

Pandora's Box Graphics Engine - Uma Engine Gráfica com Aplicação em Visualização de Campos Tensoriais

Andrew Toshiaki Nakayama Kurauchi

Victor Kendy Harada

Orientador: Prof. Dr. Marcel Parolin Jackowski

1 de dezembro de 2011

Sumário

1	Introdução	3
1.1	Motivação	4
1.2	Objetivos	4
2	Tecnologias estudadas	6
2.1	Linguagem C++98	6
2.1.1	A Standard Template Library (STL)	9
2.2	Boost C++ Libraries	10
2.3	OpenGL	11
2.4	GLEW	13
2.5	Windows API	13
3	Conceitos estudados	14
3.1	Transformações lineares	14
3.2	Tensores de imagens de ressonância magnética de difusão	15
3.3	Representação elipsoidal de tensores de difusão	15
3.4	Anisotropia Fracionada	16
4	Atividades realizadas	18
4.1	Objetos da API gráfica	18
4.2	Grafo de cena	25
4.2.1	Tipos de nós	25
4.2.2	Node Visitors	26
4.3	Renderizador	28
4.3.1	Algoritmos de processamento de cena pré-definidos	29
4.3.2	Algoritmos de pós-processamento de cena pré-definidos	30
4.3.3	Algoritmos de pós-processamento customizados	30
4.4	Mecanismos da Engine	32
4.4.1	Mapeamento e gerenciamento dos estados	32
4.4.2	Desenho de modelos	32
4.4.3	Passagem de parâmetros para o GPUProgram	33
4.5	Visualização de campos tensoriais	34
4.5.1	O formato Analyze	35
4.5.2	O formato Compiled Tensor Field (.ctf)	35
4.5.3	Apresentação do campo compilado	36
4.6	Técnicas aplicadas	36
4.6.1	Depth Peeling	36

5 Resultados e produtos obtidos	38
5.1 Compilador de campos tensoriais	38
5.2 Visualizador de campos tensoriais	38
5.2.1 Movimentação e interação com o campo tensorial	38
5.3 Exemplo de código	41
6 Conclusões	45
A Apêndice	49
A.1 A Windows API	49
A.2 Uso dinâmico de DLLs	53

1 Introdução

O rápido desenvolvimento da tecnologia nas últimas décadas permitiu um avanço significativo na qualidade e velocidade de geração de imagens de computação gráfica. Entretanto ainda existem muitos desafios.

A quantidade de informação a ser processada é grande se comparada a capacidade de processamento. Além disso imagens foto realistas podem levar dias para serem geradas. Por esse motivo é necessária a utilização de diversas técnicas de otimização visando produzir um resultado de melhor aparência de forma eficiente (tanto com relação ao tempo, quanto aos recursos consumidos).

Nesse contexto a visualização científica pode adicionar complexidade ao problema. Tal aplicação de computação gráfica está muitas vezes associada a uma grande quantidade de dados complexos a serem processados e visualizados. Por outro lado existem informações que podem ser irrelevantes para o usuário desse tipo de aplicação, tais como iluminação (e sombra), textura de materiais, ou até mesmo a profundidade. Sendo assim é possível utilizar outros tipos de técnicas que se adequem melhor a essas restrições.

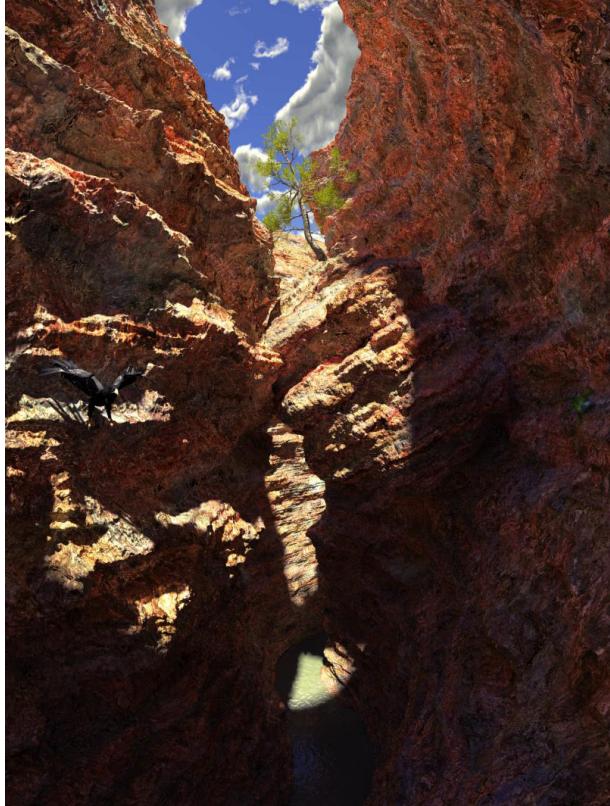


Figura 1: Cena complexa [17] gerada a partir de um conjunto de técnicas avançadas de computação gráfica, dentre as quais o raytracing [28], que consiste em gerar raios de luz partindo da câmera até os objetos.

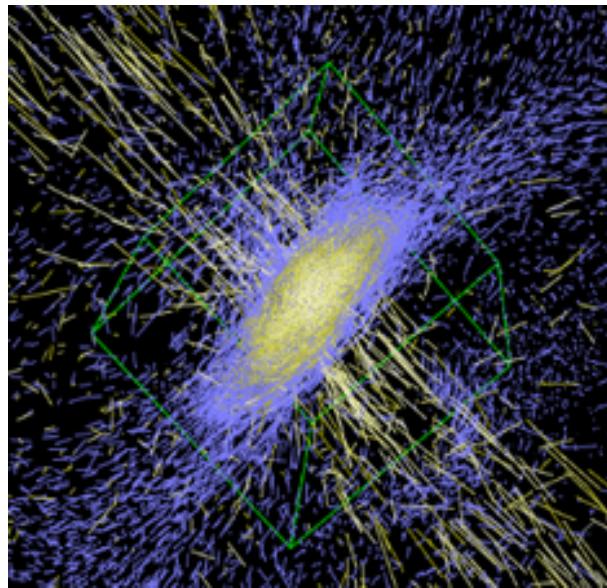


Figura 2: Imagem resultante de uma simulação da formação de uma galáxia de disco [12]. Essa simulação inclui a visualização da formação estelar e ventos galácticos, o que aumenta a complexidade da cena.

1.1 Motivação

As técnicas de otimização aplicadas no desenvolvimento de programas de computação gráfica são recorrentes em diversos aplicativos. Por esse motivo são desenvolvidas engines gráficas como OpenSceneGraph [27] e Ogre3D [21], que possibilitam a utilização de diversas técnicas previamente implementadas, assim como disponibilizam maneiras de descrever cenas complexas através de estruturas de dados conhecidas. Entretanto, tais engines evoluíram de tal forma que o tempo de aprendizagem é longo.

1.2 Objetivos

O intuito desse trabalho é desenvolver a Pandora's Box Graphics Engine, uma engine gráfica que implemente técnicas básicas de computação gráfica e que seja fácil de ser aprendida e utilizada. Além disso deseja-se que ela seja customizável, permitindo a sua utilização em diversos tipos de aplicação, como a geração de imagens foto realistas, o processamento de imagens ou a visualização científica. Como exemplo de visualização científica será implementado um visualizador de campos tensoriais com o auxílio da engine.

O estudo de tensores e campos tensoriais é de importância fundamental para diversas áreas do conhecimento. São encontradas aplicações no estudo de fenômenos sísmicos [9], estruturas eletrônicas [15] e imagens de ressonância magnética sensíveis a difusão [16]. A visualização de tais campos é, portanto, de grande relevância para o avanço do conhecimento. Na aplicação desenvolvida será utilizada para testes a visualização da representação elipsoidal de campos tensoriais provenientes de imagens de ressonância magnética sensíveis a difusão.

2 Tecnologias estudadas

Nessa seção serão apresentadas as tecnologias que foram estudadas para o desenvolvimento da engine gráfica Pandora's Box.

2.1 Linguagem C++98

A linguagem de programação C++ é uma linguagem multiparadigma, que suporta tanto programação estruturada quanto orientada a objetos. Ela foi desenvolvida como uma evolução da linguagem C e, por esse motivo, suas sintaxes são semelhantes.

Variáveis e funções

As variáveis e funções são declaradas com a mesma sintaxe utilizada na linguagem C (um tipo seguido de um nome, podendo receber um valor inicial):

```
<tipo> <nome> [= <valor inicial>]
```

Ou seja, para declarar uma variável inteira chamada `contador` com valor inicial zero é utilizada a seguinte sintaxe:

```
int contador = 0;
```

Classes e objetos

Classes são definidas pelas palavras reservadas `class` ou `struct`, sendo que a diferença entre elas é o fato de atributos de uma `class` serem privados por padrão e os de uma `struct`, públicos. É possível adicionar modificadores de acesso através das palavras `public`, `private` e `protected` que definem acesso livre em todos os escopos, acesso restrito à classe e acesso restrito à classe e suas subclasses respectivamente. Construtores são declarados como funções com o nome da classe sem valor e tipo de retorno. A declaração de uma classe `Coordenada2d` poderia ocorrer das seguintes maneiras:

Utilizando `class`:

```
class Coordenada2d {
public:
    Coordenada2d(double x, double y);
private:
    double x,y;
};
```

Utilizando `struct`:

```
struct Coordenada2d {  
    double x,y;  
};
```

Para se instanciar um objeto existem duas opções. A primeira consiste em declarar uma variável cujo tipo é o nome da classe desejada (da mesma forma como variáveis do tipo `int` são declaradas, por exemplo). Na declaração da variável a classe será instanciada e inicializada. A segunda maneira é a utilização da palavra reservada `new` seguida do nome da classe. O valor devolvido é um ponteiro para uma instância da classe desejada. Segundo o exemplo anterior, a classe `Coordenada2d` poderia ser instanciada das seguintes formas:

```
Coordenada2d coordenada1;  
Coordenada2d * coordenada2 = new Coordenada2d;
```

Os argumentos do construtor são passados da seguinte maneira:

```
Coordenada2d coordenada1(1.0 , 2.0);  
Coordenada2d * coordenada2 = new Coordenada2d(1.0 , 2.0);
```

Namespaces

Para evitar colisão de identificadores como nomes de classes, variáveis e funções é possível utilizar um `namespace`:

```
namespace nome {  
// Entidades  
    int n;  
}
```

Onde `nome` é qualquer identificador válido e `Entidades` é o conjunto de classes, variáveis e funções que ficarão protegidas dentro de um `namespace`.

Para referenciar identificadores de um `namespace` é utilizada a seguinte sintaxe:

```
<nome do namespace>::<identificador>
```

Para acessar a variável `n` do exemplo utiliza-se o seguinte código:

```
nome::n
```

Templates

O template é um recurso da linguagem C++ que permite programação genérica. É

utilizado para a implementação de funções ou classes cujas funcionalidades podem ser adaptadas para tipos diferentes com o mesmo código. Para isso é utilizada a palavra reservada **template** seguida por um parâmetro de template, que é um tipo especial de parâmetro que pode ser utilizado para enviar tipos como argumento.

Uma função **maximo** poderia ser definida da seguinte maneira:

```
template <class T>
T maximo(T a, T b) {
    T resultado;
    resultado = (a > b) ? a : b;
    return resultado;
}
```

O código acima pode ser utilizado da seguinte maneira:

```
int c = 4, d = 5;
maximo(c,d);
```

Um nó de uma lista ligada genérica poderia ser implementado de maneira análoga:

```
template <class T>
class No {
public:
    No(T valorInicial) {
        valor = valorInicial;
        proximo = NULL;
    }
    T getValor() {
        return valor;
    }
    void setValor(T novoValor) {
        valor = novoValor;
    }
    No<T> * getProximo() {
        return proximo;
    }
    void setProximo(No<T> *novoProximo) {
        proximo = novoProximo;
    }
private:
    T valor;
    No<T> *proximo;
};
```

A utilização também é análoga à de funções, porém como o compilador não consegue deduzir o tipo de T, deve-se declará-lo explicitamente:

```
No<int> meuNo(5);
```

2.1.1 A Standard Template Library (STL)

A Standard Template Library (STL) [29] é a biblioteca padrão do C++ que oferece um conjunto grande de algoritmos e estruturas de dados genéricos.

