UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

IMPLEMENTAÇÃO DE UM EDITOR VISUAL PARA A CRIAÇÃO DE SHADERS

TRABALHO DE GRADUAÇÃO

Leonardo Gonçalves Früh

Santa Maria, RS, Brasil
2011

IMPLEMENTAÇÃO DE UM EDITOR VISUAL PARA A CRIAÇÃO DE SHADERS

por

Leonardo Gonçalves Früh

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Orientador: Prof. Dr. Cesar Tadeu Pozzer

Trabalho de Graduação Nº 309
Santa Maria, RS, Brasil

2011

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada, aprova o Trabalho de Graduação

IMPLEMENTAÇÃO DE UM EDITOR VISUAL PARA A CRIAÇÃO DE SHADERS

elaborado por
Leonardo Gonçalves Früh

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Prof. Dr. Cesar Tadeu Pozzer
(Presidente/Orientador)

Prof. Dr. Eng. Marcos Cordeiro d'Ornellas (UFSM)

Prof^{a} Dr^{a} Márcia Pasin (UFSM)

Santa Maria, 12 de dezembro de 2011.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

IMPLEMENTAÇÃO DE UM EDITOR VISUAL PARA A CRIAÇÃO DE SHADERS
Autor: Leonardo Gonçalves Früh
Orientador: Prof. Dr. Cesar Tadeu Pozzer
Local e data da defesa: Santa Maria, 01 de novembro de 2011

Shaders são programas utilizados para redefinir o pipeline de execução do hardware gráfico. Desta forma torna-se possível a criação de aplicações que apresentem efeitos visuais mais sofisticados e que são executados mais rapidamente na favorável arquitetura das GPU's. Shaders podem ser escritos em diversas linguagens, tanto para a renderização offline como para aplicações em tempo real. Uma característica comum às linguagens de shader é que elas evitam os conceitos que linguagens de propósito geral possuem para facilitar a programação, como encapsulamento e abstração, aproximando a lógica do programa às características do hardware. Este fator também dificulta o trabalho entre programadores e artistas gráficos de times de desenvolvimento de software, pois os artistas projetam os efeitos visuais, mas precisam esperar os programadores implementá-los. Para solucionar este problema, este trabalho propõe um editor visual que os artistas possam utilizar para projetar os efeitos visuais desejados e gerar o código do shader em GLSL. A edição dos shaders será baseada na conexão de nós de forma a criar um grafo acíclico. Para definir características do efeito visual implementado pelos shaders, utiliza-se nós que representam mapas de textura e operações de combinação de cores. Desta forma, torna-se possível a implementação de shaders sem a necessidade de conhecer os conceitos de programação e matemática inerentes aos mesmos.

Palavras-chave: Shader, GLSL, computação gráfica, OpenGL, editor visual.

ABSTRACT

Undergraduate Final Work
Undergraduate Program in Computer Science
Federal University of Santa Maria

IMPLEMENTATION OF A VISUAL EDITOR FOR THE CREATION OF SHADERS

Author: Leonardo Gonçalves Früh
Advisor: Prof. Dr. Cesar Tadeu Pozzer

Shaders are programs used to define the rendering pipeline of the graphics hardware. Thus it becomes possible to create applications that have more sophisticated visual effects and run faster on the favorable GPU's architecture. Shaders can be written in several languages, for both offline rendering and real-time applications. A common characteristic of shader languages is that they avoid the concepts the general purpose languages have to make programming easier, such as encapsulation and abstraction, moving the program's logic close to the hardware characteristics. This fact also complicates the work of programmers and graphic artists of software development teams, as the artists design the visual effects, but need to wait for programmers to implement them. To solve this problem, this paper proposes a visual editor that artists can use to design the desired visual effects and generate the code for the shader in GLSL. The edition of shaders will be based on the connection of nodes in order to create na acyclic graph. Nodes that represent texture maps and color blending operations are used to define characteristics of the visual effect implemented by the shaders. This way, it's possible to implement shaders without the knowledge of programming and mathematics concepts inherent to them.

Keywords: Shader, GLSL, computer graphics, OpenGL, visual editor.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
Cg	C for Graphics
CRT	Cathode Ray Tube
FBO	Frame Buffer Object
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
HLSL	High Level Shader Language
OpenGL	Open Graphics Library
SIMD	Single Instruction Multiple Data

SUMÁRIO

1 INTRODUÇÃO

A expressão “computação gráfica” foi criada em 1960 por William Fetter. Nesta década o campo da computação gráfica obteve significativos avanços com a integração de diversas interfaces humano-computador, como o monitor de Tubo de Raios Catódicos (CRT) para a visualização. As imagens geradas nesse período eram vetoriais, diferentemente das representações atuais formadas por pixels. No final dessa década, ocorreram duas das mais importantes contribuições para a área que foram o “algoritmo de superfície oculta” e a “aplicação de sombreamento em modelos 3D”. A partir deste momento inicia-se o desenvolvimento das técnicas de render e modelagem.

Na década de 90, surgiram as Interfaces de Programação de Aplicativos (API’s) gráficas, como Open Graphics Library (OpenGL) e DirectX, e a utilização de placas gráficas discretas em computadores pessoais foi fortemente impulsionada pela indústria dos jogos eletrônicos.

No início dos anos 2000, com o amadurecimento das API’s OpenGL e DirectX, as Unidades de Processamento Gráfico (GPU’s) incorporaram a capacidade de programação do *pipeline* de renderização através de *shaders*, que são programas executados exclusivamente na GPU. As principais linguagens de alto nível utilizadas para a programação de *shaders* são C for Graphics (Cg), High Level Shader Language (HLSL) e OpenGL Shading Language (GLSL). As linguagens HLSL e GLSL estão intimamente ligadas às API’s gráficas DirectX e OpenGL respectivamente, sendo a primeira exclusiva para plataformas Microsoft e a última, multi-plataforma. Cg, por sua vez, foi produzida pela Nvidia em colaboração com a Microsoft. *Shaders* escritos em Cg podem ser utilizados com aplicações OpenGL e DirectX, mas há problemas de compatibilidade ao compilar os *shaders* para placas gráficas fabricadas pela ATI.

Atualmente a utilização de *shaders* no desenvolvimento de aplicações gráficas em tempo real é fundamental. Infelizmente, o processo de integração entre

o programa principal, que utiliza a API gráfica, e o programa de *shader* é complexo. As linguagens de *shader* também não apresentam conceitos avançados de estruturação para encapsulação, modularização e abstração, o que dificulta a criação de *shaders* mais complexos. Para possibilitar que artistas e programadores trabalhem em times de desenvolvimento de forma independente são necessárias ferramentas que possibilitem a criação de *shaders* mesmo por membros com pouco ou nenhum conhecimento a respeito de *shaders* e dos conceitos matemáticos inerentes aos mesmos.

Existem algumas ferramentas que auxiliam na criação de *shaders*, porém grande parte delas está integrada a *game engines*, as quais possuem licenças com preços relativamente elevados. Os poucos programas *standalone* existentes não geram código em GLSL, o que resulta em programas com portabilidade limitada. Em ambos os casos estas ferramentas também não abstraem efetivamente alguns conceitos da programação manual de *shaders*, apresentando ao usuário conceitos matemáticos, como transformações entre sistemas de coordenadas, vetores ou interpolações de variáveis. Com uma ferramenta que apresente apenas informações familiares a artistas gráficos, como mapas de textura e manipulação de canais de cores, e que gere *shaders* em GLSL, será possível que membros de times de desenvolvimento sem conhecimentos avançados de programação trabalhem independentemente, acelerando a produtividade da equipe como um todo.

1.1 Objetivos

1.1.1 Objetivo geral

O objetivo deste trabalho é implementar um editor visual de *shaders* que possua uma interface composta por elementos simples e funcionais, para que possa ser utilizado por usuários com pouco ou nenhum conhecimento de programação de *shaders*. Este editor também deve possibilitar a criação de *shaders* multi-plataforma, e sua edição deve ser independente de ferramentas externas, para dar maior liberdade ao usuário.

1.1.2 Objetivos específicos

a) Realizar uma pesquisa bibliográfica para levantar os principais efeitos visuais utilizados em aplicações gráficas atualmente;
b) Analisar aplicações semelhantes existentes, com o intuito de definir um conjunto limitado de elementos utilizados para a edição visual de *shaders* que implementem os efeitos visuais definidos no objetivo a), e encontrar maneiras de aprimorar a usabilidade deste tipo de aplicação;
c) Após a implementação da ferramenta, comparar a aplicação criada com as demais aplicações existentes;
d) Concluir a cerca da ferramenta criada.

1.2 Organização do texto

Este trabalho está estruturado da seguinte forma: o próximo capítulo apresenta uma revisão bibliográfica sobre os conceitos envolvidos na programação de *shaders*. São apresentadas as diferenças entre o *pipeline* fixo e programável do OpenGL e como é feita a comunicação entre o programa principal e os *shaders*. Também é apresentado o modelo de iluminação Phong e as principais técnicas de simulação de superfícies utilizadas atualmente. Ao final deste capítulo é feita uma análise sobre as algumas ferramentas existentes atualmente para auxiliar nas tarefas citadas na introdução.

O capítulo 3 apresenta a arquitetura do editor visual de *shaders*, detalhando seu projeto. A seguir, o capítulo 4 apresenta em detalhes como foi implementado o editor. Ao final do capítulo 4 é mostrado um exemplo de uso da ferramenta criada.

Por fim, o capítulo 5 apresenta as considerações finais e sugestões para trabalhos futuros.

2 REVISÃO DA BIBLIOGRAFIA

Este capítulo destina-se à definição dos conceitos utilizados na implementação de aplicações gráficas e *shaders*. Através dessas definições será possível ter a devida compreensão dos fundamentos utilizados na concepção da aplicação proposta. Ao fim do capítulo são explicadas algumas técnicas de renderização que são utilizadas ao longo do projeto.

2.1 Pipeline de renderização

O *pipeline* de renderização é o modelo teórico que representa o método de renderização baseado em rasterização suportado pelo hardware gráfico atual. Ele descreve o processo pelo qual dados representando uma cena no espaço tridimensional são convertidos em uma imagem 2D que possa ser mostrada nas interfaces de visualização.

Entretanto, mesmo seguindo um modelo teórico base, cada fabricante de placas gráficas possuía suas próprias instruções para o cálculo e desenhos de gráficos. Com a necessidade de uma padronização, API's para o desenvolvimento de aplicativos gráficos foram criadas. As principais API's gráficas atuais são DirectX e OpenGL.

2.2 OpenGL

2.2.1 Pipeline fixo

O OpenGL destaca-se por ser uma API multi-plataforma, multi-linguagem e gratuita [OPENGL, 2011]. Seu funcionamento é o mesmo de uma máquina de estados, com funções que ligam, desligam ou mudam atributos dessa máquina, tais como a cor atual, os cálculos de iluminação e as matrizes de transformações.

Para definir as formas geométricas que serão renderizadas, utilizam-se funções para a especificação das coordenadas de cada vértice e seus respectivos atributos. A partir dessas informações, o *pipeline* de renderização do OpenGL realiza os cálculos de iluminação através dos quatro estágios descritos a seguir:

1. Transformação de vértice;
2. Construção de primitivas e rasterização;
3. Texturização e coloração de fragmentos;
4. Operações de raster.

As operações executadas em cada estágio são descritas detalhadamente em [WRIGHT, 2010]. A figura 2.1 ilustra os estágios citados acima.

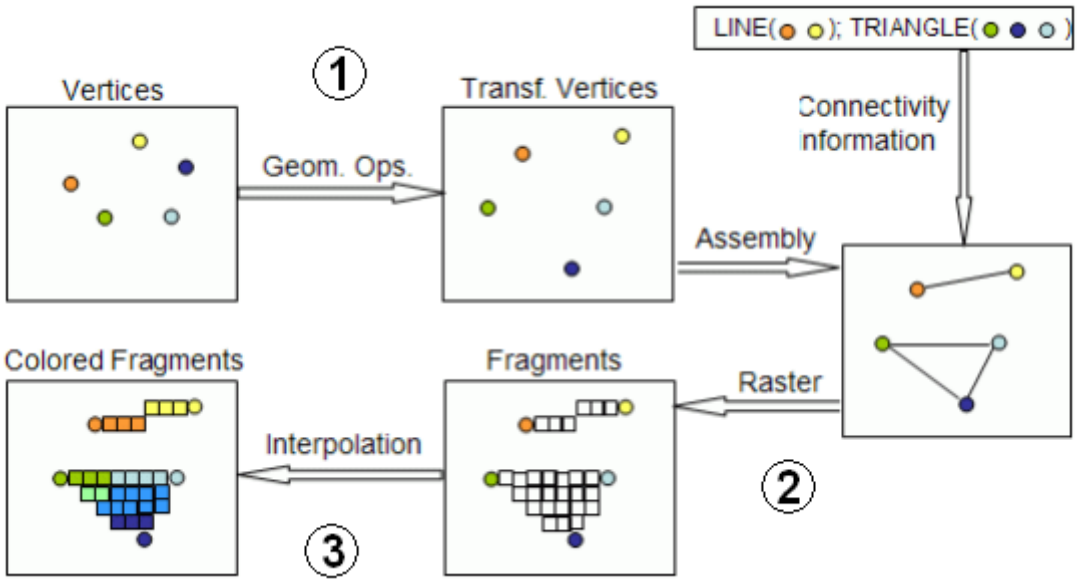


Figura 2.1 – Pipeline fixo do OpenGL
Fonte: Adaptação de [LIGHTHOUSE3D, 2011] (Pipeline Overview)

Nas primeiras versões do OpenGL, o comportamento de cada um dos estágios do *pipeline* era pré-definido. O nível de customização se limitava à seleção de um dos modelos fixos disponíveis. Para a definição da cor de preenchimento das faces dos polígonos, por exemplo, poderia ser escolhido *Flat* ou *Gouraud*. O primeiro representa toda a face de um polígono da uma mesma cor. O *Gouraud shading* por sua vez, é uma técnica um pouco mais avançada, mas ainda muito simples. Através da interpolação linear das cores de cada vértice de um polígono, é produzido um dégradé de cores ao longo da face do mesmo.

2.2.2 Pipeline programável

Com a capacidade de programação do *pipeline* de renderização adicionada às GPU's no início da década passada, tornou-se possível a produção de efeitos visuais muito mais variados e complexos através de *shaders*.

A primeira versão do OpenGL a suportar *shaders* foi a OpenGL 2.0, lançada em 2004. Juntamente com esta versão foi criada a linguagem GLSL para a programação de *shaders* com o OpenGL. A partir dessa versão, o *pipeline* do OpenGL passa a ter a forma ilustrada na figura 2.2:

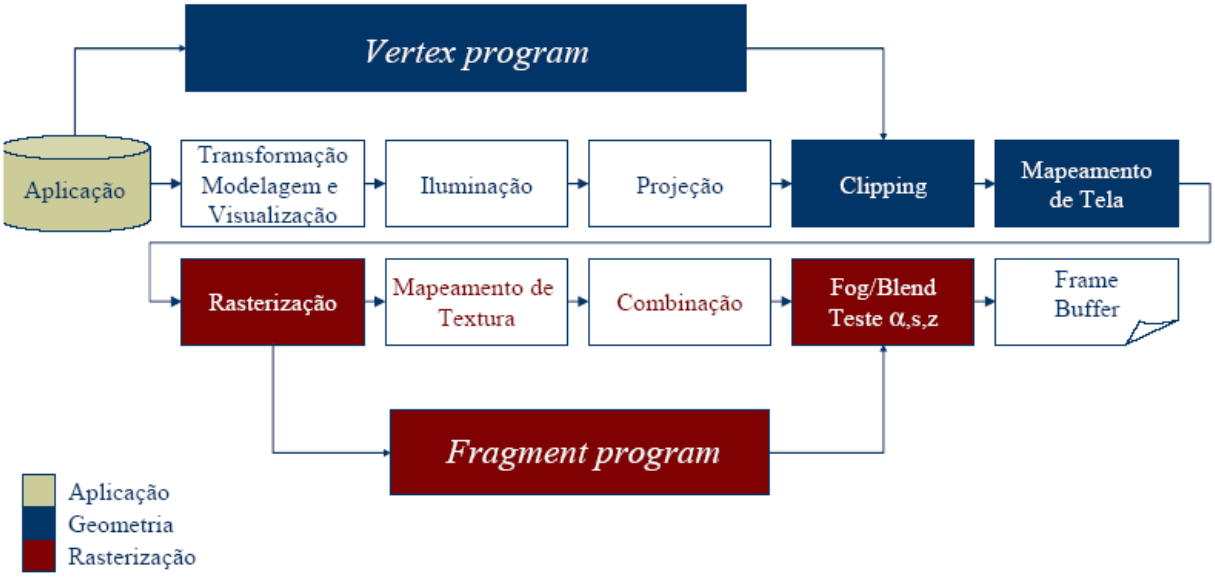


Figura 2.2 – Pipeline programável do OpenGL
Fonte: [Celes, W. Notas de Aula. PUC-Rio, 2006]

Neste modelo, as informações que definem cada vértice são transferidas da aplicação principal para os processadores de vértice. O resultado das operações realizadas nestes são encaminhadas para os processadores de fragmento que por sua vez encaminham os seus resultados ao *frame buffer* para exibição.

