Shaderlib: Uma Biblioteca Para Desenvolvimento e Aprendizado de *Shaders* com GLSL

Vitor de Andrade[1], **Ricardo Marroquim**[2]

[1]Escola Politécnica - Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro - RJ - Brasil

[2]COPPE - Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro - RJ - Brasil

vitor.andrade@poli.ufrj.br, marroquim@cos.ufrj.br

Abstract. *In the past decades, besides experiencing a huge development in terms of computation speed, we have also experienced the emergence of the Programmable GPU, giving birth to languages like GLSL and CUDA. This technology gives great flexibility for the usage this powerful hardware, attracting the interest of many researchers and programmers to this area. More recently, the release of OpenGL 4 provided even more flexibility to users, but has turned the learning curve steeper, since even basic applications demand knowledge of many Linear Algebra concepts and proficiency in Shaders and OpenGL. This paper focus in a shaders library aimed to help students, researchers and programmers in the usage of GPU programming with OpenGL 4.*

Resumo. *Nas últimas décadas, além de vivenciarmos uma enorme evolução da velocidade computacional, ocorreu o surgimento das GPUs programáveis, dando origem a linguagens como GLSL e CUDA. Isto proporcionou uma grande flexibilidade na utilização do hardware e atraiu muitos interessados. Mais recentemente, o lançamento do OpenGL 4 proveu ainda mais flexibilidade para os usuários, porém tornou a curva de aprendizado mais íngreme, já que mesmo aplicações básicas demandam o conhecimento de muitos conceitos de Álgebra Linear e proficiência em programação em Shaders e OpenGL. Este artigo apresenta uma biblioteca de shaders que visa auxiliar os estudantes, pesquisadores e programadores na utilização do OpenGL 4.*

1. Introdução

Durante a última decada, a programação em *shaders* evoluiu de um *pipeline* de funcionalidade fixa para um programável. Além disso, o número de *shaders* programáveis também aumentou. Inicialmente havia apenas o par *vertex shader* e *fragment shader*, porém atualmente já existem seis estágios programáveis, como apresentado na Figura 1. As novas versões do *OpenGL* e *GLSL* também adicionaram um extenso número de novas funcionalidades e controle de acesso para programação em GPU.

Por um lado, a criação do *pipeline* gráfico programável adicionou uma grande flexibilidade e proveu um recurso extremamente poderoso para *stream programming*. Por outro lado, quando o *OpenGL 4* foi lançado, as funcionalidades fixas tornaram-se obsoletas, forçando os programadores a implementarem seus próprios *shaders* para lidar com

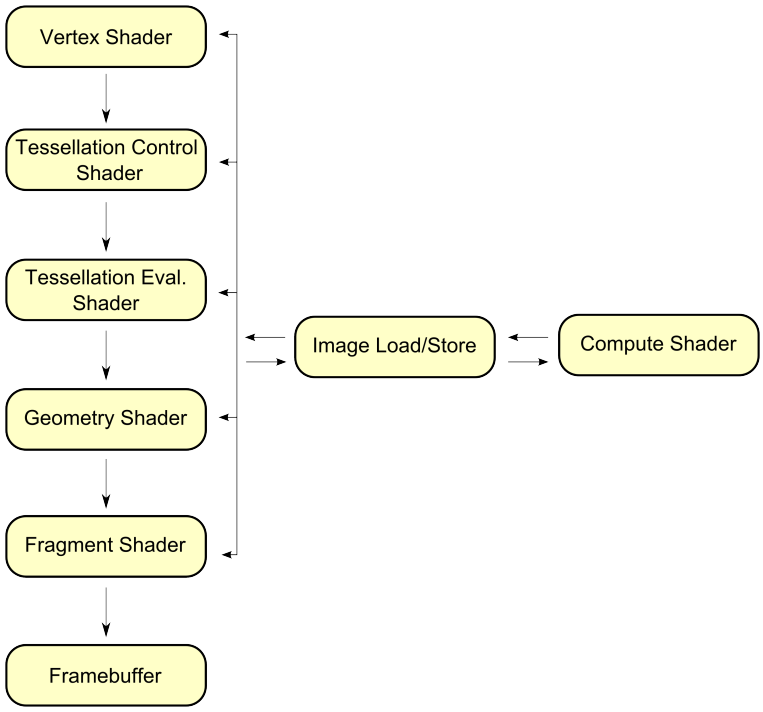

Figura 1. Esquema Simplificado do Pipeline Gráfico do *OpenGL*

todas as transformações (e.g. construção das matrizes *Model*, *View* e *Projection*), mesmo para aplicações simples. Desta forma, a curva de aprendizado para iniciantes tornou-se muito mais íngreme e a quantidade de código necessário para iniciar o desenvolvimento de uma aplicação aumentou significativamente. Uma introdução à programação com *OpenGL 4* pode ser encontrada no livro online *OpenGL Book* [Luten].

Este artigo apresenta uma biblioteca cujos objetivos são de abstrair e encapsular grande parte dos códigos de inicialização, que normalmente são repetidos na maior parte das aplicações. Isto ajuda no aprendizado de programação em GPU e permite que o desenvolvimento de aplicações gráficas seja feito mais rapidamente, uma vez que o programador pode focar diretamente no desenvolvimento da aplicação. Em seguida, são descritas algumas aplicações desenvolvidas com esta biblioteca. Mais informações podem ser encontradas *online* em [Andrade and Marroquim 2013].

