao abrir o projeto na ferramenta, o participante encontrasse: (i) o texto da atividade no painel de teoria, (ii) o código inicial de VS e FS, (iii) os parâmetros de cena (geometria, câmera, cor de fundo) e (iv) as texturas e demais recursos necessários já pré-configurados. Com isso, o participante pôde iniciar a resolução do exercício imediatamente, sem precisar configurar manualmente a cena ou carregar arquivos externos.

O procedimento para cada participante seguiu as etapas abaixo:

1. **Consentimento e questionário inicial.** O participante acessou o formulário online, leu o termo de consentimento e respondeu a perguntas sobre experiência prévia com CG, programação e álgebra (se se considerava confortável com programação e com álgebra linear, e se já havia ouvido falar em *shaders*).

2. **Acesso à ferramenta.** Em seguida, recebeu o link para a aplicação Web e os três projetos correspondentes às Lições 1–3.

3. **Execução das lições.** O comando experimental foi: "Abrir a ferramenta, carregar a Lição 1, realizar o exercício até onde conseguir e, em seguida, repetir o processo para as Lições 2 e 3, nesta ordem". Todos os testes foram assistidos, mas não foram fornecidas dicas ou respostas durante a execução: o observador apenas registrou dificuldades relatadas espontaneamente e se o participante julgou ter concluído ou não cada exercício.

4. **Questionário pós-uso.** Ao final, o mesmo formulário recolheu impressões sobre dificuldade percebida, clareza da interface, utilidade do painel de teoria e sensação de aprendizado em relação a *shaders*.

Para fins de transparência e reuso, o instrumento completo utilizado na coleta de dados incluindo as 18 questões fechadas analisadas neste capítulo e as duas questões abertas (itens 19 e 20), encontra-se transcrito no Anexo I. As lições empregadas no estudo (Lição 1, Lição 2 e Lição 3), no formato de projetos da ferramenta, estão disponíveis publicamente em: https://drive.google.com/drive/u/0/folders/12mt8L0MDsH5dJTjt1evL2GaK0rYvyyp7.

A aplicação Web utilizada pelos participantes pode ser acessada em https://gustavo-tomas.github.io/Chroma/. Hospedado por meio do serviço GitHub Pages, ou seja, os arquivos estáticos da ferramenta (HTML, JavaScript, *shaders* e recursos associados) são servidos diretamente a partir de um repositório no GitHub.

5.2 Espaço amostral

Participaram do estudo **20** pessoas, todas com algum grau de familiaridade com programação. A partir das respostas ao formulário, foi possível caracterizar o grupo da seguinte forma:

- **Contato prévio com *shaders*.** 5 participantes (25,0%) nunca tiveram contato com *shaders*; 7 (35,0%) já haviam ouvido falar, mas nunca tinham programado; 7 (35,0%) já tinham feito tutoriais simples; e 1 (5,0%) relatou desenvolver *shaders* com frequência. Assim, 75,0% tinham algum contato prévio com o tema.

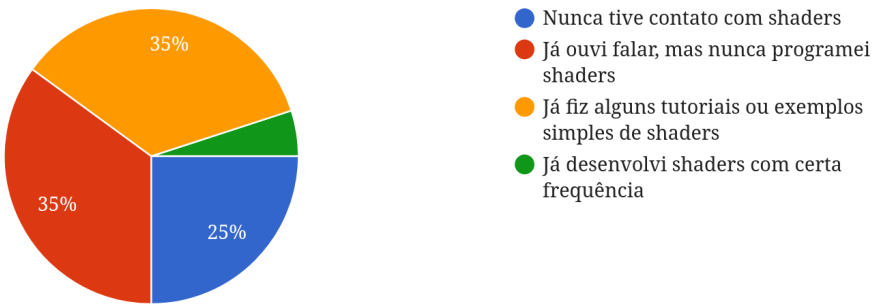


Figura 5.1: Distribuição dos participantes quanto ao contato prévio com *shaders*.

- **Experiência geral em programação.** 3 (15,0%) se classificaram como Iniciante; 10 (50,0%) como Intermediário; e 7 (35,0%) como Avançado. Cerca de 85,0% tinham ao menos nível intermediário de programação.

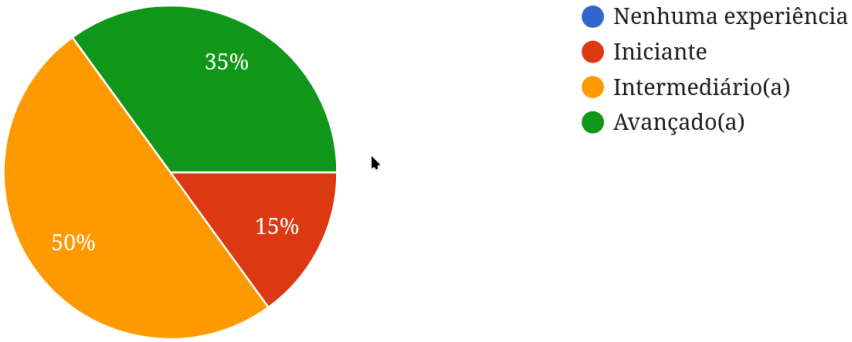


Figura 5.2: Distribuição dos participantes quanto à experiência geral em programação.

- **Fundamentos matemáticos.** 2 (10,0%) declararam ter muita dificuldade; 6 (30,0%) alguma dificuldade; 9 (45,0%) razoável conforto; e 3 (15,0%) muito conforto. Aproximadamente 60,0% sentem-se confortáveis com álgebra.

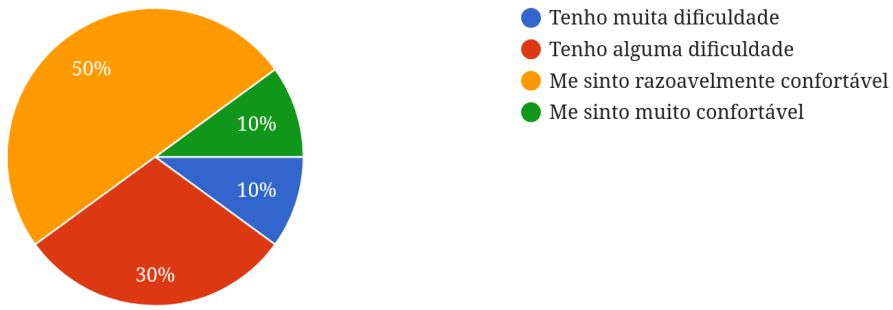


Figura 5.3: Distribuição dos participantes quanto à confiança em álgebra.