As operações realizadas pelo *shader* de vértice são:

- Transformações de posição do vértice;
- Transformações de normais;
- Geração de coordenadas de textura;
- Cálculos de cor.

O *shader* de fragmento por sua vez realiza as seguintes operações:

- Cálculo de cores e coordenadas de textura por pixel;
- Aplicação de textura;
- Cálculo de neblina.

2.3 GLSL

Uma aplicação em OpenGL que faça o uso de *shaders* é composta pelo programa principal, que pode ser escrito em qualquer linguagem suportada pelo OpenGL, como C++, e pelos programas de *shader* escritos em GLSL. O programa principal irá definir o comportamento das partes fixas do *pipeline* de renderização a partir de funções específicas do OpenGL, e os *shaders* de vértice e de fragmento serão executados nos processadores de vértice e fragmento da GPU, especificamente. Os *shaders* são executados seguindo a arquitetura *single instruction, multiple data* (SIMD), sendo que as instruções são únicas tanto para o *shader* de vértice, quanto para o de fragmento, e os dados variam conforme o vértice ou fragmento sendo computado. O *shader* de vértice executa uma vez para cada vértice dos modelos da cena. Por sua vez, o *shader* de fragmento executa em cada fragmento originado pela formação das primitivas. Na maioria dos casos, o *shader* de fragmento é executado mais vezes do que o de vértice, pois cada primitiva definida por três vértices pode gerar de dezenas a milhares de fragmentos.

Os *shaders* em si são apenas um conjunto de *strings*. Portanto, se os seus códigos-fonte estão em arquivos separados, a aplicação principal deve primeiramente ler e armazenar o seu conteúdo em *arrays* para então enviar esta informação ao *driver* da placa de vídeo para compilação em tempo de execução. A figura 2.3 resume o processo de preparação e utilização de *shaders* em uma aplicação escrita em C/C++ que use OpenGL.

Para cada *shader* deve-se criar um objeto que atuará como um container. Primeiro utiliza-se a função *glCreateShader* (1). Esta função recebe como parâmetro o tipo de *shader* que será criado, e retorna um identificador que será utilizado como referência para o mesmo.

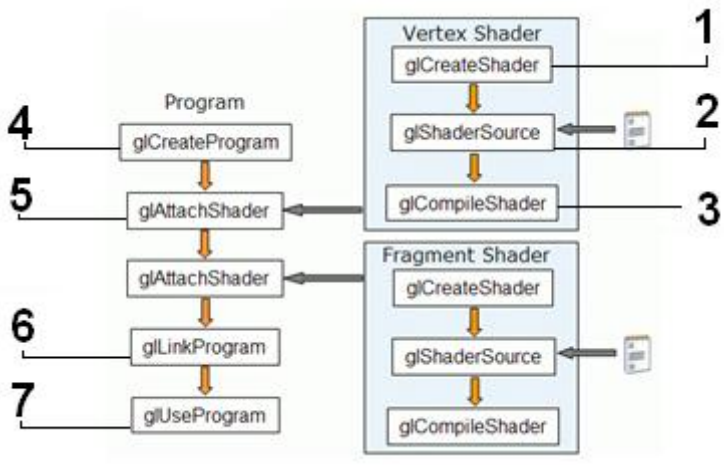


Figura 2.3 – Preparação do programa de shader
Fonte: Adaptação de [LIGHTHOUSE3D, 2011] (Creating a Shader) e [LIGHTHOUSE3D, 2011] (Creating a Program)

A seguir deve-se definir o código-fonte do *shader*. Para isso utiliza-se a função *glShaderSource* (2), que recebe como parâmetros o identificador retornado pela função *glCreateShader*, o número de strings que compõem o código-fonte, um *array* contendo essas *strings* e um *array* com o tamanho de cada uma das *strings*. Quando são utilizados arquivos separados que contém os códigos-fonte dos *shaders*, pode-se passar apenas uma *string* terminada por "/0", e "NULL" como último parâmetro.

Por fim, o *shader* deve ser compilado através da função *glCompileShader* (3), que recebe como parâmetro apenas o identificador retornado inicialmente pela *glCreateShader.* Após a compilação dos *shaders* de vértice e fragmento, deve-se uni-los em um programa de *shader*. A primeira função a ser chamada é *glCreateProgram* (4), que funciona de maneira similar à *glCreateShader*, retornando um identificador do programa sendo criado. A seguir, anexa-se os *shaders*, criados e compilados anteriormente, ao programa de *shader* através da função *glAttachShader* (5). Esta função receberá como parâmetros os identificadores do programa e do *shader* sendo anexado. Cada programa de *shader* pode conter vários *shaders* de cada tipo, mas, para cada tipo de *shader*, pode haver apenas um que possua a função *main*. A *main* é o programa em si, e definirá o comportamento do estágio correspondente do *pipeline* do OpenGL.

Após anexar os *shaders*, o programa deve ser linkado (6). Esta operação irá fazer, entre outras coisas, a conexão entre o *shader* de vértice e o de fragmento. Para utilizar um programa de *shader* e ativar o *pipeline* programável, a função

glUseProgram (7) é chamada com o identificador do programa a ser utilizado como parâmetro. A figura 2.4 apresenta um trecho de código que poderia ser utilizado para a preparação do programa de *shader*:

```
1   char *vs, *fs;
2
3   v = glCreateShader(GL_VERTEX_SHADER);
4   f = glCreateShader(GL_FRAGMENT_SHADER);
5
6   vs = readFile("toon.vert");
7   fs = readFile("toon.frag");
8
9   const char* vv = vs;
10  const char* ff = fs;
11
12  glShaderSource(v, 1, &vv, NULL);
13  glShaderSource(f, 1, &ff, NULL);
14
15  free(vs);
16  free(fs);
17
18  glCompileShader(v);
19  glCompileShader(f);
20
21  p = glCreateProgram();
22
23  glAttachShader(p, v);
24  glAttachShader(p, f);
25
26  glLinkProgram(p);
27  glUseProgram(p);
```

Figura 2.4 – Programa exemplo para a preparação de um programa de shader

2.3.1 Comunicação OpenGL/GLSL

Como foi mostrado na figura 2.2, os processadores de vértice e fragmento recebem informações da aplicação principal para realizar as suas operações. Para realizar a comunicação entre o programa principal e o programa de *shader*, os *shaders* tem acesso a algumas partes do estado do OpenGL. Além disso, o usuário pode definir variáveis e enviá-las aos *shaders*. Essas variáveis são definidas no *shader* em escopo global, com um qualificador especial e são de somente leitura. O qualificador utilizado para definir a variável vai depender de sua utilização.

Variáveis *uniform* só podem ter seu valor trocado por primitiva, ou seja, seu valor não pode ser modificado entre chamadas a *glBegin* e *glEnd*. Este tipo de variável pode ser definido tanto em *shaders* de vértice como de fragmento.

O OpenGL possui um grupo de funções para definir os valores de variáveis *uniform* no programa principal. Primeiro obtém-se o endereço de memória da variável que se deseja definir. Utiliza-se a função *glGetUniformLocation*, passando como parâmetros o identificador do programa de *shader* e um *array* com o nome da variável desejada. Tendo o local da variável em memória utiliza-se o conjunto de funções *glUniform{1,2,3,4}{f,i}[v]* ou *glUniformMatrix{2,3,4}{f,i}v* dependendo do tipo da variável. As figuras 2.5 e 2.6 ilustram a utilização de variáveis *uniform* para a definição de texturas.

```
1  GLint tex1 = glGetUniformLocation(p, "tex1");
2  glUseProgram(p);
3  glUniform1i(tex1, 1);
4
5  cena->render();
```

Figura 2.5 – Exemplo de utilização de variável uniform na aplicação C++

No exemplo da figura 2.5, utiliza-se a função *glGetUniformLocation* (1) para se obter a localização da variável *uniform* “tex1” do programa de *shader* “p”. Neste exemplo, assume-se que o programa de *shader* já foi compilado e linkado, e a variável “p” possui o seu identificador. Com a função *glUseProgram* (2), ativa-se o programa de *shader* “p”. Por último, utiliza-se a função *glUniform1i* (3) para atribuir o valor “1” à variável “tex1”. O valor “1” é a unidade de textura que se deseja usar com a variável “tex1” do programa de *shader*. Após estes passos, pode-se renderizar os objetos da cena que utilizarão o *shader* “p”. A figura 2.6 representa o *shader* de fragmento que poderia ser utilizado para amostrar a textura carregada no exemplo da figura 2.5.

Neste exemplo, faz-se uma simples atribuição da cor da textura ao fragmento sendo processado. A função *texture2D* (5) recebe como argumentos a unidade de textura a ser amostrada e as coordenadas, e retorna um vetor de quatro componentes (*vec4*) representando os canais de cor R,G,B e A.

```
1  uniform sampler2D tex1;
2
3  void main()
4  {
5          gl_FragColor = texture2D(tex1, gl_TexCoord[0].st);
6  }
```

Figura 2.6 – Exemplo de utilização de variável uniform no shader de fragmento

O outro qualificador de variáveis é o *attribute*. Este tipo de variável contém atributos de vértices, logo, só pode ser definido em *shaders* de vértice. A definição de valores de variáveis *attribute* é feita de maneira semelhante à de *uniforms*, através das funções *glGetAttribLocation* e *glVertexAttrib{1,2,3,4}{f,i}[v]*. As figuras 2.7 e 2.8 ilustram a utilização de variáveis *attribute* para a definição dos vetores tangente de cada vértice.

```
1  GLint tan = glGetAttribLocation(p, "tan");
2  glUseProgram(p);
3  GLfloat tan_v[] = {0, 1, 0};
4
5  glBegin(GL_QUADS);
6          glNormal3f(0, 0, 1);
7          glVertexAttrib3f(tan, tan_v);
8          glVertex3f(.0f, .0f, .0f);
9          glVertex3f(2.0f, .0f, .0f);
10         glVertex3f(2.0f, 2.0f, .0f);
11         glVertex3f(.0f, 2.0f, .0f);
12 glEnd();
```

Figura 2.7 – Exemplo de utilização de variável attribute na aplicação C++

O exemplo da figura 2.7 inicia obtendo a localização da variável *attribute* “tan” do programa de *shader* “p” (linha 1). Em seguida, cria-se um vetor tridimensional que será utilizado como o vetor tangente dos vértices a serem definidos (linha 3). Logo abaixo, a partir da linha 5, define-se um quadrilátero com os vetores “(0, 0, 1)” e

“tan_v” como normal e tangente, respectivamente. Deve-se notar que, embora se tenha utilizado o mesmo vetor tangente para os quatro vértices, é possível especificar um valor diferente para cada atributo de cada vértice. A figura 2.8 representa o *shader* de vértice que poderia ser utilizado para calcular o vetor bitangente, através do produto vetorial dos vetores normal e tangente.

```
1  attribute vec3 tan;
2  varying vec3 normal;
3
4  void main()
5  {
6          normal = normalize(gl_NormalMatrix * gl_Normal);
7          vec3 bitan = normalize(cross(normalize(tan), normal));
8          gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
9  }
```

Figura 2.8 – Exemplo de utilização de variável attribute no shader de vértice

2.3.2 Comunicação Vertex shader/Fragment Shader

O *pipeline* programável também possui comunicação entre o *shader* de vértice e o de fragmento. Essa comunicação é realizada com um terceiro qualificador de variáveis: *varying*. Variáveis *varying* devem ser declaradas em ambos os *shaders*. Durante a execução do *shader* de vértice esta variável deve ser inicializada. Após a execução do *shader* de vértice, os valores das variáveis *varying* são interpolados para cada fragmento. No *shader* de fragmento a variável é de somente leitura. No exemplo da figura 2.8 utiliza-se uma variável *varying* para interpolar o vetor normal dos vértices através da superfície da primitiva.

Vale ressaltar que a partir da versão 3.1 o OpenGL não possui mais o *pipeline* fixo e a utilização de *shaders* é mandatória. Para acessar as funcionalidades das versões anteriores foi criada uma extensão de compatibilidade.

2.4 Técnicas de renderização

Para a geração de imagens de alta qualidade que representem objetos em três dimensões em computadores, foram desenvolvidas diversas técnicas ao longo do tempo. Para simular a maneira como a luz se comporta e como ela influencia a coloração de objetos ao incidir sobre os mesmos, foram criados modelos de iluminação simplificados para que seja possível a sua computação em um período de tempo aceitável. Dentre estes modelos, destaca-se o modelo de iluminação Phong [PHONG, 1975].

Porém o modelo de iluminação é apenas um dos componentes necessários para a geração da imagem. Os modelos de objetos utilizados irão definir o que é visto na cena. A superfície de objetos reais é composta por uma quantidade muito grande de ranhuras macro e microscópicas. Mesmo que os modelos fossem simplificados para mostrar apenas os detalhes mais relevantes, a geometria ainda seria muito complexa para o processamento no *hardware* atual. Logo, também é necessária a utilização de uma técnica que simplifique esta representação. Para esta finalidade, a técnica mais comumente utilizada é o *bump mapping* [BLINN, 1978], que utiliza uma textura para perturbar a direção da normal da superfície. Como o modelo Phong utiliza o vetor normal à superfície em grande parte dos cálculos de iluminação, o *bump map* gera resultados bastante satisfatórios. Existem outras técnicas mais complexas para a simulação de superfícies, como o *Parallax Mapping* [KANEKO, T. et al., 2001] e o *Relief Mapping* [OLIVEIRA, M. M., 2000]. Estas técnicas apresentam melhores resultados, ao representar com maior fidelidade silhuetas e sombras, mas possuem um pior desempenho e não podem ser utilizadas com a mesma frequência que o *bump mapping*.

Com a utilização do modelo Phong e da técnica de *bump map* obtém-se um razoável grau de realismo. No entanto, algumas aplicações requerem efeitos visuais variados, que não necessariamente sejam realistas. Dentre estes, destaca-se o *toon shading.* Esta técnica tem como objetivo fazer com que o tipo de iluminação da cena se assemelhe ao de desenhos à mão.

As seções a seguir irão abordar em detalhes as técnicas citadas acima e suas implementações com *shaders*.

2.4.1 Modelo de iluminação Phong

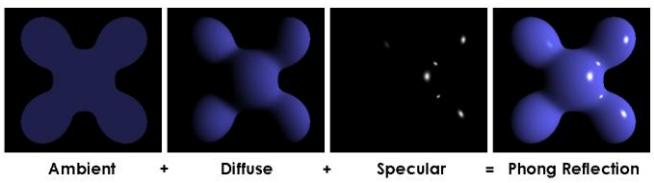
O modelo de reflexão Phong foi desenvolvido por Bui Tuong Phong em 1973. Juntamente com o modelo de iluminação também foi criado um método para interpolação dos valores de cada pixel rasterizado a partir de um polígono. Esta técnica de interpolação é chamada de Phong Shading.

O modelo Phong é essencialmente uma aproximação empírica de um sistema de iluminação local. Ele descreve a reflexão da luz na superfície de um objeto como a combinação de três componentes: reflexão ambiente, reflexão difusa e reflexão especular.