Existem outras bibliotecas com propostas similares, como *ShaderLabs* [Vianna et al. 2012], [Gomes et al. 2012] e *SpiderGL* [Di Benedetto et al. 2010]. A primeira foca no aprendizado de programação em *shaders* e a segunda é uma API para programação em *shader* sob *WebGL*. Diferentemente da *ShaderLabs*, o nosso objetivo é criar uma API para reduzir o tempo de programação e facilitar o uso de recursos. Mais ainda, focamos nas versões mais modernas das bibliotecas gráficas, i.e. *OpenGL* 4 e *GLSL* 4, para suportar os recursos mais recentes, tal como os *Compute Shaders*. É importante notar que o objetivo da biblioteca não é melhorar a performance ou otimizar quaisquer algoritmos existentes. Ela apenas traduz as chamadas de *OpenGL* e *GLSL* para chamadas mais simples e intuitivas. Desta forma, não há impacto na performance causado pela sua utilização.

2. Biblioteca de Shaders

2.1. Estrutura Básica da Biblioteca

A biblioteca introduz a ideia de um Efeito (*Effect*) sendo qualquer algoritmo que se deseja implementar utilizando a programação em GPU. A maior parte do código específico da aplicação é então implementada através da classe *Effect*, auxiliada pelas outras classes. Atualmente, a biblioteca consiste em 7 classes, que serão explicadas em detalhes nas seções seguintes.

2.2. Classe *Effect*

A classe *Effect* é a classe base usada para o desenvolvimento da aplicação. Uma aplicação típica em GPU desenvolvida com esta biblioteca estende esta classe para conter a maior parte do corpo da aplicação. Ela tipicamente lida com quaisquer pré-computações em CPU e métodos de renderização.

Como descrito na seção de Aplicações (Sec. 3), um efeito pode ser utilizado para representar, por exemplo, qualquer tipo de *shading effect* aplicado a uma malha 3D, um *ray tracer* baseado em GPU ou até mesmo cálculos não-gráficos em GPGPU, mesmo que este não seja realmente o foco da biblioteca.

2.3. *Framebuffer*

Esta classe é responsável pela geração e armazenamento do *framebuffer*. Um FBO pode conter muitas texturas que podem ser utilizadas para leitura e escrita durante as etapas de *shader*. Usualmente eles são usados para *offscreen rendering*, escrita em múltiplos *buffers* ou técnicas de multipassadas. Apesar destas técnicas serem muito comuns em programação em GPU, a criação e uso de um FBO é extremamente tediosa. Esta classe abstrai a maior parte da configuração e permite que o programador crie e use os *framebuffers* com poucas linhas de código. Ela lida com a criação das texturas e dos *buffers*, assim como seu *binding* e utilização, que envolvem alocar *slots* de textura e passá-los para os *shaders*.

Um FBO pode ser criado em apenas uma linha:

```
Framebuffer* fbo = new Framebuffer(w, h, num_buffers, texType);
```

Onde *w* e *h* são a largura e altura do *framebuffer*, *num_buffers* é o número de texturas (ou *attachments*) que o FBO armazenará e *textype* é o tipo do *texture object* do *framebuffer* (ex. *GL_TEXTURE_2D*).

Na hora da utilização do *framebuffer*, a biblioteca automaticamente procura um *slot* de textura vazio, liberando-o após a sua utilização. Isto significa que a biblioteca é responsável pelo gerenciamento de todos os *attachments*, enquanto o usuário deve se preocupar apenas com o *binding* e *unbinding* do *framebuffer*, de acordo com suas necessidades:

```
fbo->bind();
fbo->bindAttachment(attachment);
//... Application code in here ...
fbo->unbind();
```

A segunda linha de código liga a textura relacionada a um determinado *attachment* do *framebuffer* ao primeiro *slot* de textura disponível, retornando o número do *slot* utilizado. Existem também muitas outras funções para executar operações importantes do *framebuffer*, como por exemplo a leitura do *buffer*, o *binding* de um *attachment* a uma unidade de textura específica, entre outros. Para ilustrar um exemplo comum em programação em GPU, a seguinte linha de código indica como passar uma textura do FBO como entrada para um *shader*:

```
shader->setUniform("textureName", fbo->bindAttachment(attachment
    ));
```

Note que o programador não tem que trabalhar explicitamente com a identificação da unidade de textura. A textura do *FBO* pode ser prontamente acessada nos *shaders* através da chamada:

```
uniform sampler2D textureName;
```

2.4. *Mesh*

Esta é uma classe auxiliar responsável por carregar e trabalhar com arquivos típicos de malhas. Ela tem um *loader* de arquivos do tipo *wavefront obj*. Seu principal objetivo, entretanto, é lidar com a criação e gerenciamento dos *Buffer Objects*. Como o *OpenGL* 4 tornou obsoleta a forma usual de criação de polígonos, vetores normais, coordenadas de textura, etc., tudo deve agora ser diretamente armazenado em GPU com a utilização dos *Buffers*. Entretanto, a sua configuração é, novamente, longa e normalmente muito semelhante de aplicação para aplicação.

Tipicamente o programador cria *arrays* para armazenar a informação de malha obtida do arquivo. Então os *buffers* são criados e os dados são carregados efetivamente para a GPU. Estas operações são traduzidas em um grande número de linhas de configuração para simplesmente carregar e renderizar uma malha triangular, por menor que ela seja. Mais informações sobre a utilização de *buffers* pode ser encontrada em [Andrade and Marroquim 2012].