É possível, por exemplo, aplicar uma função para todos os itens de um vetor de inteiros da seguinte maneira:

```
void imprime(int i) {
    printf("%d ", i);
}
int main() {
    std::vector<int> vetor;
    // Adicionando os valores 1, 2 e 3 no vetor
    vetor.push_back(1);
    vetor.push_back(2);
    vetor.push_back(3);

    for_each(vetor.begin(), vetor.end(), imprime);
    // Saída: 1 2 3
    return 0;
}
```

O método `for_each` pode, alternativamente, receber no lugar da função, uma instância de uma classe que implementa o método `operator()`:

```
struct impressor {
    void operator()(int i) {
        printf("%d ", i);
    }
} meuImpressor;

int main() {
    std::vector<int> vetor;
    vetor.push_back(1);
    vetor.push_back(2);
    vetor.push_back(3);

    for_each(vetor.begin(), vetor.end(), meuImpressor);
    // Saída: 1 2 3
    return 0;
}
```

Há diversos outros tipos de algoritmos implementados na STL [2].

2.2 Boost C++ Libraries

Boost [6] é uma biblioteca de C++ de código aberto inicialmente desenvolvida como uma extensão para a STL sob o namespace `boost`. Os principais módulos dessa biblioteca utilizados no desenvolvimento da engine foram `bind`, `function` e `smart_ptr`.

O módulo `function` cria functors (instâncias de classes que implementam o método `operator()`) que são capazes de armazenar tanto funções nativas da linguagem quanto outros functors. Essas funções são criadas através da seguinte construção:

```
void f (int i, int j, int k) {
    // implementação
}

// A boost::function recebe o número n de argumentos, o tipo do valor
// de retorno e os tipos dos n argumentos
// boost::function[n]<tipo de retorno, tipos dos argumentos>
boost::function3<void, int, int, int> funcao = f;
```

A biblioteca `bind` cria functors a partir de uma função ou outro functor mas com alguns argumentos já definidos, por exemplo:

```
boost::bind(f, _1, _2)
```

Nesse caso `boost::bind` cria um functor que recebe apenas dois argumentos, indicados pelos nomes `_1` e `_2`, o primeiro e o segundo argumento do functor criado respectivamente.

As funcionalidades de `bind` e `function` podem ser combinadas:

```
// functor sem retorno que recebe dois argumentos inteiros
boost::function2<void, int, int> functor1 =
    boost::bind(f, _1, _2);

// cria um boost::function a partir do resultado do boost::bind
// de uma boost::function
boost::function1<void, int> functor2 = boost::bind(functor1, _2, _1);
```

Os módulos `bind` e `function` da boost são utilizados principalmente em conjunto com algoritmos da STL, como o `std::for_each`:

```

void imprime(int i) {
    printf(“%d ”, i);
}

int main() {
    std :: vector<int> vetor;
    boost :: function1<void, int> impressor = imprime;

    // Adicionando os valores 1, 2 e 3 no vetor
    vetor.push_back(1);
    vetor.push_back(2);
    vetor.push_back(3);

    for_each(vetor.begin(), vetor.end(), impressor);
    // Saída: 1 2 3

    return 0;
}

```

Os **smart_ptr** são classes que guardam ponteiros para áreas de memória alocadas dinamicamente. Essas classes tem comportamento semelhante ao de ponteiros nativos da linguagem, com a diferença de que elas desalocam o recurso a que referenciam em momento adequado. Existem seis tipos **smart_ptr** definidos:

- **scoped_ptr**: representa o conceito de ponteiro não compartilhável, logo não pode ser copiado. Desaloca a memória ao sair de escopo.
- **scoped_array**: mesmo que **scoped_ptr** mas para um array de objetos.
- **shared_ptr**: representa um ponteiro com diversos donos, pode ser copiado e só é destruído quando todos os donos forem destruídos.
- **shared_array**: mesmo que **shared_ptr** mas para arrays.
- **weak_ptr**: são ponteiros para recursos de um **shared_ptr** mas que não são donos do recurso.
- **intrusive_ptr**: são **shared_ptr** que guardam dentro do objeto para o qual apontam o contador de referências utilizado internamente.

2.3 OpenGL

O OpenGL é uma especificação aberta, extensível [26], amplamente adotada no mercado e independente do sistema operacional de uma interface de software para o hardware gráfico.

Para se manter minimalista, o OpenGL não especifica nada que dependa do sistema operacional, como gerenciamento de janelas ou controle de entrada e saída de dados, assim como também não suporta nenhum tipo de formato de arquivo como imagens (png, jpg, bmp) e modelos (3ds [1], blend [5], ou obj [20]), ele apenas oferece formas de se combinar diferentes vértices para se formar primitivas e de fazer com que essas primitivas sejam transformadas em pixels.

As primitivas criadas na CPU (Central Processing Unit) são enviadas para a GPU (Graphics Processing Unit) onde serão processadas e utilizadas para gerar os pixels de uma imagem que será mostrada na janela. Esse processo é conhecido como pipeline e é dividido em 5 estágios principais compostos por shaders (programas executados pela GPU):

- **Vertex shader:** cada vértice é transformado para a sua coordenada final.
- **Tesselation shaders:** as primitivas formadas após a execução do vertex shader podem ser opcionalmente quebradas em primitivas menores.
- **Geometry shader:** a partir das primitivas geradas nos passos anteriores, o geometry shader decide se novas primitivas devem ser geradas.
- **Rasterização:** nessa etapa as primitivas de 3 dimensões são projetadas em um plano, que representa a janela do usuário. Essa projeção gera fragmentos, que são candidatos a pixels.
- **Fragment shader:** cada fragmento gerado na rasterização é processado pelo fragment shader e, após a execução, passa por um teste de profundidade, que determina se o fragmento é visível. Caso passe no teste, ele é escrito no framebuffer e se torna um pixel visível, senão, é descartado.

No OpenGL todos esses processos (com exceção da rasterização) são customizáveis, entretanto na Pandora's Box somente o vertex shader e o fragment shader podem ser modificados. Mais detalhes são apresentados na seção 4.

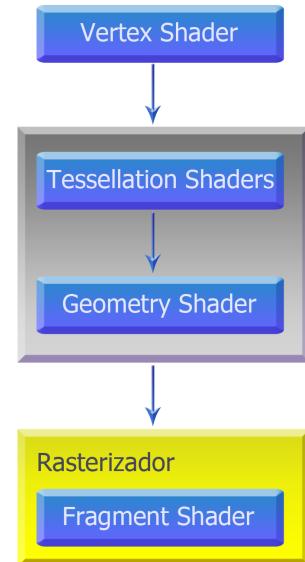


Figura 3: Na GPU as primitivas do OpenGL passam pelos 5 estágios do pipeline representados acima.

O contexto do OpenGL

O OpenGL é implementado internamente como uma máquina de estados. Tais estados, assim como outras informações, como a janela e texturas são guardados em um contexto, que deve ser criado e inicializado para o funcionamento da aplicação. Essa criação é dependente da plataforma e por esse motivo não está definida na API do OpenGL.

2.4 GLEW

OpenGL Extension Wrangler Library [13], conhecida como GLEW é uma biblioteca que auxilia no gerenciamento de extensões do OpenGL. Ela define constantes em tempo de execução contendo as extensões suportadas pelo sistema do usuário e, além disso, obtém os ponteiros das novas funções definidas nas extensões.

O uso da GLEW é ilustrado no código abaixo:

```
#include <GL/glew.h>
// Após a inicialização do contexto do OpenGL
glewInit();
// após a inicialização as funções definidas através de extensões
// estão disponíveis

// verificar se a extensão GL_ARB_vertex_program está disponível
if(GLEW_ARB_vertex_program) {
    ...
}
// ou
if(glewIsSupported("GL_ARB_vertex_program")) {

}
```

2.5 Windows API

Como citado anteriormente, o OpenGL não é responsável por criar janelas e tratar eventos gerados pelo sistema operacional, esse tratamento é dependente do ambiente no qual o programa será executado. Nesse trabalho foi utilizada apenas a API do sistema operacional Windows que está contida no `windows.h`.

Um texto detalhado sobre a criação de janelas pode ser encontrado no apêndice A.1 e a documentação da Windows API encontra-se no site da MSDN [19]

Carregamento dinâmico de DLL

Para que seja possível suportar diversas implementações independentes de especificações de APIs gráficas, a engine possui um mecanismo de carregamento dinâmico de DLLs (os detalhes sobre o carregamento de DLLs podem ser encontrados no apêndice A.2).

3 Conceitos estudados

A seguir serão apresentados os conceitos que foram estudados para a implementação da engine, assim como na aplicação de exemplo (visualização de campos tensoriais)

3.1 Transformações lineares

O conceito de transformações lineares é de importância fundamental para a computação gráfica, pois podem ser aplicadas sobre geometrias sem alterar a sua forma (realizando movimentos de translação ou rotação por exemplo). Podem ser representadas como matrizes bastando multiplicá-las para realizar composições.

Em computação gráfica, é comum a utilização de coordenadas homogêneas para a representação de vértices [22] e de matrizes de transformação 4×4 . As transformações mais aplicadas sobre modelos geométricos e suas respectivas matrizes são:

- Translação:

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Onde t_x, t_y, t_z são as translações nos eixos x, y, z respectivamente.

- Rotação de θ em torno do eixo x:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Rotação de θ em torno do eixo y:

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Rotação de θ em torno do eixo z:

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Escala:

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Onde s_x, s_y, s_z são as escalas aplicadas nos eixos x, y, z respectivamente.

3.2 Tensores de imagens de ressonância magnética de difusão

Difusão é o nome dado ao movimento aleatório de moléculas em um fluido (líquido ou gasoso) e denomina-se coeficiente de difusão a facilidade com que uma molécula se desloca em determinado meio. Em fluidos homogêneos, como a água, o coeficiente de difusão é o mesmo em todas as direções. A esse tipo de meio dá-se o nome de isotrópico. Em fluidos heterogêneos o coeficiente pode variar dependendo da direção. Esses são os meios anisotrópicos. Um exemplo de meio anisotrópico são os tecidos biológicos.

Imagens de ressonância magnética de difusão são uma das formas de se obter informações sobre o coeficiente de difusão de moléculas de água em diferentes direções em tecido biológico.

Para representar tais coeficientes são utilizados tensores, que são abstrações de escalares, vetores e matrizes com diversas aplicações na ciência e matemática. Em imagens de ressonância magnética de difusão, tais tensores são comumente representados por matrizes 3×3 simétricas.

3.3 Representação elipsoidal de tensores de difusão

Em meios isotrópicos o coeficiente de difusão pode ser representado por uma esfera, pois o movimento das moléculas se distribui igualmente para todas as direções em um determinado período de tempo. Já em meios anisotrópicos, como o movimento depende da direção, o coeficiente é modelado como um elipsóide, sendo que os semi-eixos têm comprimento proporcional aos autovalores do tensor ($\lambda_1 > \lambda_2 > \lambda_3$) ao longo dos seus autovetores $\epsilon_1, \epsilon_2, \epsilon_3$.