A componente ambiente é uma constante que representa a luz que é refletida por outros objetos da cena e atinge o objeto indiretamente. Esta componente é necessária para que áreas da cena que não são iluminadas diretamente por nenhuma fonte de luz não fiquem excessivamente escuras.

A segunda componente do modelo Phong é a reflexão difusa. Ela corresponde à luz que as partes ásperas da superfície de um objeto refletem. Devido à rugosidade da superfície, considera-se que a luz é refletida igualmente em todas as direções. Esta reflexão é calculada segundo a lei de Lambert, que define que a energia refletida por uma superfície é proporcional ao cosseno do ângulo entre a direção de incidência e a normal da superfície. Nota-se que a intensidade desta componente é independente do ponto de visão do observador, variando apenas conforme a posição do objeto em relação à fonte de luz.

A componente especular corresponde à reflexão especular das superfícies lustrosas. Por definir um cone de reflexão de luz em uma determinada direção, esta componente varia em função da posição do observador e da posição do objeto em relação à fonte de luz. A figura 2.9 ilustra graficamente a soma das três componentes deste modelo. As fórmulas para o cálculo de cada componente podem ser encontradas em [PHONG, 1975].



2.9 – Representação gráfica da fórmula do modelo Phong
Fonte: [WIKIPEDIA CONTRIBUTORS, 2011] (Phong reflection model)

O *pipeline* fixo do OpenGL utiliza o modelo de iluminação Phong, mas o método de interpolação utilizado é o *Gouraud shading*. Este método calcula a iluminação (cor) em cada vértice e interpola linearmente esses valores através da superfície da primitiva. Para a implementação do modelo Phong com a utilização de *shaders*, utiliza-se a técnica de *Phong Shading*. Com o *Phong Shading*, apenas os atributos dos vértices são interpolados e, através desses atributos, calcula-se a iluminação (cor) por fragmento. No *shader* de vértice calculam-se as normais, coordenadas de textura, transformações de vértice e sua posição. Os vetores normais e as posições de cada vértice serão interpolados linearmente e repassados aos *shaders* de fragmento, que também recebem as posições das fontes luminosas através do programa principal. Com essas informações, a cor de cada pixel é calculada no *shader* de fragmento. Este método apresenta uma iluminação mais precisa principalmente em casos onde os objetos são representados por um número menor de vértices.

A figura 2.10 mostra um comparativo entre a utilização do *Gouraud shading* (pipeline fixo) e do *Phong shading* (pipeline programável). Vale ressaltar que em ambos os casos o modelo de iluminação utilizado é o Phong.

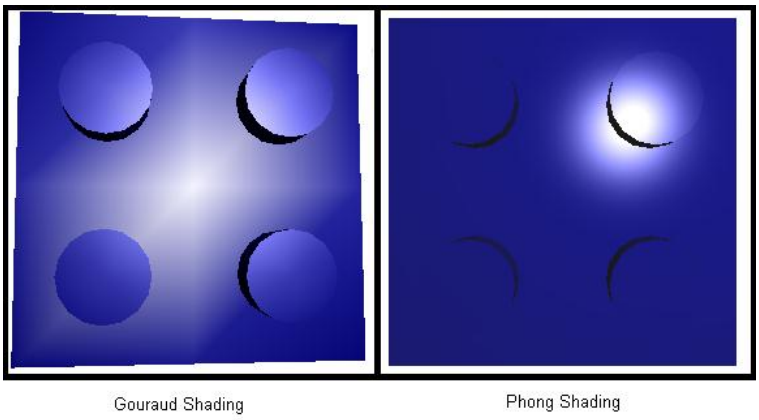


Figura 2.10 – Comparativo entre Gouraud shading e Phong shading

2.4.2 Bump mapping

Uma desvantagem da utilização das normais definidas por vértice é a necessidade de modelos formados por malhas de muitos vértices para que cada polígono corresponda a apenas um pixel da imagem final, o que resultaria na máxima qualidade de detalhes possível. Modelos com este nível de definição ainda

são inviáveis para aplicações em tempo real. Para contornar esta limitação foram criadas técnicas de mapeamento de informações através de texturas. Dentre essas técnicas destaca-se o *bump mapping*. Esta técnica utiliza uma textura em níveis de cinza para definir perturbações sobre a normal originalmente interpolada ao longo da superfície. Para calcular o nível de perturbação da normal em cada ponto da superfície, devem-se calcular as derivadas parciais em relação às coordenadas U e V da textura utilizada.

2.4.3 Normal Mapping

Como o custo computacional para o cálculo das derivadas parciais em tempo de execução ainda é alto, foi criada uma técnica mais eficiente com a mesma finalidade. O *normal mapping* [COHEN, J., 1998] utiliza uma textura RGB, no lugar da textura em níveis de cinza. Esta textura possui as coordenadas das normais pré-calculadas, logo, o custo adicional para o cálculo das derivadas parciais é eliminado.

A informação do mapa de normais é codificada como vetores no espaço tangente à superfície do objeto. No sistema de coordenadas tangente, a origem está localizada na superfície e a normal está alinhada com o eixo Z (0, 0, 1). Logo, os eixos X e Y são tangentes à face do polígono.

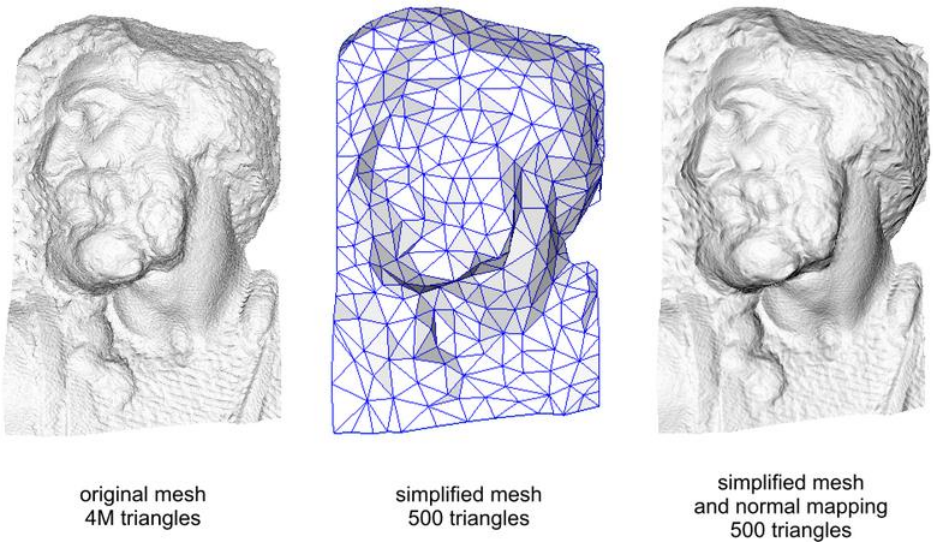


Figura 2.11 – Normal mapping
Fonte: [WIKIPEDIA CONTRIBUTORS, 2011] (Normal mapping)

Utilizando um mapa de normais codificado no espaço tangente, pode-se mapeá-lo a qualquer modelo e a informação mapeada permanecerá válida independentemente das transformações aplicadas ao modelo, já que o sistema de coordenadas é relativo á superfície mapeada e não ao mundo.

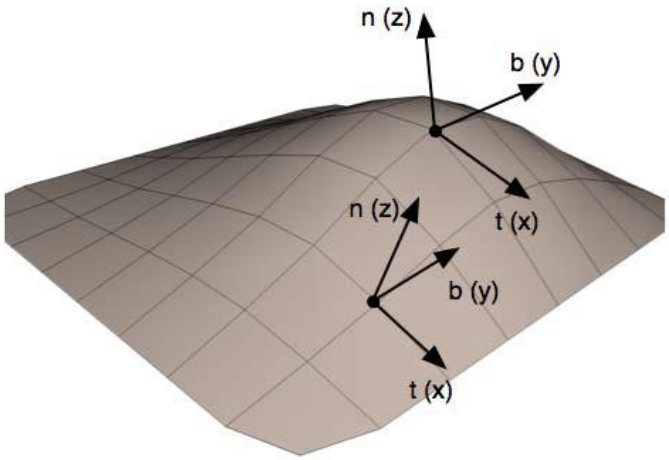


Figura 2.12 – Espaço tangente
Fonte: [WOLFF, 2011] (p. 118)

Para utilizar o mapa de normais, devem-se realizar todos os cálculos em espaço tangente. Para isso, devem-se transformar os vetores utilizados no sistema de iluminação. Para definir uma matriz de transformação do sistema de coordenadas de visualização para o espaço tangente, são necessários três vetores ortogonais e normalizados que definam este espaço. Estes vetores são o próprio vetor normal da superfície, e os vetores tangente e bitangente que estão alinhados respectivamente aos eixos X e Y do espaço tangente.

Tendo-se definido estes vetores, obtém-se a matriz de transformação:

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix}$$

Figura 2.13 – Matriz TBN

Vale salientar que é necessário pré-calcular os vetores tangente para cada vértice do modelo e enviá-los para o *shader* de vértice. Este, por sua vez, irá transformar os vetores necessários do espaço de visualização para o espaço tangente antes de encaminhar os valores interpolados ao *shader* de fragmento.

2.4.4 Toon shading

Como foi visto até o momento, a utilização de *shaders* é essencial para a implementação de efeitos visuais complexos e realistas, porém também se pode utilizá-los para definir modelos de iluminação customizados, que não necessariamente imitam a realidade. Como exemplo, tem-se a técnica de *toon shading*. Está técnica tem como objetivo fazer com que o tipo de iluminação da cena se assemelhe ao de desenhos à mão.

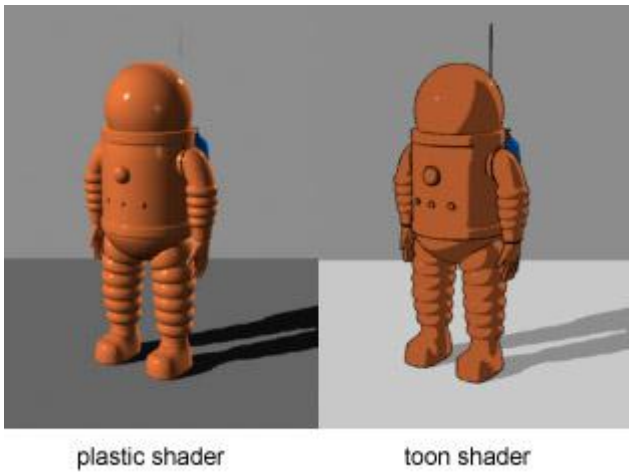


Figura 2.14 – Comparação entre Phong e Toon shading
Fonte: [WIKIPEDIA CONTRIBUTORS, 2011] (Cel-shaded animation)

Uma das técnicas utilizadas para se atingir este objetivo é a discretização dos níveis de intensidade luminosa. Como exemplo, pode-se utilizar o trecho de código da figura 2.15 para esta finalidade:

```
if (intensity > 0.98)
        color = vec4(0.8, 0.8, 0.8, 1.0);
else if (intensity > 0.5)
        color = vec4(0.4, 0.4, 0.8, 1.0);
else if (intensity > 0.25)
        color = vec4(0.2, 0.2, 0.4, 1.0);
else
        color = vec4(0.1, 0.1, 0.1, 1.0);

gl_FragColor = color;
```

Figura 2.15 – Trecho de código de toon shader

Neste exemplo, calcula-se a intensidade da reflexão difusa normalmente. Em seguida, este valor é mapeado para apenas quatro valores possíveis de intensidade, resultando nos sombreamentos mais bruscos característicos de ilustrações.

Para dar mais fidelidade ao efeito, podem-se simular os contornos escuros dos desenhos utilizando uma variedade de técnicas. Uma maneira simples de se obter este efeito é com uma renderização em dois passos. Primeiro deve-se renderizar apenas os polígonos do modelo que não estão de frente para a câmera. Esta renderização deve ser feita com uma escala um pouco maior que a padrão do objeto e na cor que se deseja para o contorno. Neste passo pode-se utilizar o *shader* de vértice para deslocar cada um dos vértices na direção de sua normal para expandir o modelo. No segundo passo, renderiza-se o objeto normalmente. Como os polígonos direcionados à câmera sempre estão mais próximos, apenas as bordas da primeira renderização serão visíveis, atingindo o efeito esperado.

Vale salientar que esta técnica não é viável para modelos compostos por muitos vértices, pois cada objeto será renderizado duas vezes. Nestes casos é aconselhado utilizar um algoritmo que utilize filtros de imagens para obter o efeito de contornos. Desta forma, a cena é renderizada apenas uma vez para uma textura, utilizando um *frame buffer object* (FBO), e um filtro de detecção de bordas adiciona os contornos.

2.5 Análise de aplicações para edição de shaders

Esta seção apresentará uma análise detalhada sobre alguns editores de *shaders*, expondo suas principais características, vantagens e desvantagens. A partir desta análise é definido o conjunto de elementos disponíveis para a edição dos *shaders*.

2.5.1 Source Shader Editor

O Source Shader Editor [BIOHAZARD, 2011] é uma ferramenta que faz parte da Source Engine. Ele é um editor baseado em nós conectados para formar um fluxograma, tanto para o *shader* de vértice como para o de fragmento.

Com este editor é possível criar *shaders* para materiais e efeitos de pós-processamento, bem como modificações na geometria dos modelos através do *vertex shader*. A figura 2.16 ilustra a interface do Source Shader Editor.

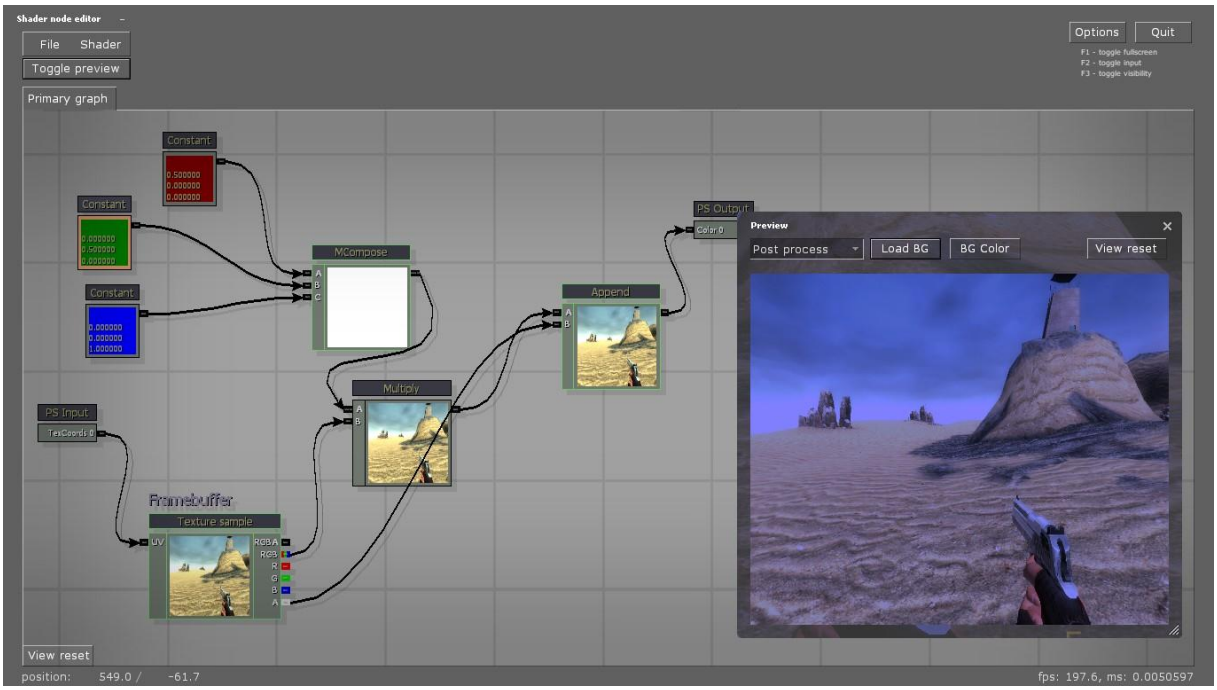


Figura 2.16 – Source Shader Editor

Os nós disponíveis para edição estão divididos em oito grupos descritos a seguir:

1. Semântica:

Possui nós de entrada e saída de dados dos *shaders*. Esses nós são equivalentes às variáveis *uniform*, *varying* e *attribute* utilizadas em *shaders* escritos em GLSL.