A classe *Mesh* é importante porque ela poupa o programador de todo este trabalho. Elas são criadas com uma única linha de código e a configuração dos *buffers* é feita com poucas linhas:

```
Mesh* mesh = new Mesh;
mesh->createBuffers();
mesh->loadOBJFile("myMesh.obj");
```

Os *buffers* podem ser acessados dentro do *shader* através dos *vertex attribute locations*, utilizando o *layout qualifier*. Estes são números gerenciados pelo usuário identificando aonde cada *buffer* está atrelado. Entretanto, em nossa biblioteca os atributos mais comuns de uma malha triangular têm *locations* pré-definidos de acordo com a Tabela 1.

Além disso, a biblioteca lida com o *binding* de todos os *buffers*, então a malha pode ser renderizada facilmente com apenas uma linha de código:

```
mesh->render();
```

Tabela 1. *ATTRIBUTE LOCATIONS* Definidos na Classe *MESH*

Location	Buffer
0	Vertex Buffer
1	Normals Buffer
2	Colors Buffer
3	Texture Coordinate Buffer

Nos *shaders*, os atributos estão prontos para serem acessados. Por exemplo, no *vertex shader*, fazemos o acesso aos vértices e às normais da seguinte forma:

```
layout(location=0) in vec4 in_Position;
layout(location=1) in vec3 in_Normal;
```

Ainda mais, quando a malha é carregada, a biblioteca também executa alguns cálculos geométricos. Por exemplo, seu centro é detectado utilizando um algoritmo de *axis-aligned bounding box* e fatores de normalização. Estes são normalmente úteis para manipulação de modelos 3D.

Finalmente, esta classe contém também algumas funções para gerar malhas básicas, e.g. um quadrilátero (interessante para *off-screen rendering* sem o uso de *compute shaders*) ou um cubo para testes básicos.

2.5. *Shaders*

2.5.1. Inicialização dos *Shaders*

A classe *Shader* contém tudo aquilo que é necessário para a criação de um programa *Shader*. Apesar de o *OpenGL* receber o *shader* como uma *string*, é mais interessante que o usuário escreva o código em arquivos de texto externos, deixando a aplicação encarregada de carregar os arquivos e inicializar o *shader*. Esta classe facilita quase toda a configuração dos *shaders*. Ela automaticamente busca um dado diretório por quaisquer *shaders* com um dado nome, criando e inicializando o seu programa.

Por exemplo, se o usuário tem *shaders* do tipo *vertex* e *fragment* definidos como *myShader.vert* e *myShader.frag*, localizados no diretório *myShadersDir*, as seguintes linhas de código são utilizadas para a inicialização completa do programa:

```
Shader* shader = new Shader("myShadersDir", "myShader");
shader->initialize();
```

A biblioteca irá então automaticamente procurar o diretório escolhido por quaisquer extensões de arquivo de *shaders*, i.e., *.vert*, *.frag*, *.geom*, *.comp*, ou *.tess*.

A partir daí, uma simples chamada ao método *enable* ou *disable* é suficiente para ativar ou desativar um *shader* para a renderização.

2.5.2. *Uniform Setting*

Em *OpenGL* 4, o usuário deve especificar previamente o tipo de variável a ser carregada nos *shaders*, já que existe uma função específica para cada tipo de variável que pode ser

carregada. Mais ainda, ele deve manter os valores dos *locations* correspondentes para cada uma das variáveis. Para simplificar este trabalho, a biblioteca encapsulou todos estes métodos, tornando o carregamento das *uniforms* mais intuitivo. Elas podem ser carregadas pelo número do *location* ou pelo nome da variável dentro do *shader*.

Todas as funções existentes para o carregamento de *uniforms* foram sobrescritas para um único método: **setUniform**. Este método chama a função apropriada do *OpenGL* dependendo dos parâmetros recebidos. Por exemplo, se o usuário precisa carregar 3 inteiros - *x*, *y*, *z* para um $ivec3$ de nome *myVector* dentro do shader, ele deve simplesmente chamar:

```
shader->setUniform("myVector",x,y,z);
```

Em vez da chamada específica para:

```
glUniform3i(uniformLocation,x,y,z);
```

Da mesma forma, uma chamada para:

```
shader->setUniform("myFloat", f);
```

Será traduzida para:

```
glUniform1f(uniformLocation,f);
```

Note entretanto que sem a biblioteca, o programador deveria ainda manter ou perguntar os *locations* destas variáveis.

2.5.3. Recarregando *Shaders*

Uma funcionalidade simples, porém muito útil, desta classe, é o método para recarregar os *shaders* em tempo de execução. O programador é capaz de modificar o *shader* e então imediatamente verificar os resultados sem ter que reiniciar a aplicação.

2.6. *Trackball*

Para lidar com transformações comuns e necessárias para uma aplicação de visualização, nós incluímos uma classe de *Trackball* na biblioteca. Ela utiliza a biblioteca *Eigen* ([Guennebaud et al.]) para a maioria das transformações. A classe *quaternion* é utilizada para as rotações gerais e o *template* de *affine matrix* é usado para as transformações do objeto e a matriz de projeção. Para reproduzir o *pipeline* usual do *OpenGL*, nós armazenamos estas transformações em três matrizes (*Model*, *View* e *Projection*) que são passadas como *uniforms* para os *shaders* quando necessário. Além disso, já que o *trackball* lida com quaisquer transformações gerais, também é possível criar um *trackball* para o posicionamento da iluminação. A classe *Trackball* possui ainda um *shader* responsável pela sua representação visual na tela.