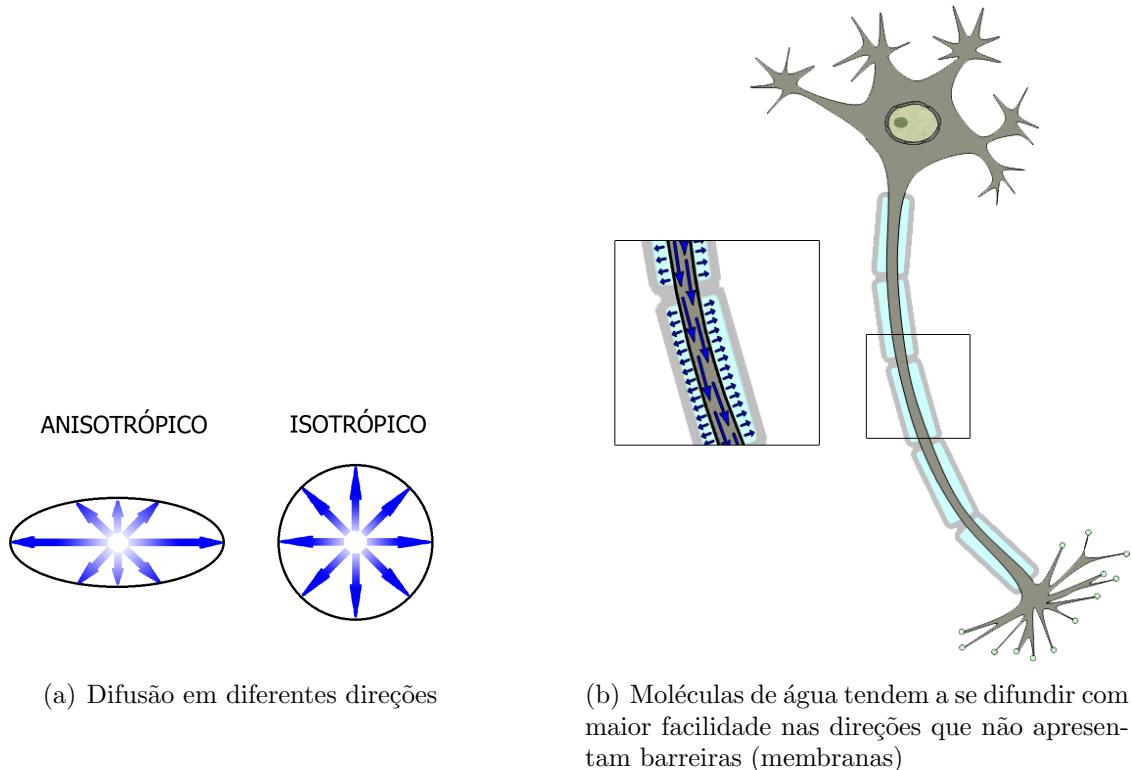


Figura 4: O conceito de isotropia e anisotropia aplicado a tecidos biológicos.

3.4 Anisotropia Fracionada

A anisotropia fracionada é um valor no intervalo $[0, 1]$ que descreve o grau de anisotropia de um processo de difusão. Sejam $\lambda_1, \lambda_2, \lambda_3$ os autovalores do tensor. A anisotropia fracionada (AF) pode ser calculada a partir da seguinte fórmula [14]:

$$AF = \sqrt{\frac{1}{2} \frac{\sqrt{(\lambda_1 - \lambda_2)^2 + (\lambda_1 - \lambda_3)^2 + (\lambda_2 - \lambda_3)^2}}{\sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}}} \quad (1)$$

A anisotropia fracionada pode também ser dividida em três medidas (linear, planar e esférica) que indicam a forma da difusão [30]. Sejam $\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq 0$:

Caso linear:

$$C_l = \frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2 + \lambda_3} \quad (2)$$

Caso planar:

$$C_p = \frac{2(\lambda_2 - \lambda_3)}{\lambda_1 + \lambda_2 + \lambda_3} \quad (3)$$

Caso esférico:

$$C_s = \frac{3\lambda_3}{\lambda_1 + \lambda_2 + \lambda_3} \quad (4)$$

Tais medidas também pertencem ao intervalo $[0, 1]$. Quanto mais próximo de 1, em cada caso, mais o tensor se assemelha à forma referente.

4 Atividades realizadas

As ideias apresentadas a seguir foram aplicadas no desenvolvimento da engine e da aplicação de visualização de campos tensoriais.

4.1 Objetos da API gráfica

Nessa seção serão descritos os mapeamentos utilizados pela engine para os objetos mais comuns definidos pelo OpenGL. Todos os mapeamentos foram projetados de forma que os objetos pudessem ser utilizados fora de um contexto inicializado e, além disso, o objeto do OpenGL só é instanciado se necessário, o que economiza recursos da placa gráfica.

Como a implementação dos objetos dessa seção é altamente dependente da API gráfica, eles devem ser construídos por um componente chamado `GraphicObjectsFactory`. Esse componente pode ser obtido a partir de uma instância de `GraphicAPI` da seguinte forma:

```
GraphicAPI * gfx = ...;
GraphicObjectsFactory * factory = gfx->getFactory();
```

Buffer

Um buffer é uma região de memória controlado pelo driver gráfico. O conceito de Buffer foi criado para resolver o gargalo da comunicação entre Memória Principal e Memória de Vídeo.

No OpenGL, buffers são criados através da função `glGenBuffers`. Depois de criado, o buffer deve ser associado ao contexto do OpenGL, para isso utiliza-se o comando `glBindBuffer`, esse comando, além de informar ao OpenGL que o buffer deve ser associado ao contexto, faz com que os dados presentes no buffer sejam interpretados de forma diferente. Por exemplo, para informar que o buffer será utilizado para fornecer vértices ao pipeline, utiliza-se:

```
glBindBuffer(GL_ARRAY_BUFFER);
```

Com o buffer associado ao contexto, é necessário inicializar seus dados. Essa inicialização é feita pelo comando `glBufferData`. Após esses procedimentos, o buffer está pronto para ser utilizado.

Uma das características mais importantes do buffer é a possibilidade de mapeá-lo em uma região de memória que pode ser acessada pela aplicação. Esse mapeamento é importante, pois permite que a aplicação informe ao OpenGL que política de acesso a aplicação adotará para ler ou escrever na memória mapeada. Por exemplo, se o buffer for mapeado utilizando-se `GL_WRITE_ONLY`, o driver gráfico não precisa copiar os dados

da memória do driver para a memória principal, o que diminui a troca de dados entre a memória de vídeo e a memória principal.

As políticas de acesso definidas pelo OpenGL são:

- **GL_READ_ONLY**: quando a aplicação planeja apenas ler a região de memória.
- **GL_WRITE_ONLY**: quando a região de memória for utilizada apenas para escrita.
- **GL_READ_WRITE**: quando a memória mapeada for utilizada para leitura e escrita.

O mapeamento de um vertex buffer que será utilizado somente para escrita pode ser feito através do seguinte código:

```
glBindBuffer(GL_ARRAY_BUFFER, bufferID );
void * mappedBuffer = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
```

Após a utilização da região mapeada, é importante cancelar o mapeamento feito. Esse cancelamento é realizado através da função `glUnmapBuffer`:

Na Pandora's Box, buffers são implementações da interface `Buffer`. O buffer padrão do OpenGL é implementado pela classe `GLBuffer`. A seguir é apresentado um exemplo de uso:

```
GraphicAPI *gfx = ...;

// cria um buffer de tamanho size e com política de uso 'usage'
// As políticas de uso permitidas são:
// STREAM_DRAW, STREAM_READ, STREAM_COPY, STATIC_DRAW, STATIC_READ,
// STATIC_COPY, DYNAMIC_DRAW, DYNAMIC_READ e DYNAMIC_COPY
Buffer *buffer = gfx->getFactory()->createBuffer(size, usage);
void * mapped = buffer->map(pbge::Buffer::WRITE_ONLY);

// operar na região mapeada
buffer->unmap();
```

Shaders

Como explicado anteriormente, o OpenGL opera nas primitivas gráficas através de um pipeline, que tem cinco estágios principais: transformação de vértices, tessellagem, criação de novas primitivas, rasterização e operações por fragmento. Com exceção da rasterização, todos os estágios podem ser customizados pelo uso de shaders.

No OpenGL, shaders são criados através da função `glCreateShader`, que devolve um identificador para o shader criado, com o qual é possível especificar o código fonte que será utilizado por meio da função `glShaderSource`. Após associar um código fonte, ele

pode ser compilado por `glCompileShader`. Para utilizar o shader criado, ele deve ser associado a um programa.

Programas, no OpenGL, são criados por `glCreateProgram`, que devolve o identificador para o programa criado. Depois de criados, os shaders podem ser associados ao programa, a associação é feita pela função `glAttachShader`. Quando todos os shaders necessários estiverem associados ao programa, ele pode ser preparado para execução através do comando `glLinkProgram`. Para sobreescrver o pipeline com o programa criado utiliza-se a função `glUseProgram`.

Na Pandora's Box, shaders são implementações da classe `Shader`. Para que possam ser utilizados para sobreescrver o pipeline, os shaders devem ser associados a um `GPUProgram`. A criação de um `GPUProgram` que sobreescrve tanto a transformação de vértices quanto o processamento de pixels é exemplificada abaixo:

```
GraphicAPI * gfx = ...;

Shader * vertexShader = gfx->createShaderFromString(
    vertexShaderSource,
    pbge::Shader::VERTEX_SHADER);
Shader * fragmentShader = gfx->createShaderFromString(
    fragmentShaderSource,
    pbge::Shader::FRAGMENT_SHADER);

std::vector<Shader*> vertexShaders;
std::vector<Shader*> fragmentShaders;
vertexShaders.push_back(vertexShader);
fragmentShaders.push_back(fragmentShader);

GPUProgram * program = gfx->getFactory()->createProgram(
    vertexShaders,
    fragmentShaders);
```

O código acima pode ainda ser simplificado para:

```
GraphicAPI * gfx = ...;
GPUProgram * program = createProgramFromString(
    vertexShaderSource, fragmentShaderSource);
```

O `GPUProgram` criado é utilizado para sobreescrver o pipeline através do código:

```
GraphicAPI * gfx = ...;
GPUProgram * program = ...;
gfx->getState()->useProgram(program);
```

Texturas

Texturas no OpenGL são simplesmente tabelas de valores. Em aplicações mais antigas eram utilizadas quase exclusivamente para o armazenamento de imagens que seriam mapeadas nos objetos da cena ou máscaras (em técnicas avançadas), porém com o surgimento da computação genérica em GPU (GPGPU), passaram a ser utilizadas de forma análoga aos vetores e matrizes de linguagens como C++.

No OpenGL as texturas são criadas com o comando `glGenTextures`, que gera um identificador para o objeto vazio criado, em seguida é necessário associar o objeto criado ao contexto do OpenGL, o que é feito pelo comando `glBindTexture`. Com a textura vazia associada ao contexto, é possível inicializar seus dados através de um comando da família `glTexImage`.

Na Pandora's Box, texturas são implementações da interface `Texture`. A inicialização de uma textura 2D é demonstrada no código abaixo:

```
GraphicAPI * gfx = ...;

// image é uma interface que representa uma fonte de dados
// para uma textura 2D.
Image * image = ...;
Texture2D * texture = gfx->getFactory()->create2DTexture();

// especifica a imagem e como os dados devem ser representados
// na GPU
texture->setImageData(image, pbge::Texture::RGBA);
```

Com o aumento da quantidade de dados enviados à GPU através de texturas, surge um problema: as texturas padrão das APIs gráficas conseguem indexar apenas um pequeno número de pixels na textura, cerca de 2048 em placas recentes. Para resolver esse problema, o OpenGL introduziu o conceito de texture buffer, uma textura de uma dimensão que utiliza um buffer para armazenar dados, essa nova textura consegue indexar no mínimo 65536 pixels, segundo a especificação do OpenGL 4.1 [25].

A criação de um texture buffer é feita de forma ligeiramente diferente de uma textura convencional. Após a criação do identificador, a textura deve ser associada ao contexto através da chamada a `glBindTexture(GL_TEXTURE_BUFFER, identificador)`. O buffer a ser utilizado pela textura é então definido pelo comando `glTexBuffer`.

Na engine, os texture buffers são texturas que implementam a interface `TextureBuffer`. Um exemplo de criação de texture buffer é dado no código abaixo:

```

GraphicAPI *gfx = ...

TextureBuffer * texture =
    gfx->getFactory()->createTextureBuffer( size );

// a textura será representada como um conjunto de 4 floats
// por texel (elemento da textura)
texture->setInternalFormat( pbge::Texture::FLOAT,
                             pbge::Texture::RGBA);

// recupera o buffer associado à textura
Buffer *buffer = texture->getBuffer();
float * data = (float*) buffer->map( pbge::Buffer::WRITE_ONLY);

// inicialização dos dados do buffer
buffer->unmap();

```

Após a criação das texturas, elas são enviadas ao shader através dos `UniformSet`, como demonstrado abaixo no caso de uma textura 2D:

```

GraphicAPI *gfx = ...;
Texture2D * texture = ...;
UniformSet uniforms;

// associa a textura texture à variável do shader do tipo sampler2D
// chamada shader_texture
uniforms->getSampler2D("shader_texture")->setValue( texture );
gfx->pushUniforms(&uniforms);

```

VertexBuffer

No início da computação gráfica, os vértices eram especificados um a um através de chamadas à API gráfica. Por exemplo, a especificação de um triângulo no OpenGL era feita da seguinte maneira:

```

glBegin(GL_TRIANGLES);
glVertex3f(0.0f, 0.0f, 0.0f);
glNormal3f(0.0f, 0.0f, 1.0f);
glVertex3f(0.0f, 1.0f, 0.0f);
glNormal3f(0.0f, 0.0f, 1.0f);
glVertex3f(1.0f, 0.0f, 0.0f);
glNormal3f(0.0f, 0.0f, 1.0f);
glEnd();

```

É possível observar que o número de chamadas de função cresce linearmente com a quantidade de vértices do modelo, além disso cada um dos atributos do vértice (posição,

cor, normal, entre outros) era especificado através de uma chamada de função diferente.