2. Matemática:

Possui nós que representam funções básicas como inversão e *swizzle*.

3. Matrizes:

Nós deste grupo representam as matrizes de transformação entre sistemas de coordenadas utilizadas na API gráfica.

4. Texturas:

Este grupo contém nós que representam mapas de textura e transformações das coordenadas UV.

5. Constantes:

Este grupo é utilizado para a criação de variáveis com valores predeterminados pelo usuário.

6. Controle de fluxo:

Este grupo representa estruturas de controle de fluxo de execução, como laços e condições.

7. Utilidade:

Contém nós utilizados para declaração e atribuição de variáveis, bem como funções para definir o efeito de neblina.

8. Utilidade / Studio model:

Esta categoria possui nós para transformações de vértice para animações (*skinning*) e cálculos de iluminação por vértice ou por pixel.

Analisando os tipos de nós citados acima, pode-se notar a estreita relação entre as funções e estruturas utilizadas na programação manual, e os nós utilizados para construir o fluxograma. Esta ferramenta segue o modelo de programação visual. Logo, para utilizá-la efetivamente, deve-se ter conhecimento prévio sobre o fluxo de execução de *shaders*, como as saídas e entradas de cada estágio do processo, bem como matrizes de transformação e tipos de dados. Desta forma, a ferramenta auxilia programadores a criar *shaders* de maneira mais eficiente, mas tem pouco a oferecer para usuários sem conhecimento de programação, como é o caso de artistas gráficos. Também vale salientar que esta ferramenta está integrada a uma *engine* e não possui a função de exportar os *shaders* gerados, limitando sua utilização a aplicações desenvolvidas com a Source Engine.

Após esta breve análise, conclui-se que um modelo semelhante ao do Source Engine Editor não é o mais apropriado para este trabalho, devido à complexidade de edição, à limitação de plataforma e à dependência de ferramentas externas.

2.5.2 Strumpy Shader Editor

O Strumpy Shader Editor [STRUMPY.NET, 2011] foi implementado como um pacote para a *engine* Unity3 Pro [UNITY TECHNOLOGIES, 2011]. Este editor também utiliza nós conectados para edição, porém o grafo final não representa um fluxograma. Ao invés disso, o usuário utiliza os nós que representam valores de entrada para definir as propriedades de cada um dos componentes do modelo de

material da *engine* Unity. São eles: Albedo, Normal, Emission, Specular, Gloss e Alpha. Estes componentes podem ser facilmente mapeados aos componentes do modelo de iluminação Phong. “Albedo” representa a cor do material. A concatenação dos componentes “albedo” e “alpha” pode ser diretamente mapeada à componente difusa do modelo Phong, em formato RGBA. O componente “normal” representa a direção do vetor normal em cada ponto da superfície do objeto. Com a utilização deste componente, o usuário pode implementar a técnica de *normal mapping* sem a necessidade de calcular a matriz de transformação TBN ou definir as variáveis que precisam ser interpoladas, normalizadas e enviadas na aplicação principal, no *shader* de vértice ou no *shader* de fragmento. Os componentes “specular” e “gloss” por sua vez, podem ser mapeados à cor da componente especular e à variável *shininess* do modelo Phong, respectivamente.

O Strumpy Shader Editor dispõe das seguintes categorias de nós para a definição dos componentes do material:

1. Constantes:

São nós que representam vetores de quatro componentes com valores definidos pelo usuário.

2. Funções:

Contém diversas funções matemáticas utilizadas na programação de *shaders*, como seno, cosseno, produto vetorial, produto escalar, raiz quadrada, comprimento, entre outras.

3. Entradas:

Possui valores de entrada do tipo cor, float, range, tempo, sampler2D, entre outros.

4. Operações:

Esta categoria é composta pelas quatro operações básicas da aritmética: adição, divisão, multiplicação e subtração.

Além destes nós de edição, também há o nó de saída ou nó mestre, que possui como entrada os componentes do modelo de material da Unity.

A abordagem utilizada pelo Strumpy Shader Editor, baseada na definição de propriedades, é útil até mesmo para usuários sem conhecimentos de programação, já que utiliza conceitos básicos de modelos de iluminação para definir a aparência final do *shader*. Outra vantagem é a unificação da edição em um único grafo, abstraindo o conceito de estágios do *pipeline* de renderização. Como desvantagens

há a limitação inerente a este tipo de editor, já que não é possível criar todos os tipos de efeitos visuais sem ter acesso a todas as funções da linguagem de programação. Também se pode citar como desvantagens a inclusão de uma grande quantidade de funções matemáticas, a representação de cores como vetores e a separação entre texturas e coordenadas de textura. Embora as funções matemáticas adicionem funcionalidade ao editor, deve-se lembrar que o objetivo deste tipo de aplicação é abstrair estes conceitos do usuário. A figura 2.17 ilustra a interface do Strumpy Shader Editor.

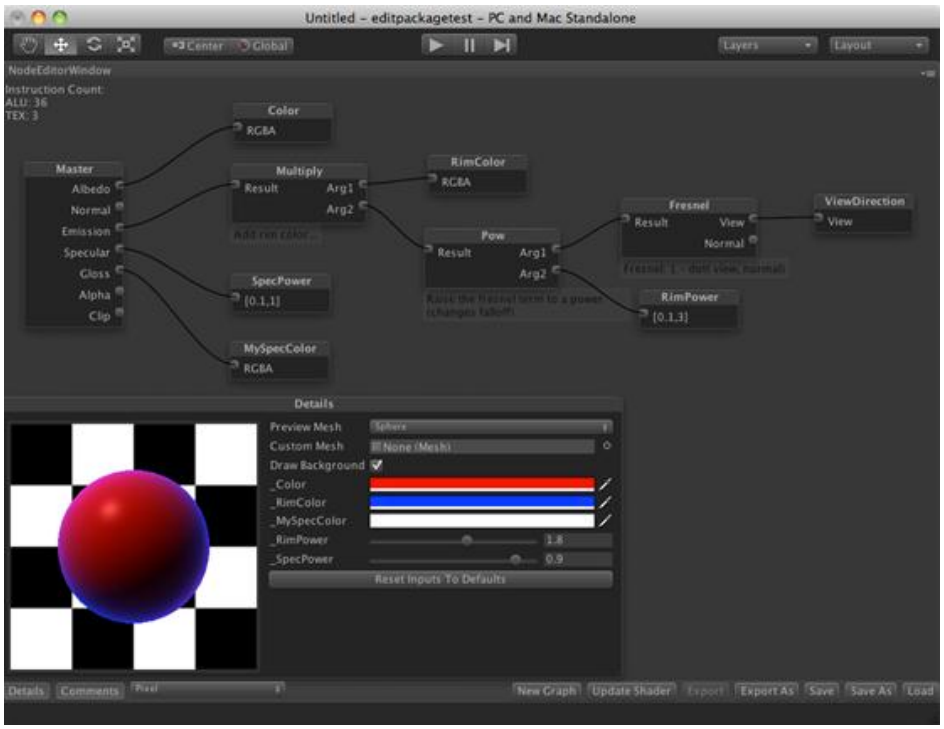


Figura 2.17 – Strumpy Shader Editor

A partir desta análise, pode-se concluir que o modelo utilizado por este editor, embora apresente algumas limitações, possibilita a implementação dos principais efeitos visuais utilizados atualmente de uma maneira simples. Para este trabalho, optou-se por apresentar os componentes do modelo de iluminação Phong como parâmetros de entrada. Além desta mudança, foram feitas simplificações na interface de edição para que o usuário se familiarize mais rapidamente com a ferramenta. Também foi adicionado um novo componente para aproximar o usuário da linguagem de programação, e auxiliá-lo a relacionar trechos de código ao efeito visual e ao grafo em edição. Estes componentes são apresentados em detalhes nos próximos capítulos, bem como detalhes de implementação da aplicação.

3 ARQUITETURA

Este capítulo apresenta de forma mais detalhada questões de projeto relacionadas ao desenvolvimento da ferramenta proposta. Primeiramente apresenta-se uma visão geral de sua estrutura. Na sequência, os principais módulos são apresentados em detalhes. Ao fim do capítulo, apresenta-se a arquitetura utilizada para a aplicação.

3.1 Visão geral

Tendo identificado os efeitos visuais suportados inicialmente e analisado aplicações utilizadas para a edição de *shaders*, definiram-se os componentes essenciais da interface e como eles seriam agregados. A ferramenta aqui proposta é composta por dois módulos principais que se comunicam constantemente: o módulo de edição e o de visualização ou *preview*.

O módulo de edição é baseado em grafos acíclicos, onde cada nó representa mapas de textura e funções de mistura ou *blend* de cores, e as arestas representam operações de *swizzle* e atribuições de variáveis. Semelhantemente à abordagem utilizada pelo Strumpy Shader Editor, que também é utilizada no editor da *engine* Unreal [EPIC GAMES, 2011], utiliza-se um nó mestre. Conectando os outros tipos de nós às entradas do nó mestre, definem-se as características do *shader* sendo editado. Este foi o maior nível de abstração encontrado a apresentar uma boa usabilidade por parte do usuário. Um nível de abstração menor, como o encontrado no Strumpy Shader Editor, pode impedir a utilização da aplicação por alguns usuários com pouco conhecimento de programação. Por outro lado, um nível de abstração muito alto, como o encontrado em [MCGUIRE, M. et al., 2006], pode confundir o usuário. Isto se deve ao fato de que as *Abstract Shade Trees* propostas apresentam nós de edição como efeitos visuais avulsos, sem regras que definam

claramente a ordem que tais efeitos são conectados e quais as entradas e saídas esperadas por cada efeito.

O segundo módulo, o módulo de visualização, compreende o painel de visualização do material, e o painel de visualização do código gerado. Ambos componentes são atualizados sempre que uma mudança significativa ocorrer no grafo sendo editado pelo usuário.

Além destes dois módulos, a interface possui componentes para a navegação por diretórios e seleção de arquivos de imagens. Estes componentes auxiliam o módulo de edição, permitindo ao usuário selecionar os arquivos de entrada de maneira intuitiva, mas serão apresentados separadamente para melhor estruturação do texto.

As próximas seções deste capítulo apresentarão cada um desses módulos em detalhes. Após esta apresentação detalhada, a arquitetura da aplicação é apresentada, mostrando como os módulos interagem entre si.

3.2 Módulo de edição

Como foi dito anteriormente, o módulo de edição é composto basicamente pelos nós de edição, sendo o usuário responsável pela adição ou remoção de nós, bem como a formação ou remoção de conexões e a mudança de configurações. Todo *shader* editado possui um, e apenas um, nó mestre. As entradas do nó mestre são as componentes difusa e especular, e as normais. Estas propriedades são diretamente mapeadas às componentes do modelo de iluminação Phong. Também há a possibilidade de configurar a componente ambiente através do painel de configurações deste nó.

Procurando simplificar a edição, reduziu-se o número de nós a componentes essenciais a qualquer *shader* de material: texturas e métodos de combinação das mesmas. As texturas são as informações de entrada do grafo. Através delas, definem-se todas as características dos *shaders*. Diferentemente das soluções empregadas em outros editores, não há nós para a definição de coordenadas de textura UV. Como coordenadas de textura não representam informações relevantes se não estão relacionadas a uma textura, integrou-se a modificação das coordenadas em um painel de configuração nos nós de textura. Com este painel de configuração pode-se definir animações na textura com escalas, rotações e

translações ao longo do tempo. Desta forma, simplifica-se o grafo de edição, removendo nós desnecessários.

Para a combinação de texturas são utilizadas funções de mistura de cores, ou *blending functions*, ao invés de funções matemáticas e operações aritméticas. Esta implementação auxilia a alcançar o nível de abstração desejado com este trabalho, ao apresentar ao usuário operações que se aplicam a cores e não a números. Embora as cores sejam implementadas como vetores de quatro números em ponto flutuante, este detalhe não precisa ser exposto ao usuário, que tem o intuito de trabalhar com cores. As *blending functions* suportadas são as funções de composição a partir do canal alfa (*alpha blending*) e a composição aditiva (*additive blending*). A figura 3.1 ilustra os nós disponíveis e seus respectivos painéis de configuração.

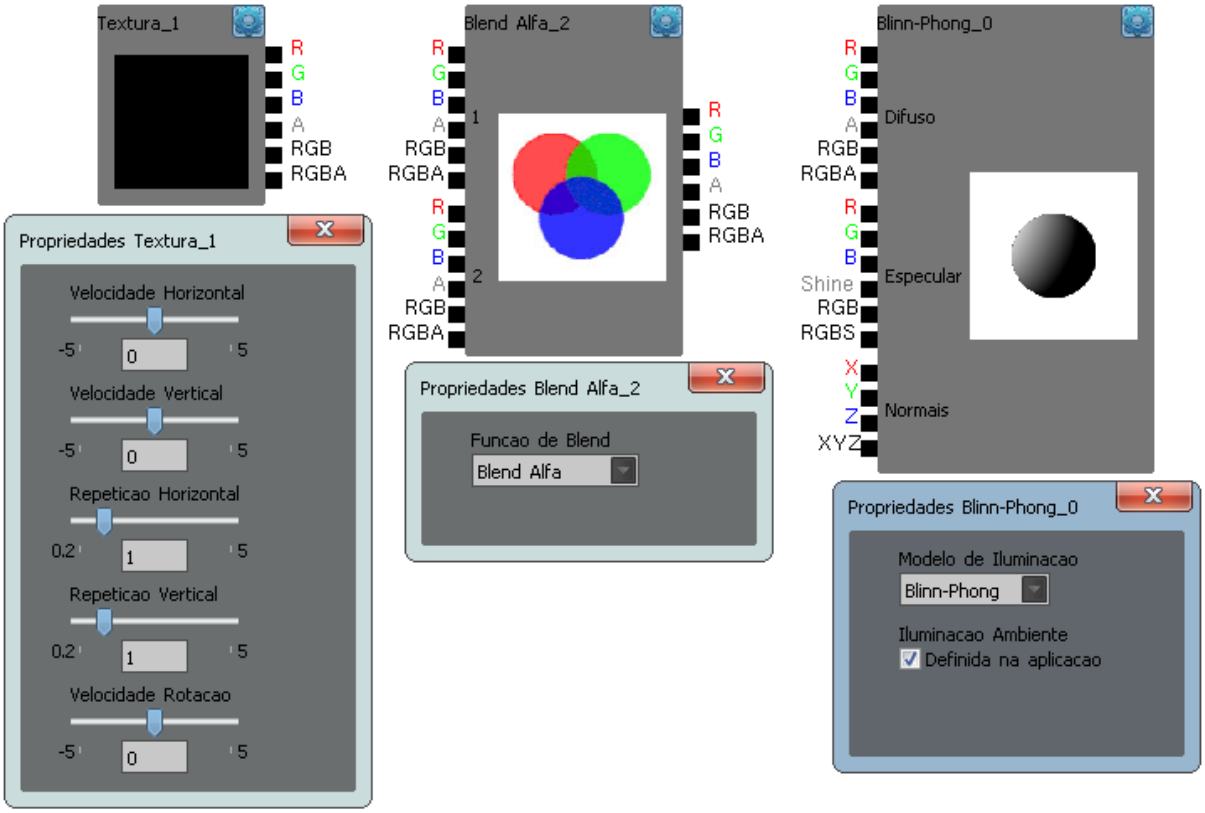


Figura 3.1 – Nós de edição

3.3 Módulo de visualização

O módulo de visualização possui a função de mostrar ao usuário o resultado do processo de edição do *shader*. Através do painel de visualização do material, o

usuário pode ver qual a aparência do *shader* sendo criado. Este componente é atualizado em tempo real para que o usuário tenha o *feedback* imediato do processo de criação.

Com o intuito de mostrar ao usuário a implementação dos *shaders*, foi criado um painel de visualização de código. Este painel mostra o código do *shader* de vértice e de fragmento que está sendo criado. Posicionando-o em um local de destaque na interface, ao lado do visualizador do material, procura-se chamar a atenção do usuário para o mesmo. Desta forma, espera-se aproximar o usuário da linguagem de programação, já que esta ainda é a maneira ideal e mais poderosa de se criar efeitos visuais. Vale salientar que com este componente não se procura ensinar o usuário a programar, mas apenas mostrar a ele como os *shaders* são implementados e permitir que este relacione o efeito visual ao código. As figuras 3.2 e 3.3 ilustram os painéis de visualização do material e do código, respectivamente.