5.3 Análise das questões

A seguir são apresentadas as análises das 18 questões fechadas, agrupadas em temas.

5.3.1 Compreensão conceitual

1) Entendi o que é um shader e o papel dele no pipeline gráfico. 14 (70,0%) Concordo completamente, 5 (25,0%) Concordo e 1 (5,0%) Neutro. Total de 95,0% de concordância, indicando excelente assimilação do conceito.

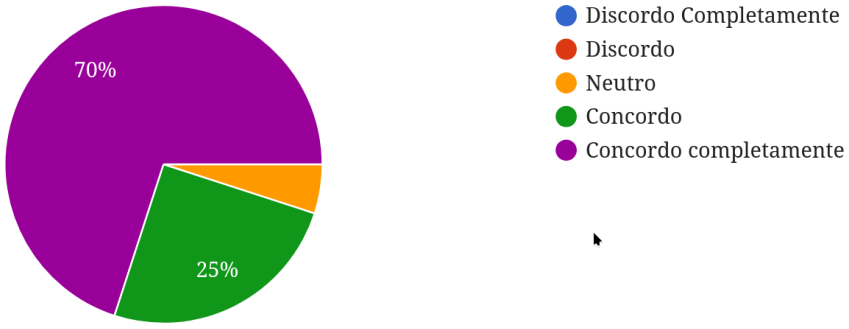


Figura 5.4: Distribuição das respostas para o Item 1 do questionário.

2) Entendi a diferença entre vertex shader e fragment shader. 6 (30,0%) Concordo completamente, 8 (40,0%) Concordo, 4 (20,0%) Neutro e 2 (10,0%) Discordo. A distinção entre VS e FS foi compreendida pela maioria, mas ainda gera alguma incerteza para iniciantes.

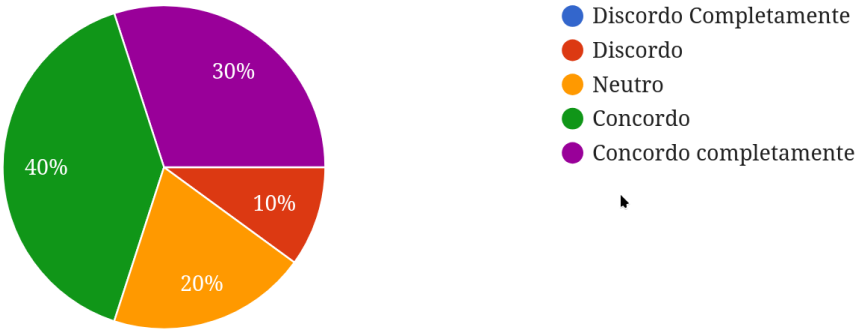


Figura 5.5: Distribuição das respostas para o Item 2 do questionário.

3) As explicações sobre coordenadas UV foram claras. 10 (50,0%) Concordo completamente, 10 (50,0%) Concordo. A clareza sobre o conceito de UV foi bem avaliada, mostrando boa didática do conteúdo.

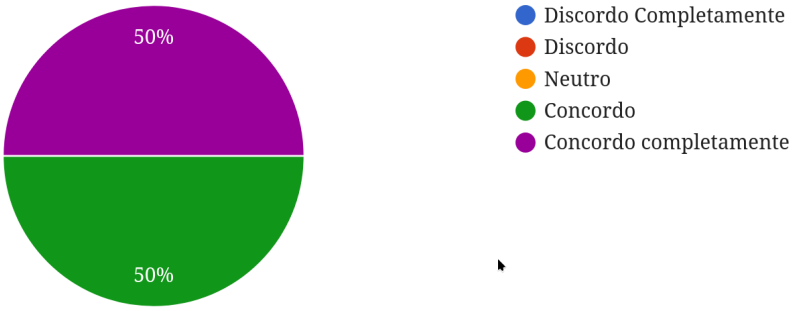


Figura 5.6: Distribuição das respostas para o Item 3 do questionário.

4) Entendi o que são uniforms (por exemplo `u_time`, `u_screenResolution`). 9 (45,0%) Concordo completamente, 5 (25,0%) Concordo, 4 (20,0%) Neutro e 2 (10,0%) Discordo. Apesar da maioria positiva, o tema "uniforms" apresentou mais dúvidas, condizente com sua abstração.

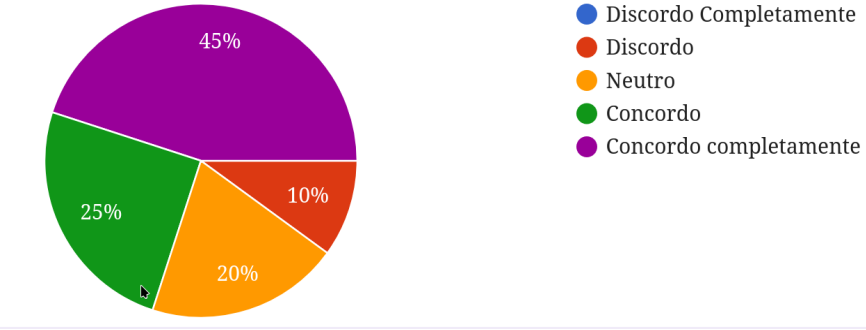


Figura 5.7: Distribuição das respostas para o Item 4 do questionário.

5) Consegui entender os exemplos de código sem precisar de muita ajuda externa. 17 (85,0%) Concordo completamente, 3 (15,0%). Indica que os exemplos foram compreensíveis.

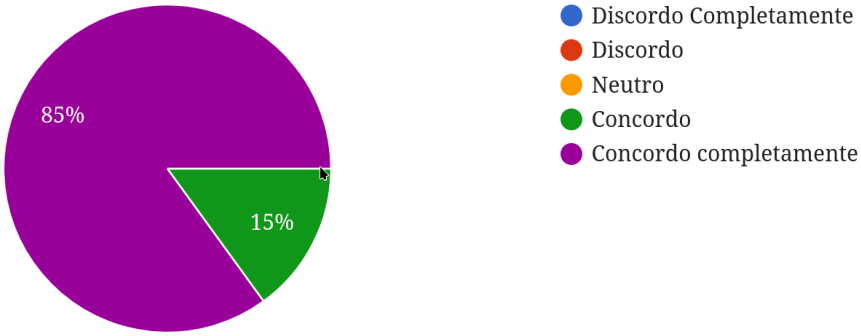


Figura 5.8: Distribuição das respostas para o Item 5 do questionário.