Com o aumento da complexidade dos modelos, foi necessário reinventar a API existente, com isso foi criado o conceito de vertex array, que é simplesmente um vetor de valores de tipos variados em que cada atributo é especificado através de seu tipo primitivo (double, float, int, short, byte) número de componentes, significado (posição, vetor normal, coordenada de textura, etc) e a distância em bytes para o próximo valor desse atributo no vetor. Essa estrutura é exemplificada na figura ao lado.

Como a especificação desse modelo de dados era flexível e permitia enviar uma grande quantidade de dados para a GPU em um número constante de chamadas de função, ele foi rapidamente adotado pelos programadores.

Entretanto esse modelo não solucionava o problema de tráfego de dados entre a memória principal (RAM) e a memória de vídeo (VRAM). A solução foi colocar o vertex array dentro de um buffer gerenciado pela implementação da API gráfica, assim, a implementação poderia colocar dados muito utilizados dentro de regiões de fácil acesso, como a VRAM. Essa solução ficou conhecida como vertex buffer.

Na Pandora's Box, vertex buffers são especificados através da classe `VertexBuffer`, sua estrutura interna é semelhante à apresentada na Figura 5. Cada vértice é especificado através da classe `VertexAttrib` que guarda informações sobre o primeiro índice do atributo dentro o buffer, seu significado, número de componentes e qual a distância entre valores consecutivos do atributo.

A forma recomendada de se criar um `VertexBuffer` é através do `VertexBufferBuilder`.

```

pbge::VertexBuffer * criaVertexBuffer(pbge::GraphicAPI * gfx) {
    // Inicializa com o número de vértices desejado
    int nVertices = ...;

    pbge::VertexBufferBuilder builder(nVertices);
    pbge::VertexAttribBuilder vertex =
        builder.addAttrib(4, VertexAttrib::VERTEX);
    pbge::VertexAttribBuilder color =
        builder.addAttrib(4, VertexAttrib::COLOR);

    for(int i = 0; i < nVertices; i++) {
        float x, y, z, w; // Inicializados com os valores desejados
        float r, g, b, a; // Inicializados com os valores desejados

        builder.on(vertex).pushValue(x, y, z, w);
        builder.on(color).pushValue(r, g, b, a);
    }
    return builder.done(Buffer::STATIC_DRAW, gfx);
}

```

FrameBufferObject

O framebuffer é o destino dos pixels gerados através do pipeline. Existem dois tipos de framebuffer:

- Framebuffer do sistema de janelas: deve ser utilizado quando deseja-se que os pixels sejam enviados para a janela da aplicação.
- Framebuffer virtual: utilizado para renderização direta em texturas.

Os framebuffer objects encapsulam o segundo tipo. No OpenGL, framebuffer objects são criados pela função `glGenFramebuffersEXT`, que gera um identificador que representa o objeto criado. Para associar uma textura ao framebuffer object criado ou para adicionar um buffer de profundidade, para permitir testes de profundidade, utiliza-se a função `glFramebufferTexture2DEXT`.

Para utilizar o framebuffer object criado, é necessário associá-lo ao contexto do OpenGL com o comando `glBindFramebufferEXT` e em seguida redirecionar os pixels gerados pelo pipeline através do comando `glDrawBuffers`.

Na engine desenvolvida, framebuffer objects são implementações da classe abstrata `FrameBufferObject`. A criação e uso de framebuffer objects dentro da Pandora's Box é exemplificado abaixo:

```

GraphicAPI * gfx = ...;
Texture2D * color = ...;
Texture2D * depth = ...;
FramebufferObject * fbo =
    gfx->getFactory()->createFramebuffer(width, height);

// associa os valores escritos pelo shader na variável color_out à
// textura color
fbo->addRenderable(color, "color_out");

// usa depth como buffer de profundidade
fbo->setDepthRenderable(depth);

// associa o framebuffer object ao contexto
gfx->bindFramebufferObject(fbo);

```

4.2 Grafo de cena

A Pandora's Box utiliza um grafo enraizado, direcionado, sem circuitos, conhecido como grafo de cena, para representar a estrutura de uma cena. As mudanças realizadas por cada nó são aplicadas somente a si mesmo e aos seus filhos.

Para realizar a renderização da cena são realizadas buscas em profundidade no grafo até que todas as informações necessárias tenham sido obtidas. Esse processo será melhor explicado em 4.2.2.

4.2.1 Tipos de nós

A engine define quatro tipos de nós padrão: `TransformationNode`, `ModelInstance`, `ModelCollection` e `CameraNode` (transformação, modelo, coleção de modelos e câmera, respectivamente).

Para implementar nós com comportamentos customizados basta criar uma nova classe filha de `Node` ou de algum de seus descendentes. A classe `Node` define essencialmente os métodos de atualização (`updatePass` e `postUpdatePass`, utilizados para preparar a cena para a renderização realizando inicialização de variáveis ou atualização de valores, por exemplo) e renderização (`renderPass` e `postRenderPass`) que devem ser implementados por seus filhos, além de outros métodos específicos da estrutura do grafo.

- **TransformationNode:** Esse nó guarda uma matriz de transformação T . No `updatePass` e `renderPass` a matriz corrente M é armazenada e multiplicada por T e o resultado é utilizado como a nova matriz de transformação corrente. Então no `postUpdatePass` e `postRenderPass` M é reatribuída à matriz corrente.

- **CameraNode:** O nó possui uma instância da classe `Camera` que é responsável por receber os parâmetros de câmera (configurações de posição e campo de visão), calcular a matriz de transformação a partir dessas informações e atualizar o estado do OpenGL para utilizá-la. Todas essas ações são realizadas no `updatePass`.
- **ModelInstance:** O método `renderPass` é responsável por adicionar os shaders e suas uniformes e então solicitar a renderização do modelo pela API gráfica. O método `postRenderPass` retira as uniformes adicionadas.
- **ModelCollection:** A implementação desse nó é análoga à do `ModelInstance`, com a diferença de que a quantidade de instâncias a serem renderizadas é recebida no construtor e enviada na solicitação da renderização do modelo pela API gráfica.

4.2.2 Node Visitors

Como já explicado, o grafo de cena é um grafo enraizado, direcionado e sem circuitos. Um visitor é uma classe que, dada a raíz do grafo de cena, consegue percorrer os nós do grafo obedecendo as seguintes regras:

- Todos os caminhos do grafo devem ser percorridos, se possível.
- O estado do visitor ao visitar um dado nó A, deve depender apenas das modificações feitas por nós dentro do caminho da raíz do grafo até A.

O segundo item da lista acima, faz com que o grafo de cena represente uma estrutura hierárquica.

Na engine, existem duas classes que implementam as regras descritas acima: `UpdaterVisitor` e `ColorPassVisitor`.

O `UpdaterVisitor` é um visitor encarregado de passar por cada nó do grafo de cena e chamar o método `updatePass`, visitar todos os nós filhos do nó atual, chamar o método `postUpdatePass` e por fim atualizar o bounding box do nó atual, como exemplificado no código abaixo:

```

void UpdaterVisitor :: dfsVisit(Node * node, GraphicAPI * gfx) {
    node->updatePass(this, gfx);

    for (...) // Node * child in node->getChildren()
        dfsVisit(child, gfx);

    node->postUpdatePass(this, gfx);
    if(node->getBoundingVolume() != NULL) {
        node->getBoundingVolume()->update(
            getCurrentTransformation());
    }
}

```

O `ColorPassVisitor` é uma implementação concreta da classe abstrata `RenderVisitor` que tem a função de chamar os métodos `renderPass` e `postRenderPass` do nó durante a execução dos métodos `visitAction` e `postVisitAction`, respectivamente, do `RenderVisitor`:

```

class ColorPassVisitor : public RenderVisitor {
public:
    void visitAction(Node * node, GraphicAPI * gfx) {
        node->renderPass(this, gfx);
    }
    void postVisitAction(Node * node, GraphicAPI * gfx) {
        node->postRenderPass(this, gfx);
    }
};

```

O `RenderVisitor` é a classe base de todos os `visitors` que fazem renderização. Para extendê-la, a classe filha deve implementar dois métodos: `visitAction` e `postVisitAction`. Esse visitor é importante, pois implementa uma técnica chamada frustum culling.

Frustum culling é o processo de renderizar apenas objetos que são visíveis para a câmera atual. Na Pandora's Box, um nó é considerado não visível se ele não colide com o frustum da câmera corrente e ao ser considerado não visível, ele e seus nós filhos não são visitados pelo `RenderVisitor`. Esse teste de visibilidade é executado dentro do método `dfsVisit` do `RenderVisitor`:

```

void RenderVisitor :: dfsVisit (Node * node , GraphicAPI * gfx) {
    AABB * boundingVolume = node->getBoundingVolume ();

    if (boundingVolume == NULL ||  

        boundingVolume->frustumCullingTest (boundingFrustum)) {  

        visitAction (node , gfx);  

        for (...) // Node * child in node->getChildren ()  

            dfsVisit (child , gfx);  

        postVisitAction (node , gfx);
    }
}

```

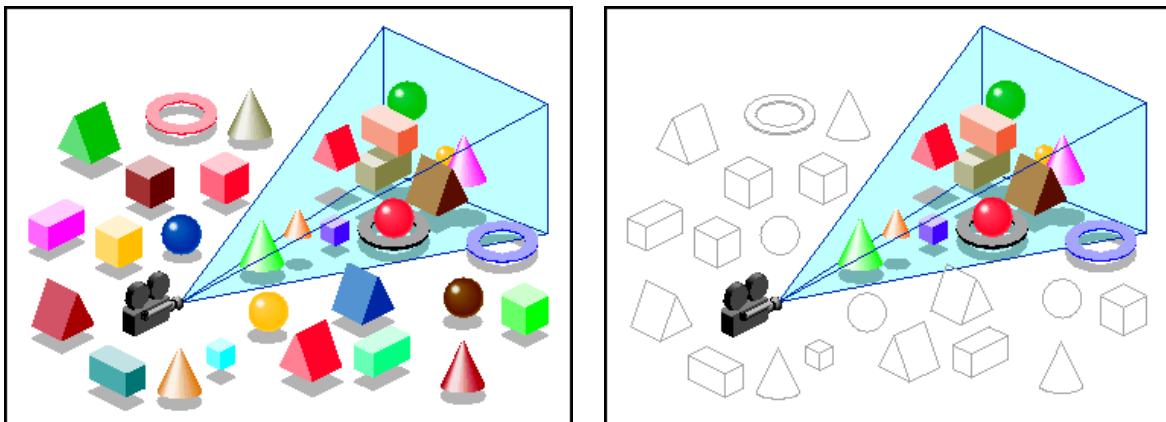


Figura 6: Com o frustum culling os objetos que estão fora do campo de visão da câmera não são enviados para o pipeline.

4.3 Renderizador

O renderizador da engine gráfica utiliza um algoritmo de 3 fases:

- Atualização dos nós do grafo de cena.
- Processamento do grafo de cena.
- Pós-Processamento da imagem gerada pela fase de processamento.

Fase de atualização

Nessa fase, um `UpdaterVisitor` é utilizado para visitar e atualizar todos os nós do grafo de cena. É a única fase não customizável do algoritmo do renderizador.