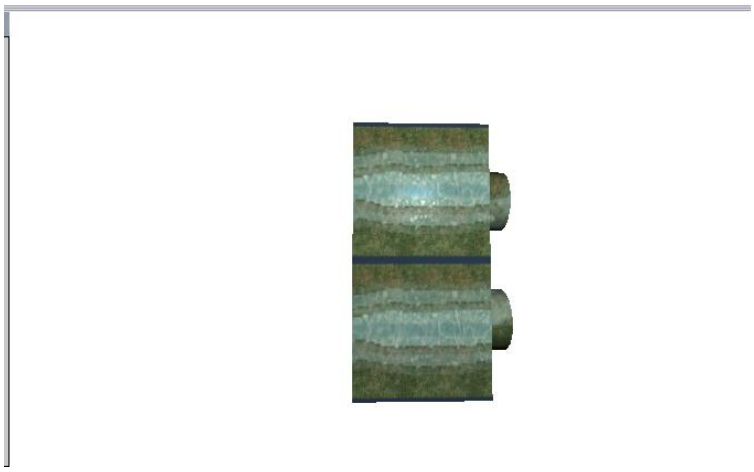


Figura 3.2 – Painel de visualização do material

Vertex Shader | Pixel Shader

```
varying vec3 normal;
varying vec3 view_dir;
varying vec3 light_dir;
varying vec2 Textura_1_coord;
uniform float time;
varying vec2 Textura_11_coord;
varying vec2 Textura_6_coord;
varying vec2 Textura_9_coord;
varying vec2 Textura_2_coord;
varying vec2 Textura_14_coord;
varying vec2 Textura_3_coord;

void main()
{
normal = normalize(gl_NormalMatrix * gl_Normal);
vec3 position = vec3(gl_ModelViewMatrix * gl_Vertex);
view_dir = normalize(-position);
light_dir = normalize(gl_LightSource[0].position.xyz - position);
Textura_1_coord = gl_MultiTexCoord0.st;
Textura_1_coord.s += mod(time*-0.2, 1.0)*1;
Textura_11_coord = gl_MultiTexCoord0.st;
Textura_6_coord = gl_MultiTexCoord0.st;
Textura_6_coord.s *= 2;
Textura_6_coord.t *= 2;
```

Figura 3.3 – Painel de visualização do código

3.4 Componentes para seleção de conteúdo

Para que o usuário possa escolher os arquivos de imagem que serão utilizados nos nós de entrada do tipo textura, há um navegador de diretórios em forma de árvore e um painel que lista todos os arquivos de imagem suportados no diretório selecionado. Com ações *drag-and-drop* define-se a imagem a ser utilizada para os nós de textura. As figuras 3.4, 3.5 e 3.6 ilustram o navegador de diretórios, o painel de conteúdo e uma visão geral da interface, respectivamente.

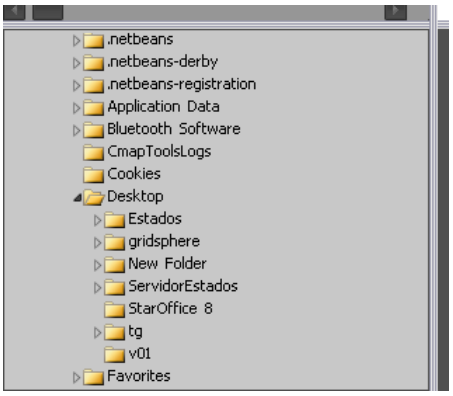


Figura 3.4 – Navegador de diretórios

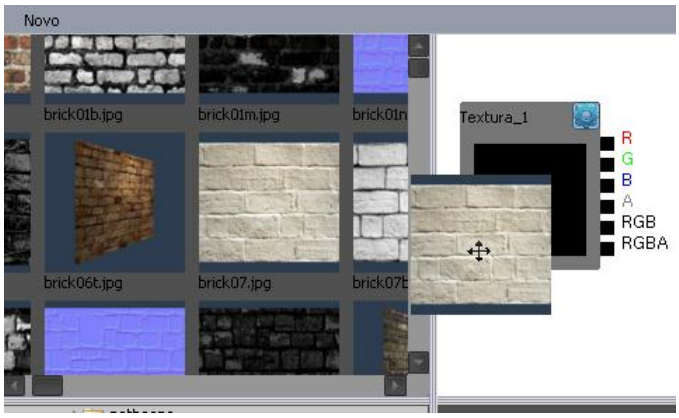


Figura 3.5 – Painel de conteúdo

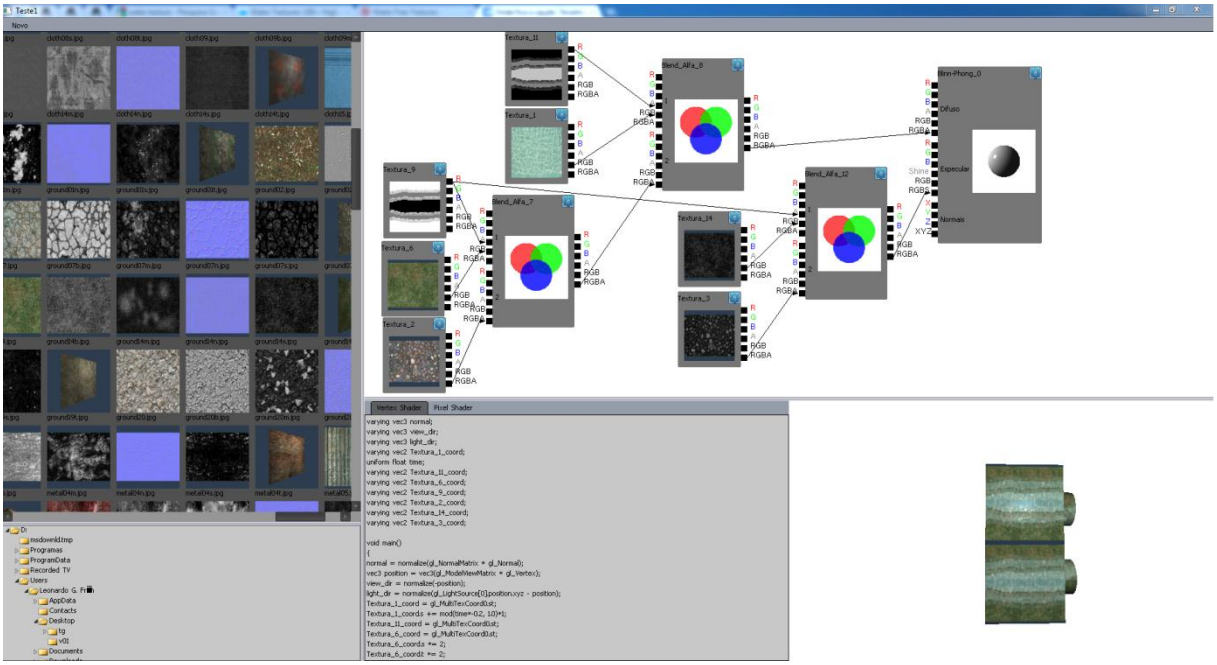


Figura 3.6 – Estrutura da interface

3.5 Arquitetura implementada

Um grande problema encontrado durante o projeto da aplicação foi a atualização dos painéis de visualização de forma dinâmica. A primeira solução encontrada foi a adição de um botão que o usuário utilizaria para recompilar o *shader* e atualizar a interface. O problema desta abordagem é que o usuário precisaria utilizar este comando frequentemente para ver os resultados da edição.

A outra solução encontrada para este problema, e que foi implementada como arquitetura da aplicação, consta em adicionar, a todos componentes do grafo de edição, um campo que indique se este componente sofreu alguma interação por parte do usuário. A partir desta informação, a estrutura que mantém o grafo pode definir se o grafo como um todo sofreu alguma modificação ou não. Em caso positivo, o *parser* é invocado para recompilar o código do *shader* e atualizar os componentes necessários. Desta forma, o usuário pode editar o *shader*, e qualquer modificação relevante irá disparar um evento de atualização da interface automaticamente.

Para iniciar a edição, o usuário precisa adicionar nós à área de edição do grafo. Os nós de textura e *blend* são criados através da barra de menus da aplicação. Todos nós criados são iniciados sem conexões com o grafo e inicializados com valores padrão. As regras de conexão, bem como a detecção de

nós relevantes à estrutura do grafo são executadas pela estrutura que mantém o conjunto de nós existentes. Utilizando as informações mantidas pela estrutura do grafo, o *parser* atualiza o código do programa de *shader* se necessário. Após a atualização do código, os dados são persistidos para recompilação, integração ao *pipeline* de renderização e atualização do painel de visualização de código.

Ainda como parte do processo de edição, há os componentes para seleção de conteúdo citados na seção 3.4. Estes componentes executam as tarefas de gerenciar as visualizações (*thumbnails*) dos arquivos de imagem contidos no diretório selecionado pelo usuário, e enviar o endereço do arquivo de imagem selecionado ao grafo, para modificação de um nó de textura. A figura 3.7 representa a arquitetura proposta.

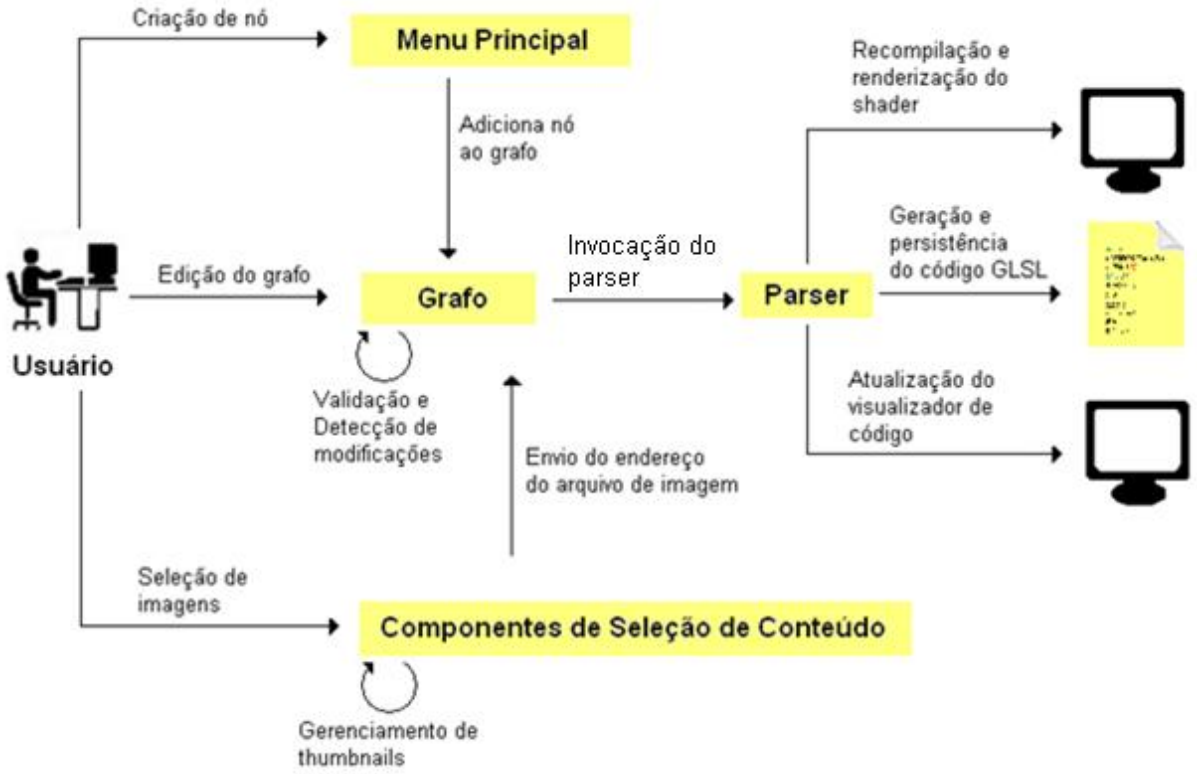


Figura 3.7 – Arquitetura do editor

4 IMPLEMENTAÇÃO

Este capítulo apresenta detalhes da implementação do editor proposto. Primeiramente são especificadas as ferramentas utilizadas para o desenvolvimento da aplicação. Em seguida, são apresentadas as classes que foram criadas para cada módulo, detalhando os métodos mais importantes. Ao fim do capítulo, é feita uma avaliação da ferramenta desenvolvida.

4.1 Ferramentas utilizadas

A aplicação foi desenvolvida em C++, utilizando o ambiente de desenvolvimento integrado (IDE) Microsoft Visual C++ 2010 Express. Para a implementação da interface foi utilizada a API Simple Components for Visual (SCV) [PAHINS, C. A. L., 2010]. Esta API foi desenvolvida no Laboratório de Computação Aplicada (LaCA) da Universidade Federal de Santa Maria (UFSM). Optou-se pela utilização da mesma devido à facilidade de integrá-la a aplicações OpenGL/GLSL. Além disso, os *widgets* da SCV são facilmente customizáveis através da derivação de suas classes. Além da SCV, foram utilizadas as bibliotecas Boost.Filesystem e FreeImage. A primeira possui diversas classes que facilitam a manipulação de arquivos e diretórios. A FreeImage, foi utilizada para a criação de *thumbnails* para visualização dos arquivos de imagem. Esta biblioteca é usada pela SCV, logo, não foi necessário acrescentar arquivos adicionais ao projeto para a sua utilização.

4.2 Classes implementadas

A seguir serão apresentadas as principais classes implementadas e seus métodos mais relevantes. O diagrama de classes apresentado nas figuras 4.1 e 4.2 ilustra como o programa foi organizado. Este diagrama foi simplificado e separado em duas figuras para uma melhor visualização.