5.3.2 Interface e fluxo de uso

6) Foi fácil rodar os exemplos e ver o resultado dos shaders na tela. 19 (95,0%) Concordo completamente e 1 (5,0%) Concordo. A execução e o feedback visual imediato foram percebidos como simples e diretos.

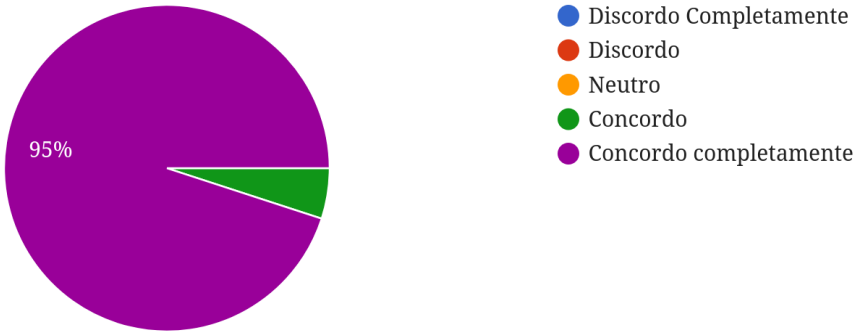


Figura 5.9: Distribuição das respostas para o Item 6 do questionário.

7) Consegui identificar rapidamente onde editar o código de vertex e fragment. 6 30,0%) Concordo completamente, 5 (25,0%) Concordo, 4 (25,0%) Neutro, 3 (15,0%) Discordo e 1 (5,0%) Discordo completamente. Mostra que a localização do código ainda causa confusão para 40,0% dos usuários, sugerindo necessidade de melhorias de UI/UX.

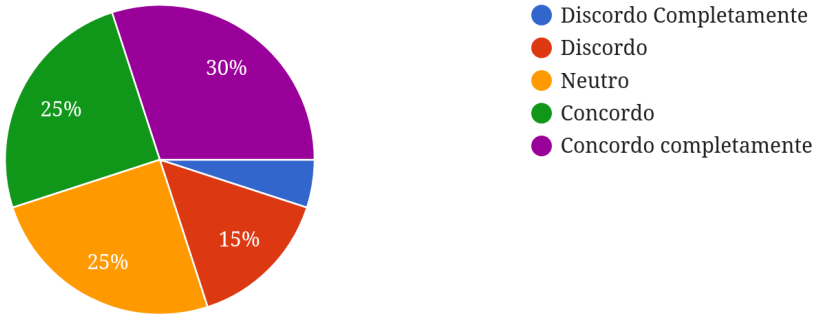


Figura 5.10: Distribuição das respostas para o Item 7 do questionário.

8) O painel de Input Geometry foi fácil de entender e usar. 12 (60,0%) Concordo completamente, 5 (25,0%) Concordo e 3 (15,0%) Neutro. Indica clareza e funcionalidade no painel de geometria, apesar de compartilhar da dificuldade das abas de Fragment e Vertex, demonstrando necessidade em melhorias de UI/UX.

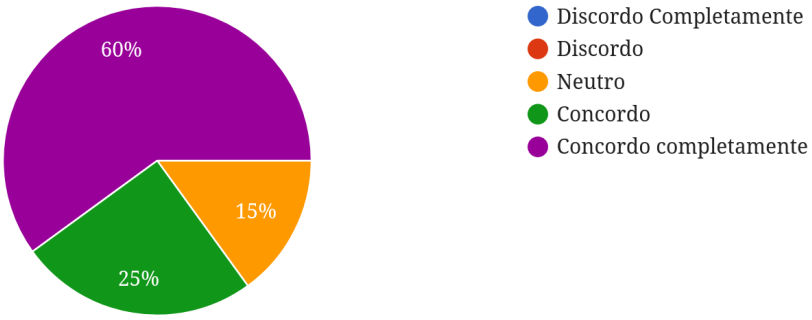


Figura 5.11: Distribuição das respostas para o Item 8 do questionário.

9) Ativar o wireframe ajudou a visualizar a geometria. 17 (85,0%) Concordo completamente, 3 (15,0%) Concordo. O recurso foi considerado útil pela grande maioria.

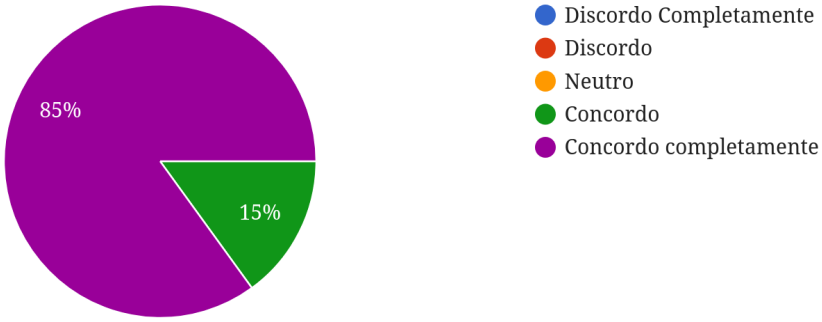


Figura 5.12: Distribuição das respostas para o Item 9 do questionário.

5.3.3 Percepção das lições

10) Entendi o objetivo principal da Lição 1. 16 (80,0%) Concordo Completamente, 3(15%) Concordo e 1 (5,0%) Discordo.Respostas positivas, mas com algumas pessoas com maior dificuldade na primeira lição para se situarem.

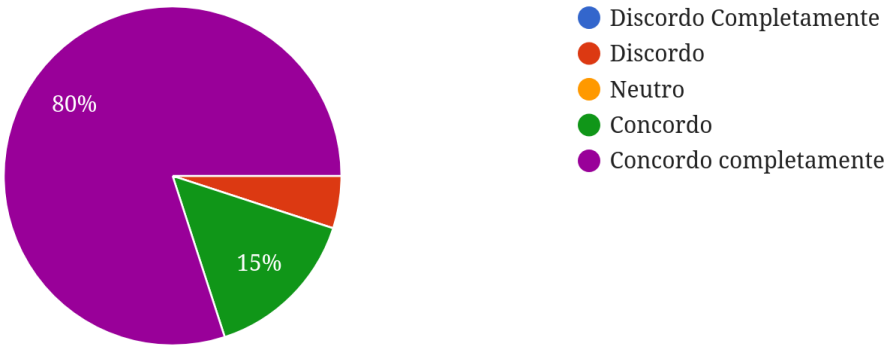


Figura 5.13: Distribuição das respostas para o Item 10 do questionário.