2.7. *Texture Manager*

Normalmente, para que sejam utilizadas texturas nas aplicações gráficas, é necessário que o programador mantenha os números dos *slots* para cada textura utilizada, o que se

reflete em algumas variáveis e linhas de código a mais na aplicação. Além disso, requer a atenção do programador para quais unidades já foram utilizadas e quais ainda estão disponíveis. Para um número elevado de texturas utilizadas, se torna recomendável que o programador desenvolva algum algoritmo que gerencie a utilização destas unidades de textura. Pensando em simplificar e abstrair este trabalho, foi desenvolvida uma classe chamada *TextureManager*. Ela possui um vetor que identifica quais unidades de textura estão atualmente em uso e busca uma unidade disponível no momento em que o *binding* for solicitado. O *binding* das texturas pode ser efetuado chamando os métodos desta classe, passando o número de identificação das texturas como parâmetro ou através da utilização da classe *Texture*, explicada em mais detalhes na seção 2.8.

Esta classe possui ainda algumas funcionalidades extras como a de retornar o número da primeira unidade de textura disponível ou marcar uma determinada unidade como disponível ou indisponível.

É importante notar que, uma vez que esta classe armazena as informações de disponibilidade de unidade de textura, devemos garantir que exista apenas um objeto desta classe instanciado na aplicação. Caso existisse mais de um objeto deste tipo, suas informações poderiam entrar em conflito e o gerenciamento não seria feito de forma adequada. Para garantir a correta utilização do gerenciador, esta classe foi desenvolvida através do *design pattern Singleton*, que será explicado em mais detalhes na próxima subseção.

2.7.1. O Padrão *Singleton*

O padrão *Singleton* é um *design pattern* utilizado quando existem duas condições indispensáveis em uma classe. Primeiro, é primordial que exista apenas um objeto desta classe em toda a aplicação. Segundo, devemos ter acesso global a este objeto.

O desenvolvimento de uma classe com este padrão é bem simples. Basicamente, é necessário que a classe possua os métodos construtor, *copy constructor* e *operator=* privados, para garantirmos que um objeto desta classe nunca será criado externamente, diretamente da aplicação. Além disso, ela deve ter um ponteiro estático, também privado, para um objeto do tipo *TextureManager*. Este será o único objeto existente desta classe na aplicação. O acesso a este vetor se dá através de uma função estática pública, chamada **Instance()**, implementada da seguinte forma:

```
TextureManager* TextureManager::Instance() {
        if (!pInstance) {
            pInstance = new TextureManager;
        }
return pInstance;
}
```

Na primeira vez em que a aplicação acessar o gerenciador, este método irá instanciar um novo objeto do tipo *TextureManager* e retornar o ponteiro para este objeto. A partir daí, este método irá apenas retornar este ponteiro, garantindo que nenhum outro objeto deste tipo seja criado durante a aplicação.

Normalmente, o acesso a um objeto criado a partir deste padrão é feito através de uma chamada a:

```
TextureManager::Instance()
```

Porém, para tornar o acesso mais intuitivo e rápido, em nossa biblioteca acrescentamos uma diretiva de pré-processador nesta classe, como indicado a seguir:

```
#define texManager TextureManager::Instance()
```

Com isso, o acesso a este objeto é feito através de chamadas a *texManager*, como se este fosse um ponteiro para um objeto *TextureManager*. Por exemplo, uma chamada a

```
texManager->bindTexture(GL_TEXTURE_2D, texID);
```

É traduzida para

```
TextureManager::Instance()->bindTexture
(GL_TEXTURE_2D, texID);
```

2.8. Texture

Há ainda uma classe para armazenar as informações sobre um objeto de textura, como por exemplo seu número de identificação (*texID*) e parâmetros como o tipo de textura, dimensões, etc. Esta classe contém diversos métodos visando a facilitar o *binding* das texturas, através da utilização do *Texture Manager*. Por exemplo, o código a seguir ilustra os passos necessários para a criação e o *binding* de uma textura 2D com alguns parâmetros comumente utilizados, de dimensões *texWidth* e *texHeight*, a ser utilizada no shader *myShader* através da *uniform "mySampler"*:

```
//Creating:
Texture* myTexture = new Texture();
myTexture->create(GL_TEXTURE_2D,GL_RGBA, texWidth, texHeight,
    GL_RGBA, GL_FLOAT);

//Binding:
myShader->setUniform("mySampler",myTexture->bind());

//Set other uniforms and call rendering methods...

//Unbinding:
myTexture->unbind();
```

O método *bind()* busca a primeira unidade de textura disponível, efetua o *binding* e retorna o valor desta unidade. Desta forma, o programador em nenhum momento precisou armazenar o valor da unidade de textura ao qual o *binding* é feito. Este valor é enviado diretamente ao shader, onde o acesso à textura é feito através da chamada:

```
uniform sampler2D mySampler;
```

Algumas funcionalidades extras desta classe incluem o *binding* a uma determinada unidade de textura, a atualização do conteúdo da textura mantendo-se os mesmos parâmetros e o *binding* da textura como uma *Image Texture*.