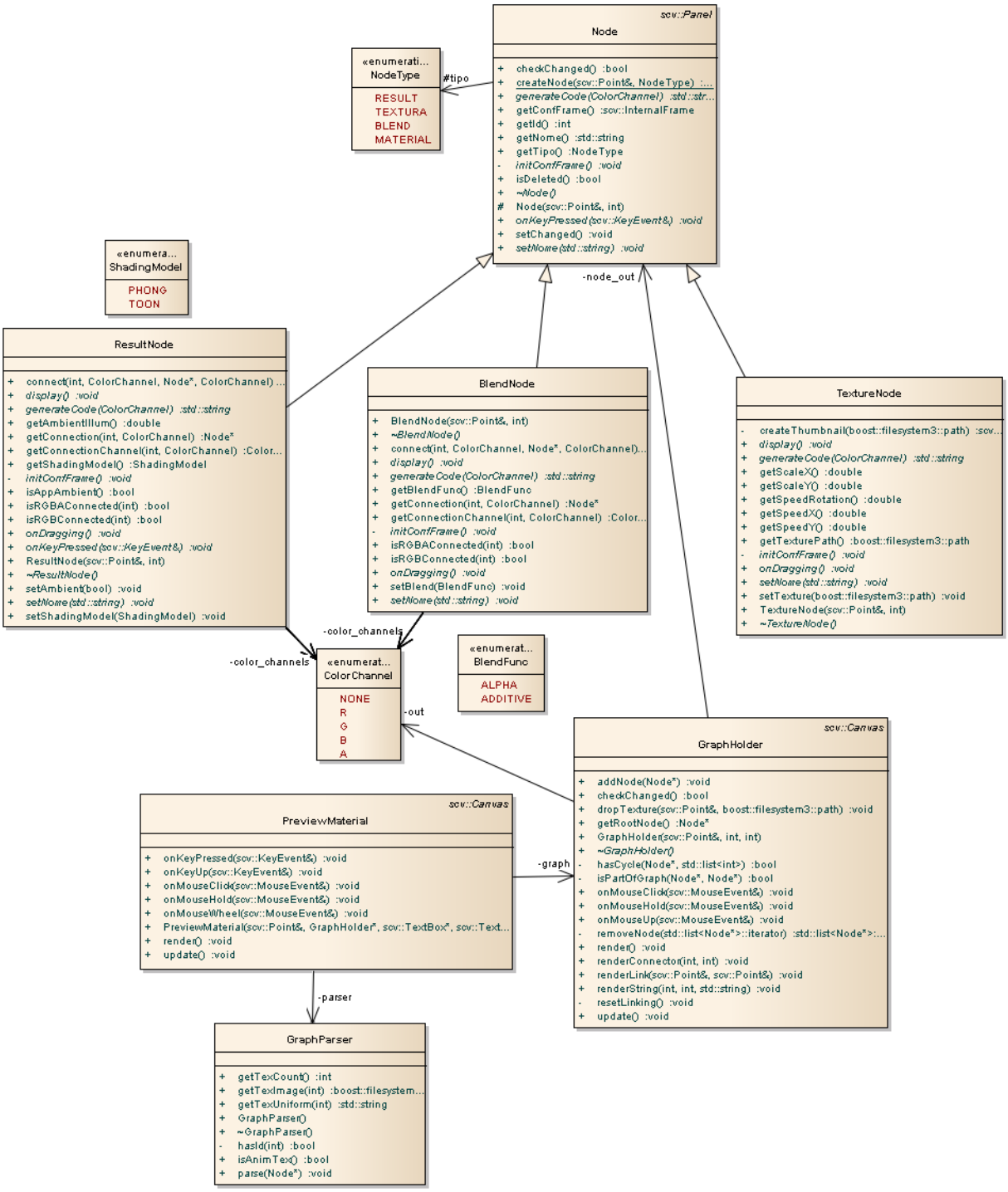


Figura 4.1 – Diagrama de classes da aplicação em UML (parte 1)

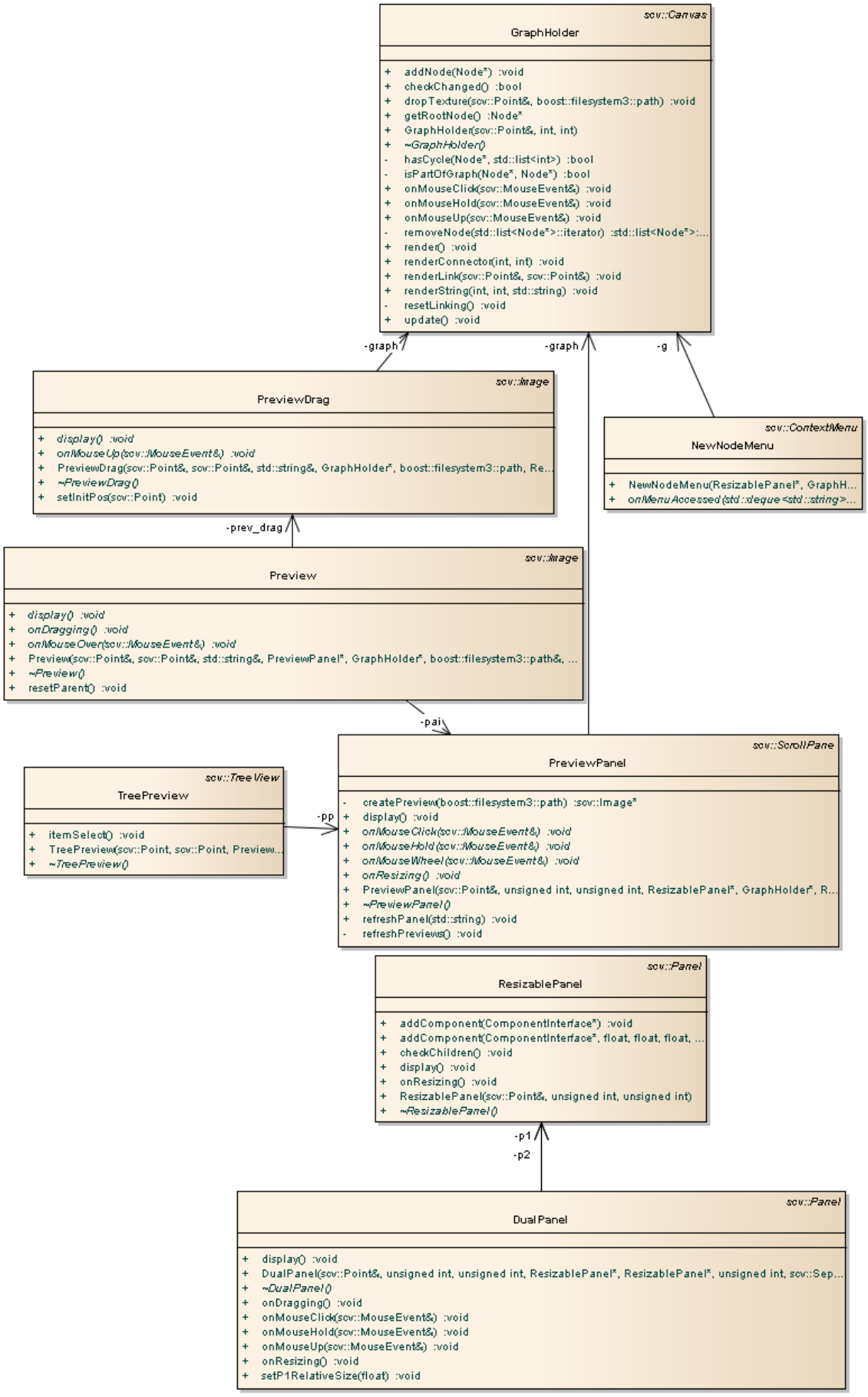


Figura 4.2 – Diagrama de classes da aplicação em UML (parte 2)

4.2.1 ResizablePanel

A fim de personalizar a interface ao nível desejado, foram criadas classes que estendem as funcionalidades da *widget scv:Panel* da SCV. A classe *ResizablePanel* é um simples container para outros componentes. O seu diferencial é a capacidade de se auto-redimensionar quando não possui nenhum componente como pai. Utilizando este componente como pai de todos os outros componentes da interface, é possível redimensionar a janela da aplicação e refletir essa mudança na posição e no tamanho de todos os componentes da interface. Outra característica importante é a capacidade de adicionar componentes a ele especificando uma posição relativa. Desta forma, um componente pode ser especificado para sempre estar posicionado no centro da janela, por exemplo, mesmo que esta varie de tamanho.

4.2.2 DualPanel

Outro componente criado para comportar outros componentes da interface é o *DualPanel*. Este componente é um componente composto, formado por dois *ResizablePanels* e um separador entre eles. Sua função é permitir ao usuário redimensionar determinadas seções da interface quando necessário. Por exemplo, utilizando um *ResizablePanel* para comportar a seção de edição e a seção de visualização, o usuário pode alocar uma área maior da tela para a tarefa que ele esteja desempenhando no momento: edição ou visualização do efeito visual criado.

4.2.3 Node

Para a implementação dos nós do grafo foi implementada uma classe abstrata *Node*. Esta classe possui todos os atributos comuns a todos nós, bem como um método estático para a criação de nós. Logo, esta classe também se comporta como uma *factory* de nós. Esta implementação foi necessária para que seja garantida a atribuição de ID’s únicos para cada nó criado, com o intuito de identificá-los em outras etapas da implementação.

Os nós do grafo são compostos por uma *scv::Image* para a parte de visualização, um *scv::Label* que apresenta o nome do nó, e um *ConfButton* para

abrir e fechar o painel de configuração. O painel de configuração foi implementado com o *widget scv::InternalFrame*.

4.2.4 ResultNode

A classe *ResultNode* implementa o nó principal da aplicação. Todo *shader* a ser editado possui este nó e apenas uma instância do mesmo. Para armazenar as informações referentes aos nós que estão conectados às suas entradas, este nó possui atributos do tipo *ColorChannel* e ponteiros para *Node*. *ColorChannel* define qual o canal de cor que está conectado. Este pode ser “R”, “G”, “B”, “A” ou “NONE”. Os ponteiros para *Node* identificam a qual nó que pertence o canal de cor conectado. Com esta estrutura pode-se navegar no grafo a partir das entradas do nó mestre, utilizando os ponteiros para *Node* até chegar a algum nó do tipo “TEXTURA”.

4.2.5 BlendNode

Os nós de *blend* são implementados pela classe *BlendNode*. Este tipo de nó possui conectores de entrada e saída. Como os nós que recebem conexões vindas deste já armazenam as referências, este nó possui atributos apenas para seus conectores de entrada. As funções representadas por este nó são o *blend* aditivo e o *blend* alfa. Os cálculos realizados pelo *blend* alfa são representados nas seguintes fórmulas:

$$out_A = src_A + dst_A(1 - src_A) \quad (1)$$

$$out_{RGB} = (src_{RGB} src_A + dst_{RGB} dst_A(1 - src_A)) \div out_A \quad (2)$$

O blend aditivo realiza os seguintes cálculos:

$$out_{RGBA} = src_{RGBA} + dst_{RGBA} \quad (3)$$

Esta última função simplesmente soma os dois canais de entrada. Com o *blend* alfa, obtém-se um efeito de transparência das duas cores. Através do canal alfa, define-se qual cor contribuirá em maior intensidade para a cor de saída.

4.2.6 TextureNode

Esta classe implementa os nós de textura. Por ser um nó de entrada, a única informação armazenada neste nó é a imagem utilizada como textura e as suas configurações, que podem ser modificadas através do painel de configurações. Para simplificar a implementação, este nó é inicializado com uma imagem preta de trinta e dois por trinta e dois pixels no formato png. Esta imagem está incluída na pasta de imagens da aplicação.

4.2.7 GraphHolder

Para armazenar todos os nós do grafo, até mesmo aqueles que não estão conectados uns aos outros, foi implementada a classe *GraphHolder*. Esta classe é composta por uma *std::list* de ponteiros de *Node*, que comporta todos os nós existentes na área de edição, e métodos auxiliares. Um destes métodos é o *hasCycle*. Este método é chamado toda vez que o usuário tentar conectar um nó a outro. Ele retorna *true* se o nó a ser conectado forma um ciclo no grafo, e *false* em caso contrário. Através deste retorno, o *GraphHolder* decide se invoca ou não o método *connect* dos nós escolhidos pelo usuário.

Como o *GraphHolder* é derivado da classe *scv::Canvas*, ele possui todas as *callbacks* desta última. São estas *callbacks* que são chamadas toda vez que o usuário tenta conectar ou desconectar os nós. O processo de movimentação dos nós é tratado pela callback de cada nó. O processo de deleção de nós é tratado em conjunto pelas duas classes. Quando o usuário pressiona a tecla “delete”, a *callback* de teclado do nó é chamada e a sua variável *deleted* é modificada para *true*. Desta forma, na próxima vez que a *callback* de atualização do *GraphHolder* for chamada, o nó é detectado como inválido e o método *removeNode* do *GraphHolder* é executado. Este método percorre a lista de nós buscando todos nós que possuem alguma conexão com o nó a ser apagado. Quando essas conexões são encontradas, seus valores são definidos para *ColorChannel::NONE* e o ponteiro de *Node* é modificado para “0”.

4.2.8 NewNodeMenu

A classe *NewNodeMenu* implementa o menu de criação de nós. Um dos parâmetros do construtor desta classe é um ponteiro para *GraphHolder*. Assim que o usuário seleciona uma das opções deste menu, esta classe invoca o método estático *createNode* da classe *Node* e, em seguida, o método *addNode* da classe *GraphHolder* para adicionar o nó criado à área de edição.

4.2.9 PreviewPanel

A parte da interface utilizada para escolha de conteúdo é composta por uma *TreePreview* para a navegação por diretórios e um *PreviewPanel* que mostra as imagens que podem ser utilizadas como textura contidas no diretório selecionado.

O *PreviewPanel* foi implementado com a *widget scv::ScrollPane* para que o usuário possa visualizar os itens sem precisar alocar toda a área da janela para este componente. Para criar as miniaturas que apresentam o conteúdo relevante do diretório, foi necessário criar *thumbnails* das imagens, para que o consumo de memória da aplicação seja mantido a níveis praticáveis. Para isso utiliza-se a biblioteca FreeImage. No entanto, mesmo com a criação das miniaturas, se o usuário abrir um diretório que possua muitas imagens, o consumo de memória pode chegar a níveis críticos. Para resolver este problema, a aplicação aloca apenas as imagens que estão próximas da porção visível do *PreviewPanel*. Sempre que a quantidade de imagens alocadas extrapolar um determinado limite, as imagens que não estão próximas da porção visível são desalocadas. Estas medidas também foram tomadas nas seções de visualização dos nós de textura. A imagem alocada para ser mostrada no painel destes nós é uma miniatura da imagem original. Vale salientar que, embora as seções de pré-visualização utilizem miniaturas, o *shader* sendo editado utiliza as imagens originais para não prejudicar a qualidade final do efeito visual criado.

4.2.10 GraphParser

A classe *GraphParser* possui a função de gerar o código GLSL a partir dos nós conectados. A partir das entradas do nó mestre que possuem conexões,

determina-se qual tipo de *shader* será criado. Se não há nenhum nó conectado à entrada de normais, gera-se apenas o algoritmo para o *Phong shading*. Se não há nós conectados à entrada especular, também não será necessário calcular a componente especular do modelo de iluminação Phong, o que resulta em um *shader* mais eficiente. As componentes ambiente e difusa sempre são calculadas.

Ao percorrer o grafo para gerar o código, esta classe cria uma variável do tipo *vec4* para cada nó de *blend*. Estas variáveis são nomeadas segundo o nome do nó de *blend*, logo, nunca serão criadas variáveis duplicadas, o que resultaria em código inválido. Estas variáveis são inicializadas no *shader* a partir dos nós conectados às suas entradas, conforme as equações (1), (2) e (3).

Quando o algoritmo de *parser* encontra um nó de textura, é necessário criar uma variável do tipo *sampler2D* no código do *shader* e alocar uma nova variável na aplicação em C++ para guardar a localização da variável do programa de *shader*. Esta localização é obtida através da função *glGetUniformLocation* depois que a geração do código GLSL estiver completa e o programa tenha sido compilado e linkado. Além desses passos, se algum nó de textura tiver opções de animação, seja ela rotação ou translação, é necessário adicionar uma variável *uniform* “time”. Esta variável será passada da aplicação C++ para o *shader* de vértice informando o tempo de execução da aplicação. A partir deste tempo, calculam-se as coordenadas de textura corretas para a animação configurada. Para cada nó de textura que possuir configurações de animação, é necessário adicionar uma variável *varying* aos *shaders* de vértice e fragmento. Estas variáveis irão armazenar as coordenadas de textura que foram calculadas no *shader* de vértice, interpoladas e enviadas ao *shader* de fragmento. Após todas as variáveis necessárias terem sido criadas, calcula-se cada componente do modelo de iluminação Phong no *shader* de fragmento. Ao fim do código deste, se o usuário selecionou a opção de *Toon shading* nas configurações do nó mestre, a cor do fragmento é discretizada em cinco intervalos de intensidade, de maneira semelhante ao código exemplo da figura 2.15.

4.3 Avaliação da ferramenta

Esta seção apresentará um exemplo de uso da ferramenta desenvolvida. O exemplo irá mostrar a edição de um *shader* simples para a simulação de um terreno

com água corrente. Serão utilizadas sete imagens para o mapeamento de cores, máscaras do canal alfa e mapa especular.

4.3.1 Interface inicial

Ao executar a aplicação, o formato inicial da interface é o ilustrado na figura 4.3. A área de edição possui apenas o nó mestre, o navegador de diretórios apresenta os diretórios raiz de todos os discos acessíveis do sistema e o *shader* inicial possui apenas o código essencial.