Fase de processamento do grafo de cena

Nessa fase executa-se uma sequência de algoritmos definida pelo usuário da engine, a execução é equivalente ao código:

```
std :: vector<SceneProcessor*>::iterator it;
for(it = processors.begin(); it != processors.end(); it++){
    if(it->isActive()){
        it->process(gfx, renderer, currentCamera);
    }
}
```

Cada algoritmo deve implementar a interface `SceneProcessor`, que tem os seguintes métodos:

- `bool isInitialized(GraphicAPI*)`: método que indica se o algoritmo já preparou todas as suas dependências.
- `void initialize(GraphicAPI*, Renderer*)`: esse método deve criar todas as dependências do método `process`.
- `void process(GraphicAPI*, Renderer*)`: executa o algoritmo de processamento.
- `bool isActive()`: indica se o algoritmo deve ou não ser executado.

Fase de pós-processamento

Essa fase é semelhante à anterior, porém o teste de profundidade do fragmento não é executado por padrão, a implementação do algoritmo deve ativá-lo manualmente.

A interface que deve ser implementada é a `ScenePostProcessor` que define métodos com a mesma semântica dos métodos do `SceneProcessor`.

4.3.1 Algoritmos de processamento de cena pré-definidos

O `RenderPassProcessor` é, atualmente, o único algoritmo de processamento de cena padrão da engine. Esse algoritmo utiliza `ColorPassVisitor` para renderizar os nós do grafo de cena.

4.3.2 Algoritmos de pós-processamento de cena pré-definidos

Atualmente existem 2 algoritmos de pós-processamento padrão implementados pelas classes: `FramebufferImageProcessor` e `BlitToFramebuffer`.

O `FramebufferImageProcessor` é utilizado para fazer o pós-processamento da imagem que estiver no buffer chamado "color" dentro do `FramebufferObject` atual. O algoritmo executado por esse objeto é descrito pelo código abaixo:

```
void FramebufferImageProcessor :: process (GraphicAPI *gfx ,
                                            Renderer *renderer) {
    std :: map<std :: string , Texture2D*> & renderables ;
    renderables = renderer->getRenderables () ;

    Texture2D * auxBuffer = renderables [ "color_aux" ] ;
    Texture2D * colorBuffer = renderables [ "color" ] ;

    renderables [ "color" ] = auxBuffer ;
    renderables [ "color_aux" ] = colorBuffer ;

    FramebufferObject * fbo = renderer->getFramebufferObject () ;
    fbo->removeRenderable ( "color" ) ;
    fbo->addRenderable ( auxBuffer , "color" ) ;
    fbo->update ( gfx ) ;

    UniformSampler2D* sampler =
        dynamic_cast<UniformSampler2D*>( gfx->getUniformValue (
            UniformInfo ( "color" , pbge :: SAMPLER_2D ) ) ) ;

    sampler->setValue ( colorBuffer ) ;
    renderer->renderScreenQuad ( program . get () ) ;
}
```

O `BlitToFramebuffer` renderiza um retângulo com as dimensões da janela com a textura de nome "color" armazenada no renderizador. É utilizado para renderizar a imagem armazenada na textura "color" para o framebuffer do sistema de janelas.

4.3.3 Algoritmos de pós-processamento customizados

Foram desenvolvidos alguns algoritmos de pós-processamento customizados como exemplo. Eles estão disponíveis na aplicação de visualização de campos tensoriais. A implementação de cada algoritmo se resume a um fragment shader utilizado para instanciar um `FramebufferImageProcessor`. Esse fragment shader recebe a posição do fragmento e uma textura contendo a imagem a ser renderizada.

- **Inversor de cores:** O algoritmo de inversão de cores cria um vetor com os componentes r, g, b , calcula seu complemento e utiliza o resultado como a cor do fragmento:

```
pbge :: FramebufferImageProcessor * colorInversor() {
    return new pbge :: FramebufferImageProcessor(
        "uniform sampler2D color;\n"
        "varying vec2 position;\n"
        "void main() {\n"
        "    vec3 color = (texture2D(color, position.xy)).rgb;\n"
        "    color = 1 - color;\n"
        "    gl_FragColor = vec4(color, 1);\n"
        "}"
    );
}
```

- **Filtro de vermelho:** O filtro lê o componente vermelho da cor enviada pelo vertex shader e a utiliza no fragmento, sendo todos os outros componentes iguais a zero:

```
pbge :: FramebufferImageProcessor * chooseRed() {
    return new pbge :: FramebufferImageProcessor(
        "uniform sampler2D color;\n"
        "varying vec2 position;\n"
        "void main() {\n"
        "    float r = (texture2D(color, position.xy)).r;\n"
        "    gl_FragColor = vec4(r, 0, 0, 1);\n"
        "}"
    );
}
```

- **Lente senoidal:** A posição (x_0, y_0) recebida é tal que $0 \leq x_0, y_0 \leq 1$. Ela é então mapeada para (x_1, y_1) , onde $-1 \leq x_0, y_0 \leq 1$. É calculado então o seno das componentes x, y da posição multiplicadas por um fator que aumenta o efeito da lente. O resultado pertence ao intervalo $[-1, 1]$, entretanto as coordenadas de textura estão no intervalo $[0, 1]$. Por esse motivo o seno é multiplicado por 0.5 e somado a 0.5 resultando em um valor no mesmo intervalo das coordenadas de textura. Esse valor é utilizado para ler uma posição com um pequeno deslocamento da posição do fragmento atual, com isso é gerada uma leve deformação que simula o efeito de uma lente:

```

pbge::FramebufferImageProcessor * senoidalLens() {
    return new pbge::FramebufferImageProcessor(
        "varying vec2 position;\n"
        "uniform sampler2D color;\n"
        "void main()\n"
        "    vec2 x = 2 * position - 1.0;\n"
        "    gl_FragColor = "
        "        texture2D(color, 0.5 + 0.5 * sin(1.5 * x));\n"
    );
}

```

4.4 Mecanismos da Engine

4.4.1 Mapeamento e gerenciamento dos estados

Em placas de vídeo modernas, dispositivos altamente paralelos, as trocas de estado podem fazer com que o pipeline gráfico tenha que ser esvaziado, o que pode causar um grande impacto no desempenho da aplicação. Por isso, é necessário evitar trocas de estado redundantes, além de minimizá-las e agrupá-las sempre que possível.

Na Pandora's Box, esse trabalho é delegado à classe `StateSet` e à duas classes que controlam estados que podem ser desabilitados, `BlendController` e `DepthController`.

O `StateSet` é uma classe que controla a associação de objetos ao contexto gráfico. As modificações feitas através da `StateSet` são sempre acumuladas e só ocorrem quando o método `updateState` é chamado. O `updateState` primeiro atualiza as associações dos objetos gráficos ao contexto e em seguida atualiza os parâmetros do shader.

O `BlendController` controla a combinação de fragmentos no framebuffer. Após ser gerado, se o fragmento não for descartado pelo teste de profundidade, ele é enviado ao framebuffer e substitui o pixel correspondente, porém utilizando o `BlendController`, pode-se fazer com que o fragmento gerado se combine de forma diferente com o pixel existente no framebuffer.

O `DepthController` configura o teste de profundidade, o teste de visibilidade que é feito após o fragment shader. Com esse controller é possível, por exemplo desabilitar o teste de profundidade.

4.4.2 Desenho de modelos

Existem 2 modos de se enviar vértices para serem processados pelo pipeline, através de `VertexBuffer` ou através de chamadas de função obsoletas definidas pelo OpenGL, além disso, algumas vezes deseja-se que um mesmo modelo seja renderizado diversas vezes em

um laço (instanced draw). Para lidar com essas situações, a Pandora's Box utiliza a classe `DrawController`.

O `DrawController` é responsável por enviar corretamente ao pipeline modelos definidos por instâncias de `VertexBuffer` (na engine tais modelos são instâncias de `VBOModel`) assim como instâncias de modelos que utilizam as funções obsoletas. Além de gerenciar a renderização de um modelo, o `DrawController` é responsável por implementar o instanced draw.

Na Pandora's Box, instanced draw é implementado de forma nativa para `VBOModel`, ou seja, utilizando-se as funções específicas do OpenGL que implementam a técnica, e de forma simulada se o modelo não for instância de `VBOModel`. A versão simulada da técnica é conhecida como pseudo-instanciação. Em ambas as versões, uma variável que indica a instância do modelo atualmente sendo renderizada é disponibilizada. Essa técnica pode ser ilustrada pelo código abaixo:

```
GraphicAPI * gfx = ...;
Model * model = ...;

// prepara o modelo para a renderização
model->beforeRender( gfx );

for(int i = 0; i < number_of_instances; i++) {
    int instanceID = i;

    // renderiza a instância instanceID
    // a variável instanceID fica disponível no vertex shader
    model->render( gfx )
}

model->afterRender( gfx );
instanceID = 0;
```

4.4.3 Passagem de parâmetros para o GPUProgram

A customização do pipeline através de shaders gera grande flexibilidade, porém essa customização só é interessante devido a possibilidade de passagem de diferentes parâmetros para o shader.

Um programa do OpenGL pode receber três tipos de parâmetros:

- Uniformes: um valor que é constante para uma dada primitiva, por exemplo, para um triângulo, o processamento de seus três vértices utilizam o mesmo valor de uniforme.

- Uniformes built-in: são valores que se comportam como uniformes mas que são enviados automaticamente pelo OpenGL. As matrizes de transformação fazem parte dessa classe de valores.
- Atributos: um valor que é constante para um vértice. Atributos só podem ser acessados dentro do `vertex shader`. No exemplo acima, cada um dos vértices do triângulo poderia ter um valor diferente para o atributo.

Dentro da Pandora's Box, o mecanismo de passagem de parâmetros para o shader é implementado através das classes `UniformSet`, `UniformStack`, `GPUProgram`, `UniformValue` e `BuiltInUniformBinder` para uniformes e da classe `AttribBinder` para atributos.

Como foi citado anteriormente, a última etapa da atualização de estados é a sincronização dos parâmetros do shader. Nessa fase, inicialmente, cada um dos `UniformInfo` gerados durante a compilação do shader é utilizado para buscar um `UniformValue` dentro da `UniformStack`, o uniform value encontrado é então associado ao programa através do mecanismo ilustrado no código abaixo para o caso do `GPUProgram` implementado para OpenGL:

```
void GLProgram::updateUniforms(GraphicAPI * gfx) {
    std::vector<UniformBindAndInfo>::iterator it;
    for(it = uniforms.begin(); it != uniforms.end(); it++) {
        UniformValue * value = gfx->searchUniform(it->getInfo());
        if(it->shouldUpdate(value)) {
            it->update(value);
            // associa o valor da uniforme ao shader
            value->bindValueOn(this, it->getInfo(), gfx);
        }
    }
}
```

Após essa etapa de associação de valores, as uniformes built-in são enviadas ao shader. O envio de built-ins é feito através dos `BuiltInUniformBinder`, que são classes especializadas para cada um dos tipos de valor.

O mecanismo para o envio dos atributos é semelhante ao utilizada para enviar as uniformes built-in. Para cada um dos tipos de atributo definidos na enum `Type` dentro da classe `pbge::VertexAttrib` existe uma classe (`AttrBinder`) especializada em associar o atributo ao shader.

4.5 Visualização de campos tensoriais

A aplicação de visualização de campos tensoriais é dividida em duas etapas:

- Compilação do formato Analyze [4] para o formato `.ctf` (Compiled Tensor Field).

- Apresentação do campo contido no arquivo `.ctf`.

4.5.1 O formato Analyze

O formato Analyze é um formato de armazenamento de informações de imagens de ressonância magnética. A informação é dividida em dois arquivos: um cabeçalho (extensão `.hdr`) com informações sobre o campo (dimensões, ordenação, identificação e histórico) e o arquivo de imagem (extensão `.img`) contendo somente os valores da imagem (organizados conforme a descrição do cabeçalho).

Para o desenvolvimento do leitor de arquivos Analyze foi feita a suposição de que os nomes dos arquivos (`.hdr` e `.img`) são iguais visando simplificar a utilização e implementação.