11) Consegui acompanhar os passos de texto + código da Lição 1 sem me perder. 14 (70,0%) Concordo completamente, 4 (20,0%) Concordo, 1 (5,0%) Neutro e 1 (5,0%) Discordo. O fluxo entre teoria e prática foi considerado bem integrado mas algumas pessoas ainda se sentiram perdidas na primeira lição.

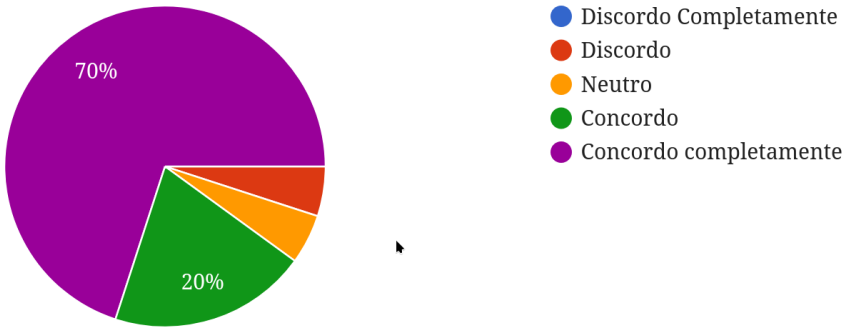


Figura 5.14: Distribuição das respostas para o Item 11 do questionário.

12) Os exercícios da Lição 1 me ajudaram a praticar e fixar o conteúdo. 16 (80,0%) Concordo completamente, 3 (15,0%) Concordo e 1 (5,0%) Discordo. Indica em suma que a Lição 1 foi em sua maioria muito bem recebido mas apresentando para poucas pessoas certa dificuldade em compreender seu fluxo de aprendizado, levando em conta o resultado das últimas questões, dá para supor que pela alta carga cognitiva com muitas informações e conteúdo já desde início pode ter prejudicado o aprendizado.

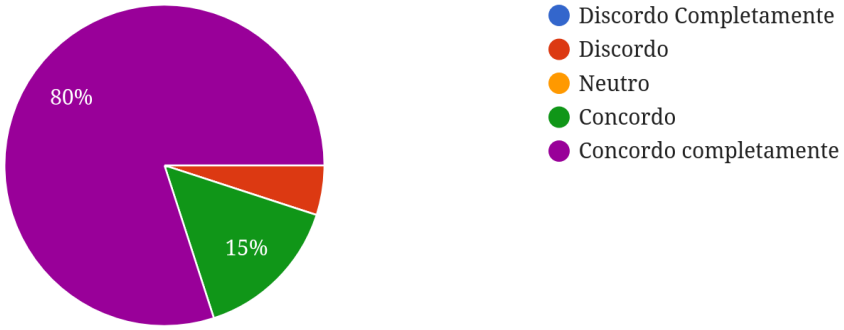


Figura 5.15: Distribuição das respostas para o Item 12 do questionário.

13) Entendi o objetivo principal da Lição 2. 16 (80,0%) Concordo completamente e 4 (20,0%) Concordo. Mantém alto índice de clareza conceitual.

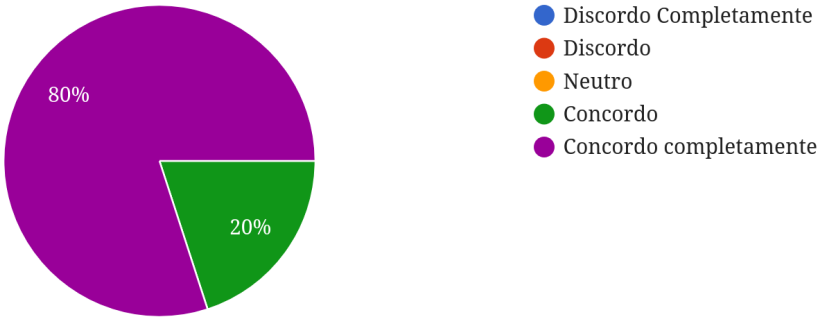


Figura 5.16: Distribuição das respostas para o Item 13 do questionário.

14) Consegui acompanhar os passos de texto + código da Lição 2 sem me perder. 17 (85,0%) Concordo completamente e 3 (15,0%) Concordo.

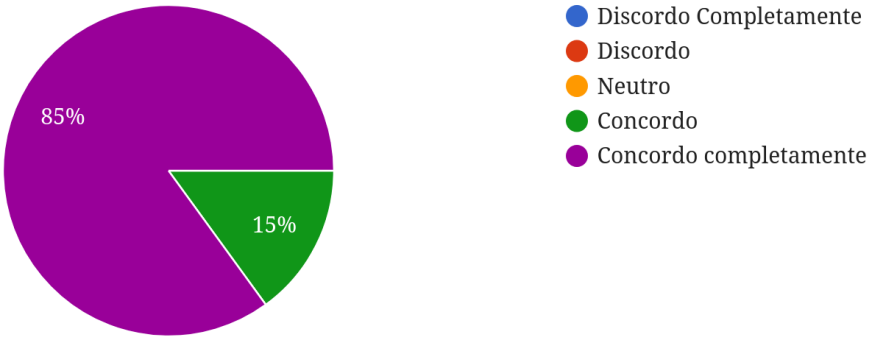


Figura 5.17: Distribuição das respostas para o Item 14 do questionário.

15) Os exercícios da Lição 2 me ajudaram a praticar e fixar o conteúdo. 16 (80,0%) Concordo completamente, 3 (15,0%) Concordo e 1 (5,0%) Neutro. Reflete que a progressão da segunda lição foi percebida como clara e útil.

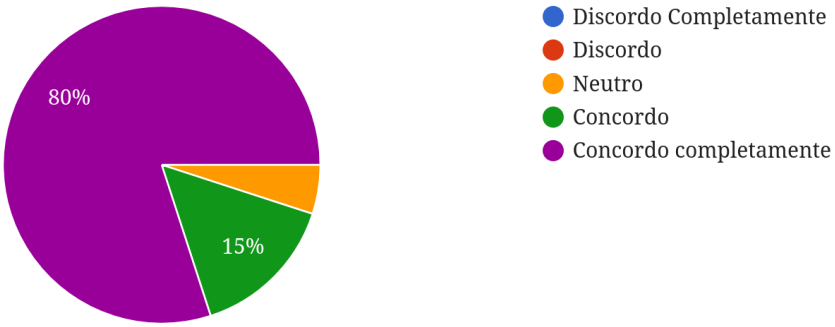


Figura 5.18: Distribuição das respostas para o Item 15 do questionário.