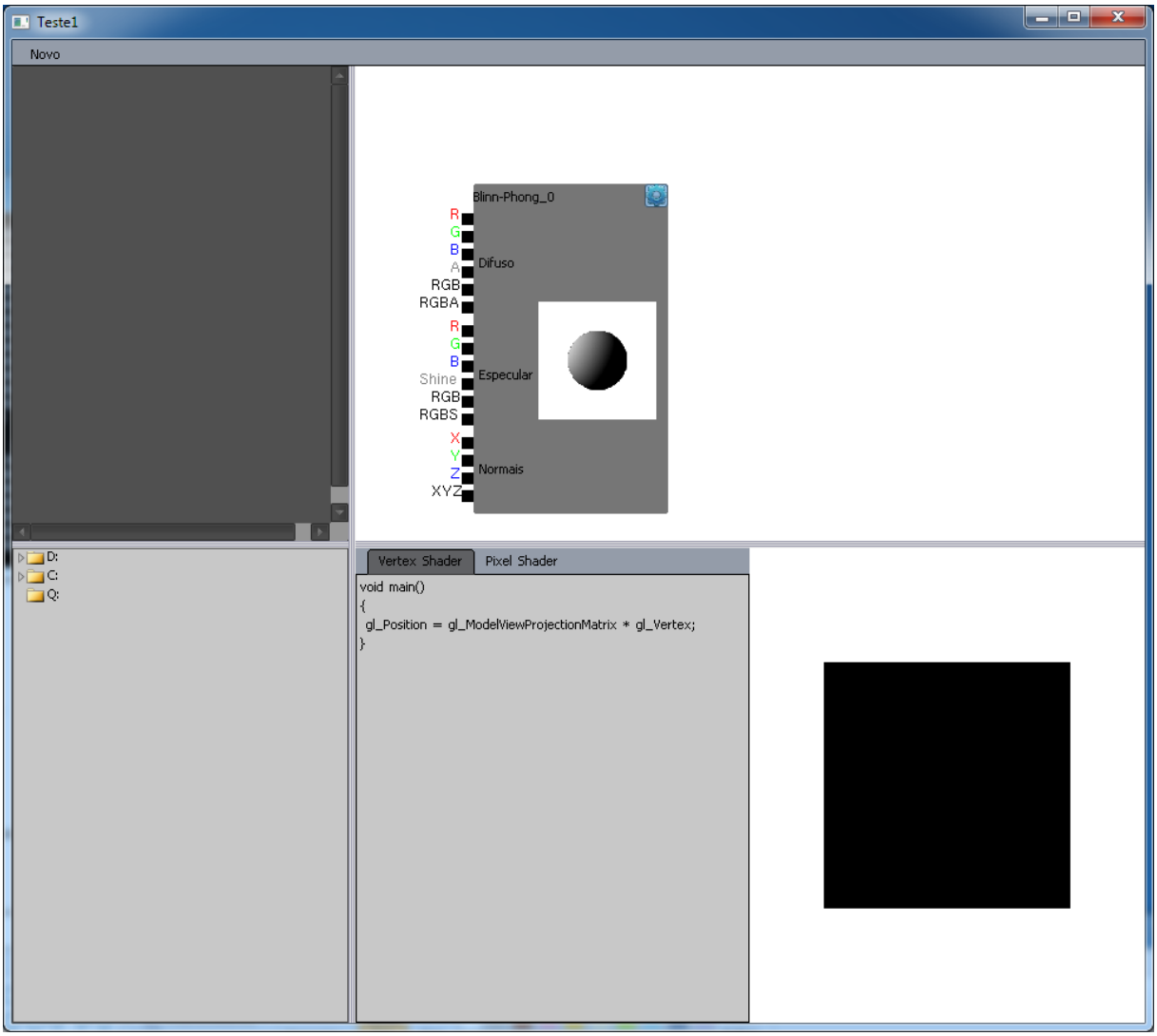


Figura 4.3 – Interface inicial

A figura 4.4 mostra em detalhes o código inicial. O *vertex shader* executa as transformações de projeção e visualização em cada vértice, e o *pixel shader* define a cor de cada fragmento para preto opaco.

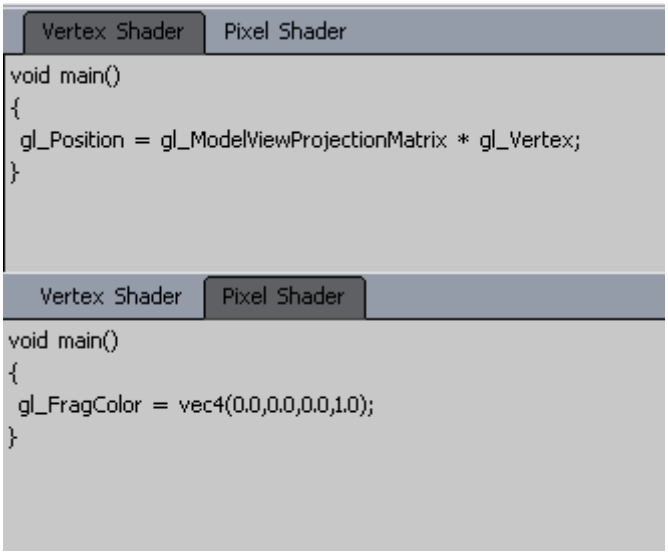


Figura 4.4 – Código inicial

4.3.2 Criação de nós

Para adicionar nós de edição, utiliza-se a barra de menus. Os nós disponíveis são os nós de *blend* e textura. Neste exemplo são usados sete nós de textura e três nós de *blend*. A figura 4.5 mostra a área de edição após a criação dos dez nós. Como pode ser visto, os nós de textura são inicializados com uma imagem preta padrão e os nós de *blend* são inicializados com a função de *blend* alfa.

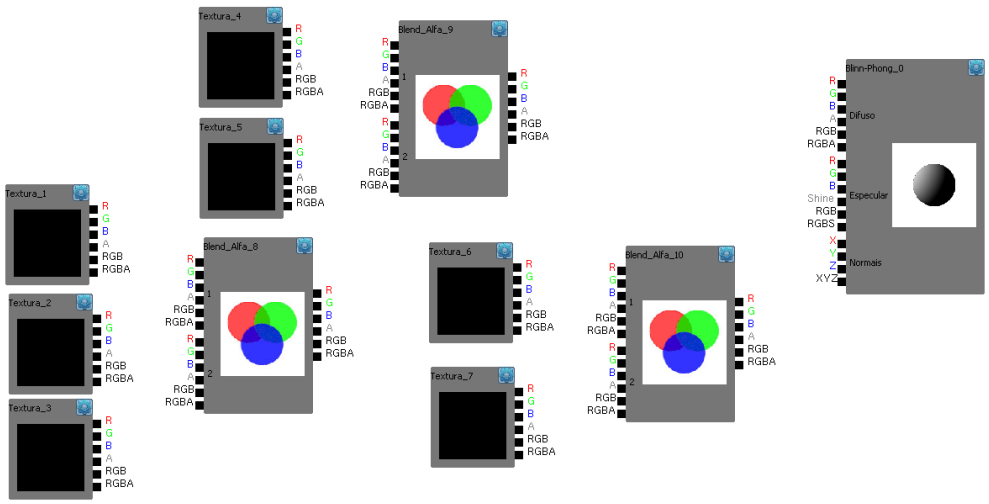


Figura 4.5 – Nós de edição criados

4.3.3 Configuração dos nós

Após a criação dos nós de edição, é necessário configurá-los. Utilizando os componentes de seleção de imagem, definem-se os arquivos de imagem que serão utilizados em cada nó de textura. A figura 4.6 ilustra este processo.

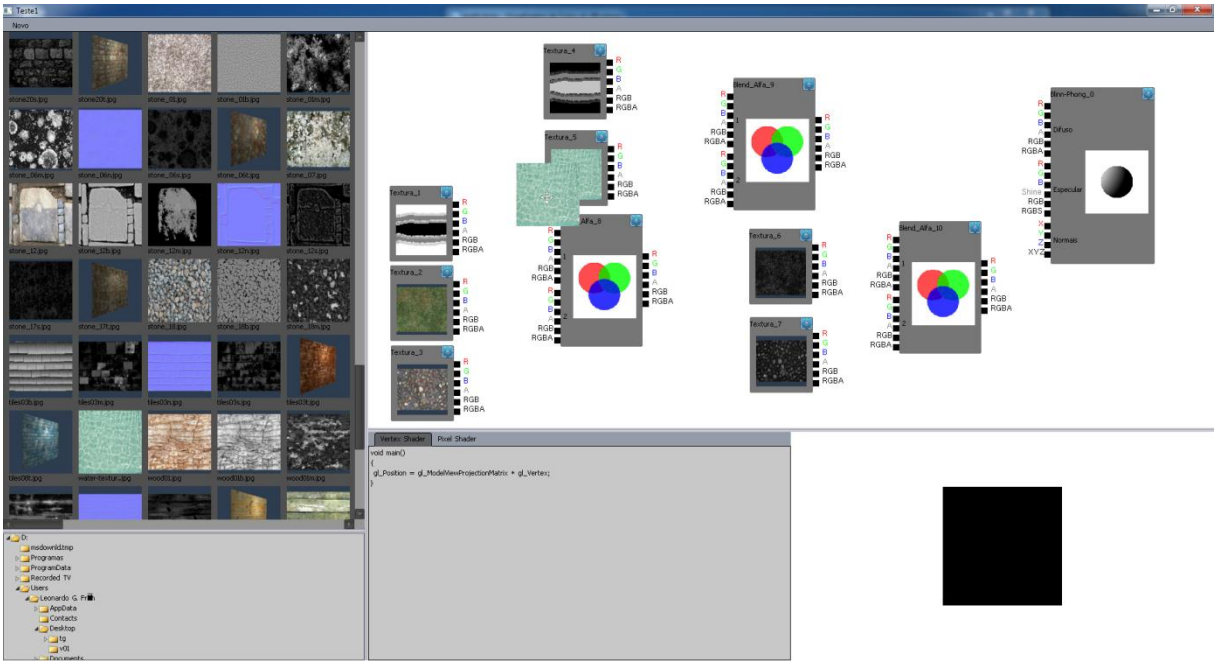


Figura 4.6 – Seleção de imagens

O painel de configuração dos nós de textura permite a definição de transformações sobre as coordenadas de textura. As opções disponíveis são: velocidade vertical e horizontal, escala vertical e horizontal e velocidade de rotação. A velocidade de deslocamento vertical e horizontal executa translações ao longo do tempo nas coordenadas V e U da textura, respectivamente. As opções de escala e rotação são auto-explicativas. A figura 4.7 ilustra como os nós de textura são configurados para este exemplo. Com estas configurações, as pedras da textura “Textura_3” apresentam um tamanho adequado e a textura “Textura_5” irá se deslocar horizontalmente de forma contínua, dando a impressão de água corrente.

A configuração dos nós de *blend* não é necessária neste exemplo, pois eles utilizam apenas o *blend* alfa, que é o valor padrão deste tipo de nó. O *blend* alfa é utilizado juntamente com as texturas “Textura_1” e “Textura_4” que são utilizadas como máscaras do canal alfa.

Figura 4.7 – Configuração dos nós de textura

No painel de configurações do nó mestre, podem-se configurar os cálculos de iluminação que serão realizados. As opções disponíveis são “Blinn-Phong” e “Toon”. O primeiro calcula o modelo de iluminação Phong com as componentes ambiente, difusa e especular. O segundo realiza a discretização da intensidade luminosa em cinco níveis pré-definidos, para causar o efeito de sombreamento característico da técnica de *toon shading*. Outra opção de configuração do nó mestre é a definição da intensidade da componente de iluminação ambiente. Pode-se escolher por utilizar o valor definido na aplicação OpenGL ou definir um valor manualmente. Neste

exemplo o nó mestre é configurado para utilizar apenas o modelo de iluminação Phong, e o valor da iluminação ambiente é definido manualmente para zero (0), para obter um maior contraste na imagem final. A figura 4.8 ilustra a configuração utilizada.

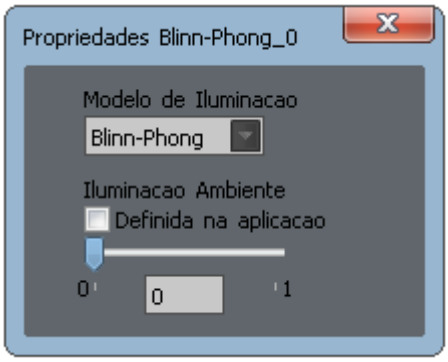


Figura 4.8 – Configuração do nó mestre

4.3.4 Conexão dos nós

Para que o *shader* seja criado, é necessário conectar os nós de edição às entradas do nó mestre para definir as propriedades do material. Todos os nós possuem canais de cor RGBA como entradas e/ou saídas. Para facilitar a conexão dos canais de cores, os nós também possuem conectores “RGB” e “RGBA”. Ao conectar estes pontos, os canais RGB e RGBA são interligados nesta ordem. Também é possível conectar os canais de cores em ordens arbitrárias. Neste exemplo, as texturas utilizadas como máscara de canal alfa são imagens em níveis de cinza. Como este tipo de imagem possui apenas um canal de cor, é necessário conectar o canal R, G ou B ao canal alfa do nó de *blend* correspondente. As figuras 4.9 e 4.10 mostram como os nós são conectados para este exemplo. Nota-se que nenhum código foi gerado com a configuração apresentada na figura 4.9, mas houve a geração com a configuração da figura 4.10. Isto se deve ao fato de que, embora houvesse conexões entre alguns nós no primeiro caso, nenhuma conexão havia sido feita às entradas do nó mestre. Logo, as propriedades do material ainda estavam indefinidas. As figuras 4.11 e 4.12 mostram o código GLSL gerado para os *shaders* de vértice e fragmento, respectivamente. A figura 4.13 mostra o *shader* criado.

Através deste exemplo, pode-se notar que conceitos matemáticos e de programação são completamente abstraídos da interface do usuário. Até mesmo

conceitos mais simples, como transformações sobre coordenadas de textura, são abordados de uma maneira menos técnica através dos painéis de configuração. Também pode-se observar que a localização do painel de visualização de código, ao lado do visualizador de material, ajuda o usuário a relacionar o efeito visual criado e o código correspondente.