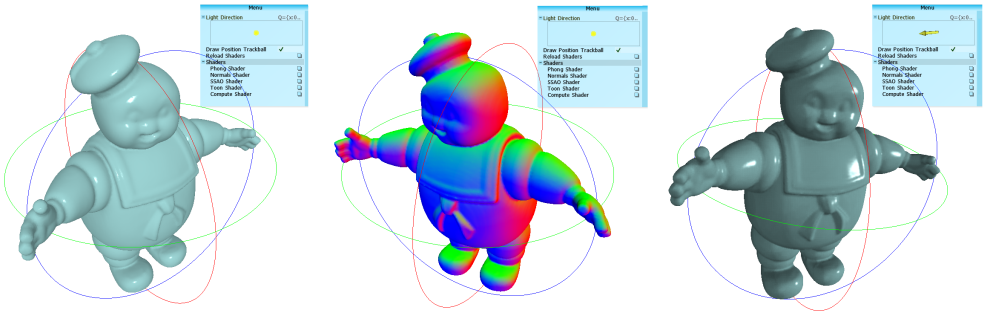

Figura 2. Exemplo de Aplicação: Visualizador de *Shading Effects*. Três diferentes efeitos sendo aplicados a uma malha: *Phong Shading* (esquerda), Mapeamento de Normais para Cor (meio), *Screen Space Ambient Occlusion (direita).*

3. Aplicações

Nós desenvolvemos algumas aplicações para demonstrar o uso da biblioteca nos diferentes contextos. Elas serão brevemente explicadas nas subseções seguintes.

3.1. Visualizador de Shading Effects

3.1.1. A Aplicação

Esta aplicação consiste em uma plataforma para aplicar diferentes efeitos de *shader* em uma malha 3D (ver Figura 2). Ela usa GLFW [Geelnard and Berglund] para criar um contexto de janela, AntTweakBar [Decaudin] para os menus e Eigen [Guennebaud et al.] para álgebra linear (tais como operações com vetores e matrizes). O fluxograma do sistema está ilustrado na Figura 3.

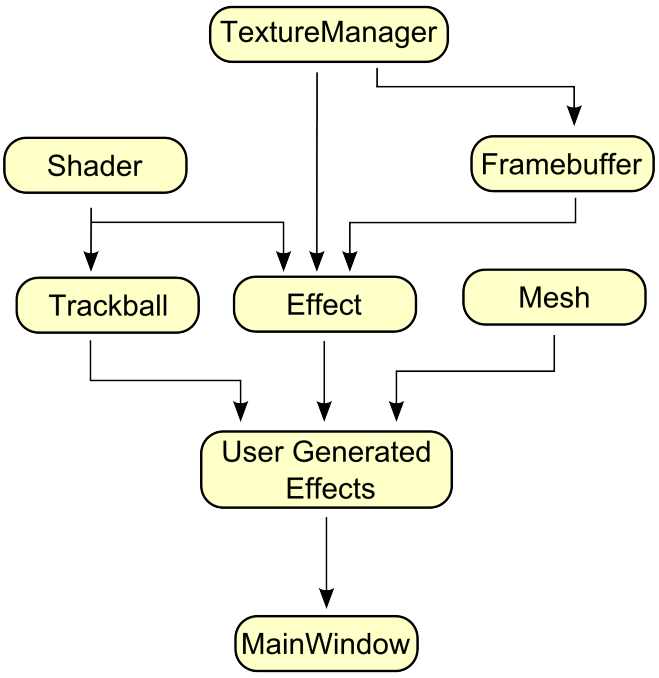

Figura 3. Fluxograma do Visualizador de *Shading Effects*

Apesar de um *shading effect* ser normalmente definido através da herança da classe *Effect*, quando o efeito é muito simples e não precisa de nenhum tipo de pré-computação ou armazenamento de variável (e.g. o *Phong Shading Effect*), ele pode ser implementado usando apenas a classe *shader*.

Na Figura 4 nós temos uma ilustração da aplicação. No lado direito da imagem localiza-se o menu contendo todos os efeitos disponíveis, algumas opções de configurações e a representação da direção da luz, que é manipulada usando o *light trackball*. Neste menu, o usuário pode escolher entre um dos *shading effects*, ajustar parâmetros para cada efeito e visualizá-los em tempo real.