16) Entendi o objetivo principal da Lição 3. 16 (80,0%) Concordo completamente, 3 (15,0%) Concordo e 1 (5,0%) Neutro. Confirma que o propósito final da sequência foi compreendido.

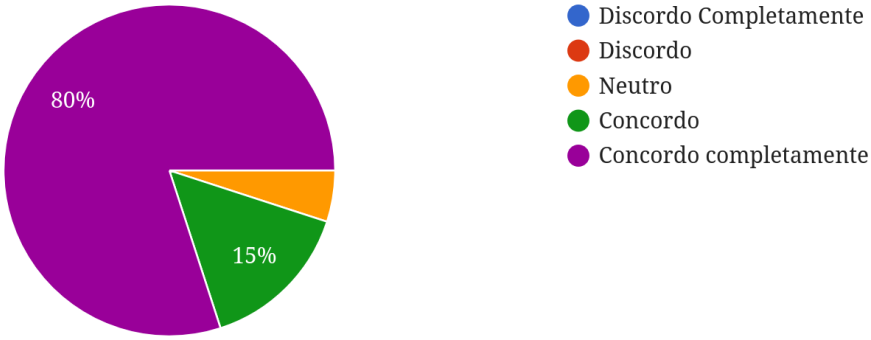


Figura 5.19: Distribuição das respostas para o Item 16 do questionário.

17) Consegui acompanhar os passos de texto + código da Lição 3 sem me perder. 17 (85,0%) Concordo completamente, 3 (15,0%) Concordo. O fluxo da terceira lição manteve boa consistência de uso.

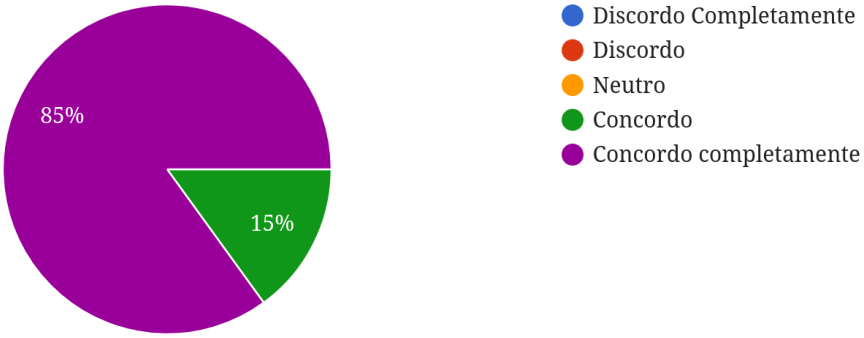


Figura 5.20: Distribuição das respostas para o Item 17 do questionário.

18) Os exercícios da Lição 3 me ajudaram a praticar e fixar o conteúdo. 18 (90,0%) Concordo completamente ,1 (5,0%) Concordo, 1 (5,0%) Discordo. Demonstra que a proposta de prática contínua teve pleno êxito.

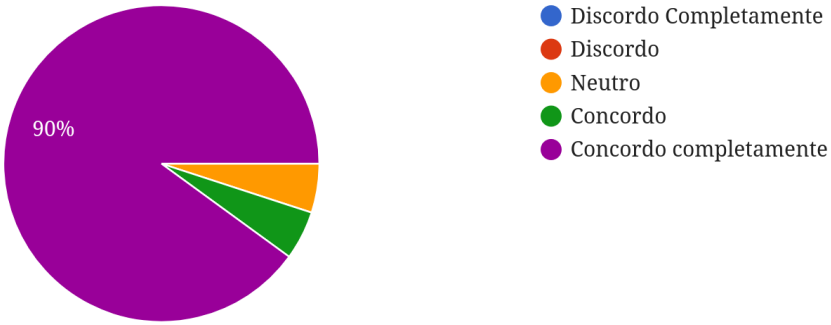


Figura 5.21: Distribuição das respostas para o Item 18 do questionário.

5.4 Discussão e implicações

Os resultados quantitativos indicam que a ferramenta cumpriu, em boa medida, seu objetivo principal de apoiar a compreensão inicial de *shaders*: a grande maioria dos participantes declarou entender o papel de um shader no pipeline gráfico e diferenciar, em algum grau, os estágios de *vertex* e *fragment*. As taxas de concordância nos itens associados aos exercícios (itens 12, 15 e 18) mostram um padrão consistente: praticamente todos os respondentes perceberam as atividades práticas como úteis para fixar o conteúdo das três lições, com proporções elevadas de "Concordo completamente" à medida que avançam na sequência. Isso sugere que, uma vez estabelecido um mínimo de familiaridade com a

ferramenta, o formato de "texto + código + preview" passa a ser sentido como natural e produtivo.

Por outro lado, a combinação dos dados com as respostas abertas revela pontos de fricção importantes, especialmente nos momentos iniciais de uso. Em vários relatos, participantes mencionam explicitamente que "demoraram a entender o que era para fazer" ou que, na Lição 1, não estava imediatamente claro qual trecho de código deveria ser editado para produzir o efeito mostrado no texto. O item 7, que trata da capacidade de identificar rapidamente onde editar o código de *vertex* e *fragment*, reforça esse quadro: uma parcela expressiva dos participantes não deu resposta claramente positiva, indicando que a principal dificuldade não era "escrever shaders em si", mas entender o fluxo básico de interação com a interface.

Em particular, as respostas sugerem que muitos usuários não perceberam, de imediato, que era necessário alternar entre as abas de *vertex shader* e *fragment shader* para localizar o código que efetivamente controlava cada parte da cena. Alguns comentários relatam que o participante chegou a clicar em vários elementos da tela ou a rolar o painel de teoria em busca de instruções mais explícitas, antes de notar que havia abas específicas para cada estágio do shader. Essa dificuldade em "descobrir" a necessidade de trocar entre as abas para editar o código foi, de forma recorrente, a principal fonte de desorientação inicial, mesmo entre participantes com boa experiência prévia em programação.

Esse conjunto de evidências aponta diretamente para a necessidade de um *onboarding* mais claro e simples. Em vez de exigir que o usuário deduza, por tentativa e erro, que deve alternar abas e editar trechos específicos, a ferramenta poderia oferecer um pequeno tutorial de navegação na primeira execução, com destaques visuais (por exemplo, caixas de diálogo sobrepostas ou setas) indicando: (i) onde estão as abas de *vertex* e *fragment*; (ii) onde o código deve ser modificado naquele momento; e (iii) como o botão de execução se relaciona com o *preview*. Esse tipo de orientação inicial, curta e focada, tende a reduzir significativamente a sensação de "não saber o que fazer" e a separar melhor as dificuldades de navegação das dificuldades conceituais de *shading*.