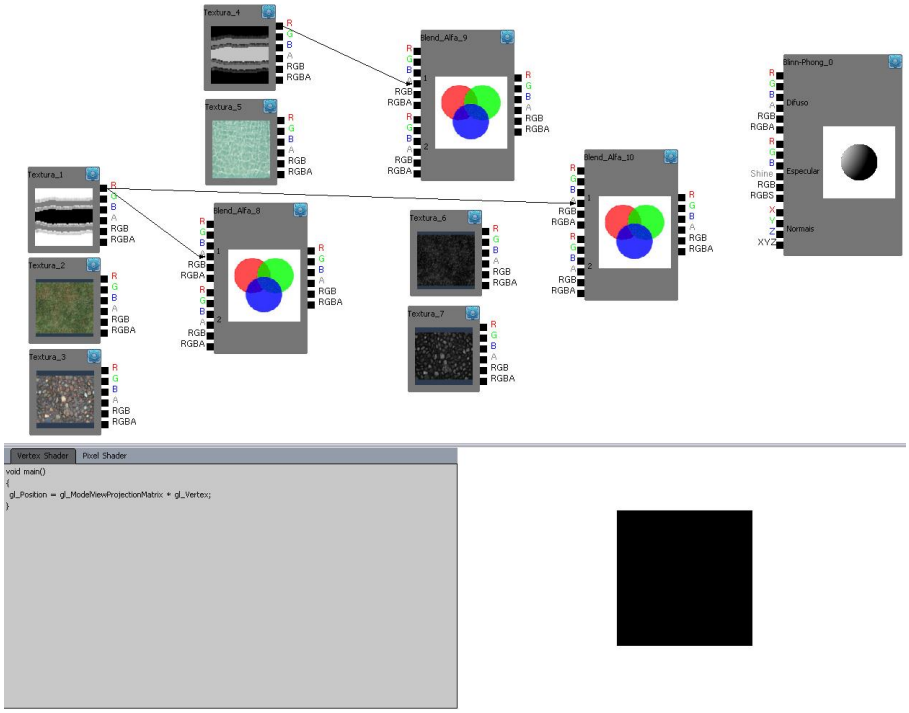


Figura 4.9 – Conexão dos nós de alpha mask

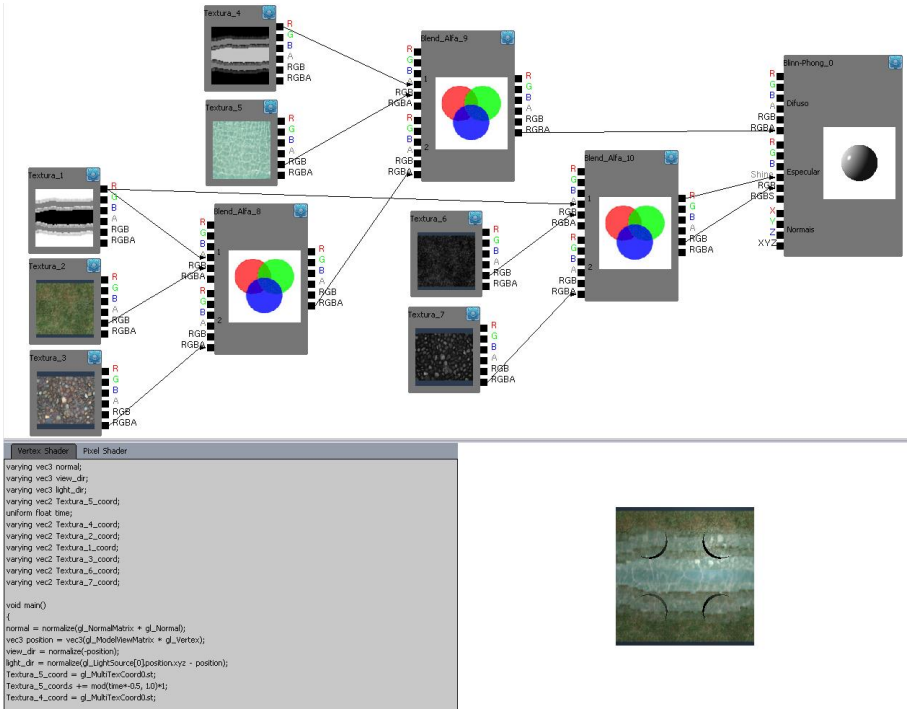


Figura 4.10 – Conexão completa

Vertex Shader | Pixel Shader

```
varying vec3 normal;
varying vec3 view_dir;
varying vec3 light_dir;
varying vec2 Textura_5_coord;
uniform float time;
varying vec2 Textura_4_coord;
varying vec2 Textura_2_coord;
varying vec2 Textura_1_coord;
varying vec2 Textura_3_coord;
varying vec2 Textura_6_coord;
varying vec2 Textura_7_coord;

void main()
{
normal = normalize(gl_NormalMatrix * gl_Normal);
vec3 position = vec3(gl_ModelViewMatrix * gl_Vertex);
view_dir = normalize(-position);
light_dir = normalize(gl_LightSource[0].position.xyz - position);

Textura_5_coord = gl_MultiTexCoord0.st;
Textura_5_coord.s += mod(time*-0.5, 1.0)*1;

Textura_4_coord = gl_MultiTexCoord0.st;

Textura_2_coord = gl_MultiTexCoord0.st;
Textura_2_coord.s *= 2;
Textura_2_coord.t *= 2;

Textura_1_coord = gl_MultiTexCoord0.st;

Textura_3_coord = gl_MultiTexCoord0.st;
Textura_3_coord.s *= 2;
Textura_3_coord.t *= 2;

Textura_6_coord = gl_MultiTexCoord0.st;
Textura_6_coord.s *= 2;
Textura_6_coord.t *= 2;

Textura_7_coord = gl_MultiTexCoord0.st;
Textura_7_coord.s *= 2;
Textura_7_coord.t *= 2;

gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Figura 4.11 – Código gerado para o vertex shader

Vertex Shader | Pixel Shader

```
varying vec3 normal;
varying vec3 view_dir;
varying vec3 light_dir;
varying vec2 Textura_5_coord;
uniform sampler2D Textura_5;
varying vec2 Textura_4_coord;
uniform sampler2D Textura_4;
varying vec2 Textura_2_coord;
uniform sampler2D Textura_2;
varying vec2 Textura_1_coord;
uniform sampler2D Textura_1;
varying vec2 Textura_3_coord;
uniform sampler2D Textura_3;
varying vec2 Textura_6_coord;
uniform sampler2D Textura_6;
varying vec2 Textura_7_coord;
uniform sampler2D Textura_7;

void main()
{
vec3 n;
vec3 vd = normalize(view_dir);
vec3 ld = normalize(light_dir);
vec3 h = normalize(ld + vd);

vec4 Blend_Alfa_10 = vec4(vec3(texture2D(Textura_6, Textura_6_coord).r, texture2D(Textura_6, Textura_6_coord).g, texture2D(Textura_6, Textura_6_coord).b)*texture2D(Textura_1, Tex
tura_1_coord).r + vec3(texture2D(Textura_7, Textura_7_coord).r, texture2D(Textura_7, Textura_7_coord).g, texture2D(Textura_7, Textura_7_coord).b)*texture2D(Textura_7, Textura_7_c
oord).a*(1.0-texture2D(Textura_1, Textura_1_coord).r), texture2D(Textura_1, Textura_1_coord).r+texture2D(Textura_7, Textura_7_coord).a*(1.0-texture2D(Textura_1, Textura_1_coord).r));

Blend_Alfa_10.rgb /= Blend_Alfa_10.a;

vec4 Blend_Alfa_8 = vec4(vec3(texture2D(Textura_2, Textura_2_coord).r, texture2D(Textura_2, Textura_2_coord).g, texture2D(Textura_2, Textura_2_coord).b)*texture2D(Textura_1, Text
ura_1_coord).r + vec3(texture2D(Textura_3, Textura_3_coord).r, texture2D(Textura_3, Textura_3_coord).g, texture2D(Textura_3, Textura_3_coord).b)*texture2D(Textura_3, Textura_3_co
ord).a*(1.0-texture2D(Textura_1, Textura_1_coord).r), texture2D(Textura_1, Textura_1_coord).r+texture2D(Textura_3, Textura_3_coord).a*(1.0-texture2D(Textura_1, Textura_1_coord).r));

Blend_Alfa_8.rgb /= Blend_Alfa_8.a;

vec4 Blend_Alfa_9 = vec4(vec3(texture2D(Textura_5, Textura_5_coord).r, texture2D(Textura_5, Textura_5_coord).g, texture2D(Textura_5, Textura_5_coord).b)*texture2D(Textura_4, Tex
tura_4_coord).r + vec3(Blend_Alfa_8.r, Blend_Alfa_8.g, Blend_Alfa_8.b)*Blend_Alfa_8.a*(1.0-texture2D(Textura_4, Textura_4_coord).r), texture2D(Textura_4, Textura_4_coord).r+Blend_Alf
a_8.a*(1.0-texture2D(Textura_4, Textura_4_coord).r));

Blend_Alfa_9.rgb /= Blend_Alfa_9.a;
vec4 diffuse_color = Blend_Alfa_9;
vec3 specular_color = Blend_Alfa_10.rgb;
float shininess = Blend_Alfa_10.r*255.0;
n = normalize(normal);
if(!gl_FrontFacing){
n *= -1.0;
}
vec3 ambient = 0 * diffuse_color.rgb;
vec3 diffuse = gl_LightSource[0].diffuse.rgb * diffuse_color.rgb * max(dot(ld,n), 0.0);
vec3 specular = gl_LightSource[0].specular.rgb * specular_color * pow(max(dot(h,n), 0.0), 2.0*shininess);
gl_FragColor = vec4(ambient+diffuse+specular, diffuse_color.a);
}
```

Figura 4.12 – Código gerado para o fragment shader

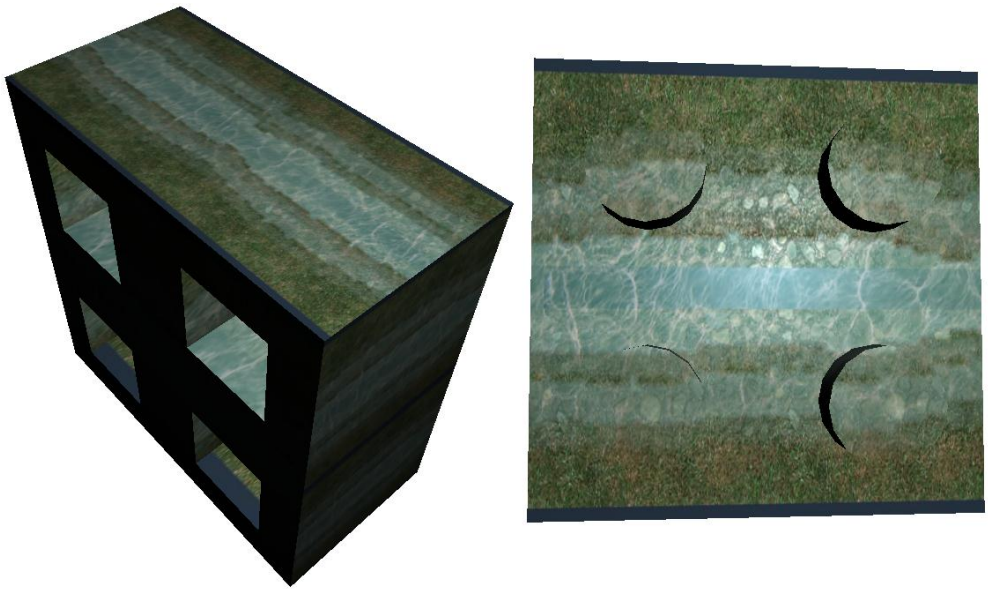


Figura 4.13 – Shader criado

5 CONCLUSÃO

Este trabalho apresentou a implementação de um editor visual de *shaders standalone* e que gera código GLSL. Este trabalho pode ser utilizado para facilitar a criação de *shaders* de materiais através de uma interface de simples utilização.

Primeiramente, foi realizado um estudo de técnicas de renderização que poderiam ser implementadas através de um grupo limitado de nós de edição que fossem de simples compreensão por parte do usuário. Após, foi realizada a análise de algumas ferramentas semelhantes existentes, buscando definir um grupo de componentes essencial para edição e adicionar um novo componente que auxiliasse a introduzir o usuário à implementação de *shaders*.

Implementou-se então um editor visual que tem como objetivos principais apresentar uma interface de edição simples, porém eficaz, e introduzir o usuário à implementação de *shaders*. Foram então adicionados painéis de configuração aos nós de edição, o que reduz a quantidade de componentes e agrupa funcionalidades relacionadas. Com a adição de um painel de visualização de código, alcançaram-se os objetivos iniciais.

Após a implementação da ferramenta, fez-se uma avaliação de sua utilização para a criação de um *shader* que simula um terreno com água corrente. Pôde-se observar que conceitos matemáticos e de programação foram efetivamente abstraídos do processo de edição, e o painel de visualização de código permite que o usuário veja o código gerado de uma maneira não invasiva, facilitando o contato inicial com a linguagem.

A API SCV mostrou-se bastante útil para a implementação de uma interface customizada e responsiva. Porém, um obstáculo encontrado foi a limitação de alguns *widgets* padrão, como o *scv::Panel* e seus derivados. Este componente não atualiza sua dimensão nem a de seus filhos quando a janela é redimensionada, o que resulta em uma interface de dimensão constante. Para contornar este obstáculo foi implementada a classe *ResizablePanel*, que reimplementa as *callbacks render* e

onResize da *scv::Panel*. Outra limitação da API é a incapacidade de remover componentes da lista de renderização, o que resulta em um aumento permanente no consumo de memória.

Embora tenha-se alcançado os objetivos iniciais, ainda há o que ser feito. O sistema de edição está limitado ao modelo de iluminação Phong. Para trabalhos futuros, sugere-se adicionar novos nós de edição, para incorporar a componente de iluminação emissiva e efeitos de reflexão e refração. Outra melhoria a ser realizada diz respeito à combinação de efeitos visuais completos. Este recurso pode ser implementado através da mudança do nó mestre para nó de processamento. Desta forma, a saída do programa seria uma cor no formato RGBA, e nós de processamento poderiam aplicar cálculos de modelos de iluminação em qualquer estágio do grafo. Com a implementação destes recursos, obter-se-ia uma ferramenta com mais funcionalidades de edição e que mantém a facilidade de utilização.

REFERÊNCIAS

BIOHAZARD. In: Source Shader Editor Showcase · Share your creations. Disponível em: <http://sourceshadereditor.com>. Acesso em: 29 nov. 2011.

BLINN, J. F. Simulation of Wrinkled Surfaces. In: PROC. 5TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1978, Nova Iorque. **Anais eletrônicos...** Nova Iorque: ACM, 1978.

COHEN, J.; OLANO, M.; MANOCHA, D. Appearance-Preserving Simplification. In: Proceedings of the 25th annual conference on computer graphics and interactive techniques, 1998, Orlando. **Anais eletrônicos...** Nova Iorque: ACM, 1998.

EPIC GAMES. In: Unreal Technology. Disponível em: <http://www.unrealengine.com>. Acesso em: 29 nov. 2011.

KANEKO, T. et al. Detailed Shape Representation with Parallax Mapping. In: Proceedings of the ICAT, 2001, Tóquio. **Anais eletrônicos...** Tóquio: ICAT, 2001.

LIGHTHOUSE3D. In: Creating a Program. Disponível em: <http://www.lighthouse3d.com/tutorials/glsl-tutorial/creating-a-program/>. Acesso em 16 ago. 2011.

LIGHTHOUSE3D. In: Creating a Shader. Disponível em: <http://www.lighthouse3d.com/tutorials/glsl-tutorial/creating-a-shader/>. Acesso em 16 ago. 2011.

LIGHTHOUSE3D. In: GLSL 1.2 Tutorial – Pipeline Overview. Disponível em: <http://www.lighthouse3d.com/tutorials/glsl-tutorial/pipeline-overview/>. Acesso em 16 ago. 2011.

MCGUIRE, M. et al. Abstract Shade Trees. **Brown University GVI**, Providence, 2006. Disponível em: <http://graphics.cs.brown.edu/games/AbstractShadeTrees/abstract-shade-trees-2006.pdf>. Acesso em: 16 ago. 2011.

OLIVEIRA, M. M.; BISHOP, G.; MCALLISTER, D. Relief Texture Mapping. In: Proceedings of SIGGRAPH, 2000, Nova Orleans. **Anais eletrônicos...** Nova Iorque: ACM, 2000.

OPENGL. In: OpenGL, The Industry Standard for High Performance Graphics. Disponível em: <http:// www.opengl.org>. Acesso em: 16 ago. 2011.

PAHINS, C. A. L.; LIMBERGER, F. A.; HENZ, B.; SPERONI, E. A.; GOTTIN, V. M.; POZZER, C. T. Uma API Livre para Composição de GUI em APlicativos Gráficos. Forum Internacional de Software Livre 2010 - Workshop de Software Livre, Porto Alegre, RS, BR, 2010.

PHONG, B. T. Illumination for Computer Generated Pictures. In: Communications of the ACM, 1975, Nova Iorque. **Anais eletrônicos...** Nova Iorque: ACM, 1975.

ROST, R. J. et al. **OpenGL Shading Language.** 3. ed. Addison-Wesley Professional, 2009. 792 p.

STRUMPY.NET. In: strumpy.net, Programming and Game Design in Unity with Stramit and Texel. Disponível em: <http://www.strumpy.net>. Acesso em: 29 nov. 2011.

UNITY TECHNOLOGIES. In: UNITY: Game Development Tool. Disponível em: <http://unity3d.com>. Acesso em: 29 nov. 2011.

WEIBLEN, M. OpenGL Shading Language Quick Reference Guide. Nederland: 2005. Disponível em: <http://mew.cx/glsl_quickref.pdf>. Acesso em: 28 nov. 2011.

WIKIPEDIA CONTRIBUTORS. In: Cel-shaded animation, Wikipedia, The Free Encyclopedia. Disponível em: <http://en.wikipedia.org/wiki/Cel-shaded_animation>. Acesso em: 16 ago. 2011.

WIKIPEDIA CONTRIBUTORS. In: Normal mapping, Wikipedia, The Free Encyclopedia. Disponível em: <http://en.wikipedia.org/wiki/Normal_mapping>. Acesso em: 16 ago. 2011.

WIKIPEDIA CONTRIBUTORS. In: Phong reflection model, Wikipedia, The Free Encyclopedia. Disponível em: <http://en.wikipedia.org/wiki/Phong_reflection_model>. Acesso em: 16 ago. 2011.

WOLFF, D. **OpenGL 4.0 Shading Language Cookbook.** Packt Publishing, 2011. 340 p.

WRIGHT, R. S. **OpenGL SuperBible: Comprehensive Tutorial and Reference.** 5. ed. Addison-Wesley Professional, 2010. 1008 p.