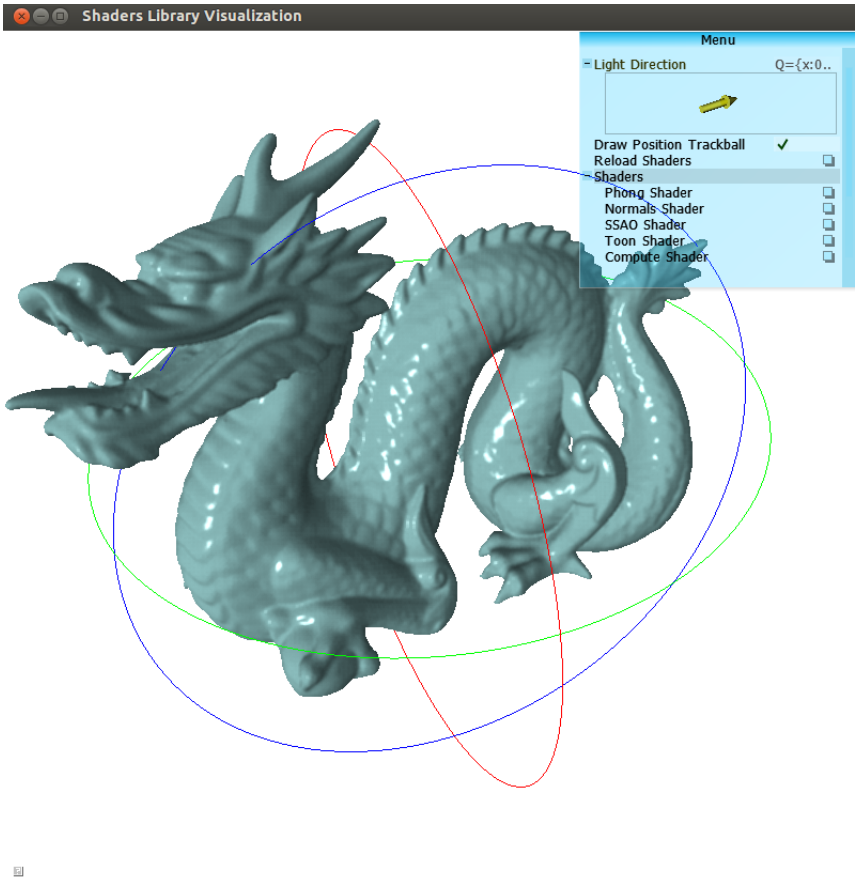

Figura 4. Interface do Visualizador de *Shading Effects*

Quando o programador deseja adicionar um novo efeito à aplicação, a classe *Effect* é estendida e dois métodos devem ser sobrescritos: *initialization* e *rendering*. Por exemplo, para criar o efeito *Toon*, a inicialização é muito simples, já que ele contém apenas um shader:

```
Toon::initialize() {
  shader = new Shader("shadersDir", "toonShader");
  shader->initialize();
}
```

A função de renderização ativa o *shader*, carrega as *uniforms* apropriadas, renderiza a malha e finalmente desabilita o *shader*:

```
shader->enable();

shader->setUniform("projectionMatrix", cameraTrackball->
    getProjectionMatrix(), matrixDimension);
//... Sets the other uniforms here...

mesh->render();

shader->disable();
```

Para usar um efeito na aplicação, basicamente tudo o que é necessário é passar o seu nome e o comprimento e largura dos *buffers* (normalmente iguais aos do *viewport*):

```
MyEffect myEffect = new MyEffect("effectName", bufferWidth,
    bufferHeight);
myEffect->initialize();
```

Finalmente, para renderizar a malha com o efeito selecionado, a aplicação apenas chama sua função de renderização, passando a malha e os *trackballs* como parâmetros:

```
activeEffect->render(mesh, cameraTrackball, lightTrackball);
```

Note que quaisquer combinações destes três parâmetros pode ser usada. Por exemplo, em uma aplicação simples de processamento de imagens, todos os três parâmetros seriam tipicamente configurados como *NULL*.

3.1.2. SSAO

Para dar um exemplo de um efeito de renderização menos trivial, nós descreveremos basicamente a implementação de um *Screen Space Ambient Occlusion*. Esta é uma aproximação do algoritmo de *Ambient Occlusion* calculado em tempo real diretamente na GPU. Para mais detalhes sobre esta técnica, consultar [Shanmugam and Arikan 2007], [Mittring 2007] e [Carlsson 2010].

O cálculo do *SSAO* consiste em um algoritmo de três passadas, todas implementadas diretamente dentro do método de renderização da classe herdada pela *Effect*, como pode ser visto na Figura 5. Note que a aplicação não difere qual efeito está sendo chamado, ela simplesmente chama a função de renderização do efeito ativo.

A primeira passada de renderização com *SSAO* consiste em um *off-screen rendering* para armazenar a informação da profundidade:

```
deferredShader->enable();

fbo->bindRenderBuffer(depthTextureAttachment);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

mesh->render();
```

```
fbo->unbind();
deferredShader->disable();
```