As sugestões qualitativas convergem em outras direções de melhoria de UI/UX. Vários participantes pedem a possibilidade de aumentar o painel de teoria, reduzir blocos textuais muito longos em passos menores, destacar com mais clareza os termos-chave e objetivos de cada exercício, e dar sinais visuais mais fortes de "é aqui que você deve editar agora". Também aparecem ideias mais avançadas, como mecanismos de destaque dinâmico entre código e visualização (por exemplo, passar o mouse sobre uma linha de código e ver a região correspondente da cena realçada no *preview*). Em termos gerais, esses comentários indicam que a interface atual é funcional, mas ainda oferece pouco apoio explícito à descoberta do fluxo de trabalho, o que pesa especialmente nos primeiros minutos de

interação.

Do ponto de vista do desenho das lições, os dados e relatos apontam que a Lição 1 concentra muitos conceitos novos, tanto de pipeline quanto de uso da ferramenta, enquanto as Lições 2 e 3 são percebidas como mais diretas e até "simples" após a superação da fase inicial. Alguns participantes mencionam que gostariam de ver mais exercícios distribuídos ao longo do texto, não apenas ao final, e que se sentiram mais seguros quando começaram a "mexer no código" em tarefas pequenas e imediatas. Isso sugere que um redesenho didático com micro-tarefas intermediárias ("modifique este valor e veja o que acontece") pode ajudar a quebrar a barreira inicial, alinhando melhor o ritmo de introdução conceitual com a curva de descoberta da interface.

Outro aspecto importante é que, nesta fase, a ferramenta foi avaliada quase exclusivamente sob a ótica do estudante. O fluxo estudado consistiu em abrir lições já preparadas, ler o texto, editar *shaders* e observar os resultados. Ainda não há dados sistemáticos sobre o uso por docentes, em especial quanto ao processo de autoria: como é, na prática, escrever o texto da aula, configurar cenas e *uniforms*, ajustar o layout dos painéis e estruturar as atividades para que o aluno não fique "sem saber o que fazer". Dado que uma contribuição central da ferramenta é justamente permitir a criação e o compartilhamento de lições customizadas, torna-se fundamental, em trabalhos futuros, investigar também esse fluxo de criação do material, em paralelo ao uso discente.

Em síntese, os resultados sugerem que o núcleo interativo da proposta de editar código, executar e ver o efeito imediatamente, é bem recebido e eficaz para fixar conceitos, mas que o primeiro contato com a interface ainda impõe uma carga cognitiva desnecessária, sobretudo na identificação de onde e como começar, em especial quanto a identificação das abas de Input Geometry, Fragment Shader e Vertex Fragment. Melhorias em UI/UX, com foco em *onboarding*, sinalização das abas e adaptação do layout, aliadas a um refinamento didático das lições iniciais e a testes específicos com docentes no papel de autores, tendem a aproximar a ferramenta do cenário ideal: um ambiente em que o estudante não precise lutar para descobrir "o que fazer", podendo concentrar sua atenção em explorar, experimentar e compreender o comportamento dos *shaders*.

Capítulo 6

Conclusão

Este trabalho apresentou o desenvolvimento e a avaliação inicial de uma ferramenta *in-browser* para apoio ao ensino de *shaders*, com ênfase na co-localização de texto, código e visualização em uma única tela. Partindo da motivação discutida na fundamentação teórica, buscou-se reduzir a distância entre explicação conceitual e experimentação prática no ensino de Computação Gráfica, em particular no que diz respeito ao pipeline programável de vertex e fragment.

Ao longo do texto, foram detalhados: (i) os objetivos pedagógicos da ferramenta, articulados em torno da ideia de *learning-by-doing*; (ii) a arquitetura geral da aplicação, construída como uma SPA com uso de GLSL, three.js e um editor embutido; (iii) o formato de projeto `.chroma`, que empacota código, parâmetros de cena, texto da aula e recursos visuais em um único artefato; e (iv) a organização da interface em torno de um painel de teoria, um editor com abas (*Input Geometry*, *Vertex Shader*, *Fragment Shader*) e um *preview* com canais de textura (*u_texture0..3*) atualizados em tempo real.

A implementação foi descrita em termos de módulos e responsabilidades: desde a captura do código no editor e sua passagem para o *runtime* gráfico, até o gerenciamento de *uniforms* didáticos (como *u_time*, *u_screenResolution* e *u_mousePosition*) e de parâmetros de cena (geometria, câmera e cor de fundo). O formato *.chroma*, em particular, foi concebido para restaurar o estado de uma sessão de forma reprodutível, permitindo que o mesmo pacote seja reaberto em diferentes momentos preservando a associação entre teoria e código.

Em suma, a ferramenta atingiu o objetivo de demonstrar a viabilidade de um ambiente integrado para ensino de *shaders* no navegador, unindo texto, código e *preview* em um fluxo contínuo de interação. A primeira rodada de testes sinaliza que o núcleo interativo (editar → executar → ver) é percebido como fluido, mas que ainda há espaço importante para amadurecer tanto o desenho da interface quanto o desenho das próprias lições.

6.1 Considerações finais

Do ponto de vista pedagógico, os resultados sugerem que a combinação de: (i) código executável em GLSL; (ii) visualização imediata no *canvas*; e (iii) instruções textuais próximas ao editor favorece a compreensão inicial de conceitos como papel do *shader*, diferença entre VS/FS e uso de *uniforms*. O formato de projetos *.chroma* adiciona a esse cenário a possibilidade de empacotar lições completas, o que é relevante tanto para continuidade de estudo quanto para compartilhamento entre docentes e discentes.

Ao mesmo tempo, a própria avaliação deixa claro que a ferramenta, na forma atual, foi exercitada principalmente sob a perspectiva do estudante, com lições preparadas previamente pelo autor deste trabalho. Ainda não se explorou de maneira sistemática como professores criam, adaptam e reutilizam conteúdos dentro da ferramenta, nem como ela se comporta quando utilizada ao longo de uma disciplina completa. Assim, os resultados devem ser lidos como uma evidência inicial de aderência da abordagem, e não como um veredito definitivo sobre seu impacto em contextos curriculares amplos.

6.2 Trabalhos futuros

Os dados coletados e as limitações identificadas ao longo deste estudo apontam uma série de desdobramentos naturais para trabalhos futuros, tanto no plano técnico quanto no pedagógico.