4.5.2 O formato Compiled Tensor Field (`.ctf`)

O formato Compiled Tensor Field (`.ctf`) foi desenvolvido para armazenamento de informações sobre o campo tensorial a ser mostrado na aplicação de visualização de campos tensoriais. É um formato binário que contém o número de elipsóides (representação visual do tensor) no arquivo seguido por um conjunto de matrizes de transformação linear. Cada matriz será aplicada a uma esfera para obter um elipsóide na posição correta no campo. Por esse motivo ela contém uma escala (proporcional aos autovalores do tensor aplicados nos eixos cartesianos), uma rotação (dos eixos cartesianos para os eixos definidos pelos autovetores do tensor) e uma translação (para posicionar o elipsóide corretamente no campo).

Na etapa de compilação a imagem de ressonância magnética é lida e os tensores armazenados em um vetor. Nessa etapa todos os tensores nulos são ignorados. Um tensor $A_{3 \times 3}$ é considerado nulo quando o módulo de todas as suas entradas é maior do que um dado $\varepsilon > 0$. Ou seja, A é nulo quando a seguinte expressão é válida para $i, j \in \mathbb{Z}, 1 \leq i, j \leq 3$:

$$|A_{i,j}| < \varepsilon$$

São então calculados os autovalores e autovetores de todos os tensores não nulos, geradas as matrizes de transformação linear que levam uma esfera centralizada na origem para um elipsóide na posição correta no campo e armazenadas em um vetor.

Como tais matrizes são matrizes de transformação homogênea, a última linha sempre será $(0, 0, 0, 1)$. Assim é possível ocultar tais dados e enviar outras informações em seu lugar (é necessário substituir os valores dessa linha para realizar quaisquer operações com a matriz). Nessa linha são armazenados os valores das equações (1), (2), (3) e (4) (definidas na página 16) que serão utilizados como diferentes políticas de escolha de nível de transparência (alfa) e cor dos elipsóides.

As matrizes são então reorganizadas em blocos de proximidade para otimizar a utilização pela aplicação de visualização. O vetor de matrizes reordenado é finalmente escrito no arquivo, além das informações iniciais sobre o campo.

4.5.3 Apresentação do campo compilado

O arquivo `.ctf` é lido e cada bloco de matrizes é enviado como `uniform` para o `shader` que as aplica a esferas. A política de transparência (`alfa`) dos elipsóides é escolhida pelo usuário, sendo o valor de `alfa` algum dos quatro resultados das equações de anisotropia fracionada. A cor aplicada a cada elipsóide é calculada a partir do valor de `alfa` em uma rampa de cores também definida pelo usuário.

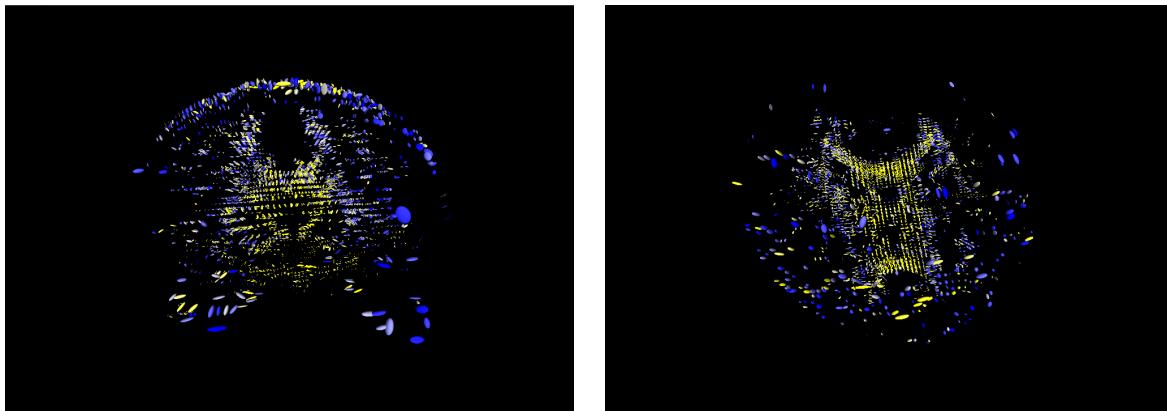


Figura 7: Imagens da visualização do campo tensorial de um cérebro humano. As cores são aplicadas a partir do valor de anisotropia fracionada em uma rampa de cores que varia do azul (menor anisotropia) para o amarelo (maior anisotropia).

Para a translação e rotação do campo foram implementados um `KeyboardEventHandler` e um `MouseEventHandler` respectivamente que atualizam os nós de transformação.

4.6 Técnicas aplicadas

4.6.1 Depth Peeling

No campo tensorial em diversos casos existem elipsóides sobrepostos. Para observar alguns tipos de estrutura é interessante que tais elipsóides possuam algum nível de transparência. Para isso é necessário simular a transparência através de combinações de cores sobrepostas. Existem técnicas [3] que dependem dos objetos serem renderizados dos mais distantes para os mais próximos da câmera, entretanto no caso do campo esse tipo de processo se torna

muito lento devido à grande quantidade de objetos a serem ordenados antes de cada renderização. Por esse motivo é necessário algum algoritmo independente da ordem de renderização.

Depth peeling [10] é uma técnica iterativa que consiste de remoções de camadas próximas a cada iteração. Inicialmente a cena é renderizada normalmente armazenando o color buffer e o buffer em buffers auxiliares. A cada nova iteração todos os fragmentos com profundidade menor ou igual à profundidade armazenada no depth buffer auxiliar são descartados. Um novo depth buffer auxiliar é gerado a partir das profundidades dos fragmentos restantes, que são então renderizados e o resultado (color buffer) é acumulado no color buffer auxiliar.

Para a realização da técnica completa são necessárias N iterações, onde N é o número máximo de fragmentos sobrepostos na cena. Entretanto na aplicação foi fixado um número de iterações para diminuir a complexidade da técnica.

Na aplicação de exemplo o depth peeling foi implementado como um processador de cena sendo que a cada iteração o grafo de cena é percorrido uma vez.

As imagens abaixo exemplificam essa técnica com duas esferas acompanhadas do resultado da iteração.

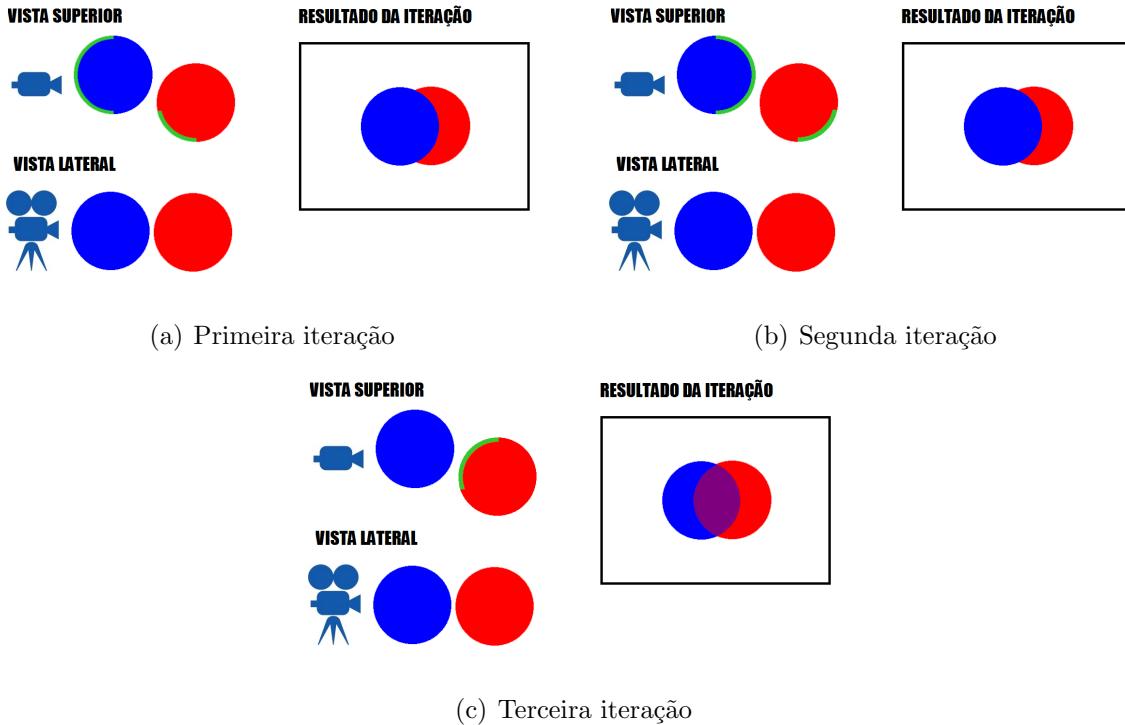


Figura 8: Em cada iteração do depth peeling é desconsiderada uma camada de fragmentos. A imagem é combinada ao resultado anterior considerando-se a opacidade de cada pixel.

5 Resultados e produtos obtidos

O código do Pandora's Box Graphics Engine, assim como da aplicação de visualização de campos tensoriais e outros exemplos pode ser encontrado no seguinte endereço:

<https://github.com/victorkendy/PandoraBox>

5.1 Compilador de campos tensoriais

A aplicação de compilação de campos tensoriais pode ser executada através da linha de comando com os seguintes argumentos opcionais:

```
field_compiler.exe [ARQUIVO DO CAMPO] [ARQUIVO CTF]
```

Caso o programa seja executado sem nenhum argumento o usuário deve escolher entre as opções 1 e 2 (campo de uma dupla hélice sintética e de um cérebro humano respectivamente). A opção ARQUIVO DO CAMPO deve indicar o nome de um arquivo .img sem a extensão (supõe-se que o arquivo .hdr tenha o mesmo nome), ou seja, supondo que existam os arquivos campo.hdr e campo.img, então o argumento deve ser campo. O último argumento deve ser o nome do arquivo .ctf a ser criado, por exemplo campo.ctf.

5.2 Visualizador de campos tensoriais

O visualizador de campos tensoriais também pode ser executado através da linha de comando da seguinte maneira:

```
tensor_field.exe [ARQUIVO CTF]
```

A execução sem nenhum argumento permitirá ao usuário escolher entre os dois campos utilizados como teste (dupla hélice e cérebro). É possível visualizar outros campos no formato .ctf enviando como argumento o nome do arquivo, por exemplo campo.ctf.

5.2.1 Movimentação e interação com o campo tensorial

É possível utilizar as seguintes teclas para interagir com o campo tensorial:

Alternância de efeitos	
1	Ativa/desativa o inversor de cores
2	Ativa/desativa o filtro do componente vermelho das cores
3	Ativa/desativa o efeito de lentes senoidais

4	Ativa/desativa o depth peeling
Movimentação da câmera	
W	Move a câmera para cima
S	Move a câmera para baixo
A	Move a câmera para a esquerda
D	Move a câmera para a direita
Q	Move a câmera em direção ao campo
E	Move a câmera na direção oposta ao campo
Nível de anisotropia considerado (só somente mostrados os elipsóides que representam tensores cuja anisotropia fracionada pertence ao intervalo [min_AF,max_AF])	
Z	Aumenta o valor de min_AF
X	Diminui o valor de min_AF
V	Aumenta o valor de max_AF
C	Diminui o valor de max_AF
Escala dos elipsóides	
O	Diminui o tamanho dos elipsóides
P	Aumenta o tamanho dos elipsóides
Cálculo da anisotropia fracionada	
R	Utiliza o cálculo da forma linear
T	Utiliza o cálculo da forma planar
Y	Utiliza o cálculo da forma esférica
U	Utiliza o cálculo geral da anisotropia fracionada
Grade (eixos cartesianos)	
G	Mostra/esconde a grade
Rampa de cores	
F	Alternar entre as rampas de cores disponíveis

Além da interação com o teclado é possível rotacionar o campo utilizando o mouse. O movimento se inicia com o clique do botão esquerdo do mouse e termina quando o botão é solto.