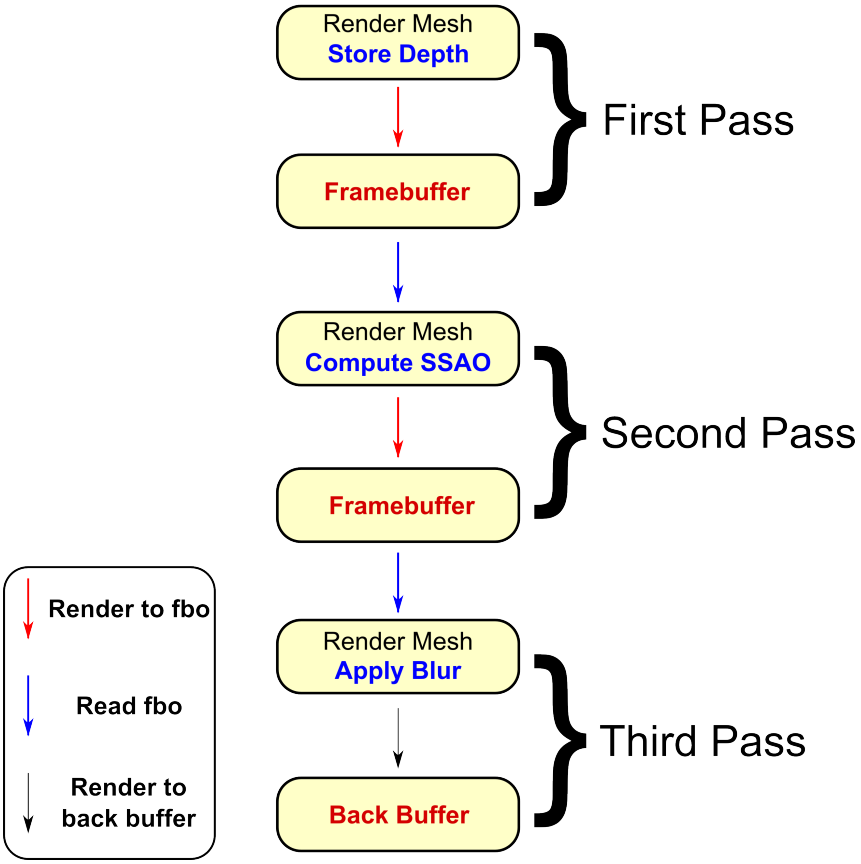

Figura 5. Fluxograma de Renderização do *Screen Space Ambient Occlusion*

Na segunda passada, a informação de profundidade obtida da primeira é usada para computar o termo de *Ambient Occlusion*. O resultado é renderizado novamente para uma textura que será usada na terceira passada para remover o ruído de alta frequência gerado pelo algoritmo *SSAO*.

```
ssaoShader->enable();

fbo->bindRenderBuffer(blurTextureAttachment);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

ssaoShader->setUniform("depthTexture",fbo->bindAttachment(
    depthTextureAttachment));
//... Sets other uniforms here...

mesh->render();

ssaoShader->disable();
fbo->unbindAttachments();
```

Na terceira passada, um *Gaussian Blur* é aplicado para remover o ruído da segunda passada:

```
blurShader->enable();
glDrawBuffer(GL_BACK);

blurShader->setUniform("blurTexture", fbo->bindAttachment(
    blurTextureAttachment));
//Sets other uniforms here...

mesh->render();

fbo->unbindAll();

blurShader->disable();
```

3.2. Ray Casting Baseado em GPU

Para mostrar como a biblioteca pode ser empregada na escrita de um método de renderização mais complexo, nós desenvolvemos um *Volume Render* simples utilizando *ray-casting*. A sua interface e a imagem de saída gerada pelo *ray-casting* podem ser vistas na Figura 6. Além da janela do visualizador, similar ao último exemplo, ele contém ainda uma classe armazenando uma textura 3D representando o Volume a ser visualizado.

Para explicar brevemente, uma resolução de imagem de saída é configurada e um raio é enviado para cada pixel. Cada raio atravessa o volume e amostra o valor de cor da textura do volume em cada iteração (*traversal step*). Os valores são acumulados e quando o percurso chega ao fim uma cor é atribuída ao pixel correspondente na imagem final, que é finalmente visualizada como um quadrilátero texturizado.

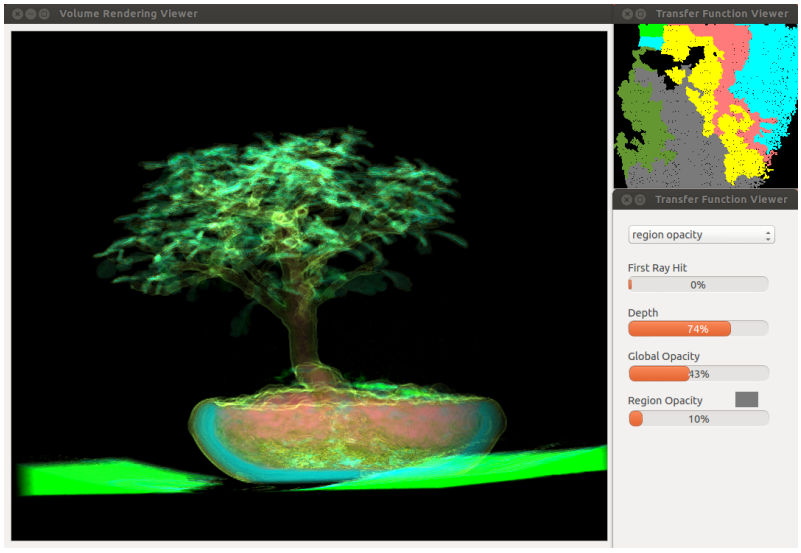

Figura 6. Uma renderização usando o *ray-casting* implementado com a nossa biblioteca (esquerda); uma janela de função de transferência 2D e parâmetros do *volume rendering* (direita).

Já que texturas de FBO não podem ser configuradas para leitura e escrita simultaneamente em uma passada, o *ray casting traversal* seria normalmente implementado através de um esquema *ping-pong* para uma implementação em GPU (Figura 7). Assim, o FBO leria de um *attachment* e escreveria no outro, trocando os papéis em cada passada. Outra possibilidade seria iterar diretamente dentro do *shader* usando uma estrutura em *loop*. Entretanto, esta seria proibitiva para *traversals* com muitos passos, ja que o número de iterações dentro do *loop* é limitado.