6.2.1 Extensão dos recursos gráficos

No plano técnico, um primeiro eixo de evolução consiste em ampliar o *vocabulário gráfico* disponível na ferramenta. Entre as possibilidades:

- **Carregamento de modelos 3D.** Permitir importar modelos externos em formatos como `OBJ` ou `glTF` diretamente para a cena, indo além das geometrias canônicas (plano e caixa). Isso abriria espaço para lições sobre mapeamento de texturas em malhas complexas, iluminação básica e *normal mapping*, aproximando o ambiente de casos de uso mais próximos da prática profissional.
- **Gizmos e controles de cena mais robustos.** Introduzir *gizmos* para manipulação direta de câmera e objetos (por exemplo, *orbit controls*, *trackball*, manipuladores de translação/rotação/escala), bem como indicadores visuais de frustum e eixos. Isso pode tornar a exploração espacial mais acessível, especialmente para quem ainda não domina plenamente transformações em coordenadas homogêneas.

6.2.2 Melhorias de layout e UI/UX

Um segundo eixo diz respeito à interface e à experiência de uso. Os comentários dos participantes destacam, em particular:

- **Layout mais flexível.** Tornar o painel de teoria redimensionável, permitir reorganizar regiões (por exemplo, colocar código e *preview* lado a lado em diferentes proporções) e adaptar melhor o layout a telas menores. Essas mudanças podem reduzir a sensação de "tela apertada" relatada por alguns usuários.
- **Sinalização mais clara do fluxo.** Destacar de forma mais enfática qual aba deve ser usada em cada momento, indicar visualmente quais trechos de código estão sendo discutidos no texto e, quando possível, oferecer ligações mais explícitas entre código e *preview* (por exemplo, realçar uma região da tela ao focar uma função específica).
- **Tutoriais de onboarding.** Incluir um conjunto de micro-tutoriais iniciais que apresentem, de maneira guiada, cada elemento da interface (painel de teoria, abas, `u_texture0..3`, controles de geometria e câmera). Isso pode reduzir a barreira de entrada observada na Lição 1, quando parte dos participantes relatou não saber exatamente "o que fazer" na primeira interação.

6.2.3 Maturação do material didático

Embora as lições 1–3 tenham sido suficientes para testar a ferramenta, os resultados apontam para a importância de desenvolver um material didático mais cuidadosamente elaborado:

- **Progressão mais suave.** Redistribuir conceitos hoje concentrados na Lição 1 em unidades menores, introduzindo micro-exercícios ao longo do texto, e não apenas ao final. Isso pode tornar a curva de aprendizagem menos íngreme e evitar que o aluno precise "segurar" muitas ideias novas antes de praticá-las.
- **Conjunto de lições alinhado a uma disciplina.** Projetar e testar um conjunto maior de lições que cubram, de forma mais sistemática, tópicos típicos de uma disciplina de Computação Gráfica (coordenadas, projeções, texturas, iluminação básica, etc.), utilizando a ferramenta como plataforma principal de experimentação.

6.2.4 Avaliação com docentes e em contexto de disciplina

Por fim, há um conjunto de investigações futuras diretamente relacionadas à avaliação da ferramenta em contextos mais amplos:

- **Estudos com foco na autoria.** Conduzir experimentos em que o foco principal seja o docente: observar como professores criam lições a partir de um projeto em branco, como utilizam o formato `.chroma` para salvar e versionar atividades e quais dificuldades emergem no processo de autoria e manutenção de material.
- **Uso ao longo de uma disciplina completa.** Integrar a ferramenta a uma disciplina inteira (por exemplo, um semestre de Computação Gráfica) e comparar turmas com e sem o uso sistemático da aplicação. Isso permitiria avaliar não apenas percepções pontuais, mas também efeitos mais amplos sobre retenção de conceitos, desempenho em avaliações e engajamento dos estudantes.

Em conjunto, essas linhas de trabalho reforçam a visão de que a ferramenta aqui apresentada é um ponto de partida: um protótipo funcional que já mostra benefícios concretos para a aprendizagem de *shaders*, mas que ainda pode evoluir significativamente em termos de recursos, usabilidade e integração curricular. A expectativa é que, ao avançar nessas direções, seja possível consolidar um ambiente que apoie não apenas a compreensão técnica dos *shaders*, mas também a prática contínua de autoria e experimentação visual no ensino de Computação Gráfica.

Referências

Edward Angel. The case for teaching computer graphics with webgl: A 25-year perspective. *IEEE Computer Graphics and Applications*, 37(2):106–112, 2017. doi: 10.1109/MCG.2017.26. 2, 12, 14, 15, 17, 18, 19, 22

Edward Angel and Dave Shreiner. *Interactive Computer Graphics: A Top-Down Approach with WebGL*. Pearson, 7th edition, 2014. 1, 6, 7, 8, 9, 10, 19, 24

CodeMirror. Codemirror documentation, 2025. URL https://codemirror.net/docs/. 18

Joey de Vries. Learnopengl, 2020. URL https://learnopengl.com/. 2, 21, 23, 29, 30

Heinrich Fink, Thomas Weber, and Michael Wimmer. Teaching a modern graphics pipeline using a shader-based software renderer. In *Eurographics 2012 - Education Papers*, pages 73–80, Cagliari, Italy, 2012. Eurographics Association. 2, 3, 12, 18, 22, 23, 33

John F. Hughes, Andries Van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, and Kurt Akeley. *Computer Graphics: Principles and Practice*. Addison-Wesley, 3rd edition, 2014. 1, 2, 5, 7, 8, 9, 10, 11, 12, 25, 32

Khronos Group. Opengl registry & reference pages, 2025. URL https://registry.khronos.org/OpenGL/. 9, 18

Steve Marschner and Peter Shirley. *Fundamentals of Computer Graphics*. A K Peters/-CRC Press, 4th edition, 2015. 1, 2, 8

Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL Shading Language*. Addison-Wesley, 3rd edition, 2009. 1

Jerry O. Talton and Darren N. Fitzpatrick. Teaching graphics with the opengl shading language. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, pages 259–263. ACM, 2007. 2, 3

Three.js. Three.js documentation, 2025. URL https://threejs.org/docs/. 18

Anexo I

Formulário de avaliação aplicado aos participantes

Uso de uma ferramenta interativa para aprender shaders

TERMO DE CONSENTIMENTO LIVRE E ESCLARECIDO (TCLE)

Você está sendo convidado(a) a participar de uma pesquisa sobre o **uso de uma ferramenta interativa para aprender shaders.**

A participação é **voluntária** e você pode desistir a qualquer momento, sem qualquer prejuízo.