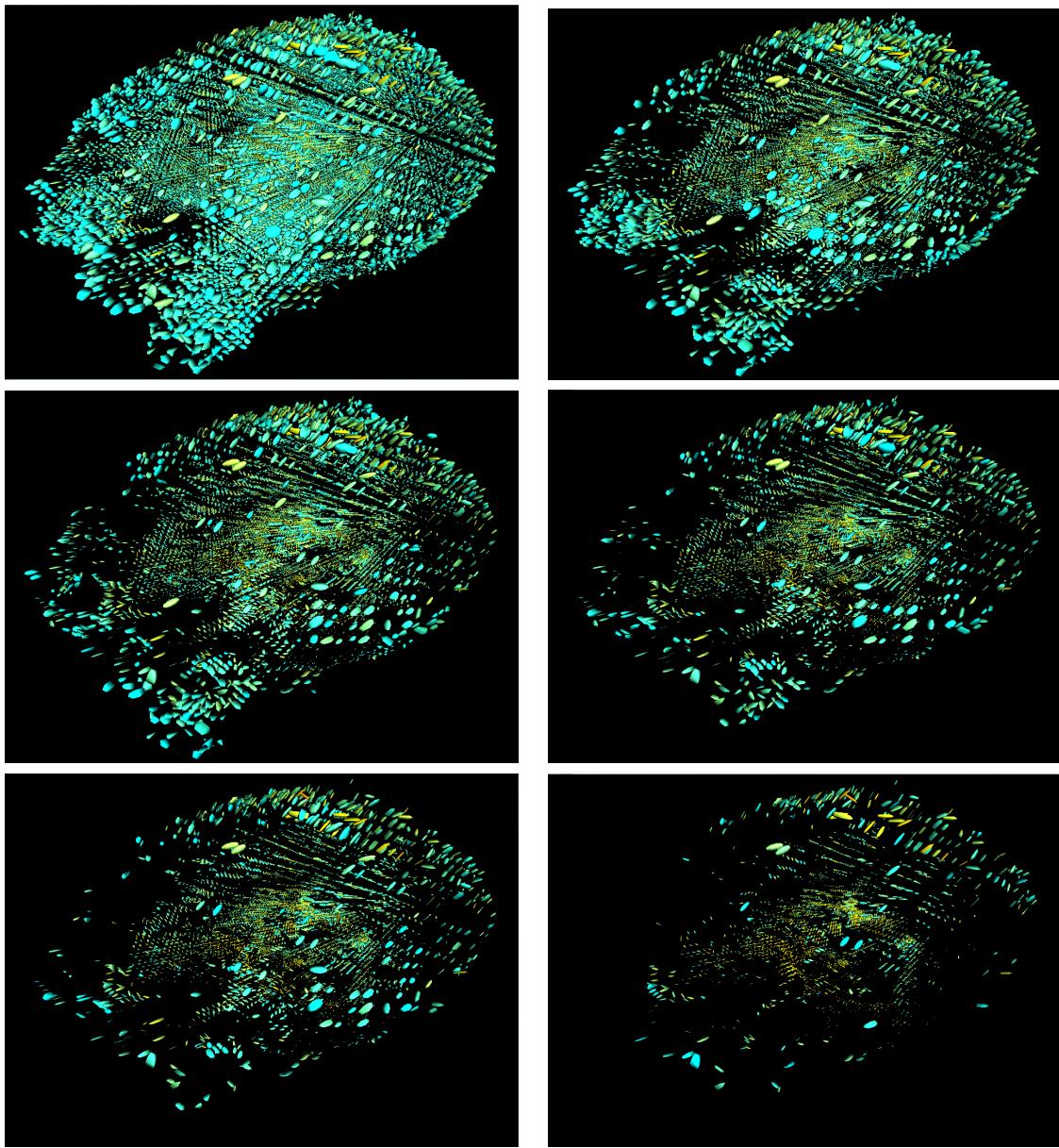


Figura 9: Imagens do campo tensorial de um cérebro humano com diferentes valores de min_AF (o valor aumenta da primeira para a última imagem).

5.3 Exemplo de código

A seguir será explicada a utilização básica da Pandora's Box. O exemplo desenvolvido utiliza um modelo de uma esfera que é transformada em um elipsóide no vertex shader. Inicialmente é incluído o header da engine:

```
#include "pbge/pbge.h"
```

Na função `main` são definidas as configurações da janela (título e dimensões), associado um inicializador de cena customizado ao gerenciador de janelas e então é enviado o sinal para a engine iniciar a renderização das imagens na janela. É necessário que o usuário implemente um `SceneInitializer` que definirá a estrutura do grafo de cena.

```
int main() {
    pbge::Manager manager;
    MySceneInitializer sceneInitializer;

    manager.setWindowTitle("Ellipsoid demo");
    manager.setWindowDimensions(1024, 768);
    manager.setSceneInitializer(&sceneInitializer);
    manager.displayGraphics();
    return 0;
}
```

O inicializador de cena deve herdar de `SceneInitializer` e implementar o operador `()`. É possível ter acesso ao renderizador através da instância de `Window` recebida como argumento. São então adicionados no renderizador os processadores de cena (`RenderPassProcessor` e `BlitToFramebuffer`). Em seguida é criado o grafo de cena com um nó de transformação identidade como raiz. O grafo é então preenchido com outros nós e devolvido.

```

class MySceneInitializer : public pbge::SceneInitializer {
public:
    pbge::SceneGraph * operator () (pbge::GraphicAPI * gfx,
                                    pbge::Window * window) {
        pbge::Renderer * renderer = window->getRenderer();

        renderer->addSceneProcessor (new pbge::RenderPassProcessor);
        renderer->addPostProcessor (new pbge::BlitToFramebuffer);

        pbge::Node * root = new pbge::TransformationNode;
        pbge::SceneGraph * graph = new pbge::SceneGraph (root);

        configureCamera (root, gfx);
        createModel (root, gfx);

        return graph;
    }

private:
    void configureCamera (pbge::Node *, pbge::GraphicAPI *);
    void createModel (pbge::Node *, pbge::GraphicAPI *);
};

```

Para definir a posição da câmera utiliza-se um nó de transformação linear que é adicionado como filho da raíz. A câmera é instanciada, configurada (direção e perspectiva) e adicionada aos filhos da transformação.

```

void configureCamera (pbge::Node * parent, pbge::GraphicAPI * gfx) {
    pbge::TransformationNode * cameraParent =
        pbge::TransformationNode::translation (0, 0, 10);

    pbge::CameraNode * camera = new pbge::CameraNode;
    camera->lookAt (math3d::vector4 (0, 1, 0),
                     math3d::vector4 (0, 0, -1));
    camera->setPerspective (90, 1.0f, 2.0f, 30.0f);

    cameraParent->addChild (camera);
    parent->addChild (cameraParent);
}

```

A esfera utilizada possui raio 2 e 100 subdivisões. O nó de modelo (**ModelInstance**) é criado a partir do modelo da esfera. O shader criado é adicionado ao nó, que é então adicionado ao grafo.

```

void createModel(pbge::Node * parent, pbge::GraphicAPI * gfx) {
    pbge::VBOModel * sphere =
        pbge::Geometrics::createSphere(2,100,gfx);
    pbge::ModelInstance * model = new pbge::ModelInstance(sphere);

    pbge::GPUProgram * shader =
        gfx->getFactory()->createProgramFromString(
            "#version 150\n"
            "in vec4 pbge_Vertex;\n"
            "out vec4 color;\n"
            "uniform mat4 pbge_ModelViewProjectionMatrix;\n"
            "void main() {\n"
            "    mat4 scale = mat4(1,0,0,0,\n"
            "                      0,2,0,0,\n"
            "                      0,0,1,0,\n"
            "                      0,0,0,1);\n"
            "    gl_Position = "
            "        pbge_ModelViewProjectionMatrix*scale*pbge_Vertex;\n"
            "    color = vec4(pbge_Vertex.xyz, 1);\n"
            "}"
            "in vec4 color;\n"
            "void main() {\n"
            "    gl_FragColor = color;\n"
            "}"
            );
    model->setRenderPassProgram(shader);
    parent->addChild(model);
}

```

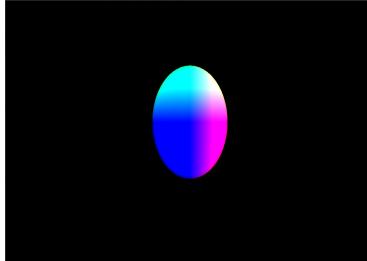


Figura 10: Após ser processado pelos shaders a esfera se transforma em um elipsóide e as cores são aplicadas nos fragmentos dependendo de sua posição, resultando na imagem acima.

O método `createProgramFromString` recebe duas strings, sendo a primeira o código do vertex shader e a segunda do fragment shader. Nesse exemplo o vertex shader definido recebe o vértice e a matriz `ModelViewProjection` e devolve a cor do fragmento. É criada uma matriz de escala que multiplica o vértice e o resultado é por sua vez multiplicado pela `ModelViewProjection` resultando na posição. A cor definida utiliza as coordenadas `x,y,z` como componentes `r,g,b`. O fragment shader define a cor do fragmento como a cor recebida do vertex shader.

Apesar de ser um exemplo simples não são necessárias muitas modificações para obter um resultado mais complexo. A Figura 11 foi gerada com o auxílio da engine com um código¹ semelhante ao apresentado acima. Foi aplicada a

¹https://github.com/victorkendy/PandoraBox/tree/master/grass_field

técnica de instanced draw sendo a figura mostrada no canto superior esquerdo a geometria utilizada.



Figura 11: Gramado gerado a partir da instanciação da geometria apresentada no canto superior esquerdo da imagem ao longo do campo em posições aleatórias.

6 Conclusões

Para o desenvolvimento de aplicações de computação gráfica são necessárias diversas técnicas de otimização e organização das informações. A Pandora's Box Graphics Engine tem por objetivo disponibilizar uma interface simples para utilização de algumas dessas técnicas:

- As cenas são representadas por grafos de cena nos quais é possível utilizar os nós já definidos na engine ou criar novos tipos para implementar outras técnicas.
- Uma cena pode ser renderizada através dos processadores de cena adicionados ao renderizador sendo possível também criar novos processadores (que percorram o grafo executando métodos específicos de nós customizados, por exemplo).
- A utilização de shaders se resume a escrever o código e associá-lo a algum nó do grafo (a compilação e o bind são gerenciados pela engine).
- Texturas são gerenciadas pela engine, cabendo ao usuário carregá-las e associá-las a nós do grafo de cena.

Com o auxílio da Pandora's Box foi possível desenvolver algumas aplicações de exemplo, incluindo o visualizador de campos tensoriais de imagens de ressonância magnética sensíveis à difusão. Essa aplicação exigiu a utilização de diversas técnicas de otimização para ser possível a visualização de campos com grandes quantidades de tensores em tempo real.

Nessa aplicação é possível restringir a informação apresentada retirando-se tensores dependendo do seu nível de anisotropia e observar estruturas internas. Ainda existem diversas melhorias a serem feitas tanto na engine quanto na aplicações de exemplo, entretanto já é possível utilizá-las da maneira como foram propostas. Com as aplicações desenvolvidas a Pandora's Box Graphics Engine se mostrou capaz de lidar com grandes quantidades de informação, além de ser facilmente customizável para diferentes finalidades.

Para o prosseguimento do trabalho é necessário o desenvolvimento de mais casos de uso aplicando diferentes técnicas de computação gráfica, facilitando a detecção de bugs, gargalos de processamento, assim como a necessidade de novas funcionalidades. É também desejável dar suporte a outros sistemas operacionais além do Windows. Além disso a utilização de uma thread separada para o renderizador, assim como outras otimizações também está prevista para o futuro.