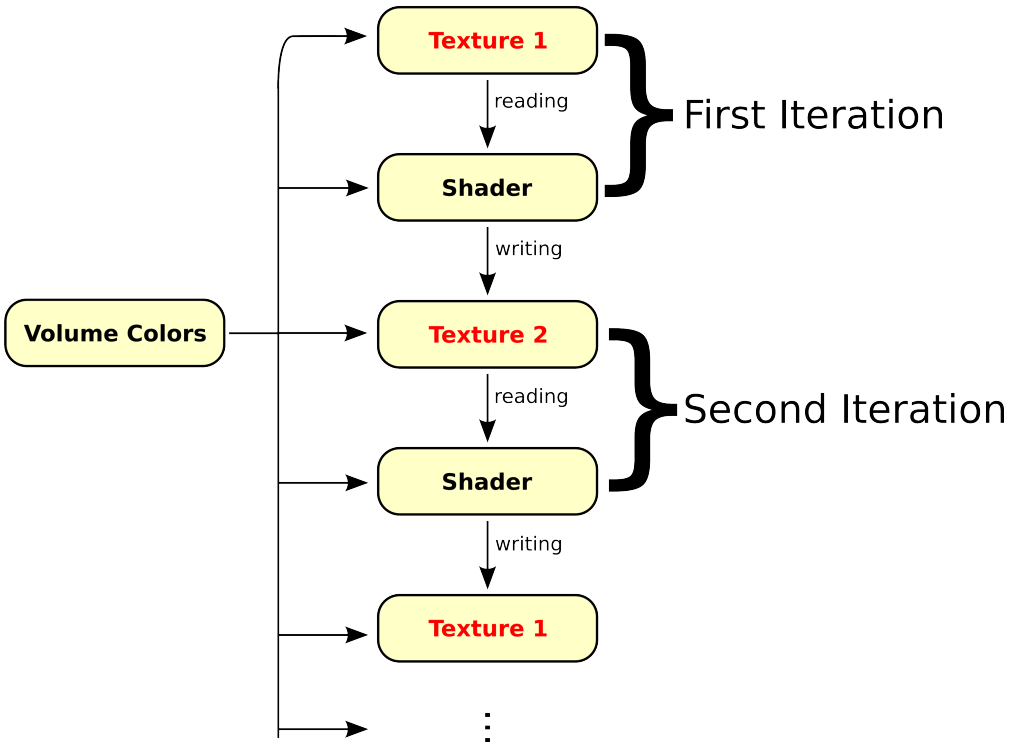

Figura 7. Estrutura de Renderização *Ping Pong* para o *Ray Casting*

Para demonstrar o poder dos novos *compute shaders*, nós implementamos o *traversal* evitando o esquema *ping-pong*. Com a funcionalidade de *Image Load/Store*, a implementação foi extremamente simplificada, já que uma imagem pode ser usada para leitura e escrita simultaneamente. O número de grupos chamados com o *compute shader* é igual ao tamanho do *viewport* em cada eixo, significando que o *compute shader* será chamado uma vez por fragmento, enviando um raio por pixel na tela. O *binding* das texturas e a chamada do *compute shader* está indicado em seguida:

```
//...Ray Casting setup definitions ...

computeShader->enable();

//Bind Textures for Shader Reading:
currentColorsTexture->bindImageRW();
volume->getTexture3D()->bind();

//... Set uniforms here...
```

```
//Call Compute Shader:
glDispatchCompute(viewportSize[0], viewportSize[1], 1);

//Unbind Textures:
currentColorsTexture->unbind();
volume->getTexture3D()->unbind();

computeShader->disable();
```

Novamente, as iterações que amostram através de um volume podem ser feitas fora do shader, chamando-o uma vez por passada, ou dentro do shader, amostrando múltiplas fatias em cada *shader*, diminuindo o número de chamadas.

4. Trabalhos Futuros

O próximo passo do desenvolvimento da biblioteca é a implementação do suporte ao *Tesselation Shader*, assim como exemplos de *Tesselation* e *Geometry Shaders*.

5. Conclusões

A biblioteca simplificou substancialmente o desenvolvimento de aplicações utilizando *OpenGL 4* e *GLSL 4*. A quantidade de código para configuração e inicialização diminuiu consistentemente. Ao abstrair todo o trabalho de configurações extensivas e complicadas, o programador pode focar no código da aplicação em si. Além disso, a biblioteca também serve como um *framework* simples para testar novos *shaders* de forma fácil e rápida.

6. Agradecimentos

Os autores gostariam de agradecer ao Daniel Santos, estudante de mestrado na COPPE-UFRJ, por seus trabalhos desenvolvendo o *Ray Casting* Baseado em GPU e por prover sua imagem utilizada neste paper. Gostariam ainda de agradecer ao CNPQ pela disponibilização da bolsa de Iniciação Científica como incentivo ao desenvolvimento destes trabalhos.

Referências

Andrade, V. and Marroquim, R. (2012). Técnicas de programação em gpu aplicadas à visualização de modelos com múltiplas texturas. In *Workshop of Undergraduate Works (WUW) in SIBGRAPI 2012 (XXV Conference on Graphics, Patterns and Images)*.

Andrade, V. and Marroquim, R. (2013). Shaderlib. http://www.lcg.ufrj.br/shaderlib.

Carlsson, J. (2010). Realistic ambient occlusion in real-time - survey of the aov algorithm and propositions for improvements.

Decaudin, P. Anttweakbar. http://anttweakbar.sourceforge.net/doc/.

Di Benedetto, M., Ponchio, F., Ganovelli, F., and Scopigno, R. (2010). Spidergl: A javascript 3d graphics library for next-generation www. In *Web3D 2010. 15th Conference on 3D Web technology*.

Geelnard, M. and Berglund, C. Glfw. http://www.glfw.org/index.html.

Gomes, T., Estevao, L., Toledo, R., and Roma, P. (2012). A survey of glsl examples. volume 0, pages 60–73.

Guennebaud, G., Jacob, B., et al. Eigen v3. http://eigen.tuxfamily.org.

Luten, E. Opengl book. http://openglbook.com/.

Mittring, M. (2007). Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*.

Shanmugam, P. and Arikan, O. (2007). Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*.

Vianna, F., Elias, T., and Toledo, R. (2012). Shaderlabs: Desenvolvimento ágil de uma ide para opengl shaders.