1) Objetivo e metodologia

O objetivo desta pesquisa é **avaliar a clareza, a usabilidade e a experiência** de estudantes ao utilizar as lições 1, 2 e 3 sobre shaders.

Se você aceitar participar:

- Você irá **usar a ferramenta e realizar as lições 1, 2 e 3**.
- Em seguida, responderá a **um questionário online**, com perguntas rápidas de múltipla escolha e algumas questões abertas.

O tempo estimado de preenchimento do questionário é de **5 a 10 minutos**.

2) Participação voluntária

- Sua participação é **livre e não obrigatória**.
- Você pode **interromper** a qualquer momento, sem necessidade de justificar.
- Se decidir sair, seus dados poderão ser excluídos da análise, se você solicitar.

3) Confidencialidade e anonimato

- As informações fornecidas serão tratadas de forma **anônima e confidencial**.
- Não serão solicitados dados que permitam sua identificação direta (como nome completo).
- Os resultados poderão ser usados em **trabalhos acadêmicos** e **relatórios**, sempre de forma **agregada**, sem identificar participantes individualmente.

4) Riscos e benefícios

- **Riscos**: mínimos, semelhantes a qualquer atividade de responder a um questionário online.

Se alguma pergunta causar desconforto, você pode optar por **não responder.**

- **Benefícios:** sua participação ajuda a **melhorar** o material de ensino de **shaders**, contribuindo para pesquisas na área de computação gráfica/educação.

5) Contato

Em caso de dúvidas sobre a pesquisa ou sobre sua participação, você pode entrar em contato com o(a) pesquisador(a) responsável:

> Nome: Felipe Dantas Borges
> E-mail: felipedbbsb@gmail.com

* Indica uma pergunta obrigatória

1. Você leu o Termo de Consentimento Livre e Esclarecido acima e concorda em participar desta pesquisa? *

Marcar apenas uma oval.

- ◯ Sim, li e concordo em participar.
- ◯ Não concordo em participar.

A. Perfil e experiência prévia

2. Você já teve contato com shaders antes desta atividade?

Marcar apenas uma oval.

- ◯ Nunca tive contato com shaders
- ◯ Já ouvi falar, mas nunca programei shaders
- ◯ Já fiz alguns tutoriais ou exemplos simples de shaders
- ◯ Já desenvolvi shaders com certa frequência

3. Como você avalia sua experiência geral em programação?

Marcar apenas uma oval.

- ◯ Nenhuma experiência
- ◯ Iniciante
- ◯ Intermediário(a)
- ◯ Avançado(a)

4. Como você se sente em relação a conteúdos de álgebra (vetores, matrizes, coordenadas, etc.)?

Marcar apenas uma oval.

- ◯ Tenho muita dificuldade
- ◯ Tenho alguma dificuldade
- ◯ Me sinto razoavelmente confortável
- ◯ Me sinto muito confortável

B. Clareza e entendimento geral

Instruções gerais
Para as afirmações, use a seguinte escala:

1. Discordo completamente
2. Discordo
3. Neutro
4. Concordo
5. Concordo completamente

5. **1)** Entendi o que é um shader e o papel dele no pipeline gráfico. *

Marcar apenas uma oval.

- ◯ Discordo Completamente
- ◯ Discordo
- ◯ Neutro
- ◯ Concordo
- ◯ Concordo completamente

6. **2)** Entendi a diferença entre vertex shader e fragment shader *

Marcar apenas uma oval.

- ◯ Discordo Completamente
- ◯ Discordo
- ◯ Neutro
- ◯ Concordo
- ◯ Concordo completamente

7. **3)** As explicações sobre coordenadas UV foram claras. *

Marcar apenas uma oval.

- ◯ Discordo Completamente
- ◯ Discordo
- ◯ Neutro
- ◯ Concordo
- ◯ Concordo completamente

8. **4)** Entendi o que são uniforms (por exemplo u_time, u_screenResolutio`, `u_texture0`). *

Marcar apenas uma oval.

- ◯ Discordo Completamente
- ◯ Discordo
- ◯ Neutro
- ◯ Concordo
- ◯ Concordo completamente

9. **5)** Consegui entender os exemplos de código sem precisar de muita ajuda externa. *

Marcar apenas uma oval.

- ◯ Discordo Completamente
- ◯ Discordo
- ◯ Neutro
- ◯ Concordo
- ◯ Concordo completamente

C. Ferramenta / editor

10. **6)** Foi fácil rodar os exemplos e ver o resultado dos shaders na tela. *

Marcar apenas uma oval.

- ◯ Discordo Completamente
- ◯ Discordo
- ◯ Neutro
- ◯ Concordo
- ◯ Concordo completamente

11. **7)** Consegui identificar rapidamente onde editar o código de vertex e fragment. *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

12. **8)** O painel de Input Geometry (Plane/Box e tamanhos) foi fácil de entender e usar. *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

13. **9)** Ativar o wireframe ajudou a visualizar a geometria (vértices/triângulos). *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

D. Lições específicas

Lição 1 — Gradiente e UV

14. **10)** Entendi o objetivo principal da Lição 1 *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

15. **11)** Consegui acompanhar os passos de texto + código da Lição 1 sem me perder. *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

16. **12)** Os exercícios da Lição 1 me ajudaram a praticar e fixar o conteúdo. *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

Lição 2 — Texturas e moldura

17. **13)** Entendi o objetivo principal da Lição 2. *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

18. **14)** Consegui acompanhar os passos de texto + código da Lição 2 sem me perder. *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

19. **15)** Os exercícios da Lição 2 me ajudaram a praticar e fixar o conteúdo. *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

Lição 3 — Mouse na cor, tempo na geometria

20. **16)** Entendi o objetivo principal da Lição 3. *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

21. **17)** Consegui acompanhar os passos de texto + código da Lição 3 sem me perder. *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

22. **18)** Os exercícios da Lição 3 me ajudaram a praticar e fixar o conteúdo. *

Marcar apenas uma oval.

- Discordo Completamente
- Discordo
- Neutro
- Concordo
- Concordo completamente

E. Campo aberto (opcional)

23. **19)** O que mais te ajudou a entender shaders nessas lições?(resposta aberta)

24. **20)** O que você mudaria ou melhoraria nas lições 1, 2 e 3?

Este conteúdo não foi criado nem aprovado pelo Google.

Google Formulários