Referências

- [1] 3D-Studio File Format. <http://www.martinreddy.net/gfx/3d/3DS.spec>. Acessado em novembro de 2011.
- [2] STL Algorithms. <http://www.cplusplus.com/reference/algorithm/>. Acessado em setembro de 2011.
- [3] Transparency Sorting. http://www.opengl.org/wiki/Transparency_Sorting. Acessado em novembro de 2011.
- [4] Mayo/SPM "Analyze" Format Spec Compilation. <http://www.grahamwideman.com/gw/brain/analyze/formatdoc.htm>. Acessado em setembro de 2011.
- [5] Blender File Format. <http://www.blender.org/development/architecture/blender-file-format/>. Acessado em novembro de 2011.
- [6] Boost C++ Libraries. <http://www.boost.org/>. Acessado em setembro de 2011.
- [7] C. CALLIOLI, R.C.F. COSTA, and H.H. DOMINGUES. *ALGEBRA LINEAR E APLICAÇÕES*. ATUAL EDITORA, 2007.
- [8] cplusplus.com - the C++ resources network. <http://wwwcplusplus.com/>. Acessado em agosto de 2011.
- [9] Chris Engelsma and Dave Hale. CWP-655 Visualization of 3D tensor fields derived from seismic images. <http://www.cwp.mines.edu/Meetings/Project10/cwp-655.pdf>. Acessado em junho de 2011.
- [10] Cass Everitt. Interactive Order-Independent Transparency, 2001.
- [11] view.frustum.culling.gif. http://techpubs.sgi.com/library/dynaweb_docs/0650/SGI_Developer/books/Optimizer_PG/sgi_html/figures/view.frustum.culling.gif. Acessado em novembro de 2011.
- [12] fountain.png. http://www.mpa-garching.mpg.de/galform/data_vis/. Acessado em novembro de 2011.
- [13] The OpenGL Extension Wrangler Library. <http://glew.sourceforge.net/>. Acessado em novembro de 2011.
- [14] Patric Hagmann, Lisa Jonasson, Philippe Maeder, Jean-Philippe Thiran, Van J. Wedeen, and Reto Meuli. Understanding Diffusion MR Imaging Techniques: From Scalar Diffusion-weighted Imaging to Diffusion Tensor Imaging and Beyond1. *Radiographics*, 26(suppl 1):S205–S223, October 2006.

- [15] Martin Head-Gordon. Tensor concepts in electronic structure theory: Application to self-consistent field methods and electron correlation techniques. *Modern Methods and Algorithms of Quantum Chemistry*, J. Grotendorst (Ed.), John von Neumann Institute for Computing, Julich, NIC Series, Vol. 1, ISBN 3-00-005618-1, 1:561–562, 2000.
- [16] Peter B. Kingsley. Introduction to diffusion tensor imaging mathematics: Part I. Tensors, rotations, and eigenvectors. *Concepts in Magnetic Resonance Part A*, 28A(2):101–122, March 2006.
- [17] POV-Ray Hall of Fame: "The Lizard". http://hof.povray.org/lizard_Big.html. Acessado em novembro de 2011.
- [18] Scott Meyers. Effective C++.
- [19] Biblioteca MSDN. <http://msdn.microsoft.com/library/default.aspx>. Acessado em novembro de 2011.
- [20] Wavefront OBJ File Format Summary. <http://www.fileformat.info/format/wavefrontobj/egff.htm>. Acessado em novembro de 2011.
- [21] Ogre3D. <http://www.ogre3d.org>. Acessado em maio de 2011.
- [22] Opengl, D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.
- [23] OpenGL 2.1 Reference Pages. <http://www.opengl.org/sdk/docs/man>. Acessado em junho de 2011.
- [24] OpenGL 3.3 Reference Pages. <http://www.opengl.org/sdk/docs/man3>. Acessado em junho de 2011.
- [25] OpenGL 4.1 Reference Pages. <http://www.opengl.org/sdk/docs/man4>. Acessado em junho de 2011.
- [26] OpenGL Registry. <http://www.opengl.org/registry/>. Acessado em novembro de 2011.
- [27] OpenSceneGraph. <http://www.openscenegraph.org>. Acessado em maio de 2011.
- [28] Ray Tracing. <http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm>. Acessado em novembro de 2011.

- [29] Standard Template Library Programmer's Guide. <http://www.sgi.com/tech/stl/>. Acessado em setembro de 2011.
- [30] C.-F. Westin, S. Peled, H. Gudbjartsson, R. Kikinis, and F. A. Jolesz. Geometrical diffusion measures for MRI from tensor basis analysis. In *ISMRM '97*, page 1742, Vancouver Canada, April 1997.

A Apêndice

A.1 A Windows API

Para se inicializar um contexto para a execução de programas em OpenGL é necessário criar uma janela, para isso utiliza-se a seguinte função:

```
HWND WINAPI CreateWindow(
    __in_opt LPCTSTR lpClassName,
    __in_opt LPCTSTR lpWindowName,
    __in     DWORD dwStyle,
    __in     int   x,
    __in     int   y,
    __in     int   nWidth,
    __in     int   nHeight,
    __in_opt HWND hWndParent,
    __in_opt HMENU hMenu,
    __in_opt HINSTANCE hInstance,
    __in_opt LPVOID lpParam
);
```

Onde:

- **lpClassName**: representa uma classe registrada anteriormente no sistema.
- **lpWindowName**: representa o título da janela.
- **dwStyle**: conjunto de flags que definem o estilo da janela. exibida: com bordas, maximizada, minimizada, etc.
- **x, y**: posição inicial da janela.
- **nWidth, nHeight**: dimensões iniciais da janela.
- **hWndParent**: se a janela que está sendo criada é uma subjanela de uma já existente, passa-se o handle (tipo especial de ponteiro inteligente) da janela pai nesse argumento, caso contrário, passa-se NULL.
- **hMenu**: se for necessário criar uma menu, deve-se passar o handle do menu nesse argumento, caso contrário, passa-se NULL.
- **hInstance**: o handle da instância do programa que está sendo executado, esse parâmetro é utilizado dentro do sistema para a criação de um identificador único para a janela.

- **lpParam**: dados do usuário. Esse parâmetro recebe um ponteiro para qualquer tipo de dado que o usuário necessite.

Para se registrar uma classe no sistema deve-se chamar a função:

```
ATOM WINAPI RegisterClass(
    __in const WNDCLASS *lpWndClass
);
```

O argumento **lpWndClass** é um ponteiro para uma instância de **WNDCLASS**, uma estrutura que guarda, entre outras coisas, o nome da classe que está sendo registrada, o ponteiro para a função que irá tratar os eventos gerados pelo sistema e uma flag que indica se o contexto do dispositivo pode ser compartilhado. Quando se cria um contexto do OpenGL é importante sempre registrar a classe como dona do contexto do dispositivo.

Após criada, deve-se avisar o sistema que a janela é visível, para isso, executa-se:

```
BOOL WINAPI ShowWindow(
    __in HWND hWnd,
    __in int nCmdShow
);
```

Onde **hWnd** é o handle devolvido pela função **CreateWindow** e **nCmdShow** controla como a janela deve ser exibida (maximizada, minimizada, simplesmente exibida, etc)

Após todo esse processo, pode-se consultar as mensagens recebidas pela janela através da função:

```
BOOL WINAPI GetMessage(
    __out LPMSG lpMsg,
    __in_opt HWND hWnd,
    __in     UINT wMsgFilterMin,
    __in     UINT wMsgFilterMax
);
```

Que recebe como parâmetros um ponteiro para uma estrutura do tipo **MSG**, o handle devolvido pela função **CreateWindow**, **wMsgFilterMin** e **wMsgFilterMax** são constantes que podem ser passadas para indicar que tipo de mensagem deve ser retornada. O valor de retorno dessa função é um inteiro que indica qual o tipo de mensagem foi retornada ou se ocorreu um erro durante a execução da função.

Após recuperar a mensagem ela precisa ser traduzida e em seguida enviada para o seu destino final. A função responsável pela tradução da mensagem é:

```
BOOL WINAPI TranslateMessage(
    __in const MSG *lpMsg
);
```

e a função que envia a mensagem para a janela é:

```
HRESULT WINAPI DispatchMessage(
    --in const MSG *lpmsg
);
```

Quando se recebe a mensagem WM_QUIT, o retorno de GetMessage é 0, nesse momento a tradução e envio de mensagens deve parar e o programa deve apagar o registro da classe criada por RegisterClass. O registro é apagado pela função:

```
BOOL WINAPI UnregisterClass(
    --in LPCTSTR lpClassName,
    --in_opt HINSTANCE hInstance
);
```

que recebe como argumentos o nome com o qual a classe foi registrada e a instância do programa que registrou a classe.

O laço de eventos da janela

A função registrada na WNDCLASS que tratará os eventos gerados pelo sistema deve ter a seguinte assinatura:

```
HRESULT CALLBACK WindowProc(
    --in HWND hwnd,
    --in UINT uMsg,
    --in WPARAM wParam,
    --in LPARAM lParam
);
```

Nessa função, o argumento hwnd é o handle da janela corrente, uMsg é um inteiro sem sinal que codifica a mensagem recebida e os argumentos wParam e lParam são os parâmetros da mensagem.

Um laço de eventos típico para um programa em OpenGL tem a seguinte forma:

```

LRESULT CALLBACK WindowProc(HWND hWnd,
                           UINT msg,
                           WPARAM wParam,
                           LPARAM lParam) {
    switch(msg) {
        case WM_CREATE:
            // Inicialização do contexto de renderização
            // no caso da WM_CREATE o valor de lParam é um
            // ponteiro para um struct do tipo CREATESTRUCT
            // que contém o ponteiro para os dados do usuário
            // passado na função CreateWindow

        case WM_DESTROY:
            // Liberação de recursos gráficos e deleção do
            // contexto criado
        case WM_CLOSE:
            // O usuário clicou no botão de fechamento da janela
            // Então deve-se avisar para o sistema operacional
            // que a aplicação deve ser encerrada.
            PostQuitMessage(0);
            break;
        case WM_PAINT:
            // O sistema requisitou a atualização da janela
            // à aplicação

            // Funções do wgl para limpar e renderizar
            wglSwapBuffers();
            // avisa ao sistema que o evento já foi
            // completamente tratado
            return 0;
    }
    // Executa a rotina padrão para tratar o evento gerado
    return (DefWindowProc(hwnd, msg, wParam, lParam));
}

```

Ao receber WM_CREATE a aplicação deve inicializar todos os recursos para que a janela possa ser utilizada. No caso da Pandora's Box utilizando OpenGL como API gráfica, deve-se inicializar o contexto de renderização que será utilizado.

A criação do contexto é gerenciada pela API do wgl, que é a implementação de OpenGL para o Windows. O contexto que será utilizado pela aplicação deve ser criado através da função `wglCreateContext`:

```

HGLRC WINAPI wglCreateContext(
    HDC hdc
);

```

Essa função tem como argumento o handle para o dispositivo utilizado pela janela. Tal handle pode ser obtido através da função GetDC:

```
HDC GetDC(  
    __in  HWND hWnd  
) ;
```

Logo a criação do contexto pode ser feita pelo seguinte trecho de código:

```
case WM_CREATE:  
    HDC hDC = GetDC(hWnd);  
    // indica quantos bits devem ser utilizados por pixel,  
    // bits do depth buffer, entre outras coisas  
    setPixelFormatDescriptor();  
    HGLRC hglrc = wglCreateContext(hDC);  
    wglMakeCurrent( hglrc );  
    break;
```

A.2 Uso dinâmico de DLLs

Para se executar código presente em uma DLL, é necessário carregá-la durante a execução do programa. Através da windows API, pode-se abrir um arquivo DLL com a função LoadLibrary:

```
HMODULE WINAPI LoadLibrary(  
    __in  LPCTSTR lpFileName  
) ;
```

essa função retorna o handle para a biblioteca carregada ou NULL caso tenha ocorrido um erro. Se o carregamento for bem sucedido, pode-se utilizar o handle obtido para procurar funções dentro da DLL.

Na windows API, a função que cuida da busca de ponteiros de função é a GetProcAddress

```
FARPROC WINAPI GetProcAddress(  
    __in  HMODULE hModule,  
    __in  LPCSTR lpProcName  
) ;
```

Essa função devolve um ponteiro genérico do tipo void* que precisa ser transformado para um ponteiro de função.

Um detalhe importante do carregamento dinâmico de bibliotecas escritas em C++, é que para permitir o uso de namespaces, classes e sobrecarga de funções, o compilador da linguagem muda os nomes das funções geradas. Para evitar a mudança de nomes, as

funções a serem exportadas devem ser declaradas da forma abaixo:

```
extern "C" {
    // declaração das funções a serem exportadas.
}
```

Além disso, os compiladores podem exigir algum tipo de anotação nas funções que devem ser exportadas. No caso do compilador da Microsoft®, a declaração do protótipo da função deve ser precedida por `_declspec(dllexport)`, como no código abaixo:

```
_declspec(dllexport) TipoDeRetorno nome_da_funcao(parametros);
```